aws

デベロッパーガイド

Amazon DynamoDB



API バージョン 2012-08-10

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon DynamoDB: デベロッパーガイド

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスは、Amazon のものではない製品またはサービスと関連付けてはならず、また、お客様に混乱を招くような形や Amazon の信用を傷つけたり失わせたりする形で使用することはできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

Amazon DynamoDB とは ... 1
特性 ... 2
サーバーレス ... 2
NoSQL ... 2
フルマネージド型 ... 2
あらゆるスケールで 1 桁ミリ秒のパフォーマンス ... 3
ユースケース ... 3
機能 ... 4
グローバルテーブルによるマルチアクティブレプリケーション ... 4
ACID トランザクション ... 4
変更データキャプチャ ... 5
セカンダリインデックス ... 5
サービス統合 ... 5
サーバーレス統合 ... 5
Amazon S3 との間のデータのインポートとエクスポート ... 6
ゼロ ETL 統合 ... 6
キャッシュ ... 6
セキュリティ ... 6
レジリエンス ... 7
グローバルテーブル ... 7
継続的バックアップとポイントインタイムリカバリ ... 8
オンデマンドバックアップと復元 ... 8
DynamoDB にアクセスする ... 8
料金 ... 8
使用開始 ... 9
DynamoDB の使用開始 ... 10
初回ユーザー向けのリソース ... 11
Amazon DynamoDB の初回ユーザー向けの追加のベストプラクティス ... 12
AWS CLI リソース ... 12
プログラミングリソース ... 12
DynamoDB にアクセスする ... 13
コンソールを使用する ... 13
の使用AWS CLI ... 14
API の使用 ... 17

NoSQL Workbench の使用 17
IP アドレスの範囲 18
インターネットプロトコルバージョン 6 (IPv6) のデュアルスタックエンドポイント 18
前提条件 19
DynamoDB のセットアップ 20
DynamoDB (ウェブサービス) の設定 20
DynamoDB local (ダウンロード可能バージョン) のセットアップ 21
ステップ 1: テーブルを作成します 48
ステップ 2: データを書き込みます 102
ステップ 3: データを読み込みます 134
ステップ 4: データを更新します 162
ステップ 5: データをクエリします 195
ステップ 6: (オプション) クリーンアップする 234
次のステップ 253
Console-to-Code 253
仕組み 254
DynamoDB で Console-to-Code を使用する利点 254
ユースケースの例 255
開始方法 255
仕組み 256
チートシート 256
初期セットアップ 256
SDK または CLI 257
基本アクション 257
名前付けルール 258
サービスクォータの基本 259
詳細情報 261
コアコンポーネント 261
テーブル、項目、属性 262
プライマリキー 265
セカンダリインデックス 267
DynamoDB Streams 270
DynamoDB API 271
コントロールプレーン 272
データプレーン 272
DynamoDB Streams 274

取引 ... 275
サポートされるデータ型と命名規則 ... 275
名前付けルール ... 276
データ型 ... 277
データ型記述子 ... 282
DynamoDB テーブルクラス ... 282
DynamoDB におけるパーティションとデータ分散 ... 283
データ分散: パーティションキー ... 283
データ分散: パーティションキーおよびソートキー ... 285
SQL から NoSQL に移行する方法について学ぶ ... 287
リレーショナルまたは NoSQL? ... 288
アクセスと認証 ... 290
テーブルの作成 ... 294
テーブルに関する情報の取得 ... 296
テーブルへのデータの書き込み ... 298
テーブルからデータの読み込み ... 302
インデックスの管理 ... 312
テーブルのデータ変更 ... 318
テーブルからデータを削除する ... 321
テーブルの削除 ... 323
Amazon DynamoDB の学習リソースとツール ... 324
コーディングと可視化のツール ... 324
規範的ガイダンス ... 325
ナレッジセンター ... 326
ブログ投稿、リポジトリ、ガイド ... 327
データモデリングと設計パターン ... 328
トレーニングコース ... 328
読み込みと書き込み ... 330
DynamoDB の読み取り整合性 ... 330
結果整合性のある読み込み ... 330
強力な整合性のある読み込み ... 331
グローバルテーブルの読み込み整合性 ... 331
読み込みと書き込みのオペレーション ... 331
読み込みオペレーションの消費量 ... 332
書き込みオペレーションの消費量 ... 333
DynamoDB のスループットキャパシティ ... 336

オンデマンドモード ... 336
プロビジョニングモード ... 337
DynamoDB オンデマンドキャパシティモード ... 337
読み取りリクエスト単位と書き込みリクエスト単位 ... 338
初期スループットとスケーリングのプロパティ ... 339
オンデマンドテーブルの DynamoDB 最大スループット ... 340
プロビジョンドキャパシティモード ... 342
読み込みおよび書き込みキャパシティユニット ... 343
初期スループット設定の選択 ... 344
DynamoDB 自動スケーリング ... 345
Auto Scaling によるスループットキャパシティの管理 ... 346
リザーブドキャパシティ ... 377
ウォームスループット ... 378
テーブルのウォームスループットを確認する ... 379
テーブルのウォームスループットを増やす ... 382
ウォームスループットが高いテーブルを作成する ... 390
ウォームスループットのシナリオ ... 404
バーストキャパシティとアダプティブキャパシティ ... 406
バーストキャパシティ ... 406
アダプティブキャパシティ ... 406
キャパシティモードの切り替え ... 408
プロビジョンドモードからオンデマンドモードへ ... 408
オンデマンドモードからプロビジョンドモードへ ... 410
DynamoDB を使用したプログラミング ... 412
DynamoDB に対する AWS SDK サポートの概要 ... 412
AWS アカウントベースのエンドポイントの SDK サポート ... 414
DynamoDB で動作するプログラムインターフェイス ... 415
上位レベルのプログラミングインターフェイス ... 422
コード例の実行 ... 486
低レベル API ... 493
Python によるプログラミング ... 500
Boto について ... 500
Boto ドキュメント ... 501
クライアント層とリソース層 ... 502
batch_writer の使用 ... 505
その他のコード例 ... 506

セッションとスレッドセーフティ 506
構成 507
エラー処理 512
ログ記録 514
イベントフック 515
ページネーションとページネーター 517
ウェーター 519
JavaScript を使用したプログラミング 519
AWS SDK for JavaScript について 520
AWS SDK for JavaScript V3 520
JavaScript のドキュメント 520
抽象化レイヤー 521
marshall ユーティリティ関数 523
項目の読み込み 524
条件付きの書き込み 526
ページ分割 526
構成 529
ウェーター 532
エラー処理 532
ログ記録 534
考慮事項 535
AWS SDK for Java 2.x を使用したプログラミング 536
AWS SDK for Java 2.x について 537
ご利用開始にあたって 538
SDK for Java 2.x ドキュメント 547
サポートされているインターフェイス 548
その他のコード例 563
同期プログラミングと非同期プログラミング 563
HTTP クライアント 564
構成 565
エラー処理 573
AWS リクエスト ID 574
ログ記録 574
Pagination (ページ分割) 577
データクラス注釈 578
ベストプラクティス 578

エラー処理 ... 579
エラーコンポーネント ... 579
トランザクションエラー ... 580
エラーメッセージおよびコード ... 580
アプリケーションのエラー処理 ... 586
エラーの再試行とエクスポネンシャルバックオフ ... 586
バッチオペレーションとエラー処理 ... 587
AWS SDK の操作 ... 588
DynamoDB の操作 ... 590
テーブルの操作 ... 590
テーブルの基本オペレーション ... 591
DynamoDB でテーブルクラスを選択する場合の考慮事項 ... 600
タグとラベル ... 601
グローバルテーブルの操作 ... 607
整合性モード ... 607
アカウント設定 ... 607
重要な概念 ... 609
同一アカウントグローバルテーブル ... 614
マルチアカウントグローバルテーブル ... 676
グローバルテーブルの請求 ... 700
グローバルテーブルのバージョン ... 703
グローバルテーブルのベストプラクティス ... 714
項目の操作 ... 716
項目サイズと形式 ... 718
項目の読み込み ... 719
項目を書き込みます ... 720
戻り値 ... 722
バッチオペレーション ... 724
アトミックカウンタ ... 726
条件付きの書き込み ... 727
式の使用 ... 733
有効期限 (TTL) ... 775
テーブルに対するクエリの実行 ... 805
テーブルのスキャン ... 814
PartiQL クエリ言語 ... 823
項目の操作: Java ... 873

項目の操作: .NET ... 905
セカンダリインデックスの操作 ... 938
グローバルセカンダリインデックス ... 943
ローカルセカンダリインデックス ... 1040
ベクトルインデックスの使用 ... 1093
ベクトルインデックスのユースケース ... 1094
ベクトルインデックスとセカンダリインデックスの比較 ... 1095
距離関数 ... 1096
距離関数による結果のランク付け ... 1098
SearchSchema ... 1099
計画 ... 1100
ベクトルインデックスを他の DynamoDB 機能と併用する ... 1101
ベクトルインデックスの作成と検索 ... 1103
要件と制限 ... 1114
セキュリティとアクセスコントロール ... 1115
チュートリアル: 初めてのベクトル検索 ... 1116
データ同期 ... 1130
ベストプラクティス ... 1131
ストレージに関する考慮事項 ... 1134
キャパシティのモニタリング ... 1134
トラブルシューティング ... 1137
グローバルテーブル ... 1141
トランザクションでの使用 ... 1143
仕組み ... 1144
トランザクションでの IAM の使用 ... 1153
サンプルのコード ... 1156
ストリームの使用 ... 1160
Options ... 1161
Kinesis Data Streams の操作 ... 1163
DynamoDB Streams の操作 ... 1182
DAX とインメモリアクセラレーション ... 1275
DAX のユースケース ... 1276
DAX の使用に関する注意事項 ... 1277
仕組み ... 1278
DAX でのリクエスト処理方法 ... 1280
項目キャッシュ ... 1282

クエリキャッシュ 1283
DAX クラスターコンポーネント 1283
ノード 1284
クラスター 1284
リージョンとアベイラビリティーゾーン 1286
パラメータグループ 1287
セキュリティグループ 1287
クラスター ARN 1287
クラスターエンドポイント 1287
ノードエンドポイント 1288
サブネットグループ 1288
Events 1288
メンテナンスウィンドウ 1289
DAX クラスターの作成 1290
DynamoDB にアクセスする DAX 用の IAM サービスロールを作成します 1291
DAX と IPv6 1293
AWS CLI の使用 1295
コンソールを使用する 1301
整合性モデル 1306
DAX クラスターノード間の整合性 1307
DAX 項目キャッシュの動作 1307
DAX クエリキャッシュの動作 1310
強力な整合性のあるトランザクション読み込み 1311
ネガティブキャッシング 1312
書き込みの方法 1312
DAX クライアントで開発する 1316
チュートリアル: サンプルアプリケーションを実行する 1316
既存のアプリケーションを DAX を使用するように変更する 1380
DAX クラスターの管理 1381
DAX クラスターを管理するための IAM アクセス許可 1381
DAX クラスターのスケーリング 1384
DAX クラスターの設定のカスタマイズ 1386
TTL 設定の構成 1387
DAX のタグ付けサポート 1388
AWS CloudTrail の統合 1390
クラスターを削除する 1390

DynamoDB Accelerator のモニタリング 1390
DAX モニタリングツール 1391
CloudWatch によるモニタリング 1392
AWS CloudTrail を使用した DAX オペレーションのログ記録 1418
DAX T3/T2 バーストインスタンス 1419
DAX T2 インスタンスファミリー 1419
DAX T3 インスタンスファミリー 1419
DAX のアクセスコントロール 1420
DAX 用の IAM サービスロール 1422
DAX クラスターのアクセスを許可する IAM ポリシー 1424
導入事例: DynamoDB と DAX にアクセスする 1425
DynamoDB へのアクセスと DAX へのアクセス防止 1426
DynamoDB および DAX へのアクセス 1429
DAX 経由での DynamoDB へのアクセスの許可と DynamoDB への直接アクセスの防止 ... 1435
保管時の DAX 暗号化 1437
AWS マネジメントコンソール を使用した保管時の暗号化の有効化 1440
転送時の DAX 暗号化 1440
DAX のサービスにリンクされたロールの使用 1441
DAX のサービスにリンクされたロールにおけるアクセス許可 1442
DAX でサービスにリンクされたロールの作成 1443
DAX のサービスにリンクされたロールの編集 1444
DAX でサービスにリンクされたロールの削除 1444
AWS アカウント間での DAX へのアクセス 1446
IAM のセットアップ 1446
VPC をセットアップする 1449
クロスアカウントアクセスを許可するように DAX クライアントを変更します 1451
DAX クラスターサイジングガイド 1456
概要 1456
トラフィックの見積もり 1457
負荷テスト 1458
データモデリング 1460
項目コレクションの操作 1461
項目コレクションでデータを整理してクエリを高速化する 1462
データモデリングの基盤 1463
シングルテーブル設計 1463
マルチテーブル設計 1466

データモデリングの構成要素 1467
複合ソートキー 1468
マルチテナンシー 1469
スパースインデックス 1470
Time to live 1472
Time to Live (アーカイブ用) 1474
垂直パーティショニング 1474
書き込みシャーディング 1477
データモデリングスキーマ設計パッケージ 1478
前提条件 1479
ソーシャルネットワーク 1479
ゲームプロファイル 1489
苦情管理システム 1498
定期払い 1516
デバイスステータスの更新 1521
オンラインショップ 1534
リレーショナルモデル化 1558
従来のリレーショナルデータベースモデル 1559
DynamoDB によって JOIN オペレーションが不要になる理由 1561
DynamoDB トランザクションが書き込みプロセスのオーバーヘッドを排除する方法 1562
最初のステップ 1563
例 1565
DynamoDB への移行 1578
移行すべき理由 1578
移行する際の考慮事項 1579
仕組み 1581
移行ツール 1582
移行戦略の選択 1583
オフライン移行 1585
ハイブリッド移行 1587
オンライン - 各テーブルを 1 対 1 で移行 1589
オンライン - カスタムステージングテーブルによる移行 1590
NoSQL Workbench 1593
ダウンロード 1594
データモデラー 1595
新しいモデルの作成 1596

既存のモデルのインポート 1598
既存モデルの編集 1598
サンプルデータの追加 1599
アクセスパターンの追加と検証 1599
CSV からのインポート 1601
ファセット 1601
集約ビュー 1602
モデルのエクスポート 1603
データモデルのコミット 1603
オペレーションビルダー 1604
データセットへの接続 1605
オペレーションの構築 1606
テーブルのクローン作成 1618
CSV へエクスポート 1620
サンプルデータモデル 1620
従業員データモデル 1621
ディスカッションフォーラムデータモデル 1621
ミュージックライブラリデータモデル 1622
スキーリゾートデータモデル 1622
クレジットカードオファーデータモデル 1623
ブックマークデータモデル 1623
リリース履歴 1624
バックアップと復元 1632
ポイントインタイムバックアップ 1634
[開始する前に] 1634
ポイントインタイムリカバリを有効にする 1635
オンデマンドバックアップ 1640
仕組み 1640
テーブルのバックアップ 1643
テーブルの復元 1646
テーブルバックアップの削除 1652
IAM の使用 1653
バックアップの請求 1661
仕組み 1661
DynamoDB バックアップの請求例 1662
復元 1666

ポイントインタイムリカバリを使用したテーブルの復元 1666
DynamoDB テーブルを特定の時点に復元 1668
AWS Backup の使用 1674
仕組み 1676
バックアップの作成 1679
バックアップのコピー 1680
テーブルの復元 1682
バックアップの削除 1683
AWS Backup とDynamoDB によるオンデマンドバックアップの管理の比較 1683
Amazon DynamoDB ワークロードのディザスタリカバリ戦略の選択 1684
DynamoDB DR アプローチを選択するための決定ポイント 1685
ディザスタリカバリオプション 1686
コードの例 1694
基本 1697
Hello DynamoDB 1698
基本を学ぶ 1708
アクション 1866
シナリオ 2278
DAX で読み取りを高速化 2280
グローバルセカンダリインデックスの高度なシナリオ 2289
DynamoDB テーブルにデータを送信するアプリケーションを構築する 2292
複数の値を 1 つの属性と比較する 2293
項目の TTL を条件付きで更新する 2316
ローカルインスタンスに接続する 2323
カウント式演算子 2324
COVID-19 データを追跡する REST API を作成する 2357
メッセンジャーアプリケーションを作成する 2358
サーバーレスアプリケーションを作成して写真の管理 2359
グローバルセカンダリインデックスを含むテーブルを作成する 2363
ウォームスループットを有効にしたテーブルを作成する 2367
DynamoDB データを追跡するウェブアプリケーションを作成する 2377
WebSocket チャットアプリケーションを作成する 2379
TTL を含む項目を作成する 2380
MRSC グローバルテーブルの作成と管理 2386
MREC を示すグローバルテーブルの作成と管理 2406
PartiQL DELETE を使用してデータを削除する 2416

イメージ内の PPE を検出する 2422
NoSQL データベースの開始方法 2423
PartiQL INSERT を使用してデータを挿入する 2430
ブラウザからの Lambda 関数の呼び出し 2435
グローバルセカンダリインデックスを管理する 2435
リソースベースのポリシーを管理する 2438
DynamoDB のパフォーマンスのモニタリング 2441
高度なクエリオペレーションを実行する 2441
リストオペレーションを実行する 2476
マップオペレーションを実行する 2509
セットオペレーションを実行する 2534
PartiQL ステートメントのバッチを使用してテーブルにクエリを実行する 2564
PartiQL を使用してテーブルに対してクエリを実行する 2627
グローバルセカンダリインデックスを使用してテーブルをクエリする 2681
begins_with 条件を使用してテーブルをクエリする 2690
日付範囲を使用してテーブルをクエリする 2696
複雑なフィルター式を使用してテーブルをクエリする 2706
動的フィルター式を使用してテーブルをクエリする 2714
フィルター式と制限を使用してテーブルをクエリする 2722
ネストされた属性を使用してテーブルをクエリする 2727
ページ分割を使用してテーブルをクエリする 2737
強力な整合性のある読み込みを使用してテーブルをクエリする 2747
PartiQL SELECT を使用してデータをクエリする 2752
TTL 項目のクエリ 2758
日付と時刻のパターンを使用してテーブルをクエリする 2762
EXIF およびその他のイメージ情報を保存します 2776
属性ベースのアクセス制御を設定する 2777
式順序の更新を理解する 2780
テーブルのウォームスループット設定を更新する 2812
項目を TTL を更新する 2817
PartiQL UPDATE を使用してデータを更新する 2821
API Gateway を使用して Lambda 関数を呼び出す 2828
Step Functions を使用して Lambda 関数を呼び出す 2830
ドキュメントモデルを使用する 2831
高レベルのオブジェクト永続性モデルを使用する 2847
アトミックカウンタオペレーションを使用する 2856

条件付きオペレーションを使用する 2879
式の属性名を使用する 2909
スケジュールされたイベントを使用した Lambda 関数の呼び出し 2937
ローカルセカンダリインデックスを操作する 2939
Streams と Time-to-Live を操作する 2942
グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する 2945
リソースのタグ付けを操作する 2949
テーブルの暗号化を操作する 2952
サーバーレスサンプル 2954
DynamoDB トリガーから Lambda 関数を呼び出す 2954
DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする 2963
AWS コミュニティへの貢献 2974
サーバーレスアプリケーションの構築とテスト 2975
セキュリティ 2977
AWS マネージドポリシー 2978
AWS マネージドポリシー 2978
AWS マネージドポリシー: AmazonDynamoDBFullAccess_v2 2979
AmazonDynamoDBReadOnlyAccess 2979
DynamoDB での AWS マネージドポリシーの更新 2981
リソースベースのポリシー 2982
テーブルの作成 2984
リソースベースのポリシーのアタッチ 2990
ストリームへのポリシーのアタッチ 2994
リソースベースのポリシーの削除 2997
クロスアカウントアクセス 2998
パブリックアクセスのブロック 3001
API オペレーション 3004
IAM 認可 3012
例 3013
考慮事項 3021
ベストプラクティス 3022
属性ベースのアクセス制御 3024
ABAC を使用する理由 3025
条件キー 3026
考慮事項 3026

テーブルとインデックスでの ABAC 3027
DynamoDB Streams での ABAC 3047
データ保護 3063
保管中の暗号化 3064
DynamoDB 接続の保護 3091
IAM 3093
Identity and Access Management 3094
条件の使用 3126
コンプライアンス検証 3154
耐障害性 3155
インフラストラクチャセキュリティ 3156
VPC エンドポイントの使用 3157
AWS PrivateLink for DynamoDB 3167
Amazon VPC エンドポイントのタイプ 3168
AWS PrivateLink for Amazon DynamoDB を使用する場合の考慮事項 3169
Amazon VPC エンドポイントの作成 3170
Amazon DynamoDB インターフェイスエンドポイントへのアクセス 3170
DynamoDB インターフェイスエンドポイントから DynamoDB テーブルおよびコントロール API オペレーションへのアクセス 3170
オンプレミスの DNS 設定の更新 3172
Amazon VPC エンドポイントポリシーの作成 3175
AWS マネジメントコンソール プライベートアクセスでの DynamoDB エンドポイントの使用 3176
AWS PrivateLink for DynamoDB Streams 3176
AWS PrivateLinkDAX 用 3183
設定と脆弱性の分析 3186
セキュリティのベストプラクティス 3186
予防的セキュリティのベストプラクティス 3187
セキュリティ問題の検出ベストプラクティス 3189
モニタリングとログ記録 3193
モニタリング計画 3193
パフォーマンスのベースライン 3193
統合サービス 3194
自動モニタリングツール 3194
メトリクスのモニタリング 3195
DynamoDB メトリクスの使用方法 3195

CloudWatch コンソールでのメトリクスの表示 ... 3197
AWS CLI でのメトリクスの表示 ... 3197
メトリクスとディメンション ... 3198
DynamoDB に CloudWatch アラームを作成する ... 3231
オペレーションのログ記録 ... 3235
CloudTrail 内の DynamoDB 情報 ... 3235
DynamoDB ログファイルのエントリについて ... 3240
Contributor Insights ... 3259
仕組み ... 3259
開始方法 ... 3270
IAM の使用 ... 3280
ベストプラクティス ... 3287
NoSQL 設計 ... 3288
NoSQL と RDBMS の比較 ... 3288
2 つの重要な概念 ... 3288
一般的なアプローチ ... 3289
NoSQL Workbench ... 3290
DynamoDB の Well-Architected フレームレンズ ... 3291
コスト最適化 ... 3291
Amazon DynamoDB Well-Architected レンズによるレビューの実施 ... 3340
Amazon DynamoDB の Well-Architected レンズの柱 ... 3340
パーティションキーの設計 ... 3342
ワークロードを分散する ... 3343
書き込みシャーディング ... 3345
データを効率的にアップロードする ... 3346
ソートキーの設計 ... 3348
バージョンコントロール ... 3349
セカンダリインデックス ... 3350
一般的なガイドライン ... 3351
スパースインデックス ... 3354
集約 ... 3356
GSI の多重定義 ... 3363
GSI シャーディング ... 3364
レプリカの作成 ... 3369
大きな項目 ... 3370
圧縮 ... 3371

垂直パーティショニング ... 3371
Amazon S3 の使用 ... 3372
時系列データ ... 3372
時系列データの設計パターン ... 3372
Standard-IA テーブルクラスを使用してストレージコストを最適化する ... 3373
時系列テーブルの例 ... 3374
多対多の関係 ... 3374
隣接関係のリスト ... 3374
マテリアライズされたグラフ ... 3376
クエリとスキャン ... 3382
スキャンのパフォーマンス ... 3382
スパイクの回避 ... 3382
並列スキャン ... 3386
テーブル設計 ... 3387
グローバルテーブルを使用する ... 3387
重要な事実 ... 3388
MREC に関する重要な事実 ... 3389
MRSC に関する重要な事実 ... 3391
ユースケース ... 3393
マルチアカウントグローバルテーブルに関する重要な事実 ... 3394
書き込みモード ... 3395
DynamoDB のルーティング戦略 ... 3402
退避プロセス ... 3408
スループットキャパシティプランニング ... 3412
準備チェックリスト ... 3413
結論とリソース ... 3422
コントロールプレーン ... 3423
一括データオペレーション ... 3423
条件付きバッチ更新 ... 3424
効率的な一括オペレーション ... 3429
同時更新の処理 ... 3431
オプティミスティックロック ... 3432
トランザクションによる悲観的ロック ... 3434
分散ロック ... 3437
同時実行制御戦略の選択 ... 3440
請求および使用状況レポート ... 3441

スループット容量 3444
Streams 3449
ストレージ 3449
バックアップと復元 3450
データ転送 3454
CloudWatch 3455
DAX 3456
DynamoDB テーブルをアカウントから別のアカウントに移行する 3457
クロスアカウントバックアップと復元用の AWS Backup を使用してテーブルを移行する 3458
S3 へのエクスポートと S3 からのインポートを使用してテーブルを移行する 3460
DAX の規範ガイダンス 3463
DAX の適合性の評価 3464
DAX クライアントの設定 3466
DAX クラスターの設定 3468
クラスターのサイジング 3474
クラスターのデプロイ 3481
クラスターオペレーション 3483
DAX のモニタリング 3486
PITR 復旧 3488
テーブルの復元を開始して復旧する 3488
不要な書き込みをインプレースでロールバックして復旧する 3490
DynamoDB を他の AWS サービスで使用する 3492
Amazon Cognito との統合 3492
Amazon Redshift との統合 3495
CMK を使用したクロスアカウント統合に関する考慮事項 3495
Amazon Redshift とのゼロ ETL 統合 3499
COPY を使用した DynamoDB から Amazon Redshift へのデータのロード 3508
Amazon EMR との統合 3509
概要 3510
チュートリアル:Amazon DynamoDB と Apache Hive の使用 3511
Hive に外部テーブルを作成します 3520
HiveQL ステートメントの処理 3524
DynamoDB 内データのクエリ 3525
Amazon DynamoDB との間でデータをコピーします 3528
パフォーマンスチューニング 3542
S3 との統合 3548

Amazon S3 からのインポート ... 3548
Amazon S3 へのエクスポート ... 3570
Amazon SageMaker Lakehouse との統合 ... 3605
Amazon SageMaker Lakehouse とのゼロ ETL 統合 ... 3605
Amazon OpenSearch Service との統合 ... 3607
仕組み ... 3607
統合の作成 ... 3608
次のステップ ... 3608
重大な変更の処理 ... 3609
OpenSearch Service とのゼロ ETL 統合 ... 3613
Amazon EventBridge との統合 ... 3615
仕組み ... 3616
コンソールを使用した統合の作成 ... 3617
次のステップ ... 3620
Amazon MSK との統合 ... 3620
仕組み ... 3621
統合の例 ... 3621
次のステップ ... 3628
統合に関するベストプラクティス ... 3628
スナップショットの作成 ... 3628
変更データキャプチャ ... 3629
DynamoDB でのエージェンティック AI の使用 ... 3630
DynamoDB の生成 AI ユースケース ... 3630
RAG とエージェントメモリのベクトル検索 ... 3631
AI フレームワークコネクタ ... 3632
Strands Agents ストレージ ... 3633
LangChain 統合 ... 3641
LangGraph チェックポイントストア ... 3644
LangGraph 長期メモリ ... 3650
DynamoDB の生成 AI ブログ ... 3656
OpenSearch Service と DynamoDB のゼロ ETL 統合を活用する ... 3656
クォータと制約 ... 3658
クォータ管理タスクの実行 ... 3658
DynamoDB クォータへのアクセス ... 3658
コンソールで現在のクォータを表示するには ... 3659
AWS CLI を使用した現在のクォータの表示 ... 3659

クォータ引き上げのリクエスト 3661
クォータ 3662
読み取り/書き込みスループット 3663
リザーブドキャパシティ 377
テーブル 3667
グローバルテーブル 3667
セカンダリインデックス 3669
ベクトルインデックス 3669
射影されたセカンダリインデックスの属性 3670
DynamoDB Streams 3670
Amazon S3 からのインポート 3671
Amazon S3 へのテーブルのエクスポート 3671
バックアップと復元 3671
Contributor Insights 3671
制約 3672
読み取り/書き込みキャパシティモード 3672
セカンダリインデックス 3669
パーティションキーおよびソートキー 3674
名前付けルール 3674
データ型 3675
項目 3676
属性 3676
式パラメータ 3677
DynamoDB のトランザクション 3678
DynamoDB Streams 3679
DynamoDB Accelerator (DAX) 3679
API 固有の制約 3680
保管時の DynamoDB 暗号化 3683
API リファレンス 3684
トラブルシューティング 3685
内部サーバーエラー 3685
内部サーバーエラーの調査 3686
内部サーバーエラーの影響を最小限に抑える 3686
運用意識の向上 3687
レイテンシー 3688
スロットリング 3691

スロットリングの診断 3693
解決ガイド 3700
GSI 書き込みスロットリングとバックプレッシャー 3727
CloudWatch スロットリングメトリクス 3729
付録 3732
DynamoDB で SSL/TLS 接続の確立に関する問題をトラブルシューティングする 3732
アプリケーションまたはサービスをテストする 3732
クライアントブラウザのテスト 3733
ソフトウェアアプリケーションクライアントの更新中 3733
クライアントブラウザを更新する 3734
手動で証明書バンドルを更新する 3734
DynamoDB で使用するテーブルとデータの例 3735
サンプルデータファイル 3736
サンプルテーブルを作成してデータをアップロードする 3749
サンプルテーブルを作成してデータをアップロードする - Java 3749
サンプルテーブルを作成してデータをアップロードする - .NET 3759
AWS SDK for Python (Boto3) を使用したサンプルアプリケーション 3771
ステップ 1: ローカルにデプロイおよびテストを実行します 3772
ステップ 2: データモデルと実装の詳細を調べます 3777
ステップ 3: 本稼働環境でのデプロイ 3788
ステップ 4: リソースをクリーンアップする 3798
DynamoDB の予約語 3798
AWS SDK for Java 1.x の例 3811
DAX および Java SDK v1 3812
既存の SDK for Java 1.x アプリケーションを DAX を使用するように変更する 3824
SDK for Java 1.x を使用したグローバルセカンダリインデックスのクエリ 3829
AWS SDK for Go 1.x の例 3833
Go および DAX 3833
AWS SDK for Node.js 2.x の例 3836
Node.js および DAX 3836
ドキュメント履歴 3847
以前の更新 3886
レガシー機能 3922
グローバルテーブルバージョン 2017.11.29 (レガシー) 3922
仕組み 3923
ベストプラクティスと要件 3928

グローバルテーブルの作成 (バージョン 2017.11.29) 3932
グローバルテーブルのモニタリング 3937
グローバルテーブルで IAM を使用します 3939
以前の低レベル DynamoDB API バージョン (2011-12-05) 3942
BatchGetItem 3943
BatchWriteItem 3950
CreateTable 3958
DeleteItem 3965
DeleteTable 3972
DescribeTables 3976
GetItem 3980
ListTables 3984
PutItem 3987
Query 3994
Scan 4010
UpdateItem 4029
UpdateTable 4039
レガシー DynamoDB 条件パラメータ 4044
AttributesToGet 4046
AttributeUpdates 4047
ConditionalOperator 4050
Expected 4050
KeyConditions 4056
QueryFilter 4059
ScanFilter 4061
レガシーパラメータを使用した条件の記述 4063

Amazon DynamoDB とは

Amazon DynamoDB は、サーバーレスのフルマネージド分散 NoSQL データベースで、あらゆる規模で 1 桁ミリ秒のパフォーマンスを実現します。

DynamoDB を使用すると、リレーショナルデータベースのスケーリングと運用の複雑さの課題に対応できます。DynamoDB は、あらゆる規模で一貫したパフォーマンスを必要とするオペレーションワークロード向けに構築され、最適化されています。例えば、DynamoDB は、10 ユーザーの場合でも 1 億ユーザーの場合でも、ショッピングカートのユースケースで一貫した 1 桁ミリ秒のパフォーマンスを提供します。2012 年にリリースされた DynamoDB は、コストの削減と大規模運用におけるパフォーマンスの向上を実現しながら、リレーショナルデータベースからの移行を支援し続けています。

DynamoDB を使用して、すべての規模、業界、地域のお客様が、小規模から始めてグローバルにスケールできる最新のサーバーレスアプリケーションを構築しています。DynamoDB は、ほぼすべてのサイズのテーブルをサポートするようにスケールしながら、一貫した 1 桁ミリ秒のパフォーマンスと高可用性を提供します。

Amazon プライムデーなどのイベントの場合、DynamoDB は、Alexa、Amazon.com サイト、すべての Amazon フルフィルメントセンターなど、トラフィックの多い複数の Amazon の施設とシステムを強化します。このようなイベントでは、DynamoDB API は、Amazon の施設とシステムからの数兆にもおよぶ呼び出しを処理してきました。DynamoDB は、ピークトラフィックが 1 秒あたり 50 万件を超えるテーブルで、数百人ものお客様に継続的にサービスを提供しています。また、テーブルサイズが 200 TB を超える数百のお客様に対応し、1 時間あたり 10 億件を超えるリクエストを処理します。

トピック

- DynamoDB の特徴
- DynamoDB のユースケース
- DynamoDB の機能
- サービス統合
- セキュリティ
- レジリエンス
- DynamoDB にアクセスする
- DynamoDB の料金

- DynamoDB の使用開始

DynamoDB の特徴

サーバーレス

DynamoDB では、サーバーのプロビジョニングや、ソフトウェアへのパッチの適用、ソフトウェアの管理、インストール、保守、運用は必要はありません。DynamoDB はダウンタイムのないメンテナンスを提供します。バージョン (メジャー、マイナー、パッチ) は存在せず、メンテナンスウィンドウもありません。

DynamoDB のオンデマンドキャパシティモードは、読み込みおよび書き込みリクエストの料金が従量制であるため、お客様は使用した分だけを支払います。オンデマンドでは、DynamoDB はテーブルを即座にスケールアップまたはスケールダウンして容量を調整し、管理なしでパフォーマンスを維持します。また、ゼロまでスケールダウンできるため、テーブルにトラフィックがなく、コールドスタートがない場合はスループットに対する料金は発生しません。

NoSQL

DynamoDB は NoSQL データベースで、従来のリレーショナルデータベースと比較して、パフォーマンス、スケーラビリティ、管理性、柔軟性を向上させる目的のために構築されています。DynamoDB は、さまざまなユースケースをサポートするために、key-value モデルとドキュメントデータモデルの両方をサポートしています。

リレーショナルデータベースとは異なり、DynamoDB は JOIN 演算子をサポートしません。クエリへの回答に必要なデータベースのラウンドトリップと処理能力を減らすために、データモデルを非正規化することをお勧めします。NoSQL データベースとして、DynamoDB はエンタープライズグレードのアプリケーションを構築するための強力な読み取り整合性と ACID トランザクションを提供します。

フルマネージド型

DynamoDB はフルマネージド型のデータベースサービスで、データベース管理の画一的で負荷の高いを作業を処理し、お客様は顧客への価値の提供に集中できます。DynamoDB は、セットアップ、設定、メンテナンス、高可用性、ハードウェアプロビジョニング、セキュリティ、バックアップ、モニタリングなどをお客様に代わって行います。これにより、DynamoDB テーブルを作成するだけで、本番環境のワークロードに対応できるようになります。DynamoDB は、アップグレードやダウ

ンタイムを必要とせず、可用性、信頼性、パフォーマンス、セキュリティ、機能を継続的に改善します。

あらゆるスケールで 1 桁ミリ秒のパフォーマンス

DynamoDB は、リレーショナルデータベースのパフォーマンスとスケーラビリティを向上させ、あらゆる規模で 1 桁ミリ秒のパフォーマンスを実現することを目的として構築されています。このスケールとパフォーマンスを実現するために、DynamoDB は高パフォーマンスのワークロードに最適化され、データベースの効率的な使用を促進する API を提供します。JOIN オペレーションなど、非効率でパフォーマンスの低い機能を大規模に除外します。DynamoDB は、100 ユーザーの場合でも 1 億ユーザーの場合でも、アプリケーションで一貫した 1 桁ミリ秒のパフォーマンスを提供します。

DynamoDB のユースケース

DynamoDB を使用して、すべての規模、業界、地域のお客様が、小規模から始めてグローバルにスケールできる最新のサーバーレスアプリケーションを構築しています。DynamoDB は、運用上のオーバーヘッドがほとんどないかゼロで、あらゆる規模で一貫したパフォーマンスを必要とするユースケースに最適です。次のリストは、DynamoDB を使用できるいくつかのユースケースを示しています。

- 金融サービスアプリケーション — ライブ取引とルーティング、ローン管理、トークン生成、取引台帳などのアプリケーションを構築する金融サービス企業の場合、DynamoDB のグローバルテーブルを使用すると、アプリケーションはイベントに応答し、選択した AWS リージョン からのトラフィックを高速なローカル読み取り/書き込みパフォーマンスで処理できます。

  DynamoDB は、可用性要件が最も厳しいアプリケーションに適しています。DynamoDB を使用すると、ストレージやスループットの強化、バージョニング、ライセンスのために。インスタンスを手動でスケーリングする運用上の負担がなくなります。

  DynamoDB トランザクションを使用して、1 回のリクエストで 1 つ以上のテーブルでアトミック性、整合性、分離性、耐久性 (ACID) を実現できます。(ACID) トランザクションは、金融取引の処理や注文の処理を含むワークロードに適しています。DynamoDB はワークロードの増減に瞬時に対応するため、取引時間などの市場状況に合わせてデータベースを効率的にスケーリングできます。
- ゲームアプリケーション – ゲーム会社では、ゲームの状態、プレイヤーデータ、セッション履歴、リーダーボードなど、ゲームプラットフォームのすべての部分において DynamoDB を使用できます。DynamoDB は、スケーラビリティ、一貫したパフォーマンス、およびサーバーレスアーキテクチャによる操作の容易性を提供します。DynamoDB は、ゲームの成功に必要なスケールア

ウトアーキテクチャに適しています。DynamoDB は、ゲームのスループットをすばやくスケールインおよびスケールアウト (コールドスタートなしでゼロにスケーリング) します。このスケーラビリティにより、ピークトラフィックをスケールアウトする場合も、ゲームプレイの使用量が少ない場合にスケールバックする場合も、アーキテクチャの効率が最適化されます。

- ストリーミングアプリケーション – メディアおよびエンターテイメント企業は、コンテンツのメタデータインデックス、コンテンツ管理サービス、スポーツのスタッツのほぼリアルタイムの提供のために DynamoDB を使用することができます。また、DynamoDB を使用して、ユーザーのウォッチリストとブックマークサービスを提供し、レコメンデーションを生成するために 1 日に数十億の顧客イベントを処理することができます。これらのお客様は、DynamoDB のスケーラビリティ、パフォーマンス、耐障害性からメリットを得られます。DynamoDB は、ワークロードの増減に合わせてスケールし、あらゆるレベルの需要に対応できるストリーミングメディアのユースケースを可能にします。

さまざまな業界における DynamoDB のお客様事例の詳細については、「Amazon DynamoDB Customers」および「This is My Architecture」を参照してください。

DynamoDB の機能

グローバルテーブルによるマルチアクティブレプリケーション

グローバルテーブルは、99.999% の可用性で、選択した AWS リージョン 全体でデータのマルチアクティブレプリケーションを提供します。グローバルテーブルは、マルチリージョン、マルチアクティブデータベースをデプロイするためのフルマネージドソリューションです。独自のレプリケーションソリューションを構築および管理する必要はありません。グローバルテーブルを使用すると、テーブルを利用できる AWS リージョン リージョンを指定できます。DynamoDB は、進行中のデータ変更をこれらのすべてのテーブルにレプリケートします。

グローバルに分散されたアプリケーションは、選択したリージョン内のデータにローカルにアクセスして、1 桁ミリ秒の読み取りおよび書き込みパフォーマンスを実現します。グローバルテーブルはマルチアクティブであるため、プライマリテーブルは必要ありません。つまり、リージョン間でアプリケーションをフェイルオーバーする場合、複雑なフェイルオーバーやフェイルオーバーの遅延、データベースのダウンタイムはありません。

ACID トランザクション

DynamoDB はミッションクリティカルなワークロード向けに構築されています。これには、複雑なビジネスロジックを必要とするアプリケーションでの (ACID) トランザクションのサポートが含まれ

ます。DynamoDB は、トランザクションのサーバー側サポートをネイティブに提供し、テーブル内およびテーブル間の複数の項目を調整したり、変更しないといった、開発者エクスペリエンスを簡素化します。

イベント駆動型アーキテクチャ向けの変更データキャプチャ

DynamoDB は、項目レベルの変更データキャプチャ (CDC) レコードのストリーミングをほぼリアルタイムでサポートします。DynamoDB には、DynamoDB Streams と DynamoDB 用 Kinesis Data Streams の 2 つの CDC 用 ストリーミングモデルがあります。アプリケーションがテーブル内の項目を作成、更新、または削除するたびに、ストリームは項目レベルのすべての変更の時系列シーケンスをほぼリアルタイムで記録します。これにより、DynamoDB Streams は、イベント駆動型アーキテクチャのアプリケーションでの変更の消費と変更への対応に適しています。

セカンダリインデックス

DynamoDB は、グローバルセカンダリインデックスとローカルセカンダリインデックスの両方を作成するオプションを備えており、代替キーを使用してテーブルデータをクエリできます。これらのセカンダリインデックスを使用すると、プライマリキー以外の属性でデータにアクセスできるため、データに最大限の柔軟性がもたらされます。

サービス統合

DynamoDB は複数の AWS のサービス と広く統合されており、データからより多くの価値を引き出し、画一的で負荷の高い作業を排除して、ワークロードの大規模な運用に役立ちます。これらのサービスには、AWS CloudFormation、Amazon CloudWatch、Amazon S3、AWS Identity and Access Management (IAM)、AWS Auto Scaling が含まれます。以下のセクションでは、DynamoDB を使用して実行できるサービス統合の一部について説明します。

サーバーレス統合

エンドツーエンドのサーバーレスアプリケーションを構築するために、DynamoDB は多数のサーバーレス AWS のサービス とネイティブに統合できます。例えば、DynamoDB を AWS Lambda と統合すると、DynamoDB Streams のイベントに自動的に応答するコードであるトリガーを作成することができます。トリガーを使用すると、DynamoDB テーブル内のデータ変更に対応するイベント駆動型アプリケーションを構築できます。コストを最適化するために、Lambda が DynamoDB ストリームから処理するイベントをフィルタリングできます。

次のリストは、DynamoDB とのサーバーレス統合の例を示しています。

- AWS AppSync での GraphQL API の作成
- Amazon API Gateway での REST API の作成
- サーバーレスコンピューティングでの Lambda
- Amazon Kinesis Data Streams での変更データキャプチャ (CDC)

Amazon S3 との間のデータのインポートとエクスポート

DynamoDB を Amazon S3 と統合することで、分析と機械学習のためにデータを Amazon S3 バケットに簡単にエクスポートできます。DynamoDB は、テーブルのフルエクスポートと増分エクスポートをサポートし、指定した期間に変更、更新、または削除されたデータをエクスポートできます。また、新しい DynamoDB テーブルに Amazon S3 からデータをインポートすることもできます。

ゼロ ETL 統合

DynamoDB は、Amazon Redshift とのゼロ ETL 統合および Amazon DynamoDB での OpenSearch Ingestion の使用をサポートしています。これらの統合により、複雑な分析を実行し、DynamoDB テーブルデータで高度な検索機能を使用できます。ゼロ ETL 統合は、DynamoDB で実行されている本番稼働ワークロードには影響しません。

キャッシュ

DynamoDB Accelerator (DAX) は、DynamoDB 用に構築されたフルマネージド型の高可用性キャッシュサービスです。DAX は、1 秒あたり数百万のリクエストの場合でも、最大 10 倍のパフォーマンス向上 (ミリ秒からマイクロ秒) を実現します。DAX は、ユーザーによるキャッシュの無効化の管理、データの入力、クラスターの管理を必要とすることなく、DynamoDB テーブルへのインメモリアクセラレーションの追加に必要なすべての作業を行います。

セキュリティ

DynamoDB は IAM を使用して、DynamoDB リソースへのアクセスをコントロールします。IAM を使用すると、DynamoDB ユーザーがアクセスできるリソースをコントロールするアクセス許可を一元的に管理できます。IAM を使用して、誰を認証 (サインイン) し、誰にリソースの使用を認可する (アクセス許可を付与する) かを制御します。DynamoDB は IAM を使用するため、DynamoDB にアクセスするためのユーザー名やパスワードは必要ありません。管理すべき複雑なパスワードローテーションポリシーがないため、セキュリティ体制が簡素化されます。また IAM では、きめ細かなアクセスコントロールを有効にして、属性レベルでの認可を行うことができます。IAM Access Analyzer

とブロックパブリックアクセス (BPA) をサポートするリソースベースのポリシーを定義して、ポリシー管理を簡素化することもできます。

デフォルトで、DynamoDB は保管中の顧客データをすべて暗号化します。保管時の暗号化は、AWS Key Management Service (AWS KMS) に保存されている暗号化キーを使用して、データのセキュリティを強化します。保管時に暗号化することで、セキュリティを重視したアプリケーションを構築して、暗号化のコンプライアンスと規制の厳格な要件を満たすことができます。暗号化されたテーブルにアクセスすると、DynamoDB はテーブルデータを透過的に復号化します。暗号化されたテーブルの使用あるいは管理のためにコードやアプリケーションを変更する必要はありません。DynamoDB は、期待される 1 桁ミリ秒のレイテンシーを継続的に提供し、すべての DynamoDB クエリは暗号化されたデータでシームレスに機能します。

DynamoDB でのユーザーデータの暗号化には、AWS 所有のキー (デフォルトの暗号化キー)、AWS マネージドキー、またはカスタマーマネージドキーのいずれかの使用を指定できます。AWS 所有の KMS キーを使用したデフォルトの暗号化は、追加料金なしで利用できます。クライアント側の暗号化には、AWS Database Encryption SDK を使用できます。

また、DynamoDB は HIPAA、PCI DSS、GDPR などの複数のコンプライアンス標準に準拠しているため、規制要件も満たすことができます。

レジリエンス

デフォルトで、DynamoDB は 3 つのアベイラビリティーゾーンにデータを自動的にレプリケートし、高い耐久性と 99.99% の可用性 SLA を提供します。DynamoDB は、ビジネス継続性とディザスタリカバリの目標の達成に役立つ追加機能も備えています。

DynamoDB には、データの耐障害性とバックアップのニーズのサポートに役立つ以下の機能が含まれています。

機能

- グローバルテーブル
- 継続的バックアップとポイントインタイムリカバリ
- オンデマンドバックアップと復元

グローバルテーブル

DynamoDB グローバルテーブルは、99.999% の可用性 SLA とマルチリージョンレジリエンスを実現します。これにより、回復力の高いアプリケーションを構築し、目標復旧時間 (RTO) と目標復

旧時点 (RPO) を最小限に抑えるように最適化できます。また、グローバルテーブルは AWS Fault Injection Service (AWS FIS) と統合し、グローバルテーブルワークロードでフォールトインジェクションテストを行います。例えば、レプリカテーブルへのグローバルテーブルレプリケーションを一時停止します。

継続的バックアップとポイントインタイムリカバリ

継続的バックアップは、秒単位の粒度とポイントインタイムリカバリを開始する機能を提供します。ポイントインタイムリカバリを使用すると、過去 35 日間の任意の時点に、秒単位でテーブルを復元することができます。リカバリ期間は、1 ~ 35 日間の任意の値を設定できます。

継続的バックアップとポイントインタイム復元の開始では、プロビジョニングされたキャパシティは使用されません。また、アプリケーションのパフォーマンスや可用性にも影響しません。

オンデマンドバックアップと復元

オンデマンドバックアップと復元を使用して、テーブルの完全なバックアップを作成し、規制やコンプライアンス要件を満たすための長期間の保存とアーカイブを行うことができます。バックアップはテーブルのパフォーマンスには影響せず、任意のサイズのテーブルをバックアップできます。AWS Backup 統合では、AWS Backup を使用して DynamoDB のオンデマンドバックアップのライフサイクルのスケジューリング、コピー、タグ付け、管理を自動的に行えます。AWS Backup を使用すると、アカウント間およびリージョン間でオンデマンドバックアップをコピーし、古いバックアップをコールドストレージに移行してコストを最適化できます。

DynamoDB にアクセスする

DynamoDB は、AWS マネジメントコンソール、AWS Command Line Interface、DynamoDB 用の NoSQL Workbench、または DynamoDB API を使用して操作することができます。

詳細については、「DynamoDB にアクセスする」を参照してください。

DynamoDB の料金

DynamoDB では、テーブル内のデータの読み取り、書き込み、保存、および有効化されたオプション機能に対して料金が発生します。DynamoDB にはオンデマンドとプロビジョニング済みという 2 種類のキャパシティーモードがあり、それぞれのモードにおけるテーブルの読み書き処理について個別の請求オプションがあります。

DynamoDB は、常時無料利用枠にも含まれており、25 GB のストレージを提供します。常時無料利用枠には、25 個のプロビジョニング済み書き込みキャパシティーユニット、および 25 個のプロビジョニング済み読み込みキャパシティーユニット (WCU、RCU) も含まれています。これは、1 か月あたり 2 億件のリクエストを処理するのに十分なキャパシティーです。

詳細については、「Amazon DynamoDB 料金」を参照してください。

DynamoDB の使用開始

DynamoDB を初めて使用するユーザーは、初めに以下のトピックを読むことをお勧めします。

- DynamoDB の使用開始 – DynamoDB のセットアップ、サンプルテーブルの作成、データのアップロードのプロセスについて説明します。このトピックでは、AWS マネジメントコンソール、AWS CLI、NoSQL Workbench、および DynamoDB API を使用して基本的なデータベースオペレーションを実行する方法についても説明します。
- DynamoDB のコアコンポーネント — DynamoDB の基本的な概念について説明します。
- DynamoDB を使用した設計とアーキテクチャの設計に関するベストプラクティス – NoSQL の設計、DynamoDB の Well-Architected レンズ、テーブルの設計、その他のいくつかの DynamoDB 機能に関するベストプラクティスについて説明します。これらのベストプラクティスは、DynamoDB を使用する際のパフォーマンスを最大化し、スループットコストを最小限に抑えるのに役立ちます。

DynamoDB を理解するためのエンドツーエンドの一連の手順について説明する、以下のチュートリアルもお勧めします。これらのチュートリアルは、常時無料利用枠機能を使用して完了できます。

- Amazon DynamoDB で NoSQL テーブルを作成してクエリを実行する
- Build an Application Using a NoSQL Key-Value Data Store (NoSQL キー値データストアを使ってアプリケーションを構築する

DynamoDB に移行するためのリソース、ツール、戦略については、DynamoDB への移行」を参照してください。最新のブログとホワイトペーパーを読むには、「Amazon DynamoDB リソース」を参照してください。

DynamoDB の使用開始

DynamoDB テーブルの接続、作成、管理の方法について、以下のセクションで説明します。

開始する前に、Amazon DynamoDB の基本概念を理解する必要があります。Amazon DynamoDB とは の簡単な概要と、Amazon DynamoDB のコアコンポーネント のより詳細な説明を確認できます。次に、「前提条件」に進みます。

> **Note**
>
> AWS にサインアップすると、AWS 無料利用枠を利用して、DynamoDB の使用を開始できます。Amazon DynamoDB の無料利用枠の特典をまだ超えていない場合、このセクションの例を完了しても料金は発生しません。それ以外の場合は、テーブルを作成した時点からテーブルを削除するまで、標準の DynamoDB 使用料が発生します。
> 無料利用枠アカウントにサインアップしない場合は、DynamoDB local (ダウンロード可能バージョン) をコンピュータでセットアップできます。ダウンロード可能なバージョンでは、AWS アカウントにサインアップしたり、DynamoDB ウェブサービスにアクセスしたりせずに、ローカルでアプリケーションを開発およびテストできます。

トピック

- 初回ユーザー向けの Amazon DynamoDB リソース
- DynamoDB にアクセスする
- 前提条件
- DynamoDB のセットアップ
- ステップ 1: DynamoDB にテーブルを作成する
- ステップ 2: DynamoDB テーブルにデータを書き込む
- ステップ 3: DynamoDB テーブルからデータを読み込む
- ステップ 4: DynamoDB テーブルのデータを更新する
- ステップ 5: DynamoDB テーブル内のデータにクエリを実行
- ステップ 6: (オプション) DynamoDB テーブルを削除してリソースをクリーンアップする
- DynamoDB について学習を続ける
- Console-to-Code を使用して Amazon DynamoDB のインフラストラクチャコードを生成する

初回ユーザー向けの Amazon DynamoDB リソース

初めて使用する方は、以下のセクションに目を通し、必要に応じて参照することをお勧めします。

主なサービスと料金 DynamoDB の一般的な製品概要、一般的なユースケース、主なサービス、料金について説明します。	DynamoDB リソース サービス、データモデリングの概念、主要な特徴と機能を紹介する動画、チュートリアル、規範ガイダンスがあります。
開始方法 DynamoDB のセットアップ、サンプルテーブルの作成、データのアップロードに関する情報が記載されています。	DynamoDB 基本コース テーブル設計、データ型、基本オペレーションなど、DynamoDB の基本を学習する無料のデジタルコース。
DynamoDB ナゲット DynamoDB の主要な概念と機能を説明する、焦点を絞った短い動画チュートリアルのコレクション。	DynamoDB コードサンプルリポジトリ さまざまなプログラミング言語での実践的な DynamoDB コード例。
無料の DynamoDB トレーニング AWS は、DynamoDB の概念、機能、ベストプラクティスを対象とした無料のデジタルトレーニングコースを提供します。	DynamoDB 用の NoSQL Workbench データモデリング、データの視覚化、クエリ開発機能を提供する統合ビジュアルツール。
DynamoDB 設計パターン 実用的なコードサンプルを使用した、さまざまなユースケースのベストプラクティスとデータモデリングの例。	実践的なチュートリアル 一般的な DynamoDB タスクについて説明する AWS マネジメントコンソールにおけるステップバイステップのチュートリアル。
DynamoDB への移行 データベースを DynamoDB に移行するためのプロセス、ツール、戦略の概要。	DynamoDB 用の AWS Well-Architected レンズ DynamoDB を使用して信頼性、安全性、効率性、費用対効果の高いアプリケーションを設計

	および運用するためのアーキテクチャのベストプラクティスを提供します。

Amazon DynamoDB の初回ユーザー向けの追加のベストプラクティス

前述のセクションを完了したら、これらのセクションを参照してください。

- DynamoDB のスループットキャパシティ

  DynamoDB テーブルで使用できる 2 つのスループットモードの概要と、アプリケーションに適したキャパシティモードを選択する際の考慮事項について説明します。オンデマンドモードは、ほとんどの DynamoDB ワークロードで推奨されるデフォルトのスループットオプションです。

- ベストプラクティス

  問題を特定して対処し、DynamoDB を使用する際のパフォーマンスを最大化し、コストを最小限に抑えます。

AWS CLI リソース

AWS コマンドラインインターフェイス (AWS CLI) を使用する場合は、これらのドキュメントが使用開始に役立ちます。

- AWS CLI ドキュメント

  このセクションには、AWS CLI のダウンロード、システムでの AWS CLI の実行、AWS 認証情報の指定に関する情報が記載されています。

- DynamoDB 向けの AWS CLI ドキュメント

  この別個のドキュメントでは、DynamoDB 用のすべての AWS CLI コマンド (構文と例を含む) について説明します。

プログラミングリソース

さまざまな一般的プログラミング言語を使用して、DynamoDB API を使用するアプリケーションプログラムを記述できます。次にいくつかのリソースを示します。

- Amazon Web Services のツール

AWS には、DynamoDB をサポートする多数の Software Development Kit (SDK) が用意されています。DynamoDB のコードは、Java、.NET、PHP、Ruby、その他の言語で作成できます。これらの SDK は、DynamoDB へのリクエストのフォーマット、レスポンスの解析、再試行ロジックとエラー処理の提供により、アプリケーション開発を大幅に簡素化します。

- DynamoDB API リファレンス

  AWS SDK を使用しない場合は、DynamoDB API を使用して DynamoDB を直接操作できます。このドキュメントでは、DynamoDB のすべての API オペレーション (構文と例を含む) について説明します。リクエストを作成および認証してレスポンスを処理する際のトラブルシューティングのヒントと情報をこのセクションで確認できます。

DynamoDB にアクセスする

Amazon DynamoDB には、AWS マネジメントコンソール、AWS Command Line Interface (AWS CLI)、または DynamoDB API を使用してアクセスできます。

トピック

- コンソールを使用する
- の使用AWS CLI
- API の使用
- DynamoDB 用の NoSQL Workbench の使用
- IP アドレスの範囲
- インターネットプロトコルバージョン 6 (IPv6) のデュアルスタックエンドポイント

コンソールを使用する

Amazon DynamoDB の AWS マネジメントコンソール、https://console.aws.amazon.com/dynamodb/home からアクセスできます。

DynamoDB コンソールで実行できるアクションの一部を以下に示します。

- テーブルの管理: テーブルを作成、更新、削除する。キャパシティ計算ツールは、キャパシティ要件の見積もりに役立ちます。
- データの操作: テーブル内の項目を表示、追加、更新、削除する。Time to Live (TTL) 設定を管理する。

- モニタリングと分析: ダッシュボードの表示、アラームのモニタリングと設定、DynamoDB テーブルのメトリクスとアラートの分析を行う。
- 最適化と拡張: セカンダリインデックス、ストリーム、トリガー、リザーブドキャパシティ、その他の高度な機能を管理し、DynamoDB の使用を強化する。

DynamoDB コンソールは、DynamoDB リソースを管理するための包括的なインターフェイスを提供します。コンソールにアクセスして、調べて詳細を確認してください。

の使用AWS CLI

AWS Command Line Interface (AWS CLI) を使用すると、複数の AWS のサービスをコマンドラインから制御したり、スクリプトで自動化したりできます。テーブルの作成など、その場限りのオペレーションに AWS CLI を使用できます。また、ユーティリティスクリプト内に Amazon DynamoDB オペレーションを埋め込むときにも使用できます。

DynamoDB で AWS CLI を使用するには、事前にアクセスキー ID とシークレットアクセスキーを取得する必要があります。詳細については、「管理者権限を持つ IAM ユーザーを作成する」を参照してください。

AWS CLI で DynamoDB 向けに使用できるすべてのコマンドの完全な一覧については、「AWS CLI コマンドリファレンス」を参照してください。

AWS CLI のダウンロードと設定

AWS CLI は、http://aws.amazon.com/cli で入手できます。Windows、macOS、または Linux 上で実行できます。AWS CLI をダウンロードしたら、以下の手順に従って、インストールと設定を行います。

1. AWS Command Line Interface ユーザーガイドに移動します。
2. AWS CLI のインストール、AWS CLI の設定の手順に従います。

DynamoDB での AWS CLI の使用

コマンドラインの形式は、DynamoDB オペレーション名と、それに続くそのオペレーションのパラメータで構成されます。AWS CLI では、パラメータ値の短縮構文および JSON をサポートしています。

例えば、次のコマンドでは、Music という名前のテーブルを作成します。パーティションキーは Artist で、ソートキーは SongTitle です。(読みやすくするために、このセクションの長いコマンドは、複数の行に分かれています)。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

次のコマンドでは、新しい項目をテーブルに追加します。この例では、短縮構文と JSON を組み合わせて使用しています。

```
aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"},
 "AlbumTitle": {"S": "Somewhat Famous"}}' \
    --return-consumed-capacity TOTAL

aws dynamodb put-item \
    --table-name Music \
    --item '{
        "Artist": {"S": "Acme Band"},
        "SongTitle": {"S": "Happy Day"},
        "AlbumTitle": {"S": "Songs About Life"} }' \
    --return-consumed-capacity TOTAL
```

コマンドラインで、有効な JSON を作成するのは難しい場合があります。ただし、AWS CLI で JSON ファイルを読み込むことができます。例えば、key-conditions.json という名前のファイルに格納されている次の JSON コードスニペットがあるとします。

```
{
    "Artist": {
        "AttributeValueList": [
            {
                "S": "No One You Know"
```

```
            }
        ],
        "ComparisonOperator": "EQ"
    },
    "SongTitle": {
        "AttributeValueList": [
            {
                "S": "Call Me Today"
            }
        ],
        "ComparisonOperator": "EQ"
    }
}
```

次のように、Query を使用して、AWS CLI リクエストを発行できます。この例では、key-conditions.json ファイルの内容は、--key-conditions パラメータに使用されます。

```
aws dynamodb query --table-name Music --key-conditions file://key-conditions.json
```

DynamoDB Local での AWS CLI の使用

AWS CLI を使用して、コンピュータで実行されている DynamoDB Local (ダウンロード可能なバージョン) を操作することもできます。これを有効にするには、各コマンドに次のパラメータを追加します。

```
--endpoint-url http://localhost:8000
```

次の例では、AWS CLI を使用して、ローカルデータベースのテーブルを一覧表示します。

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

DynamoDB がデフォルト (8000) 以外のポート番号を使用している場合は、それに応じて --endpoint-url 値を変更する必要があります。

> **Note**
>
> AWS CLI では、DynamoDB Local (ダウンロード可能なバージョン) をデフォルトのエンドポイントとして使用することはできません。そのため、各コマンドで --endpoint-url を指定する必要があります。

API の使用

AWS マネジメントコンソール と AWS Command Line Interface を使用して、Amazon DynamoDB とインタラクティブに作業できます。ただし、DynamoDB を最大限に活用するためには、AWS SDK を使用してアプリケーションコードを記述できます。

AWS SDK は、DynamoDB の広範なサポート (Java、ブラウザの JavaScript、.NET、Node.js、PHP、Python、Ruby、C++、Go、Android、iOS) を提供します。

DynamoDB を使用して AWS を使用するには、AWS アクセスキー ID とシークレットアクセスキーを取得する必要があります。詳細については、「DynamoDB (ウェブサービス) の設定」を参照してください。

AWS SDK を使用した DynamoDB アプリケーションのプログラミングに関する概要については、「DynamoDB と AWS SDK を使用したプログラミング」を参照してください。

DynamoDB 用の NoSQL Workbench の使用

また、DynamoDB には、DynamoDB 用の NoSQL Workbench をダウンロードして使用することによってアクセスできます。

Amazon DynamoDB 用の NoSQL Workbench は、最新のデータベース開発および運用向けのクロスプラットフォームのクライアント側 GUI アプリケーションです。Windows、macOS、Linux で使用できます。NoSQL Workbench は、DynamoDB テーブルの設計、作成、クエリ、管理に役立つデータモデリング、データ可視化、クエリ開発といった特徴を提供する視覚的開発ツールです。NoSQL Workbench に、インストールプロセスのオプションとして DynamoDB local が含まれるようになったため、DynamoDB local でデータを簡単にモデル化できます。DynamoDB local とその要件の詳細については、「DynamoDB local (ダウンロード可能バージョン) のセットアップ」を参照してください。

> **Note**
>
> NoSQL Workbench for DynamoDB は、現在、2 要素認証 (2FA) で構成されている AWS ログインをサポートしていません。

データモデリング

DynamoDB 用の NoSQL Workbench を使用すると、アプリケーションのデータアクセスパターンを満たす既存のデータモデルから新しいデータモデルを構築したり、既存のデータモデルに

基づいてモデルを設計したりできます。プロセスの最後に、設計されたデータモデルをインポートおよびエクスポートすることもできます。詳細については、「NoSQL Workbench を使用したデータモデルの構築」を参照してください。

オペレーション構築

NoSQL Workbench は、クエリを開発およびテストするための豊富なグラフィカルユーザーインターフェイスを提供します。オペレーションビルダーを使用して、データセットを表示、探索、およびクエリできます。構造化オペレーションビルダーを使用して、データプレーンオペレーションを構築および実行することもできます。プロジェクションと条件式をサポートし、複数の言語でサンプルコードを生成できます。詳細については、「NoSQL Workbench を使用したデータセットの探索とオペレーションの構築」を参照してください。

IP アドレスの範囲

Amazon Web Services (AWS) は、その現在の IP アドレス範囲を JSON 形式で公開します。現在の範囲を表示するには、「ip-ranges.json」ファイルをダウンロードします。詳細については、AWS 全般のリファレンス の AWS IP アドレスの範囲をご参照ください。

DynamoDB テーブルおよびインデックスへのアクセスに使用できる IP アドレス範囲を見つけるには、ip-ranges.json で文字列 `"service": "DYNAMODB"` を検索します。

> **Note**
>
> この IP アドレス範囲は、DynamoDB Streams および DynamoDB Accelerator (DAX) には適用されません。

インターネットプロトコルバージョン 6 (IPv6) のデュアルスタックエンドポイント

DynamoDB は、IPv4 と IPv6 の両方と互換性のあるデュアルスタックエンドポイントを提供します。エンドポイントの命名規則は次のとおりです。

- `dynamodb.<region>.api.aws`
- `<account-id>.ddb.<region>.api.aws`
- `streams-dynamodb.<region>.api.aws`

- search-dynamodb.*<region>*.api.aws
- dax.*<region>*.api.aws
- dynamodb-fips.*<region>*.api.aws

DynamoDB エンドポイントとリージョンの可用性の完全なリストについては、「AWS 全般のリファレンスガイド」の「Amazon DynamoDB エンドポイントとクォータ」のトピックを参照してください。

デュアルスタックエンドポイントを使用するように AWS CLI を設定する方法の詳細については、「AWS コマンドラインインターフェイスガイド」の「すべての AWS のサービスでデュアルスタックのエンドポイントを使用するように設定」のセクションを参照してください。

デュアルスタックエンドポイントを使用するように SDK クライアントを設定する方法の詳細については、「AWS SDK とツールガイド」の「デュアルスタックと FIPS エンドポイント」のトピックを参照してください。

IPv6 で DynamoDB を使用する前に、IPv6 アドレスの範囲を含めるために IP アドレスフィルタリングに使用する IAM ユーザーロールまたはリソースベースのポリシーを更新する必要があります。IPv6 アドレスを考慮しない IP アドレスフィルタリングポリシーは、アクセスの問題を引き起こす可能性があります。詳細については、「AWS Identity and Access Management ガイド」の「IP アドレス条件演算子」のセクションを参照してください。

前提条件

Amazon DynamoDB チュートリアルを開始する前に、DynamoDB にアクセスする で DynamoDB にアクセスする方法について説明します。次に、ウェブサービスを使用するか、DynamoDB のセットアップ でローカルにダウンロードしたバージョンを使用して DynamoDB を設定します。その後、「ステップ 1: DynamoDB にテーブルを作成する」に進みます。

> **Note**
>
> - AWS マネジメントコンソール のみを使用して DynamoDB とやり取りする場合、AWS アクセスキーは必要ありません。「AWS にサインアップする」のステップを完了し、「ステップ 1: DynamoDB にテーブルを作成する」に進んでください。
> - 無料利用枠アカウントにサインアップしない場合は、DynamoDB local (ダウンロード可能バージョン) をセットアップできます。次に、「ステップ 1: DynamoDB にテーブルを作成する」に進んでください。

- Linux と Windows のターミナルで CLI コマンドを操作する場合、違いがあります。次のガイドでは、Linux ターミナル (macOSを含む) 用にフォーマットされたコマンドと、Windows CMD 用にフォーマットされたコマンドを紹介します。使用しているターミナルアプリケーションに最適なコマンドを選択してください。

DynamoDB のセットアップ

Amazon DynamoDB ウェブサービスに加えて、AWS はコンピュータで実行できる DynamoDB のダウンロード可能バージョンを提供します。ダウンロード可能なバージョンは、コードの開発とテストに役立ちます。これを使用することで、DynamoDB ウェブサービスにアクセスせずに、ローカルでアプリケーションを書き込んでテストすることができます。

このセクションのトピックでは、DynamoDB (ダウンロード可能バージョン) と DynamoDB ウェブサービスをセットアップする方法について説明します。

トピック

- DynamoDB (ウェブサービス) の設定
- DynamoDB local (ダウンロード可能バージョン) のセットアップ

DynamoDB (ウェブサービス) の設定

Amazon DynamoDB ウェブサービスを使用するには:

1. AWS アカウント へのサインアップ
2. 認証情報の設定 (プログラムで DynamoDB にアクセスするために使用)。

> **Note**
>
> DynamoDB のみを使用して AWS マネジメントコンソール とやり取りする場合、AWS アクセスキーは必要ないため、「コンソールを使用する」に進むことができます。

へのサインアップAWS

AWS アカウント へのサインアップ

AWS を使い始めるには、AWS アカウントが必要です。AWS アカウントの作成の詳細については、「AWS アカウント管理 リファレンスガイド」の「AWS アカウントの開始方法」を参照してください。

認証情報の設定

プログラムまたは AWS CLI を介して DynamoDB にアクセスする前に、アプリケーションの認可が有効になるように認証情報を設定する必要があります。

この方法には、いくつかあります。たとえば、認証情報ファイルを手動で作成して、お客様のアクセスキー ID とシークレットアクセスキーを格納します。また、このファイルは、AWS CLI コマンド `aws configure` を使用して自動的に作成することもできます。または、環境変数を使用できます。認証情報の設定に関する詳細については、プログラム固有の AWS SDK デベロッパーガイドを参照してください。

AWS CLI をインストールして設定する方法については、「の使用AWS CLI」を参照してください。

DynamoDB の他のサービスとの統合

DynamoDB AWS は他の多くのサービスと統合できます。詳細については次を参照してください:

- DynamoDB を他の AWS サービスで使用する
- CloudFormationDynamoDB の
- DynamoDB での AWS Backup の使用
- AWS Identity and Access Management (IAM) と DynamoDB
- Amazon DynamoDB で AWS Lambda を使用する

DynamoDB local (ダウンロード可能バージョン) のセットアップ

ダウンロード可能なバージョンの Amazon DynamoDB では、DynamoDB ウェブサービスにアクセスせずに、アプリケーションを開発してテストすることができます。代わりに、データベースはコンピュータ上で自己完結型となります。アプリケーションを本番稼働環境にデプロイする準備ができたら、コード内のローカルエンドポイントを削除します。その後、これは DynamoDB ウェブサービスを指します。

このローカルバージョンを使用することで、スループットやデータストレージ、データ転送料金を節約しやすくなります。また、アプリケーションを開発している間インターネットに接続しておく必要はありません。

DynamoDB local は、ダウンロード (JRE が必要)、Apache Maven 依存関係、または Docker イメージとして使用できます。

Amazon DynamoDB ウェブサービスを代わりに使用する場合は、「DynamoDB (ウェブサービス) の設定」を参照してください。

トピック

- コンピュータ上で DynamoDB をローカルでデプロイする
- DynamoDB local の使用に関する注意事項
- DynamoDB local のリリース履歴
- DynamoDB local のテレメトリ

コンピュータ上で DynamoDB をローカルでデプロイする

> **Note**
>
> - DynamoDB local は、v3.x (現行)、v2.x (レガシー)、v1.x (廃止) の 3 つのバージョンで利用できます。
> - DynamoDB v3.x は、ローカルでのテストと開発での使用に推奨されます。
> - DynamoDB local V2.x から V3.x に移行するには、インポートステートメントを `com.amazonaws.services.dynamodbv2` から `software.amazon.dynamodb` に更新し、Maven ユーザーは Maven 依存関係を更新する必要があります。
> - SDK for Java v1.x を使用するアプリケーションを SDK for Java 2.x に移行する場合は、AWS SDK for Java 2.x の手順に従います。

DynamoDB Local のダウンロード

これらのステップに従って DynamoDB をコンピュータにセットアップして実行します。

コンピュータに DynamoDB をセットアップするには

1. DynamoDB Local は、次のいずれかの場所から無料でダウンロードできます。

ダウンロードリンク	チェックサム
.tar.gz | .zip	.tar.gz.sha256 | .zip.sha256

> **Important**
>
> コンピュータで DynamoDB v2.6.0 以降を実行するには、Java ランタイム環境 (JRE) バージョン 17.x 以降が必要です。アプリケーションは、以前のバージョンの JRE では動作しません。

2. アーカイブをダウンロードしたら、内容を抽出し、抽出されたディレクトリを任意の場所にコピーします。
3. コンピュータで DynamoDB を開始するには、コマンドプロンプトウィンドウを開き、DynamoDBLocal.jar を抽出したディレクトリに移動し、次のコマンドを入力します。

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
```

> **Note**
>
> Windows PowerShell を使用している場合は、パラメータ名または名前全体と値を次のように囲んでください。
> `java "-Djava.library.path=./DynamoDBLocal_lib" -jar DynamoDBLocal.jar`
> DynamoDB は、停止するまで受信リクエストを処理します。DynamoDB を停止するには、コマンドプロンプトで Ctrl+C を押します。
> DynamoDB は、デフォルトではポート 8000 を使用します。ポート 8000 を使用できない場合、このコマンドにより例外がスローされます。DynamoDB ランタイムオプション (-port を含む) の詳細なリストを表示するには、次のコマンドを入力します。
> `java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help`

4. プログラムまたは AWS Command Line Interface (AWS CLI) を介して DynamoDB にアクセスする前に、アプリケーションで認可が有効になるように認証情報を設定する必要があります。以下の例に示されているように、ダウンロード可能な DynamoDB には、認証情報が必要です。

```
AWS Access Key ID: "fakeMyKeyId"
AWS Secret Access Key: "fakeSecretAccessKey"
Default Region Name: "fakeRegion"
```

`aws configure` の AWS CLI コマンドを使用して、認証情報を設定できます。詳細については、「」を参照してくださいの使用AWS CLI

5. アプリケーションの書き込みを開始します。AWS CLI を使用してローカルで実行中の DynamoDB にアクセスするには、`--endpoint-url` パラメータを使用します。たとえば、次のコマンドを使用して、DynamoDB テーブルを一覧表示します。

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

DynamoDB Local を Docker イメージとして実行する

Amazon DynamoDB のダウンロード可能バージョンは、Docker イメージとして入手できます。詳細については、「dynamodb-local」を参照してください。DynamoDB Local の現在のバージョンを確認するには、次のコマンドを入力します。

```
java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -version
```

AWS Serverless Application Model (AWS SAM) で構築した REST アプリケーションの一部として DynamoDB local を使用する例については、「注文を管理するための SAM DynamoDB アプリケーション」を参照してください。このサンプルアプリケーションは、DynamoDB local をテストで使用する方法を示しています。

DynamoDB local のコンテナも使用するマルチコンテナアプリケーションを実行する場合は、Docker Compose を使用して、アプリケーション内のすべてのサービス (DynamoDB local を含む) を定義して実行します。

Docker Compose を使用して DynamoDB local をインストールして実行するには

1. Docker Desktop をダウンロードしてインストールします。
2. 以下のコードをファイルにコピーし、`docker-compose.yml` として保存します。

```
services:
 dynamodb-local:
```

```
  command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
  image: "amazon/dynamodb-local:latest"
  container_name: dynamodb-local
  ports:
    - "8000:8000"
  volumes:
    - "./docker/dynamodb:/home/dynamodblocal/data"
  working_dir: /home/dynamodblocal
```

アプリケーションと DynamoDB local を別々のコンテナにしたい場合は、次の yaml ファイルを使用します。

```
services:
 dynamodb-local:
   command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
   image: "amazon/dynamodb-local:latest"
   container_name: dynamodb-local
   ports:
     - "8000:8000"
   volumes:
     - "./docker/dynamodb:/home/dynamodblocal/data"
   working_dir: /home/dynamodblocal
 app-node:
   depends_on:
     - dynamodb-local
   image: amazon/aws-cli
   container_name: app-node
   ports:
    - "8080:8080"
   environment:
     AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
     AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
   command:
     dynamodb describe-limits --endpoint-url http://dynamodb-local:8000 --region
 us-west-2
```

この docker-compose.yml スクリプトは、app-node コンテナと dynamodb-local コンテナを作成します。スクリプトは、AWS CLI を使用して app-node コンテナに接続し、アカウントとテーブルの制限を記述するコマンドを dynamodb-local コンテナで実行します。

独自のアプリケーションイメージで使用するには、以下の例の image 値をアプリケーションの値に置き換えます。

```
services:
 dynamodb-local:
   command: "-jar DynamoDBLocal.jar -sharedDb -dbPath ./data"
   image: "amazon/dynamodb-local:latest"
   container_name: dynamodb-local
   ports:
     - "8000:8000"
   volumes:
     - "./docker/dynamodb:/home/dynamodblocal/data"
   working_dir: /home/dynamodblocal
 app-node:
   image: location-of-your-dynamodb-demo-app:latest
   container_name: app-node
   ports:
     - "8080:8080"
   depends_on:
     - "dynamodb-local"
   links:
     - "dynamodb-local"
   environment:
     AWS_ACCESS_KEY_ID: 'DUMMYIDEXAMPLE'
     AWS_SECRET_ACCESS_KEY: 'DUMMYEXAMPLEKEY'
     REGION: 'eu-west-1'
```

> **Note**
>
> YAML スクリプトでは、AWS アクセスキーと AWS シークレットキーを指定する必要がありますが、DynamoDB local にアクセスするための有効な AWS キーである必要はありません。

3. 次のコマンドラインのコマンドを実行します。

```
docker-compose up
```

DynamoDB Local を Apache Maven 依存関係として実行する

> **Note**
>
> SDK for Java v1.x を使用するアプリケーションを SDK for Java 2.x に移行する場合は、AWS SDK for Java 2.x の手順に従います。

Amazon DynamoDB をアプリケーション内で依存関係として使用するには、これらのステップに従います。

DynamoDB を Apache Maven リポジトリとしてデプロイするには

1. Apache Maven をダウンロードし、インストールします。詳細については、「Apache Maven のダウンロード」および「Apache Maven のインストール」を参照してください。
2. DynamoDB の Maven リポジトリをアプリケーションのプロジェクトオブジェクトモデル (POM) ファイルに追加します。

   ```
   <!--Dependency:-->
   <dependencies>
      <dependency>
         <groupId>software.amazon.dynamodb</groupId>
         <artifactId>DynamoDBLocal</artifactId>
         <version>3.3.0</version>
      </dependency>
   </dependencies>
   ```

   Spring Boot 3 や Spring Framework 6 で使用するテンプレートの例:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
   maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>org.example</groupId>
   <artifactId>SpringMavenDynamoDB</artifactId>
   <version>1.0-SNAPSHOT</version>

   <properties>
   ```

```
    <spring-boot.version>3.0.1</spring-boot.version>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.0</version>
    </parent>

<dependencies>
    <dependency>
        <groupId>software.amazon.dynamodb</groupId>
        <artifactId>DynamoDBLocal</artifactId>
        <version>3.3.0</version>
    </dependency>
    <!-- Spring Boot -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>${spring-boot.version}</version>
    </dependency>
    <!-- Spring Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>${spring-boot.version}</version>
    </dependency>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>${spring-boot.version}</version>
    </dependency>
    <!-- Other Spring dependencies -->
    <!-- Replace the version numbers with the desired version -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.0</version>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>6.0.0</version>
    </dependency>
    <!-- Add other Spring dependencies as needed -->
    <!-- Add any other dependencies your project requires -->
</dependencies>
</project>
```

> **Note**
>
> Maven 中央リポジトリの URL を使用することもできます。

DynamoDB local を AWS CloudShell で実行する

AWS CloudShell はブラウザベースの事前に認証されたシェルで、 から直接起動できます。AWS マネジメントコンソールAWS マネジメントコンソールから AWS CloudShell に移動するには、いくつかの方法があります。詳細については、「AWS CloudShell の開始方法」を参照してください。

AWS マネジメントコンソールのどこからでも AWS CloudShell で DynamoDB local を実行するには、次の手順に従います。

AWS CloudShellから AWS マネジメントコンソール で DynamoDB local を実行するには

1. コンソールインターフェースから AWS CloudShell を起動し、使用可能な AWS リージョンを選択して、希望するシェル (Bash、PowerShell、Z シェルなど) に切り替えます。
2. AWS リージョン を選択するには、[リージョンを選択] メニューに移動し、[サポートされている AWS リージョン] を選択します (使用可能なリージョンがハイライト表示されます)。
3. AWS マネジメントコンソールから、次のオプションのいずれかを選択して AWS CloudShellを起動します。

    a. ナビゲーションバーで、AWS CloudShell アイコンを選択します。

    b. [検索] ボックスに「CloudShell」と入力し、[CloudShell] を選択します。

    c. 最近アクセスしたウィジェットで、[CloudShell] を選択します。

    d. コンソールのツールバーから [CloudShell] を選択します。
4. DynamoDB local を AWS CloudShell で実行するには、`dynamodb-local` エイリアスを使用できます。追加のコマンドラインオプションを指定して、DynamoDB local の設定を変更できま

す。使用可能なオプションについては、「the section called “使用に関する注意事項”」を参照してください。

> **Note**
>
> DynamoDB local をバックグラウンドで 実行するには、`dynamodb-local &` を使用して AWS CloudShell で DynamoDB local を実行します。

5. AWS CLI を使用して AWS CloudShell でローカル実行中の DynamoDB にアクセスするには、`--endpoint-url` パラメータを使用します。例えば、次のコマンドを使用して DynamoDB テーブルを一覧表示します。

   ```
   aws dynamodb list-tables --endpoint-url http://localhost:8000
   ```

JAR ファイルのダウンロード、Docker イメージとしての実行、Maven 依存関係としての使用など、DynamoDB をローカルに設定して使用する複数の方法を紹介するサンプルプロジェクトの例については、「DynamoDB Local Sample Java Project」を参照してください。

DynamoDB local の使用に関する注意事項

エンドポイントを除き、ダウンロード可能なバージョンの Amazon DynamoDB で実行するアプリケーションは DynamoDB ウェブサービスでも動作します。ただし、DynamoDB をローカルで使用する場合は、以下に注意する必要があります。

- `-sharedDb` オプションを使用すると、DynamoDB は shared-local-instance.db という名前の単一のデータベースファイルを作成します。DynamoDB に接続するプログラムはいずれも、このファイルにアクセスします。このファイルを削除すると、保存されたすべてのデータを失うことになります。
- `-sharedDb` を省略する場合のデータベースファイルの名前は、myaccesskeyid_region.db で、アプリケーション設定に表示されるとおりの AWS アクセスキー ID と AWS リージョンが使用されます。このファイルを削除すると、保存されたすべてのデータを失うことになります。
- `-inMemory` オプションを使用した場合、DynamoDB はデータベースファイルの書き込みをまったく行いません。代わりに、すべてのデータがメモリに書き込まれ、DynamoDB を終了するときにデータは保存されません。
- `-optimizeDbBeforeStartup` オプションを使用した場合は、`-dbPath` パラメータも指定し、DynamoDB がそのデータベースファイルを見つけられるようにする必要があります。

- DynamoDB 用 AWS SDK では、アプリケーション設定でアクセスキーバリューと AWS リージョンの値を指定する必要があります。-sharedDb または -inMemory オプションを使用している場合を除き、DynamoDB はこれらの値を使用してローカルデータベースファイルに名前を付けます。これらの値は、ローカルで動作する有効な AWS 値である必要はありません。ただし、使用しているエンドポイントを変更して、クラウドでもコードを実行できるように、有効な値を使用する方が便利な場合があります。
- DynamoDB local は billingModeSummary. に対して常に null を返します。
- DynamoDB local の AWS_ACCESS_KEY_ID には、文字 (A ~ Z、a ~ z) と数字 (0 ~ 9) のみを使用できます。
- DynamoDB local はポイントインタイムリカバリ (PITR) をサポートしていません。

トピック

- コマンドラインオプション
- ローカルエンドポイントの設定
- ダウンロード可能な DynamoDB と DynamoDB ウェブサービスの違い

コマンドラインオプション

次のコマンドラインオプションは、ダウンロード可能なバージョンの DynamoDB で使用できます。

- -cors value — JavaScript でクロスオリジンリソース共有 (CORS) のサポートを有効にします。特定のドメインのカンマ区切りの "許可" リストを指定する必要があります。-cors のデフォルト設定は、パブリックアクセスを許可するアスタリスク (*) です。
- -dbPath value — DynamoDB がそのデータベースファイルを書き込むディレクトリ。このオプションを指定しない場合、ファイルは現在のディレクトリに書き込まれます。-dbPath と -inMemory の両方を同時に指定することはできません。
- -delayTransientStatuses — DynamoDB によって特定のオペレーションに遅延が生じる原因になります。DynamoDB (ダウンロード可能バージョン) では、テーブルやインデックスの作成/更新/削除オペレーションなどの一部のタスクを瞬時に行うことができます。ただし、DynamoDB サービスでこれらのタスクを行うには、より時間がかかります。このパラメータを設定すると、コンピュータ上で実行されている DynamoDB で DynamoDB ウェブサービスの動作をより正確にシミュレートしやすくなります。(現在、このパラメータではステータスが CREATING または DELETING のグローバルセカンダリインデックスに対してのみ遅延が発生します。)
- -help — 使用方法の概要とオプションを出力します。

- `-inMemory` — DynamoDB は、データベースファイルを使用する代わりにメモリで実行されます。DynamoDB を停止すると、データは一切保存されません。`-dbPath` と `-inMemory` の両方を同時に指定することはできません。
- `-optimizeDbBeforeStartup` — コンピュータで DynamoDB を開始する前に、基になるデータベーステーブルを最適化します。このパラメータを使用するときは、`-dbPath` も指定する必要があります。
- `-port value` — DynamoDB がアプリケーションと通信するために使用するポート番号。このオプションを指定しない場合、デフォルトポートは `8000` になります。

> **Note**
>
> DynamoDB は、デフォルトではポート 8000 を使用します。ポート 8000 を使用できない場合、このコマンドにより例外がスローされます。`-port` オプションを使用すると、異なるポート番号を指定できます。DynamoDB ランタイムオプション (`-port` を含む) の詳細なリストを表示するには、次のコマンドを入力します。
>
> ```
> java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -help
> ```

- `-sharedDb` — `-sharedDb` を指定した場合、DynamoDB では、認証情報やリージョンごとに別のファイルを使用せずに、単一のデータベースファイルを使用します。
- `-disableTelemetry` — 指定した場合、DynamoDB local はテレメトリを送信しません。
- `-version` — DynamoDB Local のバージョンを出力します。

ローカルエンドポイントの設定

AWS SDK およびツールは、デフォルトで Amazon DynamoDB ウェブサービスのエンドポイントを使用します。ダウンロード可能なバージョンの DynamoDB を使用して SDK およびツールを使用するには、ローカルエンドポイントを指定する必要があります。

```
http://localhost:8000
```

AWS Command Line Interface

AWS Command Line Interface (AWS CLI) を使用して、ダウンロード可能な DynamoDB を操作できます。

ローカルで実行中の DynamoDB にアクセスするには、`--endpoint-url` パラメータを使用します。AWS CLI を使用して、コンピュータにある DynamoDB のテーブルを一覧表示する例を次に示します。

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

> **Note**
>
> AWS CLI では、ダウンロード可能なバージョンの DynamoDB をデフォルトのエンドポイントとして使用することはできません。そのため、各 `--endpoint-url` コマンドで AWS CLI を指定する必要があります。

AWS SDK

エンドポイントを指定する方法は、使用しているプログラミング言語と AWS SDK によって異なります。以下のセクションでは、エンドポイントの指定方法について説明します。

- Java: AWS リージョンとエンドポイントの設定 (DynamoDB local は AWS SDK for Java 1.x および 2.x をサポートしています。SDK for Java 2.x については、「AWS SDK for Java 2.x デベロッパーガイド」を参照してください)
- CodeSamples.Java.RegionAndEndpoint .NET: AWS リージョンとエンドポイントの設定

ダウンロード可能な DynamoDB と DynamoDB ウェブサービスの違い

ダウンロード可能バージョンの DynamoDB は、開発とテストのみを目的としています。これに対して、DynamoDB ウェブサービスは、スケーラビリティ、可用性、耐久性を特徴とする本稼働環境用に最適なマネージドサービスです。

ダウンロード可能なバージョンの DynamoDB とウェブサービスは、次の点で異なります。

- AWS リージョンと個別の AWS アカウントは、クライアントレベルでサポートされません。
- プロビジョンされたスループット設定は、ダウンロード可能な DynamoDB で無視されます。ただし、この設定は `CreateTable` オペレーションで必要です。`CreateTable` の場合、プロビジョニングされた読み込みおよび書き込みスループットに対して任意の数値を指定できます。ただし、この数値は使用されません。`UpdateTable` は 1 日に必要な回数呼び出すことができます。ただし、プロビジョニングされたスループット値に対する変更はいずれも無視されます。

- Scan 操作が連続的に行われます。並列スキャンはサポートされていません。Segment と TotalSegments パラメータ (Scan オペレーション) は無視されます。
- 読み込みおよび書き込みオペレーションの速度は、コンピュータの速度によってのみ制限を受けます。CreateTable、 UpdateTable、および DeleteTable オペレーションはすぐに実行されます。テーブルの状態は常に ACTIVE です。テーブルやグローバルセカンダリインデックスでプロビジョニングされたスループット設定のみを変更する UpdateTable オペレーションは、すぐに実行されます。UpdateTable オペレーションがグローバルセカンダリインデックスを作成または削除する場合、それらのインデックスは、通常の状態 (CREATING や DELETING など) に移行してから ACTIVE 状態になります。この間、テーブルは ACTIVE のままになります。
- 読み込みオペレーションには結果整合性があります。ただし、コンピュータで実行されている DynamoDB の速度が原因で、ほとんどの読み取りには強力な整合性があるように見えます。
- 項目コレクションのメトリクスや項目コレクションのサイズは追跡されません。オペレーションレスポンスでは、項目コレクションのメトリクスの代わりに、null が返されます。
- DynamoDB では、結果セットごとに、返されるデータに 1 MB の制限があります。DynamoDB ウェブサービスとダウンロード可能バージョンのいずれにもこの制限が適用されます。ただし、インデックスのクエリを実行しているとき、DynamoDB サービスは、射影されたキーと属性のサイズのみを計算します。一方で、ダウンロード可能バージョンの DynamoDB は、項目全体のサイズを計算します。
- DynamoDB Streams を使用している場合、シャードが作成される速度が異なる可能性があります。DynamoDB ウェブサービスでは、シャードの作成動作は部分的にテーブルパーティションアクティビティの影響を受けます。ローカルで DynamoDB を実行している場合は、テーブルパーティションがありません。どちらの場合も、シャードはエフェメラルのため、アプリケーションがシャードの動作の影響を受けることはありません。
- ダウンロード可能な DynamoDB では、TransactionConflictExceptions はトランザクション API に対してスローされません。Java モック作成フレームワークを使用して DynamoDB ハンドラで TransactionConflictExceptions をシミュレートし、競合するトランザクションに対するアプリケーションの応答をテストすることをお勧めします。
- DynamoDB ウェブサービスでは、コンソールまたは AWS CLI のどちらからアクセスする場合でも、テーブル名の大文字と小文字は区別されます。Authors という名前のテーブルと authors という名前のテーブルが、別のテーブルとして両方存在できます。ダウンロード可能バージョンでは、テーブル名で大文字と小文字が区別されず、これら 2 つのテーブルを作成しようとすると、エラーが発生します。
- ダウンロード可能なバージョンの DynamoDB で、タグ付けはサポートされていません。
- ダウンロード可能なバージョンの DynamoDB では、ExecuteStatement の Limit パラメータは無視されます。

- ベクトルインデックスはサポートされていません。`CreateTable` はエラーなしで `VectorIndexes` パラメータを受け入れて無視するため、`DescribeTable` はベクトルインデックスをレポートせず、`SearchVectors` は使用できません。DynamoDB ウェブサービスに対してベクトルインデックスの動作をテストします。

DynamoDB local のリリース履歴

次の表に、DynamoDB local のリリース別の重要な変更点を示します。

バージョン	変更	説明	日付
3.3.1	PartiQL 依存関係を含める	• pom.xml への PartiQL 依存関係の追加 • SDK Java バージョンを最新のパブリックバージョン 2.48.3 に更新 • CVE を修正するための依存関係のアップグレード	2026 年 5 月 28 日
3.3.0	グローバルセカンダリインデックスのマルチ属性キーのサポートを追加	• グローバルセカンダリインデックスのマルチ属性キーのサポートを追加 • SDK Java バージョンを 2.41.0 から 2.41.7 に最新のパブリックバージョンに更新	2026 年 1 月 19 日
3.2.0	複数の Kotlin バージョンとの互換性の問題を修正	• DescribeStream API に ShardFilter パラメータのサポートを追加	2026 年 1 月 9 日

バージョン	変更	説明	日付
		• SDK Java バージョンを 2.33.0 から 2.41.0 に最新のパブリックバージョンに更新 • UpdateTable での DeletionProtection バグの修正	
3.1.0	Joda 時間の依存関係を含む PartiQL クエリのパフォーマンスの向上	• SDK Java バージョンを 2.25.50 から 2.33.0 に最新のパブリックバージョンに更新 • Pom.xml ファイル内に Joda 時間の依存関係を含める • PartiQL クエリのパフォーマンスの向上 • 複数の CVE 脆弱性の問題を修正するための依存関係のアップグレード	2025 年 9 月 14 日

バージョン	変更	説明	日付
3.0.0	AWS SDK Java V1 から V2 への移行	• AWS SDK Java V1 から V2 への移行 • パッケージ構造を `com.amazonaws.services.dynamodbv2` から `software.amazon.dynamodb.services` に更新 • AWS SDK Java V1 の依存関係を削除	2025 年 7 月 17 日

バージョン	変更	説明	日付
2.6.0	DynamoDB API でテーブル ARN をテーブル名としてサポート パフォーマンスの修正とセキュリティ更新	• 複数の DynamoDB API でテーブル ARN をテーブル名としてサポート • Mac M3 などの高性能マシンでの `CreateStreamTable` のバグの修正 • 脆弱性の問題を修正するための依存関係のアップグレード(CVE-2022-49043、CVE-2024-56732、CVE-2020-29582、CVE-2025-21502、CVE-2024-50602、CVE-2025-24970、CVE-2025-25193)	2025 年 3 月 13 日
2.5.4	Jetty の依存関係へのアップグレード	• Jetty 12.0.8 から Jetty 12.0.14 へのアップれーど (CVE-2024-6763、CVE-2024-8184、CVE-2024-47535 を解決) (CVE-2024-21634) の軽減	2024 年 12 月 12 日

バージョン	変更	説明	日付
2.5.3	Log4j Core の Jackson Dependencies を 2.17.x にアップグレード (CVE-2022-1471 を解決)	• Log4j Core の Jackson Dependencies を 2.17.x にアップグレード (CVE-2022-1471 を解決) して、一時的な依存関係である SnakeYAML ライブラリの重要なセキュリティ脆弱性に対処	2024 年 11 月 6 日
2.5.2	テーブル更新ワークフローのバグ修正	• update table を使用して請求モードのオンデマンドで GSI を使ってプロビジョン済みのテーブルを更新しようとする際のワークフローのバグ修正	2024 年 6 月 20 日
2.5.1	OndemandThroughPut 機能で導入されたバグのパッチ	• OndemandThroughPut に関連するいくつかのバグを修正しました。	2024 年 6 月 5 日

バージョン	変更	説明	日付
2.5.0	オンデマンドテーブル用の設定可能な最大スループットのサポート：ReturnValuesOnConditionCheckFailure 、BatchExecuteStatement 、および ExecuteTransaction Request	• 埋め込みモードへのテレメトリの追加 • ConditionalCheckException の SDKv2 翻訳の修正	2024 年 5 月 28 日
2.4.0	ReturnValuesOnConditionCheckFailure のサポート -埋め込みモード	• 複数ストリームでのオペレーションに関する TrimmedDataAccessException の埋め込みモードの修正 • 埋め込みモードの SDKv2 に関する例外変換の修正	2024 年 4 月 17 日
2.3.0	Jetty および JDK のアップグレード	• Jetty 12.0.2 へのアップグレード • JDK 17 へのアップグレード • ANTLR4 から 4.10.1 へのアップグレード	2024 年 3 月 14 日

バージョン	変更	説明	日付
2.2.0	テーブル削除保護と `ReturnValuesOnConditionCheckFailure` パラメータのサポートが追加されました。	• テーブル削除保護のサポートが追加されました。 • ReturnValuesOnConditionCheckFailure サポートが追加されました。 • -version フラグのサポートが追加されました。	2023 年 12 月 14 日
2.1.0	Maven プロジェクト用の SQLLite ネイティブライブラリのサポートとテレメトリの追加	• DynamoDB local へのテレメトリの追加 • Maven プロジェクト用の SQLLite ネイティブライブラリを動的にコピー • io.github.ganadist.sqlite4java ライブラリを Maven 依存関係から削除 • GoogleGuava を 32.1.1-jre にアップグレード	2023 年 10 月 23 日

バージョン	変更	説明	日付
2.0.0	javax から jakarta 名前空間への移行と JDK11 サポート	• javax から jakarta 名前空間への移行と JDK11 サポート • サーバー起動時の無効なアクセスとシークレットキーの処理を修正 • 依存関係を更新して Maven が特定した脆弱性を修正	2023 年 7 月 5 日
1.25.1	Log4j Core の Jackson Dependencies を 2.17.x にアップグレード (CVE-2022-1471 を解決)	Log4j Core の Jackson Dependencies を 2.17.x にアップグレード (CVE-2022-1471 を解決) して、一時的な依存関係である SnakeYAML ライブラリの重要なセキュリティ脆弱性に対処	2024 年 11 月 6 日
1.25.0	テーブル削除保護と `ReturnValuesOnConditionCheckFailure` パラメータのサポートが追加されました。	• テーブル削除保護のサポートが追加されました。 • ReturnValuesOnConditionCheckFailure サポートが追加されました。 • -version フラグのサポートが追加されました。	2023 年 12 月 18 日

バージョン	変更	説明	日付
1.24.0	Maven プロジェクト用の SQLLite ネイティブライブラリのサポートとテレメトリの追加	• DynamoDB local へのテレメトリの追加 • Maven プロジェクト用の SQLLite ネイティブライブラリを動的にコピー • io.github.ganadist.sqlite4java ライブラリを Maven 依存関係から削除 • GoogleGuava を 32.1.1-jre にアップグレード	2023 年 10 月 23 日
1.23.0	サーバー起動時の無効なアクセスとシークレットキーに対処	• サーバー起動時の無効なアクセスとシークレットキーの処理を修正 • 依存関係を更新して Maven が特定した脆弱性を修正	2023 年 6 月 28 日
1.22.0	PartiQL のリミット操作のサポート	• PartiQL 用の IN 句を最適化 • リミット操作のサポート • Maven プロジェクトの M1 サポート	2023 年 6 月 8 日

バージョン	変更	説明	日付
1.21.0	トランザクションあたり 100 アクションをサポート	• トランザクションあたりのアクション数を 25 から 100 に増加 • Docker イメージの Open JDK を 11 にアップグレード • BatchExecuteStatement の項目が重複している場合に発生する例外のパリティを修正	2023 年 1 月 26 日
1.20.0	M1 Mac のサポートを追加	• M1 Mac のサポートを追加 • Jetty の依存関係を 9.4.48.v20220622 にアップグレード	2022 年 9 月 12 日
1.19.0	PartiQL Parser をアップグレード	PartiQL Parser およびその他の関連ライブラリをアップグレード	2022 年 7 月 27 日
1.18.0	log4j-core と Jackson-core をアップグレード	log4j-core を 2.17.1 に、Jackson-core 2.10.x を 2.12.0 にアップグレード	2022 年 1 月 10 日
1.17.2	log4j-core をアップグレード	log4j-core の依存関係をバージョン 2.16 にアップグレード	2021 年 1 月 16 日

バージョン	変更	説明	日付
1.17.1	log4j-core をアップグレード	log4j-core の依存関係を更新し、リモートコード実行を防ぐためにゼロデイエクスプロイトにパッチを適用 - Log4Shell	2021 年 1 月 10 日
1.17.0	JavaScript Web Shell を非推奨化	• AWS SDK 依存関係を AWS SDK for Java 1.12.x に更新 • JavaScript Web Shell を非推奨化	2021 年 1 月 8 日

DynamoDB local のテレメトリ

AWS では、お客様とのやり取りから学んだことに基づいてサービスを開発して提供し、お客様からのフィードバックを使用して製品を練り直しています。テレメトリは、お客様のニーズをより深く理解して、問題を診断し、カスタマーエクスペリエンスを向上させる機能を提供するために役立つ追加情報です。

DynamoDB local は、一般的な使用状況メトリクス、システムおよび環境の情報、エラーなどのテレメトリを収集します。収集されるテレメトリのタイプの詳細については、「収集される情報のタイプ」を参照してください。

DynamoDB local は、ユーザー名や E メールアドレスなどの個人情報は収集しません。また、プロジェクトレベルの機密情報も抽出しません。

お客様は、テレメトリを有効にするかどうかを制御し、いつでも設定を変更できます。テレメトリが有効化されたままの場合、DynamoDB local はテレメトリデータをバックグラウンドで送信します。この際、お客様との追加のやり取りは不要です。

コマンドラインオプションを使用してテレメトリを無効にする

オプション `-disableTelemetry` によって DynamoDB local を起動するときに、コマンドラインオプションを使用してテレメトリを無効にすることができます。詳細については、「コマンドラインオプション」を参照してください。

単一セッションのテレメトリを無効にする

macOS および Linux オペレーティングシステムでは、単一セッションのテレメトリを無効にできます。現在のセッションのテレメトリを無効にするには、以下のコマンドを実行して環境変数 DDB_LOCAL_TELEMETRY を false に設定します。新しいターミナルまたはセッションに対して、このコマンドを繰り返します。

```
export DDB_LOCAL_TELEMETRY=0
```

すべてのセッションでのプロファイルのテレメトリの無効化

オペレーティングシステムで DynamoDB local を実行している場合は、以下のコマンドを実行してすべてのセッションのテレメトリを無効にします。

Linux でテレメトリを無効にするには

1. 以下を実行します:

   ```
   echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
   ```

2. 以下を実行します:

   ```
   source ~/.profile
   ```

macOS でテレメトリを無効にするには

1. 以下を実行します:

   ```
   echo "export DDB_LOCAL_TELEMETRY=0" >>~/.profile
   ```

2. 以下を実行します:

   ```
   source ~/.profile
   ```

Windows でテレメトリを無効にするには

1. 以下を実行します:

```
setx DDB_LOCAL_TELEMETRY 0
```

2. 以下を実行します:

```
refreshenv
```

Maven プロジェクトに埋め込まれた DynamoDB Local を使用したテレメトリの無効化

Maven プロジェクトに埋め込まれた DynamoDB Local を使用して、テレメトリを無効化することができます。

```
boolean disableTelemetry = true;
// AWS SDK v1
 AmazonDynamoDB amazonDynamoDB =
 DynamoDBEmbedded.create(disableTelemetry).amazonDynamoDB();

// AWS SDK v2
DynamoDbClient ddbClientSDKv2Local =
 DynamoDBEmbedded.create(disableTelemetry).dynamoDbClient();
```

収集される情報のタイプ

- 使用状況の情報 — サーバーの起動/停止、呼び出された API やオペレーションなどの一般的なテレメトリ。
- システムおよび環境の情報 - Java のバージョン、オペレーティングシステム (Windows、Linux、または macOS)、DynamoDB local が実行する環境 (スタンドアロン JAR、Docker コンテナ、Maven 依存関係など)、および使用状況属性のハッシュ値。

詳細情報

DynamoDB local で収集されるテレメトリデータは、AWS データプライバシーポリシーに準拠します。詳細については次を参照してください:

- AWS サービス利用規約
- データプライバシーに関するよくある質問

ステップ 1: DynamoDB にテーブルを作成する

このステップでは、Amazon DynamoDB で `Music` テーブルを作成していきます。このテーブルには次の詳細があります。

- パーティションキー – `Artist`
- ソートキー – `SongTitle`

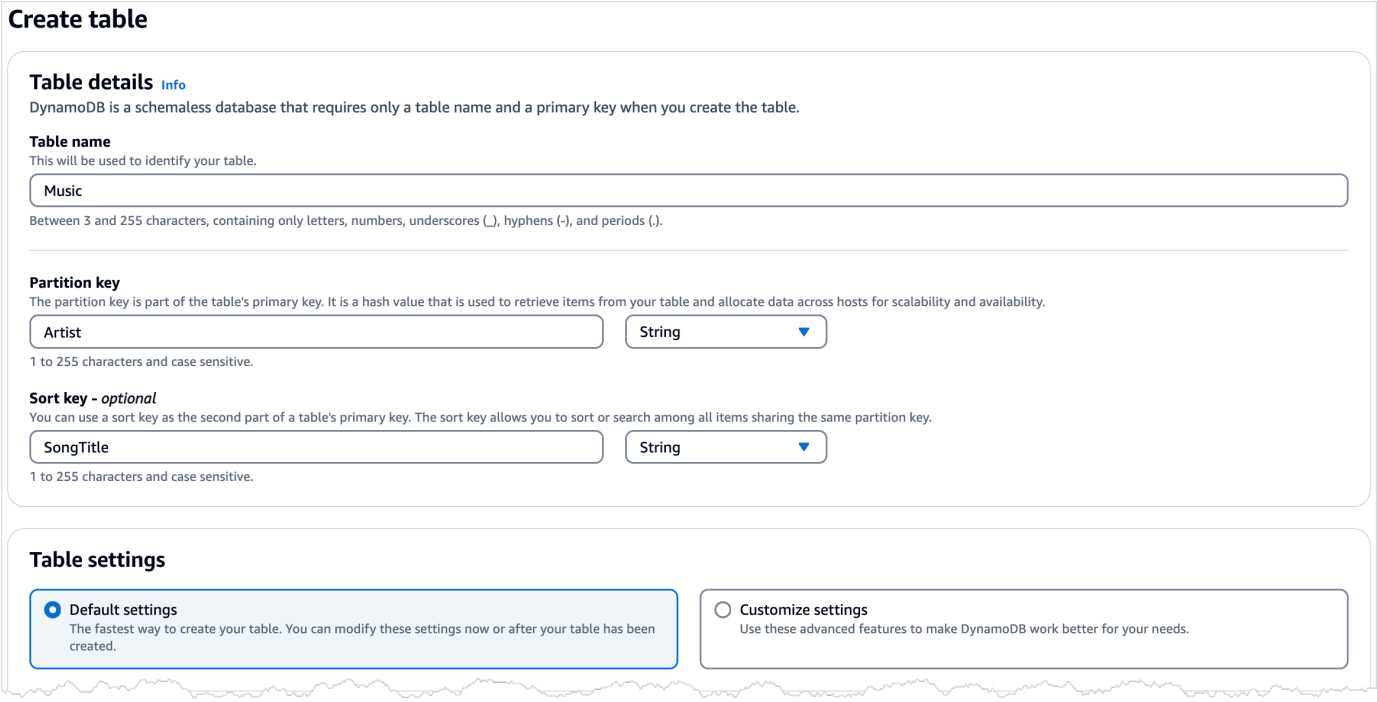
テーブルオペレーションの詳細については、「DynamoDB でのテーブルとデータの操作」を参照してください。

> **Note**
>
> 開始する前に、必ず「前提条件」のステップに従ってください。

AWS マネジメントコンソール

DynamoDB コンソールを使用して新しい `Music` テーブルを作成するには

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. [テーブルの作成] を選択します。
4. [テーブルの詳細] を次のように入力します。
   a. [テーブル名] に **`Music`** と入力します。
   b. [パーティションキー] に **`Artist`** と入力します。
   c. [ソートキー] に **`SongTitle`** と入力します。
5. [テーブル設定] は、[デフォルト設定] を選択したままにします。
6. [テーブルを作成] を選択してテーブルを作成します。



7. テーブルのステータスが [ACTIVE] になったら、次の手順を実行して、テーブルの [DynamoDB のポイントインタイムバックアップ] を有効にすることをお勧めします。

   a. テーブル名を選択してテーブルを開きます。

   b. [バックアップ] を選択します。

   c. [ポイントインタイムリカバリ (PITR)] セクションで、[編集] を選択します。

   d. [ポイントインタイムリカバリ設定の編集] ページで、[ポイントタイムリカバリの有効化] を選択します。

   e. [Save changes] (変更の保存) をクリックします。

AWS CLI

次の AWS CLI の例では、`Music` を使用して新しい `create-table` テーブルを作成します。

Linux

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
```

```
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

Windows CMD

```
aws dynamodb create-table ^
    --table-name Music ^
    --attribute-definitions ^
        AttributeName=Artist,AttributeType=S ^
        AttributeName=SongTitle,AttributeType=S ^
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE ^
    --billing-mode PAY_PER_REQUEST ^
    --table-class STANDARD
```

`create-table` を使用すると、次のサンプル結果が返されます。

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "Music",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
```

```
        "CreationDateTime": "2023-03-29T12:11:43.379000-04:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:111122223333:table/Music",
        "TableId": "60abf404-1839-4917-a89b-a8b0ab2a1b87",
        "TableClassSummary": {
            "TableClass": "STANDARD"
        }
    }
}
}
```

TableStatus フィールドの値は CREATING に設定されていることに注意してください。

DynamoDB が Music テーブルの作成を終了したことを確認するには、describe-table コマンドを使用します。

Linux

```
aws dynamodb describe-table --table-name Music | grep TableStatus
```

Windows CMD

```
aws dynamodb describe-table --table-name Music | findstr TableStatus
```

このコマンドは、次の結果を返します。DynamoDB がテーブルの作成を終了すると、TableStatus フィールドの値が ACTIVE に設定されます。

```
"TableStatus": "ACTIVE",
```

テーブルのステータスが ACTIVE になったら、以下のコマンドを実行して、テーブルで DynamoDB のポイントインタイムバックアップ を有効にするのがベストプラクティスと考えられています。

Linux

```

```

```
aws dynamodb update-continuous-backups \
    --table-name Music \
    --point-in-time-recovery-specification \
        PointInTimeRecoveryEnabled=true
```

Windows CMD

```
aws dynamodb update-continuous-backups --table-name Music --point-in-time-recovery-
specification PointInTimeRecoveryEnabled=true
```

このコマンドは、次の結果を返します。

```
{
    "ContinuousBackupsDescription": {
        "ContinuousBackupsStatus": "ENABLED",
        "PointInTimeRecoveryDescription": {
            "PointInTimeRecoveryStatus": "ENABLED",
            "EarliestRestorableDateTime": "2023-03-29T12:18:19-04:00",
            "LatestRestorableDateTime": "2023-03-29T12:18:19-04:00"
        }
    }
}
```

> **Note**
>
> ポイントインタイムリカバリによる連続バックアップを有効にすると、コストがかかります。料金の詳細については、「Amazon DynamoDB の料金表」を参照してください。

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルを作成する方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
 CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
```

```
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };

            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
```

```
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
 table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
 {tableName}. {ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「CreateTable」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
```

```
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table with on-demand billing."
    echo " -n table_name  -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
 their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
 types."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:a:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      a) attribute_definitions="${OPTARG}" ;;
      k) key_schema="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
```

```
    return 1
  fi

  if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
 -a parameter."
    usage
    return 1
  fi

  if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
 parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
#############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
```

```
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「CreateTable」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Create an Amazon DynamoDB table.
/*!
  \sa createTable()
  \param tableName: Name for the DynamoDB table.
  \param primaryKey: Primary key for the DynamoDB table.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,
                                   const Aws::String &primaryKey,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
```

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Creating table " << tableName <<
              " with a simple primary key: \"" << primaryKey << "\"." <<
 std::endl;

    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition hashKey;
    hashKey.SetAttributeName(primaryKey);
    hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);
    request.AddAttributeDefinitions(hashKey);

    Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;
    keySchemaElement.WithAttributeName(primaryKey).WithKeyType(
            Aws::DynamoDB::Model::KeyType::HASH);
    request.AddKeySchema(keySchemaElement);

    Aws::DynamoDB::Model::ProvisionedThroughput throughput;
    throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);
    request.SetProvisionedThroughput(throughput);
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::CreateTableOutcome &outcome =
 dynamoClient.CreateTable(
              request);
    if (outcome.IsSuccess()) {
        std::cout << "Table \""
                  << outcome.GetResult().GetTableDescription().GetTableName() <<
                  " created!" << std::endl;
    }
    else {
        std::cerr << "Failed to create table: " <<
 outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「CreateTable」を参照してください。

CLI

AWS CLI

例 1: タグ付きのテーブルを作成するには

次の create-table の例では、指定された属性とキースキーマを使用して、MusicCollection という名前のテーブルを作成します。このテーブルはプロビジョニングされたスループットを使用し、保存時にはデフォルトの AWS 所有の CMK を使用して暗号化されます。またこのコマンドは、Owner キーと blueTeam 値を使用して、テーブルにタグを適用します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
```

```
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "CREATING",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 2: オンデマンドモードでテーブルを作成するには

次の例は、プロビジョニングされたスループットモードではなく、オンデマンドモードを使用して MusicCollection というテーブルを作成します。これは、ワークロードが予測できないテーブルに役立ちます。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:44:10.807000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 3: テーブルを作成してカスタマーマネージド CMK で暗号化するには

次の例では、MusicCollection という名前のテーブルを作成し、カスタマーマネージド CMK を使用して暗号化します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",
```

```
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "SSEDescription": {
            "Status": "ENABLED",
            "SSEType": "KMS",
            "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-
abcd-1234-a123-ab1234a1b234"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 4: ローカルセカンダリインデックスを持つテーブルを作成するには

次の例では、指定された属性とキースキーマを使用して、AlbumTitleIndex という名前のローカルセカンダリインデックスを持つ MusicCollection という名前のテーブルを作成します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\": \"Artist\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\": \"AlbumTitle\",\"KeyType\":\"RANGE\"}
```

```
                ],
                \"Projection\": {
                    \"ProjectionType\": \"INCLUDE\",
                    \"NonKeyAttributes\": [\"Genre\", \"Year\"]
                }
            }
        ]"
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
```

```
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Genre",
                        "Year"
                    ]
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 5: グローバルセカンダリインデックスを持つテーブルを作成するには

次の例では、GameTitleIndex という名前のグローバルセカンダリインデックスを持つ GameScores という名前のテーブルを作成します。ベーステーブルには、パーティションキー UserId とソートキー GameTitle があり、特定のゲームの個々のユーザーのベストス

コアを効率的に見つけることができます。一方、GSI にはパーティションキー GameTitle とソートキー TopScore があり、特定のゲームの全体的な最高スコアをすばやく見つけることができます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
```

```
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
```

```
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 6: 複数のグローバルセカンダリインデックスを持つテーブルを一度に作成するには

次の例では、2 つのグローバルセカンダリインデックスを持つ GameScores という名前のテーブルを作成します。GSI スキーマはコマンドラインではなくファイルを介して渡されます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes file://gsi.json
```

gsi.json の内容:

```
[
    {
        "IndexName": "GameTitleIndex",
```

```
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "TopScore",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        }
    },
    {
        "IndexName": "GameDateIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "Date",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        }
    }
]
```

出力:

```
{
```

```
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-08-04T16:40:55.524000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
```

```
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            },
            {
                "IndexName": "GameDateIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "Date",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
```

```
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameDateIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 7: ストリームが有効なテーブルを作成するには

次の例では、DynamoDB ストリームを有効にした状態の GameScores という名前のテーブルを作成します。各アイテムの新しいイメージと古いイメージの両方がストリームに書き込まれます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
```

```
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "LatestStreamLabel": "2020-05-27T17:49:34.056",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 8: Keys-Only ストリームが有効なテーブルを作成するには

次の例では、DynamoDB ストリームを有効にした状態の GameScores という名前のテーブルを作成します。変更された項目のキー属性のみがストリームに書き込まれます。

```
aws dynamodb create-table \
    --table-name GameScores \
```

```
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:45:34.140000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
```

```
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "KEYS_ONLY"
        },
        "LatestStreamLabel": "2023-05-25T18:45:34.140",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2023-05-25T18:45:34.140",
        "DeletionProtectionEnabled": false
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB Streams の変更データキャプチャ」を参照してください。

例 9: Standard Infrequent Access クラスでテーブルを作成するには

次の例は、GameScores という名前のテーブルを作成し、Standard-Infrequent Access (DynamoDB Standard-IA) テーブルクラスを割り当てます。このテーブルクラスは、ストレージが主なコストとなるように最適化されています。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
```

```
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "TableClassSummary": {
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "DeletionProtectionEnabled": false
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブルクラス」を参照してください。

例 10: 削除保護を有効にしたテーブルを作成するには

次の例では、GameScores というテーブルを作成し、削除保護を有効にします。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
```

```
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --deletion-protection-enabled
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "DeletionProtectionEnabled": true
    }
```

```
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「削除保護の使用」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「CreateTable」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
  TableName:   aws.String(basics.TableName),
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}
```

- API の詳細については、『AWS SDK for Go API リファレンス』の「CreateTable」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class CreateTable {
```

```
   public static void main(String[] args) {
       final String usage = """

           Usage:
               <tableName> <key>

           Where:
               tableName - The Amazon DynamoDB table to create (for example,
Music3).
               key - The key for the Amazon DynamoDB table (for example,
Artist).
           """;

       if (args.length != 2) {
           System.out.println(usage);
           System.exit(1);
       }

       String tableName = args[0];
       String key = args[1];
       System.out.println("Creating an Amazon DynamoDB table " + tableName + "
with a simple primary key: " + key);
       Region region = Region.US_EAST_1;
       DynamoDbClient ddb = DynamoDbClient.builder()
           .region(region)
           .build();

       String result = createTable(ddb, tableName, key);
       System.out.println("New table is " + result);
       ddb.close();
   }

   public static String createTable(DynamoDbClient ddb, String tableName, String
key) {
       DynamoDbWaiter dbWaiter = ddb.waiter();
       CreateTableRequest request = CreateTableRequest.builder()
           .attributeDefinitions(AttributeDefinition.builder()
               .attributeName(key)
               .attributeType(ScalarAttributeType.S)
               .build())
           .keySchema(KeySchemaElement.builder()
               .attributeName(key)
               .keyType(KeyType.HASH)
               .build())
```

```
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
 scales based on traffic.
            .tableName(tableName)
            .build();

        String newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
 dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            newTable = response.tableDescription().tableName();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「CreateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    BillingMode: "PAY_PER_REQUEST",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「CreateTable」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
```

```
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「CreateTable」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun createNewTable(
    tableNameVal: String,
    key: String,
): String? {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
```

```
            attributeName = key
            keyType = KeyType.Hash
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef)
            keySchema = listOf(keySchemaVal)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        var tableArn: String
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「CreateTable」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成します。

```
        $tableName = "ddb_demo_table_$uuid";
```

```
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

    public function createTable(string $tableName, array $attributes)
    {
        $keySchema = [];
        $attributeDefinitions = [];
        foreach ($attributes as $attribute) {
            if (is_a($attribute, DynamoDBAttribute::class)) {
                $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
                $attributeDefinitions[] =
                    ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
            }
        }

        $this->dynamoDbClient->createTable([
            'TableName' => $tableName,
            'KeySchema' => $keySchema,
            'AttributeDefinitions' => $attributeDefinitions,
            'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「CreateTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ「Thread」という名前のテーブルを作成します。テーブル

の作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

例 2: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ Thread という名前のテーブルを作成します。ローカルセカンダリインデックスも定義されます。ローカルセカンダリインデックスのキーは、テーブルのプライマリハッシュキー (ForumName) から自動的に設定されます。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
```

```
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

例 3: この例では、単一のパイプラインを使用して、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキー、およびローカルセカンダリインデックスを持つ「Thread」という名前のテーブルを作成する方法を示します。TableSchema がパイプラインまたは -Schema パラメータから提供されない場合、Add-DDBKeySchema と Add-DDBIndexSchema によって新しい TableSchema オブジェクトが作成されます。

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「CreateTable」を参照してください。

Tools for PowerShell V5

例 1: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ「Thread」という名前のテーブルを作成します。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

例 2: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ Thread という名前のテーブルを作成します。ローカルセカンダリインデックスも定義されます。ローカルセカンダリインデックスのキーは、テーブルのプライマリハッシュキー (ForumName) から自動的に設定されます。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
```

```
LocalSecondaryIndexes  : {LastPostIndex}
```

例 3: この例では、単一のパイプラインを使用して、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキー、およびローカルセカンダリインデックスを持つ「Thread」という名前のテーブルを作成する方法を示します。TableSchema がパイプラインまたは -Schema パラメータから提供されない場合、Add-DDBKeySchema と Add-DDBIndexSchema によって新しい TableSchema オブジェクトが作成されます。

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「CreateTable」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

映画データを格納するためのテーブルを作成します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
```

```
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「CreateTable」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Creates an Amazon DynamoDB table that can be used to store movie data.
  # The table uses the release year of the movie as the partition key and the
  # title as the sort key.
  #
  # @param table_name [String] The name of the table to create.
  # @return [Aws::DynamoDB::Table] The newly created table.
  def create_table(table_name)
    @table = @dynamo_resource.create_table(
      table_name: table_name,
      key_schema: [
        { attribute_name: 'year', key_type: 'HASH' }, # Partition key
        { attribute_name: 'title', key_type: 'RANGE' } # Sort key
      ],
      attribute_definitions: [
        { attribute_name: 'year', attribute_type: 'N' },
        { attribute_name: 'title', attribute_type: 'S' }
      ],
      billing_mode: 'PAY_PER_REQUEST'
    )
    @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
    @table
  rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「CreateTable」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn create_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<CreateTableOutput, Error> {
    let a_name: String = key.into();
    let table_name: String = table.into();

    let ad = AttributeDefinition::builder()
        .attribute_name(&a_name)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .map_err(Error::BuildError)?;

    let ks = KeySchemaElement::builder()
        .attribute_name(&a_name)
        .key_type(KeyType::Hash)
        .build()
        .map_err(Error::BuildError)?;

    let create_table_response = client
        .create_table()
        .table_name(table_name)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await;

    match create_table_response {
        Ok(out) => {
            println!("Added table {} with key {}", table, key);
```

```
            Ok(out)
        }
        Err(e) => {
            eprintln!("Got an error creating table:");
            eprintln!("{}", e);
            Err(Error::unhandled(e))
        }
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「CreateTable」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                           iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                           iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
```

```
        oo_result = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
returning.
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
        " This exception can happen if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
{ lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「CreateTable」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
```

```
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「CreateTable」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

新しいテーブルの作成後、「ステップ 2: DynamoDB テーブルにデータを書き込む」に進みます。

ステップ 2: DynamoDB テーブルにデータを書き込む

このステップでは、ステップ 1: DynamoDB にテーブルを作成する で作成した `Music` テーブルに複数の項目を挿入します。

書き込みオペレーションの詳細については、「項目を書き込みます」を参照してください。

AWS マネジメントコンソール

以下の手順に従い、DynamoDB コンソールを使用して `Music` テーブルにデータを書き込みます。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. [テーブル] ページで、[Music] テーブルを選択します。
4. [Explore table items] (テーブルアイテムの探索) を選択します。
5. [返された項目] セクションで、[項目を作成] を選択します。
6. [項目を作成] ページで、次の操作を行ってテーブルに項目を追加します。
    a. [新規属性を追加] を選択し、[数値] を選択します。
    b. [属性名] に、「**`Awards`**」と入力します。
    c. このプロセスを繰り返して、[String (文字列型)] の **`AlbumTitle`** を作成します。
    d. 項目に以下の値を入力します。
        i. [Artist] に **`No One You Know`** と入力します。
        ii. [SongTitle] に **`Call Me Today`** と入力します。
        iii. [AlbumTitle] に「**`Somewhat Famous`**」と入力します。
        iv. [Awards] に **`1`** と入力します。
7. [項目を作成] を選択します。
8. このプロセスを繰り返して、次の値を持つ別の項目を作成します。
    a. [Artist] に「**`Acme Band`**」と入力します。
    b. [SongTitle] に「**`Happy Day`**」と入力します。
    c. [AlbumTitle] に「**`Songs About Life`**」と入力します。
    d. [Awards] に **`10`** と入力します。
9. これをもう一度実行して、前のステップと同じ Artist を含むが、その他の属性の値が異なる項目を作成します。

a. [Artist] に「**Acme Band**」と入力します。
b. [SongTitle] に「**PartiQL Rocks**」と入力します。
c. [AlbumTitle] に「**Another Album Title**」と入力します。
d. [Awards] に **8** と入力します。

AWS CLI

次の AWS CLI の例では、Music テーブルで複数の新しい項目を作成します。この操作は、DynamoDB API または PartiQL (DynamoDB の SQL 互換クエリ言語) を介して行うことができます。

DynamoDB API

Linux

```
aws dynamodb put-item \
    --table-name Music  \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"},
 "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
    --table-name Music  \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Howdy"},
 "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "2"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"},
 "AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "PartiQL Rocks"},
 "AlbumTitle": {"S": "Another Album Title"}, "Awards": {"N": "8"}}'
```

Windows CMD

```
aws dynamodb put-item ^
    --table-name Music  ^
    --item ^
        "{\"Artist\": {\"S\": \"No One You Know\"}, \"SongTitle\": {\"S\": \"Call
 Me Today\"}, \"AlbumTitle\": {\"S\": \"Somewhat Famous\"}, \"Awards\": {\"N\":
 \"1\"}}"

aws dynamodb put-item ^
    --table-name Music  ^
    --item ^
        "{\"Artist\": {\"S\": \"No One You Know\"}, \"SongTitle\": {\"S\": \"Howdy
\"}, \"AlbumTitle\": {\"S\": \"Somewhat Famous\"}, \"Awards\": {\"N\": \"2\"}}"

aws dynamodb put-item ^
    --table-name Music ^
    --item ^
        "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day\"},
 \"AlbumTitle\": {\"S\": \"Songs About Life\"}, \"Awards\": {\"N\": \"10\"}}"

aws dynamodb put-item ^
    --table-name Music ^
    --item ^
        "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"PartiQL Rocks
\"}, \"AlbumTitle\": {\"S\": \"Another Album Title\"}, \"Awards\": {\"N\": \"8\"}}"
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'No One You Know','SongTitle':'Call Me Today',
 'AlbumTitle':'Somewhat Famous', 'Awards':'1'}"

aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'No One You Know','SongTitle':'Howdy',
 'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"

aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'Acme Band','SongTitle':'Happy Day', 'AlbumTitle':'Songs
 About Life', 'Awards':'10'}"
```

```
aws dynamodb execute-statement --statement "INSERT INTO Music  \
                VALUE  \
                {'Artist':'Acme Band','SongTitle':'PartiQL Rocks',
 'AlbumTitle':'Another Album Title', 'Awards':'8'}"
```

Windows CMD

```
aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No
 One You Know','SongTitle':'Call Me Today', 'AlbumTitle':'Somewhat Famous',
 'Awards':'1'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'No
 One You Know','SongTitle':'Howdy', 'AlbumTitle':'Somewhat Famous', 'Awards':'2'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme
 Band','SongTitle':'Happy Day', 'AlbumTitle':'Songs About Life', 'Awards':'10'}"

aws dynamodb execute-statement --statement "INSERT INTO Music VALUE {'Artist':'Acme
 Band','SongTitle':'PartiQL Rocks', 'AlbumTitle':'Another Album Title',
 'Awards':'8'}"
```

PartiQL を使用したデータの書き込みについては、「PartiQL 挿入ステートメント」を参照してください。

DynamoDB でサポートされるデータ型の詳細については、「データ型」を参照してください。

JSON で DynamoDB のデータ型を表す方法については、「属性値」を参照してください。

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルに項目を書き込む方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };

            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
```

```
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「PutItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
###############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
#
#  Returns:
```

```
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }

  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
```

```
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
  fi

  return 0

}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
```

```
# This function outputs everything sent to it to STDERR (standard error output).
#########################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#########################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#########################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Put an item in an Amazon DynamoDB table.
/*!
  \sa putItem()
  \param tableName: The table name.
  \param artistKey: The artist key. This is the partition key for the table.
  \param artistValue: The artist value.
  \param albumTitleKey: The album title key.
  \param albumTitleValue: The album title value.
  \param awardsKey: The awards key.
  \param awardsValue: The awards value.
  \param songTitleKey: The song title key.
  \param songTitleValue: The song title value.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                               const Aws::String &artistKey,
                               const Aws::String &artistValue,
                               const Aws::String &albumTitleKey,
                               const Aws::String &albumTitleValue,
                               const Aws::String &awardsKey,
                               const Aws::String &awardsValue,
                               const Aws::String &songTitleKey,
                               const Aws::String &songTitleValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(tableName);

    putItemRequest.AddItem(artistKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            artistValue)); // This is the hash key.
    putItemRequest.AddItem(albumTitleKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            albumTitleValue));
    putItemRequest.AddItem(awardsKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(awardsValue));
    putItemRequest.AddItem(songTitleKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(songTitleValue));

    const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
            putItemRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully added Item!" << std::endl;
    }
    else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {
```

```
    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスのの「PutItem」を参照してください。

CLI

AWS CLI

例 1: テーブルに項目を追加するには

次の put-item の例は、MusicCollection テーブルに新しい項目を追加します。

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

item.json の内容:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Greatest Hits"}
}
```

出力:

```
{
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

例 2: テーブル内の項目を条件付きで上書きするには

次の put-item の例は、MusicCollection テーブル内の既存の項目に Greatest Hits の値を持つ AlbumTitle 属性がある場合にのみ、その項目を上書きします。このコマンドは、その項目の以前の値を返します。

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --condition-expression "#A = :A" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

item.json の内容:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Somewhat Famous"}
}
```

names.json の内容:

```
{
    "#A": "AlbumTitle"
}
```

values.json の内容:

```
{
    ":A": {"S": "Greatest Hits"}
}
```

出力:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Greatest Hits"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
```

```
        }
    }
}
```

キーが存在する場合は、次のような出力が表示されます。

```
A client error (ConditionalCheckFailedException) occurred when calling the
 PutItem operation: The conditional request failed.
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
```

```
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「PutItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルに項目を配置します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
 <awards> <awardsval> <Songtitle> <songtitleval>

                Where:
```

```
                    tableName - The Amazon DynamoDB table in which an item is
placed (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example,
Happy Day).
                **Warning** This program will  place an item that you specify
into a table!
                """;

        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }
```

```
    public static void putItemInTable(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle,
 AttributeValue.builder().s(songTitleVal).build());
        itemValues.put(albumTitle,
 AttributeValue.builder().s(albumTitleValue).build());
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The
 request id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「PutItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「PutCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルに項目を配置します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

DynamoDB ドキュメントクライアントを使用して、テーブルに項目を配置します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun putItemInTable(
    tableNameVal: String,
    key: String,
    keyVal: String,
    albumTitle: String,
```

```
    albumTitleValue: String,
    awards: String,
    awardVal: String,
    songTitle: String,
    songTitleVal: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()

    // Add all content to the table.
    itemValues[key] = AttributeValue.S(keyVal)
    itemValues[songTitle] = AttributeValue.S(songTitleVal)
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)
    itemValues[awards] = AttributeValue.S(awardVal)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println(" A new item was placed into $tableNameVal.")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「PutItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        echo "What's the name of the last movie you watched?\n";
```

```
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);

    public function putItem(array $array)
    {
        $this->dynamoDbClient->putItem($array);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「PutItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 新しい項目を作成する、または既存の項目を新しい項目で置き換えます。

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
```

```
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「PutItem」を参照してください。

Tools for PowerShell V5

例 1: 新しい項目を作成する、または既存の項目を新しい項目で置き換えます。

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「PutItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
```

```
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def add_movie(self, title, year, plot, rating):
        """
        Adds a movie to the table.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :param plot: The plot summary of the movie.
        :param rating: The quality rating of the movie.
        """
        try:
            self.table.put_item(
                Item={
                    "year": year,
                    "title": title,
                    "info": {"plot": plot, "rating": Decimal(str(rating))},
                }
            )
        except ClientError as err:
            logger.error(
                "Couldn't add movie %s to table %s. Here's why: %s: %s",
```

```
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「PutItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Adds a movie to the table.
  #
  # @param movie [Hash] The title, year, plot, and rating of the movie.
  def add_item(movie)
    @table.put_item(
      item: {
        'year' => movie[:year],
        'title' => movie[:title],
        'info' => { 'plot' => movie[:plot], 'rating' => movie[:rating] }
      }
```

```
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「PutItem」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
 Result<ItemOut, Error> {
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;
```

```
    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    );

    Ok(ItemOut {
        p_type,
        age,
        username,
        first_name,
        last_name,
    })
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「PutItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lo_resp) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = it_item ).
```

```
        MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「PutItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()
```

```
            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
```

```
        item["info"] = .m(details)

        return item
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「PutItem」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

テーブルにデータを書き込んだら、「ステップ 3: DynamoDB テーブルからデータを読み込む」に進みます。

ステップ 3: DynamoDB テーブルからデータを読み込む

このステップでは、「ステップ 2: DynamoDB テーブルにデータを書き込む」で作成した項目の 1 つを読み込みます。DynamoDB コンソールまたは AWS CLI を使用して、`Artist` および `SongTitle` を指定しながら `Music` テーブルの項目を読み込むことができます。

DynamoDB の読み込みオペレーションの詳細については、「項目の読み込み」を参照してください。

AWS マネジメントコンソール

以下の手順に従い DynamoDB コンソールを使用して、`Music` テーブルからデータを読み込みます。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. [テーブル] ページで、[Music] テーブルを選択します。
4. [テーブルアイテムの探索] を選択します。
5. [返された項目] セクションで、テーブルに保存されている項目のリストを、`Artist` と `SongTitle` でソートして表示します。リストの最初の項目は、[Artist] として [Acme Band]、[SongTitle] として [PartiQL Rocks] になります。

AWS CLI

次の AWS CLI の例では、Music から項目を読み込みます。この操作は、DynamoDB API または PartiQL (DynamoDB の SQL 互換クエリ言語) を介して行うことができます。

DynamoDB API

> **Note**
>
> DynamoDB では、デフォルトで整合性のある読み込みを行います。以下では、consistent-read パラメータを使用して強力な整合性のある読み込みを示します。

Linux

```
aws dynamodb get-item --consistent-read \
    --table-name Music \
    --key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}'
```

Windows CMD

```
aws dynamodb get-item --consistent-read ^
    --table-name Music ^
    --key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day
\"}}"
```

get-item を使用すると、次のサンプル結果が返されます。

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "Awards": {
            "S": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
```

```
        "SongTitle": {
            "S": "Happy Day"
        }
    }
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
WHERE Artist='Acme Band' AND SongTitle='Happy Day'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme
 Band' AND SongTitle='Happy Day'"
```

PartiQL `Select` ステートメントを使用すると、次のサンプル結果が返されます。

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Songs About Life"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        }
    ]
}
```

PartiQL を使用したデータの読み込みについては、「PartiQL 選択ステートメント」を参照してください。

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルから項目を読み込む方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new GetItemRequest
            {
                Key = key,
                TableName = tableName,
            };
```

```
            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「GetItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
```

```
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
  }
  query=""
  while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
```

```
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
  fi

  if [[ -n "$query" ]]; then
    response=$(aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text \
      --query "$query")
  else
    response=$(
      aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"$keys" \
        --output text
    )
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
  fi

  if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
query inserts on some strings.
  else
    echo "$response"
```

```
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
################################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
```

```
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Get an item from an Amazon DynamoDB table.
/*!
  \sa getItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,
                               const Aws::String &partitionKey,
                               const Aws::String &partitionValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
    Aws::DynamoDB::Model::GetItemRequest request;

    // Set up the request.
    request.SetTableName(tableName);
    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));

    // Retrieve the item's fields and values.
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =
 dynamoClient.GetItem(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved fields/values.
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =
 outcome.GetResult().GetItem();
        if (!item.empty()) {
            // Output each retrieved field and its value.
            for (const auto &i: item)
                std::cout << "Values: " << i.first << ": " << i.second.GetS()
                          << std::endl;
        }
        else {
            std::cout << "No item found with the key " << partitionKey <<
 std::endl;
        }
    }
    else {
        std::cerr << "Failed to get item: " << outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「GetItem」を参照してください。

CLI

AWS CLI

例 1: テーブル内の項目を読み込むには

次の get-item の例は、MusicCollection テーブルから項目を取得します。テーブルにはハッシュおよび範囲プライマリキー (ArtistおよびSongTitle) があるため、これらの属性の両方を指定する必要があります。このコマンドは、オペレーションによって消費される読み込み容量に関する情報も要求します。

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-consumed-capacity TOTAL
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

出力:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

例 2: 整合性のある読み込みを使用して項目を読み込むには

次の例では、強力な整合性のある読み込みを使用して MusicCollection テーブルから項目を読み込みます。

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --consistent-read \
    --return-consumed-capacity TOTAL
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

出力:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

例 3: 項目の特定の属性を取得するには

次の例は、射影式を使用して目的のアイテムの 3 つの属性のみを取得します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "102"}}' \
    --projection-expression "#T, #C, #P" \
    --expression-attribute-names file://names.json
```

names.json の内容:

```
{
    "#T": "Title",
    "#C": "ProductCategory",
    "#P": "Price"
}
```

出力:

```
{
    "Item": {
        "Price": {
            "N": "20"
        },
        "Title": {
            "S": "Book 102 Title"
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
```

```
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
```

```
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「GetItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用して、テーブルから項目を取得します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal>

                Where:
                    tableName - The Amazon DynamoDB table from which an item is
 retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
 tableName);
        Region region = Region.US_EAST_1;
```

```
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
 String key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        GetItemRequest request = GetItemRequest.builder()
                .key(keyToGet)
                .tableName(tableName)
                .build();

        try {
            // If there is no matching item, GetItem does not return any data.
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();
            if (returnedItem.isEmpty())
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「GetItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「GetCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「GetItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルから項目を取得します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

DynamoDB ドキュメントクライアントを使用して、テーブルから項目を取得します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「GetItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun getSpecificItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
```

```
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「GetItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";

    public function getItemByKey(string $tableName, array $key)
    {
        return $this->dynamoDbClient->getItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「GetItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目を返します。

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「GetItem」を参照してください。

Tools for PowerShell V5

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目を返します。

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

出力する。

```
Name                            Value
----                            -----
Genre                           Country
SongTitle                       Somewhere Down The Road
Price                           1.94
Artist                          No One You Know
CriticRating                    9
AlbumTitle                      Somewhat Famous
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「GetItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
```

```
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def get_movie(self, title, year):
    """
    Gets movie data from the table for a specific movie.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :return: The data about the requested movie.
    """
    try:
        response = self.table.get_item(Key={"year": year, "title": title})
    except ClientError as err:
        logger.error(
            "Couldn't get movie %s from table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Item"]
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「GetItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Gets movie data from the table for a specific movie.
  #
  # @param title [String] The title of the movie.
  # @param year [Integer] The release year of the movie.
  # @return [Hash] The data about the requested movie.
  def get_item(title, year)
    @table.get_item(key: { 'year' => year, 'title' => title })
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「GetItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_item = lo_dyn->getitem(
          iv_tablename                = iv_table_name
          it_key                      = it_key ).
        DATA(lt_attr) = oo_item->get_item( ).
        DATA(lo_title) = lt_attr[ key = 'title' ]-value.
        DATA(lo_year) = lt_attr[ key = 'year' ]-value.
        DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( )
          && 'Movie year is: ' && lo_year->get_n( )
          && 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「GetItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB
```

```
    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「GetItem」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

テーブルのデータを更新するには、「ステップ 4: DynamoDB テーブルのデータを更新する」に進みます。

ステップ 4: DynamoDB テーブルのデータを更新する

このステップでは、「ステップ 2: DynamoDB テーブルにデータを書き込む」で作成した項目を更新できます。DynamoDB コンソールまたは AWS CLI を使用して、`Artist`、`SongTitle` および更新された `AlbumTitle` を指定して `Music` テーブルの項目の `AlbumTitle` を更新できます。

書き込みオペレーションの詳細については、「項目を書き込みます」を参照してください。

AWS マネジメントコンソール

DynamoDB コンソールを使用して、`Music` テーブルのデータを更新できます。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. テーブルのリストから [Music] テーブルを選択します。
4. [テーブルアイテムの探索] を選択します。
5. [返された項目] の項目行 ([Acme Band] という [Artist] と [Happy Day] という [SongTitle]) で、次の操作を行います。

    a. [Songs About Life] という名前の [AlbumTitle] にカーソルを置きます。
    b. [編集] アイコンを選択します。
    c. [文字列を編集] ポップアップウィンドウに、「**`Songs of Twilight`**」と入力します。
    d. [保存] を選択します。

> **Tip**
>
> または、項目を更新するには、[返された項目] セクションで次の操作を行います。
>
> 1. [Artist] として [Acme Band] および [SongTitle] として [Happy Day] を示す項目行を選択します。
> 2. [アクション] ドロップダウンリストで、[項目の編集] を選択します。

3. [AlbumTitle] に「**Songs of Twilight**」と入力します。
4. [保存して閉じる] を選択します。

AWS CLI

次の AWS CLI の例では、Music テーブルの項目を更新します。この操作は、DynamoDB API または PartiQL (DynamoDB の SQL 互換クエリ言語) を介して行うことができます。

DynamoDB API

Linux

```
aws dynamodb update-item \
    --table-name Music \
    --key '{ "Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}' \
    --update-expression "SET AlbumTitle = :newval" \
    --expression-attribute-values '{":newval":{"S":"Updated Album Title"}}' \
    --return-values ALL_NEW
```

Windows CMD

```
aws dynamodb update-item ^
    --table-name Music ^
    --key "{\"Artist\": {\"S\": \"Acme Band\"}, \"SongTitle\": {\"S\": \"Happy Day
\"}}" ^
    --update-expression "SET AlbumTitle = :newval" ^
    --expression-attribute-values "{\":newval\":{\"S\":\"Updated Album Title\"}}" ^
    --return-values ALL_NEW
```

update-item を使用すると、return-values ALL_NEW が指定されたために次のサンプル結果が返されます。

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Updated Album Title"
        },
        "Awards": {
            "S": "10"
        },
```

```
        "Artist": {
            "S": "Acme Band"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    }
}
```

PartiQL for DynamoDB

Linux

```
aws dynamodb execute-statement --statement "UPDATE Music  \
    SET AlbumTitle='Updated Album Title'  \
    WHERE Artist='Acme Band' AND SongTitle='Happy Day' \
    RETURNING ALL NEW *"
```

Windows CMD

```
aws dynamodb execute-statement --statement "UPDATE Music SET AlbumTitle='Updated
 Album Title' WHERE Artist='Acme Band' AND SongTitle='Happy Day' RETURNING ALL NEW
 *"
```

Update ステートメントを使用すると、RETURNING ALL NEW * が指定されたために次のサンプル結果が返されます。

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
```

```
        }
    ]
}
```

PartiQL を使用したデータの更新については、「PartiQL 更新ステートメント」を参照してください。

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルで項目を更新する方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
```

```
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };

            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} or item was not found.
{ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
```

```
            Console.WriteLine($"An error occurred while updating item.
 {ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「UpdateItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
###############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
```

```
 local option OPTARG # Required to use getopts command in a function.

 ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_update_item"
   echo "Update an item in a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k keys  -- Path to json file containing the keys that identify the
item to update."
   echo " -e update expression  -- An expression that defines one or more
attributes to be updated."
   echo " -v values  -- Path to json file containing the update values."
   echo ""
 }

 while getopts "n:k:e:v:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) keys="${OPTARG}" ;;
     e) update_expression="${OPTARG}" ;;
     v) values="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$keys" ]]; then
   errecho "ERROR: You must provide a keys json file path the -k parameter."
```

```
    usage
    return 1
  fi
  if [[ -z "$update_expression" ]]; then
    errecho "ERROR: You must provide an update expression with the -e parameter."
    usage
    return 1
  fi

  if [[ -z "$values" ]]; then
    errecho "ERROR: You must provide a values json file path the -v parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho "    update_expression:   $update_expression"
  iecho "    values:   $values"

  response=$(aws dynamodb update-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --update-expression "$update_expression" \
    --expression-attribute-values file://"$values")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
```

```
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
```

```
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Update an Amazon DynamoDB table item.
/*!
  \sa updateItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param attributeKey: The key for the attribute to be updated.
  \param attributeValue: The value for the attribute to be updated.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 *  The example code only sets/updates an attribute value. It processes
```

```
 *  the attribute value as a string, even if the value could be interpreted
 *  as a number. Also, the example code does not remove an existing attribute
 *  from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &attributeKey,
                                  const Aws::String &attributeValue,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // *** Define UpdateItem request arguments.
    // Define TableName argument.
    Aws::DynamoDB::Model::UpdateItemRequest request;
    request.SetTableName(tableName);

    // Define KeyName argument.
    Aws::DynamoDB::Model::AttributeValue attribValue;
    attribValue.SetS(partitionValue);
    request.AddKey(partitionKey, attribValue);

    // Construct the SET update expression argument.
    Aws::String update_expression("SET #a = :valueA");
    request.SetUpdateExpression(update_expression);

    // Construct attribute name argument.
    Aws::Map<Aws::String, Aws::String> expressionAttributeNames;
    expressionAttributeNames["#a"] = attributeKey;
    request.SetExpressionAttributeNames(expressionAttributeNames);

    // Construct attribute value argument.
    Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;
    attributeUpdatedValue.SetS(attributeValue);
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 expressionAttributeValues;
    expressionAttributeValues[":valueA"] = attributeUpdatedValue;
    request.SetExpressionAttributeValues(expressionAttributeValues);

    // Update the item.
    const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =
 dynamoClient.UpdateItem(
```

```
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Item was updated" << std::endl;
    } else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
```

```
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「UpdateItem」を参照してください。

CLI

AWS CLI

例 1: テーブル内の項目を更新するには

次の update-item の例では、MusicCollection テーブルの項目を更新します。新しい属性 (Year) を追加して、AlbumTitle 属性を更新します。更新後に表示される項目内の属性はすべて、レスポンスで返されます。

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --return-values ALL_NEW \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
```

```
}
```

expression-attribute-names.json の内容:

```
{
    "#Y":"Year", "#AT":"AlbumTitle"
}
```

expression-attribute-values.json の内容:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

出力:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Louder Than Ever"
        },
        "Awards": {
            "N": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
        "Year": {
            "N": "2015"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 3.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
```

```
                "S": "Acme Band"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

例 2: 項目を条件付きで更新するには

次の例は、既存の項目に Year 属性がない場合にのみ、MusicCollection テーブル内の項目を更新します。

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --condition-expression "attribute_not_exists(#Y)"
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

expression-attribute-names.json の内容:

```
{
    "#Y":"Year",
    "#AT":"AlbumTitle"
}
```

expression-attribute-values.json の内容:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

項目にすでに Year 属性がある場合、DynamoDB は次の出力を返します。

```
An error occurred (ConditionalCheckFailedException) when calling the UpdateItem
 operation: The conditional request failed
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
   }
  }
```

```
 }
 return attributeMap, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
```

```
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「UpdateItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルで項目を更新します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
```

```
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
 * practice to use the
 * Enhanced Client, See the EnhancedModifyItem example.
 */
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <name> <updateVal>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music3).
                    key - The name of the key in the table (for example, Artist).
                    keyVal - The value of the key (for example, Famous Band).
                    name - The name of the column where the value is updated (for
 example, Awards).
                    updateVal - The value used to update an item (for example,
 14).
                 Example:
                    UpdateItem Music3 Artist Famous Band Awards 14
                """;

        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
```

```
    }

    public static void updateTableItem(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String name,
            String updateVal) {

        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put(name, AttributeValueUpdate.builder()
                .value(AttributeValue.builder().s(updateVal).build())
                .action(AttributeAction.PUT)
                .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(itemKey)
                .attributeUpdates(updatedValues)
                .build();

        try {
            ddb.updateItem(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("The Amazon DynamoDB table was updated!");
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「UpdateCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun updateTableItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
    name: String,
    updateVal: String,
) {
    val itemKey = mutableMapOf<String, AttributeValue>()
    itemKey[keyName] = AttributeValue.S(keyVal)

    val updatedValues = mutableMapOf<String, AttributeValueUpdate>()
    updatedValues[name] =
        AttributeValueUpdate {
            value = AttributeValue.S(updateVal)
            action = AttributeAction.Put
        }

    val request =
        UpdateItemRequest {
            tableName = tableNameVal
            key = itemKey
            attributeUpdates = updatedValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.updateItem(request)
        println("Item in $tableNameVal was updated")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「UpdateItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
 $rating);

    public function updateItemAttributeByKey(
        string $tableName,
        array $key,
        string $attributeName,
        string $attributeType,
        string $newValue
    ) {
        $this->dynamoDbClient->updateItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
            'UpdateExpression' => "set #NV=:NV",
            'ExpressionAttributeNames' => [
                '#NV' => $attributeName,
            ],
            'ExpressionAttributeValues' => [
                ':NV' => [
                    $attributeType => $newValue
                ]
```

```
            ],
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「UpdateItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目のジャンル属性を「Rap」に設定します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Rap
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「UpdateItem」を参照してください。

Tools for PowerShell V5

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目のジャンル属性を「Rap」に設定します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Rap
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

更新式を使用して項目を更新します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
```

```
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
 plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

算術演算を含む更新式を使用して、項目を更新します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
 operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
 movie.
        :return: The updated rating.
        """
```

```
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

特定の条件を満たす場合にのみ、項目を更新します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def remove_actors(self, title, year, actor_threshold):
        """
        Removes an actor from a movie, but only when the number of actors is
 greater
        than a specified threshold. If the movie does not list more than the
 threshold,
        no actors are removed.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param actor_threshold: The threshold of actors to check.
        :return: The movie data after the update.
        """
        try:
            response = self.table.update_item(
```

```
                Key={"year": year, "title": title},
                UpdateExpression="remove info.actors[0]",
                ConditionExpression="size(info.actors) > :num",
                ExpressionAttributeValues={":num": actor_threshold},
                ReturnValues="ALL_NEW",
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"ConditionalCheckFailedException":
                logger.warning(
                    "Didn't update %s because it has fewer than %s actors.",
                    title,
                    actor_threshold + 1,
                )
            else:
                logger.error(
                    "Couldn't update movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Attributes"]
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
```

```
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)
    response = @table.update_item(
      key: { 'year' => movie[:year], 'title' => movie[:title] },
      update_expression: 'set info.rating=:r',
      expression_attribute_values: { ':r' => movie[:rating] },
      return_values: 'UPDATED_NEW'
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't update movie #{movie[:title]} (#{movie[:year]}) in table
#{@table.name}\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.attributes
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「UpdateItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_output = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = it_item_key
          it_attributeupdates = it_attribute_updates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「UpdateItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
```

```
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
[:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
```

```
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「UpdateItem」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

Music テーブルのデータをクエリするには、「ステップ 5: DynamoDB テーブル内のデータにクエリを実行」に進みます。

ステップ 5: DynamoDB テーブル内のデータにクエリを実行

このステップでは、Music を指定して the section called “ステップ 2: データを書き込みます” の Artist テーブルに書き込んだデータをクエリします。これにより、パーティションキー Artist に関連付けられているすべての曲が表示されます。

クエリオペレーションの詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

AWS マネジメントコンソール

DynamoDB コンソールを使用して Music テーブルのデータをクエリするには、次のステップに従います。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。

3. テーブルのリストから [Music] テーブルを選択します。
4. [テーブルアイテムの探索] を選択します。
5. [項目のスキャンまたはクエリ] で、[クエリ] が選択されていることを確認します。
6. [パーティションキー] に「**`Acme Band`**」と入力し、[実行] を選択します。

AWS CLI

次の AWS CLI の例では、`Music` テーブルの項目にクエリを実行します。この操作は、DynamoDB API または PartiQL (DynamoDB の SQL 互換クエリ言語) を介して行うことができます。

DynamoDB API

`query` を使用してパーティションキーを指定することで、DynamoDB API を介して項目にクエリを実行できます。

Linux

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression "Artist = :name" \
    --expression-attribute-values  '{":name":{"S":"Acme Band"}}'
```

Windows CMD

```
aws dynamodb query ^
    --table-name Music ^
    --key-condition-expression "Artist = :name" ^
    --expression-attribute-values  "{\":name\":{\"S\":\"Acme Band\"}}"
```

`query` を使用すると、この特定の `Artist` に関連付けられているすべての曲が返されます。

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "N": "10"
```

```
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        },
        {
            "AlbumTitle": {
                "S": "Another Album Title"
            },
            "Awards": {
                "N": "8"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

PartiQL for DynamoDB

`Select` ステートメントを使用してパーティションキーを指定することで、PartiQL を介して項目にクエリを実行できます。

Linux

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
                                            WHERE Artist='Acme Band'"
```

Windows CMD

```
aws dynamodb execute-statement --statement "SELECT * FROM Music WHERE Artist='Acme
 Band'"
```

このように Select ステートメントを使用すると、この特定の Artist に関連付けられているすべての曲が返されます。

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "S": "10"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "Happy Day"
            }
        },
        {
            "AlbumTitle": {
                "S": "Another Album Title"
            },
            "Awards": {
                "S": "8"
            },
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

PartiQL を使用したデータへのクエリの実行については、「PartiQL 選択ステートメント」を参照してください。

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルに対してクエリを実行する方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public async Task<int> QueryMoviesAsync(string tableName, int year)
{
    try
    {
        var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
            .AddHashKey("year", DynamoDBEntryType.Numeric)
            .AddRangeKey("title", DynamoDBEntryType.String)
            .Build();

        var filter = new QueryFilter("year", QueryOperator.Equal, year);

        Console.WriteLine("\nFind movies released in: {year}:");

        var config = new QueryOperationConfig()
        {
            Limit = 10, // 10 items per page.
            Select = SelectValues.SpecificAttributes,
            AttributesToGet = new List<string>
```

```
                {
                    "title",
                    "year",
                },
                ConsistentRead = true,
                Filter = filter,
            };

            // Value used to track how many movies match the
            // supplied criteria.
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while querying movies.
{ex.Message}");
            throw;
        }
```

```
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「Query」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_query() {
  local table_name key_condition_expression attribute_names attribute_values
 projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
```

```
 # Function usage explanation
 ####################################
 function usage() {
   echo "function dynamodb_query"
   echo "Query a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k key_condition_expression -- The key condition expression."
   echo " -a attribute_names -- Path to JSON file containing the attribute
names."
   echo " -v attribute_values -- Path to JSON file containing the attribute
values."
   echo " [-p projection_expression]  -- Optional projection expression."
   echo ""
 }

 while getopts "n:k:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) key_condition_expression="${OPTARG}" ;;
     a) attribute_names="${OPTARG}" ;;
     v) attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$key_condition_expression" ]]; then
   errecho "ERROR: You must provide a key condition expression with the -k
parameter."
```

```
    usage
    return 1
  fi

  if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values")
  else
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
```

```
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「Query」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Perform a query on an Amazon DynamoDB Table and retrieve items.
/*!
  \sa queryItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param projectionExpression: The projections expression, which is ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 * The partition key attribute is searched with the specified value. By default,
 all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
 */
```

```
bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &projectionExpression,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
   Aws::DynamoDB::Model::QueryRequest request;

   request.SetTableName(tableName);

   if (!projectionExpression.empty()) {
       request.SetProjectionExpression(projectionExpression);
   }

   // Set query key condition expression.
   request.SetKeyConditionExpression(partitionKey + "= :valueToMatch");

   // Set Expression AttributeValues.
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
   attributeValues.emplace(":valueToMatch", partitionValue);

   request.SetExpressionAttributeValues(attributeValues);

   bool result = true;

   // "exclusiveStartKey" is used for pagination.
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
   do {
       if (!exclusiveStartKey.empty()) {
           request.SetExclusiveStartKey(exclusiveStartKey);
           exclusiveStartKey.clear();
       }
       // Perform Query operation.
       const Aws::DynamoDB::Model::QueryOutcome &outcome =
dynamoClient.Query(request);
       if (outcome.IsSuccess()) {
           // Reference the retrieved items.
           const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
           if (!items.empty()) {
               std::cout << "Number of items retrieved from Query: " <<
items.size()
```

```
                          << std::endl;
                // Iterate each item and print.
                for (const auto &item: items) {
                    std::cout
                            <<
"*************************************************"
                            << std::endl;
                    // Output each retrieved field and its value.
                    for (const auto &i: item)
                        std::cout << i.first << ": " << i.second.GetS() <<
std::endl;
                }
            }
            else {
                std::cout << "No item found in table: " << tableName <<
std::endl;
            }

            exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
        }
        else {
            std::cerr << "Failed to Query items: " <<
outcome.GetError().GetMessage();
            result = false;
            break;
        }
    } while (!exclusiveStartKey.empty());

    return result;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「Query」を参照してください。

CLI

AWS CLI

例 1: テーブルにクエリを実行するには

次の query の例では、MusicCollection テーブルの項目にクエリを実行します。テーブルにはハッシュおよび範囲プライマリキー (ArtistおよびSongTitle) がありますが、このク

エリではハッシュキー値のみを指定します。「No One You Know」という名前のアーティストの曲タイトルが返されます。

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --return-consumed-capacity TOTAL
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 2: 強力な整合性のある読み込みを使用してテーブルにクエリを実行し、インデックスを降順で走査するには

次の例では、最初の例と同じクエリを実行しますが、結果は逆の順序で返され、強力な整合性のある読み込みが使用されます。

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --consistent-read \
    --no-scan-index-forward \
    --return-consumed-capacity TOTAL
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 3: 特定の結果を除外するには

次の例は、MusicCollection をクエリしますが、AlbumTitle 属性に特定の値を含む結果を除外します。このフィルターは項目が読み込まれた後に適用されるため、ScannedCount または ConsumedCapacity には影響しないことに注意してください。

```
aws dynamodb query \
    --table-name MusicCollection \
    --key-condition-expression "#n1 = :v1" \
    --filter-expression "NOT (#n2 IN (:v2, :v3))" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-consumed-capacity TOTAL
```

values.json の内容:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

names.json の内容:

```
{
    "#n1": "Artist",
    "#n2": "AlbumTitle"
}
```

出力:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
```

```
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 4: 項目数だけを取得するには

次の例は、クエリに一致する項目数を取得しますが、項目自体は取得しません。

```
aws dynamodb query \
    --table-name MusicCollection \
    --select COUNT \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 5: インデックスをクエリするには

次の例は、ローカルセカンダリインデックス AlbumTitleIndex をクエリします。クエリは、ローカルセカンダリインデックスに射影されたベーステーブルのすべての属性を返します。ローカルセカンダリインデックスまたはグローバルセカンダリインデックスをクエリする場合は、table-name パラメータを使用してベーステーブルの名前も指定する必要があることに注意してください。

```
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --select ALL_PROJECTED_ATTRIBUTES \
    --return-consumed-capacity INDEXES
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
```

```
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5,
        "Table": {
            "CapacityUnits": 0.0
        },
        "LocalSecondaryIndexes": {
            "AlbumTitleIndex": {
                "CapacityUnits": 0.5
            }
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「Query」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
```

```
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「Query」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルに対してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedQueryRecords example.
 */
public class Query {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <partitionKeyName> <partitionKeyVal>

                Where:
                    tableName - The Amazon DynamoDB table to put the item in (for
 example, Music3).
```

```
                    partitionKeyName - The partition key name of the Amazon
 DynamoDB table (for example, Artist).
                    partitionKeyVal - The value of the partition key that should
 match (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];

        // For more information about an alias, see:
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";

        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
 partitionAlias);
        System.out.println("There were " + count + "  record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
 partitionKeyName, String partitionKeyVal,
            String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
 word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put(":" + partitionKeyName, AttributeValue.builder()
```

```
                .s(partitionKeyVal)
                .build());

        QueryRequest queryReq = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression(partitionAlias + " = :" +
 partitionKeyName)
                .expressionAttributeNames(attrNameAlias)
                .expressionAttributeValues(attrValues)
                .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

DynamoDbClient とセカンダリインデックスを使用してテーブルに対してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributesNames = new HashMap<>();
            expressionAttributesNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
            expressionAttributeValues.put(":yearValue",
 AttributeValue.builder().n("2013").build());

            QueryRequest request = QueryRequest.builder()
                    .tableName(tableName)
                    .indexName("year-index")
                    .keyConditionExpression("#year = :yearValue")
                    .expressionAttributeNames(expressionAttributesNames)
                    .expressionAttributeValues(expressionAttributeValues)
                    .build();

            System.out.println("=== Movie Titles ===");
            QueryResponse response = ddb.query(request);
            response.items()
                    .forEach(movie ->
 System.out.println(movie.get("title").s()));
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「QueryCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
    ConsistentRead: true,
```

```
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Success", data.Items);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun queryDynTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionKeyVal: String,
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = partitionKeyName

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)

    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }
```

```
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「Query」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);

    public function query(string $tableName, $key)
    {
        $expressionAttributeValues = [];
        $expressionAttributeNames = [];
        $keyConditionExpression = "";
        $index = 1;
        foreach ($key as $name => $value) {
            $keyConditionExpression .= "#" . array_key_first($value) . " = :v
$index,";
            $expressionAttributeNames["#" . array_key_first($value)] =
 array_key_first($value);
            $hold = array_pop($value);
```

```
            $expressionAttributeValues[":v$index"] = [
                array_key_first($hold) => array_pop($hold),
            ];
        }
        $keyConditionExpression = substr($keyConditionExpression, 0, -1);
        $query = [
            'ExpressionAttributeValues' => $expressionAttributeValues,
            'ExpressionAttributeNames' => $expressionAttributeNames,
            'KeyConditionExpression' => $keyConditionExpression,
            'TableName' => $tableName,
        ];
        return $this->dynamoDbClient->query($query);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「Query」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定の SongTitle と Artist を含む DynamoDB 項目を返すクエリを呼び出します。

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
```

```
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「Query」を参照してください。

Tools for PowerShell V5

例 1: 指定の SongTitle と Artist を含む DynamoDB 項目を返すクエリを呼び出します。

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「Query」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

キー条件式を使用して項目に対してクエリを実行します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
 self.table.query(KeyConditionExpression=Key("year").eq(year))
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Items"]
```

項目に対してクエリを実行し、データのサブセットを返すように射影します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
 titles
        that start within a range of letters. A projection expression is used
        to return a subset of data for each movie.

        :param year: The release year to query.
        :param title_bounds: The range of starting letters to query.
        :return: The list of movies.
        """
```

```
        try:
            response = self.table.query(
                ProjectionExpression="#yr, title, info.genres, info.actors[0]",
                ExpressionAttributeNames={"#yr": "year"},
                KeyConditionExpression=(
                    Key("year").eq(year)
                    & Key("title").between(
                        title_bounds["first"], title_bounds["second"]
                    )
                ),
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ValidationException":
                logger.warning(
                    "There's a validation error. Here's the message: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            else:
                logger.error(
                    "Couldn't query for movies. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Items"]
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Queries for movies that were released in the specified year.
  #
  # @param year [Integer] The year to query.
  # @return [Array] The list of movies that were released in the specified year.
  def query_items(year)
    response = @table.query(
      key_condition_expression: '#yr = :year',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: { ':year' => year }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't query for movies released in #{year}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.items
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「Query」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

指定した年に作られた映画を検索します。

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
 AttributeValue::N(year.to_string()))
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「Query」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
```

```
        " Query movies for a given year .
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
        DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |EQ|
          ) ) ) ).
        oo_result = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_key_conditions ).
        DATA(lt_items) = oo_result->get_items( ).
        "You can loop over the results to get item attributes.
        LOOP AT lt_items INTO DATA(lt_item).
          DATA(lo_title) = lt_item[ key = 'title' ]-value.
          DATA(lo_year) = lt_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_result->get_count( ).
        MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「Query」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB
```

```
    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
```

```
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「Query」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

テーブルのグローバルセカンダリインデックスを作成するには、「ステップ 6: (オプション) DynamoDB テーブルを削除してリソースをクリーンアップする」に進みます。

ステップ 6: (オプション) DynamoDB テーブルを削除してリソースをクリーンアップする

チュートリアルで作成した Amazon DynamoDB テーブルが不要になった場合は、削除します。このステップにより、使用していないリソースに対して課金されることがなくなります。DynamoDB コンソールまたは AWS CLI を使用して、ステップ 1: DynamoDB にテーブルを作成する で作成した `Music` テーブルを削除することができます。

DynamoDB のテーブルオペレーションの詳細については、「DynamoDB でのテーブルとデータの操作」を参照してください。

AWS マネジメントコンソール

コンソールを使用して `Music` テーブルを削除するには:

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. テーブルリストで、[Music] テーブルの横のチェックボックスをオンにします。
4. [削除] を選択します。

AWS CLI

以下の AWS CLI の例では、Music を使用して delete-table テーブルを削除します。

```
aws dynamodb delete-table --table-name Music
```

AWS SDK

次のコード例は、AWS SDK を使用して DynamoDB テーブルを削除する方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Deletes a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);

            Console.WriteLine($"Table {response.TableDescription.TableName}
 successfully deleted.");
            return true;

        }
```

```
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
{tableName}. {ex.Message}");
            return false;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeleteTable」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
###############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
```

```
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""
```

```
  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
```

```
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
########################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteTable」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Delete an Amazon DynamoDB table.
/*!
  \sa deleteTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteTable(const Aws::String &tableName,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「DeleteTable」を参照してください。

CLI

AWS CLI

テーブルを削除するには

以下の `delete-table` の例は `MusicCollection` テーブルを削除します。

```
aws dynamodb delete-table \
    --table-name MusicCollection
```

出力:

```
{
    "TableDescription": {
        "TableStatus": "DELETING",
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableName": "MusicCollection",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブルの削除」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteTable」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
```

```
 return err
}
```

- API の詳細については、『AWS SDK for Go API リファレンス』の「DeleteTable」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class DeleteTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
```

```
                    tableName - The Amazon DynamoDB table to delete (for example,
 Music3).

                **Warning** This program will delete the table that you specify!
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Deleting the Amazon DynamoDB table %s...\n",
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
 {
        DeleteTableRequest request = DeleteTableRequest.builder()
                .tableName(tableName)
                .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンス の「DeleteTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「DeleteTable」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「DeleteTable」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun deleteDynamoDBTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
```

```
        println("$tableNameVal was deleted")
    }
}
```

- API の詳細については、『AWS SDK for Kotlin API リファレンス』の「DeleteTable」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    public function deleteTable(string $TableName)
    {
        $this->customWaiter(function () use ($TableName) {
            return $this->dynamoDbClient->deleteTable([
                'TableName' => $TableName,
            ]);
        });
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「DeleteTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたテーブルを削除します。操作を続行する前に確認画面が表示されます。

```
Remove-DDBTable -TableName "myTable"
```

例 2: 指定されたテーブルを削除します。操作を続行する前に確認画面は表示されません。

```
Remove-DDBTable -TableName "myTable" -Force
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「DeleteTable」を参照してください。

Tools for PowerShell V5

例 1: 指定されたテーブルを削除します。操作を続行する前に確認画面が表示されます。

```
Remove-DDBTable -TableName "myTable"
```

例 2: 指定されたテーブルを削除します。操作を続行する前に確認画面は表示されません。

```
Remove-DDBTable -TableName "myTable" -Force
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「DeleteTable」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
```

```
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def delete_table(self):
        """
        Deletes the table.
        """
        try:
            self.table.delete()
            self.table = None
        except ClientError as err:
            logger.error(
                "Couldn't delete table. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンス の「DeleteTable」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Deletes the table.
  def delete_table
    @table.delete
    @table = nil
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete table. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンス の「DeleteTable」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn delete_table(client: &Client, table: &str) ->
 Result<DeleteTableOutput, Error> {
    let resp = client.delete_table().table_name(table).send().await;

    match resp {
        Ok(out) => {
            println!("Deleted table");
            Ok(out)
        }
        Err(e) => Err(Error::Unhandled(e.into())),
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「DeleteTable」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        " Wait till the table is actually deleted.
```

```
        lo_dyn->get_waiter( )->tablenotexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'Table ' && iv_table_name && ' deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table ' && iv_table_name && ' does not exist' TYPE 'E'.
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted since it is in use' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DeleteTable」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
```

```
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「DeleteTable」を参照してください。

DynamoDB のその他の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

DynamoDB について学習を続ける

Amazon DynamoDB の使用に関する詳細については、以下のトピックを参照してください。

- DynamoDB でのテーブルとデータの操作
- DynamoDB での項目と属性の操作
- DynamoDB のテーブルに対するクエリの実行
- DynamoDB のグローバルセカンダリインデックスの使用
- トランザクションでの使用
- DynamoDB Accelerator (DAX) とインメモリアクセラレーション
- DynamoDB と AWS SDK を使用したプログラミング

Console-to-Code を使用して Amazon DynamoDB のインフラストラクチャコードを生成する

Amazon Q Developer の Console-to-Code 機能は、手動でのテーブル作成の手順を再現可能なオートメーションコードに変換することで、Amazon DynamoDB のインフラストラクチャ管理を簡素化します。この機能は、開発者がデータベースリソース設定を環境全体で効率的にスケールするのに役立ちます。詳細については、「Amazon Q Developer Console-to-Code での AWS のサービスの自動化」を参照してください。

Console-to-Code は、パーティションキー、ソートキー、プロビジョニングされたスループット設定、セカンダリインデックスなど、詳細な DynamoDB テーブル設定をキャプチャし、正確な

infrastructure-as-code テンプレートに変換します。生成 AI を使用すると、ツールは生成されたコードがコンソールワークフロー中に確立されたパラメータの互換性を維持するようにします。

開発者は、次のような複数の形式で DynamoDB インフラストラクチャコードを生成できます。

- TypeScript、Python、Java の AWS Cloud Development Kit (AWS CDK)
- YAML または JSON の CloudFormation

このアプローチにより、チームは次のことが可能になります。

- データベースリソース管理を標準化する
- バージョン管理下のインフラストラクチャを実装する
- 手動設定のエラーを減らす

Amazon DynamoDB の Console-to-Code は、すべての商用 AWS リージョンで利用可能で、手動設定を自動化された再現可能なインフラストラクチャコードに変換するための強力なソリューションを提供します。

仕組み

DynamoDB で Console-to-Code を使用する場合、通常、プロセスには次が含まれます。

1. コンソールでのプロトタイプ作成 - DynamoDB コンソールを使用して、テーブルなどのリソースを作成し、設定します。詳細については、「Amazon DynamoDB に接続する」を参照してください。
2. アクションの記録 - Console-to-Code は、実行するときにこれらのアクションを記録します。
3. コードの生成 - この機能は Amazon Q Developer の生成 AI 機能を使用して、コンソールアクションを任意の形式で再利用可能なコードに変換します。
4. コードのカスタマイズ - その後、このコードをコピーまたはダウンロードし、本番稼働用ワークロードに合わせてさらにカスタマイズできます。

DynamoDB で Console-to-Code を使用する利点

オートメーションの簡素化

手動の DynamoDB テーブルの作成と設定を、ワンクリックで再利用可能なコードに変換します。

ベストプラクティス

生成されたコードは、AWS ガイドの信頼性の高いデプロイのベストプラクティスに従います。

コンソールとコード間のブリッジ

AWS マネジメントコンソール または Infrastructure-as-Code (IaC) のどちらを使用するかを選択する必要はありません。両方のアプローチを一緒に使用できます。

開発の加速

本番稼働環境用にさらにカスタマイズできるオートメーションコードをすぐに使用開始できます。

ユースケースの例

- 特定の属性、キー、容量設定を使用した DynamoDB テーブルの作成
- グローバルセカンダリインデックスとローカルセカンダリインデックスの設定
- DynamoDB テーブルの自動スケーリングポリシーの設定
- バックアップと復元の設定の確立
- DynamoDB ストリームの作成と管理

開始方法

DynamoDB で Console-to-Code の使用を開始するには。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodbv2/) を開きます。
2. コンソールインターフェイスを使用して DynamoDB リソースの作成または変更を開始します。
3. Console-to-Code 機能を使用して、任意の形式でアクションのコードを生成します。
4. 生成されたコードをコピーまたはダウンロードし、特定の要件に応じて必要に応じてカスタマイズします。

Console-to-Code の使用方法に関する詳細と手順については、「Amazon Q Developer ユーザーガイド」の「Amazon Q Developer Console-to-Code を使用した AWS のサービスの自動化」を参照してください。

Amazon DynamoDB: 仕組み

以下のセクションでは、Amazon DynamoDB サービスコンポーネントと、それらの対話方法の概要を示します。

トピック

- DynamoDB のチートシート
- Amazon DynamoDB のコアコンポーネント
- DynamoDB API
- Amazon DynamoDB でサポートされるデータ型と命名規則
- DynamoDB テーブルクラス
- DynamoDB におけるパーティションとデータ分散
- SQL から NoSQL に移行する方法について学ぶ
- Amazon DynamoDB の学習リソースとツール

DynamoDB のチートシート

このチートシートは、Amazon DynamoDB とそのさまざまな AWS SDK を操作するためのクイックリファレンスを提供します。

初期セットアップ

1. にサインアップします。AWS
2. プログラムで DynamoDB にアクセスするための AWS アクセスキーを取得します。
3. DynamoDB 認証情報を設定します。

以下も参照してください。

- DynamoDB (ウェブサービス) の設定
- DynamoDB の使用開始
- コアコンポーネントの基本概要

SDK または CLI

任意の SDK を選択するか、AWS CLI を設定します。

> **Note**
>
> Windows で AWS CLI を使用すると、引用符の中にないバックスラッシュ (\) はキャリッジリターンとして扱われます。また、他の引用符の中にある引用符や中かっこはエスケープする必要があります。例については、次のセクションの「テーブルの作成」の [Windows] タブを参照してください。

以下も参照してください。

- AWS CLIDynamoDB での
- DynamoDB の使用開始 – ステップ 2

基本アクション

このセクションでは、基本的な DynamoDB タスクのコードを提供します。これらのタスクの詳細については、「DynamoDB とAWS SDK の使用開始」を参照してください。

テーブルを作成する

Default

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD
```

Windows

```
aws dynamodb create-table ^
```

```
    --table-name Music ^
    --attribute-definitions ^
        AttributeName=Artist,AttributeType=S ^
        AttributeName=SongTitle,AttributeType=S ^
    --key-schema AttributeName=Artist,KeyType=HASH
AttributeName=SongTitle,KeyType=RANGE ^
    --billing-mode PAY_PER_REQUEST ^
    --table-class STANDARD
```

テーブルへの項目の書き込み

```
aws dynamodb put-item \ --table-name Music \ --item file://item.json
```

テーブルから項目の読み込み

```
aws dynamodb get-item \ --table-name Music \ --item file://item.json
```

テーブルから項目の削除

```
aws dynamodb delete-item --table-name Music --key file://key.json
```

テーブルに対してクエリを実行する

```
aws dynamodb query --table-name Music
--key-condition-expression "ArtistName=:Artist and SongName=:Songtitle"
```

テーブルを削除する

```
aws dynamodb delete-table --table-name Music
```

テーブル名のリスト化

```
aws dynamodb list-tables
```

名前付けルール

- すべての名前は UTF-8 を使用してエンコードする必要があり、大文字と小文字が区別されます。

- テーブル名とインデックス名の長さは 3～255 文字で、次の文字のみを含めることができます。
  - a-z
  - A-Z
  - 0-9
  - _ (下線)
  - - (ハイフン)
  - . (ドット)
- 属性名は 1 文字以上、64 KB 未満のサイズである必要があります。

詳細については、「命名規則」を参照してください。

サービスクォータの基本

読み込みユニットと書き込みユニット

- 読み込みキャパシティユニット (RCU) – 最大サイズが 4 KB の項目について、読み込みキャパシティユニット = 1 秒あたり 1 回の強力な整合性のある読み込み、あるいは 1 秒あたり 2 回の結果整合性のある読み込み。
- 書き込みキャパシティユニット (WCU) – 最大サイズが 1 KB の項目について、書き込みキャパシティユニット = 1 秒あたり 1 回の書き込み。

テーブルリミット

- テーブルサイズ – テーブルのサイズには実用的な制限はありません。テーブルは項目数やバイト数について制限がありません。
- テーブルの数 – AWS アカウントについては、AWS リージョンごとに 2,500 個のテーブルという初期クォータがあります。
- クエリとスキャンのページサイズの制限 – クエリまたはスキャンごとに、1 ページあたり 1 MB に制限されています。クエリパラメータまたはテーブルでのスキャン操作の結果データが 1 MB を超える場合、DynamoDB は最初に一致した項目を返します。また、新しいリクエストで次のページを読むときに使用できる LastEvaluatedKey プロパティも返されます。

インデックス

- ローカルセカンダリインデックス (LSI) – 最大 5 つのローカルセカンダリインデックスを定義できます。LSI は主に、インデックスがベーステーブルと強い一貫性を持たなければならない場合に役立ちます。
- グローバルセカンダリインデックス (GSI) – デフォルトクォータとして、テーブルごとに 20 個のグローバルセカンダリインデックスがあります。
- テーブル毎のセカンダリインデックス属性の射影 – 合計最大 100 の属性を、1 つのテーブルのすべてのグローバルセカンダリインデックスに射影することができます。これは、ユーザー指定の射影された属性だけに適用されます。

パーティションキー

- パーティションキーと値の最小長は 1 バイトです。最大長は 2048 バイトです
- テーブルまたはセカンダリインデックスについて、パーティションキー値の明確な数に関する実質的な制限はありません。
- ソートキーと値の最小長は 1 バイトです。最大長は 1024 バイトです
- 一般的に、パーティションキーの値ごとのソートキーの値の数について、実質的に制限はありません。セカンダリインデックスを持つテーブルは例外です。

セカンダリインデックス、パーティションキー設計、ソートキー設計の詳細については、「ベストプラクティス」を参照してください。

一般的に使用されるデータ型の制限

- 文字列 – 文字列の長さは、項目の最大サイズである 400 KB に制約されます。文字列は、UTF-8 バイナリエンコードの Unicode です。
- 数値 – 数値は、最大 38 桁の精度であり、正、負、または 0 のいずれかです。
- バイナリ – バイナリの長さは、項目の最大サイズである 400 KB に制約されます。バイナリ属性を操作するアプリケーションは、データを DynamoDB に送信する前に、base64 でエンコードする必要があります。

サポートされているデータ型のリストについては、「データ型」を参照してください。詳細については、「Service Quotas」(サービスクォータ) を参照してください。

項目、属性、式パラメータ

DynamoDB の項目の最大サイズは 400 KB で、属性名のバイナリの長さ (UTF-8 の長さ) と属性値の長さ (UTF-8 の長さ) を含みます。属性名はサイズ制限に反映されます。

値を含む項目が 400 KB の制限内である限り、リスト、マップ、またはセットにおける値の最大数の制限はありません。

式パラメータでは、式文字列の最大長は 4 KB です。

項目サイズ、属性、および式パラメータの詳細については、「Service Quotas」(サービスクォータ) を参照してください。

詳細情報

- セキュリティ
- モニタリングと口グ記録:
- ストリームの使用
- バックアップとポイントインタイムリカバリ
- 他の AWS のサービスとの統合
- API リファレンス
- アーキテクチャセンター: データベースのベストプラクティス
- ビデオチュートリアル
- DynamoDB フォーラム

Amazon DynamoDB のコアコンポーネント

DynamoDB では、テーブル、項目、および属性が、操作するコアコンポーネントです。テーブルは項目の集合であり、各項目は属性の集合です。DynamoDB は、テーブルの各項目を一意に識別するために、プライマリキーを使用します。DynamoDB Streams を使用して、DynamoDB テーブルのデータ変更イベントをキャプチャできます。

DynamoDB には制限があります。詳細については、「Amazon DynamoDB のクォータ」を参照してください。

次の動画では、テーブル、項目、および属性の概要を説明します。

テーブル、項目、属性

テーブル、項目、属性

People

Primary key Partition key: PersonID	Attributes			
101	LastName	FirstName	Phone	
	Smith	Fred	555-4321	
102	LastName	FirstName	Address	
	Jones	Mary	{"Street":"123 Main","City":"Anytown","State":"OH","ZIPCode":12345}	
103	LastName	FirstName	FavoriteColor	Address
	Stephens	Howard	Blue	{"Street":"123 Main","City":"London","PostalCode":"ER3 5K8"}

基本的な DynamoDB コンポーネントは以下のとおりです。

- テーブル – 他のデータベースシステムと同様、DynamoDB はデータをテーブルに保存します。テーブルは、データのコレクションです。たとえば、テーブルの例 (People) を参照してください。このテーブルは、友人、家族、関心のある人に関する個人の連絡先情報を保存するのに使用できます。また、その人たちが運転する車に関する情報を保存する Cars テーブルを作成することもできます。
- 項目 – 各テーブルにはゼロ以上の項目が含まれています。項目は、他のすべての項目間で一意に識別可能な属性のグループです。People テーブルの各項目は、人を表します。Cars テーブルの各項目は 1 台の車を表します。DynamoDB の項目は、多くの点で他のデータベースシステムの行、レコード、またはタプルに似ています。DynamoDB では、テーブルに保存できる項目数に制限はありません。
- 属性 – 各項目は 1 つ以上の属性で構成されます。属性は、基盤となるデータ要素であり、それ以上分割する必要がないものです。例えば、People テーブルの項目には、PersonID、LastName、FirstName といった名前の属性が含まれます。Department テーブルでは、項目が DepartmentID、Name、Manager などの属性を設定することができます。DynamoDB 内の属性は、多くの点で他のデータベースシステムのフィールドや列に似ています。

次の図は、いくつかの項目と属性の例を含む、People という名前のテーブルを示しています。

```
People

{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}

{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}

{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
                "Street": "123 Main",
                "City": "London",
                "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

People テーブルについて、以下の点に注意してください。

- テーブルの各項目には一意の識別子があります。これは、テーブルの他のすべての項目からその項目を区別するプライマリキーです。People テーブルで、プライマリキーは 1 つの属性 (PersonID) で構成されます。

- プライマリキー以外、People テーブルはスキーマレスです。つまり、属性またはデータ型を事前に定義する必要はありません。各項目は、独自の固有の属性を持つことができます。
- 属性のほとんどはスカラーです。つまり、1 つの値のみを持つことができます。文字列と数値はスカラーの一般的な例です。
- 一部の項目には、ネストされた属性 (アドレス) があります。DynamoDB は深さが最大 32 レベルの入れ子の属性をサポートします。

以下は、音楽コレクションを追跡するために使用できる、Music という名前の別のサンプルテーブルです。

```
Music

{
    "Artist": "No One You Know",
    "SongTitle": "My Dog Spot",
    "AlbumTitle": "Hey Now",
    "Price": 1.98,
    "Genre": "Country",
    "CriticRating": 8.4
}

{
    "Artist": "No One You Know",
    "SongTitle": "Somewhere Down The Road",
    "AlbumTitle": "Somewhat Famous",
    "Genre": "Country",
    "CriticRating": 8.4,
    "Year": 1984
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Still in Love",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 2.47,
    "Genre": "Rock",
    "PromotionInfo": {
        "RadioStationsPlaying": [
            "KHCR",
            "KQBX",
```

```
            "WTNR",
            "WJJH"
        ],
        "TourDates": {
            "Seattle": "20150622",
            "Cleveland": "20150630"
        },
        "Rotation": "Heavy"
    }
}

{
    "Artist": "The Acme Band",
    "SongTitle": "Look Out, World",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 0.99,
    "Genre": "Rock"
}
```

Music テーブルについて、以下の点に注意してください。

- Music のプライマリキーは 2 つの属性 (Artist および SongTitle) で構成されます。テーブルの各項目にはこれら 2 つの属性が必要です。Artist および SongTitle の組み合わせにより、テーブルの各項目が他のすべての項目から区別されます。
- プライマリキー以外、Music テーブルはスキーマレスです。つまり、属性またはデータ型を事前に定義する必要はありません。各項目は、独自の固有の属性を持つことができます。
- 項目の 1 つに、入れ子の属性 (PromotionInfo) があります。これには、入れ子の他の属性が含まれます。DynamoDB は深さが最大 32 レベルの入れ子の属性をサポートします。

詳細については、「DynamoDB でのテーブルとデータの操作」を参照してください。

プライマリキー

テーブルを作成する場合には、テーブル名に加えて、テーブルのプライマリキーを指定する必要があります。プライマリキーはテーブルの各項目を一意に識別するため、テーブル内の 2 つの項目が同じキーを持つことはありません。

DynamoDB は 2 種類の異なるプライマリキーをサポートします。

- パーティションキー – パーティションキーという 1 つの属性で構成されたシンプルなプライマリキー。

  DynamoDB は、パーティションキーの値を内部ハッシュ関数への入力として使用します。ハッシュ関数からの出力により、項目が保存されるパーティション (DynamoDB 内部の物理ストレージ) が決まります。

  パーティションキーのみを含むテーブルでは、2 つの項目が同じパーティションキー値を持つことはできません。

  テーブル、項目、属性 で説明されている People テーブルは、単純なプライマリキー (PersonID) を持つテーブルの例です。People テーブル内の任意の項目に直接アクセスするには、その項目の PersonId 値を指定します。

- パーティションとソートキー – 複合プライマリキーと呼ばれるこのキーのタイプは、2 つの属性で構成されます。最初の属性はパーティションキーであり、2 番目の属性はソートキーです。

  DynamoDB は、パーティションキーバリューを内部ハッシュ関数への入力として使用します。ハッシュ関数からの出力により、項目が保存されるパーティション (DynamoDB 内部の物理ストレージ) が決まります。同じパーティションキー値を持つすべての項目は、ソートキー値でソートされてまとめて保存されます。

  パーティションキーとソートキーが存在するテーブルでは、同じパーティションのキーバリューが複数の項目に割り当てられることがあります。ただし、ソートキー値は複数の項目で異なる必要があります。

  テーブル、項目、属性 で説明されている Music テーブルは、複合プライマリキー (Artist および SongTitle) を持つテーブルの例です。その項目に Artist と SongTitle の値を指定すると、Music テーブルの任意の項目に直接アクセスできます。

  複合プライマリキーは、データのクエリを実行するときに柔軟性を高めます。たとえば、Artist の値のみを指定した場合、DynamoDB はそのアーティストのすべての曲を取得します。特定のアーティストの曲のサブセットのみを取得するには、Artist の値と SongTitle の値範囲を指定します。

> **Note**
>
> 項目のパーティションキーは、そのハッシュ属性とも呼ばれます。ハッシュ属性という用語は、DynamoDB が内部のハッシュ関数を使用し、パーティションキーバリューに基づいてパーティション間でデータ項目を均等に分散することに由来しています。

> 項目のソートキーは、範囲属性とも呼ばれます。範囲属性という用語は、ソートキーバリューで並べ替えられた順に、DynamoDB が同じパーティションキーを持つ項目同士を物理的に近くに保存する方法に由来しています。

各プライマリキー属性はスカラー値 (単一値のみを保持できる) である必要があります。プライマリキー属性に許可される唯一のデータ型は、文字列、数値、またはバイナリです。他のキー以外の属性では、このような制限はありません。

セカンダリインデックス

テーブルで 1 つ以上のセカンダリインデックスを作成できます。セカンダリインデックスでは、プライマリキーに対するクエリに加えて、代替キーを使用して、テーブル内のデータのクエリを行うことができます。DynamoDB では、インデックスを使用する必要はありませんが、インデックスを使用すると、データのクエリを行う際にアプリケーションの柔軟性が高まります。テーブルにグローバルセカンダリインデックスを作成すると、テーブルから行う場合とほぼ同じ方法でインデックスからデータを読み取ることができます。

DynamoDB では、次の 2 種類のインデックスをサポートしています。

- グローバルセカンダリインデックス – パーティションキーおよびソートキーを持つインデックス。テーブルのものとは異なる場合があります。グローバルセカンダリインデックスのプライマリキーの値は一意である必要はありません。
- ローカルセカンダリインデックス – パーティションキーはテーブルと同じですが、ソートキーが異なるインデックスです。

DynamoDB では、グローバルセカンダリインデックス (GSI) はテーブル全体にまたがるインデックスであり、すべてのパーティションキーをクエリできます。ローカルセカンダリインデックス (LSI) は、ベーステーブルとパーティションキーは同じで、ソートキーが異なるインデックスです。

DynamoDB の各テーブルには、20 個のグローバルセカンダリーインデックス (デフォルトのクォータ) と、5 個のローカルセカンダリーインデックスのクォータがあります。

前に示した Music サンプルテーブルでは、Artist (パーティションキー) または Artist および SongTitle (パーティションキーとソートキー) によってデータ項目にクエリを実行できます。Genre および AlbumTitle によってデータにクエリを実行する場合はどうでしょうか。これを行うには、Genre および AlbumTitle にインデックスを作成し、Music テーブルのクエリと同様に、インデックスにクエリを実行できます。

次の図表は、GenreAlbumTitle という新しいインデックスを持つ Music テーブルの例を示しています。このインデックスでは、Genre がパーティションキーで、AlbumTitle がソートキーです。

Music テーブル	GenreAlbumTitle

<pre>{
"Artist": "No One You Know",
"SongTitle": "My Dog Spot",
"AlbumTitle": "Hey Now",
"Price": 1.98,
"Genre": "Country",
"CriticRating": 8.4
}</pre>
	<pre>{
	"Genre": "Country",
	"AlbumTitle": "Hey Now",
	"Artist": "No One You Know",
	"SongTitle": "My Dog Spot"
	}</pre>

<pre>{
"Artist": "No One You Know",
"SongTitle": "Somewhere Down The Road",
"AlbumTitle": "Somewhat Famous",
"Genre": "Country",
"CriticRating": 8.4,
"Year": 1984
}</pre>
	<pre>{
	"Genre": "Country",
	"AlbumTitle": "Somewhat Famous",
	"Artist": "No One You Know",
	"SongTitle": "Somewhere Down The Road"
	}</pre>

<pre>{
"Artist": "The Acme Band",
"SongTitle": "Still in Love",
"AlbumTitle": "The Buck Starts Here",
"Price": 2.47,
"Genre": "Rock",
"PromotionInfo": {
"RadioStationsPlaying": {
"KHCR",
"KQBX",
"WTNR",</pre>
	<pre>{
	"Genre": "Rock",
	"AlbumTitle": "The Buck Starts Here",
	"Artist": "The Acme Band",
	"SongTitle": "Still In Love"
	}</pre>

Music テーブル	GenreAlbumTitle

<pre>"WJJH"
},
"TourDates": {
"Seattle": "20150622",
"Cleveland": "20150630"
},
"Rotation": "Heavy"
}
}</pre>	

<pre>{
"Artist": "The Acme Band",
"SongTitle": "Look Out, World",
"AlbumTitle": "The Buck Starts
Here",
"Price": 0.99,
"Genre": "Rock"
}</pre>
	<pre>{
	"Genre": "Rock",
	"AlbumTitle": "The Buck Starts
	Here",
	"Artist": "The Acme Band",
	"SongTitle": "Look Out, World"
	}</pre>

GenreAlbumTitle インデックスについて、以下の点に注意してください。

- 各インデックスはテーブルに属します。これをインデックスの基本テーブルと呼びます。前述の例では、Music が GenreAlbumTitle インデックスの基本テーブルです。
- DynamoDB はインデックスを自動的に維持します。基本テーブルの項目を追加、更新、または削除すると、DynamoDB はそのテーブルに属するすべてのインデックスの対応する項目を追加、更新、または削除します。
- インデックスを作成するときは、基本テーブルからインデックスにコピーまたは射影される属性を指定します。少なくとも、DynamoDB は基本テーブルからインデックスにキー属性を射影します。これは `GenreAlbumTitle` のケースで、`Music` テーブルのキー属性のみがインデックスに射影されます。

GenreAlbumTitle インデックスにクエリを実行し、特定のジャンルのすべてのアルバム (たとえば、すべての Rock アルバム) を検索できます。また、インデックスにクエリを実行して、特定のジャン

ル内のすべてのアルバムのうち、特定のアルバムタイトル (たとえば、タイトルが文字 H で始まるすべての Country アルバム) のみを検索することもできます。

詳細については、「DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上」を参照してください。

DynamoDB Streams

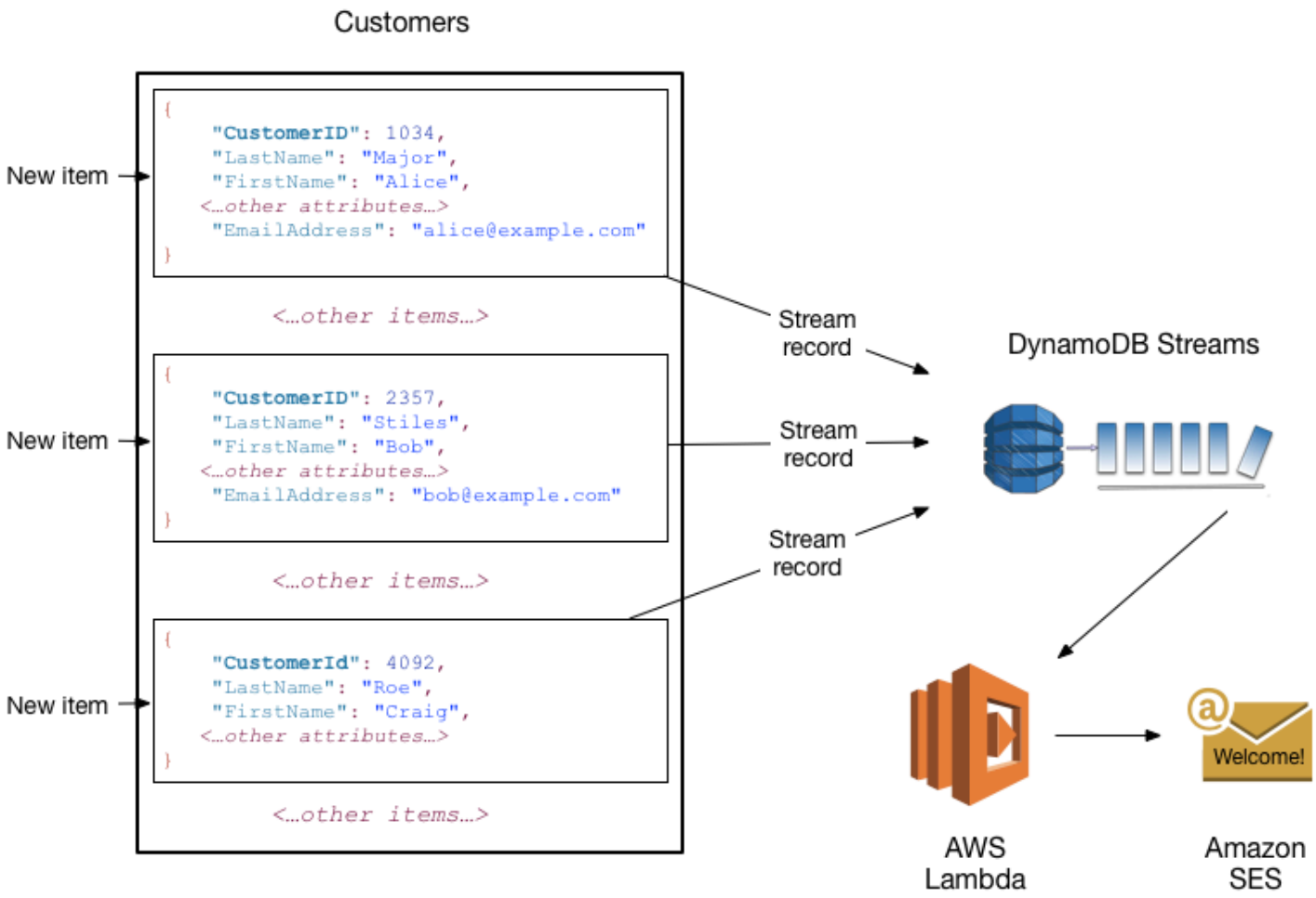
DynamoDB Streams は、DynamoDB テーブルのデータ変更イベントをキャプチャするオプションの特徴です。これらのイベントに関するデータは、ほとんどリアルタイムに、イベントの発生順にストリームに表示されます。

各イベントはストリームレコードによって表されます。テーブルでストリーミングを有効にすると、DynamoDB Streams は次のいずれかのイベントが発生するたびに、ストリーミングレコードを書き込みます。

- 新しい項目がテーブルに追加された場合: ストリームは、すべての属性を含む項目全体のイメージをキャプチャします。
- 項目が更新された場合: ストリームは、項目で変更された属性について、「前」と「後」のイメージをキャプチャします。
- テーブルから項目が削除された場合: ストリームは、項目が削除される前に項目全体のイメージをキャプチャします。

各ストリームレコードには、テーブルの名前、イベントのタイムスタンプ、およびその他のメタデータも含まれます。ストリームレコードには 24 時間の有効期間があり、その後はストリームから自動的に削除されます。

DynamoDB Streams を AWS Lambda と共に使用して、トリガーを作成できます。トリガーは、対象イベントがストリーミングに表示されるたびに自動的に実行されます。たとえば、会社の顧客情報を含む Customers テーブルがあるとします。新規の各顧客に、「ようこそ」 E メールを送信するとします。そのテーブルでストリーミングを有効にし、そのストリーミングを Lambda 関数に関連付けます。Lambda 関数は、新しいストリーミングレコードが表示されるたびに実行されますが、Customers テーブルに追加された新しい項目のみを処理します。`EmailAddress` 属性を持つ項目について、Lambda 関数は Amazon Simple Email Service (Amazon SES) をコールしてそのアドレスに E メールを送信します。



> **Note**
>
> この例で、最後の顧客 Craig Roe には `EmailAddress` がないため E メールを受信することはありません。

トリガーに加えて、DynamoDB Streams は AWS リージョン内全体のデータレプリケーション、DynamoDB テーブル内のデータのマテリアライズドビュー、Kinesis のマテリアライズドビューを使用したデータ分析など、数多くの強力なソリューションを可能にします。

詳細については、「DynamoDB Streams の変更データキャプチャ」を参照してください。

DynamoDB API

Amazon DynamoDB を使用するには、アプリケーションでいくつかの簡単な API オペレーションを使用する必要があります。以下に、カテゴリー別にこれらのオペレーションの概要を示します。

> **Note**
>
> API オペレーションの完全なリストについては、「Amazon DynamoDB API リファレンス」を参照してください。

トピック

- コントロールプレーン
- データプレーン
- DynamoDB Streams
- 取引

コントロールプレーン

コントロールプレーンのオペレーションでは、DynamoDB テーブルを作成および管理できます。また、インデックス、ストリーム、およびテーブルに依存する他のオブジェクトを操作できます。

- `CreateTable` – 新しいテーブルを作成します。オプションで、1 つ以上のセカンダリインデックスを作成し、テーブルに対して DynamoDB Streams を有効にできます。
- `DescribeTable` – プライマリキーのスキーマ、スループット設定、インデックス情報など、テーブルに関する情報を返します。
- `ListTables` – リストのすべてのテーブルの名前を返します。
- `UpdateTable` – テーブルまたはそのインデックスの設定を変更、テーブルの新しいインデックスを作成または削除、またはテーブルの DynamoDB Streams 設定を変更します。
- `DeleteTable` – テーブルとそのすべての依存オブジェクトを DynamoDB から削除します。

データプレーン

データプレーンオペレーションでは、テーブルのデータで、作成、読み込み、更新、および削除 (CRUD とも呼ばれる) アクションを実行できます。一部のデータプレーンオペレーションでも、セカンダリインデックスからデータを読み込むことができます。

PartiQL: Amazon DynamoDB 用の SQL 互換クエリ言語 を使用してこれらの CRUD オペレーションを実行するか、各オペレーションを個別の API コールに分離する DynamoDB の従来の CRUD API を使用できます。

PartiQL - SQL 互換クエリ言語

- ExecuteStatement – テーブルから複数の項目を読み込みます。テーブルから単一の項目を書き込むか、更新することもできます。単一の項目を書き込むか、更新する場合は、プライマリキー属性を指定する必要があります。
- BatchExecuteStatement – テーブルから複数の項目を書き込み、更新または読み込みます。これは、ExecuteStatement よりも効率的です。アプリケーションで項目を書き込んだり読み込んだりするために、1 回のネットワークラウンドトリップのみで済むためです。

Classic API

データの作成

- PutItem – テーブルに単一の項目を書き込みます。プライマリキー属性を指定する必要がありますが、その他の属性を指定する必要はありません。
- BatchWriteItem – 最大 25 個の項目をテーブルに書き込みます。これは、PutItem を複数回呼び出すよりも効率的です。アプリケーションで項目を書き込むために、1 回のネットワークラウンドトリップのみで済むためです。

データの読み込み

- GetItem – テーブルから単一の項目を取り出します。目的の項目のプライマリキーを指定する必要があります。項目全体またはその属性のサブセットのみを取り出すことができます。
- BatchGetItem – 1 つ以上のテーブルから最大 100 個の項目を取り出します。これは、GetItem を複数回呼び出すよりも効率的です。アプリケーションで項目を読み込むために、1 回のネットワークラウンドトリップのみで済むためです。
- Query – 特定のパーティションキーがあるすべての項目を取り出します。パーティションキーの値を指定する必要があります。項目全体またはその属性のサブセットのみを取り出すことができます。オプションで、ソートキーの値に条件を適用し、同じパーティションキーがあるデータのサブセットだけを取り出すことができます。テーブルにパーティションキーとソートキーの両方を持つテーブルがある場合、テーブルでこのオペレーションを使用できます。また、インデックスにパーティションキーとソートキーの両方がある場合、インデックスでこのオペレーションを使用できます。
- Scan – 指定されたテーブルまたはインデックスのすべての項目を取り出します。項目全体またはその属性のサブセットのみを取り出すことができます。オプションでフィルタリング条件を適用すると、関心のある値のみを返し、残りは破棄できます。

データの更新

- UpdateItem – 項目の 1 つ以上の属性を変更します。変更する項目のプライマリキーを指定する必要があります。新しい属性を追加したり、既存の属性を変更または削除したりできます。ユーザー定義の条件を満たす場合にのみ更新が成功するように、条件付きの更新を実行できます。オプションで、アトミックカウンターを実装できます。このカウンタは、他の書き込みリクエストを妨害することなく、数値属性をインクリメントまたはデクリメントします。

データの削除

- DeleteItem – テーブルから単一の項目を削除します。削除する項目のプライマリキーを指定する必要があります。
- BatchWriteItem – 1 つ以上のテーブルから最大 25 個の項目を削除します。これは、DeleteItem を複数回呼び出すよりも効率的です。アプリケーションで項目を削除するために、1 回のネットワークラウンドトリップのみで済むためです。

> **Note**
>
> BatchWriteItem は、データの作成とデータの削除の両方に使用できます。

DynamoDB Streams

DynamoDB Streams オペレーションでは、テーブルのストリーミングを有効または無効にし、ストリーミングに含まれるデータ変更レコードにアクセスするように許可します。

- ListStreams – すべてのストリーミングのリスト、または特定のテーブルのストリーミングのみを返します。
- DescribeStream – Amazon リソースネーム (ARN) およびアプリケーションが最初のいくつかのストリーミングレコードの読み込みを開始できる場所など、ストリーミングに関する情報を返します。
- GetShardIterator – シャードイテレーターを返します。これは、ストリーミングからレコードを取得するためにアプリケーションが使用するデータ構造です。
- GetRecords – 特定のシャードイテレーターを使用して 1 つ以上のストリーミングレコードを取得します。

取引

トランザクションによって不可分性、一貫性、分離性、耐久性 (ACID) が実現されるため、アプリケーション内でのデータの精度を維持することがさらに容易になります。

PartiQL: Amazon DynamoDB 用の SQL 互換クエリ言語 を使用してこれらのトランザクションオペレーションを実行するか、各オペレーションを個別の API コールに分離する DynamoDB の従来の CRUD API を使用できます。

PartiQL - SQL 互換クエリ言語

- `ExecuteTransaction` – テーブル内または複数のテーブル間の複数の項目に対して、オールオアナッシングの結果が保証された CRUD オペレーションを実行できるバッチ操作です。

Classic API

- `TransactWriteItems` – テーブル内または複数のテーブル間の複数の項目に対して、オールオアナッシングの結果が保証された `Put`、`Update` および `Delete` オペレーションを実行できるバッチオペレーションです。
- `TransactGetItems` – 1 つ以上のテーブルから複数の項目を取得する `Get` オペレーションを実行できるバッチ操作です。

Amazon DynamoDB でサポートされるデータ型と命名規則

このセクションでは、Amazon DynamoDB の命名規則と、DynamoDB がサポートするさまざまなデータ型について説明します。データタイプに適用される制限があります。詳細については、「データ型」を参照してください。

トピック

- 名前付けルール
- データ型
- データ型記述子

名前付けルール

DynamoDB のテーブル、属性、および他のオブジェクトには名前が必要です。名前には意味があり、簡潔でなければなりません。たとえば、製品、ブック、および著者などの名前は一目瞭然です。

DynamoDB の命名規則は次のとおりです。

- すべての名前は UTF-8 を使用してエンコードする必要があり、大文字と小文字が区別されます。
- テーブル名とインデックス名の長さは 3 ~ 255 文字で、次の文字のみを含めることができます。
  - `a-z`
  - `A-Z`
  - `0-9`
  - _ (下線)
  - - (ダッシュ)
  - . (ドット)
- 属性名は 1 文字以上の長さ、64 KB 未満のサイズにする必要があります。属性名はできるだけ短くすることがベストプラクティスであると考えられています。これにより、属性名がストレージとスループットの使用量の測定に含まれるため、消費される読み取りリクエストユニットを減らすことができます。

  以下の例外があります。これらの属性名は 255 文字以下である必要があります。
  - セカンダリインデックスのパーティションキー名
  - セカンダリインデックスのソートキー名
  - ユーザー指定の射影された属性の名前 (ローカルセカンダリインデックスにのみ適用)

予約語と特殊文字

DynamoDB には予約語と特殊文字のリストもあります。詳細な一覧については、「DynamoDB の予約語」を参照してください。DynamoDB では、# (ハッシュ) および : (コロン) に特別な意味があります。

DynamoDB では、命名目的でこれらの予約語と特殊文字を使用することができますが、お勧めしません。これは、式でこれらの名前を使用するたびに、プレースホルダー変数を定義する必要があるためです。詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

データ型

DynamoDB では、テーブル内の属性に対してさまざまなデータ型がサポートされています。データ型は次のように分類できます。

- スカラー型 – スカラー型は 1 つの値を表すことができます。スカラー型は、数値、文字列、バイナリ、ブール、および null です。
- ドキュメント型 – JSON ドキュメントで見られるように、入れ子の属性を持つ複雑な構造を表すことができます。ドキュメント型は、リストとマップです。
- セット型 – セット型は複数のスカラー値を表すことができます。セット型は、文字セット、数値セット、およびバイナリセットです。

テーブルまたはセカンダリインデックスを作成するときは、各プライマリキー属性 (パーティションキーとソートキー) の名前とデータ型を指定する必要があります。さらに、各プライマリキー属性は、文字列、数値、またはバイナリとして定義する必要があります。

DynamoDB は NoSQL データベースであり、スキーマレスです。つまり、テーブルを作成する場合、プライマリキー属性を除き、属性やデータ型を定義する必要はありません。これに対して、リレーショナルデータベースでは、テーブルの作成時に各列の名前とデータ型を定義する必要があります。

ここでは、各データ型について JSON 形式の例を挙げて説明します。

スカラー型

スカラー型は、数値、文字列、バイナリ、ブール、および null です。

Number

数値は、正、負、または 0 とすることができます。数値は最大 38 桁の精度を持つことができます。これを超えると例外が発生します。38 桁を超える精度が必要な場合は、文字列を使用できます。

- 正の範囲: 1E-130 ~ 9.99999999999999999999999999999999999999E+125
- 負の範囲: -9.99999999999999999999999999999999999999E+125 ~ -1E-130

DynamoDB では、数値は可変長で表されます。先頭と末尾の 0 は切り捨てられます。

すべての数値はネットワーク経由で文字列として DynamoDB に送信され、言語やライブラリ間の互換性が最大になります。ただし、DynamoDB は算術演算では数値を数値型属性として扱います。

数値データ型を使用して、日付またはタイムスタンプを表すことができます。これを行うための方法の 1 つは、1970 年 1 月 1 日 00:00:00 UTC から経過した秒数であるエポックタイムを使用することです。たとえば、エポック時間 1437136300 は、2015 年 7 月 17 日の 12:31:40 PM UTC を表します。

タイムスタンプを数値として保存すると、同等の ISO 8601 文字列よりも使用するストレージが少なくなります。これは、数百万または数十億の項目を含むテーブル全体で積み重なると、大きな差異となる可能性があります。エポック秒として保存すると、DynamoDB 有効期限 (TTL) 機能で数値を使用して、期限切れの項目を自動的に削除することもできます。あるいは、日付と時刻の値を、人間が読み取れる ISO 8601 形式で文字列として保存することができます。詳細については、「DynamoDB での Time to Live (TTL) の使用」を参照してください。

詳細については、http://en.wikipedia.org/wiki/Unix_time を参照してください。

String

文字列は、UTF-8 バイナリエンコードの Unicode です。属性がインデックスまたはテーブルのキーとして使用されない場合、文字列の最小長は 0 になります。また、DynamoDB 項目の最大サイズ上限である 400 KB の制約があります。

文字列型として定義されているプライマリキー属性には、さらに以下の制約が適用されます。

- シンプルなプライマリキーの場合、最初の属性値 (パーティションキー) の最大長は 2,048 バイトです。
- 複合プライマリキーの場合、2 番目の属性値 (ソートキー) の最大長は 1,024 バイトです。

DynamoDB は、基礎となる UTF-8 文字列エンコードのバイトを使用して文字列を照合し、比較します。たとえば、「a」 (0x61) は「A」 (0x41) より大きく、「¿」 (0xC2BF) は 「z」 (0x7A) より大きいです。

文字列データ型を使用して、日付またはタイムスタンプを表すことができます。これを行う 1 つの方法は、これらの例に示すように、ISO 8601 文字列を使用することです。

- `2016-02-15`
- `2015-12-21T17:42:34Z`
- `20150311T122706Z`

詳細については、http://en.wikipedia.org/wiki/ISO_8601 を参照してください。

> **Note**
>
> 従来のリレーショナルデータベースとは異なり、DynamoDB は日付と時刻のデータ型をネイティブにサポートしていません。代わりに、Unix エポック時間を使用して、日付と時刻のデータを数値データ型として保存すると便利です。

バイナリ

バイナリ型の属性には、圧縮テキスト、暗号化データ、イメージなど、任意のバイナリデータが保存されます。DynamoDB は、バイナリ値を比較するたびに、各バイナリデータを符号なしとして扱います。

属性がインデックスまたはテーブルのキーとして使用されず、DynamoDB 項目の最大サイズ上限が 400 KB に制約されている場合、バイナリ属性の長さは 0 になります。

プライマリキー属性をバイナリ型属性として定義する場合、以下の制約がさらに適用されます。

- シンプルなプライマリキーの場合、最初の属性値 (パーティションキー) の最大長は 2,048 バイトです。
- 複合プライマリキーの場合、2 番目の属性値 (ソートキー) の最大長は 1,024 バイトです。

アプリケーションは、DynamoDB に送信する前に、base64 エンコード形式のバイナリ値をエンコードする必要があります。DynamoDB は、これらの値を受信すると、データを符号なしバイト配列にデコードし、それをバイナリ属性の長さとして使用します。

次に示しているのはバイナリ属性の例であり、Base64 でエンコードされたテキストを使用しています。

```
dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk
```

ブール値

ブール型の属性には、`true` または `false` が格納されます。

Null

Null は不明または未定義の状態の属性を表します。

ドキュメント型

ドキュメント型は、リストとマップです。これらのデータ型は、相互に入れ子にして、最大 32 レベルの深さまで複雑なデータ構造を表すことができます。

値を含む項目が DynamoDB のサイズ制限 (400 KB) 内である限り、リストまたはマップの値の最大数の制限はありません。

属性値がテーブルまたはインデックスキーに使用されていない場合、空の文字列または空のバイナリ値にすることができます。属性値は空の文字列または空のセット (文字列セット、数値セット、またはバイナリセット) にすることはできませんが、空のリストとマップは許可されます。リストとマップ内では、空の文字列とバイナリ値が許可されます。詳細については、「属性」を参照してください。

リスト

リスト型の属性には、順序付きの値のコレクションを保存できます。リストは角括弧で囲まれます: `[ ... ]`

リストは JSON 配列に似ています。リスト要素に保存できるデータ型に制限はなく、リスト要素の要素が同じ型である必要はありません。

次の例は、2 つの文字列と数が含まれるリストを示します。

```
FavoriteThings: ["Cookies", "Coffee", 3.14159]
```

> **Note**
>
> DynamoDB では、要素が深い入れ子になっていても、リスト内の個々の要素を操作できます。詳細については、「DynamoDB での式の使用」を参照してください。

マップ

マップ型属性は、順序なしの名前と値のペアのコレクションを保存できます。マップは中括弧で囲まれます: `{ ... }`

マップは、JSON オブジェクトと同様です。マップの要素に保存できるデータ型に制限はなく、マップの要素が同じ型である必要はありません。

マップは DynamoDB で JSON ドキュメントを保存するのに最適です。以下の例は、文字列、数値、別のマップを含む入れ子のリストを要素とするマップを示しています。

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity : 3},
            Pencils: { Quantity : 2},
            Erasers: { Quantity : 1}
        }
    ]
}
```

> **Note**
>
> DynamoDB では、要素が深い入れ子になっていても、マップ内の個々の要素を操作できます。詳細については、「DynamoDB での式の使用」を参照してください。

セット

DynamoDB は、数値、文字列、またはバイナリ値のセットを表す型をサポートします。セット内の要素はすべて、同じ型である必要があります。例えば、数値セットには数値のみを含めることができ、文字列セットには文字列のみを含めることができます。

値を含む項目が DynamoDB のサイズ制限 (400 KB) 内である限り、セットの値の最大数の制限はありません。

設定内の各値は一意である必要があります。設定内の値の順序は保持されません。したがって、アプリケーションは、設定内の要素の特定の順序に依存することはできません。DynamoDB は空のセットをサポートしていませんが、セット内の空の文字列とバイナリ値は許可されます。

次の例では、文字列セット、設定された numbet とバイナリ設定を示します:

```
["Black", "Green", "Red"]

[42.2, -19, 7.5, 3.14]
```

```
["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
```

データ型記述子

低レベルの DynamoDB API プロトコルは、各属性を解釈する方法を DynamoDB に伝えるトークンとして、データ型記述子を使用します。

DynamoDB データ型記述子の一覧を次に示します。

- **S** — 文字列
- **N** — 数値
- **B** — バイナリ
- **BOOL** — ブール
- **NULL** — Null
- **M** — マップ
- **L** — リスト
- **SS** — 文字列セット
- **NS** — 数値セット
- **BS** — バイナリセット

DynamoDB テーブルクラス

DynamoDB には、コストの最適化に役立つように設計された 2 つのテーブルクラスが用意されています。DynamoDB 標準テーブルクラスがデフォルトで、大半のワークロードで推奨されています。DynamoDB Standard-Infrequent Access (DynamoDB 標準-IA) テーブルクラスは、ストレージが主要なコストとなるテーブル用に最適化されています。例えば、アプリケーションログ、古いソーシャルメディアの投稿、e コマースの注文履歴、過去のゲーム実績など、アクセス頻度の低いデータを格納するテーブルは、標準-IA テーブルクラスの候補として適しています。料金の詳細については、「Amazon DynamoDB の料金表」を参照してください。

すべての DynamoDB テーブルは、テーブルクラス (デフォルトでは DynamoDB Standard) に関連付けられます 。テーブルに関連付けられたすべてのセカンダリインデックスは、同じテーブルクラスを使用します。各テーブルクラスで、データストレージと読み込み/書き込みリクエストに対して異なる料金が適用されます。ストレージとスループットの使用パターンに基づいて、テーブルに対して最も費用対効果の高いテーブルクラスを選択できます。

テーブルクラスの選択は永続的ではありません。この設定は、AWS マネジメントコンソール、AWS CLI、または AWS SDK を使用して変更できます。DynamoDB は、シングルリージョンテーブルとグローバルテーブルに対して AWS CloudFormation を使用したテーブルクラスの管理もサポートしています。テーブルクラスの選択の詳細については、「DynamoDB でテーブルクラスを選択する場合の考慮事項」を参照してください。

DynamoDB におけるパーティションとデータ分散

Amazon DynamoDB は、データをパーティションに保存します。パーティションは、ソリッドステートドライブ (SSD) によってバックアップされ、AWS リージョン内の複数のアベイラビリティーゾーン間で自動的にレプリケートされる、テーブル用のストレージの割り当てです。パーティション管理は DynamoDB によって完全に処理されます。パーティションをご自身が管理する必要はありません。

テーブルを作成するときに、テーブルの最初のステータスは `CREATING` になります。このフェーズの間に、DynamoDB はテーブルに十分なパーティションを割り当て、プロビジョンされたスループット要件に対応できるようにします。テーブルのステータスが `ACTIVE` に変わったらテーブルデータの書き込みと読み取りを開始できます。

DynamoDB は次の状況でテーブルに追加のパーティションを割り当てます。

- テーブルのプロビジョニングされたスループット設定を、既存のパーティションがサポートできる以上に増やした。
- 既存のパーティションが容量いっぱいになり、より多くのストレージ領域が必要になった。

パーティション管理は自動的にバックグラウンドで自動的に発生し、アプリケーションに対して透過的です。テーブルは利用可能な状態のままで、プロビジョニングされたスループット要件を完全にサポートします。

詳細については、パーティションキーの設計を参照してください。

DynamoDB のグローバルセカンダリインデックスもパーティションで構成されます。グローバルセカンダリインデックスのデータは、基本テーブルのデータとは別に保存されますが、インデックスパーティションはテーブルパーティションと同様に動作します。

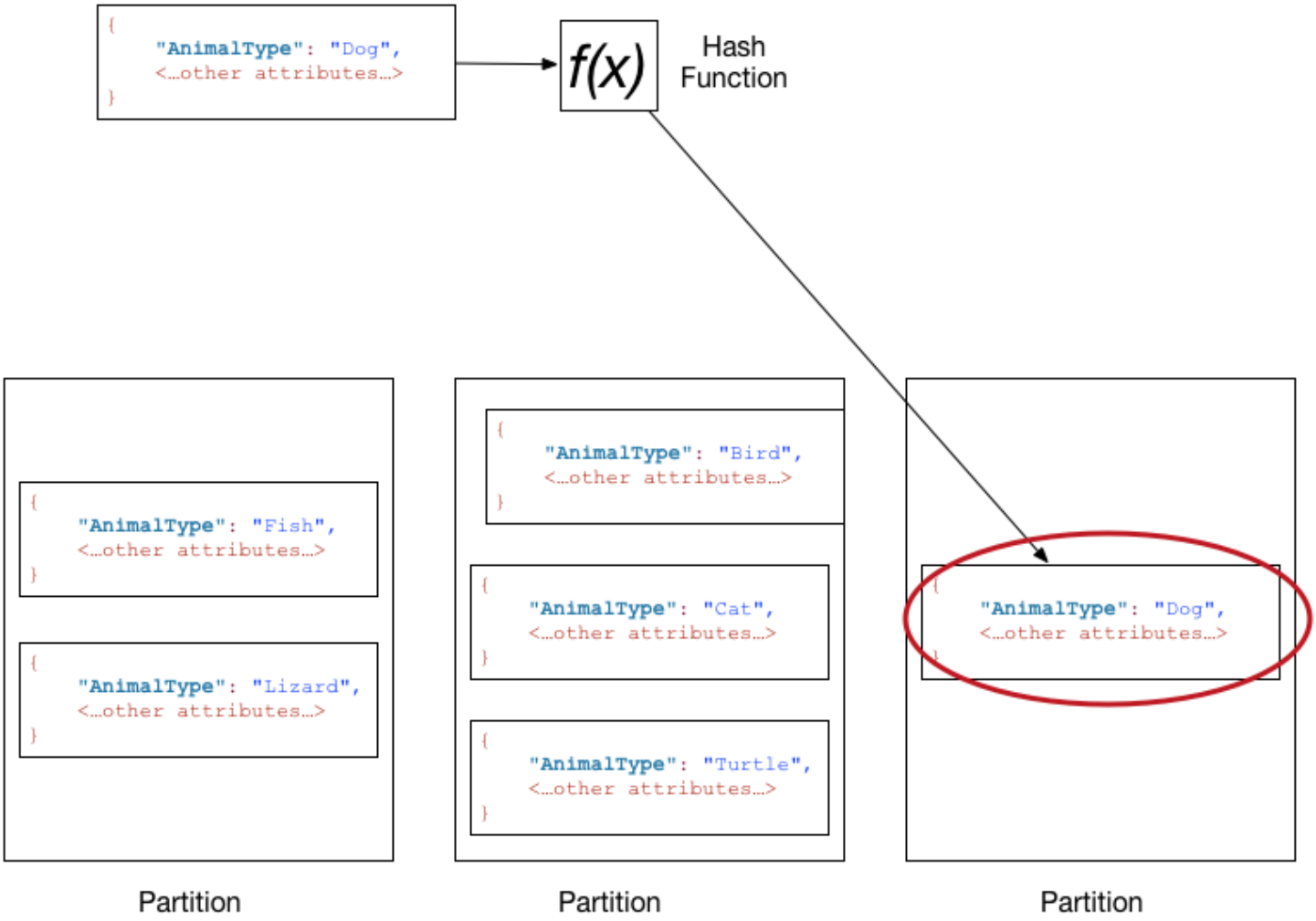
データ分散: パーティションキー

テーブルにシンプルなプライマリキー (パーティションキーのみ) がある場合、DynamoDB はパーティションキーバリューに基づいて、各項目を保存および取得します。

DynamoDB は項目をテーブルに書き込むため、パーティションキーバリューを内部ハッシュ関数への入力として使用します。ハッシュ関数からの出力値によって、項目が保存されるパーティションが決まります。

テーブルから項目を読み込むには、項目のパーティションキーバリューを指定する必要があります。DynamoDB はこの値をハッシュ関数への入力として使用し、項目が見つかるパーティションを提供します。

次の図は、複数のパーティションにまたがる Pets という名前のテーブルを示しています。テーブルのプライマリキーは AnimalType (このキー属性のみが表示されます。) DynamoDB は、ハッシュ関数を使用して、新しい項目を保存する場所を決定します。この場合は、文字列 Dog のハッシュ値に基づいています。項目はソート順に保存されないことに注意してください。各アイテムの場所は、そのパーティションキーのハッシュ値によって決まります。



> **Note**
>
> DynamoDB は、パーティション数にかかわらず、項目がテーブルのパーティション全体に渡って均一にディストリビューションされている状態に対して最適化されています。テーブルの項目数に対して大きな個別の値を持つことができるパーティションキーを選択することをお勧めします。

データ分散: パーティションキーおよびソートキー

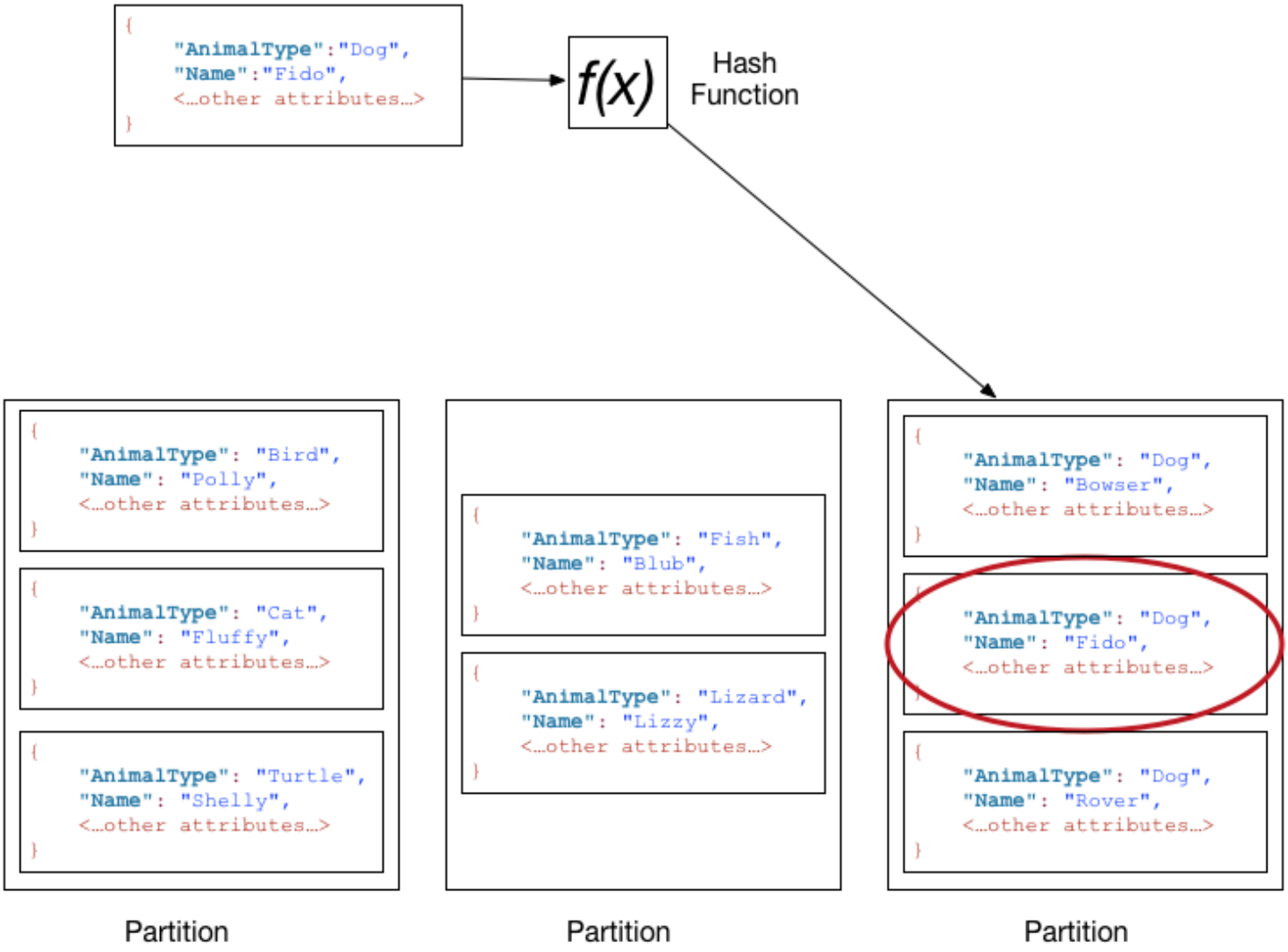
テーブルに複合プライマリキー (パーティションキーとソートキー) がある場合、DynamoDB は データ分散: パーティションキー で説明したのと同じ方法でパーティションキーのハッシュ値を計算します。ただし、パーティションキーの値が同じ項目は互いに近く、ソートキー属性の値によってソートされた順序になる傾向があります。パーティションキー属性の値が同じ項目のセットは、項目コレクションと呼ばれます。項目コレクションは、コレクション内の項目の範囲を効率的に取得できるように最適化されています。テーブルにローカルセカンダリインデックスがない場合、DynamoDB は、データを保存し、読み取りと書き込みのスループットを実現するために必要な数のパーティションに自動的に項目コレクションを分割します。

テーブルに項目を書き込むため、DynamoDB はパーティションキーのハッシュ値を計算し、項目を含めるパーティションを決定します。そのパーティションでは、いくつかの項目で同じパーティションキー値を持つことができます。そのため、DynamoDB には、同じパーティションキーを持つ他の項目とソートキーの昇順で項目が保存されます。

テーブルから項目を読み込むには、パーティションのキーバリューとソートキーのキーバリューを指定する必要があります。DynamoDB は、パーティションキーのハッシュ値を計算し、項目が見つかるパーティションを提供します。

目的の項目に同じパーティションキーバリューがある場合、単一のオペレーション (`Query`) でテーブルから複数の項目を読み取ることができます。DynamoDB は、そのパーティションのキーバリューを持つすべての項目を返します。オプションでソートキーに条件を適用し、特定範囲内の値を持つ項目だけを返すことができます。

Pets テーブルに、AnimalType (パーティションキー) と Name (ソートキー) で構成される複合プライマリキーがあるとします。次の図表は、DynamoDB がパーティションのキーバリューが Dog でソートキー値が Fido の項目を書き込んでいるところを示しています。



Pets テーブルから同じ項目を読み込むために、DynamoDB は、ハッシュ値 Dog を計算し、これらの項目が保存されたパーティションを提供します。次に、DynamoDB は、Fido が見つかるまで、ソートキーの属性値をスキャンします。

AnimalType が Dog のすべての項目を読み込むには、ソートキーの条件を指定しないで `Query` オペレーションを発行できます。デフォルトでは、項目は保存されている順序 (つまり、ソートキーによって昇順でソート) で返されます。オプションで、代わりに降順をリクエストできます。

一部の Dog 項目のみをクエリするには、ソートキーに条件を適用できます (たとえば、Name が A から K の範囲内の文字で始まる Dog 項目のみ)。

> **Note**
>
> DynamoDB テーブルには、パーティションキーバリューごとに個別のソートキーバリューの数に上限はありません。何十億もの Dog 項目を Pets テーブルに保存する必要がある場合、DynamoDB はこの要件を自動的に処理するのに十分なストレージを割り当てます。

SQL から NoSQL に移行する方法について学ぶ

アプリケーションデベロッパーなら、リレーショナルデータベース管理システム (RDBMS) および構造化クエリ言語 (SQL) を使用した経験があるかもしれません。Amazon DynamoDB を使い始めると、多くの類似点があると同時に、異なる点も多くあることに気づきます。NoSQL は、可用性が高く、スケーラブルで、高パフォーマンス用に最適化された、非リレーショナルデータベースシステムについて説明するのに使用される用語です。NoSQL データベース (DynamoDB など) は、リレーショナルモデルの代わりに、キーバリューのペアやドキュメントストレージなど、データ管理のための代替モデルを使用します。詳細については、「NoSQL とは」を参照してください。

Amazon DynamoDB は PartiQL をサポートしています。PartiQL は、オープンソースの SQL 互換のクエリ言語で、データの格納場所や形式に関係なく、データへの効率的なクエリの実行が簡単に行えます。PartiQL を使用すると、リレーショナルデータベースの構造化データ、オープンデータ形式の半構造化データおよびネストされたデータ、さらには行ごとに異なる属性を許可する NoSQL またはドキュメントデータベース内のスキーマレスデータを簡単に処理できます。詳細については、「PartiQL Query Language」(PartiQL クエリ言語) を参照してください。

次のセクションでは、SQL ステートメントを同等の DynamoDB オペレーションと比較および対比しながら、一般的なデータベースタスクについて説明します。

> **Note**
>
> このセクションの SQL の例は、MySQL の RDBMS と互換性があります。
> このセクションの DynamoDB 例では、JSON 形式のオペレーションのパラメータと共に DynamoDB オペレーションの名前を表示します。

トピック

- リレーショナル (SQL) または NoSQL の選択
- リレーショナル (SQL) データベースと DynamoDB へのアクセスの相違点

- テーブル作成時のリレーショナル (SQL) データベースと DynamoDB の相違点
- リレーショナル (SQL) データベースと DynamoDB からテーブル情報を取得するときの相違点
- テーブルにデータを書き込むときのリレーショナル (SQL) データベースと DynamoDB の相違点
- テーブルからデータを読み込むときの、リレーショナル (SQL) データベースと DynamoDB の相違点
- インデックスを管理するときのリレーショナル (SQL) データベースと DynamoDB の相違点
- テーブル内のデータを変更するときのリレーショナル (SQL) データベースと DynamoDB の相違点
- テーブルからデータを削除するときのリレーショナル (SQL) データベースと DynamoDB の相違点
- テーブルを削除するときのリレーショナル (SQL) データベースと DynamoDB の相違点

リレーショナル (SQL) または NoSQL の選択

現在のアプリケーションには今までになく厳しい要件があります。たとえば、あるオンラインゲームを、小数のユーザーおよび非常に小さいデータ量で開始するかもしれません。しかし、ゲームが成功すれば、それは基盤となるデータベース管理システムのリソースを簡単に上回ります。ウェブベースのアプリケーションに、数百、数千、数百万の同時ユーザーがいて、テラバイトあるいはそれ以上の新しいデータが毎日生成される、というのはよくあることです。そのようなアプリケーションのデータベースの場合は、1 秒あたり数万 (あるいは数十万) の読み取り/書き込みの処理が必要です。

Amazon DynamoDB は、これらのワークロードに適しています。デベロッパーとして、アプリケーションを小さく開始し、人気が出るにつれて徐々に増加させることができます。DynamoDB は、大量のデータや多数のユーザーの処理をシームレスにスケーリングします。

従来のリレーショナルデータベースモデリングと DynamoDB に適応させる方法の詳細については、「DynamoDB でリレーショナルデータをモデル化するためのベストプラクティス」を参照してください。

次の表に、リレーショナルデータベース管理システム (RDBMS) と DynamoDB の高度な相違点を示します。

特性	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
最適なワークロード	アドホッククエリ、データウェアハウス、OLAP (オンライン分析処理)。	ソーシャルネットワーク、ゲーム、メディア共有、Internet of Things (IoT) を含む、

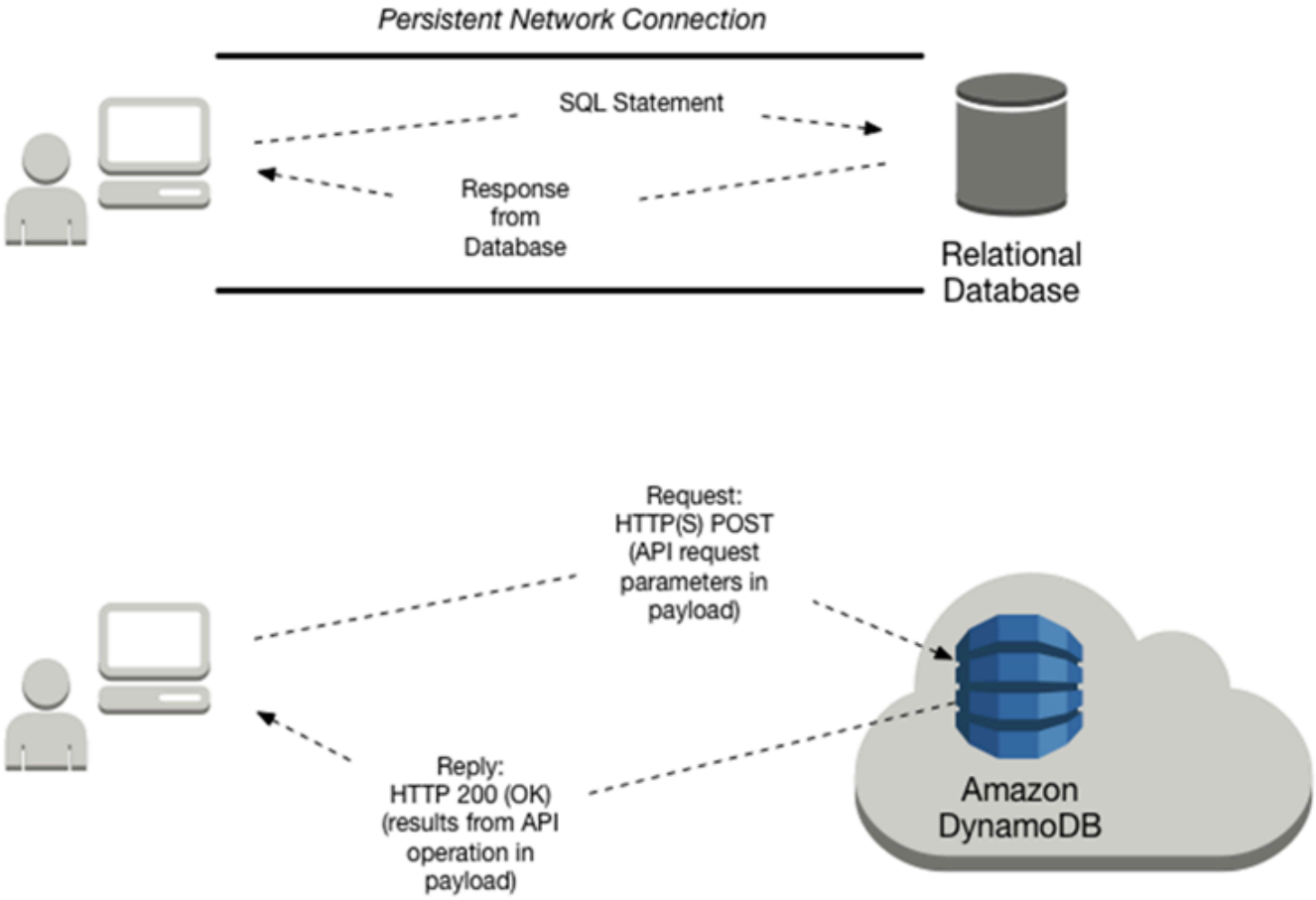
特性	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
		ウェブスケールアプリケーションです。
データモデル	リレーショナルモデルには、データを、テーブル、行、列に正規化する、明確に定義されたスキーマが必要です。さらに、関係のすべては、テーブル、列、インデックスおよび他のデータベースの工レメント間で定義されます。	DynamoDB はスキーマレスです。すべてのテーブルに、各データ項目を一意に識別するプライマリキーが必要ですが、他の非キー属性のような制約はありません。DynamoDB では、JSON ドキュメントを含む構造化データまたは半構造化データを管理できます。
データアクセス	SQL は、データを保存および取得するための基準です。リレーショナルデータベースにはデータベース駆動型アプリケーションの開発を簡素化するためのツールが豊富ですが、これらのツールはすべて、SQL を使用します。	AWS マネジメントコンソール、AWS CLI、または NoSQL WorkBench を使用して DynamoDB を操作し、アドホックタスクを実行できます。PartiQL は SQL 互換のクエリ言語であり、DynamoDB でデータの選択、挿入、更新、および削除を行うことができます。アプリケーションでは、オブジェクトベースでドキュメント中心、または低レベルのインターフェイスを使用しながら、AWS ソフトウェア開発キット (SDK) を活用し、DynamoDB を操作できます。

特性	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
パフォーマンス	リレーショナルデータベースは、ストレージに最適化されていますので、パフォーマンスは通常、ディスクサブシステムによって異なります。デベロッパーとデータベース管理者は、ピークパフォーマンスを達成するために、クエリ、インデックスおよびテーブルの構造を最適化する必要があります。	DynamoDB は、コンピューティングに最適化されていますので、パフォーマンスは主に、基盤となるハードウェアとネットワークレイテンシーの機能です。マネージドサービスとして、DynamoDB は、これらの実装の詳細からアプリケーションを隔離し、堅牢で高パフォーマンスなアプリケーションの設計と構築に集中できるようにします。
スケーリング	より高速なハードウェアでスケールアップするのが最も簡単です。データベーステーブルが分散システムの複数のホストにまたがることは可能ですが、この場合、追加の投資が必要です。リレーショナルデータベースは、スケーラビリティに上限を課すファイルの数とサイズが最大サイズです。	DynamoDB は、ハードウェアの分散型クラスターを使用してスケールアウトできるように設計されています。この設計により、レイテンシーの増加なしでスループットを強化することができます。顧客がスループット要件を指定すると、DynamoDB は、その要件に対応するために十分なリソースを割り当てます。テーブル単位の項目数、およびそのテーブルの合計サイズに上限はありません。

リレーショナル (SQL) データベースと DynamoDB へのアクセスの相違点

アプリケーションがデータベースにアクセスする前に、アプリケーションがデータベースの使用を許可されるように認証する必要があります。アプリケーションで、許可されているアクションのみ実行できるように、許可する必要があります。

次の図表は、クライアントのリレーショナルデータベースおよび Amazon DynamoDB とのやり取りを示します。



次の表にクライアントのやり取りのタスクについての詳細があります。

特徴	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
データベースにアクセスするためのツール	ほとんどのリレーショナルデータベースは、コマンドラインインターフェイス (CLI) を提供しており、アドホックな SQL ステートメントを入力して、結果をすぐに見ることができます。	ほとんどの場合、アプリケーションコードを書き込みます。AWS マネジメントコンソール、AWS Command Line Interface (AWS CLI)、または NoSQL Workbench を使用して、アドホックリクエストを DynamoDB に送信し、結果を表示することもできま

特徴	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
		す。PartiQL は SQL 互換のクエリ言語であり、DynamoDB でデータの選択、挿入、更新、および削除を行うことができます。
データベースに接続	アプリケーションプログラムは、データベースを使用したネットワーク接続を確立し、維持します。アプリケーションが終了すると、接続を終了します。	DynamoDB は、ウェブサービスで、その操作はステートレスです。アプリケーションは永続的ネットワーク接続を維持する必要はありません。代わりに、DynamoDB の操作は HTTP(S) リクエストおよびレスポンスを使用して行われます。
認証	アプリケーションが認証されるまでデータベースに接続できません。RDBMS は認証自体を実行できますし、ホストのオペレーティングシステムやディレクトリサービスにこのタスクをオフロードすることもできます。	DynamoDB に対するすべてのリクエストは、その特定のリクエストを認証する暗号署名と共に使用しなければなりません。AWS SDK は、署名の作成とリクエストの署名に必要なすべての論理を提供します。詳細については、「AWS 全般のリファレンス.」の「AWS API リクエストの署名」を参照してください。

特徴	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
Authorization	アプリケーションは承認されたアクションのみ実行できます。データベース管理者またはアプリケーション所有者は、SQL GRANT および REVOKE ステートメントを使用して、データベースオブジェクト (テーブルなど)、データ (テーブル内の行など)、特定の SQL ステートメントを発行する機能へのアクセスを制御できます。	DynamoDB では、AWS Identity and Access Management (IAM) が承認を行います。DynamoDB リソース (テーブルなど) へのアクセス許可を付与する IAM ポリシーを記述し、ユーザーとロールがそのポリシーを使用できるようにします。IAM では、DynamoDB テーブルの個々のデータ項目に対する詳細なアクセス制御といった特徴も備えています。詳細については、「Amazon DynamoDB の Identity and Access Management」を参照してください。
リクエストを送信	アプリケーションは、実行するすべてのデータベース操作に対する SQL ステートメントを発行します。SQL ステートメントを受信すると、RDBMS は構文を確認し、オペレーションを実行するための計画を作成してから、計画を実行します。	アプリケーションは、HTTP(S) リクエストを DynamoDB に送信します。リクエストには、パラメータとともに、実行する DynamoDB オペレーションの名前が含まれます。DynamoDB はリクエストを直ちに実行します。

特徴	リレーショナルデータベース管理システム (RDBMS)	Amazon DynamoDB
レスポンスを受信	RDBMS は SQL ステートメントから結果を返します。エラーがある場合は、RDBMS はエラー状況とエラーメッセージを返します。	DynamoDB は、オペレーションの結果を含む HTTP(S) レスポンスを返します。エラーがあると、DynamoDB は、HTTP エラー状況およびエラーメッセージを返します。

テーブル作成時のリレーショナル (SQL) データベースと DynamoDB の相違点

テーブルは、リレーショナルデータベースおよび Amazon DynamoDB の基本的なデータ構造です。リレーショナルデータベース管理システム (RDBMS) では、作成時に、テーブルのスキーマを定義する必要があります。対照的に、DynamoDB テーブルはスキーマレスであるため、プライマリキーを除いて、テーブルを作成する際に、追加の属性やデータ型を定義する必要はありません。

次のセクションでは、SQL を使用してテーブルを作成する方法と、DynamoDB を使用してテーブルを作成する方法を比較します。

トピック

- SQL を使用してテーブルを作成する
- DynamoDB を使用してテーブルを作成する

SQL を使用してテーブルを作成する

SQL では、次の例に示すように、CREATE TABLE ステートメントを使用して、テーブルを作成します。

```
CREATE TABLE Music (
    Artist VARCHAR(20) NOT NULL,
    SongTitle VARCHAR(30) NOT NULL,
    AlbumTitle VARCHAR(25),
    Year INT,
    Price FLOAT,
```

```
    Genre VARCHAR(10),
    Tags TEXT,
    PRIMARY KEY(Artist, SongTitle)
);
```

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。

テーブルの列とデータ型すべて、およびテーブルのプライマリキーを定義する必要があります。(これらの定義は、ALTER TABLE ステートメントを使用して、必要に応じて後で変更することができます。)

多くの SQL 実装では、CREATE TABLE ステートメントの一部として、テーブルのストレージ仕様を定義することができます。他に明記されていない限り、テーブルはデフォルトのストレージ設定で作成されます。本稼働環境では、データベース管理者は最適なストレージパラメータを特定することもできます。

DynamoDB を使用してテーブルを作成する

CreateTable オペレーションを使用してプロビジョニングモードのテーブルを作成し、次に示すように、パラメータを指定します。

```
{
    TableName : "Music",
    KeySchema: [
        {
            AttributeName: "Artist",
            KeyType: "HASH" //Partition key
        },
        {
            AttributeName: "SongTitle",
            KeyType: "RANGE" //Sort key
        }
    ],
    AttributeDefinitions: [
        {
            AttributeName: "Artist",
            AttributeType: "S"
        },
        {
            AttributeName: "SongTitle",
            AttributeType: "S"
        }
```

```
    ],
    ProvisionedThroughput: {       // Only specified if using provisioned mode
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
}
```

このテーブルのプライマリキーは、Artist (パーティションキー) および SongTitle (ソートキー) で構成されています。

`CreateTable` に以下のパラメータを提供する必要があります。

- `TableName` – テーブルの名前。
- `KeySchema` – プライマリキーに使用する属性。詳細については、「テーブル、項目、属性」および「プライマリキー」を参照してください。
- `AttributeDefinitions` – キースキーマ属性のデータ型。
- `ProvisionedThroughput (for provisioned tables)` – このテーブルに必要な 1 秒あたりの読み取り/書き込み数。DynamoDB は、スループット要件を常に満たすように、十分なストレージとシステムリソースを確保しています。これらは、`UpdateTable` オペレーションを使用し、必要に応じて後で変更できます。DynamoDB がストレージ割り当てを完全に管理しているため、テーブルのストレージ要件を指定する必要はありません。

リレーショナル (SQL) データベースと DynamoDB からテーブル情報を取得するときの相違点

テーブルが仕様に従って作成されたことを確認できます。リレーショナルデータベースでは、テーブルのスキーマがすべて表示されます。Amazon DynamoDB テーブルはスキーマレスであるため、プライマリキー属性のみが表示されます。

トピック

- SQL を使ってテーブルに関する情報を取得する
- DynamoDB でテーブルに関する情報を取得する

SQL を使ってテーブルに関する情報を取得する

ほとんどのリレーショナルデータベース管理システム (RDBMS) では、テーブルの構造 (列、データ型、プライマリキー定義など) を記述できます。SQL にはこれを行うための標準的な方法はありませ

ん。ただし、データベースシステムの多くが DESCRIBE コマンドを提供しています。MySQL の例を以下に示します。

```
DESCRIBE Music;
```

すべての列名、データ型、サイズを含むテーブルの構造が返ります。

```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| Artist     | varchar(20) | NO   | PRI | NULL    |       |
| SongTitle  | varchar(30) | NO   | PRI | NULL    |       |
| AlbumTitle | varchar(25) | YES  |     | NULL    |       |
| Year       | int(11)     | YES  |     | NULL    |       |
| Price      | float       | YES  |     | NULL    |       |
| Genre      | varchar(10) | YES  |     | NULL    |       |
| Tags       | text        | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。

DynamoDB でテーブルに関する情報を取得する

DynamoDB には、同様の DescribeTable オペレーションがあります。パラメータは、テーブル名のみです。

```
{
    TableName : "Music"
}
```

DescribeTable からの返信は、次のようになります。

```
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "Artist",
        "AttributeType": "S"
      },
      {
```

```
        "AttributeName": "SongTitle",
        "AttributeType": "S"
      }
    ],
    "TableName": "Music",
    "KeySchema": [
      {
        "AttributeName": "Artist",
        "KeyType": "HASH"  //Partition key
      },
      {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"  //Sort key
      }
    ],

    ...
```

DescribeTable は、テーブルのインデックス、プロビジョニングされたスループット設定、概算項目数、その他メタデータに関する情報を返します。

テーブルにデータを書き込むときのリレーショナル (SQL) データベースと DynamoDB の相違点

リレーショナルデータベースのテーブルには、データの行が含まれます。行は列で構成されます。Amazon DynamoDB テーブルには、項目が含まれています。項目は属性で構成されます。

このセクションでは、テーブルに 1 つの行 (または項目) を書き込む方法を説明します。

トピック

- SQL を使ってテーブルにデータを書き込む
- DynamoDB のテーブルにデータを書き込む

SQL を使ってテーブルにデータを書き込む

リレーショナルデータベースのテーブルは、行と列で構成される、2 つのディメンションのデータ構造です。一部のデータベース管理システムは、通常、ネイティブ JSON または XML データ型を使用する半構造化データのサポートを提供します。ただし、実装の詳細はベンダー間で変わります。

SQL では、INSERT ステートメントを使用して、テーブルに行を追加します。

```
INSERT INTO Music
    (Artist, SongTitle, AlbumTitle,
    Year, Price, Genre,
    Tags)
VALUES(
    'No One You Know', 'Call Me Today', 'Somewhat Famous',
    2015, 2.14, 'Country',
    '{"Composers": ["Smith", "Jones", "Davis"],"LengthInSeconds": 214}'
);
```

このテーブルのプライマリキーは、Artist と SongTitle で構成されます。これらの列の値を指定する必要があります。

> **Note**
>
> この例では、Tags 列を使用して、Music テーブル内の曲に関する半構造化データを保存します。Tags 列は TEXT 型として定義され、MySQL で最大 65,535 文字を保存できます。

DynamoDB のテーブルにデータを書き込む

Amazon DynamoDB では、DynamoDB API または PartiQL (SQL 互換のクエリ言語) を使用して、テーブルに項目を追加できます。

DynamoDB API

DynamoDB API では、`PutItem` オペレーションを使用して、テーブルに項目を追加します。

```
{
    TableName: "Music",
    Item: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today",
        "AlbumTitle":"Somewhat Famous",
        "Year": 2015,
        "Price": 2.14,
        "Genre": "Country",
        "Tags": {
            "Composers": [
                  "Smith",
                  "Jones",
```

```
                    "Davis"
                ],
                "LengthInSeconds": 214
            }
        }
}
```

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。これらの属性の値を指定する必要があります。

この PutItem 例に関する主要事項を示します。

- DynamoDB では、JSON を使用して、ドキュメントのネイティブサポートを提供しています。これにより DynamoDB は Tags などの半構造化データを保存する場合に最適になります。また、JSON ドキュメント内からデータを取得および操作できます。
- Music テーブルには、プライマリキー (Artist および SongTitle) 以外には事前定義された属性はありません。
- ほとんどの SQL データベースはトランザクション指向です。INSERT ステートメントを発行すると、データ変更は、COMMIT ステートメントを発行するまで永続的ではありません。Amazon DynamoDB によって、PutItem オペレーションの効果は、DynamoDB が HTTP 200 ステータスコード (OK) で応答する場合、永続的になります。

次に、PutItem の他の例をいくつか示します。

```
{
    TableName: "Music",
    Item: {
        "Artist": "No One You Know",
        "SongTitle": "My Dog Spot",
        "AlbumTitle":"Hey Now",
        "Price": 1.98,
        "Genre": "Country",
        "CriticRating": 8.4
    }
}
```

```
{
    TableName: "Music",
    Item: {
```

```
        "Artist": "No One You Know",
        "SongTitle": "Somewhere Down The Road",
        "AlbumTitle":"Somewhat Famous",
        "Genre": "Country",
        "CriticRating": 8.4,
        "Year": 1984
    }
}
```

```
{
    TableName: "Music",
    Item: {
        "Artist": "The Acme Band",
        "SongTitle": "Still In Love",
        "AlbumTitle":"The Buck Starts Here",
        "Price": 2.47,
        "Genre": "Rock",
        "PromotionInfo": {
            "RadioStationsPlaying":[
                 "KHCR", "KBQX", "WTNR", "WJJH"
            ],
            "TourDates": {
                "Seattle": "20150625",
                "Cleveland": "20150630"
            },
            "Rotation": "Heavy"
        }
    }
}
```

```
{
    TableName: "Music",
    Item: {
        "Artist": "The Acme Band",
        "SongTitle": "Look Out, World",
        "AlbumTitle":"The Buck Starts Here",
        "Price": 0.99,
        "Genre": "Rock"
    }
}
```

> **Note**
>
> PutItem に加えて、DynamoDB は、同時に複数の項目に書き込むための BatchWriteItem オペレーションもサポートします。

PartiQL for DynamoDB

PartiQL では、PartiQL ExecuteStatement ステートメントを利用する Insert オペレーションを使用して、テーブルに項目を追加します。

```
INSERT into Music value {
    'Artist': 'No One You Know',
    'SongTitle': 'Call Me Today',
    'AlbumTitle': 'Somewhat Famous',
    'Year' : '2015',
    'Genre' : 'Acme'
}
```

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。これらの属性の値を指定する必要があります。

> **Note**
>
> Insert と ExecuteStatement を使用したコード例については、DynamoDB 用の PartiQL 挿入ステートメント を参照してください。

テーブルからデータを読み込むときの、リレーショナル (SQL) データベースと DynamoDB の相違点

SQL では、SELECT ステートメントを使用して、テーブルから 1 つ以上の行を取得します。WHERE 句を使用して、返されたデータを調べます。

これは、Amazon DynamoDB を使用する場合とは異なります。Amazon DynamoDb は、データの読み取りに次のオペレーションを提供します。

- `ExecuteStatement` がテーブルから 1 つまたは複数の項目を取得します。`BatchExecuteStatement` が 1 回のオペレーションで異なるテーブルから複数の項目を取得します。どちらのオペレーションでも、SQL 互換のクエリ言語である PartiQL が使用されます。
- `GetItem` – テーブルから単一の項目を取り出します。これは、項目の物理的な場所に直接アクセスできるため、単一の項目を読み込むうえで最も効率的な方法です。(DynamoDB では、`BatchGetItem` オペレーションを提供し、1 回のオペレーションで最大 100 回の `GetItem` コールを実行できます。)
- `Query` – 特定のパーティションキーがあるすべての項目を取得します。これらの項目内では、ソートキーに条件を適用し、データのサブセットのみ取得できます。`Query` は、データが格納されているパーティションに、すばやく効率的にアクセスできます。(詳しくは、DynamoDB におけるパーティションとデータ分散 を参照してください)。
- `Scan` – 指定されたテーブルで、すべての項目を取得します。(大量のリソースシステムを消費するため、このオペレーションは大きなテーブルでは使用しないでください。)

> **Note**
>
> リレーショナルデータベースでは、SELECT ステートメントを使用して、複数のテーブルからデータを結合し、その結果を返すことができます。結合は、リレーショナルモデルにおいて必須です。結合を効率的に実行するために、データベースおよびアプリケーションは、継続的にパフォーマンスを調整する必要があります。DynamoDB は、テーブル結合をサポートしない非リレーショナル NoSQL データベースです。その代わり、アプリケーションは一度に 1 つのテーブルからデータを読み込みます。

次のセクションでは、データを読み込むための異なるユースケースおよびこれらのタスクをリレーショナルデータベースと DynamoDB を使って実行する方法を説明します。

トピック

- プライマリキーを使用して項目を読み込むときの相違点
- テーブルのクエリにおける相違点
- テーブルのスキャンにおける相違点

プライマリキーを使用して項目を読み込むときの相違点

データベースの一般的なアクセスパターンの 1 つは、テーブルから単一項目を読み取ることです。目的の項目のプライマリキーを指定する必要があります。

トピック

- SQL にプライマリキーを使用して項目を読み込む
- DynamoDB でプライマリキーを使用して項目を読み込む

SQL にプライマリキーを使用して項目を読み込む

SQL では、SELECT ステートメントを使用して、テーブルからデータを取得します。結果の 1 つ以上の列 (* オペレーター を使用すれば、すべて) をリクエストできます。WHERE 句は返る行を判別します。

以下は、Music テーブルから単一の行を取得するための SELECT ステートメントです。WHERE 句はプライマリキー値を指定します。

```
SELECT *
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

列のサブセットのみを取得するようにこのクエリを変更できます。

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

このテーブルのプライマリキーが、Artist および SongTitle で構成されていることに注意してください。

DynamoDB でプライマリキーを使用して項目を読み込む

Amazon DynamoDB では、DynamoDB API または PartiQL (SQL 互換のクエリ言語) を使用して、テーブルから項目を読み込むことができます。

DynamoDB API

DynamoDB API では、PutItem オペレーションを使用して、テーブルに項目を追加します。

DynamoDB は、プライマリキーにより項目を取得するための GetItem オペレーションを提供します。GetItem は、項目の物理的な場所への直接アクセスを提供するため非常に効率的です。（詳しくは、DynamoDB におけるパーティションとデータ分散 を参照してください）。

デフォルトでは、GetItem は、すべての属性を含む項目全体を返します。

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    }
}
```

一部の属性のみが返されるように、ProjectionExpression パラメータを追加できます。

```
{
    TableName: "Music",
    Key: {
        "Artist": "No One You Know",
        "SongTitle": "Call Me Today"
    },
    "ProjectionExpression": "AlbumTitle, Year, Price"
}
```

このテーブルのプライマリキーが、Artist および SongTitle で構成されていることに注意してください。

DynamoDB GetItem オペレーションは非常に効率的です。プライマリキー値を使用して、該当する項目の正確な格納場所を特定し、そこから直接取得します。SQL SELECT ステートメントは、プライマリキー値によって項目を取得する場合、同様に効率的です。

SQL SELECT ステートメントは、さまざまな種類のクエリとテーブルスキャンをサポートしています。DynamoDB は、テーブルのクエリにおける相違点 および テーブルのスキャンにおける相違点 で説明されている Query および Scan オペレーションと同様の機能を提供します。

SQL SELECT ステートメントは、テーブルの結合を実行でき、同時に複数のテーブルからデータを取得できます。データベーステーブルが正規化され、テーブル間の関係が明確である場合、結合は最も効果的です。ただし、1 つの SELECTステートメントであまりに多くのテーブルを結合すると、アプリケーションパフォーマンスが影響を受けます。データベースレプリケーション、

マテリアライズドビュー、またはクエリの書き換えを使用して、このような問題を回避できます。

DynamoDB は、非リレーショナルデータベースのため、テーブルの結合はサポートされません。リレーショナルデータベースから既存のアプリケーションを DynamoDB に移行する場合、結合の必要を排除するためデータモデルを非正規化する必要があります。

PartiQL for DynamoDB

PartiQL では、PartiQL `Select` ステートメントを利用する `ExecuteStatement` オペレーションを使用して、テーブルから項目を読みます。

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

このテーブルのプライマリキーが、Artist および SongTitle で構成されていることに注意してください。

> **Note**
>
> 選択 PartiQL ステートメントは、DynamoDB テーブルにクエリやスキャンを実行する場合にも使用できます

`Select` と `ExecuteStatement` を使用したコード例については、DynamoDB 用の PartiQL select ステートメント を参照してください。

テーブルのクエリにおける相違点

もう 1 つの一般的なアクセスパターンは、クエリ条件に基づき、テーブルから複数の項目を読み込むことです。

トピック

- SQL を使用してテーブルのクエリを実行する
- DynamoDB でテーブルのクエリを実行する

SQL を使用してテーブルのクエリを実行する

SQL を使用すると、SELECT ステートメントは、キー列、非キー列、または任意の組み合わせに対してクエリを実行できます。WHERE 句は、次の例に示すように、返る行を決定します。

```
/* Return a single song, by primary key */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today';
```

```
/* Return all of the songs by an artist */

SELECT * FROM Music
WHERE Artist='No One You Know';
```

```
/* Return all of the songs by an artist, matching first part of title */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle LIKE 'Call%';
```

```
/* Return all of the songs by an artist, only if the price is less than 1.00 */

SELECT * FROM Music
WHERE Artist='No One You Know'
AND Price < 1.00;
```

このテーブルのプライマリキーが、Artist および SongTitle で構成されていることに注意してください。

DynamoDB でテーブルのクエリを実行する

Amazon DynamoDB では、DynamoDB API または PartiQL (SQL 互換のクエリ言語) を使用して、テーブルの項目にクエリを実行できます。

DynamoDB API

Amazon DynamoDB では、Query オペレーションを使用すると、同様の方法でデータを取得できます。Query オペレーションは、データが保存されている物理的な場所にすばやく効率的にアクセスすることができます。詳細については、「DynamoDB におけるパーティションとデータ分散」を参照してください。

Query は任意のテーブルまたはセカンダリインデックスでを使用できます。パーティションキーの値に対して等価条件を指定する必要があります。ソートキー属性が定義されている場合は、必要に応じて別の条件を指定できます。

KeyConditionExpression パラメータは、クエリを実行するキー値を指定します。オプションの FilterExpression を使用して、結果が返される前に、そこから特定の項目を削除できます。

DynamoDB では、式パラメータ (KeyConditionExpression や FilterExpression など) で ExpressionAttributeValues をプレースホルダーとして使用する必要があります。これは、ランタイム時に実際の値を SELECT ステートメントに代入する、リレーショナルデータベースでのバインド変数の使用に類似しています。

このテーブルのプライマリキーが、Artist および SongTitle で構成されていることに注意してください。

次に、DynamoDB Query の例をいくつか示します。

```
// Return a single song, by primary key

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and SongTitle = :t",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call Me Today"
    }
}
```

```
// Return all of the songs by an artist

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    ExpressionAttributeValues: {
        ":a": "No One You Know"
    }
}
```

```
// Return all of the songs by an artist, matching first part of title
```

```
{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and begins_with(SongTitle, :t)",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call"
    }
}
```

```
// Return all of the songs by an artist, only if the price is less than 1.00

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a",
    FilterExpression: "Price < :p",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":p": 1.00
    }
}
```

> **Note**
>
> FilterExpression は、Query が一致する項目を読み取った後に適用されるため、消費される読み取り容量は減少しません。可能な場合は、効率的なクエリのために範囲条件でソートキーに対して KeyConditionExpression を使用するようにデータをモデル化します。詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

PartiQL for DynamoDB

PartiQL では、パーティションキーの ExecuteStatement オペレーションと Select ステートメントを使用してクエリを実行できます。

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know'
```

このように SELECT ステートメントを使用すると、この特定の Artist に関連付けられているすべての曲が返されます。

Select と ExecuteStatement を使用したコード例については、DynamoDB 用の PartiQL select ステートメント を参照してください。

テーブルのスキャンにおける相違点

SQL では、SELECT 句なしの WHERE ステートメントは、テーブルのすべての行を返します。Amazon DynamoDB では、Scan オペレーションで同様のアクションを行います。どちらの場合も、すべて、または一部の項目を取得できます。

SQL または NoSQL データベースのいずれを使用するにしても、スキャンは大量のシステムリソースを消費するので、控え目に使用する必要があります。スキャンが適切 (小さなテーブルをスキャンするなど) または不可避 (データの一括エクスポートの実行など) な場合はあります。しかし、一般的なルールとして、スキャンを実行しないようアプリケーションを設計する必要があります。詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

> **Note**
>
> 一括エクスポートを実行すると、パーティションごとに少なくとも 1 つのファイルも作成されます。各ファイル内のすべての項目は、その特定のパーティションのハッシュされたキースペースからのものです。

トピック

- SQL を使用してテーブルをスキャンする
- DynamoDB でテーブルをスキャンする

SQL を使用してテーブルをスキャンする

SQL を使用すると、SELECT 句を指定せずに、WHERE ステートメントを使用してテーブルをスキャンし、データのすべてを取得することができます。結果の 1 つ以上の列をリクエストできます。または、ワイルドカード文字「*」を使用する場合は、すべての項目をリクエストできます。

以下は SELECT ステートメントの使用例です。

```
/* Return all of the data in the table */
```

```
SELECT * FROM Music;
```

```
/* Return all of the values for Artist and Title */
SELECT Artist, Title FROM Music;
```

DynamoDB でテーブルをスキャンする

Amazon DynamoDB では、DynamoDB API または PartiQL (SQL 互換のクエリ言語) を使用して、テーブルに対してスキャンを実行します。

DynamoDB API

DynamoDB API では、テーブルまたはセカンダリインデックスの各項目にアクセスして、1 つまたは複数の項目または項目属性を返す Scan オペレーションを使用します。

```
// Return all of the data in the table
{
    TableName:  "Music"
}
```

```
// Return all of the values for Artist and Title
{
    TableName:  "Music",
    ProjectionExpression: "Artist, Title"
}
```

Scan オペレーションでは、FilterExpression パラメータも指定できます。これを使用して、結果に表示しない項目を破棄することができます。FilterExpression は、スキャンの実行後、結果が返されるまでの間に適用されます。(これは、大きなテーブルではお勧めしません。返される一致項目がごく少数でも、Scan 全体に対して料金が請求されます。)

PartiQL for DynamoDB

PartiQL では、Select ステートメントを使用するテーブルに対してすべての内容を返す ExecuteStatement オペレーションを使用してスキャンを実行します。

```
SELECT AlbumTitle, Year, Price
FROM Music
```

このステートメントは、Music テーブルのすべての項目を返すことに注意してください。

`Select` と `ExecuteStatement` を使用したコード例については、DynamoDB 用の PartiQL select ステートメント を参照してください。

インデックスを管理するときのリレーショナル (SQL) データベースと DynamoDB の相違点

インデックスは代替のクエリパターンへのアクセスを付与し、クエリを高速化できます。このセクションでは、SQL と Amazon DynamoDB におけるインデックスの作成と使用を比較します。

リレーショナルデータベースまたは DynamoDB のいずれを使用するにしても、インデックスの作成は慎重に判断する必要があります。テーブルに書き込みが発生するたびに、テーブルのインデックスはすべて、更新する必要があります。大きなテーブルでの書き込み量が多い環境では、大量のシステムリソースを消費する可能性があります。読み取り専用または読み取りがほとんどの環境では、これは大きな問題にはなりません。ただし、インデックスがアプリケーションによって実際に使用されており、ただ容量を取っていることがないよう確認する必要があります。

トピック

- インデックスを作成するときのリレーショナル (SQL) データベースと DynamoDB の相違点
- インデックスのクエリとスキャンをするときのリレーショナル (SQL) データベースと DynamoDB の相違点

インデックスを作成するときのリレーショナル (SQL) データベースと DynamoDB の相違点

SQL の `CREATE INDEX` ステートメントを Amazon DynamoDB の `UpdateTable` オペレーションと比較します。

トピック

- SQL を使ってインデックスを作成する
- DynamoDB でインデックスを作成する

SQL を使ってインデックスを作成する

リレーショナルデータベースのインデックスは、テーブルの異なる列に対して高速にクエリを実行できるデータ構造です。`CREATE INDEX` SQL ステートメントを使用して、既存のテーブルにインデックスが追加し、インデックスを作成する列を指定します。インデックスの作成後は、通常どおりテー

ブルのデータにクエリを実行できますが、テーブル全体をスキャンする代わりに、テーブルの指定された行を迅速に検索するため、データベースがインデックスを使用することができます。

インデックスの作成後は、データベースが維持します。テーブルのデータを変更した場合は、インデックスが自動的に変更され、テーブル内の変更を反映します。

MySQL では、次のようなインデックスを作成します。

```
CREATE INDEX GenreAndPriceIndex
ON Music (genre, price);
```

DynamoDB でインデックスを作成する

DynamoDB では、同様の目的のために、セカンダリインデックスを作成して使用できます。

DynamoDB のインデックスはリレーショナルな対応物とは異なります。セカンダリインデックスを作成する際、そのキー属性 (パーティションキーとソートキー) を指定する必要があります。セカンダリインデックスを作成したら、テーブルと同じように Query または Scan を実行できます。DynamoDB にはクエリオプティマイザーがないため、セカンダリインデックスは Query または Scan の場合にのみ使用されます。

DynamoDB では、次の 2 種類のインデックスをサポートしています。

- グローバルセカンダリインデックス – インデックスのプライマリキーは、テーブルからの任意の 2 つの属性になります。
- ローカルセカンダリインデックス – インデックスのパーティションキーは、テーブルのパーティションキーと同じである必要があります。ただし、ソートキーは、他の任意の属性にすることができます。

DynamoDB を使えば、セカンダリインデックスのデータは結果的にテーブルと整合性が取れます。テーブルまたはローカルセカンダリインデックスでの強い整合性を持つ Query または Scan オペレーションをリクエストできます。ただし、グローバルセカンダリインデックスは結果整合性のみをサポートします。

UpdateTable オペレーションを使用し、GlobalSecondaryIndexUpdates を指定して、既存のテーブルにグローバルセカンダリインデックスを追加できます。

```
{
    TableName: "Music",
```

```
    AttributeDefinitions:[
        {AttributeName: "Genre", AttributeType: "S"},
        {AttributeName: "Price", AttributeType: "N"}
    ],
    GlobalSecondaryIndexUpdates: [
        {
            Create: {
                IndexName: "GenreAndPriceIndex",
                KeySchema: [
                    {AttributeName: "Genre", KeyType: "HASH"}, //Partition key
                    {AttributeName: "Price", KeyType: "RANGE"}, //Sort key
                ],
                Projection: {
                    "ProjectionType": "ALL"
                },
                ProvisionedThroughput: {                                  // Only
 specified if using provisioned mode
                    "ReadCapacityUnits": 1,"WriteCapacityUnits": 1
                }
            }
        }
    ]
}
```

UpdateTable に以下のパラメータを提供する必要があります。

- TableName – インデックスが関連付けられるテーブル。
- AttributeDefinitions – インデックスのキースキーマ属性用のデータ型。
- GlobalSecondaryIndexUpdates – 作成するインデックスに関する詳細。
  - IndexName – インデックスの名前。
  - KeySchema – インデックスのプライマリキーに使用する属性。
  - Projection – テーブルからインデックスにコピーされる属性。この場合、ALL は、すべての属性がコピーされることを意味します。
  - ProvisionedThroughput (for provisioned tables) – このインデックスに必要な 1 秒あたりの読み取り/書き込み数。(これは、テーブルのプロビジョニングされたスループット設定とは異なります。)

このオペレーションの一部は、テーブルから新しいインデックスにデータをバックフィリングすることを含みます。バックフィリング中、テーブルは使用可能なままになります。ただし、インデック

スは Backfilling 属性が true から false に変わるまで、準備ができていません。DescribeTable オペレーションを使用して、この属性を表示できます。

インデックスのクエリとスキャンをするときのリレーショナル (SQL) データベースと DynamoDB の相違点

SQL の SELECT ステートメントを使用したインデックスのクエリとスキャンを Amazon DynamoDB の Query および Scan オペレーションと比較します。

トピック

- SQL を使ってインデックスのクエリを実行してスキャンする
- DynamoDB でインデックスのクエリを実行してスキャンする

SQL を使ってインデックスのクエリを実行してスキャンする

リレーショナルデータベースでは、インデックスを直接使用しません。代わりに、SELECT ステートメントの発行により、テーブルでクエリを実行し、クエリオプティマイザは任意のインデックスを利用できます。

クエリオプティマイザは、使用できるインデックスを評価し、クエリを高速化するためにそれらを使用できるかを決定するリレーショナルデータベース管理システム (RDBMS) のコンポーネントです。クエリを高速化するためにインデックスが使用できる場合、RDBMS は最初にインデックスにアクセスし、それを使用してテーブルのデータを特定します。

GenreAndPriceIndex を使用してパフォーマンスを向上できる SQL ステートメントを次に示します。ここでは、Music テーブルに十分なデータがあり、クエリオプティマイザが、テーブル全体にスキャンするのではなく、このインデックスを使用することを前提にします。

```
/* All of the rock songs */

SELECT * FROM Music
WHERE Genre = 'Rock';
```

```
/* All of the cheap country songs */

SELECT Artist, SongTitle, Price FROM Music
WHERE Genre = 'Country' AND Price < 0.50;
```

DynamoDB でインデックスのクエリを実行してスキャンする

DynamoDB では、テーブルに対して実行するのと同じように、インデックスに対して直接 `Query` および `Scan` オペレーションを実行します。DynamoDB API または PartiQL (SQL 互換のクエリ言語) を使用して、インデックスにクエリやスキャンを実行できます。`TableName` と `IndexName` の両方を指定する必要があります。

以下は、DynamoDB の GenreAndPriceIndex に対するいくつかのクエリです。(このインデックスのキースキーマは、ジャンルと価格で構成されています。)

DynamoDB API

```
// All of the rock songs

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre",
    ExpressionAttributeValues: {
        ":genre": "Rock"
    },
};
```

この例では `ProjectionExpression` を使用して、属性すべてではなく、一部のみを結果に表示することを示します。

```
// All of the cheap country songs

{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre and Price < :price",
    ExpressionAttributeValues: {
        ":genre": "Country",
        ":price": 0.50
    },
    ProjectionExpression: "Artist, SongTitle, Price"
};
```

次に示すのは、GenreAndPriceIndex に対するスキャンです。

```
// Return all of the data in the index
```

```
{
    TableName:  "Music",
    IndexName: "GenreAndPriceIndex"
}
```

PartiQL for DynamoDB

PartiQL では、Select ステートメントを使用して、インデックスに対してクエリとスキャンを実行します。

```
// All of the rock songs

SELECT *
FROM Music.GenreAndPriceIndex
WHERE Genre = 'Rock'
```

```
// All of the cheap country songs

SELECT *
FROM Music.GenreAndPriceIndex
WHERE Genre = 'Rock' AND Price < 0.50
```

次に示すのは、GenreAndPriceIndex に対するスキャンです。

```
// Return all of the data in the index

SELECT *
FROM Music.GenreAndPriceIndex
```

> **Note**
>
> Select を使用したコード例については、「DynamoDB 用の PartiQL select ステートメント」を参照してください。

テーブル内のデータを変更するときのリレーショナル (SQL) データベースと DynamoDB の相違点

SQL 言語は、データを変更するための UPDATE ステートメントを提供します。Amazon DynamoDB は、この UpdateItem オペレーションを使用して同様のタスクを実行します。

トピック

- SQL を使用してテーブルのデータを変更する
- DynamoDB でテーブルのデータを変更する

SQL を使用してテーブルのデータを変更する

SQL では、UPDATE ステートメントを使用して、1 つ以上の行を変更します。SET 句は、1 つ以上の行に新しい値を指定し、WHERE 句は変更する行を決定します。以下はその例です。

```
UPDATE Music
SET RecordLabel = 'Global Records'
WHERE Artist = 'No One You Know' AND SongTitle = 'Call Me Today';
```

WHERE 句に行が一致しない場合、UPDATE ステートメントは何も実行しません。

DynamoDB でテーブルのデータを変更する

DynamoDB では、クラシック API または PartiQL (SQL 互換のクエリ言語) を使用して、単一の項目を変更できます。複数の項目を変更する場合は、複数のオペレーションを使用する必要があります。

DynamoDB API

DynamoDB API では、UpdateItem オペレーションを使用して、単一の項目を変更します。

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ExpressionAttributeValues: {
        ":label": "Global Records"
```

```
    }
}
```

属性値を指定するには、変更する項目の Key 属性および UpdateExpression を指定する必要があります。UpdateItem は、「アップサート」オペレーションのように動作します。テーブルに項目が存在する場合は更新され、存在しない場合は新しい項目が追加 (挿入) されます。

UpdateItem は、条件付き書き込みをサポートしており、特定の ConditionExpression が true と評価された場合のみ、オペレーションが成功します。たとえば、次の UpdateItem オペレーションは、曲の価格が 2.00 以上でない限り、更新を実行しません。

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET RecordLabel = :label",
    ConditionExpression: "Price >= :p",
    ExpressionAttributeValues: {
        ":label": "Global Records",
        ":p": 2.00
    }
}
```

また、UpdateItem はアトミックカウンター、または増加減少する Number 型の属性をサポートしています。アトミックカウンターは多くの点で SQL データベースのシーケンスジェネレーター、IDENTITY 列、または自動インクリメントフィールドと似ています。

UpdateItem オペレーションを使用して、新しい属性 (Plays) を初期化し、曲の再生回数を追跡する例を次に示します。

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = :val",
    ExpressionAttributeValues: {
        ":val": 0
```

```
    },
    ReturnValues: "UPDATED_NEW"
}
```

ReturnValues パラメータは、更新された任意の属性の新しい値を返す、UPDATED_NEW に設定されています。この場合は、0 (ゼロ) を返します。

ユーザーがこの曲を再生するたびに、次の UpdateItem オペレーションを使用して Plays を 1 ずつ増分できます。

```
{
    TableName: "Music",
    Key: {
        "Artist":"No One You Know",
        "SongTitle":"Call Me Today"
    },
    UpdateExpression: "SET Plays = Plays + :incr",
    ExpressionAttributeValues: {
        ":incr": 1
    },
    ReturnValues: "UPDATED_NEW"
}
```

PartiQL for DynamoDB

PartiQL では、Update ステートメントを利用する ExecuteStatement オペレーションを使用して、テーブル内の項目を変更します。

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。これらの属性の値を指定する必要があります。

```
UPDATE Music
SET RecordLabel ='Global Records'
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

また、次の例のように、複数のフィールドを一度に変更することもできます。

```
UPDATE Music
SET RecordLabel = 'Global Records'
SET AwardsWon = 10
WHERE Artist ='No One You Know' AND SongTitle='Call Me Today'
```

また、Update はアトミックカウンター、または増加減少する Number 型の属性をサポートしています。アトミックカウンターは多くの点で SQL データベースのシーケンスジェネレーター、IDENTITY 列、または自動インクリメントフィールドと似ています。

次は、曲が再生された回数を追跡するための新しい属性 (Plays) を初期化する Update ステートメントの例です。

```
UPDATE Music
SET Plays = 0
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

ユーザーがこの曲を再生するたびに、次の Update ステートメントを使用して Plays を 1 ずつ増分できます。

```
UPDATE Music
SET Plays = Plays + 1
WHERE Artist='No One You Know' AND SongTitle='Call Me Today'
```

> **Note**
>
> Update と ExecuteStatement を使用したコード例については、DynamoDB 用の PartiQL 更新ステートメント を参照してください。

テーブルからデータを削除するときのリレーショナル (SQL) データベースと DynamoDB の相違点

SQL では、DELETE ステートメントを使用してテーブルから 1 つ以上の行を削除します。Amazon DynamoDB では、この DeleteItem オペレーションを使用して一度に 1 つの項目を削除します。

トピック

- SQL を使ってテーブルからデータを削除する
- DynamoDB でテーブルからデータを削除する

SQL を使ってテーブルからデータを削除する

SQL では、DELETE ステートメントを使用して、1 つ以上の行を削除します。WHERE 句は、変更する行を決定します。以下はその例です。

```
DELETE FROM Music
WHERE Artist = 'The Acme Band' AND SongTitle = 'Look Out, World';
```

WHERE 句を変更して、複数行を削除できます。たとえば、次の例に示すように、特定のアーティストのすべての曲を削除できます。

```
DELETE FROM Music WHERE Artist = 'The Acme Band'
```

DynamoDB でテーブルからデータを削除する

DynamoDB では、クラシック API または PartiQL (SQL 互換のクエリ言語) を使用して、単一の項目を削除できます。複数の項目を変更する場合は、複数のオペレーションを使用する必要があります。

DynamoDB API

DynamoDB API では、DeleteItem オペレーションを使用して、テーブルからデータを一度に 1 項目ずつ削除します。項目のプライマリキー値を指定する必要があります。

```
{
    TableName: "Music",
    Key: {
        Artist: "The Acme Band",
        SongTitle: "Look Out, World"
    }
}
```

> **Note**
>
> DeleteItem に加えて、Amazon DynamoDB は、同時に複数の項目を削除するための BatchWriteItem オペレーションをサポートしています。

DeleteItem は、条件付き書き込みをサポートしており、特定の ConditionExpression が true と評価された場合のみ、オペレーションが成功します。たとえば、次の DeleteItem オペレーションは RecordLabel 属性がある場合のみ項目を削除します。

```
{
    TableName: "Music",
    Key: {
```

```
        Artist: "The Acme Band",
        SongTitle: "Look Out, World"
    },
   ConditionExpression: "attribute_exists(RecordLabel)"
}
```

PartiQL for DynamoDB

PartiQL では、ExecuteStatement オペレーションでの Deleteステートメントを使用して、テーブルからデータを一度に 1 項目ずつ削除します。項目のプライマリキー値を指定する必要があります。

このテーブルのプライマリキーは、Artist および SongTitle で構成されます。これらの属性の値を指定する必要があります。

```
DELETE FROM Music
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks'
```

オペレーションには、追加の条件を指定することもできます。次の DELETE オペレーションは、11 を超える Awards が項目に含まれる場合にのみ、その項目を削除します。

```
DELETE FROM Music
WHERE Artist = 'Acme Band' AND SongTitle = 'PartiQL Rocks' AND Awards > 11
```

> **Note**
>
> DELETE と ExecuteStatement を使用したコード例については、DynamoDB 用の PartiQL 削除ステートメント を参照してください。

テーブルを削除するときのリレーショナル (SQL) データベースと DynamoDB の相違点

SQL では、DROP TABLE ステートメントを使用して、テーブルを削除します。Amazon DynamoDB では、DeleteTable オペレーションを使用します。

トピック

- SQL を使用してテーブルを削除する
- DynamoDB でテーブルを削除する

SQL を使用してテーブルを削除する

不要になったテーブルを完全に廃棄する場合、SQL の DROP TABLE ステートメントを使用します。

```
DROP TABLE Music;
```

テーブルは削除された後、復元できません。(一部のリレーショナルデータベースは DROP TABLE オペレーションを取消すことができますが、これはベンダー固有の機能であり、一般的には実装されていません。)

DynamoDB でテーブルを削除する

DynamoDB には、DeleteTable が同様のオペレーションです。次の例では、テーブルは完全に削除されます。

```
{
    TableName: "Music"
}
```

Amazon DynamoDB の学習リソースとツール

以下に示すその他のリソースを使用して、DynamoDB について理解し、作業を行うことができます。

トピック

- コーディングと可視化のツール
- 「規範的ガイダンス」の記事
- ナレッジセンターの記事
- ブログ投稿、リポジトリ、ガイド
- データモデリングと設計パターンのプレゼンテーション
- トレーニングコース

コーディングと可視化のツール

次のコーディングおよび視覚化ツールを使用して DynamoDB を操作できます。

- Amazon DynamoDB 用の NoSQL Workbench – DynamoDB テーブルの設計、作成、クエリ、管理に役立つ、統合されたビジュアルツールです。このツールは、データモデリング、データの視覚化、クエリ開発機能を備えています。
- Dynobase – DynamoDB テーブルの表示と操作、アプリケーションコードの作成、リアルタイム検証を伴うレコードの編集を簡単に行うデスクトップツール。
- DynamoDB Toolbox – データモデリング、JavaScript および Node.js での作業に役立つユーティリティを提供する Jeremy Daly 氏のプロジェクト。
- DynamoDB Streams Processor – DynamoDB Streams の操作を簡単にするために使用できるシンプルなツール。

「規範的ガイダンス」の記事

AWS「 の規範的ガイダンス」には、プロジェクトを加速させるのに役立つ、実績のある戦略、ガイド、パターンが記載されています。これらのリソースは、お客様のビジネス目標の達成を支援してきた長年の経験に基づいて、AWS テクノロジーのエキスパートと AWS パートナーのグローバルコミュニティによって開発されました。

データモデリングと移行

- DynamoDB の階層型データモデル
- DynamoDB によるデータのモデリング
- を使用して Oracle データベースを DynamoDB に移行するAWS DMS

グローバルテーブル

- Amazon DynamoDB グローバルテーブルを使用する

Serverless

- を使用してサーバーレス Saga パターンを実装するAWS Step Functions

SaaS アーキテクチャ

- 複数の SaaS 製品間のテナントを単一のコントロールプレーンで管理する
- C# と AWS CDK を使用するサイロモデル用の SaaS アーキテクチャでのテナントオンボーディング

データ保護とデータ移動

- Amazon DynamoDB へのクロスアカウントアクセスを設定する
- DynamoDB のフルテーブルコピーオプション
- でのデータベースのディザスタリカバリ戦略AWS

雑則

- DynamoDB でタグ付けを適用するのに役立ちます

規範ガイダンスの動画チュートリアル

- サーバーレスアーキテクチャを使用したデータパイプラインの作成
- Novartis - 購買エンジン: AI を活用した調達ポータル
- Veritiv: インサイトを活用して AWS データレイクで売上需要を予測
- mimik: AWS を活用してエッジマイクロサービスメッシュをサポートするハイブリッドエッジクラウド
- Amazon DynamoDB を使用した変更データキャプチャ

DynamoDB に関するその他の規範的ガイダンスの記事と動画については、「規範的ガイダンス」を参照してください。

ナレッジセンターの記事

AWS ナレッジセンターの記事と動画には、AWS のお客様から寄せられるよくある質問やリクエストが網羅されています。DynamoDB に関連した特定のタスクに関する最新のナレッジセンターの記事を、いくつか以下に示します。

コスト最適化

- Amazon DynamoDB を使用してコストを最適化する方法を教えてください。

スロットリングとレイテンシー

- Amazon DynamoDB テーブルでレイテンシーが高い場合のトラブルシューティング方法を教えてください。

- DynamoDB テーブルがスロットリングされているのはなぜですか?
- オンデマンド DynamoDB テーブルがスロットリングされているのはなぜですか?

Pagination (ページ分割

- DynamoDB にページネーションを実装するにはどうすればよいですか

トランザクション

- DynamoDB で `TransactWriteItems` API コールが失敗するのはなぜですか?

トラブルシューティング

- DynamoDB 自動スケーリングの問題を解決する方法を教えてください。
- DynamoDB の HTTP 4XX エラーをトラブルシューティングする方法を教えてください

DynamoDB に関するその他の記事や動画については、ナレッジセンターの記事を参照してください。

ブログ投稿、リポジトリ、ガイド

「DynamoDB デベロッパーガイド」の他に、DynamoDB を使用するうえで役立つリソースが多数あります。DynamoDB を使用するためのブログ投稿、リポジトリ、ガイドをいくつか紹介します。

- さまざまな AWS SDK 言語での DynamoDB コード例のAWSリポジトリ: Node.js、Java、Python、.Net、Go、Rust。
- DynamoDB ブック – Alex DeBrie 氏からの包括的なガイドで、DynamoDB を使用したデータモデリングに対する戦略主導型のアプローチについて説明します。
- DynamoDB ガイド – Alex DeBrie 氏からのオープンガイドで、DynamoDB NoSQL データベースの基本的な概念と高度な機能について説明します。
- How to switch from RDBMS to DynamoDB in 20 easy steps – Jeremy Daly 氏からデータモデリングを学ぶために役立つステップのリスト。
- DynamoDB JavaScript DocumentClient cheat sheet – Node.js または JavaScript 環境で DynamoDB を使用してアプリケーションの構築を開始するのに役立つデータシート。

- DynamoDB Core Concept Videos – このプレイリストでは、DynamoDB の多くの重要な概念が説明されています。

データモデリングと設計パターンのプレゼンテーション

DynamoDB を最大限に活用するには、データモデリングと設計パターンに関する次のリソースを使用できます。

- AWSre:Invent 2019: DynamoDB を使用したデータモデリング
  - DynamoDB データモデリングの原則の使用開始に役立つ、Alex DeBrie 氏の講演
- AWSre:Invent 2020: DynamoDB を使用したデータモデリング – パート 1
- AWSre:Invent 2020: DynamoDB を使用したデータモデリング – パート 2
- AWS re:Invent 2017: Advanced design patterns
- AWS re:Invent 2018: Advanced design patterns
- AWS re:Invent 2019: Advanced design patterns
  - このセッションでは、Jeremy Daly が 12 の重要ポイントを説明しています。
- AWS re:Invent 2020: DynamoDB advanced design patterns – Part 1
- AWS re:Invent 2020: DynamoDB advanced design patterns – Part 2
- Twitch での DynamoDB の営業時間

> **Note**
>
> 各セッションでは、さまざまなユースケースと例について説明しています。

トレーニングコース

DynamoDB についてさらに学ぶためのさまざまなトレーニングコースや教育オプションがあります。現在の例をいくつかご紹介します。

- Developing with Amazon DynamoDB – AWS による設計で、Amazon DynamoDB のデータモデリングを使用して実際のアプリケーションを開発する初心者をエキスパートに育成します。
- DynamoDB deep dive course – Pluralsight のコース。

- Amazon DynamoDB: Building NoSQL database-driven applications – edX でホストされる AWS トレーニングと認定チームのコース。

Amazon DynamoDB の読み込みと書き込み

DynamoDB の読み込みと書き込みは、テーブルからのデータの取得 (読み込み) と、テーブル内のデータの挿入、更新、削除 (書き込み) に関するオペレーションを指します。DynamoDB を使用する場合は、読み込みと書き込みの概念を理解することが不可欠です。読み込みと書き込みは、アプリケーションのパフォーマンスとコストに直接影響します。

このトピックでは、DynamoDB に適用されるさまざまなタイプの読み込み整合性について詳しく説明します。また、実行するさまざまな読み込みおよび書き込みオペレーションのユニット消費量についても説明します。

トピック

- DynamoDB の読み取り整合性
- DynamoDB の読み込みと書き込みのオペレーション

DynamoDB の読み取り整合性

Amazon DynamoDB は、テーブル、ローカルセカンダリインデックス (LSI)、グローバルセカンダリインデックス (GSI)、およびストリームからデータを読み取ります。詳細については、「Amazon DynamoDB のコアコンポーネント」を参照してください。テーブルと LSI のどちらにも、結果整合性のある読み込み (デフォルト) と強力な整合性のある読み込みという 2 つの読み込み整合性オプションが用意されています。GSI とストリームからのすべての読み取りは、結果整合性のある読み込みです。

アプリケーションが DynamoDB テーブルにデータを書き込み、HTTP 200 応答 (OK) を受信した場合、これは書き込みが正常に完了し、永続的に保持されたことを意味します。DynamoDB は、コミット済み読み取り分離を提供し、読み取りオペレーションが常に項目のコミット済みの値を返すようにします。読み取りでは、最終的に成功しなかった書き込みの項目に対してビューが表示されることはありません。コミット済み読み取り分離では、読み取りオペレーションの直後に項目の変更が防止されません。

結果整合性のある読み込み

結果整合性のある読み込みは、すべての読み取りオペレーションのデフォルトの読み込み整合性モデルです。DynamoDB テーブルまたはインデックスに対して結果整合性のある読み込みを発行すると、最近完了した書き込みオペレーションの結果が応答に反映されない場合があります。少し時間

が経ってから読み取りリクエストを繰り返すと、最終的に、より最新の項目が応答で返されるはずです。結果整合性のある読み込みは、テーブル、ローカルセカンダリインデックス、およびグローバルセカンダリインデックスでサポートされています。また、DynamoDB ストリームからのすべての読み取りも、結果整合性のある読み込みであることに注意してください。

結果整合性のある読み込みは、強力な整合性のある読み込みの半分のコストで済みます。詳細については、「Amazon DynamoDB 料金」を参照してください。

強力な整合性のある読み込み

読み取りオペレーション (GetItem、Query、Scan など) には、オプションの ConsistentRead パラメータがあります。ConsistentRead を true に設定すると、DynamoDB は、成功した以前のすべての書き込みオペレーションからの更新を反映した、最新データを含む応答を返します。強力な整合性のある読み込みは、テーブルとセカンダリインデックスでのみサポートされています。グローバルセカンダリインデックスまたは DynamoDB ストリームからの強力な整合性のある読み込みはサポートされていません。

グローバルテーブルの読み込み整合性

DynamoDB は、マルチアクティブおよびマルチリージョンのレプリケーション用のグローバルテーブルもサポートしています。グローバルテーブルは、異なる AWS リージョン間の複数のレプリカテーブルで構成されています。レプリカテーブル内の項目に加えられた変更は、同じグローバルテーブル内の他のすべてのレプリカにレプリケートされます。

グローバルテーブルは、2 つの整合性モードをサポートしています。マルチリージョンの結果整合性 (MREC) はデフォルトモードであり、項目の変更は通常 1 秒以内に他のレプリカにレプリケートされ、リージョン間で結果整合性が保たれます。マルチリージョンの強力な整合性 (MRSC) では、書き込みが返される前に項目の変更が別のリージョンに同期的にレプリケートされ、すべてのレプリカでの強力な整合性のある読み込みオペレーションは、常に項目の最新バージョンを返します。詳細については、「整合性モード」を参照してください。

DynamoDB の読み込みと書き込みのオペレーション

DynamoDB の読み込みオペレーションでは、パーティションキー値とオプションのソートキー値を指定して、テーブルから 1 つ以上の項目を取得できます。DynamoDB の書き込みオペレーションでは、テーブル内の項目を挿入、更新、または削除できます。このトピックでは、これら 2 つのオペレーションのキャパシティユニット消費量について説明します。

トピック

- 読み込みオペレーションのキャパシティユニット消費量
- 書き込みオペレーションのキャパシティユニット消費量

読み込みオペレーションのキャパシティユニット消費量

DynamoDB 読み込みリクエストは、強力な整合性、結果整合性、またはトランザクションのいずれかになります。

- 4 KB 以下の項目の強力な整合性のある読み込みリクエストには、1 つの読み込みユニットが必要です。
- 4 KB 以下の項目の結果整合性のある読み込みリクエストには、2 分の 1 の読み込みユニットが必要です。
- 4 KB 以下の項目のトランザクション読み込みリクエストには、2 つの読み込みユニットが必要です。

DynamoDB 読み込み整合性モデルの詳細については「DynamoDB の読み取り整合性」を参照してください。

読み込みの項目サイズは、次の 4 KB の倍数に切り上げられます。たとえば、3,500 バイトの項目の読み込みは、4 KB の項目の読み込みと同じスループットを消費します。

4 KB より大きい項目を読み込む必要がある場合、DynamoDB には追加の読み込みユニットが必要です。必要な読み込みユニットの合計数は、項目のサイズと、結果整合性のある読み込みまたは強力な整合性のある読み込みが必要かどうかによって異なります。例えば、項目のサイズが 8 KB の場合、1 つの強力な整合性のある読み込みを維持するには 2 つの読み込みユニットが必要です。結果整合性のある読み込みを選択した場合は 1 つの読み込みユニットが必要です。トランザクション読み込みリクエストを選択した場合は 4 つの読み込みユニットが必要です。

DynamoDB の読み込みオペレーションが読み込みユニットをどのように消費するかを以下に示します。

- GetItem: テーブルから単一の項目を読み込みます。`GetItem` が消費する読み込みユニットの数を決定するには、項目のサイズを次の 4 KB の倍数まで切り上げます。強力な整合性のある読み込みを指定した場合は、この数値が必要な読み込みユニットの数になります。結果整合性のある読み込み (デフォルト) の場合は、この数値を 2 で割ります。

  たとえば、3.5 KB の項目を読み取ると、DynamoDB は項目サイズを 4 KB まで切り上げます。10 KB の項目を読み取ると、DynamoDB は項目サイズを 12 KB まで切り上げます。

- BatchGetItem: 1 つ以上のテーブルから最大 100 個の項目を読み込みます。DynamoDB は、バッチ内の各項目を個別の GetItem リクエストとして処理します。DynamoDB は、まず各項目のサイズを次の 4 KB の倍数に切り上げてから合計サイズを計算します。この結果は、すべての項目の合計サイズと必ずしも一致しません。例えば、BatchGetItem がサイズ 1.5 KB と 6.5 KB の 2 つの項目を読み込むと、DynamoDB はサイズを 12 KB (4 KB + 8 KB) として計算します。DynamoDB はサイズを 8 KB (1.5 KB + 6.5 KB) として計算しません。
- Query: 同じパーティションキー値を持つ複数の項目を読み込みます。返されるすべての項目は単一の読み込みオペレーションとして扱われ、DynamoDB はすべての項目の合計サイズを計算します。次に、DynamoDB はサイズを次の 4 KB の倍数に切り上げます。たとえば、クエリの結果、合計サイズが 40.8 KB になる 10 項目が返されるとします。DynamoDB は、オペレーションの項目サイズを 44 KB に切り上げます。クエリの結果、64 バイトの項目が 1,500 項目返されると、累積サイズは 96 KB になります。
- Scan: テーブル内のすべての項目を読み込みます。DynamoDB は、スキャンにより返される項目のサイズではなく、評価される項目のサイズを考慮します。Scan オペレーションの詳細については、「DynamoDB でのテーブルのスキャン」を参照してください。

> **Important**
>
> 存在しない項目に対して読み込みオペレーションを実行した場合でも、DynamoDB は、上で説明したとおりに読み込みスループットを消費します。Query/Scan オペレーションでは、データが存在しない場合でも、読み込み整合性のタイプとリクエストを処理するために検索したパーティションの数に基づいて、追加の読み込みスループットの料金が発生します。

項目を返す任意のオペレーションで、属性のサブセットを取得するようリクエストできます。ただし、そうすることで項目サイズの計算に影響は生じません。また、Query と Scan は、属性値の代わりに項目数を返します。項目数の取得では、項目ごとに読み込みユニット数が同じで、項目サイズの計算方法も同じとします。これは、DynamoDB では、項目数を増加させるために項目単位で読み込む必要があるためです。

書き込みオペレーションのキャパシティユニット消費量

1 つの書き込みユニットは、最大サイズが 1 KB の項目について 1 回の書き込みを表します。1 KB より大きい項目を書き込む必要がある場合、DynamoDB は追加の書き込みユニットを消費する必要があります。トランザクション書き込みリクエストでは、最大 1 KB の項目を 1 回書き込むのに書き込みユニットが 2 個必要です。必要な書き込みリクエストユニットの合計数は、項目サイズに応じ

て異なります。例えば、項目のサイズが 2 KB の場合、1 回の書き込みリクエストを維持するには書き込みユニットが 2 個必要です。トランザクション書き込みリクエストには書き込みユニットが 4 個必要です。

書き込みの項目サイズは、次の 1 KB の倍数に切り上げられます。たとえば、500 バイトの項目の書き込みは、1 KB の項目の書き込みと同じスループットを消費します。

DynamoDB の書き込みオペレーションが書き込みユニットをどのように消費するかを以下に示します。

- PutItem: テーブルに 1 つの項目を書き込みます。同じプライマリキーを持つ項目がテーブル内に存在する場合、このオペレーションによって項目が置き換えられます。プロビジョニングされたスループットの消費量を算出する場合、対象となる項目サイズは 2 つのうち大きい方となります。
- UpdateItem: テーブル内の 1 つの項目を変更します。DynamoDB は更新の前後で実際の項目のサイズを考慮します。プロビジョニングされたスループットの消費は、これらの項目サイズの大きい方を反映しています。項目の属性の一部を更新する場合でも、`UpdateItem` は、プロビジョニングされたスループットの総量 (「前」の項目サイズと「後」の項目サイズで、より大きい方) を消費します。
- DeleteItem: テーブルから 1 つの項目を削除します。プロビジョニングされたスループットの消費量は、削除された項目のサイズに基づいています。
- BatchWriteItem: 1 つ以上のテーブルに最大 25 個の項目を書き込みます。DynamoDB は、バッチ内の各項目を個別の `PutItem` または `DeleteItem` リクエストとして処理します (更新はサポートされません)。DynamoDB は、まず各項目のサイズを次の 1 KB の倍数に切り上げてから合計サイズを計算します。この結果は、すべての項目の合計サイズと必ずしも一致しません。例えば、`BatchWriteItem` がサイズ 500 バイトと 3.5 KB の 2 つの項目を書き込むと、DynamoDB はサイズを 5 KB (1 KB + 4 KB) として計算します。DynamoDB はサイズを 4 KB (500 バイト + 3.5 KB) として計算しません。
- TransactWriteItems: 複数の書き込みアクション (`Put`、`Update`、`Delete`、`ConditionCheck`) を、すべてが成功するか、すべてが失敗する単一トランザクションにグループ化します。DynamoDB は、各項目に対して 2 つの基盤となる書き込み (トランザクションを準備する書き込みと、トランザクションをコミットする書き込み) を実行するため、1 KB までの項目のトランザクション書き込みは 2 つの書き込みユニットを消費します。この容量は、トランザクションがキャンセルされたときでも消費されます (条件チェックが失敗した場合など)。詳細については、「トランザクションの容量管理」を参照してください。

PutItem、UpdateItem、および DeleteItem の各オペレーションでは、DynamoDB は項目のサイズを次の 1 KB に切り上げます。たとえば、1.6 KB の項目を入力または削除すると、DynamoDB は項目サイズを 2 KB まで切り上げます。

PutItem、UpdateItem、DeleteItem の各オペレーションでは、条件付き書き込みが許可されます。この場合、オペレーションが成功するためには true と必ず評価される式を指定します。式が false と評価された場合でも、DynamoDB はテーブルの書き込みキャパシティユニットを消費します。消費される書き込みキャパシティユニットの数は、項目のサイズによって異なります この項目は、テーブル内の既存の項目になるか、作成または更新しようとしている新しい項目になる場合があります。例えば、既存の項目が 300 KB であるとします。作成または更新しようとしている新しい項目が 310 KB であるとします。この場合、消費される書き込みキャパシティユニットは、新しい項目の 310 KB に基づきます。

DynamoDB のスループットキャパシティ

このセクションでは、DynamoDB テーブルで使用できる 2 つのスループットモードの概要と、アプリケーションに適したキャパシティモードを選択する際の考慮事項について説明します。テーブルのスループットモードによって、テーブルのキャパシティの管理方法が決まります。また、スループットモードによって、テーブルの読み込みおよび書き込みオペレーションに対する課金方法も決まります。Amazon DynamoDB では、さまざまなワークロード要件に対応するために、テーブルのオンデマンドモードとプロビジョンドモードを使い分けることができます。

トピック

- オンデマンドモード
- プロビジョニングモード
- DynamoDB オンデマンドキャパシティモード
- DynamoDB プロビジョンドキャパシティモード
- DynamoDB ウォームスループットについて
- DynamoDB バーストキャパシティとアダプティブキャパシティ
- DynamoDB でキャパシティモードを切り替える際の考慮事項

オンデマンドモード

Amazon DynamoDB オンデマンドモードは、データベース管理を簡素化し、顧客の最も要求の厳しいアプリケーションをサポートするように自動的にスケールするサーバーレススループットオプションです。DynamoDB オンデマンドを使用すると、キャパシティプランニング、使用状況のモニタリング、スケーリングポリシーの設定を気にせずにテーブルを作成できます。DynamoDB オンデマンドは、読み込みおよび書き込みリクエストの料金が従量制であるため、使用した分だけを支払います。オンデマンドモードのテーブルでは、アプリケーションで実行することが予測される読み込みおよび書き込みスループットを指定する必要はありません。

オンデマンドモードは、ほとんどの DynamoDB ワークロードで推奨されるデフォルトのスループットオプションです。DynamoDB は、スループット管理とスケーリングのあらゆる側面を処理し、小規模から始めて 1 秒あたり数百万のリクエストにスケールできるワークロードをサポートします。テーブルのスループットキャパシティを管理することなく、必要に応じて DynamoDB テーブルの読み取りと書き込みを行うことができます。詳細については、「DynamoDB オンデマンドキャパシティモード」を参照してください。

プロビジョニングモード

プロビジョンドモードでは、アプリケーションに必要な 1 秒あたりの読み込みと書き込みの回数を指定する必要があります。プロビジョニングしたキャパシティの実際に消費した量ではなく、プロビジョニングした時間単位の読み込みおよび書き込みキャパシティに基づいて課金されます。これにより、コストの予測可能性を得るため、定義されたリクエストレート以下に維持されるように DynamoDB を制御することができます。

予測可能な増加を伴う安定したワークロードがあり、アプリケーションのキャパシティ要件を確実に予測できる場合は、プロビジョンドキャパシティの使用を選択できます。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。

DynamoDB オンデマンドキャパシティモード

Amazon DynamoDB オンデマンドは、キャパシティプランニングを行わずに、最も要求の厳しいワークロードに合わせて自動的にスケールする、真のサーバーレスデータベースエクスペリエンスを提供します。オンデマンドでは、セットアッププロセスを簡素化し、キャパシティの管理とモニタリングを排除し、高速の自動スケーリングを提供します。リクエストごとの支払い料金では、実際に使用したスループットに対してのみ料金を支払うため、アイドル状態のキャパシティについて心配する必要はありません。読み取りまたは書き込みリクエストごとに課金されるため、コストは実際の使用量を直接反映します。

オンデマンドモードを選択すると、DynamoDB は、前に到達したトラフィックレベルまで拡張または縮小して、ワークロードを即座に受け入れることができるようにします。ワークロードのトラフィックレベルが新しいピークに達すると、DynamoDB はスループット要件の増加に合わせて自動的にスケーリングします。オンデマンドモードは、小規模から始めて 1 秒あたり数百万のリクエストにスケールできる最新のサーバーレスアプリケーションの構築を簡素化するため、デフォルトかつ推奨されるスループットオプションです。オンデマンドテーブルをスケールアウトすると、将来的にはスロットリングなしで、同じスループットをすぐに再び達成できます。テーブルへのトラフィックをゼロにする場合、オンデマンドではスループットに対して課金されません。オンデマンドモードのスケーリングプロパティの詳細については、「初期スループットとスケーリングのプロパティ」を参照してください。

オンデマンドモードを使用するテーブルは、同じ 1 桁ミリ秒のレイテンシー、サービスレベルアグリーメント (SLA)、DynamoDB プロビジョンドモードが提供しているセキュリティを実現します。

> **Note**
>
> デフォルトでは、DynamoDB は意図しない過剰な使用から保護します。アカウント内のすべてのテーブルでテーブルレベルの読み取りおよび書き込みスループット制限の 40,000 を超えてスケールするには、このクォータの引き上げをリクエストできます。デフォルトのテーブルのスループットクォータを超えるスループットリクエストはスロットリングされます。詳細については、「スループットのデフォルトクォータ」を参照してください。

オプションで、個別のオンデマンドテーブルやグローバルセカンダリインデックスに対して 1 秒あたりの読み込み/書き込み (またはその両方) の最大スループットを設定することもできます。スループットを設定することで、テーブルレベルの使用量とコストを制限し、不注意によるリソース消費量の急増を防ぎ、過剰な使用をなくして予測可能なコスト管理を行うことができます。テーブルの最大スループットを超えるスループットリクエストはスロットリングされます。テーブル別の最大スループットは、アプリケーションの要件に基づいていつでも変更できます。詳細については、「オンデマンドテーブルの DynamoDB 最大スループット」を参照してください。

開始するには、オンデマンドモードを使用するテーブルを作成または更新します。詳細については、「DynamoDB テーブルの基本的なオペレーション」を参照してください。

24 時間のローリングウィンドウで、テーブルをプロビジョンドキャパシティモードからオンデマンドモードに最大 4 回切り替えることができます。テーブルは、オンデマンドモードからプロビジョンドキャパシティモードにいつでも切り替えることができます。

読み込みおよび書き込みキャパシティモードの詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。オンデマンドテーブルクォータについては、「読み取り/書き込みスループット」を参照してください。

トピック

- 読み取りリクエスト単位と書き込みリクエスト単位
- 初期スループットとスケーリングのプロパティ
- オンデマンドテーブルの DynamoDB 最大スループット

読み取りリクエスト単位と書き込みリクエスト単位

DynamoDB では、アプリケーションがテーブルに対して実行する読み込みと書き込みに対して、読み込みリクエストユニットと書き込みリクエストユニットに基づいて課金します。

1 つの読み込みリクエストユニットは、最大サイズが 4 KB の項目に対する 1 秒あたり 1 回の強力な整合性のある読み込み、または 1 秒あたり 2 回の結果整合性のある読み込みを表します。DynamoDB の読み込み整合性モデルの詳細については「DynamoDB の読み取り整合性」を参照してください。

1 つの書き込みリクエストユニットは、最大サイズが 1 KB の項目に対する 1 秒あたり 1 回の書き込みオペレーションを表します。

読み込みユニットと書き込みユニットの消費方法の詳細については、「DynamoDB の読み込みと書き込みのオペレーション」を参照してください。

初期スループットとスケーリングのプロパティ

オンデマンドキャパシティーモードを使用する DynamoDB テーブルは、アプリケーションのトラフィックボリュームに自動的に対応します。新しいオンデマンドテーブルでは、1 秒あたり最大 4,000 回の書き込みと 1 秒あたり最大 12,000 回の読み込みを維持できます。オンデマンドキャパシティーモードは、テーブルにおける前のピークトラフィックの最大 2 倍まで瞬時に対応します。例えば、アプリケーションのトラフィックパターンが 1 秒あたり 25,000 ~ 50,000 回の強力な整合性のある読み込みの範囲で変動し、1 秒あたり 50,000 回の読み込みが前回のトラフィックのピークであったとします。オンデマンドキャパシティモードは、1 秒あたり最大 100,000 回の読み込みの持続的なトラフィックに即座に対応します。アプリケーションが 1 秒あたり 100,000 回の読み込みのトラフィックを維持する場合、このピークが新たに前回のピークとなります。この前回のピークにより、後続のトラフィックは 1 秒あたり最大 200,000 回の読み込みに到達できます。

1 つのテーブルで前回のピークの 2 倍を超えるピークがワークロードで発生した場合、DynamoDB は、トラフィックボリュームの増加に合わせて、キャパシティの割り当てを自動的に増加します。このキャパシティの割り当てにより、ワークロードでスロットリングが発生しなくなります。ただし、30 分以内に前のピークの 2 倍を超えた場合、スロットリングが発生する可能性があります。例えば、アプリケーションのトラフィックパターンが 1 秒あたり 25,000 ~ 50,000 回の強力な整合性のある読み込みの範囲で変動し、1 秒あたり 50,000 回の読み込みが前回到達したトラフィックのピークであったとします。1 秒あたり 100,000 回を超える読み込みを駆動する前に、テーブルを事前ウォーミングするか、少なくとも 30 分間にわたってトラフィックの増加の間隔を空けることをお勧めします。事前ウォーミングの詳細については、「DynamoDB ウォームスループットについて」を参照してください。

ワークロードのピークトラフィックが前のピークの 2 倍以内である場合、DynamoDB は 30 分間のスロットリング制限を行いません。ピークトラフィックが前のピークの 2 倍を超える場合は、前のピークから 30 分後にこの増加が発生することを確認してください。

オンデマンドテーブルの DynamoDB 最大スループット

オンデマンドテーブルの場合、オプションで、個々のテーブルおよび関連するグローバルセカンダリインデックス (GSI) の 1 秒あたりの読み込みまたは書き込み (または両方) の最大スループットを指定できます。オンデマンドの最大スループットを指定すると、テーブルレベルの使用量とコストを制限できます。デフォルトでは、最大スループットの設定は適用されません。オンデマンドのスループットレートは、アカウント内のすべてのテーブルでテーブルレベルの読み取りおよび書き込みスループットに対する 40,000 の AWS サービスクォータによって制限されます。必要に応じて、サービスクォータの引き上げをリクエストできます。

オンデマンドテーブルの最大スループットを設定すると、指定した最大量を超えるスループットリクエストはスロットリングされます。テーブルレベルのスループット設定は、アプリケーションの要件に応じていつでも変更できます。

オンデマンドテーブルの最大スループットを使用することでメリットを得られる一般的なユースケースの例は以下のとおりです。

- スループットのコスト最適化 — オンデマンドテーブルの最大スループットを使用すると、コスト予測可能性と管理可能性を高めることができます。さらに、トラフィックパターンと予算が異なるワークロードをサポートするために、オンデマンドモードを使用する柔軟性も高まります。
- 過剰な使用からの保護 – 最大スループットを設定することで、オンデマンドテーブルに対する最適化されていないコードや不正なプロセスに伴う読み込みや書き込みの消費量の偶発的な急増を防止できます。このテーブルレベルの設定により、組織は特定の期間にわたる過剰なリソースの消費を防止できます。
- ダウンストリームサービスの保護 — 顧客アプリケーションには、サーバーレステクノロジーと非サーバーレステクノロジーを含めることができます。サーバーレスアーキテクチャは、需要に合わせて迅速にスケールできます。ただし、固定キャパシティを持つダウンストリームコンポーネントでは、対応できなくなる場合があります。オンデマンドテーブルに最大スループット設定を実装すると、複数のダウンストリームコンポーネントに対する大量のイベントの伝播に伴う予期しない副作用を防止できます。

オンデマンドモードの最大スループットは、新規および既存の単一リージョンのテーブル、グローバルテーブル、GSI に対して設定できます。最大スループットは、Amazon S3 ワークフローからのテーブルの復元時とデータのインポート時にも設定できます。

オンデマンドテーブルの最大スループット設定を指定するには、DynamoDB コンソール、AWS CLI、AWS CloudFormation、または DynamoDB API を使用できます。

> **Note**
>
> オンデマンドテーブルの最大スループットは、ベストエフォートベースで適用されるため、保証されたリクエストの上限としてではなく、目標として考える必要があります。バーストキャパシティにより、ワークロードが、指定した最大スループットを一時的に超える場合があります。場合によっては、DynamoDB がバーストキャパシティを使用して、テーブルの最大スループット設定を超える読み込みや書き込みに対応します。バーストキャパシティーにより、スロットリングされていた可能性のある読み込みまたは書き込みリクエストが成功します。

トピック

- オンデマンドモードの最大スループットを使用する際の考慮事項
- リクエストのスロットリングと CloudWatch メトリクス

オンデマンドモードの最大スループットを使用する際の考慮事項

オンデマンドモードでテーブルの最大スループットを使用する場合、次の考慮事項が適用されます。

- オンデマンドテーブルまたはテーブル内のグローバルセカンダリインデックスごとに読み込みと書き込みの最大スループットを個別に設定し、特定の要件に基づいてアプローチを微調整できます。
- Amazon CloudWatch を使用して、DynamoDB テーブルレベルの使用状況メトリクスをモニタリングおよび把握し、オンデマンドモードの適切な最大スループット設定を決定できます。詳細については、「DynamoDB のメトリクスとディメンション」を参照してください。
- 1 つのグローバルテーブルレプリカで読み込みや書き込み (または両方) の最大スループット設定を指定すると、同じ最大スループット設定がすべてのレプリカテーブルに自動的に適用されます。データが適切にレプリケートされるためには、グローバルテーブル内のレプリカテーブルとセカンダリインデックスの書き込みスループット設定が同じであることが重要です。詳細については、「グローバルテーブルのベストプラクティス」を参照してください。
- 読み込みまたは書き込みの最大スループットとして指定できる最小値は、1 秒あたり 1 つのリクエストユニットです。
- 指定する最大スループットは、オンデマンドテーブルまたはテーブル内の個別のグローバルセカンダリインデックスで使用できるデフォルトのスループットクォータより小さくする必要があります。

リクエストのスロットリングと CloudWatch メトリクス

アプリケーションがオンデマンドテーブルに設定した読み込みまたは書き込みの最大スループットを超えると、DynamoDB はリクエストのスロットリングを開始します。DynamoDB は、読み込みや書き込みをスロットリングすると、発信者に `ThrottlingException` を返します。その後、必要に応じて適切なアクションを実行できます。例えば、テーブルの最大スループット設定を増加または無効化したり、リクエストを再試行する前に短い間待ったりできます。

テーブルまたはグローバルセカンダリインデックスに設定した最大スループットのモニタリングを簡素化するために、CloudWatch はメトリクスとして OnDemandMaxReadRequestUnits と OnDemandMaxWriteRequestUnits を提供しています。

DynamoDB プロビジョンドキャパシティモード

Amazon DynamoDB で新しいプロビジョンドテーブルを作成する場合は、テーブルのプロビジョンドスループットキャパシティを指定する必要があります。これは、テーブルがサポートできる読み込みおよび書き込みスループットの量です。プロビジョニングしたキャパシティの実際に消費した量ではなく、プロビジョニングした時間単位の読み込みおよび書き込みキャパシティに基づいて課金されます。

アプリケーションのデータとアクセス要件が変わると、テーブルのスループット設定の調整が必要になる場合があります。Auto Scaling を使用すると、トラフィックの変更に応じて、テーブルのプロビジョンドキャパシティーを自動的に調整できます。DynamoDB 自動スケーリングは、Application Auto Scaling でスケーリングポリシーを使用します。DynamoDB で自動スケーリングを設定するには、ターゲット使用率に加えて、読み取りおよび書き込みキャパシティの最小レベルと最大レベルを設定します。Application Auto Scaling は、メトリクスとターゲットから逸脱したときにスケーリングイベントをトリガーする CloudWatch アラームを作成および管理します。Auto Scaling は、テーブルのアクティビティをモニタリングし、事前設定されたしきい値に基づいてキャパシティ設定を調整します。消費されたキャパシティが、設定したターゲット使用率を 2 分間連続して超過すると、自動スケーリングがトリガーされます。CloudWatch アラームは、自動スケーリングをトリガーする前に、最大数分の短い遅延を伴う場合があります。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

DynamoDB Auto Scaling を使用している場合、スループット設定は実際のワークロードに応じて自動的に調整されます。UpdateTable オペレーションを使用して、テーブルのスループットキャパシティを手動で調整することもできます。例えば、既存のデータストアから新しい DynamoDB テーブルにデータをバルクロードする必要がある場合などに便利です。大容量の書き込みスループットを設

定したテーブルを作成し、データのバルクロードが完了してからこの設定を削減ですることができます。

> **Note**
>
> デフォルトでは、DynamoDB は意図しない過剰な使用から保護します。アカウント内のすべてのテーブルでテーブルレベルの読み取りおよび書き込みスループット制限の 40,000 を超えてスケールするには、このクォータの引き上げをリクエストできます。デフォルトのテーブルのスループットクォータを超えるスループットリクエストはスロットリングされます。詳細については、「スループットのデフォルトクォータ」を参照してください。

24 時間のローリングウィンドウで、テーブルをプロビジョンドキャパシティモードからオンデマンドモードに最大 4 回切り替えることができます。テーブルは、オンデマンドモードからプロビジョンドキャパシティモードにいつでも切り替えることができます。

読み込みおよび書き込みキャパシティモードの詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。

トピック

- 読み取りキャパシティユニットと書き込みキャパシティユニット
- 初期スループット設定の選択
- DynamoDB 自動スケーリング
- DynamoDB Auto Scaling によるスループットキャパシティの自動管理
- DynamoDB リザーブドキャパシティ

読み取りキャパシティユニットと書き込みキャパシティユニット

プロビジョンドモードテーブルでは、スループット要件をキャパシティユニットの数で指定します。キャパシティユニットは、アプリケーションが 1 秒あたりに読み込みまたは書き込みを行う必要があるデータの量を表します。これらの設定は必要に応じて後から変更できます。また、DynamoDB Auto Scaling を有効化して自動的に変更することもできます。

最大サイズ 4 KB の項目の場合、1 つの読み込みキャパシティユニット (RCU) は、1 秒あたり 1 回の強力な整合性のある読み込み、または 1 秒あたり 2 回の結果整合性のある読み込みを表します。DynamoDB の読み込み整合性モデルの詳細については「DynamoDB の読み取り整合性」を参照してください。

1 つの書き込みキャパシティユニット (WCU) は、最大サイズが 1 KB の項目の場合、1 秒あたり 1 回の書き込みを表します。さまざまな読み込みおよび書き込みオペレーションの詳細については、「DynamoDB の読み込みと書き込みのオペレーション」を参照してください。

初期スループット設定の選択

アプリケーションごとにデータベースの読み込みおよび書き込みの要件は異なります。DynamoDB テーブルの初期スループット設定を決定する際は、以下の点を考慮してください。

- 予想される読み込みおよび書き込みリクエストレート — 1 秒あたりに実行する必要がある読み込みおよび書き込みの数を推定する必要があります。
- 項目のサイズ — 一部の項目は小さいため、読み込みや書き込みには 1 つのキャパシティユニットを使用できます。項目のサイズが大きくなると、複数のキャパシティーユニットが必要になります。テーブルに追加する項目の平均サイズを見積もることで、テーブルのプロビジョニングされたスループットに対する正確な設定を指定できます。
- 読み込みの整合性要件 — 読み込みキャパシティユニットは、強力な整合性のある読み込みオペレーションに基づいており、結果整合性のある読み込みの 2 倍のデータベースリソースを消費します。アプリケーションが強力な整合性のある読み込みを要求するか、またはこの要件を緩和して結果的に整合性のある読み込み行うかどうかを決定する必要があります DynamoDB での読み込みオペレーションは、デフォルトでは結果整合性のある読み込みです。必要に応じて、これらのオペレーションに強力な整合性のある読み込みをリクエストできます。

例えば、テーブルから 1 秒あたり 80 項目を読み込むとします。項目のサイズは 3 KB で、強力な整合性のある読み込みが必要です。このシナリオでは、読み込みごとに 1 つのプロビジョニングされた読み込みキャパシティユニットが必要です。この数を決定するには、オペレーションの項目サイズを 4 KB で割ります。次に、以下の例に示すように、最も近い整数に切り上げます。

- 3 KB / 4 KB = 0.75、または 1 つの読み込みキャパシティユニット

したがって、テーブルから 1 秒あたり 80 項目を読み込むには、次の例に示すように、テーブルのプロビジョニングされた読み込みスループットを 80 個の読み込みキャパシティユニットに設定します。

- 1 読み込み容量単位/項目 x 80 読み込み/秒 = 80 読み込み容量単位

次に、テーブルに 1 秒あたり 100 項目を書き込み、各項目のサイズが 512 バイトであるとします。この場合、書き込みごとに 1 つのプロビジョニングされた書き込みキャパシティユニットが必要で

す。この数を決定するには、オペレーションの項目サイズを 1 KB で割ります。次の例に示すように、結果を最も近い整数に切り上げます。

- 512 バイト/1 KB = 0.5 または 1 つの書き込みキャパシティユニット

テーブルに 1 秒あたり 100 項目を書き込むには、テーブルのプロビジョニングされた書き込みスループットを 100 個の書き込みキャパシティユニットに設定します。

- 1 書き込み容量単位/項目 x 100 書き込み/秒 = 100 書き込み容量単位

DynamoDB 自動スケーリング

DynamoDB 自動スケーリングでは、テーブルとグローバルセカンダリインデックスのプロビジョニングされたスループットキャパシティを自動的に管理します。読み込みおよび書き込みのキャパシティーユニットの範囲 (上限と下限) と、また、その範囲内で目標使用率を定義します。DynamoDB Auto Scaling では、アプリケーションのワークロードが増減しても、ターゲットの使用率が維持されます。

DynamoDB Auto Scaling では、急激なトラフィック増加をリクエストのスロットリングなしに処理するために、テーブルまたはグローバルセカンダリインデックスのプロビジョンされた読み込み容量と書き込み容量を増やすことができます。ワークロードが減ると、DynamoDB Auto Scaling はスループットを低下させ、未使用のプロビジョンされた容量に料金が発生しないようにします。

> **Note**
>
> AWS マネジメントコンソール を使用してテーブルまたはグローバルセカンダリインデックスを作成すると、デフォルトで DynamoDB オートスケーリングが有効になります。Auto Scaling の設定は、コンソール、AWS CLI、またはいずれかの AWS SDK を使用していつでも管理できます。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

使用率

使用率は、プロビジョニングキャパシティを超過しているかどうかを判断するのに役立ちます。超過している場合は、テーブルのキャパシティを減らしてコストを削減する必要があります。逆に、プロビジョニングキャパシティを下回っているかどうかを判断するのにも役立ちます。この場合、予期しないトラフィック増によりリクエストがスロットリングされる可能性を防ぐため、テーブルのキャパ

シティを増やす必要があります。詳細については、「Amazon DynamoDB auto scaling: Performance and cost optimization at any scale」を参照してください。

DynamoDB 自動スケーリングを使用している場合は、ターゲット使用率も設定する必要があります。自動スケーリングでは、この使用率をターゲットとして使用し、キャパシティを上下に調整します。ターゲット使用率は 70% に設定することをお勧めします。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

DynamoDB Auto Scaling によるスループットキャパシティの自動管理

多くのデータベースワークロードは本質的に循環的なものであり、前もって予測することが困難なものもあります。例えば、日中の時間帯に大部分のユーザーがアクティブなソーシャルネットワーキングアプリがあるとします。データベースは日中のアクティビティを処理できる必要がありますが、夜間のスループットに同じレベルは必要ありません。別の例としては、予想以上に急速に普及した新しいモバイルゲームアプリが挙げられます。ゲームがあまりに人気になると、利用可能なデータベースリソースを超過し、パフォーマンスが低下して顧客が不満を感じるようになります。この種のワークロードでは多くの場合、手動介入によってデータベースリソースを使用レベルに応じて上下させる必要があります。

Amazon DynamoDB Auto Scaling は AWS Application Auto Scaling サービスを使用し、実際のトラフィックパターンに応じてプロビジョンドスループット性能をユーザーに代わって動的に調節します。これにより、テーブルまたはグローバルセカンダリインデックス (GSI) では、読み取りと書き込みのプロビジョンドキャパシティを増やし、スロットリングなしでトラフィックの急増に対処できます。ワークロードが減ると、Application Auto Scaling はスループットを低下させ、未使用のプロビジョンされた容量に料金が発生しないようにします。

> **Note**
>
> AWS マネジメントコンソール を使用してテーブルまたはグローバルセカンダリインデックスを作成すると、デフォルトで DynamoDB Auto Scaling が有効になります。Auto Scaling の設定はいつでも変更できます。詳細については、「AWS マネジメントコンソール と DynamoDB Auto Scaling の使用」を参照してください。
>
> テーブルまたはグローバルテーブルレプリカを削除しても、関連するスケーラブルターゲット、スケーリングポリシー、または CloudWatch アラームが共に自動的に削除されることはありません。

Application Auto Scaling を使用して、テーブルまたはグローバルセカンダリインデックスのスケーリングポリシーを作成します。スケーリングポリシーは、テーブルまたはインデックスの読み込み

キャパシティまたは書き込みキャパシティ (またはその両方) およびプロビジョニングされたキャパシティユニット設定をスケーリングするかどうかを指定します。

スケーリングポリシーには、ターゲット使用率 (ある時点で消費したプロビジョン済みスループットの割合) も含まれます。Application Auto Scaling はのターゲット追跡アルゴリズムを使用して、実際のワークロードに応じてテーブル (インデックス) のプロビジョンスループットを上下に調整することで、実際の容量使用率がターゲット使用率に、またはその近くに留まるようにします。

DynamoDB は、消費されたプロビジョンドスループットを 1 分間出力します。消費されたキャパシティが、設定したターゲット使用率を 2 分間連続して超過すると、自動スケーリングがトリガーされます。CloudWatch アラームは、自動スケーリングをトリガーする前に、最大数分の短い遅延を伴う場合があります。この遅延により、CloudWatch メトリクスの正確な評価が保証されます。消費されたスループットのスパイク間隔が 1 分を超えると、自動スケーリングはトリガーされない場合があります。同様に、15 個の連続するデータポイントがターゲット使用率を下回ると、スケールダウンイベントが発生する場合があります。いずれの場合も、自動スケーリングのトリガー後に、UpdateTable API が呼び出されます。テーブルやインデックスのプロビジョンドキャパシティの更新には、数分かかる場合があります。この間に、テーブルの前のプロビジョンドキャパシティを超えるリクエストはスロットリングされます。

> **Important**
>
> 侵害するデータポイントの数を調整して、基礎となるアラームをトリガーすることはできません (ただし、現在の数は将来変更される可能性があります)。

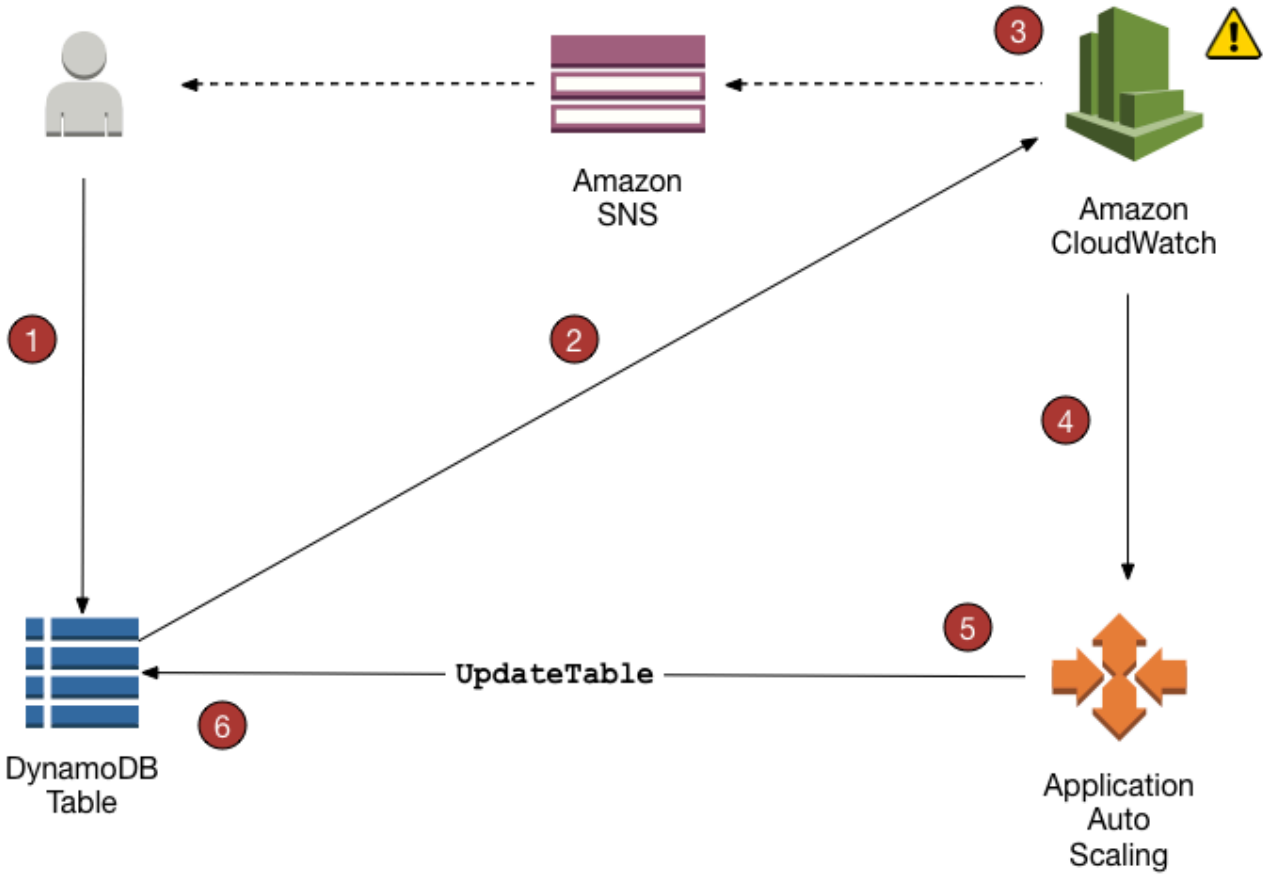
読み取りおよび書き込み容量に対して、Auto Scaling ターゲット使用率の値を 20% から 90% の間で設定できます。

> **Note**
>
> テーブルに加え、DynamoDB Auto Scaling はグローバルセカンダリインデックスもサポートします。すべてのグローバルセカンダリインデックスは、基本テーブルとは別に、固有のプロビジョンドスループット性能を持ちます。グローバルセカンダリインデックスのスケーリングポリシーを作成すると、Application Auto Scaling がインデックスのプロビジョンされたスループット設定を調整し、実際の使用率がターゲット使用率と同じか近い値で維持されるようになります。

DynamoDB Auto Scaling の仕組み

> **Note**
>
> DynamoDB Auto Scaling の簡単な使用方法については、AWS マネジメントコンソール と DynamoDB Auto Scaling の使用 を参照してください。

次の図表は、DynamoDB Auto Scaling によるテーブルのスループットキャパシティの管理方法について、高レベルの概要を示しています。



次のステップは、前の図に示された Auto Scaling のプロセスをまとめたものです。

1. DynamoDB テーブルの Application Auto Scaling ポリシーを作成します。
2. DynamoDB は、消費された容量メトリクスを Amazon CloudWatch に公開します。
3. テーブルの消費された容量が一定期間中にターゲット使用率を超える (または下回る) と、Amazon CloudWatch はアラームをトリガーします。コンソールでアラームを表示

し、Amazon Simple Notification Service (Amazon SNS) を使用して通知を受け取ることができます。

4. CloudWatch アラームは、Application Auto Scaling をコールしてスケーリングポリシーを評価します。
5. Application Auto Scaling は `UpdateTable` リクエストを発行し、テーブルのプロビジョンされたスループットを調整します。
6. DynamoDB は `UpdateTable` リクエストを処理してテーブルのプロビジョンドスループット性能を動的に増減し、ターゲット使用率に近づけます。

DynamoDB Auto Scaling の仕組みを理解するため、`ProductCatalog` という名前のテーブルがあると仮定します。テーブルにはまれにデータがバルクロードされます。そのため、書き込みアクティビティが頻繁に生じることはありません。ただし、時間で変化する高度な読み込みアクティビティが生じています。`ProductCatalog` の Amazon CloudWatch メトリクスをモニタリングすることにより、テーブルに 1,200 ユニットの読み込み容量が必要である (アクティビティのピーク時に DynamoDB が読み込みリクエストをスロットリングしないため) と判断します。さらに、読み込みトラフィックが最も低い時点で、`ProductCatalog` には最少で 150 ユニットの読み込みキャパシティが必要です。スロットリングの防止の詳細については、「Amazon DynamoDB でのスロットリングのトラブルシューティング」を参照してください。

読み込みキャパシティ 150 ~ 1,200 ユニットの範囲内で、`ProductCatalog` に適したターゲット使用率 70% を決定します。ターゲット使用率は、プロビジョンされた容量単位に対して消費された容量単位の割合がパーセンテージで示されます。Application Auto Scaling は、ターゲット追跡アルゴリズムを使用して、`ProductCatalog` のプロビジョンされた読み込み容量が必要に応じて調整され、使用率が 70% またはその近くに留まるようにします。

> **Note**
>
> DynamoDB Auto Scaling は、実際のワークロードの増減が数分間維持された場合にのみ、プロビジョンされたスループット設定を変更します。アプリケーションオートスケーリングターゲット追跡アルゴリズムはターゲット使用率を選択した値の付近に長期に渡って維持しようとします。
> アクティビティの急激かつ短時間の上昇は、テーブルに組み込まれたバーストキャパシティで対応されます。詳細については、「バーストキャパシティ」を参照してください。

`ProductCatalog` テーブルの DynamoDB Auto Scaling を有効にするには、スケーリングポリシーを作成します。このポリシーでは、以下を指定します。

- 管理するテーブルまたはグローバルセカンダリインデックス
- 管理するキャパシティタイプ (読み込みキャパシティまたは書き込みキャパシティ)
- プロビジョニングされたスループット設定の上下の境界
- ターゲット使用率

スケーリングポリシーを作成すると、Application Auto Scaling はユーザーに代わって Amazon CloudWatch アラームのペアを作成します。各ペアはプロビジョニングされたスループット設定の上下の境界を示します。CloudWatch アラームは、テーブルの実際の使用率が一定期間ターゲット使用率を逸脱したときにトリガーされます。

いずれかの CloudWatch アラームがトリガーされると、Amazon SNS は通知を送信します (有効にしている場合)。その後 CloudWatch アラームは Application Auto Scaling をコールし、`ProductCatalog` テーブルのプロビジョン容量を必要に応じて調整するように DynamoDB に通知します。

スケーリングイベント中、AWS Config は記録された設定項目ごとに課金されます。スケーリングイベントが発生すると、読み取り/書き込み自動スケーリングイベントごとに 4 つの CloudWatch アラームが作成されます。ProvisionedCapacity アラーム: ProvisionedCapacityLow、ProvisionedCapacityHigh、および ConsumedCapacity アラーム: AlarmHigh、AlarmLow です。その結果、合計 8 つのアラームが生成されます。そのため、AWS Config は、スケーリングイベントごとに 8 つの設定項目を記録します。

> **Note**
>
> また、DynamoDB のスケーリングを特定の時間に実行するようにスケジュールすることもできます。基本的な手順については、こちらを参照してください。

使用に関する注意事項

DynamoDB Auto Scaling の使用を開始する前に、以下を確認する必要があります。

- DynamoDB Auto Scaling は、自動スケーリングポリシーに従って、読み込み容量や書き込み容量を必要に応じて増加させます。Amazon DynamoDB のクォータ で説明されているように、すべての DynamoDB クォータは有効です。

- DynamoDB Auto Scaling により、プロビジョンされたスループット設定を手動で変更できなくなることはありません。これらの手動調整が、DynamoDB Auto Scaling に関連する既存の CloudWatch アラームに影響することはありません。
- 1 つ以上のグローバルセカンダリインデックスを持つテーブルの DynamoDB Auto Scaling を有効にする場合、これらのインデックスに Auto Scaling を均等に適用することをお勧めします。これにより、テーブルの書き込みと読み取りのパフォーマンスが向上し、スロットリングを回避できます。 auto scaling を有効にするには、AWS マネジメントコンソール で [Apply same settings to global secondary indexes] (グローバルセカンダリインデックスに同じ設定を適用する) を選択します。詳細については、「既存のテーブルでの DynamoDB Auto Scaling の有効化」を参照してください。
- テーブルまたはグローバルテーブルレプリカを削除しても、関連するスケーラブルターゲット、スケーリングポリシー、または CloudWatch アラームが共に自動的に削除されることはありません。
- 既存のテーブルの GSI を作成する場合、その GSI の Auto Scaling は有効になりません。GSI を構築している間は、容量を手動で管理する必要があります。GSI のバックフィルが完了し、アクティブステータスに達すると、Auto Scaling は通常どおり動作します。

AWS マネジメントコンソール と DynamoDB Auto Scaling の使用

AWS マネジメントコンソール を使用して新しいテーブルを作成すると、Amazon DynamoDB Auto Scaling はデフォルトでそのテーブルに対して有効になります。コンソールを使用して、既存テーブルの Auto Scaling の有効化、Auto Scaling 設定の変更、Auto Scaling の無効化を行うこともできます。

> **Note**
>
> スケールインおよびスケールアウトのクールダウン時間の設定など、より高度な特徴については、AWS Command Line Interface (AWS CLI) を使用して DynamoDB Auto Scaling を管理します。詳細については、「AWS CLI を使用した DynamoDB Auto Scaling の管理」を参照してください。

トピック

- 開始する前に: DynamoDB Auto Scaling のアクセス許可をユーザーに付与する
- Auto Scaling を有効にした新しいテーブルの作成
- 既存のテーブルでの DynamoDB Auto Scaling の有効化

- コンソールでの Auto Scaling アクティビティの表示
- DynamoDB Auto Scaling 設定の変更または無効化

開始する前に: DynamoDB Auto Scaling のアクセス許可をユーザーに付与する

AWS Identity and Access Management (IAM) の場合、AWS マネージドポリシー `DynamoDBFullAccess` は、DynamoDB コンソールを使用するために必要なアクセス許可を提供します。ただし、DynamoDB Auto Scaling の場合、ユーザーには追加アクセス許可が必要です。

> **Important**
>
> 自動スケーリング対応のテーブルを削除するには、`application-autoscaling:*` アクセス許可が必要です。AWS マネージドポリシー `DynamoDBFullAccess` には、必要なアクセス許可が含まれています。

DynamoDB コンソールアクセスと DynamoDB Auto Scaling 用にユーザーを設定するには、ロールを作成し、そのロールに AmazonDynamoDBFullAccess ポリシーを追加します。次に、ロールをユーザーに割り当てます。

Auto Scaling を有効にした新しいテーブルの作成

> **Note**
>
> DynamoDB Auto Scaling では、ユーザーに代わって Auto Scaling アクションを実行する、サービスリンクロール (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) の存在を必要とします。このロールは自動的に作成されます。詳細については、「Application Auto Scaling ユーザーガイド」の「Application Auto Scaling 用のサービスリンクロール」を参照してください。

Auto Scaling を有効にして新しいテーブルを作成するには

1. https://console.aws.amazon.com/dynamodb/ で DynamoDB コンソールを開きます。
2. [Create table] を選択します。
3. [テーブルを作成] ページで、テーブル名とプライマリキーの詳細を入力します。

4. [デフォルト設定] を選択すると、新しいテーブルで自動スケーリングが有効になります。

   それ以外の場合は、[設定をカスタマイズ] を選択し、以下の操作を実行してテーブルのカスタム設定を指定します。

   a. [テーブルクラス] で、デフォルト設定である [DynamoDB 標準] を選択したままにします。

   b. [読み取り/書き込みキャパシティーの設定] で、デフォルト設定の [プロビジョンド] を選択したままにして、以下の操作を行います。

      i. [読み取りキャパシティー] で、[自動スケーリング] が [オン] に設定されていることを確認します。

      ii. [書き込みキャパシティー] で、[自動スケーリング] が [オン] に設定されていることを確認します。

      iii. [読み取りキャパシティー] と [書き込みキャパシティー] で、テーブルに必要なスケーリングポリシーと、オプションでテーブルのすべてのグローバルセカンダリインデックスを設定します。

      - 最小キャパシティーユニット - Auto Scaling 範囲の下限を入力します。
      - 最大キャパシティーユニット - Auto Scaling 範囲の上限を入力します。
      - ターゲット使用率 — テーブルの目標使用率を入力します。

      > **Note**
      >
      > 新しいテーブルのグローバルセカンダリインデックスを作成する場合、作成時のインデックスの容量は、ベーステーブルの容量と同じになります。インデックスの容量は、テーブルの作成後にテーブルの設定で変更できます。

5. [テーブルの作成] を選択します。これにより、指定した自動スケーリングパラメータを使用してテーブルが作成されます。

既存のテーブルでの DynamoDB Auto Scaling の有効化

> **Note**
>
> DynamoDB Auto Scaling では、ユーザーに代わって Auto Scaling アクションを実行する、サービスリンクロール (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) の存在を必要とし

ます。このロールは自動的に作成されます。詳細については、「Application Auto Scaling 用のサービスリンクロール」を参照してください。

既存のテーブルに対して DynamoDB Auto Scaling を有効にするには

1. https://console.aws.amazon.com/dynamodb/ で DynamoDB コンソールを開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. 自動スケーリングを有効にするテーブルを選択し、以下の操作を行います。
   a. [追加の設定] タブを選択します。
   b. [読み取りキャパシティー] セクションで、[編集] を選択します。
   c. [キャパシティーモード] セクションで、[プロビジョンド] を選択します。
   d. [テーブルのキャパシティー] セクションで、[読み込みキャパシティー]、[書き込みキャパシティー]、または両方に対して、[自動スケーリング] を [オン] に設定します。これらのそれぞれについて、テーブルに必要なスケーリングポリシーを設定し、オプションでテーブルのすべてのグローバルセカンダリインデックスを設定します。
      - 最小キャパシティーユニット - Auto Scaling 範囲の下限を入力します。
      - 最大キャパシティーユニット - Auto Scaling 範囲の上限を入力します。
      - ターゲット使用率 — テーブルの目標使用率を入力します。
      - [すべてのグローバルセカンダリインデックスに同じキャパシティー読み取り/書き込みキャパシティー設定を使用する] - グローバルセカンダリインデックスがベーステーブルと同じ Auto Scaling ポリシーを使用するかどうかを選択します。

> **Note**
>
> 最高のパフォーマンスを実現するには、[すべてのグローバルセカンダリインデックスに同じ読み取り/書き込み容量設定を適用する] を有効にすることをお勧めします。このオプションを使用すれば、DynamoDB Auto Scaling は、ベーステーブル上のすべてのグローバルセカンダリインデックスを均一にスケーリングできます。これには、既存のグローバルセカンダリインデックスと、将来このテーブル用に作成するその他のインデックスが含まれます。
> このオプションを有効にすると、個々のグローバルセカンダリインデックスにスケーリングポリシーを設定できなくなります。

4. すべての設定が正しいことを確認したら、[保存] を選択します。

コンソールでの Auto Scaling アクティビティの表示

アプリケーションがテーブルへの読み込みおよび書き込みトラフィックを送ると、DynamoDB Auto Scaling はテーブルのスループット設定を動的に変更します。Amazon CloudWatch は、すべての DynamoDB テーブルとセカンダリインデックスについて、プロビジョニングされた容量、消費された容量、スロットルイベント、レイテンシー、およびその他のメトリクスを追跡します。

DynamoDB コンソールでこれらのメトリクスを表示するには、操作するテーブルを選択し、[モニタリング] タブを選択します。テーブルメトリクスのカスタマイズ可能なビューを作成するには、[CloudWatch ですべてを表示] を選択します。

DynamoDB Auto Scaling 設定の変更または無効化

AWS マネジメントコンソール を使用して、DynamoDB Auto Scaling 設定を変更できます。これを行うには、テーブルの [追加設定] タブに移動し、[読み取り/書き込みキャパシティー] セクションで [編集] を選択します。これらの設定の詳細については、「既存のテーブルでの DynamoDB Auto Scaling の有効化」をご参照ください。

AWS CLI を使用した DynamoDB Auto Scaling の管理

AWS マネジメントコンソール を使用する代わりに、AWS Command Line Interface (AWS CLI) を使用して、Amazon DynamoDB Auto Scaling を管理できます。このセクションのチュートリアルでは、DynamoDB Auto Scaling を管理するために AWS CLI をインストールして設定する方法を示します。このチュートリアルでは、以下の作業を行います。

- `TestTable` という DynamoDB テーブルの作成 初期スループット設定では、読み込み容量単位数が 5、書き込み容量単位が 5 です。
- `TestTable` 用アプリケーションの Auto Scaling ポリシーを作成します。このポリシーでは、プロビジョニングされた書き込みキャパシティに対する消費された書き込みキャパシティの割合を 50% に維持することを目指します。このメトリクスの範囲は 5 ~ 10 書き込みキャパシティユニットです。(Application Auto Scaling では、この範囲を超えるスループットを調整することはできません)。
- Python プログラムを実行して、書き込みトラフィックを `TestTable` に送ります。目標比率が一定期間 50% を超えると、Application Auto Scaling は DynamoDB に通知して `TestTable` のスループットを上方に調整して、50% の目標使用率を維持します。

- DynamoDB が `TestTable` のプロビジョンされた書き込み容量を正常に調整したことを確認します。

> **Note**
>
> また、DynamoDB のスケーリングを特定の時間に実行するようにスケジュールすることもできます。基本的な手順については、こちらを参照してください。

トピック

- 開始する前に
- ステップ 1: DynamoDB テーブルを作成する
- ステップ 2: スケーラブルなターゲットを登録する
- ステップ 3: スケーリングポリシーを作成する
- ステップ 4: 書き込みトラフィックを TestTable に送る
- ステップ 5: Application Auto Scaling アクションを表示する
- (オプション) ステップ 6: クリーンアップする

開始する前に

チュートリアルを開始する前に、以下のタスクを完了します。

AWS CLI をインストールする

まだ AWS CLI をインストールして設定していない場合は、インストールして設定する必要があります。これを行うには、AWS Command Line Interface のユーザーガイドの手順に従います。

- のインストールAWS CLI
- の設定AWS CLI

Python のインストール

このチュートリアルの一部では、Python プログラムを実行する必要があります (ステップ 4: 書き込みトラフィックを TestTable に送る を参照)。まだインスタンスしていない場合は、Python をダウンロードできます。

ステップ 1: DynamoDB テーブルを作成する

このステップでは、AWS CLI を使用して TestTable を作成します。プライマリキーは、pk (パーティションキー) と sk (ソートキー) で構成されます。これらの属性はいずれも、Number 型になります。初期スループット設定では、読み込み容量単位数が 5、書き込み容量単位が 5 です。

1. 以下の AWS CLI コマンドを使用してターゲットを作成します。

```
aws dynamodb create-table \
    --table-name TestTable \
    --attribute-definitions \
        AttributeName=pk,AttributeType=N \
        AttributeName=sk,AttributeType=N \
    --key-schema \
        AttributeName=pk,KeyType=HASH \
        AttributeName=sk,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

2. テーブルのステータスを確認するには、次のコマンドを使用します。

```
aws dynamodb describe-table \
    --table-name TestTable \
    --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

   ステータスが ACTIVE になったら、テーブルは使用できる状態になります。

ステップ 2: スケーラブルなターゲットを登録する

次に、Application Auto Scaling を使用して、テーブルの書き込み容量をスケーラブルなターゲットとして登録します。これにより、Application Auto Scaling は、TestTable にプロビジョンされた書き込み容量を調整できますが、容量単位は 5 ~ 10 の範囲内に限られます。

> **Note**
>
> DynamoDB Auto Scaling では、ユーザーに代わって Auto Scaling アクションを実行する、サービスリンクロール (AWSServiceRoleForApplicationAutoScaling_DynamoDBTable) の存在を必要とします。このロールは自動的に作成されます。詳細については、「Application Auto Scaling

ユーザーガイド」の「Application Auto Scaling 用のサービスリンクロール」を参照してください。

1. 次のコマンドを入力して、スケーラブルなターゲットを登録します。

```
aws application-autoscaling register-scalable-target \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --min-capacity 5 \
    --max-capacity 10
```

2. 次のコマンドを使用して登録を確認します。

```
aws application-autoscaling describe-scalable-targets \
    --service-namespace dynamodb \
    --resource-id "table/TestTable"
```

> **Note**
>
> グローバルセカンダリインデックスに対してスケーラブルターゲットを登録することもできます。たとえば、グローバルセカンダリインデックス (「test-index」) の場合、リソース ID とスケーラブルディメンションの引数は適切に更新されます。
>
> ```
> aws application-autoscaling register-scalable-target \
>     --service-namespace dynamodb \
>     --resource-id "table/TestTable/index/test-index" \
>     --scalable-dimension "dynamodb:index:WriteCapacityUnits" \
>     --min-capacity 5 \
>     --max-capacity 10
> ```

ステップ 3: スケーリングポリシーを作成する

このステップでは、TestTable のスケーリングポリシーを作成します。ポリシーでは、Application Auto Scaling がテーブルのプロビジョンされたスループットを調整できる詳細と、その場合に実行するアクションを定義します。このポリシーは、前のステップで定義したスケーラブルターゲットに関連付けます (TestTable テーブルの容量単位を書き込みます)。

ポリシーには、次の要素が含まれます。

- `PredefinedMetricSpecification` — Application Auto Scaling が調整できるメトリクス。DynamoDB の場合、次の値が `PredefinedMetricType` の有効な値です。
  - `DynamoDBReadCapacityUtilization`
  - `DynamoDBWriteCapacityUtilization`
- `ScaleOutCooldown` — プロビジョンされたスループットを増加させる各 Application Auto Scaling イベント間の最小時間 (秒単位)。このパラメータを使用すると、Application Auto Scaling は、実際のワークロードに応じてスループットを継続的に増加させますが、積極的に増加させることはできません。`ScaleOutCooldown` のデフォルトの設定は 0 です。
- `ScaleInCooldown` — プロビジョンされたスループットを低下させる各 Application Auto Scaling イベント間の最小時間 (秒単位)。このパラメータを使用すると、Application Auto Scaling はスループットを徐々に、予測どおりに低下させることができます。`ScaleInCooldown` のデフォルトの設定は 0 です。
- `TargetValue` — Application Auto Scaling は、プロビジョンされた容量に対する消費容量の比率がこの値またはその近くに留まるようにします。`TargetValue` をパーセンテージとして定義します。

> **Note**
>
> `TargetValue` がどのように機能するかをさらに理解するために、書き込み容量単位が 200 で、プロビジョンされたスループット設定を持つテーブルがあるとします。このテーブルのスケーリングポリシーを作成することにしました。`TargetValue` は 70% です。
> ここで、実際の書き込みスループットが 150 容量単位になるように、テーブルへの書き込みトラフィックを駆動し始めたとします。消費とプロビジョンの比率は現在 (150/200)、つまり 75% です。この比率は目標を超えているため、Application Auto Scaling はプロビジョンされた書き込み容量を 215 に増やし、比率が (150/215)、つまり 69.77% になるようにします。これは、可能な限り `TargetValue` に近いですが、超えないようにしてください。

`TestTable` の場合、`TargetValue` を 50% に設定します。Application Auto Scaling は、テーブルのプロビジョンされたスループットを 5 ~ 10 容量単位の範囲内で調整し (ステップ 2: スケーラブルなターゲットを登録する を参照)、プロビジョンされた消費量とプロビジョンされた比率が 50% またはそれに近いままになるようにします。`ScaleOutCooldown` と `ScaleInCooldown` の値を 60 秒に設定します。

1. 次の内容で、scaling-policy.json というファイルを作成します。

```
{
    "PredefinedMetricSpecification": {
        "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
    },
    "ScaleOutCooldown": 60,
    "ScaleInCooldown": 60,
    "TargetValue": 50.0
}
```

2. 次の AWS CLI コマンドを使用してポリシーを作成します。

```
aws application-autoscaling put-scaling-policy \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --policy-name "MyScalingPolicy" \
    --policy-type "TargetTrackingScaling" \
    --target-tracking-scaling-policy-configuration file://scaling-policy.json
```

3. 出力では、Application Auto Scaling が 2 つの Amazon CloudWatch アラームを作成したことに注意してください。スケーリング目標範囲の上限と下限に対応します。
4. 以下の AWS CLI コマンドを使用して、スケーリングポリシーの詳細を表示します。

```
aws application-autoscaling describe-scaling-policies \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --policy-name "MyScalingPolicy"
```

5. 出力で、ポリシー設定が ステップ 2: スケーラブルなターゲットを登録する と ステップ 3: スケーリングポリシーを作成する からの仕様と一致することを確認します。

ステップ 4: 書き込みトラフィックを TestTable に送る

これで、TestTable にデータを書き込むことでスケーリングポリシーをテストできます。これを行うには、Python プログラムを実行します。

1. 次の内容で、bulk-load-test-table.py というファイルを作成します。

```
import boto3
```

```
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table("TestTable")

filler = "x" * 100000

i = 0
while (i < 10):
    j = 0
    while (j < 10):
        print (i, j)

        table.put_item(
            Item={
                'pk':i,
                'sk':j,
                'filler':{"S":filler}
            }
        )
        j += 1
    i += 1
```

2. プログラムを実行するには、次のコマンドを入力します。

   ```
   python bulk-load-test-table.py
   ```

   TestTable のプロビジョンされた書き込み容量は非常に低いため (5 書き込み容量単位)、書き込みスロットリングが原因でプログラムが停止することがあります。これは想定される動作です。

   次のステップに進む間、プログラムを実行し続けます。

ステップ 5: Application Auto Scaling アクションを表示する

このステップでは、ユーザーに代わって開始された Application Auto Scaling アクションを表示します。また、Application Auto Scaling が TestTable のプロビジョンされた書き込み容量を更新したことを確認します。

1. 次のコマンドを入力して、Application Auto Scaling アクションを表示します。

   ```
   aws application-autoscaling describe-scaling-activities \
       --service-namespace dynamodb
   ```

Python プログラムの実行中に、このコマンドを時折再実行します。(スケーリングポリシーが呼び出されるまで数分かかります。) 結果的に、次のような出力が表示されます。

```
...
{
    "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
    "Description": "Setting write capacity units to 10.",
    "ResourceId": "table/TestTable",
    "ActivityId": "0cc6fb03-2a7c-4b51-b67f-217224c6b656",
    "StartTime": 1489088210.175,
    "ServiceNamespace": "dynamodb",
    "EndTime": 1489088246.85,
    "Cause": "monitor alarm AutoScaling-table/TestTable-
AlarmHigh-1bb3c8db-1b97-4353-baf1-4def76f4e1b9 in state ALARM triggered policy
 MyScalingPolicy",
    "StatusMessage": "Successfully set write capacity units to 10. Change
 successfully fulfilled by dynamodb.",
    "StatusCode": "Successful"
},
...
```

これは、Application Auto Scaling が DynamoDB に `UpdateTable` リクエストを発行したことを示しています。

2. DynamoDB がテーブルの書き込み容量を増やしたことを確認するには、次のコマンドを入力します。

```
aws dynamodb describe-table \
    --table-name TestTable \
    --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

`WriteCapacityUnits` は `5` から `10` にスケーリングされている必要があります。

(オプション) ステップ 6: クリーンアップする

このチュートリアルでは、いくつかのリソースを作成しました。これらのリソースが不要になった場合は、削除できます。

1. `TestTable` のスケーリングポリシーを削除します。

```
aws application-autoscaling delete-scaling-policy \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits" \
    --policy-name "MyScalingPolicy"
```

2. スケーラブルなターゲットを登録解除します。

```
aws application-autoscaling deregister-scalable-target \
    --service-namespace dynamodb \
    --resource-id "table/TestTable" \
    --scalable-dimension "dynamodb:table:WriteCapacityUnits"
```

3. TestTable テーブルを削除します。

```
aws dynamodb delete-table --table-name TestTable
```

AWS SDK を使用した Amazon DynamoDB テーブルでの Auto Scaling の設定

AWS マネジメントコンソール と AWS Command Line Interface (AWS CLI) を使用するほかに、Amazon DynamoDB Auto Scaling とやり取りするアプリケーションを作成することもできます。このセクションには、この機能をテストするために使用できる 2 つの Java プログラムが含まれています。

- EnableDynamoDBAutoscaling.java
- DisableDynamoDBAutoscaling.java

テーブルの Application Auto Scaling の有効化

次のプログラムは、DynamoDB テーブル (TestTable) の Auto Scaling ポリシーをセットアップする例を示しています。プログラムの流れは次のようになります。

- TestTable のスケーラブルなターゲットとして書き込みキャパシティユニットを登録します。このメトリクスの範囲は 5 ~ 10 書き込みキャパシティーユニットです。
- スケーラブルなターゲットを作成したら、次にターゲットの追跡設定を作成します。このポリシーでは、プロビジョニングされた書き込みキャパシティに対する消費された書き込みキャパシティの割合を 50% に維持することを目指します。

- 次に、ターゲットの追跡設定に基づいてスケーリングポリシーを作成します。

> **Note**
>
> テーブルまたはグローバルテーブルレプリカを手動で削除しても、関連するスケーラブルターゲット、スケーリングポリシー、または CloudWatch アラームは自動的には削除されません。

Java v2

```
import software.amazon.awssdk.regions.Region;
import
 software.amazon.awssdk.services.applicationautoscaling.ApplicationAutoScalingClient;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ApplicationAutoScalingExceptio
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRespons
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRespons
import software.amazon.awssdk.services.applicationautoscaling.model.PolicyType;
import
 software.amazon.awssdk.services.applicationautoscaling.model.PredefinedMetricSpecification;
import
 software.amazon.awssdk.services.applicationautoscaling.model.PutScalingPolicyRequest;
import
 software.amazon.awssdk.services.applicationautoscaling.model.RegisterScalableTargetRequest;
import software.amazon.awssdk.services.applicationautoscaling.model.ScalingPolicy;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ServiceNamespace;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ScalableDimension;
import software.amazon.awssdk.services.applicationautoscaling.model.MetricType;
import
 software.amazon.awssdk.services.applicationautoscaling.model.TargetTrackingScalingPolicyCon
import java.util.List;

/**
```

```
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class EnableDynamoDBAutoscaling {
    public static void main(String[] args) {
        final String usage = """

            Usage:
               <tableId> <roleARN> <policyName>\s

            Where:
               tableId - The table Id value (for example, table/Music).
               roleARN - The ARN of the role that has ApplicationAutoScaling
 permissions.
               policyName - The name of the policy to create.

            """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        System.out.println("This example registers an Amazon DynamoDB table, which
 is the resource to scale.");
        String tableId = args[0];
        String roleARN = args[1];
        String policyName = args[2];
        ServiceNamespace ns = ServiceNamespace.DYNAMODB;
        ScalableDimension tableWCUs =
 ScalableDimension.DYNAMODB_TABLE_WRITE_CAPACITY_UNITS;
        ApplicationAutoScalingClient appAutoScalingClient =
 ApplicationAutoScalingClient.builder()
            .region(Region.US_EAST_1)
            .build();

        registerScalableTarget(appAutoScalingClient, tableId, roleARN, ns,
 tableWCUs);
        verifyTarget(appAutoScalingClient, tableId, ns, tableWCUs);
```

```
        configureScalingPolicy(appAutoScalingClient, tableId, ns, tableWCUs,
policyName);
    }

    public static void registerScalableTarget(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, String roleARN, ServiceNamespace ns,
ScalableDimension tableWCUs) {
        try {
            RegisterScalableTargetRequest targetRequest =
RegisterScalableTargetRequest.builder()
                .serviceNamespace(ns)
                .scalableDimension(tableWCUs)
                .resourceId(tableId)
                .roleARN(roleARN)
                .minCapacity(5)
                .maxCapacity(10)
                .build();

            appAutoScalingClient.registerScalableTarget(targetRequest);
            System.out.println("You have registered " + tableId);

        } catch (ApplicationAutoScalingException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
        }
    }

    // Verify that the target was created.
    public static void verifyTarget(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs) {
        DescribeScalableTargetsRequest dscRequest =
DescribeScalableTargetsRequest.builder()
            .scalableDimension(tableWCUs)
            .serviceNamespace(ns)
            .resourceIds(tableId)
            .build();

        DescribeScalableTargetsResponse response =
appAutoScalingClient.describeScalableTargets(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(response);
    }

    // Configure a scaling policy.
```

```
   public static void configureScalingPolicy(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs, String policyName) {
       // Check if the policy exists before creating a new one.
       DescribeScalingPoliciesResponse describeScalingPoliciesResponse =
appAutoScalingClient.describeScalingPolicies(DescribeScalingPoliciesRequest.builder()
           .serviceNamespace(ns)
           .resourceId(tableId)
           .scalableDimension(tableWCUs)
           .build());

       if (!describeScalingPoliciesResponse.scalingPolicies().isEmpty()) {
           // If policies exist, consider updating an existing policy instead of
creating a new one.
           System.out.println("Policy already exists. Consider updating it
instead.");
           List<ScalingPolicy> polList =
describeScalingPoliciesResponse.scalingPolicies();
           for (ScalingPolicy pol : polList) {
               System.out.println("Policy name:" +pol.policyName());
           }
       } else {
           // If no policies exist, proceed with creating a new policy.
           PredefinedMetricSpecification specification =
PredefinedMetricSpecification.builder()

.predefinedMetricType(MetricType.DYNAMO_DB_WRITE_CAPACITY_UTILIZATION)
               .build();

           TargetTrackingScalingPolicyConfiguration policyConfiguration =
TargetTrackingScalingPolicyConfiguration.builder()
               .predefinedMetricSpecification(specification)
               .targetValue(50.0)
               .scaleInCooldown(60)
               .scaleOutCooldown(60)
               .build();

           PutScalingPolicyRequest putScalingPolicyRequest =
PutScalingPolicyRequest.builder()
               .targetTrackingScalingPolicyConfiguration(policyConfiguration)
               .serviceNamespace(ns)
               .scalableDimension(tableWCUs)
               .resourceId(tableId)
               .policyName(policyName)
```

```
                .policyType(PolicyType.TARGET_TRACKING_SCALING)
                .build();

            try {
                appAutoScalingClient.putScalingPolicy(putScalingPolicyRequest);
                System.out.println("You have successfully created a scaling policy
 for an Application Auto Scaling scalable target");
            } catch (ApplicationAutoScalingException e) {
                System.err.println("Error: " + e.awsErrorDetails().errorMessage());
            }
        }
    }
}
```

Java v1

プログラムでは、有効な Application Auto Scaling サービスリンクロールの Amazon リソースネーム (ARN) を指定する必要があります。(例: `arn:aws:iam::122517410325:role/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`。) 次のプログラムの `SERVICE_ROLE_ARN_GOES_HERE` を実際の ARN に置き換えます。

```
package com.amazonaws.codesamples.autoscaling;

import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClientBuilder;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.MetricType;
import com.amazonaws.services.applicationautoscaling.model.PolicyType;
import
 com.amazonaws.services.applicationautoscaling.model.PredefinedMetricSpecification;
import com.amazonaws.services.applicationautoscaling.model.PutScalingPolicyRequest;
import
 com.amazonaws.services.applicationautoscaling.model.RegisterScalableTargetRequest;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
```

```
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;
import
 com.amazonaws.services.applicationautoscaling.model.TargetTrackingScalingPolicyConfiguratio

public class EnableDynamoDBAutoscaling {

 static AWSApplicationAutoScalingClient aaClient = (AWSApplicationAutoScalingClient)
 AWSApplicationAutoScalingClientBuilder
   .standard().build();

 public static void main(String args[]) {

  ServiceNamespace ns = ServiceNamespace.Dynamodb;
  ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
  String resourceID = "table/TestTable";

  // Define the scalable target
  RegisterScalableTargetRequest rstRequest = new RegisterScalableTargetRequest()
    .withServiceNamespace(ns)
    .withResourceId(resourceID)
    .withScalableDimension(tableWCUs)
    .withMinCapacity(5)
    .withMaxCapacity(10)
    .withRoleARN("SERVICE_ROLE_ARN_GOES_HERE");

  try {
   aaClient.registerScalableTarget(rstRequest);
  } catch (Exception e) {
   System.err.println("Unable to register scalable target: ");
   System.err.println(e.getMessage());
  }

  // Verify that the target was created
  DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceIds(resourceID);
  try {
   DescribeScalableTargetsResult dsaResult =
 aaClient.describeScalableTargets(dscRequest);
   System.out.println("DescribeScalableTargets result: ");
   System.out.println(dsaResult);
   System.out.println();
  } catch (Exception e) {
```

```
  System.err.println("Unable to describe scalable target: ");
  System.err.println(e.getMessage());
 }

 System.out.println();

 // Configure a scaling policy
 TargetTrackingScalingPolicyConfiguration targetTrackingScalingPolicyConfiguration
= new TargetTrackingScalingPolicyConfiguration()
    .withPredefinedMetricSpecification(
      new PredefinedMetricSpecification()
        .withPredefinedMetricType(MetricType.DynamoDBWriteCapacityUtilization))
    .withTargetValue(50.0)
    .withScaleInCooldown(60)
    .withScaleOutCooldown(60);

 // Create the scaling policy, based on your configuration
 PutScalingPolicyRequest pspRequest = new PutScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID)
    .withPolicyName("MyScalingPolicy")
    .withPolicyType(PolicyType.TargetTrackingScaling)

.withTargetTrackingScalingPolicyConfiguration(targetTrackingScalingPolicyConfiguration);

 try {
  aaClient.putScalingPolicy(pspRequest);
 } catch (Exception e) {
  System.err.println("Unable to put scaling policy: ");
  System.err.println(e.getMessage());
 }

 // Verify that the scaling policy was created
 DescribeScalingPoliciesRequest dspRequest = new DescribeScalingPoliciesRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID);

 try {
  DescribeScalingPoliciesResult dspResult =
aaClient.describeScalingPolicies(dspRequest);
  System.out.println("DescribeScalingPolicies result: ");
  System.out.println(dspResult);
```

```
  } catch (Exception e) {
   e.printStackTrace();
   System.err.println("Unable to describe scaling policy: ");
   System.err.println(e.getMessage());
  }

 }

}
```

テーブルの Application Auto Scaling の無効化

次のプログラムは、前述のプロセスを元に戻します。Auto Scaling ポリシーを削除し、スケーラブルなターゲットを登録解除します。

Java v2

```
import software.amazon.awssdk.regions.Region;
import
 software.amazon.awssdk.services.applicationautoscaling.ApplicationAutoScalingClient;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ApplicationAutoScalingExceptio
import
 software.amazon.awssdk.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import
 software.amazon.awssdk.services.applicationautoscaling.model.DeregisterScalableTargetReques
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalableTargetsRespons
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRequest
import
 software.amazon.awssdk.services.applicationautoscaling.model.DescribeScalingPoliciesRespons
import
 software.amazon.awssdk.services.applicationautoscaling.model.ScalableDimension;
import
 software.amazon.awssdk.services.applicationautoscaling.model.ServiceNamespace;

/**
```

```
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class DisableDynamoDBAutoscaling {
    public static void main(String[] args) {
        final String usage = """

            Usage:
               <tableId> <policyName>\s

            Where:
               tableId - The table Id value (for example, table/Music).\s
               policyName - The name of the policy (for example, $Music5-scaling-
policy).

            """;
        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        ApplicationAutoScalingClient appAutoScalingClient =
 ApplicationAutoScalingClient.builder()
            .region(Region.US_EAST_1)
            .build();

        ServiceNamespace ns = ServiceNamespace.DYNAMODB;
        ScalableDimension tableWCUs =
 ScalableDimension.DYNAMODB_TABLE_WRITE_CAPACITY_UNITS;
        String tableId = args[0];
        String policyName = args[1];

        deletePolicy(appAutoScalingClient, policyName, tableWCUs, ns, tableId);
        verifyScalingPolicies(appAutoScalingClient, tableId, ns, tableWCUs);
        deregisterScalableTarget(appAutoScalingClient, tableId, ns, tableWCUs);
        verifyTarget(appAutoScalingClient, tableId, ns, tableWCUs);
    }
```

```
   public static void deletePolicy(ApplicationAutoScalingClient
appAutoScalingClient, String policyName, ScalableDimension tableWCUs,
ServiceNamespace ns, String tableId) {
       try {
           DeleteScalingPolicyRequest delSPRequest =
DeleteScalingPolicyRequest.builder()
               .policyName(policyName)
               .scalableDimension(tableWCUs)
               .serviceNamespace(ns)
               .resourceId(tableId)
               .build();

           appAutoScalingClient.deleteScalingPolicy(delSPRequest);
           System.out.println(policyName +" was deleted successfully.");

       } catch (ApplicationAutoScalingException e) {
           System.err.println(e.awsErrorDetails().errorMessage());
       }
   }

   // Verify that the scaling policy was deleted
   public static void verifyScalingPolicies(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs) {
       DescribeScalingPoliciesRequest dscRequest =
DescribeScalingPoliciesRequest.builder()
           .scalableDimension(tableWCUs)
           .serviceNamespace(ns)
           .resourceId(tableId)
           .build();

       DescribeScalingPoliciesResponse response =
appAutoScalingClient.describeScalingPolicies(dscRequest);
       System.out.println("DescribeScalableTargets result: ");
       System.out.println(response);
   }

   public static void deregisterScalableTarget(ApplicationAutoScalingClient
appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
tableWCUs) {
       try {
           DeregisterScalableTargetRequest targetRequest =
DeregisterScalableTargetRequest.builder()
               .scalableDimension(tableWCUs)
```

```
                .serviceNamespace(ns)
                .resourceId(tableId)
                .build();

            appAutoScalingClient.deregisterScalableTarget(targetRequest);
            System.out.println("The scalable target was deregistered.");

        } catch (ApplicationAutoScalingException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
        }
    }

    public static void verifyTarget(ApplicationAutoScalingClient
 appAutoScalingClient, String tableId, ServiceNamespace ns, ScalableDimension
 tableWCUs) {
        DescribeScalableTargetsRequest dscRequest =
 DescribeScalableTargetsRequest.builder()
            .scalableDimension(tableWCUs)
            .serviceNamespace(ns)
            .resourceIds(tableId)
            .build();

        DescribeScalableTargetsResponse response =
 appAutoScalingClient.describeScalableTargets(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(response);
    }
}
```

Java v1

```
package com.amazonaws.codesamples.autoscaling;

import
 com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
 com.amazonaws.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DeregisterScalableTargetRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
```

```
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import
 com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;

public class DisableDynamoDBAutoscaling {

 static AWSApplicationAutoScalingClient aaClient = new
 AWSApplicationAutoScalingClient();

 public static void main(String args[]) {

  ServiceNamespace ns = ServiceNamespace.Dynamodb;
  ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
  String resourceID = "table/TestTable";

  // Delete the scaling policy
  DeleteScalingPolicyRequest delSPRequest = new DeleteScalingPolicyRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID)
    .withPolicyName("MyScalingPolicy");

  try {
   aaClient.deleteScalingPolicy(delSPRequest);
  } catch (Exception e) {
   System.err.println("Unable to delete scaling policy: ");
   System.err.println(e.getMessage());
  }

  // Verify that the scaling policy was deleted
  DescribeScalingPoliciesRequest descSPRequest = new
 DescribeScalingPoliciesRequest()
    .withServiceNamespace(ns)
    .withScalableDimension(tableWCUs)
    .withResourceId(resourceID);

  try {
   DescribeScalingPoliciesResult dspResult =
 aaClient.describeScalingPolicies(descSPRequest);
   System.out.println("DescribeScalingPolicies result: ");
   System.out.println(dspResult);
```

```
  } catch (Exception e) {
   e.printStackTrace();
   System.err.println("Unable to describe scaling policy: ");
   System.err.println(e.getMessage());
  }

  System.out.println();

  // Remove the scalable target
  DeregisterScalableTargetRequest delSTRequest = new
 DeregisterScalableTargetRequest()
     .withServiceNamespace(ns)
     .withScalableDimension(tableWCUs)
     .withResourceId(resourceID);

  try {
   aaClient.deregisterScalableTarget(delSTRequest);
  } catch (Exception e) {
   System.err.println("Unable to deregister scalable target: ");
   System.err.println(e.getMessage());
  }

  // Verify that the scalable target was removed
  DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
     .withServiceNamespace(ns)
     .withScalableDimension(tableWCUs)
     .withResourceIds(resourceID);

  try {
   DescribeScalableTargetsResult dsaResult =
 aaClient.describeScalableTargets(dscRequest);
   System.out.println("DescribeScalableTargets result: ");
   System.out.println(dsaResult);
   System.out.println();
  } catch (Exception e) {
   System.err.println("Unable to describe scalable target: ");
   System.err.println(e.getMessage());
  }

 }

}
```

DynamoDB リザーブドキャパシティ

標準のテーブルクラスを使用するプロビジョンドキャパシティテーブルの場合、DynamoDB は読み込みおよび書き込みキャパシティ用にリザーブドキャパシティを購入する機能を提供しています。リザーブドキャパシティの購入は、割引料金と引き換えに、契約期間中のプロビジョニングされたスループットキャパシティの最小量に対して支払う契約です。

> **Note**
>
> レプリケートされた書き込みキャパシティユニット (rWCU) に対しては、リザーブドキャパシティを購入できません。リザーブドキャパシティは、キャパシティを購入したリージョンにのみ適用されます。リザーブドキャパシティは、DynamoDB 標準-IA テーブルクラスまたはオンデマンドキャパシティモードを使用するテーブルでも使用できません。

リザーブドキャパシティは、100 個の WCU または 100 個の RCU の割り当て単位で購入します。最小のリザーブドキャパシティは、100 個のキャパシティユニット (読み込みまたは書き込み) です。DynamoDB リザーブドキャパシティは、1 年間のコミットメント、または限定リージョンでの 3 年間のコミットメントとして提供されます。1 年契約では標準料金から最大 54%、3 年契約では標準料金から最大 77% の割引を受けることができます。購入方法と購入時期の詳細については、「Amazon DynamoDB リザーブドキャパシティ」を参照してください。

> **Note**
>
> AWS マネジメントコンソールを使用して、書き込みキャパシティーユニット (WCU) と読み取りキャパシティユニット (RCU) を合わせて最大 1,000,000 のリザーブドキャパシティユニットを購入できます。1 回の購入で 1,000,000 を超えるプロビジョンドキャパシティユニットを購入する場合、またはアクティブなリザーブドキャパシティがあり、追加のリザーブドキャパシティの購入によってアクティブなプロビジョンドキャパシティユニットの合計が 1,000,000 を超える場合は、「Amazon DynamoDB リザーブドキャパシティ」の「リザーブドキャパシティの購入方法」セクションに記載されているプロセスに従います。

DynamoDB リザーブドキャパシティを購入すると、1 回限りの一部前払い料金を支払うことで、コミットしたプロビジョンド使用量を割引時間料金で利用できます。実際の使用量に関係なく、コミットしたプロビジョンド使用量全体に料金を支払うため、コスト削減は使用量と密接に関連します。購入したリザーブドキャパシティを超えてプロビジョニングしたキャパシティには、標準のプロビジョンドキャパシティの料金が請求されます。読み込みキャパシティーユニットおよび書き込みキャパシ

ティーユニットを事前に予約することで、プロビジョニングされたキャパシティーコストの大幅なコスト削減を実現できます。

リザーブドキャパシティを販売またはキャンセルしたり、別のリージョンやアカウントに移管したりすることはできません。

DynamoDB ウォームスループットについて

ウォームスループットとは、DynamoDB テーブルが瞬時にサポートできる読み取りおよび書き込みオペレーションの数を指します。これらの値は、すべてのテーブルとグローバルセカンダリインデックス (GSI) でデフォルトで使用でき、履歴の使用に基づいてスケーリングされた量を表します。オンデマンドモードを使用している場合、またはプロビジョンドスループットをこれらの値に更新した場合、アプリケーションはこれらの値まで即座にリクエストを発行できます。

DynamoDB は、使用量が増えるとウォームスループット値を自動的に調整します。これらの値は、必要に応じて事前に増やすこともできます。これは、製品のローンチや販売などの今後のピークイベントに特に役立ちます。計画されたピークイベントで、DynamoDB テーブルへのリクエストレートが 10 倍、100 倍、またはそれ以上増加する可能性がある場合、現在のウォームスループットが予想されるトラフィックを処理するのに十分かどうかを評価できるようになりました。そうでない場合は、スループット設定や請求モードを変更せずにウォームスループット値を増やすことができます。このプロセスはテーブルの事前ウォーミングと呼ばれ、テーブルがすぐにサポートできるベースラインを設定できます。これにより、アプリケーションは、発生した瞬間からより高いリクエストレートを処理できます。一度増やした後は、ウォームスループット値を減らすことはできません。

読み取りオペレーション、書き込みオペレーション、またはその両方のウォームスループット値を増やすことができます。この値は、新規および既存の単一リージョンテーブル、グローバルテーブル、GSI で増やすことができます。グローバルテーブルの場合、この機能はバージョン 2019.11.21 (現行) で使用でき、設定したウォームスループット設定はグローバルテーブル内のすべてのレプリカテーブルに自動的に適用されます。いつでも事前ウォーミングできる DynamoDB テーブルの数に制限はありません。事前ウォーミングを完了する時間は、設定した値と、テーブルまたはインデックスのサイズによって異なります。事前ウォーミングの同時リクエストを送信できます。これらのリクエストはテーブルオペレーションを妨げません。そのリージョンのアカウントのテーブルまたはインデックスクォータ制限まで、テーブルを事前ウォーミングできます。Service Quotas コンソールを使用して現在の制限を確認し、必要に応じて引き上げます。

ウォームスループット値は、デフォルトですべてのテーブルとセカンダリインデックスで無料で使用できます。ただし、これらのデフォルトのウォームスループット値をプロアクティブに増やしてテーブルを事前ウォーミングすると、それらのリクエストに対して課金されます。詳細については、「Amazon DynamoDB 料金」を参照してください。

ウォームスループットの詳細については、以下のトピックを参照してください。

トピック

- DynamoDB テーブルの現在のウォームスループットを確認する
- 既存の DynamoDB テーブルのウォームスループットを増やす
- ウォームスループットが高い新しい DynamoDB テーブルを作成する
- さまざまなシナリオでの DynamoDB ウォームスループットについて

DynamoDB テーブルの現在のウォームスループットを確認する

テーブルまたはインデックスの現在のウォームスループット値を確認するには、次の AWS CLI および AWS コンソールの手順を使用します。

AWS マネジメントコンソール

DynamoDB コンソールを使用して DynamoDB テーブルのウォームスループットを確認するには:

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. [テーブル] ページで、目的のテーブルを選択します。
4. テーブルの詳細ページの [ウォームスループット] フィールドで、現在のウォームスループット値を確認します。この値は、1 秒あたりの読み取り単位と 1 秒あたりの書き込み単位として表示されます。

AWS CLI

次の AWS CLI の例は、DynamoDB テーブルのウォームスループットを確認する方法を示しています。

1. DynamoDB テーブルで `describe-table` オペレーションを実行します。

```
aws dynamodb describe-table --table-name GameScores
```

2. 次のようなレスポンスが表示されます。`WarmThroughput` 設定は、`ReadUnitsPerSecond` および `WriteUnitsPerSecond` として表示されます。ウォームスループット値が更新されてい

る場合、Status は UPDATING になり、新しいウォームスループット値が設定されている場合は、ACTIVE になります。

```
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1726128388.729,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST",
```

```
            "LastUpdateToPayPerRequestDateTime": 1726128388.729
        },
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 0,
                    "WriteCapacityUnits": 0
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 12000,
                    "WriteUnitsPerSecond": 4000,
                    "Status": "ACTIVE"
                }
            }
        ],
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12000,
            "WriteUnitsPerSecond": 4000,
            "Status": "ACTIVE"
        }
```

```
    }
}
```

既存の DynamoDB テーブルのウォームスループットを増やす

DynamoDB テーブルの現在のウォームスループット値を確認した後に、次のステップで値を更新できます。

AWS マネジメントコンソール

DynamoDB コンソールを使用して DynamoDB テーブルのウォームスループット値を確認するには:

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 左のナビゲーションペインで、[テーブル] を選択します。
3. [テーブル] ページで、目的のテーブルを選択します。
4. [ウォームスループット] フィールドで、[編集] を選択します。
5. [ウォームスループットを編集] ページで、[ウォームスループットを増やす] を選択します。
6. [1 秒あたりの読み取りユニット] と [1 秒あたりの書き込みユニット] を調整します。これら 2 つの設定は、テーブルが即座に処理できるスループットを定義します。
7. [保存] を選択します。
8. リクエストの処理が完了すると、[ウォームスループット] フィールドで [1 秒あたりの読み取りユニット] と [1 秒あたりの書き込みユニット] が更新されます。

> **Note**
>
> ウォームスループット値の更新は非同期タスクです。更新が完了すると、`Status` は `UPDATING` から `ACTIVE` に変わります。

AWS CLI

次の AWS CLI の例は、DynamoDB テーブルのウォームスループット値を更新する方法を示しています。

1. DynamoDB テーブルで `update-table` オペレーションを実行します。

```
aws dynamodb update-table \
    --table-name GameScores \
    --warm-throughput ReadUnitsPerSecond=12345,WriteUnitsPerSecond=4567 \
    --global-secondary-index-updates \
        "[
            {
                \"Update\": {
                    \"IndexName\": \"GameTitleIndex\",
                    \"WarmThroughput\": {
                        \"ReadUnitsPerSecond\": 88,
                        \"WriteUnitsPerSecond\": 77
                    }
                }
            }
        ]" \
    --region us-east-1
```

2. 次のようなレスポンスが表示されます。WarmThroughput 設定は、ReadUnitsPerSecond および WriteUnitsPerSecond として表示されます。ウォームスループット値が更新されている場合、Status は UPDATING になり、新しいウォームスループット値が設定されている場合は、ACTIVE になります。

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
```

```
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1730242189.965,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 20,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 50,
                    "WriteCapacityUnits": 25
                },
                "IndexSizeBytes": 0,
```

```
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 50,
                    "WriteUnitsPerSecond": 25,
                    "Status": "UPDATING"
                }
            }
        ],
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12300,
            "WriteUnitsPerSecond": 4500,
            "Status": "UPDATING"
        }
    }
}
```

AWS SDK

次の SDK の例は、DynamoDB テーブルのウォームスループット値を更新する方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndexUpdate;
import
 software.amazon.awssdk.services.dynamodb.model.UpdateGlobalSecondaryIndexAction;
import software.amazon.awssdk.services.dynamodb.model.UpdateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
```

```
            .build();
    }

    /**
     * Updates a DynamoDB table with warm throughput settings for both the table
and a global secondary index.
     *
     * @param ddb The DynamoDB client
     * @param tableName The name of the table to update
     * @param tableReadUnitsPerSecond Read units per second for the table
     * @param tableWriteUnitsPerSecond Write units per second for the table
     * @param globalSecondaryIndexName The name of the global secondary index to
update
     * @param globalSecondaryIndexReadUnitsPerSecond Read units per second for
the GSI
     * @param globalSecondaryIndexWriteUnitsPerSecond Write units per second for
the GSI
     */
    public static void updateDynamoDBTable(
        final DynamoDbClient ddb,
        final String tableName,
        final Long tableReadUnitsPerSecond,
        final Long tableWriteUnitsPerSecond,
        final String globalSecondaryIndexName,
        final Long globalSecondaryIndexReadUnitsPerSecond,
        final Long globalSecondaryIndexWriteUnitsPerSecond) {

        final WarmThroughput tableWarmThroughput =
            buildWarmThroughput(tableReadUnitsPerSecond,
tableWriteUnitsPerSecond);
        final WarmThroughput gsiWarmThroughput =
            buildWarmThroughput(globalSecondaryIndexReadUnitsPerSecond,
globalSecondaryIndexWriteUnitsPerSecond);

        final GlobalSecondaryIndexUpdate globalSecondaryIndexUpdate =
GlobalSecondaryIndexUpdate.builder()
            .update(UpdateGlobalSecondaryIndexAction.builder()
                .indexName(globalSecondaryIndexName)
                .warmThroughput(gsiWarmThroughput)
                .build())
            .build();

        final UpdateTableRequest request = UpdateTableRequest.builder()
            .tableName(tableName)
```

```
            .globalSecondaryIndexUpdates(globalSecondaryIndexUpdate)
            .warmThroughput(tableWarmThroughput)
            .build();

        try {
            ddb.updateTable(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }

        System.out.println(SUCCESS_MESSAGE);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
import { DynamoDBClient, UpdateTableCommand } from "@aws-sdk/client-dynamodb";

export async function updateDynamoDBTableWarmThroughput(
  tableName,
  tableReadUnits,
  tableWriteUnits,
  gsiName,
  gsiReadUnits,
  gsiWriteUnits,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });

    // Construct the update table request
    const updateTableRequest = {
      TableName: tableName,
```

```
      GlobalSecondaryIndexUpdates: [
        {
            Update: {
                IndexName: gsiName,
                WarmThroughput: {
                    ReadUnitsPerSecond: gsiReadUnits,
                    WriteUnitsPerSecond: gsiWriteUnits,
                },
            },
        },
      ],
      WarmThroughput: {
          ReadUnitsPerSecond: tableReadUnits,
          WriteUnitsPerSecond: tableWriteUnits,
      },
    };

    const command = new UpdateTableCommand(updateTableRequest);
    const response = await ddbClient.send(command);
    console.log(`Table updated successfully! Response:
 ${JSON.stringify(response)}`);
    return response;
  } catch (error) {
    console.error(`Error updating table: ${error}`);
    throw error;
  }
}

// Example usage (commented out for testing)
/*
updateDynamoDBTableWarmThroughput(
  'example-table',
  5, 5,
  'example-index',
  2, 2
);
*/
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateTable」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
from boto3 import client
from botocore.exceptions import ClientError


def update_dynamodb_table_warm_throughput(
    table_name,
    table_read_units,
    table_write_units,
    gsi_name,
    gsi_read_units,
    gsi_write_units,
    region_name="us-east-1",
):
    """
    Updates the warm throughput of a DynamoDB table and a global secondary index.

    :param table_name: The name of the table to update.
    :param table_read_units: The new read units per second for the table's warm
 throughput.
    :param table_write_units: The new write units per second for the table's warm
 throughput.
    :param gsi_name: The name of the global secondary index to update.
    :param gsi_read_units: The new read units per second for the GSI's warm
 throughput.
    :param gsi_write_units: The new write units per second for the GSI's warm
 throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    :return: The response from the update_table operation
    """
    try:
        ddb = client("dynamodb", region_name=region_name)

        # Update the table's warm throughput
        table_warm_throughput = {
            "ReadUnitsPerSecond": table_read_units,
            "WriteUnitsPerSecond": table_write_units,
        }
```

```
        # Update the global secondary index's warm throughput
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_read_units,
            "WriteUnitsPerSecond": gsi_write_units,
        }

        # Construct the global secondary index update
        global_secondary_index_update = [
            {"Update": {"IndexName": gsi_name, "WarmThroughput":
gsi_warm_throughput}}
        ]

        # Construct the update table request
        update_table_request = {
            "TableName": table_name,
            "GlobalSecondaryIndexUpdates": global_secondary_index_update,
            "WarmThroughput": table_warm_throughput,
        }

        # Update the table
        response = ddb.update_table(**update_table_request)
        print("Table updated successfully!")
        return response  # Make sure to return the response
    except ClientError as e:
        print(f"Error updating table: {e}")
        raise e
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の「UpdateTable」を参照してください。

ウォームスループットが高い新しい DynamoDB テーブルを作成する

DynamoDB テーブルを作成するときに、以下の手順に従ってウォームスループット値を調整できます。これらのステップは、グローバルテーブルまたはセカンダリインデックスを作成する場合にも適用されます。

AWS マネジメントコンソール

コンソールを使用して DynamoDB テーブルを作成し、ウォームスループット値を調整するには:

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. [テーブルの作成] を選択します。
3. [テーブル名]、[パーティションキー]、[ソートキー] (オプション) を選択します。
4. [テーブル設定] セクションで [設定をカスタマイズ] を選択します。
5. [ウォームスループット] フィールドで、[ウォームスループットを増やす] を選択します。
6. [1 秒あたりの読み取りユニット] と [1 秒あたりの書き込みユニット] を調整します。これら 2 つの設定は、テーブルが即座に処理できる最大スループットを定義します。
7. 残りのテーブルの詳細を追加し、[テーブルの作成] を選択します。

AWS CLI

次の AWS CLI の例は、カスタマイズされたウォームスループット値を使用して DynamoDB テーブルを作成する方法を示しています。

1. `create-table` オペレーションを実行して、次の DynamoDB テーブルを作成します。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
                            AttributeName=GameTitle,AttributeType=S \
                            AttributeName=TopScore,AttributeType=N  \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH
\"},
                                {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
                \"Projection\":{
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 50,
                    \"WriteCapacityUnits\": 25
```

```
                },\"WarmThroughput\": {
                    \"ReadUnitsPerSecond\": 1987,
                    \"WriteUnitsPerSecond\": 543
                }
            }
        ]" \
    --warm-throughput ReadUnitsPerSecond=12345,WriteUnitsPerSecond=4567 \
    --region us-east-1
```

2. 次のようなレスポンスが表示されます。WarmThroughput 設定は、ReadUnitsPerSecond および WriteUnitsPerSecond として表示されます。ウォームスループット値が更新されている場合、Status は UPDATING になり、新しいウォームスループット値が設定されている場合は、ACTIVE になります。

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": 1730241788.779,
```

```
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 20,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/GameScores",
        "TableId": "XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 50,
                    "WriteCapacityUnits": 25
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-east-1:XXXXXXXXXXXX:table/
GameScores/index/GameTitleIndex",
                "WarmThroughput": {
                    "ReadUnitsPerSecond": 1987,
                    "WriteUnitsPerSecond": 543,
                    "Status": "UPDATING"
                }
            }
```

```
        ],
        "DeletionProtectionEnabled": false,
        "WarmThroughput": {
            "ReadUnitsPerSecond": 12345,
            "WriteUnitsPerSecond": 4567,
            "Status": "UPDATING"
        }
    }
}
```

AWS SDK

次の SDK の例は、カスタマイズされたウォームスループット値を使用して DynamoDB テーブルを作成する方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
    }
```

```
    /**
     * Builds a ProvisionedThroughput object with the specified read and write
capacity units.
     *
     * @param readCapacityUnits The read capacity units
     * @param writeCapacityUnits The write capacity units
     * @return A configured ProvisionedThroughput object
     */
    public static ProvisionedThroughput buildProvisionedThroughput(
        final Long readCapacityUnits, final Long writeCapacityUnits) {
        return ProvisionedThroughput.builder()
            .readCapacityUnits(readCapacityUnits)
            .writeCapacityUnits(writeCapacityUnits)
            .build();
    }

    /**
     * Builds an AttributeDefinition with the specified name and type.
     *
     * @param attributeName The attribute name
     * @param scalarAttributeType The attribute type
     * @return A configured AttributeDefinition
     */
    private static AttributeDefinition buildAttributeDefinition(
        final String attributeName, final ScalarAttributeType
scalarAttributeType) {
        return AttributeDefinition.builder()
            .attributeName(attributeName)
            .attributeType(scalarAttributeType)
            .build();
    }

    /**
     * Builds a KeySchemaElement with the specified name and key type.
     *
     * @param attributeName The attribute name
     * @param keyType The key type (HASH or RANGE)
     * @return A configured KeySchemaElement
     */
    private static KeySchemaElement buildKeySchemaElement(final String
attributeName, final KeyType keyType) {
        return KeySchemaElement.builder()
            .attributeName(attributeName)
            .keyType(keyType)
```

```
            .build();
    }

    /**
     * Creates a DynamoDB table with the specified configuration including warm
throughput settings.
     *
     * @param ddb The DynamoDB client
     * @param tableName The name of the table to create
     * @param partitionKey The partition key attribute name
     * @param sortKey The sort key attribute name
     * @param miscellaneousKeyAttribute Additional key attribute name for GSI
     * @param nonKeyAttribute Non-key attribute to include in GSI projection
     * @param tableReadCapacityUnits Read capacity units for the table
     * @param tableWriteCapacityUnits Write capacity units for the table
     * @param tableWarmReadUnitsPerSecond Warm read units per second for the
table
     * @param tableWarmWriteUnitsPerSecond Warm write units per second for the
table
     * @param globalSecondaryIndexName The name of the GSI to create
     * @param globalSecondaryIndexReadCapacityUnits Read capacity units for the
GSI
     * @param globalSecondaryIndexWriteCapacityUnits Write capacity units for the
GSI
     * @param globalSecondaryIndexWarmReadUnitsPerSecond Warm read units per
second for the GSI
     * @param globalSecondaryIndexWarmWriteUnitsPerSecond Warm write units per
second for the GSI
     */
    public static void createDynamoDBTable(
        final DynamoDbClient ddb,
        final String tableName,
        final String partitionKey,
        final String sortKey,
        final String miscellaneousKeyAttribute,
        final String nonKeyAttribute,
        final Long tableReadCapacityUnits,
        final Long tableWriteCapacityUnits,
        final Long tableWarmReadUnitsPerSecond,
        final Long tableWarmWriteUnitsPerSecond,
        final String globalSecondaryIndexName,
        final Long globalSecondaryIndexReadCapacityUnits,
        final Long globalSecondaryIndexWriteCapacityUnits,
        final Long globalSecondaryIndexWarmReadUnitsPerSecond,
```

```
        final Long globalSecondaryIndexWarmWriteUnitsPerSecond) {

        // Define the table attributes
        final AttributeDefinition partitionKeyAttribute =
buildAttributeDefinition(partitionKey, ScalarAttributeType.S);
        final AttributeDefinition sortKeyAttribute =
buildAttributeDefinition(sortKey, ScalarAttributeType.S);
        final AttributeDefinition miscellaneousKeyAttributeDefinition =
            buildAttributeDefinition(miscellaneousKeyAttribute,
ScalarAttributeType.N);
        final AttributeDefinition[] attributeDefinitions = {
            partitionKeyAttribute, sortKeyAttribute,
miscellaneousKeyAttributeDefinition
        };

        // Define the table key schema
        final KeySchemaElement partitionKeyElement =
buildKeySchemaElement(partitionKey, KeyType.HASH);
        final KeySchemaElement sortKeyElement = buildKeySchemaElement(sortKey,
KeyType.RANGE);
        final KeySchemaElement[] keySchema = {partitionKeyElement,
sortKeyElement};

        // Define the provisioned throughput for the table
        final ProvisionedThroughput provisionedThroughput =
            buildProvisionedThroughput(tableReadCapacityUnits,
tableWriteCapacityUnits);

        // Define the Global Secondary Index (GSI)
        final KeySchemaElement globalSecondaryIndexPartitionKeyElement =
buildKeySchemaElement(sortKey, KeyType.HASH);
        final KeySchemaElement globalSecondaryIndexSortKeyElement =
            buildKeySchemaElement(miscellaneousKeyAttribute, KeyType.RANGE);
        final KeySchemaElement[] gsiKeySchema = {
            globalSecondaryIndexPartitionKeyElement,
globalSecondaryIndexSortKeyElement
        };

        final Projection gsiProjection = Projection.builder()
            .projectionType(PROJECTION_TYPE_INCLUDE)
            .nonKeyAttributes(nonKeyAttribute)
            .build();

        final ProvisionedThroughput gsiProvisionedThroughput =
```

```
            buildProvisionedThroughput(globalSecondaryIndexReadCapacityUnits,
globalSecondaryIndexWriteCapacityUnits);

        // Define the warm throughput for the Global Secondary Index (GSI)
        final WarmThroughput gsiWarmThroughput = buildWarmThroughput(
            globalSecondaryIndexWarmReadUnitsPerSecond,
globalSecondaryIndexWarmWriteUnitsPerSecond);

        final GlobalSecondaryIndex globalSecondaryIndex =
GlobalSecondaryIndex.builder()
            .indexName(globalSecondaryIndexName)
            .keySchema(gsiKeySchema)
            .projection(gsiProjection)
            .provisionedThroughput(gsiProvisionedThroughput)
            .warmThroughput(gsiWarmThroughput)
            .build();

        // Define the warm throughput for the table
        final WarmThroughput tableWarmThroughput =
            buildWarmThroughput(tableWarmReadUnitsPerSecond,
tableWarmWriteUnitsPerSecond);

        final CreateTableRequest request = CreateTableRequest.builder()
            .tableName(tableName)
            .attributeDefinitions(attributeDefinitions)
            .keySchema(keySchema)
            .provisionedThroughput(provisionedThroughput)
            .globalSecondaryIndexes(globalSecondaryIndex)
            .warmThroughput(tableWarmThroughput)
            .build();

        final CreateTableResponse response = ddb.createTable(request);
        System.out.println(response);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「CreateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

export async function createDynamoDBTableWithWarmThroughput(
  tableName,
  partitionKey,
  sortKey,
  miscKeyAttr,
  nonKeyAttr,
  tableProvisionedReadUnits,
  tableProvisionedWriteUnits,
  tableWarmReads,
  tableWarmWrites,
  indexName,
  indexProvisionedReadUnits,
  indexProvisionedWriteUnits,
  indexWarmReads,
  indexWarmWrites,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });
    const command = new CreateTableCommand({
      TableName: tableName,
      AttributeDefinitions: [
          { AttributeName: partitionKey, AttributeType: "S" },
          { AttributeName: sortKey, AttributeType: "S" },
          { AttributeName: miscKeyAttr, AttributeType: "N" },
      ],
      KeySchema: [
          { AttributeName: partitionKey, KeyType: "HASH" },
          { AttributeName: sortKey, KeyType: "RANGE" },
      ],
      ProvisionedThroughput: {
          ReadCapacityUnits: tableProvisionedReadUnits,
          WriteCapacityUnits: tableProvisionedWriteUnits,
      },
      WarmThroughput: {
```

```
            ReadUnitsPerSecond: tableWarmReads,
            WriteUnitsPerSecond: tableWarmWrites,
        },
        GlobalSecondaryIndexes: [
            {
              IndexName: indexName,
              KeySchema: [
                  { AttributeName: sortKey, KeyType: "HASH" },
                  { AttributeName: miscKeyAttr, KeyType: "RANGE" },
              ],
              Projection: {
                  ProjectionType: "INCLUDE",
                  NonKeyAttributes: [nonKeyAttr],
              },
              ProvisionedThroughput: {
                  ReadCapacityUnits: indexProvisionedReadUnits,
                  WriteCapacityUnits: indexProvisionedWriteUnits,
              },
              WarmThroughput: {
                  ReadUnitsPerSecond: indexWarmReads,
                  WriteUnitsPerSecond: indexWarmWrites,
              },
            },
        ],
      });
      const response = await ddbClient.send(command);
      console.log(response);
      return response;
    } catch (error) {
      console.error(`Error creating table: ${error}`);
      throw error;
    }
}

// Example usage (commented out for testing)
/*
createDynamoDBTableWithWarmThroughput(
  'example-table',
  'pk',
  'sk',
  'gsiKey',
  'data',
  10, 10, 5, 5,
  'example-index',
```

```
  5, 5, 2, 2
);
*/
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「CreateTable」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
from boto3 import client
from botocore.exceptions import ClientError


def create_dynamodb_table_warm_throughput(
    table_name,
    partition_key,
    sort_key,
    misc_key_attr,
    non_key_attr,
    table_provisioned_read_units,
    table_provisioned_write_units,
    table_warm_reads,
    table_warm_writes,
    gsi_name,
    gsi_provisioned_read_units,
    gsi_provisioned_write_units,
    gsi_warm_reads,
    gsi_warm_writes,
    region_name="us-east-1",
):
    """
    Creates a DynamoDB table with a warm throughput setting configured.

    :param table_name: The name of the table to be created.
    :param partition_key: The partition key for the table being created.
```

```
    :param sort_key: The sort key for the table being created.
    :param misc_key_attr: A miscellaneous key attribute for the table being
created.
    :param non_key_attr: A non-key attribute for the table being created.
    :param table_provisioned_read_units: The newly created table's provisioned
read capacity units.
    :param table_provisioned_write_units: The newly created table's provisioned
write capacity units.
    :param table_warm_reads: The read units per second setting for the table's
warm throughput.
    :param table_warm_writes: The write units per second setting for the table's
warm throughput.
    :param gsi_name: The name of the Global Secondary Index (GSI) to be created
on the table.
    :param gsi_provisioned_read_units: The configured Global Secondary Index
(GSI) provisioned read capacity units.
    :param gsi_provisioned_write_units: The configured Global Secondary Index
(GSI) provisioned write capacity units.
    :param gsi_warm_reads: The read units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
    :param gsi_warm_writes: The write units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    """
    try:
        ddb = client("dynamodb", region_name=region_name)

        # Define the table attributes
        attribute_definitions = [
            {"AttributeName": partition_key, "AttributeType": "S"},
            {"AttributeName": sort_key, "AttributeType": "S"},
            {"AttributeName": misc_key_attr, "AttributeType": "N"},
        ]

        # Define the table key schema
        key_schema = [
            {"AttributeName": partition_key, "KeyType": "HASH"},
            {"AttributeName": sort_key, "KeyType": "RANGE"},
        ]

        # Define the provisioned throughput for the table
        provisioned_throughput = {
            "ReadCapacityUnits": table_provisioned_read_units,
            "WriteCapacityUnits": table_provisioned_write_units,
```

```
        }

        # Define the global secondary index
        gsi_key_schema = [
            {"AttributeName": sort_key, "KeyType": "HASH"},
            {"AttributeName": misc_key_attr, "KeyType": "RANGE"},
        ]
        gsi_projection = {"ProjectionType": "INCLUDE", "NonKeyAttributes":
[non_key_attr]}
        gsi_provisioned_throughput = {
            "ReadCapacityUnits": gsi_provisioned_read_units,
            "WriteCapacityUnits": gsi_provisioned_write_units,
        }
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_warm_reads,
            "WriteUnitsPerSecond": gsi_warm_writes,
        }
        global_secondary_indexes = [
            {
                "IndexName": gsi_name,
                "KeySchema": gsi_key_schema,
                "Projection": gsi_projection,
                "ProvisionedThroughput": gsi_provisioned_throughput,
                "WarmThroughput": gsi_warm_throughput,
            }
        ]

        # Define the warm throughput for the table
        warm_throughput = {
            "ReadUnitsPerSecond": table_warm_reads,
            "WriteUnitsPerSecond": table_warm_writes,
        }

        # Create the DynamoDB client and create the table
        response = ddb.create_table(
            TableName=table_name,
            AttributeDefinitions=attribute_definitions,
            KeySchema=key_schema,
            ProvisionedThroughput=provisioned_throughput,
            GlobalSecondaryIndexes=global_secondary_indexes,
            WarmThroughput=warm_throughput,
        )

        print(response)
```

```
        return response
    except ClientError as e:
        print(f"Error creating table: {e}")
        raise e
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「CreateTable」を参照してください。

さまざまなシナリオでの DynamoDB ウォームスループットについて

DynamoDB ウォームスループットを使用する際に直面する可能性のあるさまざまなシナリオを次に示します。

トピック

- ウォームスループットと不均等なアクセスパターン
- プロビジョニングされたテーブルのウォームスループット
- オンデマンドテーブルのウォームスループット
- 最大スループットが設定されたオンデマンドテーブルのウォームスループット

ウォームスループットと不均等なアクセスパターン

テーブルのウォームスループットは 1 秒あたり 30,000 読み取りユニット、1 秒あたり 10,000 書き込みユニットですが、これらの値に達する前に読み取りまたは書き込みでスロットリングが発生する可能性があります。これは、ホットパーティションが原因である可能性があります。DynamoDB は実質的に無制限のスループットをサポートするようにスケーリングを続けることができますが、個々のパーティションは 1 秒あたり 1,000 書き込みユニットと 1 秒あたり 3,000 読み取りユニットに制限されています。アプリケーションがテーブルのパーティションのごく一部にトラフィックを大量に誘導する場合、テーブルのウォームスループット値に達する前でもスロットリングが発生する可能性があります。シームレスなスケーラビリティを確保し、ホットパーティションを回避するために、「DynamoDB のベストプラクティス」に従うことをお勧めします。

プロビジョニングされたテーブルのウォームスループット

ウォームスループットが 1 秒あたり 30,000 読み込みユニット、1 秒あたり 10,000 書き込みユニットであるものの、現在のプロビジョンドスループットが 4,000 RCU、8,000 WCU であるプロビジョニング済みテーブルを考えてみましょう。プロビジョンドスループット設定を更新することで、テー

ブルのプロビジョンドスループットを 30,000 RCU または 10,000 WCU まで即座にスケールできます。プロビジョンドスループットをこれらの値を超えて増やすと、新しいピークスループットが確立されるため、ウォームスループットは自動的に新たなより高い値に調整されます。例えば、プロビジョンドスループットを 50,000 RCU に設定すると、ウォームスループットは 1 秒あたり 50,000 読み込みユニットに増加します。

```
"ProvisionedThroughput":
    {
        "ReadCapacityUnits": 4000,
        "WriteCapacityUnits": 8000
    }
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 30000,
        "WriteUnitsPerSecond": 10000
    }
```

オンデマンドテーブルのウォームスループット

新しいオンデマンドテーブルは、1 秒あたり 12,000 読み取りユニット、1 秒あたり 4,000 書き込みユニットのウォームスループットから始まります。テーブルは、これらのレベルまで持続するトラフィックに即座に対応できます。リクエストが 1 秒あたり 12,000 読み込みユニットまたは 1 秒あたり 4,000 書き込みユニットを超えると、ウォームスループットは自動的により高い値に調整されます。

```
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 12000,
        "WriteUnitsPerSecond": 4000
    }
```

最大スループットが設定されたオンデマンドテーブルのウォームスループット

ウォームスループットが 1 秒あたり 30,000 読み込みユニットであるものの、最大スループットが 5,000 読み込みリクエストユニット (RRU) に設定されているオンデマンドテーブルを考えてみましょう。このシナリオでは、テーブルのスループットは、設定した最大 5,000 RRU に制限されます。この最大値を超えるスループットリクエストはスロットリングされます。ただし、テーブル別の最大スループットは、アプリケーションのニーズに基づいていつでも変更できます。

```
"OnDemandThroughput":
```

```
    {
        "MaxReadRequestUnits": 5000,
        "MaxWriteRequestUnits": 4000
    }
"WarmThroughput":
    {
        "ReadUnitsPerSecond": 30000,
        "WriteUnitsPerSecond": 10000
    }
```

DynamoDB バーストキャパシティとアダプティブキャパシティ

スループット例外に伴うスロットリングを最小限に抑えるために、DynamoDB はバーストキャパシティを使用して使用量のスパイクを処理します。DynamoDB は、アダプティブキャパシティを使用して、アクセスパターンの不均衡に対応できるようにします。

バーストキャパシティ

DynamoDB は、バーストキャパシティにより、スループットプロビジョニングに柔軟性をもたらします。利用可能なスループットを使い切っていない場合、DynamoDB は、キャパシティの未使用分を、後のスループットのバーストに備えて留保しておき、使用量のスパイクに対応します。バーストキャパシティーにより、スロットリングされていた可能性のある読み込みまたは書き込みリクエストが成功します。

現在、DynamoDB は、未使用の読み込みおよび書き込みキャパシティを最大 5 分 (300 秒) 保持します。読み込みまたは書き込みアクティビティのバーストが発生した場合、これらの余分なキャパシティユニットをすばやく消費できます。これは、テーブルに定義した 1 秒あたりのプロビジョンドスループットキャパシティよりも高速です。

また、DynamoDB はバックグラウンドメンテナンスやその他のタスクのために予告なしにバーストキャパシティを消費する場合があります。

これらのバーストキャパシティの詳細は将来変更される可能性があります。

アダプティブキャパシティ

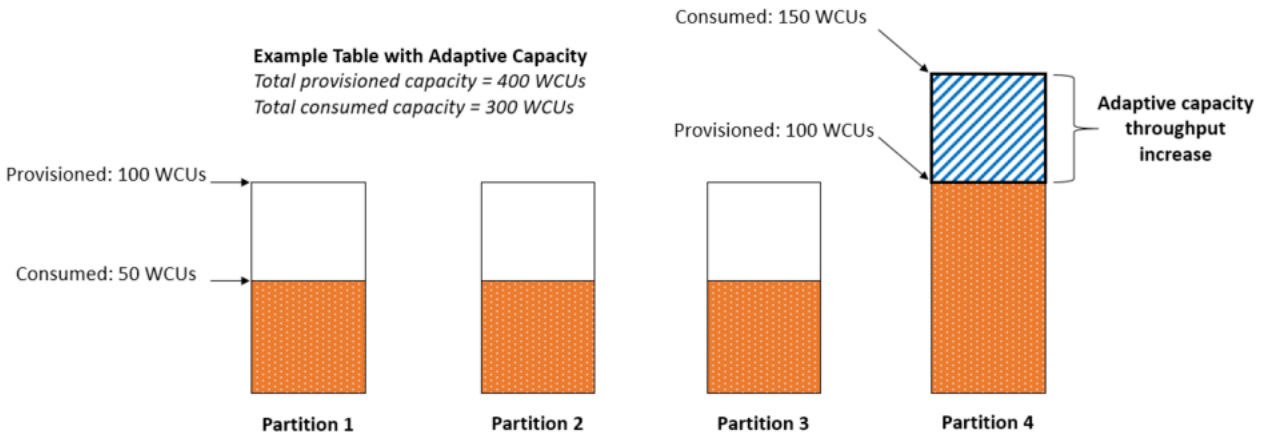
DynamoDB では、パーティション間にデータを自動的に分散します。これらは、AWS クラウドの複数のサーバーに保存されます。読み込みおよび書き込みアクティビティは常に均等に分散できるとは限りません。データアクセスが不均等の場合は、他のパーティションと比較して、「ホット」パー

ティションに大容量の読み書きトラフィックが送信されることがあります。パーティションの読み込みおよび書き込みオペレーションは個別に管理されるため、単一のパーティションが 3,000 を超える読み込みオペレーションまたは 1,000 を超える書き込みオペレーションを受け取ると、スロットリングが発生します。アダプティブキャパシティーでは、さらに多くのトラフィックを受け取るパーティションのスループットキャパシティーを自動的に増加させます。

不均等なアクセスパターンに対応できるように、DynamoDB アダプティブキャパシティを使用することで、スロットリングすることなく、アプリケーションでホットパーティションに対して読み書きを継続できます。ただし、トラフィックは、プロビジョニングされた合計容量またはパーティションの最大容量を超えないものとします。アダプティブキャパシティでは、さらに多くのトラフィックを受け取るパーティションのスループットキャパシティを自動的かつ瞬時に増加させます。

次の図は、アダプティブキャパシティの機能を示しています。このテーブルの例のプロビジョニングでは、4 つのパーティション間で 400 WCU が均等に共有されることで、各パーティションで 1 秒間に最大 100 WCU を維持できます。パーティション 1、2、3 はそれぞれ、50 WCU/秒の書き込みトラフィックを受け取ります。パーティション 4 は 150 WCU/秒の書き込みトラフィックを受け取ります。このホットパーティションは、未使用のバーストキャパシティがある場合でも書き込みトラフィックを受け入れることができます。しかし、最終的には 1 秒間に 100 WCU を超えるトラフィックのスロットリングとなります。

DynamoDB アダプティブキャパシティは、パーティション 4 のキャパシティを増大して応答するため、このパーティションはスロットリングされることなく、150 WCU/秒のより高いワークロードを維持できます。



アダプティブキャパシティはすべての DynamoDB テーブルで自動的に有効化されます。追加料金はありません。アダプティブキャパシティを明示的に有効化または無効化する必要はありません。

アクセス頻度の高い項目を分離する

お使いのアプリケーションが 1 つ以上の項目に偏った高いトラフィックを送る場合、アクセス頻度の高い項目が同一パーティション内に存在しないように、アダプティブキャパシティによってパーティションのバランスが再調整されます。アクセス頻度の高い項目を分離することで、ワークロードが単一のパーティションのスループットクォータを超えることから生じるリクエストのスロットリングの可能性が低くなります。ソートキーの単調な増減によって追跡されるトラフィックが項目のコレクションでない限り、項目のコレクションをソートキー別にセグメントに分割することもできます。

アプリケーションから単一の項目に大量のトラフィックが一貫して送信される場合、アダプティブキャパシティはデータを再調整して、そのアクセス頻度の高い単一の項目のみをパーティションに含めることがあります。この場合、DynamoDB では、単一項目のプライマリキーに対して、最大 3,000 RCU および 1,000 WCU のパーティションのスループットを提供できます。アダプティブキャパシティでは、テーブルにローカルセカンダリインデックスがある場合、テーブルの複数のパーティション間で項目コレクションを分割しません。

DynamoDB でキャパシティモードを切り替える際の考慮事項

DynamoDB テーブルを作成する場合、オンデマンドまたはプロビジョンドキャパシティモードのいずれかを選択する必要があります。

24 時間のローリングウィンドウで、テーブルをプロビジョンドキャパシティモードからオンデマンドモードに最大 4 回切り替えることができます。テーブルは、オンデマンドモードからプロビジョンドキャパシティモードにいつでも切り替えることができます。

トピック

- プロビジョンドキャパシティモードからオンデマンドキャパシティモードへの切り替え
- オンデマンドキャパシティモードからプロビジョンドキャパシティモードへの切り替え

プロビジョンドキャパシティモードからオンデマンドキャパシティモードへの切り替え

プロビジョンドモードでは、予想されるアプリケーションのニーズに基づいて読み込みおよび書き込みキャパシティを設定します。テーブルをプロビジョニングモードからオンデマンドモードに更新するときは、アプリケーションで実行することが予測される読み込みおよび書き込みスループットを指定する必要はありません。

DynamoDB オンデマンドは、読み込みおよび書き込みリクエストの料金が従量制であるため、使用した分だけを支払います。これにより、コストとパフォーマンスのバランスを簡単に取ることができます。

オプションで、個々のオンデマンドテーブルおよび関連するグローバルセカンダリインデックスの読み込みや書き込み (または両方) の最大スループットを設定して、コストと使用量を制限し続けることができます。特定のテーブルやインデックスの最大スループットの設定の詳細については、「オンデマンドテーブルの DynamoDB 最大スループット」を参照してください。

プロビジョンドキャパシティーモードからオンデマンドキャパシティーモードに切り替えるプロセスには、数分かかることがあります。切り替え期間中、テーブルは以前にプロビジョニングされた書き込みキャパシティーユニットおよび読み込みキャパシティーユニットの両方と整合性のあるスループットを提供します。

オンデマンドキャパシティモードの初期スループット

最近、既存のテーブルをオンデマンドキャパシティモードに初めて切り替えた場合、テーブルがオンデマンドキャパシティモードを使用してトラフィックを以前に処理していなくても、テーブルは次に示す以前のピーク設定を持ちます。

考えられるシナリオの例を以下に示します。

- 4,000 の WCU と 12,000 の RCU 未満に設定されたプロビジョンドテーブルが、これまでにそれを超える値にプロビジョニングされたことがない場合。このテーブルを初めてオンデマンドに切り替えると、DynamoDB は、少なくとも 4,000 の書き込みユニット/秒、12,000 の読み込みユニット/秒を即座に維持できるようにスケールアウトします。
- 8,000 の WCU と 24,000 の RCU に設定されたプロビジョンドテーブルの場合。このテーブルをオンデマンドに切り替えると、少なくとも 8,000 の書き込みユニット/秒、24,000 の読み込みユニット/秒を引き続き維持できます。
- 8,000 WCU と 24,000 RCU で設定されたプロビジョンドテーブルが、一定期間にわたり 6,000 の書き込みユニット/秒、18,000 の読み込みユニット/秒を消費した場合。このテーブルをオンデマンドに切り替えると、少なくとも 8,000 の書き込みユニット/秒、24,000 の読み込みユニット/秒を引き続き維持できます。以前のトラフィックにより、このテーブルはスロットリングなしではるかに高いレベルのトラフィックを維持できる可能性があります。
- 以前は 10,000 の WCU と 10,000 の RCU でプロビジョニングされていたテーブルが、現在は 10 の RCU と 10 の WCU でプロビジョニングされている場合。このテーブルをオンデマンドに切り替えると、少なくとも 10,000 の書き込みユニット/秒、10,000 の読み込みユニット/秒を維持できます。

ウォームスループットとは、テーブルが瞬時にサポートできる読み取りおよび書き込みオペレーションの数のことです。ウォームスループットは、すべてのテーブルとグローバルセカンダリインデックス (GSI) でデフォルトで使用でき、過去の使用状況に基づいて既にスケールされているレベルを表します。

オンデマンドモードを使用している場合、またはプロビジョンドスループットをこれらの値に更新した場合、アプリケーションはこれらの値まで即座にリクエストを発行できます。詳細については、「DynamoDB ウォームスループットについて」を参照してください。

自動スケーリング設定

テーブルをプロビジョニングモードからオンデマンドモードに更新すると、以下のようになります。

- コンソールを使用している場合、Auto Scaling 設定 (ある場合) がすべて削除されます。
- AWS CLI または AWS SDK を使用している場合、Auto Scaling 設定はすべて保持されます。これらの設定は、テーブルをもう一度プロビジョニングされた請求モードに更新すると適用できます。

DynamoDB コンソールでのキャパシティモードの一括編集

DynamoDB コンソールを使用して、複数のテーブルを一括編集して、プロビジョンドキャパシティモードからオンデマンドキャパシティモードに切り替えることができます。キャパシティモードを一括編集するには。

1. DynamoDB コンソールで [テーブル] ページに移動します。
2. オンデマンドキャパシティモードに更新するテーブルのチェックボックスをオンにします。
3. [アクション] を選択し、[オンデマンドキャパシティモードへの更新] を選択します。

この一括オペレーションにより、各テーブルを個別に更新することなく、複数のテーブルをオンデマンドキャパシティモードに効率的に切り替えることができます。

オンデマンドキャパシティモードからプロビジョンドキャパシティモードへの切り替え

オンデマンドキャパシティーモードからプロビジョンドキャパシティーモードに戻すと、テーブルは、テーブルがオンデマンドキャパシティーモードに設定されたときに到達した前のピークと整合性のあるスループットを提供します。

容量の管理

テーブルをオンデマンドモードからプロビジョニングモードに更新するときは、以下の点を考慮に入れてください。

- AWS CLI または AWS SDK を使用している場合、Amazon CloudWatch を使用して消費履歴 (`ConsumedWriteCapacityUnits` および `ConsumedReadCapacityUnits` メトリクス) を参照し、新しいスループット設定を決定することで、テーブルおよびグローバルセカンダリインデックスのプロビジョンされた適切な容量設定を選択します。

  > **Note**
  >
  > グローバルテーブルからプロビジョニングモードに切り替える場合、すべてのリージョンレプリカにわたって基本テーブルおよびグローバルセカンダリインデックスの最大消費量を調べた上で、新しいスループット設定を決定します。

- オンデマンドモードからプロビジョンドモードに戻す場合は、移行中にテーブルまたはインデックスのキャパシティを処理できるように、最初のプロビジョンドユニットを十分高い値に設定してください。

Auto Scaling の管理

テーブルをオンデマンドモードからプロビジョニングモードに更新すると、以下のようになります。

- コンソールを使用する場合は、以下のデフォルトで自動スケーリングを有効にすることをお勧めします。
  - 目標使用率: 70%
  - プロビジョニングされた最小キャパシティー: ユニット 5 個
  - プロビジョニングされた最大キャパシティー: リージョンの最大値
- AWS CLI または SDK を使用している場合、前の Auto Scaling 設定 (ある場合) が保持されます。

DynamoDB と AWS SDK を使用したプログラミング

このセクションでは、デベロッパーに関連するトピックを説明します。代わりにコードのサンプルを実行する場合は、「このデベロッパーガイドのコード例の実行」を参照してください。

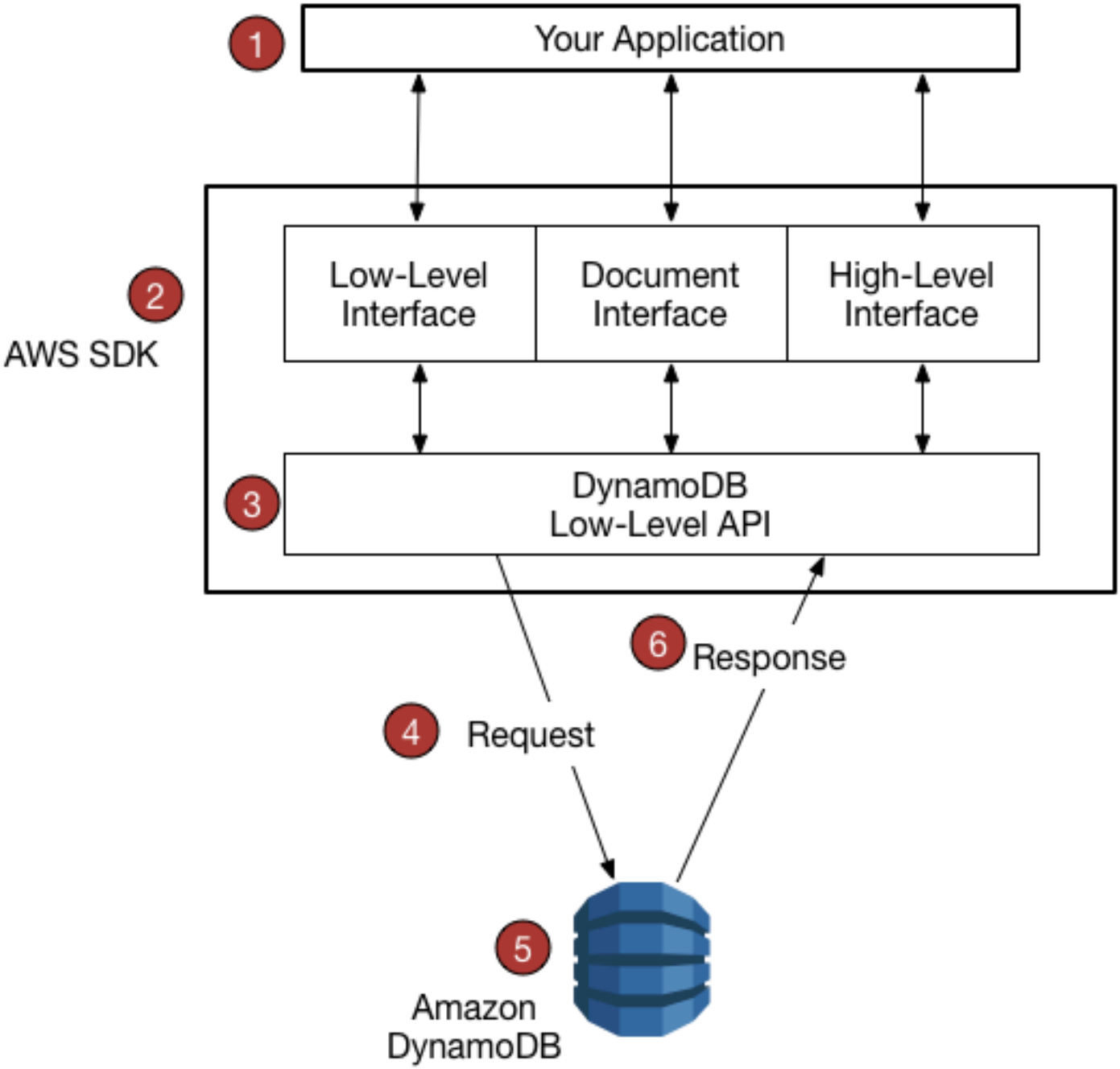
> **Note**
>
> 2017 年 12 月、AWS はすべての Amazon DynamoDB エンドポイントを移行し、Amazon Trust Services (ATS) が発行する、安全な証明書を使用するプロセスを開始しました。詳細については、「DynamoDB で SSL/TLS 接続の確立に関する問題をトラブルシューティングする」を参照してください。

トピック

- DynamoDB に対する AWS SDK サポートの概要
- Python と Boto3 による Amazon DynamoDB のプログラミング
- JavaScript による Amazon DynamoDB のプログラミング
- AWS SDK for Java 2.x を使用した DynamoDB のプログラミング
- DynamoDB でのエラー処理
- AWS SDK で DynamoDB を使用する

DynamoDB に対する AWS SDK サポートの概要

次の図表は、AWS SDK を使用した Amazon DynamoDB アプリケーションの、プログラミングの概要を示しています。



1. 自分が使用するプログラミング言語用のための AWS SDK を使用して、アプリケーションを作成します。
2. 各 AWS SDK には、DynamoDB を操作するプログラミングインターフェイスが 1 つ以上用意されています。使用できるインターフェイスは、使用するプログラミング言語と AWS SDK によって異なります。オプションには以下が含まれます。
   - DynamoDB で動作する低レベルインターフェイス
   - DynamoDB で動作するドキュメントインターフェイス

- DynamoDB で動作するオブジェクト永続性インターフェイス
- 高レベルのインターフェイス

3. AWS SDK は、低レベル DynamoDB API で使用する HTTP (S) リクエストを作成します。
4. AWS SDK は DynamoDB エンドポイントにリクエストを送信します。
5. DynamoDB がリクエストを実行します。リクエストが成功すると、DynamoDB は HTTP 200 レスポンスコード (OK) を返します。リクエストが失敗した場合は、DynamoDB は HTTP エラーコードとエラーメッセージを返します。
6. AWS SDK はレスポンスを処理し、アプリケーションに伝達します。

各 AWS SDK は、次のような重要なサービスをアプリケーションに提供します。

- HTTP (S) リクエストのフォーマットと、リクエストパラメータのシリアル化。
- リクエストごとの暗号化署名の生成。
- DynamoDB エンドポイントへのリクエストの転送と、DynamoDB からのレスポンスの受信。
- これらの応答からの結果の抽出。
- エラーが発生した場合の基本的な再試行ロジックの実装。

これらのタスクでは、コードを記述する必要はありません。

> **Note**
>
> AWS SDK のインストール方法とドキュメントの詳細については、「アマゾン ウェブ サービス用ツール」を参照してください。

AWS アカウントベースのエンドポイントの SDK サポート

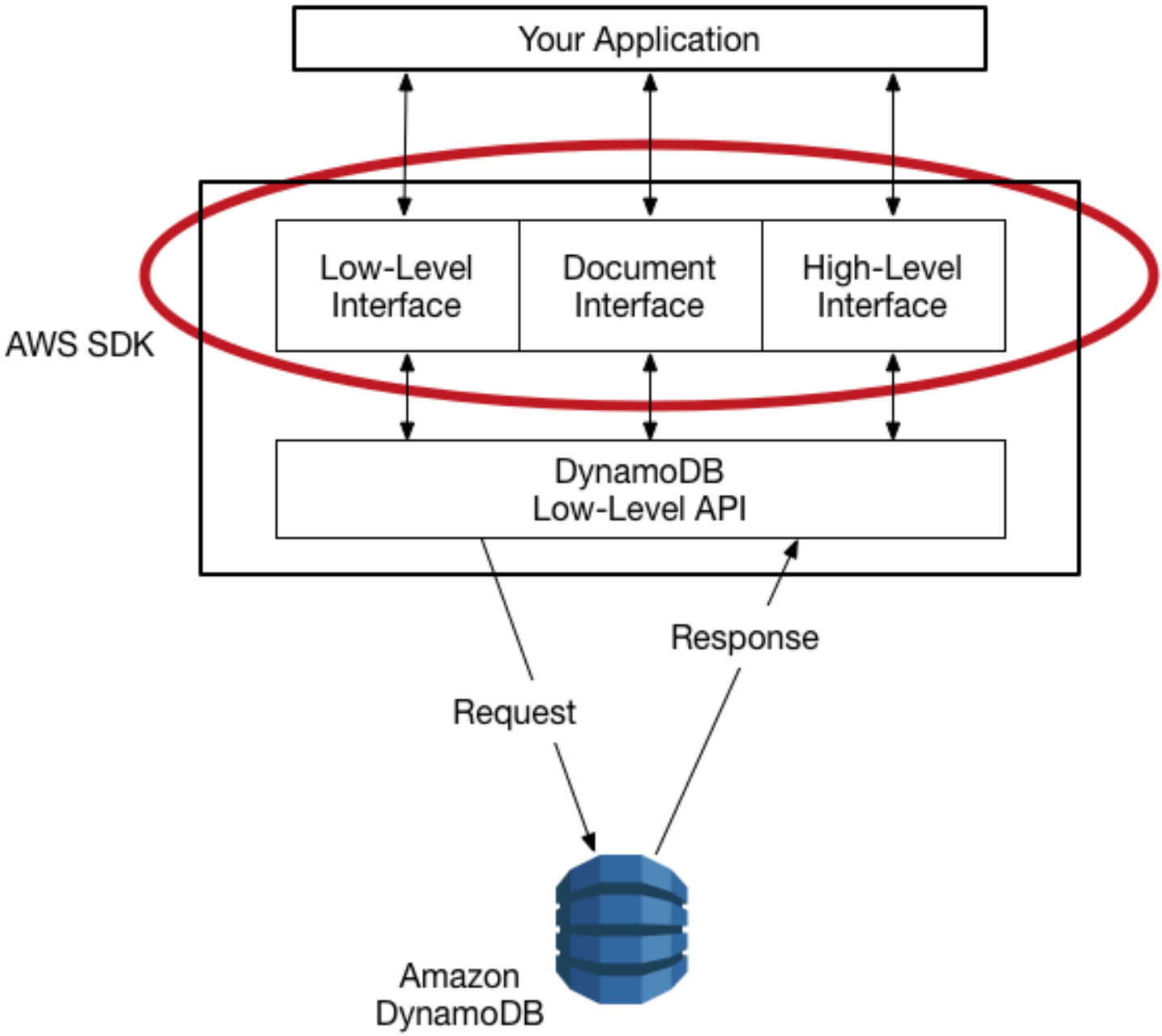
AWS は、2024 年 9 月 4 日、SDK for Java V1 から、DynamoDB の AWS アカウントベースのエンドポイントの AWS SDK サポートをロールアウトします。これらの新しいエンドポイントは、高いパフォーマンスとスケーラビリティを AWS で確保するのに役立ちます。更新された SDK では、`https://(account-id).ddb.(region).amazonaws.com` 形式の新しいエンドポイントを自動的に使用します。

SDK クライアントの 1 つのインスタンスを使用して複数のアカウントにリクエストを行う場合、アプリケーションは接続を再利用する機会が少なくなります。AWS では、SDK クライアントイ

ンスタンスあたりの接続先のアカウント数を減らすようにアプリケーションを変更することをお勧めします。代わりに、「AWS SDK およびツールリファレンスガイド」に記載されているように、ACCOUNT_ID_ENDPOINT_MODE 設定を使用してリージョンエンドポイントを引き続き使用するように SDK クライアントを設定することもできます。

DynamoDB で動作するプログラムインターフェイス

各 AWS SDK には、Amazon DynamoDB を操作するプログラミングインターフェイスが 1 つ以上用意されています。これらのインターフェイスは、シンプルな低レベルの DynamoDB ラッパーから、オブジェクト指向の永続レイヤーまで、多岐にわたります。使用可能なインターフェイスは、AWS SDK およびプログラミング言語によって異なります。



次のセクションでは、AWS SDK for Java を例として使用して、使用可能なインターフェイスをいくつかハイライトします。(すべてのインターフェイスが、どの AWS SDK でも使用できるわけではありません。)

トピック

- DynamoDB で動作する低レベルインターフェイス
- DynamoDB で動作するドキュメントインターフェイス
- DynamoDB で動作するオブジェクト永続性インターフェイス

DynamoDB で動作する低レベルインターフェイス

それぞれの言語特有の AWS SDK は、低レベルの DynamoDB API リクエストによく似たメソッドを持つ、Amazon DynamoDB 用の低レベルインターフェイスを提供します。

場合によっては データ型記述子 を使用して、文字列型の S や、数値型の N など、属性のデータ型を識別する必要があります。

> **Note**
>
> すべての言語特有の AWS SDK で、低レベルインターフェイスを使用できます。

次の Java プログラムは、AWS SDK for Java の低レベルインターフェイスを使用しています。

低レベルインターフェイスの例

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
```

```
                    <tableName> <key> <keyVal>

                Where:
                    tableName - The Amazon DynamoDB table from which an item is
retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).\s
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal, tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName, String
key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        GetItemRequest request = GetItemRequest.builder()
                .key(keyToGet)
                .tableName(tableName)
                .build();

        try {
            // If there is no matching item, GetItem does not return any data.
            Map<String, AttributeValue> returnedItem = ddb.getItem(request).item();
```

```
            if (returnedItem.isEmpty())
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

DynamoDB で動作するドキュメントインターフェイス

多くの AWS SDK はドキュメントインターフェイスを備えており、テーブルやインデックスに対してデータプレーンオペレーション (作成、読み取り、更新、削除) を実行できます。ドキュメントインターフェイスでは、データ型記述子 を指定する必要はありません。データ型は、データ自体のセマンティクスによって暗示されています。これらの AWS SDK には、JSON ドキュメントとネイティブの Amazon DynamoDB データ型とを簡単に変換するメソッドも用意されています。

> **Note**
>
> ドキュメントインターフェイスは、AWS SDK for Java、.NET、Node.js、および JavaScript SDK で使用できます。

次の Java プログラムは、AWS SDK for Java のドキュメントインターフェイスを使用します。プログラムが `Music` テーブルを表す `Table` オブジェクトを作成し、そのオブジェクトに `GetItem` の使用を依頼して、曲を取得します。その後プログラムが、曲がリリースされた年を出力します。

`software.amazon.dynamodb.document.DynamoDB` クラスは DynamoDB ドキュメントインターフェイスを実装します。低レベルクライアント (`AmazonDynamoDB`) のラッパーとして、DynamoDB がどのような役割を果たすかに注目してください。

ドキュメントインターフェイスの例

```
package com.amazonaws.codesamples.gsg;

import software.amazon.dynamodb.AmazonDynamoDB;
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import software.amazon.dynamodb.document.DynamoDB;
import software.amazon.dynamodb.document.GetItemOutcome;
import software.amazon.dynamodb.document.Table;

public class MusicDocumentDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");
        GetItemOutcome outcome = table.getItemOutcome(
                "Artist", "No One You Know",
                "SongTitle", "Call Me Today");

        int year = outcome.getItem().getInt("Year");
        System.out.println("The song was released in " + year);

    }
}
```

DynamoDB で動作するオブジェクト永続性インターフェイス

一部の AWS SDK には、データプレーンオペレーションを直接実行しない、オブジェクト永続性インターフェイスが用意されています。代わりに、Amazon DynamoDB のテーブルとインデックスの項目を表すオブジェクトを作成し、それらのオブジェクトのみとやり取りを行います。これにより、データベースを意識するのではなく、オブジェクトを中心にコードを作成していくことができます。

> **Note**
>
> オブジェクト永続性インターフェイスは、AWS SDK for Java と for .NET で利用できます。詳細については、DynamoDB の「DynamoDB の上位レベルのプログラミングインターフェイス」を参照してください。

オブジェクト永続性インターフェイスの例

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import com.example.dynamodb.Customer;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import software.amazon.awssdk.enhanced.dynamodb.model.GetItemEnhancedRequest;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;

/*
 * Before running this code example, create an Amazon DynamoDB table named Customer
 with these columns:
 *   - id - the id of the record that is the key. Be sure one of the id values is
 `id101`
 *   - custName - the customer name
 *   - email - the email value
 *   - registrationDate - an instant value when the item was added to the table. These
 values
 *                        need to be in the form of `YYYY-MM-DDTHH:mm:ssZ`, such as
 2022-07-11T00:00:00Z
 *
 * Also, ensure that you have set up your development environment, including your
 credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */

public class EnhancedGetItem {
```

```
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

        getItem(enhancedClient);
        ddb.close();
    }

    public static String getItem(DynamoDbEnhancedClient enhancedClient) {
        Customer result = null;
        try {
            DynamoDbTable<Customer> table = enhancedClient.table("Customer",
 TableSchema.fromBean(Customer.class));
            Key key = Key.builder()
                    .partitionValue("id101").sortValue("tred@noserver.com")
                    .build();

            // Get the item by using the key.
            result = table.getItem(
                    (GetItemEnhancedRequest.Builder requestBuilder) ->
 requestBuilder.key(key));
            System.out.println("******* The description value is " +
 result.getCustName());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return result.getCustName();
    }
}
```

DynamoDB の上位レベルのプログラミングインターフェイス

AWS SDK は、Amazon DynamoDB を操作するための下位レベルのインターフェイスをアプリケーションに提供します。これらのクライアント側のクラスとメソッドは、下位レベルの DynamoDB API と直接対応しています。しかし、これは多くのデベロッパーにとって、複雑なデータ型をデータ

ベーステーブル内の項目にマップする必要がある場合の接続性の悪さ、つまりインピーダンスの不整合の原因となります。下位レベルのデータベースインタフェースを使用するデベロッパーは、オブジェクトデータをデータベーステーブルに対して読み書きするためのメソッドを記述する必要があります。オブジェクトタイプとデータベーステーブルの組み合わせに、個別に対応するための余分なコードが、圧倒的な量になることもあります。

AWS SDK for Java および for .NET では、開発業務を簡素化するために、高度に抽象化されたインターフェイスが用意されています。DynamoDB の上位レベルのインターフェイスを使用すると、プログラム内のオブジェクトと、それらのオブジェクトのデータを格納するためのデータベーステーブル間の関係を定義できます。このマッピングを定義した後、`save`、`load`、`delete` などの単純なオブジェクトメソッドを呼び出すと、ユーザーに代わって、基盤となる下位レベルの DynamoDB オペレーションの呼び出しが自動化されます。これにより、データベースを意識するのではなく、オブジェクトを中心にコードを作成していくことができます。

DynamoDB の上位レベルのプログラミングインターフェイスは、Java および .NET 用の AWS SDK で使用できます。

Java:

- Java 1.x: DynamoDBMapper
- Java 2.x: DynamoDB 拡張クライアント

.NET

- DynamoDB での .NET ドキュメントの操作
- .NET オブジェクト永続性モデルと DynamoDB の使用

Java 1.x: DynamoDBMapper

> **Note**
>
> SDK for Java には、1.x と 2.x の 2 つのバージョンがあります。1.x のサポート終了は、2024 年 1 月 12 日に発表されました。サポートは 2025 年 12 月 31 日に終了します。新規開発には、2.x を使用することを強くお勧めします。

AWS SDK for Java には `DynamoDBMapper` クラスが用意されているため、クライアント側のクラスを Amazon DynamoDB テーブルにマッピングできます。`DynamoDBMapper` を使用するに

は、DynamoDB テーブル内の項目と、それが対応するコード内のオブジェクトインスタンスの間での関係性を定義します。DynamoDBMapper クラスを使用すると、項目に対する作成、読み込み、更新、削除 (CRUD) の各オペレーションの実行、およびテーブルに対するクエリやスキャンを行うことができます。

トピック

- DynamoDBMapper クラス
- DynamoDBMapper for Java でサポートされるデータ型
- DynamoDB 用の Java アノテーション
- DynamoDBMapper のオプションの構成設定
- DynamoDB およびバージョン番号を使用した楽観的ロック
- DynamoDB での任意のデータのマッピング
- DynamoDBMapper の例

> **Note**
>
> DynamoDBMapperクラスでは、テーブルを作成、更新、または削除することはできません。これらのタスクを実行するには、代わりに SDK for Java の下位レベルインターフェイスを使用します。

SDK for Java には、クラスをテーブルにマッピングするための、一連のアノテーションタイプが用意されています。たとえば、パーティションキーとして ProductCatalog を含む Id テーブルがあるとします。

```
ProductCatalog(Id, ...)
```

次の Java コードに示すように、クライアントアプリケーション内のクラスを ProductCatalog テーブルにマッピングすることができます。このコードでは、CatalogItem という名前の Plain Old Java Object (POJO) を定義しています。このオブジェクトは、アノテーションを使用して、オブジェクトフィールドを DynamoDB の属性名にマッピングします。

Example

```
package com.amazonaws.codesamples;
```

```
import java.util.Set;

import software.amazon.dynamodb.datamodeling.DynamoDBAttribute;
import software.amazon.dynamodb.datamodeling.DynamoDBHashKey;
import software.amazon.dynamodb.datamodeling.DynamoDBIgnore;
import software.amazon.dynamodb.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) {this.id = id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() {return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    @DynamoDBAttribute(attributeName="Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
 bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp; }
    public void setSomeProp(String someProp) { this.someProp = someProp; }
}
```

前述のコードでは、@DynamoDBTable 注釈によって、CatalogItem クラスが ProductCatalog テーブルにマッピングされています。個々のクラスインスタンスは、テーブル内の項目として格納できます。クラス定義では、@DynamoDBHashKey 注釈によって Id プロパティがプライマリキーにマッピングされます。

デフォルトでは、クラスプロパティはテーブル内の同じ名前属性にマッピングされます。プロパティ Title および ISBN は、テーブル内の同じ名前属性にマッピングされます。

DynamoDB の属性名がクラスで宣言されたプロパティ名と一致する場合、@DynamoDBAttribute アノテーションの使用はオプションです。これらの名前が異なる場合には、attributeName パラメータを指定しながらこのアノテーションを使用し、プロパティが DynamoDB のどの属性に対応しているかを指定します。

前述の例では、各プロパティに @DynamoDBAttribute 注釈を追加することで、プロパティ名が前のステップで作成したテーブルに確実に一致し、このガイド内の他のコード例で使用されている属性名との整合性がとられています。

クラス定義には、テーブル内のどの属性にもマッピングされないプロパティを含めることもできます。これらのプロパティを特定するには、@DynamoDBIgnore 注釈を追加します。前述の例では、SomeProp プロパティが @DynamoDBIgnore 注釈によってマーキングされています。CatalogItem インスタンスをテーブルにアップロードしたとき、DynamoDBMapper インスタンスに SomeProp プロパティは追加されません。また、このマッパーは、テーブルから項目を取り出すときにこの属性を返しません。

マッピングクラスを定義したら、DynamoDBMapper メソッドを使用して、そのクラスのインスタンスを Catalog テーブルの対応する項目に書き込むことができます。次のサンプルコードでは、この技術を示しています。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapper mapper = new DynamoDBMapper(client);

CatalogItem item = new CatalogItem();
item.setId(102);
item.setTitle("Book 102 Title");
item.setISBN("222-2222222222");
item.setBookAuthors(new HashSet<String>(Arrays.asList("Author 1", "Author 2")));
item.setSomeProp("Test");

mapper.save(item);
```

次のサンプルコードでは、項目を取得し、その属性の一部にアクセスする方法を示します。

```
CatalogItem partitionKey = new CatalogItem();
```

```
partitionKey.setId(102);
DynamoDBQueryExpression<CatalogItem> queryExpression = new
 DynamoDBQueryExpression<CatalogItem>()
    .withHashKeyValues(partitionKey);

List<CatalogItem> itemList = mapper.query(CatalogItem.class, queryExpression);

for (int i = 0; i < itemList.size(); i++) {
    System.out.println(itemList.get(i).getTitle());
    System.out.println(itemList.get(i).getBookAuthors());
}
```

DynamoDBMapper は、Java 内で DynamoDB データを操作するための直観的で自然な方法を提供します。また、オプティミスティックロック、ACID トランザクション、自動生成されるパーティションキーとソートキーの値、オブジェクトのバージョニングなど、複数の組み込み機能があります。

DynamoDBMapper クラス

DynamoDBMapper クラスは、Amazon DynamoDB のエントリポイントとなります。このクラスは、DynamoDB エンドポイントへのアクセスを提供し、さまざまなテーブルのデータを使用できるようにします。また、アイテムに対する作成、読み込み、更新、削除 (CRUD) の各オペレーションの実行、およびテーブルに対するクエリやスキャンを行うことができます。このクラスでは、DynamoDB を操作するために以下のメソッドが提供されます。

対応する Javadoc ドキュメントについては、AWS SDK for Java API リファレンスの「DynamoDBMapper」を参照してください。

トピック

- save
- load
- 削除
- query
- queryPage
- スキャン
- scanPage
- parallelScan
- batchSave

- batchLoad
- batchDelete
- バッチ書き込み
- transactionWrite
- transactionLoad
- count
- generateCreateTableRequest
- createS3Link
- getS3ClientCache

save

指定したオブジェクトがテーブルに保存されます。このメソッドで必要なパラメータは、保存するオブジェクトだけです。DynamoDBMapperConfig オブジェクトを使用して、オプションの設定パラメータを入力できます。

同じプライマリキーを持つ項目が存在しない場合は、このメソッドによってテーブル内に新しい項目が作成されます。同じプライマリキーを持つ項目が存在する場合は、その既存の項目が更新されます。パーティションキーとソートキーが String 型で、@DynamoDBAutoGeneratedKey によって注釈が付けられている場合、初期化しなければ、ランダムな UUID (Universally Unique Identifier) が与えられます。@DynamoDBVersionAttribute で注釈が付けられたバージョンフィールドは、バージョンが 1 ずつ増えていきます。さらに、バージョンフィールドが更新されるかキーが生成されると、オペレーションの結果として、渡されたオブジェクトが更新されます。

デフォルトでは、マップされたクラスプロパティに対応する属性のみが更新されます。アイテムのその他の既存の属性には影響はありません。ただし、SaveBehavior.CLOBBER を指定すると、項目が完全に上書きされるようにすることができます。

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER).build();

mapper.save(item, config);
```

バージョニングを有効にした場合は、クライアント側とサーバー側で項目のバージョンが一致する必要があります。ただし、SaveBehavior.CLOBBER オプションを使用する場合は、バージョンを一

致させる必要はありません。バージョニングの詳細については、「DynamoDB およびバージョン番号を使用した楽観的ロック」を参照してください。

load

テーブルから項目を取り出します。取得する項目のプライマリキーを入力する必要があります。DynamoDBMapperConfig オブジェクトを使用して、オプションの設定パラメータを入力できます。例えば、次の Java ステートメントに示すように、オプションで強力な整合性のある読み込みをリクエストして、このメソッドによって最新の項目の値だけを取り出すようにすることができます。

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT).build();

CatalogItem item = mapper.load(CatalogItem.class, item.getId(), config);
```

DynamoDB のデフォルトでは、結果整合性のある値を持つ項目が返されます。DynamoDB の結果整合性モデルの詳細については、「DynamoDB の読み取り整合性」を参照してください。

削除

テーブルから項目を削除します。マッピングされたクラスのオブジェクトインスタンスを渡す必要があります。

バージョニングを有効にした場合は、クライアント側とサーバー側で項目のバージョンが一致する必要があります。ただし、SaveBehavior.CLOBBER オプションを使用する場合は、バージョンを一致させる必要はありません。バージョニングの詳細については、「DynamoDB およびバージョン番号を使用した楽観的ロック」を参照してください。

query

テーブルまたはセカンダリインデックスをクエリする

フォーラムスレッドの返信を格納する Reply というテーブルがあるとします。各スレッドの件名については、0 以上の返信を受け取ることができます。Reply テーブルのプライマリキーは、Id および ReplyDateTime フィールドで構成されます。ここで、Id はプライマリキーのパーティションキー、ReplyDateTime はソートキーを表します。

```
Reply ( Id, ReplyDateTime, ... )
```

Reply クラスと、それに対応する DynamoDB 内の Reply テーブルの間で、マッピングを作成したとします。次の Java コードでは、DynamoDBMapper を使用して特定のスレッド件名に対する過去 2 週間のすべての返信を検索しています。

Example

```
String forumName = "&DDB;";
String forumSubject = "&DDB; Thread 1";
String partitionKey = forumName + "#" + forumSubject;

long twoWeeksAgoMilli = (new Date()).getTime() - (14L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS(partitionKey));
eav.put(":v2",new AttributeValue().withS(twoWeeksAgoStr.toString()));

DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryExpression<Reply>()
    .withKeyConditionExpression("Id = :v1 and ReplyDateTime > :v2")
    .withExpressionAttributeValues(eav);

List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);
```

このクエリでは、Reply オブジェクトのコレクションが返されます。

デフォルトでは、query メソッドによって、「遅延ロード」されたコレクションが返されます。最初に結果が 1 ページのみ返され、必要に応じて、さらに次ページを要求するサービス呼び出しが行われます。一致するすべての項目を取得するには、latestReplies コレクションを反復処理します。

コレクションで size() メソッドを呼び出すと、正確なカウントを提供するためにすべての結果がロードされます。これにより、プロビジョニングされたスループットが大量に消費され、非常に大きなテーブルでは JVM 内のすべてのメモリが消費されることさえあります。

インデックスにクエリを実行するには、最初にインデックスをマッパークラスとしてモデリングする必要があります。今、Reply テーブルに、PostedBy-Message-Index という名前のグローバルセカンダリインデックスが存在すると仮定します。このインデックスのパーティションキーは PostedBy キーで、ソートキーは Message です。インデックス内の項目のクラス定義は次のようになります。

```
@DynamoDBTable(tableName="Reply")
public class PostedByMessage {
    private String postedBy;
    private String message;

    @DynamoDBIndexHashKey(globalSecondaryIndexName = "PostedBy-Message-Index",
 attributeName = "PostedBy")
    public String getPostedBy() { return postedBy; }
    public void setPostedBy(String postedBy) { this.postedBy = postedBy; }

    @DynamoDBIndexRangeKey(globalSecondaryIndexName = "PostedBy-Message-Index",
 attributeName = "Message")
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }

   // Additional properties go here.
}
```

@DynamoDBTable 注釈は、このインデックスが Reply テーブルに関連付けられていることを示します。@DynamoDBIndexHashKey 注釈はインデックスのパーティションキー (PostedBy) を示し、@DynamoDBIndexRangeKey はインデックスのソートキー (Message) を示します。

ここで、DynamoDBMapper を使用してインデックスにクエリを実行し、特定のユーザーによって投稿されたメッセージのサブセットを取得できます。テーブル間やインデックス間でマッピングが競合せず、マッピングが既にマッパーで行われている場合は、インデックス名を指定する必要はありません。マッパーは、プライマリキーとソートキーに基づいて推論します。次のコードでは、グローバルセカンダリインデックスをクエリしています。グローバルセカンダリインデックスは、結果整合性のある読み込みをサポートしますが、強力な整合性のある読み込みはサポートしていないため、withConsistentRead(false) を指定する必要があります。

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1",  new AttributeValue().withS("User A"));
eav.put(":v2",  new AttributeValue().withS("DynamoDB"));

DynamoDBQueryExpression<PostedByMessage> queryExpression = new
 DynamoDBQueryExpression<PostedByMessage>()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false)
    .withKeyConditionExpression("PostedBy = :v1 and begins_with(Message, :v2)")
    .withExpressionAttributeValues(eav);
```

```
List<PostedByMessage> iList =  mapper.query(PostedByMessage.class, queryExpression);
```

このクエリでは、PostedByMessage オブジェクトのコレクションが返されます。

queryPage

テーブルまたはセカンダリインデックスのクエリを実行し、一致した結果を 1 ページ返します。query メソッドと同様、パーティションキー値とソートキー属性に適用されるクエリフィルタを指定する必要があります。ただし queryPage では、データの最初の "ページ"、つまり 1 MB 以内に収まるデータ量のみが返されます。

スキャン

テーブル全体またはセカンダリインデックスをスキャンします。オプションで FilterExpression を指定して結果セットをフィルタリングできます。

フォーラムスレッドの返信を格納する Reply というテーブルがあるとします。各スレッドの件名については、0 以上の返信を受け取ることができます。Reply テーブルのプライマリキーは、Id および ReplyDateTime フィールドで構成されます。ここで、Id はプライマリキーのパーティションキー、ReplyDateTime はソートキーを表します。

```
Reply ( Id, ReplyDateTime, ... )
```

Reply テーブルに Java クラスをマッピングした場合は、DynamoDBMapper を使用してテーブルをスキャンできます。たとえば、以下の Java コードは、Reply テーブル全体をスキャンし、特定の年の返信のみを返します。

Example

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS("2015"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("begins_with(ReplyDateTime,:v1)")
    .withExpressionAttributeValues(eav);

List<Reply> replies =  mapper.scan(Reply.class, scanExpression);
```

デフォルトでは、scan メソッドによって、「遅延ロード」されたコレクションが返されます。最初に結果が 1 ページのみ返され、必要に応じて、さらに次ページを要求するサービス呼び出しが行われます。一致するすべての項目を取得するには、replies コレクションを反復処理します。

コレクションで size() メソッドを呼び出すと、正確なカウントを提供するためにすべての結果がロードされます。これにより、プロビジョニングされたスループットが大量に消費され、非常に大きなテーブルでは JVM 内のすべてのメモリが消費されることさえあります。

インデックスをスキャンするには、最初にインデックスをマッパークラスとしてモデリングする必要があります。今、Reply テーブルには、PostedBy-Message-Index という名前のグローバルセカンダリインデックスがあると仮定します。このインデックスのパーティションキーは PostedBy キーで、ソートキーは Message です。このインデックスのマッパークラスは、query セクションに示されています。また、@DynamoDBIndexHashKey と @DynamoDBIndexRangeKey の注釈を使用して、インデックスパーティションキーとソートキーを指定します。

以下のサンプルコードでは PostedBy-Message-Index をスキャンします。スキャンフィルタを使用しないので、インデックス内のすべての項目が返されます。

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false);

    List<PostedByMessage> iList =  mapper.scan(PostedByMessage.class, scanExpression);
    Iterator<PostedByMessage> indexItems = iList.iterator();
```

scanPage

テーブルまたはセカンダリインデックスがスキャンされ、一致する結果が 1 ページ返されます。scan メソッドと同様に、オプションで FilterExpression を指定して結果セットをフィルタリングできます。ただし、scanPage では、データの最初の "ページ"、つまり、1 MB 以内に収まるデータ量のみが返されます。

parallelScan

テーブル全体、またはセカンダリインデックスの並列スキャンが実行されます。テーブルの論理セグメントの数と、結果をフィルタするスキャン式を指定します。parallelScan では、スキャンタスクが複数のワーカーに分割され、論理セグメントごとに 1 つのワーカーが割り当てられます。ワーカーは、データを並列に処理し、結果を返します。

次の Java コード例では、Product テーブルに対して並列スキャンを実行します。

```
int numberOfThreads = 4;

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":n", new AttributeValue().withN("100"));
```

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("Price <= :n")
    .withExpressionAttributeValues(eav);

List<Product> scanResult = mapper.parallelScan(Product.class, scanExpression,
 numberOfThreads);
```

batchSave

AmazonDynamoDB.batchWriteItem メソッドに対する 1 つ以上の呼び出しを使用して、1 つ以上のテーブルにオブジェクトを保存します。このメソッドでは、トランザクション保証はなされません。

次の Java コードでは、2 つの項目 (書籍) を ProductCatalog テーブルに保存します。

```
Book book1 = new Book();
book1.setId(901);
book1.setProductCategory("Book");
book1.setTitle("Book 901 Title");

Book book2 = new Book();
book2.setId(902);
book2.setProductCategory("Book");
book2.setTitle("Book 902 Title");

mapper.batchSave(Arrays.asList(book1, book2));
```

batchLoad

テーブルのプライマリキーを使用して、1 つ以上のテーブルから複数の項目を取り出します。

次の Java コードでは、2 つの異なるテーブルから 2 つの項目を取得します。

```
ArrayList<Object> itemsToGet = new ArrayList<Object>();

ForumItem forumItem = new ForumItem();
forumItem.setForumName("Amazon DynamoDB");
itemsToGet.add(forumItem);

ThreadItem threadItem = new ThreadItem();
threadItem.setForumName("Amazon DynamoDB");
threadItem.setSubject("Amazon DynamoDB thread 1 message text");
```

```
itemsToGet.add(threadItem);

Map<String, List<Object>> items = mapper.batchLoad(itemsToGet);
```

batchDelete

AmazonDynamoDB.batchWriteItem メソッドに対する 1 つ以上の呼び出しを使用して、1 つ以上のテーブルからオブジェクトを削除します。このメソッドでは、トランザクション保証はなされません。

次の Java コードでは、2 つの項目 (書籍) を ProductCatalog テーブルから削除します。

```
Book book1 = mapper.load(Book.class, 901);
Book book2 = mapper.load(Book.class, 902);
mapper.batchDelete(Arrays.asList(book1, book2));
```

バッチ書き込み

AmazonDynamoDB.batchWriteItem メソッドに対する 1 つ以上の呼び出しを使用して、1 つ以上のテーブルに対してオブジェクトの保存および削除を行います。このメソッドではトランザクション保証はなされず、バージョニング (条件付き入力または削除) もサポートされません。

次の Java コードでは、新しい項目を Forum テーブルと Thread テーブルに書き込み、ProductCatalog テーブルから項目を削除します。

```
// Create a Forum item to save
Forum forumItem = new Forum();
forumItem.setName("Test BatchWrite Forum");

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.setForumName("AmazonDynamoDB");
threadItem.setSubject("My sample question");

// Load a ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

mapper.batchWrite(objectsToWrite, objectsToDelete);
```

transactionWrite

AmazonDynamoDB.transactWriteItems メソッドに対する 1 回の呼び出しを使用して、1 つまたは複数のテーブルに対してオブジェクトの保存および削除を行います。

トランザクション固有の例外のリストについては、「TransactWriteItems エラー」を参照してください。

DynamoDB トランザクション、および提供されるアトミック性、整合性、分離、耐久性 (ACID) の保証の詳細については、「DynamoDB トランザクションで複雑なワークフローを管理する」を参照してください。

> **Note**
>
> このメソッドでは、以下をサポートしていません。
>
> - DynamoDBMapperConfig.SaveBehavior

次の Java コードでは、Forum と Thread の各テーブルに、トランザクションとして新しい項目を書き込みます。

```
Thread s3ForumThread = new Thread();
s3ForumThread.setForumName("S3 Forum");
s3ForumThread.setSubject("Sample Subject 1");
s3ForumThread.setMessage("Sample Question 1");

Forum s3Forum = new Forum();
s3Forum.setName("S3 Forum");
s3Forum.setCategory("Amazon Web Services");
s3Forum.setThreads(1);

TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
transactionWriteRequest.addPut(s3Forum);
transactionWriteRequest.addPut(s3ForumThread);
mapper.transactionWrite(transactionWriteRequest);
```

transactionLoad

AmazonDynamoDB.transactGetItems メソッドへの 1 回の呼び出しを使用して、1 つまたは複数のテーブルからオブジェクトをロードします。

トランザクション固有の例外のリストについては、「TransactGetItems エラー」を参照してください。

DynamoDB トランザクション、および提供されるアトミック性、整合性、分離、耐久性 (ACID) の保証の詳細については、「DynamoDB トランザクションで複雑なワークフローを管理する」を参照してください。

次の Java コードは、 `Forum` と `Thread` の各テーブルから、トランザクションとして一方の項目をロードします。

```
Forum dynamodbForum = new Forum();
dynamodbForum.setName("DynamoDB Forum");
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.setForumName("DynamoDB Forum");

TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();
transactionLoadRequest.addLoad(dynamodbForum);
transactionLoadRequest.addLoad(dynamodbForumThread);
mapper.transactionLoad(transactionLoadRequest);
```

count

指定されたスキャン式の値を求め、一致する項目数を返します。項目データは返されません。

generateCreateTableRequest

DynamoDB テーブルに対応する POJO クラスを解析し、そのテーブルの `CreateTableRequest` を返します。

createS3Link

Amazon S3 内にあるオブジェクトへのリンクを作成します。バケット名とキー名を指定する必要があります。キー名によって、バケット内のオブジェクトを一意に識別します。

`createS3Link` を使用するには、マッパークラスでゲッターメソッドとセッターメソッドを定義する必要があります。次のサンプルコードでは、これを示しており、新しい属性と getter/setter メソッドを `CatalogItem` クラスに追加しています。

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    ...
```

```
    public S3Link productImage;

    ....

    @DynamoDBAttribute(attributeName = "ProductImage")
    public S3Link getProductImage() {
            return productImage;
    }

    public void setProductImage(S3Link productImage) {
        this.productImage = productImage;
    }

...
}
```

次の Java コードでは、Product テーブルに書き込まれる新しい項目を定義しています。この項目には、製品イメージへのリンクが含まれ、そのイメージデータは Amazon S3 にアップロードされています。

```
CatalogItem item = new CatalogItem();

item.setId(150);
item.setTitle("Book 150 Title");

String amzn-s3-demo-bucket = "amzn-s3-demo-bucket";
String myS3Key = "productImages/book_150_cover.jpg";
item.setProductImage(mapper.createS3Link(amzn-s3-demo-bucket, myS3Key));

item.getProductImage().uploadFrom(new File("/file/path/book_150_cover.jpg"));

mapper.save(item);
```

S3Link クラスには、他にも、Amazon S3 のオブジェクトを操作するためのさまざまなメソッドが用意されています。詳細については、「Javadocs for S3Link」を参照してください。

getS3ClientCache

Amazon S3 にアクセスする際の基盤となる S3ClientCache を返します。S3ClientCache は、AmazonS3Client オブジェクトのスマートマップです。複数のクライアントがある場合、S3ClientCache によって、AWS リージョン別にクライアントを整理しやすくなり、新しい Amazon S3 クライアントをオンデマンドで作成できるようになります。

DynamoDBMapper for Java でサポートされるデータ型

このセクションでは、Amazon DynamoDB でサポートされているプリミティブな Java データ型、コレクション、および任意のデータ型について説明します。

Amazon DynamoDB では、次のプリミティブな Java データ型とプリミティブなラッパークラスがサポートされています。

- `String`
- `Boolean`, `boolean`
- `Byte`, `byte`
- `Date` (ISO_8601 ミリ秒精度文字列、UTC に転換)
- `Calendar` (ISO_8601 ミリ秒精度文字列、UTC に転換)
- `Long`, `long`
- `Integer`, `int`
- `Double`, `double`
- `Float`, `float`
- `BigDecimal`
- `BigInteger`

> **Note**
>
> - DynamoDB での命名規則およびサポートされている各データ型の詳細については、「Amazon DynamoDB でサポートされるデータ型と命名規則」を参照してください。
> - 空のバイナリ値は、DynamoDBMapper でサポートされます。
> - 空の文字列値は、AWS SDK for Java 2.x でサポートされます。
>
>   AWS SDK for Java 1.x では、DynamoDBMapper による空の文字列属性値の読み取りをサポートしています。ただし、空の文字列属性値の書き込みは、その属性がリクエストから削除されるため実行されません。

DynamoDB では、Java の Set、List、および Map コレクションタイプがサポートされています。次の表に、これらの Java 型が DynamoDB 型にどのようにマッピングされるかを示します。

Java 型	DynamoDB 型
すべての数値型	N (数値型)
文字列	S (文字列型)
Boolean	BOOL (ブール型)、0 または 1。
ByteBuffer	B (バイナリ型)
日付	S (文字列型)。Date の値は、ISO-8601 形式の文字列として格納されます。
Set コレクション型	SS (文字列セット) 型、NS (数値セット) 型、または BS (バイナリセット) 型。

DynamoDBTypeConverter インターフェイスでは、独自の任意データ型を、DynamoDB でネイティブにサポートされているデータ型にマッピングすることができます。詳細については、「」を参照してくださいDynamoDB での任意のデータのマッピング

DynamoDB 用の Java アノテーション

このセクションでは、クラスとプロパティを Amazon DynamoDB のテーブルと属性にマッピングするためのアノテーションについて説明します。

関連する Javadoc ドキュメントについては、AWS SDK for Java API リファレンスの「アノテーションタイプの概要」を参照してください。

> **Note**
>
> 次の注釈では、DynamoDBTable と DynamoDBHashKey だけが必須です。

トピック

- DynamoDBAttribute
- DynamoDBAutoGeneratedKey
- DynamoDBAutoGeneratedTimestamp

- DynamoDBDocument
- DynamoDBHashKey
- DynamoDBIgnore
- DynamoDBIndexHashKey
- DynamoDBIndexRangeKey
- DynamoDBRangeKey
- DynamoDBTable
- DynamoDBTypeConverted
- DynamoDBTyped
- DynamoDBVersionAttribute

DynamoDBAttribute

テーブルの属性にプロパティをマッピングします。デフォルトでは、各クラスのプロパティが、同じ名前の項目属性にマッピングされます。ただし名前が同じでない場合は、この注釈を使用して属性にプロパティをマッピングすることができます。次の Java コードスニペットでは、DynamoDBAttribute によって、BookAuthors プロパティがテーブル内の 属性名にマッピングされています。Authors

```
@DynamoDBAttribute(attributeName = "Authors")
public List<String> getBookAuthors() { return BookAuthors; }
public void setBookAuthors(List<String> BookAuthors) { this.BookAuthors =
 BookAuthors; }
```

DynamoDBMapper では、テーブルにオブジェクトを保存する際に、 を属性名として使用しています。Authors

DynamoDBAutoGeneratedKey

パーティションキーまたはソートキーのプロパティは、自動生成済みとしてマーキングされます。DynamoDBMapper では、これらの属性を保存するときにランダムな UUID が生成されます。String プロパティには、自動生成済みのキーとしてマーキングできます。

次の例は、自動生成されたパラメータを使用する方法を示しています。

```
@DynamoDBTable(tableName="AutoGeneratedKeysExample")
```

```
public class AutoGeneratedKeys {
    private String id;
    private String payload;

    @DynamoDBHashKey(attributeName = "Id")
    @DynamoDBAutoGeneratedKey
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    @DynamoDBAttribute(attributeName="payload")
    public String getPayload() { return this.payload; }
    public void setPayload(String payload) { this.payload = payload; }

    public static void saveItem() {
        AutoGeneratedKeys obj = new AutoGeneratedKeys();
        obj.setPayload("abc123");

        // id field is null at this point
        DynamoDBMapper mapper = new DynamoDBMapper(dynamoDBClient);
        mapper.save(obj);

        System.out.println("Object was saved with id " + obj.getId());
    }
}
```

DynamoDBAutoGeneratedTimestamp

タイムスタンプを自動生成します。

```
@DynamoDBAutoGeneratedTimestamp(strategy=DynamoDBAutoGenerateStrategy.ALWAYS)
public Date getLastUpdatedDate() { return lastUpdatedDate; }
public void setLastUpdatedDate(Date lastUpdatedDate) { this.lastUpdatedDate =
 lastUpdatedDate; }
```

オプションで、戦略属性を指定して自動生成戦略を定義できます。デフォルトは ALWAYS です。

DynamoDBDocument

Amazon DynamoDB ドキュメントにあるように、クラスは連番で作成できます。

例えば、JSON ドキュメントを Map 型 (M) の DynamoDB 属性にマッピングしたい場合を考えます。次のサンプルコードでは、マップ型の入れ子の属性 (Pictures) を含む項目を定義します。

```
public class ProductCatalogItem {

    private Integer id;  //partition key
    private Pictures pictures;
    /* ...other attributes omitted... */

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id;}
    public void setId(Integer id) {this.id = id;}

    @DynamoDBAttribute(attributeName="Pictures")
    public Pictures getPictures() { return pictures;}
    public void setPictures(Pictures pictures) {this.pictures = pictures;}

    // Additional properties go here.

    @DynamoDBDocument
    public static class Pictures {
        private String frontView;
        private String rearView;
        private String sideView;

        @DynamoDBAttribute(attributeName = "FrontView")
        public String getFrontView() { return frontView; }
        public void setFrontView(String frontView) { this.frontView = frontView; }

        @DynamoDBAttribute(attributeName = "RearView")
        public String getRearView() { return rearView; }
        public void setRearView(String rearView) { this.rearView = rearView; }

        @DynamoDBAttribute(attributeName = "SideView")
        public String getSideView() { return sideView; }
        public void setSideView(String sideView) { this.sideView = sideView; }

     }
}
```

以下の例に示されているように、ProductCatalog と合わせて、新しい Pictures 項目を保存することができます。

```
ProductCatalogItem item = new ProductCatalogItem();

Pictures pix = new Pictures();
```

```
pix.setFrontView("http://example.com/products/123_front.jpg");
pix.setRearView("http://example.com/products/123_rear.jpg");
pix.setSideView("http://example.com/products/123_left_side.jpg");
item.setPictures(pix);

item.setId(123);

mapper.save(item);
```

その結果、ProductCatalog 項目は次のようになります (JSON 形式)。

```
{
  "Id" : 123
  "Pictures" : {
    "SideView" : "http://example.com/products/123_left_side.jpg",
    "RearView" : "http://example.com/products/123_rear.jpg",
    "FrontView" : "http://example.com/products/123_front.jpg"
  }
}
```

DynamoDBHashKey

テーブルのパーティションキーにクラスプロパティをマッピングします。このプロパティは、スカラー文字列型、数値型、バイナリ型のいずれかである必要があります。コレクション型は使用できません。

ProductCatalog がプライマリキーである Id テーブルがあるとします。次の Java コードでは、CatalogItem クラスを定義し、その Id プロパティを ProductCatalog タグを使用して @DynamoDBHashKey テーブルのプライマリキーにマッピングしています。

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {
    private Integer Id;
   @DynamoDBHashKey(attributeName="Id")
   public Integer getId() {
        return Id;
   }
   public void setId(Integer Id) {
        this.Id = Id;
   }
   // Additional properties go here.
}
```

DynamoDBIgnore

DynamoDBMapper インスタンスに対して、関連するプロパティを無視するように指示します。テーブルにデータを保存する場合、DynamoDBMapper ではこのプロパティがテーブルに保存されません。

モデリングされていないプロパティの getter メソッドまたはクラスフィールドに適用されます。注釈がクラスフィールドに直接適用されている場合、対応する getter および setter が同じクラスで宣言される必要があります。

DynamoDBIndexHashKey

グローバルセカンダリインデックスのパーティションキーに、クラスプロパティをマッピングします。このプロパティは、スカラー文字列型、数値型、バイナリ型のいずれかである必要があります。コレクション型は使用できません。

グローバルセカンダリインデックスを Query したい場合に、このアノテーションを使用します。インデックス名 (globalSecondaryIndexName) を指定する必要があります。クラスプロパティの名前がインデックスのパーティションキーと異なる場合、そのインデックス属性の名前 (attributeName) も指定する必要があります。

DynamoDBIndexRangeKey

グローバルセカンダリインデックスまたはローカルセカンダリインデックスのソートキーにクラスプロパティをマッピングします。このプロパティは、スカラー文字列型、数値型、バイナリ型のいずれかである必要があります。コレクション型は使用できません。

このアノテーションは、ローカルセカンダリインデックスもしくはグローバルセカンダリインデックスを Query し、その結果をインデックスのソートキーを使用して絞り込む必要がある場合に使用します。インデックス名 (globalSecondaryIndexName または localSecondaryIndexName) を指定する必要があります。クラスプロパティの名前がインデックスのソートキーと異なる場合、そのインデックス属性の名前 (attributeName) も指定する必要があります。

DynamoDBRangeKey

テーブルのソートキーにクラスプロパティをマッピングします。このプロパティは、スカラー文字列型、数値型、バイナリ型のいずれかである必要があります。コレクション型は使用できません。

プライマリキーが複合の場合 (パーティションキーとソートキー)、このタグを使用して、クラスフィールドをソートキーにマッピングできます。たとえば、フォーラムスレッドからの返信を格納す

る Reply テーブルがあるとします。各スレッドには多数の返信がある可能性があります。したがってこのテーブルのプライマリキーは、ThreadId と ReplyDateTime の両方になります。パーティションキーは ThreadId で、ソートキーは ReplyDateTime です。

たとえば次の Java コードでは、Reply クラスを定義して、Reply テーブルにマッピングしています。ここでは、@DynamoDBHashKey タグと @DynamoDBRangeKey タグの両方を使用して、プライマリキーにマッピングされるクラスプロパティを識別しています。

```
@DynamoDBTable(tableName="Reply")
public class Reply {
    private Integer id;
    private String replyDateTime;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    @DynamoDBRangeKey(attributeName="ReplyDateTime")
    public String getReplyDateTime() { return replyDateTime; }
    public void setReplyDateTime(String replyDateTime) { this.replyDateTime =
 replyDateTime; }

   // Additional properties go here.
}
```

DynamoDBTable

DynamoDB 内でターゲットテーブルを識別します。例えば、次の Java コードでは Developer クラスを定義して、DynamoDB の People テーブルにマッピングしています。

```
@DynamoDBTable(tableName="People")
public class Developer { ...}
```

@DynamoDBTable 注釈は継承できます。Developer クラスから継承された新しいクラスも、People テーブルにマッピングされます。たとえば、Lead クラスから継承された Developer クラスを作成したとします。Developer クラスを People テーブルにマッピングしたことで、Lead クラスオブジェクトも同じテーブルに格納されます。

@DynamoDBTable もオーバーライドできます。デフォルトで Developer クラスから継承された新しいクラスは、同じ People テーブルにマッピングされます。ただし、このデフォルトのマッピングはオーバーライドできます。たとえば、Developer クラスから継承したクラスを作成した場合に

は、次の Java サンプルコードに示すように、@DynamoDBTable 注釈を追加することで、別のテーブルに明示的にマッピングできます。

```
@DynamoDBTable(tableName="Managers")
public class Manager extends Developer { ...}
```

DynamoDBTypeConverted

カスタム型コンバーターの使用時にプロパティをマークする注釈を使用できます。DynamoDBTypeConverter に追加のプロパティを渡すために、ユーザーが定義した注釈でコメントすることもできます。

DynamoDBTypeConverter インターフェイスでは、独自の任意データ型を、DynamoDB でネイティブにサポートされているデータ型にマッピングすることができます。詳細については、「DynamoDB での任意のデータのマッピング」を参照してください。

DynamoDBTyped

標準属性タイプの追加を上書きする注釈を使用できます。そのタイプでデフォルト属性のバインドを適用する場合、スタンダードの種類には注釈を必要としません。

DynamoDBVersionAttribute

オプティミスティックロックのバージョン番号を格納するためのクラスプロパティを識別します。DynamoDBMapper は、新しい項目を保存するときにこのプロパティにバージョン番号を割り当てます。バージョン番号は項目を更新するたびに増えていきます。サポートされているのは番号によるスカラー型だけです。 のデータ型の詳細については、「」を参照してくださいデータ型 バージョニングの詳細については、「DynamoDB およびバージョン番号を使用した楽観的ロック」を参照してください。

DynamoDBMapper のオプションの構成設定

DynamoDBMapper のインスタンスを作成すると、そのインスタンスには、特定のデフォルトの動作があります。DynamoDBMapperConfig クラスを使用して、このデフォルトの動作をオーバーライドできます。

次のコードスニペットでは、カスタム設定を使用して DynamoDBMapper を作成します。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapperConfig mapperConfig = DynamoDBMapperConfig.builder()
        .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
```

```
        .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT)
        .withTableNameOverride(null)

  .withPaginationLoadingStrategy(DynamoDBMapperConfig.PaginationLoadingStrategy.EAGER_LOADING)
     .build();

DynamoDBMapper mapper = new DynamoDBMapper(client, mapperConfig);
```

詳細については、AWS SDK for Java API リファレンスの「DynamoDBMapperConfig」を参照してください。

DynamoDBMapperConfig のインスタンスに対して、次の引数を指定できます。

- DynamoDBMapperConfig.ConsistentReads 列挙値:
  - EVENTUAL – マッパーインスタンスは、結果整合性のある読み込みリクエストを使用します。
  - CONSISTENT – マッパーインスタンスは、強力な整合性のある読み取リクエストを使用します。このオプションの設定は、load、query、または scan オペレーションに対して使用できます。強力な整合性のある読み込みは、パフォーマンスと請求に影響を与えます。詳細については、DynamoDB の製品の詳細ページを参照してください。

  マッパーインスタンスに読み込み整合性の設定を指定しない場合、デフォルトは EVENTUAL になります。

  > **Note**
  >
  > この値は、DynamoDBMapper の query、querypage、load、batch load の各オペレーションに適用されます。

- DynamoDBMapperConfig.PaginationLoadingStrategy 列挙値 – query または scan からの結果など、ページ分割されたデータのリストをマッパーインスタンスが処理する方法を制御します。
  - LAZY_LOADING – マッパーインスタンスは、可能な場合にデータをロードし、ロードされたすべての結果をメモリに保持します。
  - EAGER_LOADING – マッパーインスタンスは、リストが初期化されるとすぐにデータをロードします。
  - ITERATION_ONLY – リストからの読み込みに Iterator のみを使用することができます。反復中、リストは、前のすべての結果をクリアしてから、次のページをロードすることで、最大 1 ページのロードされた結果をメモリに維持するようにします。つまり、リストを反復できるのは

1 回だけです。メモリのオーバーヘッドを低減するために、大きい項目を処理するときにこの方法を使用することをお勧めします。

マッパーインスタンスにページ分割ロードの方法を指定しない場合、デフォルトは LAZY_LOADING になります。

- DynamoDBMapperConfig.SaveBehavior 列挙値 – 保存オペレーション中にマッパーインスタンスが属性を処理する方法を指定します。
  - UPDATE – 保存オペレーション中、すべてのモデル化された属性が更新され、モデル化されていない属性は影響を受けません。プリミティブな数値型 (byte、int、long) は 0 に設定されます。オブジェクト型は null に設定されます。
  - CLOBBER – 保存オペレーション中、モデル化されていない属性も含め、すべての属性をクリアし置き換えます。このオペレーションを行うには、項目を削除し、再作成します。また、バージョン付きフィールドの制約は無視されます。

  マッパーインスタンスに保存動作を指定しない場合、デフォルトは UPDATE になります。

  > **Note**
  >
  > DynamoDBMapper トランザクションオペレーションは DynamoDBMapperConfig.SaveBehavior 列挙をサポートしません。

- DynamoDBMapperConfig.TableNameOverride オブジェクト – クラスの DynamoDBTable アノテーションによって指定されたテーブル名を無視し、代わりに指定した別のテーブル名を使用するように、マッパーインスタンスに指示します。これは、実行時にデータを複数のテーブルに分割する場合に役立ちます。

必要に応じて、オペレーションごとに DynamoDBMapper のデフォルトの設定オブジェクトをオーバーライドできます。

DynamoDB およびバージョン番号を使用した楽観的ロック

楽観的ロックとは、更新 (または削除) しているクライアント側の項目と、Amazon DynamoDB の項目が確実に同じになるようにするための手段です。この方法を使用すると、データベースの書き込みは、他のユーザーの書き込みによって上書きされないように保護されます。逆の場合も同様に保護されます。

オプティミスティックロックを使用する場合、各項目には、バージョン番号として機能する属性があります。項目をテーブルから取り出すと、アプリケーションは、その項目のバージョン番号を記録し

ます。サーバー側のバージョン番号が変更されていない場合のみ、項目を更新できます。バージョンの不一致がある場合は、前に他のユーザーによってそのアイテムが変更されたことを意味します。アイテムの古いバージョンがあるため、更新の試行は失敗します。その場合は更新をやり直します。もう一度項目を取得して、更新してください。オプティミスティックロックでは、他のユーザーが行った変更を誤って上書きできないようにします。また、お客様が行った変更を他のユーザーが誤って変更することを防ぐこともできます。

独自の楽観的ロック戦略を実装することもできますが、AWS SDK for Java には `@DynamoDBVersionAttribute` アノテーションが用意されています。テーブルのマッピングクラスで、バージョン番号を保存する 1 つのプロパティを指定し、この注釈を使用してそのプロパティをマーキングします。オブジェクトを保存すると、DynamoDB テーブル内の対応する項目に、バージョン番号を格納するための属性が追加されます。`DynamoDBMapper` では、最初にオブジェクトを保存したときにバージョン番号が割り当てられ、項目を更新するたびにバージョン番号が自動的にインクリメントされます。更新または削除リクエストは、クライアント側オブジェクトのバージョン番号が、DynamoDB テーブルで対応する項目のバージョン番号に一致する場合のみ成功します。

> **Note**
>
> テーブルにまだ存在しない項目を保存すると、バージョンプロパティの値によって `DynamoDBMapper` が書き込みを構築する方法が決まります。
>
> - バージョンプロパティが `null` の場合、`DynamoDBMapper` はオペレーションを新しい項目の挿入として扱います。バージョン属性がまだ存在しないという条件を追加し、保存されたバージョンを 1 に設定します。保存は、同じキーを持つ項目が既に存在しない場合にのみ成功します。
> - バージョンプロパティが任意の値 (`0`、`1`、`5` など) に設定されている場合、`DynamoDBMapper` はオペレーションを既存の項目の更新として扱います。保存されたバージョンがその値に等しいという条件を追加し、増分します。項目がまだ存在しない場合、条件は失敗し、保存で `ConditionalCheckFailedException` がスローされます。
>
> 新しい項目を挿入するには、バージョンプロパティを `null` のままにします。既存の項目を更新するには、バージョンプロパティを最後に読み取った値に設定します。挿入と更新を区別するためにバージョン値を自分で検査する必要はありません。`DynamoDBMapper` は null-versus-set の区別を使用して判断します。`DynamoDBMapper` が未設定のバージョンを検出

できるように、バージョンプロパティは null 可能なタイプ (long ではなく Long など) である必要があります。

`ConditionalCheckFailedException`以下の場合、 はスローされます。

- オプティミスティックロックを `@DynamoDBVersionAttribute` と共に使用しており、サーバーのバージョンの値が、クライアント側の値とは異なる場合。
- `DynamoDBMapper` で `DynamoDBSaveExpression` を使用してデータを保存するときに、独自の条件付き制約を指定し、それらの制約に障害が生じた場合。

> **Note**
>
> - DynamoDB のグローバルテーブルはグローバルテーブルでは、同時更新間の「最新書き込み」の照合を行います。グローバルテーブルを使用する場合、最後に書き込まれたポリシーが採用されます。したがってこの場合、ロック戦略は想定通りに機能しません。
> - `DynamoDBMapper` トランザクションの書き込みオペレーションは、同じオブジェクトでの `@DynamoDBVersionAttribute` 注釈と条件式をサポートしていません。トランザクション書き込み内のオブジェクトに `@DynamoDBVersionAttribute` の注釈が付けられ、条件式も含まれている場合、SdkClientException がスローされます。

たとえば次の Java コードでは、複数のプロパティを持つ `CatalogItem` クラスを定義しています。`Version` プロパティは `@DynamoDBVersionAttribute` 注釈でタグ付けされています。

Example

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;
    private Long version;

    @DynamoDBHashKey(attributeName="Id")
```

```
    public Integer getId() { return id; }
    public void setId(Integer Id) { this.id = Id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN;}

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors =
 bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp;}
    public void setSomeProp(String someProp) {this.someProp = someProp;}

    @DynamoDBVersionAttribute
    public Long getVersion() { return version; }
    public void setVersion(Long version) { this.version = version;}
}
```

@DynamoDBVersionAttribute 注釈は、Long や Integer などの、null が許容された型を提供するプリミティブラッパークラスによって得られた、null が許容された型に適用できます。

オプティミスティックロックは、次の DynamoDBMapper メソッドに対して次のような影響があります。

- save – DynamoDBMapper は、新しい項目に対して初期バージョン番号 1 を割り当てます。項目を取得し、その項目の 1 つ以上のプロパティを更新して変更を保存する場合には、クライアント側とサーバー側のバージョン番号が一致する場合のみ、保存が成功します。DynamoDBMapper によってバージョン番号が自動的にインクリメントされます。
- delete – delete メソッドは 1 つのオブジェクトをパラメータとして指定し、項目を削除する前に DynamoDBMapper によるバージョンチェックが実行されます。バージョンチェックは、リクエスト内で DynamoDBMapperConfig.SaveBehavior.CLOBBER を指定して無効にすることができます。

DynamoDBMapper 内のオプティミスティックロックの内部実装では、DynamoDB の条件付き更新と条件付き削除機能が使用されます。

- transactionWrite —
  - Put – DynamoDBMapper は、新しい項目に対して初期バージョン番号 1 を割り当てます。項目を取得し、その項目の 1 つ以上のプロパティを更新して変更を保存する場合には、クライアント側とサーバー側のバージョン番号が一致する場合のみ、put オペレーションが成功します。DynamoDBMapper によってバージョン番号が自動的にインクリメントされます。
  - Update – DynamoDBMapper は、新しい項目に対して初期バージョン番号 1 を割り当てます。項目を取得し、その項目の 1 つ以上のプロパティを更新して変更を保存する場合には、クライアント側とサーバー側のバージョン番号が一致する場合のみ、更新オペレーションが成功します。DynamoDBMapper によってバージョン番号が自動的にインクリメントされます。
  - Delete – DynamoDBMapper は、項目を削除する前にバージョンチェックを実行します。削除オペレーションは、クライアント側とサーバー側のバージョン番号が一致する場合にのみ成功します。
  - ConditionCheck – @DynamoDBVersionAttribute アノテーションは、ConditionCheck オペレーションではサポートされていません。ConditionCheck 項目に @DynamoDBVersionAttribute の注釈が付いている場合、SdkClientException がスローされます。

楽観的ロックの無効化

オプティミスティックロックを無効にするには、DynamoDBMapperConfig.SaveBehavior 列挙値を UPDATE から CLOBBER に変更します。その場合は、バージョンチェックを省略する DynamoDBMapperConfig インスタンスを作成して、そのインスタンスをすべてのリクエストで使用します。DynamoDBMapperConfig.SaveBehavior とその他のオプションの DynamoDBMapper パラメータの詳細については、「DynamoDBMapper のオプションの構成設定」を参照してください。

ロック動作は、特定のオペレーションだけに設定することもできます。たとえば次の Java コードスニペットではDynamoDBMapper を使用してカタログ項目を保存しています。オプションの DynamoDBMapperConfig.SaveBehavior パラメータを DynamoDBMapperConfig メソッドに追加することで、save を指定しています。

> **Note**
>
> transactionWrite メソッドは、DynamoDBMapperConfig.SaveBehavior 構成をサポートしていません。transactionWrite のオプティミスティックロックの無効化はサポートされていません。

Example

```
DynamoDBMapper mapper = new DynamoDBMapper(client);

// Load a catalog item.
CatalogItem item = mapper.load(CatalogItem.class, 101);
item.setTitle("This is a new title for the item");
...
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

DynamoDB での任意のデータのマッピング

サポートされている Java 型 (「DynamoDBMapper for Java でサポートされるデータ型」を参照) に加えて、Amazon DynamoDB 型に直接マッピングされない、アプリケーション内の型を使用できます。これらの型をマッピングするには、複合型を DynamoDB 対応の型に (またその反対に) 変換する処理を実装し、@DynamoDBTypeConverted アノテーションを使用して複合型のアクセサーメソッドに注釈付けを行う必要があります。コンバーターコードは、オブジェクトが保存またはロードされるときにデータを変換します。これは、複合型を消費するすべてのオペレーションでも使用されます。クエリおよびスキャンオペレーションの中でデータを比較する場合、その比較は DynamoDB に格納されているデータに対して行われることにご注意ください。

たとえば、プロパティ CatalogItem (Dimension 型) を定義する、次の DimensionType クラスがあるとします。このプロパティには、項目の寸法として高さ、幅、厚さが格納されています。これらの項目のディメンションを DynamoDB の中に文字列 (8.5x11x.05 など) として格納しようとしている場合を考えます。次の例は、DimensionType オブジェクトを文字列に変換し、文字列を DimensionType に変換するコンバーターコードを示しています。

> **Note**
>
> このコード例では、アカウントの DynamoDB に対し、DynamoDB でのコード例用のテーブルの作成とデータのロード セクションの手順に従ってデータが既にロードされていることを前提としています。
> 以下の例を実行するための詳しい手順については、「Java コードの例」を参照してください。

Example

```
public class DynamoDBMapperExample {

    static AmazonDynamoDB client;

    public static void main(String[] args) throws IOException {

        // Set the AWS region you want to access.
        Regions usWest2 = Regions.US_WEST_2;
        client = AmazonDynamoDBClientBuilder.standard().withRegion(usWest2).build();

        DimensionType dimType = new DimensionType();
        dimType.setHeight("8.00");
        dimType.setLength("11.0");
        dimType.setThickness("1.0");

        Book book = new Book();
        book.setId(502);
        book.setTitle("Book 502");
        book.setISBN("555-5555555555");
        book.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));
        book.setDimensions(dimType);

        DynamoDBMapper mapper = new DynamoDBMapper(client);
        mapper.save(book);

        Book bookRetrieved = mapper.load(Book.class, 502);
        System.out.println("Book info: " + "\n" + bookRetrieved);

        bookRetrieved.getDimensions().setHeight("9.0");
        bookRetrieved.getDimensions().setLength("12.0");
        bookRetrieved.getDimensions().setThickness("2.0");
```

```
        mapper.save(bookRetrieved);

        bookRetrieved = mapper.load(Book.class, 502);
        System.out.println("Updated book info: " + "\n" + bookRetrieved);
    }

    @DynamoDBTable(tableName = "ProductCatalog")
    public static class Book {
        private int id;
        private String title;
        private String ISBN;
        private Set<String> bookAuthors;
        private DimensionType dimensionType;

        // Partition key
        @DynamoDBHashKey(attributeName = "Id")
        public int getId() {
            return id;
        }

        public void setId(int id) {
            this.id = id;
        }

        @DynamoDBAttribute(attributeName = "Title")
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        @DynamoDBAttribute(attributeName = "ISBN")
        public String getISBN() {
            return ISBN;
        }

        public void setISBN(String ISBN) {
            this.ISBN = ISBN;
        }

        @DynamoDBAttribute(attributeName = "Authors")
```

```
        public Set<String> getBookAuthors() {
            return bookAuthors;
        }

        public void setBookAuthors(Set<String> bookAuthors) {
            this.bookAuthors = bookAuthors;
        }

        @DynamoDBTypeConverted(converter = DimensionTypeConverter.class)
        @DynamoDBAttribute(attributeName = "Dimensions")
        public DimensionType getDimensions() {
            return dimensionType;
        }

        @DynamoDBAttribute(attributeName = "Dimensions")
        public void setDimensions(DimensionType dimensionType) {
            this.dimensionType = dimensionType;
        }

        @Override
        public String toString() {
            return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ",
dimensionType= "
                    + dimensionType.getHeight() + " X " + dimensionType.getLength() + "
X "
                    + dimensionType.getThickness()
                    + ", Id=" + id + ", Title=" + title + "]";
        }
    }

    static public class DimensionType {

        private String length;
        private String height;
        private String thickness;

        public String getLength() {
            return length;
        }

        public void setLength(String length) {
            this.length = length;
        }
```

```
        public String getHeight() {
            return height;
        }

        public void setHeight(String height) {
            this.height = height;
        }

        public String getThickness() {
            return thickness;
        }

        public void setThickness(String thickness) {
            this.thickness = thickness;
        }
    }

    // Converts the complex type DimensionType to a string and vice-versa.
    static public class DimensionTypeConverter implements DynamoDBTypeConverter<String,
DimensionType> {

        @Override
        public String convert(DimensionType object) {
            DimensionType itemDimensions = (DimensionType) object;
            String dimension = null;
            try {
                if (itemDimensions != null) {
                    dimension = String.format("%s x %s x %s",
itemDimensions.getLength(), itemDimensions.getHeight(),
                            itemDimensions.getThickness());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            return dimension;
        }

        @Override
        public DimensionType unconvert(String s) {

            DimensionType itemDimension = new DimensionType();
            try {
                if (s != null && s.length() != 0) {
                    String[] data = s.split("x");
```

```
                    itemDimension.setLength(data[0].trim());
                    itemDimension.setHeight(data[1].trim());
                    itemDimension.setThickness(data[2].trim());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }

            return itemDimension;
        }
    }
}
```

DynamoDBMapper の例

AWS SDK for Java には DynamoDBMapper クラスが用意されているため、クライアント側のクラスを DynamoDB テーブルにマッピングできます。DynamoDBMapper を使用するには、DynamoDB テーブル内の項目と、それが対応するコード内のオブジェクトインスタンスの間での関係性を定義します。DynamoDBMapper クラスを使用すると、項目に対する作成、読み込み、更新、削除 (CRUD) の各オペレーションの実行、およびテーブルに対するクエリやスキャンを行うことができます。

DynamoDBMapper の使用方法の詳細については、「AWS SDK for Java 1.x デベロッパーガイド」の「AWS SDK for Java を使用する DynamoDB の例」を参照してください。

Java 2.x: DynamoDB 拡張クライアント

DynamoDB 拡張クライアントは、AWS SDK for Java バージョン 2 (v2) の一部である高レベルのライブラリです。クライアント側のクラスを DynamoDB テーブルにマッピングする簡単な方法を提供します。テーブルおよび対応するモデルクラスの間の関係をコードで定義します。関係を定義したら、DynamoDB のテーブルまたは項目に対して、さまざまな作成、読み取り、更新、または削除 (CRUD) オペレーションを直感的に実行できます。

DynamoDB で拡張クライアントを使用する方法の詳細については、「AWS SDK for Java 2.x での DynamoDB 拡張クライアントの使用」を参照してください。

DynamoDB での .NET ドキュメントの操作

AWS SDK for .NET では、低レベル Amazon DynamoDB オペレーションの一部をまとめるドキュメントモデルクラスを使用して、コーディングをさらに簡略化することができます。ドキュメントモデルのプライマリクラスは Table と Document です。Table クラスでは、PutItem、GetItem、DeleteItem などのデータオペレーション方法を使用できます。Query

および Scan メソッドも使用できます。Document クラスは、テーブル内の単一の項目を表します。

前述のドキュメントモデルクラスは、Amazon.DynamoDBv2.DocumentModel 名前空間で使用できます。

> **Note**
>
> ドキュメントモデルクラスは、テーブルの作成、更新、削除に使用することはできません。ただし、ドキュメントモデルは、ほとんどの一般的なデータオペレーションをサポートしています。

トピック

- サポートされているデータ型

サポートされているデータ型

ドキュメントモデルは、プリミティブ .NET データ型とコレクションデータ型のセットをサポートします。このモデルでは、次のプリミティブデータ型がサポートされています。

- bool
- byte
- char
- DateTime
- decimal
- double
- float
- Guid
- Int16
- Int32
- Int64
- SByte
- string

- UInt16
- UInt32
- UInt64

次の表に、前述の .NET 型が DynamoDB の型にどのようにマッピングされるのかをまとめています。

.NET プリミティブ型	DynamoDB 型
すべての数値型	N (数値型)
すべての文字列型	S (文字列型)
MemoryStream、byte[]	B (バイナリ型)
ブール	N (数値型)。0 は false、1 は true を表します。
DateTime	S (文字列型)。DateTime の値は、ISO-8601 形式の文字列として格納されます。
Guid	S (文字列型)。
コレクション型 (リスト、ハッシュセット、配列)	BS (バイナリセット) 型、SS (文字列セット) 型、NS (数値セット) 型

AWS SDK for .NET は、DynamoDB のブール型、null 型、リスト型、およびマップ型を .NET ドキュメントモデル API にマッピングするための型を定義します。

- ブール型には DynamoDBBool 使用します。
- null 型には DynamoDBNull を使用します。
- リスト型には DynamoDBList を使用します。
- マップ型には Document を使用します。

> **Note**
>
> - 空のバイナリ値がサポートされています。

- 空の文字列値の読み取りがサポートされています。空の文字列属性値は、DynamoDB への書き込み中に、文字列セット型の属性値内でサポートされます。文字列型の空の文字列属性値と、リスト型またはマップ型に含まれる空の文字列値が書き込みリクエストから削除されます。

.NET オブジェクト永続性モデルと DynamoDB の使用

AWS SDK for .NET には、クライアント側クラスを Amazon DynamoDB テーブルにマッピングできるオブジェクト永続性モデルを有効にしています。各オブジェクトインスタンスが、対応するテーブルの項目にマッピングされます。クライアント側オブジェクトをテーブルに保存するために、オブジェクト永続性モデルでは、DynamoDB のエントリポイントとして `DynamoDBContext` クラスを使用できます。このクラスでは、DynamoDB に接続してテーブルにアクセスし、各種の CRUD オペレーションやクエリを実行することができます。

オブジェクト永続性モデルには、クライアント側クラスをテーブルにマッピングし、プロパティ/フィールドを属性にマッピングする、属性のセットが用意されています。

> **Note**
>
> オブジェクト永続性モデルには、テーブルを作成、更新、または削除するための API はありません。データオペレーションだけが可能になっています。テーブルの作成、更新、削除が可能なのは AWS SDK for .NET の低レベル API だけです。

次の例は、オブジェクト永続性モデルの動作を示しています。`ProductCatalog` テーブルから開始します。ここでは `Id` がプライマリキーになっています。

```
ProductCatalog(Id, ...)
```

`Book`、`Title`、`ISBN` プロパティを持つ `Authors` クラスがあるとします。次の C# サンプルコードに示すように、オブジェクト永続性モデルで定義された属性を追加することで、`Book` クラスを `ProductCatalog` テーブルにマッピングできます。

Example

```
[DynamoDBTable("ProductCatalog")]
  public class Book
  {
```

```
    [DynamoDBHashKey]
    public int Id { get; set; }

    public string Title { get; set; }
    public int ISBN { get; set; }

    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }

    [DynamoDBIgnore]
    public string CoverPage { get; set; }
  }
```

前述の例では、DynamoDBTable 属性によって、Book クラスが ProductCatalog テーブルにマッピングされています。

オブジェクト永続性モデルでは、クラスプロパティとテーブル属性との間で、明示的なマッピングとデフォルトのマッピングの両方がサポートされています。

- 明示的なマッピング - プライマリキーにプロパティをマッピングするには、オブジェクト永続性モデル属性の DynamoDBHashKey および DynamoDBRangeKey を使用する必要があります。さらに、非プライマリキー属性については、クラス内のプロパティ名と、マッピング先の対応するテーブル属性が同じでない場合は、DynamoDBProperty 属性を明示的に追加してマッピングを定義する必要があります。

  前述の例では、Id プロパティが同じ名前のプライマリキーにマッピングされ、BookAuthors プロパティが Authors テーブル内の ProductCatalog 属性にマッピングされています。
- デフォルトのマッピング – デフォルトでは、オブジェクト永続性モデルによって、クラスプロパティがテーブル内の同じ名前の属性にマッピングされます。

  前述の例では、Title および ISBN プロパティが、ProductCatalog テーブル内の同じ名前の属性にマッピングされています。

すべてのクラスプロパティをマッピングする必要はありません。これらのプロパティを特定するには、DynamoDBIgnore 属性を追加します。Book インスタンスをテーブルに保存する場合、DynamoDBContext には CoverPage プロパティは含まれません。このプロパティは、書籍インスタンスを取り出す場合にも返されません。

int や string など、.NET プリミティブ型のプロパティをマッピングできます。また、任意データをいずれかの DynamoDB 型にマッピングする適切なコンバーターがある限り、任意のデータ型をマッピ

ングすることもできます。任意の型のマッピングについては、「AWS SDK for .NET のオブジェクト永続性モデルを使用した、DynamoDB での任意データのマッピング」を参照してください。

オブジェクト永続性モデルでは、オプティミスティックロックがサポートされています。それによって、更新オペレーションで、更新する項目の最新のコピーを確実に使用することができます。詳細については、「DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック」を参照してください。

詳細については、以下のトピックを参照してください。

トピック

- サポートされているデータ型
- .NET オブジェクト永続性モデルからの DynamoDB 属性
- .NET オブジェクト永続性モデルからの DynamoDBContext クラス
- DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック
- AWS SDK for .NET のオブジェクト永続性モデルを使用した、DynamoDB での任意データのマッピング

サポートされているデータ型

オブジェクト永続性モデルでは、プリミティブな .NET データ型、コレクション、および任意のデータ型のセットがサポートされています。このモデルでは、次のプリミティブデータ型がサポートされています。

- `bool`
- `byte`
- `char`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Int16`
- `Int32`
- `Int64`
- `SByte`

- `string`
- `UInt16`
- `UInt32`
- `UInt64`

オブジェクト永続性モデルでは .NET コレクション型もサポートされています。`DynamoDBContext` は具体的なコレクション型、およびシンプルな Plain Old CLR Objects (POCOs) を変換できます。

次の表に、前述の .NET 型が DynamoDB の型にどのようにマッピングされるかを示します。

.NET プリミティブ型	DynamoDB 型
すべての数値型	N (数値型)
すべての文字列型	S (文字列型)
MemoryStream、byte[]	B (バイナリ型)
ブール	N (数値型)。0 は false、1 は true を表します。
コレクション型	BS (バイナリセット) 型、SS (文字列セット) 型、NS (数値セット) 型
DateTime	S (文字列型)。`DateTime` の値は、ISO-8601 形式の文字列として格納されます。

オブジェクト永続性モデルでは、任意のデータ型もサポートされています。ただし、複合型を DynamoDB 型にマッピングする場合には、変換用のコードを用意する必要があります。

> **Note**
>
> - 空のバイナリ値がサポートされています。
> - 空の文字列値の読み取りがサポートされています。空の文字列属性値は、DynamoDB への書き込み中に、文字列セット型の属性値内でサポートされます。文字列型の空の文字列属性値と、リスト型またはマップ型に含まれる空の文字列値が書き込みリクエストから削除されます。

.NET オブジェクト永続性モデルからの DynamoDB 属性

このセクションでは、クラスとプロパティを DynamoDB のテーブルや属性にマッピングできるように、オブジェクト永続性モデルで提供されている属性を説明します。

> **Note**
>
> 次の属性では、DynamoDBTable と DynamoDBHashKey だけが必須です。

DynamoDBGlobalSecondaryIndexHashKey

グローバルセカンダリインデックスのパーティションキーに、クラスプロパティをマッピングします。この属性は、グローバルセカンダリインデックスを Query する必要がある場合に使用します。

DynamoDBGlobalSecondaryIndexRangeKey

グローバルセカンダリインデックスのソートキーにクラスプロパティをマッピングします。この属性は、グローバルセカンダリインデックスを Query し、インデックスソートキーを使用して結果を絞り込む必要がある場合に使用します。

DynamoDBHashKey

テーブルのプライマリキーのパーティションキーにクラスプロパティをマッピングします。プライマリキーの属性をコレクション型にすることはできません。

次の C# サンプルコードでは、Book クラスを ProductCatalog テーブルに、Id プロパティをテーブルのプライマリキーのパーティションキーにマッピングしています。

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey]
    public int Id { get; set; }

    // Additional properties go here.
}
```

DynamoDBIgnore

関連するプロパティを無視するように指示します。クラスプロパティを保存しない場合は、この属性を追加することで、テーブルにオブジェクトを保存するときにこのプロパティを含めないように DynamoDBContext に指示できます。

DynamoDBLocalSecondaryIndexRangeKey

ローカルセカンダリインデックスのソートキーにクラスプロパティをマッピングします。この属性は、ローカルセカンダリインデックスを Query し、インデックスソートキーを使用して結果を絞り込む必要がある場合に使用します。

DynamoDBProperty

テーブルの属性にクラスプロパティをマッピングします。クラスプロパティを同じ名前のテーブル属性にマッピングする場合は、この属性を指定する必要はありません。ただし名前が異なる場合は、このタグを使用してマッピングを指定できます。次の C# ステートメントでは、DynamoDBProperty によって、BookAuthors プロパティがテーブル内の Authors 属性にマッピングされています。

```
[DynamoDBProperty("Authors")]
public List<string> BookAuthors { get; set; }
```

DynamoDBContext はこのマッピング情報を使用して、対応するテーブルにオブジェクトデータを保存するときに Authors 属性を作成します。

DynamoDBRenamable

クラスプロパティの代替名を指定します。これは、テーブル属性がクラスプロパティの名前と異なる DynamoDB テーブルに、任意データをマッピングするためのカスタムコンバーターを記述する場合に役立ちます。

DynamoDBRangeKey

テーブルのプライマリキーのソートキーにクラスプロパティをマッピングします。テーブルに複合プライマリキー (パーティションキーおよびソートキー) がある場合は、クラスマッピングで、DynamoDBHashKey と DynamoDBRangeKey の両方の属性を指定する必要があります。

たとえば、サンプルテーブルの Reply には、Id パーティションキーと Replenishment ソートキーで構成されたプライマリキーがあります。次の C# サンプルコードでは、Reply クラスを Reply テーブルにマッピングします。クラス定義では、プロパティのうち 2 つがプライマリキーにマッピングされることも示しています。

```
[DynamoDBTable("Reply")]
public class Reply
{
   [DynamoDBHashKey]
   public int ThreadId { get; set; }
   [DynamoDBRangeKey]
   public string Replenishment { get; set; }

   // Additional properties go here.
}
```

DynamoDBTable

クラスがマッピングされる、DynamoDB 内のターゲットテーブルを識別します。例えば次の C# サンプルコードでは、Developer クラスを DynamoDB の People テーブルにマッピングしています。

```
[DynamoDBTable("People")]
public class Developer { ...}
```

この属性は、継承またはオーバーライドすることができます。

- DynamoDBTable 属性は継承できます。前述の例では、Lead クラスから継承された新しいクラス Developer を追加すると、People テーブルにもマッピングされます。Developer テーブルに Lead と People の両方のオブジェクトが格納されます。
- DynamoDBTable 属性もオーバーライドできます。次の C# サンプルコードでは、Manager クラスは、Developer クラスから継承されます。ただし、DynamoDBTable 属性を明示的に追加すると、クラスは別のテーブル (Managers) にマップされます。

  ```
  [DynamoDBTable("Managers")]
  public class Manager : Developer { ...}
  ```

次の C# の例に示すように、オプションのパラメータ LowerCamelCaseProperties を追加すると、オブジェクトをテーブルに格納する際にプロパティ名の先頭文字を小文字にするよう、DynamoDB にリクエストを送ることができます。

```
[DynamoDBTable("People", LowerCamelCaseProperties=true)]
public class Developer
{
```

```
    string DeveloperName;
    ...
}
```

Developer クラスのインスタンスを保存する場合、DynamoDBContext では DeveloperName プロパティが developerName として保存されます。

DynamoDBVersion

項目のバージョン番号を格納するクラスプロパティを識別します。バージョニングの詳細については、「DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック」を参照してください。

.NET オブジェクト永続性モデルからの DynamoDBContext クラス

DynamoDBContext クラスは Amazon DynamoDB データベースのエントリポイントです。このクラスから DynamoDB に接続して、各種のテーブル内のデータにアクセスし、さまざまな CRUD オペレーションとクエリを実行することができます。DynamoDBContext クラスでは次のメソッドを使用できます。

トピック

- CreateMultiTableBatchGet
- CreateMultiTableBatchWrite
- CreateBatchGet
- CreateBatchWrite
- Delete
- Dispose
- ExecuteBatchGet
- ExecuteBatchWrite
- FromDocument
- FromQuery
- FromScan
- GetTargetTable
- Load
- Query
- 保存

- Scan
- ToDocument
- DynamoDBContext でのオプションパラメータの指定

CreateMultiTableBatchGet

複数の個々の MultiTableBatchGet オブジェクトで構成される BatchGet オブジェクトを作成します。これらの各 BatchGet オブジェクトは、単一の DynamoDB テーブルから項目を取り出す場合に使用します。

1 つ以上のテーブルから項目を取得するには、ExecuteBatchGet メソッドを使用し、MultiTableBatchGet オブジェクトをパラメータとして渡します。

CreateMultiTableBatchWrite

複数の個々の MultiTableBatchWrite オブジェクトで構成される BatchWrite オブジェクトを作成します。これらの各 BatchWrite オブジェクトは、単一の DynamoDB テーブルに項目を書き込んだり、それを削除したりするために使用します。

テーブルに書き込むには、ExecuteBatchWrite メソッドを使用し、MultiTableBatchWrite オブジェクトをパラメータとして渡します。

CreateBatchGet

テーブルから複数の項目を取り出すために使用できる BatchGet オブジェクトを作成します。

CreateBatchWrite

テーブルに複数の項目を入力する、またはテーブルから複数の項目を削除するために使用できる BatchWrite オブジェクトを作成します。

Delete

テーブルから項目を削除します。このメソッドでは、削除する項目のプライマリキーが必要になります。プライマリキーの値、またはこのメソッドのパラメータとしてプライマリキーの値を使用するクライアント側オブジェクトを入力できます。

- クライアント側オブジェクトをパラメータとして指定し、オプティミスティックロックを有効にすると、クライアント側とサーバー側のオブジェクトのバージョンが一致する場合のみ、削除が成功します。

- プライマリキーの値だけをパラメータとして指定すると、オプティミスティックロックを有効にしているかどうかにかかわらず、削除が成功します。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、`DeleteAsync` メソッドを使用します。

Dispose

すべてのマネージドリソースとアンマネージドリソースを破棄します。

ExecuteBatchGet

1 つ以上のテーブルからデータを読み込みます。`BatchGet` 内のすべての `MultiTableBatchGet` オブジェクトを処理します。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、`ExecuteBatchGetAsync` メソッドを使用します。

ExecuteBatchWrite

1 つ以上のテーブルにデータを書き込むまたは削除します。`BatchWrite` 内のすべての `MultiTableBatchWrite` オブジェクトを処理します。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、`ExecuteBatchWriteAsync` メソッドを使用します。

FromDocument

`Document` のインスタンスと仮定すると、`FromDocument` メソッドは、クライアント側のクラスのインスタンスを返します。

これは、オブジェクト永続性モデルと合わせてドキュメントモデルクラスを使用してデータオペレーションを行う場合に役立ちます。AWS SDK for .NET で使用されるドキュメントモデルクラスの詳細については、「DynamoDB での .NET ドキュメントの操作」を参照してください。

Document という名前の doc オブジェクトがあり、Forum 項目の表現を含んでいるとします。(このオブジェクトの構成方法については、下の ToDocument メソッドの説明を参照してください)。次の C# サンプルコードに示すように、FromDocument を使用して Forum から Document 項目を取得することができます。

Example

```
forum101 = context.FromDocument<Forum>(101);
```

> **Note**
>
> Document オブジェクトで IEnumerable インターフェイスを実装している場合、FromDocuments メソッドを使用できます。これにより、Document のすべてのクラスインスタンスを反復的に処理できます。

FromQuery

QueryOperationConfig オブジェクトに定義されたクエリパラメータを使用して、Query オペレーションを実行します。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、FromQueryAsync メソッドを使用します。

FromScan

ScanOperationConfig オブジェクトに定義されたスキャンパラメータを使用して、Scan オペレーションを実行します。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、FromScanAsync メソッドを使用します。

GetTargetTable

指定した型のターゲットテーブルを取り出します。これは、任意データを DynamoDB テーブルにマッピングするためのカスタムコンバーターを記述していて、カスタムデータ型に関連付けられているテーブルを特定する必要がある場合に役立ちます。

Load

テーブルから項目を取り出します。このメソッドでは、取り出す項目のプライマリキーだけが必要になります。

DynamoDB のデフォルトでは、結果整合性のある値が含まれる項目が返されます。結果整合性モデルの詳細については、「DynamoDB の読み取り整合性」を参照してください。

Load または LoadAsync メソッドは GetItem オペレーションを呼び出します。このオペレーションでは、テーブルのプライマリキーを指定する必要があります。GetItem は IndexName パラメータを無視するため、インデックスのパーティションまたはソートキーを使用して項目を読み込むことはできません。そのため、テーブルのプライマリキーを使用して項目を読み込む必要があります。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、LoadAsync メソッドを使用します。LoadAsync メソッドを使用して DynamoDB テーブルで高レベルの CRUD オペレーションを実行する例については、次の例を参照してください。

```
/// <summary>
/// Shows how to perform high-level CRUD operations on an Amazon DynamoDB
/// table.
/// </summary>
public class HighLevelItemCrud
{
    public static async Task Main()
    {
```

```
            var client = new AmazonDynamoDBClient();
            DynamoDBContext context = new DynamoDBContext(client);
            await PerformCRUDOperations(context);
        }

        public static async Task PerformCRUDOperations(IDynamoDBContext context)
        {
            int bookId = 1001; // Some unique value.
            Book myBook = new Book
            {
                Id = bookId,
                Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
                Isbn = "111-1111111001",
                BookAuthors = new List<string> { "Author 1", "Author 2" },
            };

            // Save the book to the ProductCatalog table.
            await context.SaveAsync(myBook);

            // Retrieve the book from the ProductCatalog table.
            Book bookRetrieved = await context.LoadAsync<Book>(bookId);

            // Update some properties.
            bookRetrieved.Isbn = "222-2222221001";

            // Update existing authors list with the following values.
            bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x" };
            await context.SaveAsync(bookRetrieved);

            // Retrieve the updated book. This time, add the optional
            // ConsistentRead parameter using DynamoDBContextConfig object.
            await context.LoadAsync<Book>(bookId, new DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            // Delete the book.
            await context.DeleteAsync<Book>(bookId);

            // Try to retrieve deleted book. It should return null.
            Book deletedBook = await context.LoadAsync<Book>(bookId, new
DynamoDBContextConfig
            {
                ConsistentRead = true,
```

```
            });

            if (deletedBook == null)
            {
                Console.WriteLine("Book is deleted");
            }
        }
    }
```

Query

指定したクエリパラメータに基づいてテーブルのクエリが実行されます。

複合プライマリキー (パーティションキーおよびソートキー) が存在する場合にのみ、テーブルにクエリを実行できます。クエリを実行する場合は、パーティションキーと、ソートキーに適用される条件を指定する必要があります。

ここで、クライアント側の Reply クラスが、DynamoDB の Reply テーブルにマッピングされている場合を考えてみます。次の C# サンプルコードでは、Reply テーブルのクエリを実行し、過去 15 日間に投稿されたフォーラムスレッドの返信を検索しています。Reply テーブルには、Id パーティションキーと ReplyDateTime ソートキーを持つプライマリキーがあります。

Example

```
DynamoDBContext context = new DynamoDBContext(client);

string replyId = "DynamoDB#DynamoDB Thread 1"; //Partition key
DateTime twoWeeksAgoDate = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Date
 to compare.
IEnumerable<Reply> latestReplies = context.Query<Reply>(replyId,
 QueryOperator.GreaterThan, twoWeeksAgoDate);
```

これにより、Reply オブジェクトのコレクションが返されます。

Query メソッドでは、「遅延ロード」された IEnumerable コレクションが返ります。最初に結果が 1 ページのみ返され、必要に応じて、さらに次ページを要求するサービス呼び出しが行われます。一致する項目をすべて取得するには、IEnumerable を反復的に処理します。

テーブルにシンプルなプライマリキー (パーティションキー) がある場合は、Query メソッドを使用できません。代わりに Load メソッドを使用して、パーティションキーを入力して項目を取り出すことができます。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、QueryAsync メソッドを使用します。

保存

指定したオブジェクトがテーブルに保存されます。入力オブジェクトで指定されたプライマリキーがテーブル内に存在しない場合は、このメソッドによって新しい項目がテーブルに追加されます。プライマリキーが存在する場合は、このメソッドによって既存の項目が更新されます。

オプティミスティックロックを設定している場合には、クライアント側とサーバー側で項目のバージョンが一致する場合のみ、更新が正常に実行されます。詳細については、「DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック」を参照してください。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、SaveAsync メソッドを使用します。

Scan

テーブル全体のスキャンを実行します。

スキャン結果をフィルタリングするには、スキャン条件を指定します。この条件は、テーブル内の任意の属性に適用することができます。例えば、クライアント側の Book クラスが、DynamoDB の ProductCatalog テーブルにマッピングされているとします。次の C# コード例では、テーブルがスキャンされ、価格が 0 未満の書籍項目だけが返されています。

Example

```
IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(
                    new ScanCondition("Price", ScanOperator.LessThan, price),
                    new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
      );
```

Scan メソッドでは、「遅延ロード」された IEnumerable コレクションが返ります。最初に結果が 1 ページのみ返され、必要に応じて、さらに次ページを要求するサービス呼び出しが行われます。一致するすべての項目は、IEnumerable を反復的に処理するだけで取得できます。

パフォーマンス上の理由から、テーブルについてはスキャンを避け、クエリを行うようにしてください。

> **Note**
>
> このオペレーションをバックグラウンドで実行するには、ScanAsync メソッドを使用します。

ToDocument

クラスインスタンスから、Document ドキュメントモデルクラスのインスタンスが返されます。

これは、オブジェクト永続性モデルと合わせてドキュメントモデルクラスを使用してデータオペレーションを行う場合に役立ちます。AWS SDK for .NET で使用されるドキュメントモデルクラスの詳細については、「DynamoDB での .NET ドキュメントの操作」を参照してください。

クライアント側のクラスが Forum サンプルテーブルにマッピングされているとします。その場合は次の C# サンプルコードに示すように、DynamoDBContext を使用して、Document テーブルから項目を Forum オブジェクトとして取得することができます。

Example

```
DynamoDBContext context = new DynamoDBContext(client);

Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by primary key.
Document doc = context.ToDocument<Forum>(forum101);
```

DynamoDBContext でのオプションパラメータの指定

オブジェクト永続性モデルを使用する場合は、DynamoDBContext に次のオプションパラメータを指定できます。

- **ConsistentRead**-Load、Query、または Scan オペレーションを使用してデータを取得する場合、このオプションパラメータを追加して、データの最新の値をリクエストすることができます。

- **IgnoreNullValues**-このパラメータにより、Save オペレーション時に属性の null 値を無視するように DynamoDBContext に指示できます。このパラメータが false の場合 (または設定されていない場合)、null 値は、特定の属性を削除するディレクティブと見なされます。
- **SkipVersionCheck**–このパラメータは、項目の保存または削除を実行する際、バージョンの比較を行わないように DynamoDBContext に指示します。バージョニングの詳細については、「DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック」を参照してください。
- **TableNamePrefix**-すべてのテーブル名に特定の文字列をプレフィックスします。このパラメータが null の場合 (または設定されていない場合) 、プレフィックスは使用されません。
- DynamoDBEntryConversion – クライアントで使用する変換スキーマを指定します。このパラメータはバージョン V1 または V2 に設定できます。デフォルトのバージョンは V1 です。

  設定したバージョンに応じて、このパラメータの動作が変わります。例えば、次のようになります。

  - V1 では、bool データ型が N 数値型に変換されます。0 は false、1 は true を表します。V2 では、bool が BOOL に変換されます。
  - V2 の場合、リストと配列は HashSets と一緒にグループ化されません。数値、文字列ベースの型、バイナリベースの型のリストと配列は L (List) 型に変換され、リストを更新するために空で送信できます。この動作は、空のリストがワイヤ経由で送信されない V1 とは異なります。

    V1 の場合、List、HashSet、配列などのコレクションタイプは同じように扱われます。List、HashSet、数値の配列は NS (数値セット) 型に変換されます。

  次の例では、変換スキーマのバージョンを V2 に設定し、.NET 型と DynamoDB データ型間の変換動作を変更します。

```
var config = new DynamoDBContextConfig
{
    Conversion = DynamoDBEntryConversion.V2
};
var contextV2 = new DynamoDBContext(client, config);
```

次の C# サンプルコードでは、前述のオプションパラメータのうち 2 つ (ConsistentRead と SkipVersionCheck) を指定して、新しい DynamoDBContext を作成します。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context =
       new DynamoDBContext(client, new DynamoDBContextConfig { ConsistentRead = true,
 SkipVersionCheck = true});
```

DynamoDBContext では、このコンテキストを使用して送信した各リクエストに、これらのオプションパラメータが含められます。

次の C# サンプルコードに示すように、これらのパラメータを DynamoDBContext レベルで設定する代わりに、DynamoDBContext を使用して実行するオペレーションに対して個別に指定することもできます。この例では特定の書籍項目がロードされています。DynamoDBContext の Load メソッドでは、オプションパラメータとして ConsistentRead と SkipVersionCheck を指定します。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context = new DynamoDBContext(client);
Book bookItem = context.Load<Book>(productId,new DynamoDBContextConfig{ ConsistentRead
 = true, SkipVersionCheck = true });
```

この場合 DynamoDBContext には、Get リクエストを送信する場合のみ、これらのパラメータが含まれます。

DynamoDB と AWS SDK for .NET のオブジェクト永続性モデルを使用した楽観的ロック

オブジェクト永続性モデルではオプティミスティックロックがサポートされており、項目を更新または削除する前に、アプリケーションの項目バージョンとサーバー側の項目バージョンが同じになります。更新する項目を取得するとします。しかし、更新を返送する前に、他のアプリケーションが同じ項目を更新しました。この場合、アプリケーションに項目の古いコピーが残ることになります。オプティミスティックロックがない場合に更新を行うと、他のアプリケーションで行われた更新が上書きされます。

オブジェクト永続性モデルのオプティミスティックロック機能では、オプティミスティックロックを有効にするために DynamoDBVersion タグを使用できます。この機能を使用するには、バージョン番号を格納するためのプロパティをクラスに追加します。DynamoDBVersion 属性をプロパティに

追加します。最初にオブジェクトを保存すると、DynamoDBContext によってバージョン番号が割り当てられ、項目を更新するたびにその値が増えていきます。

更新または削除リクエストは、クライアント側のオブジェクトのバージョンが、サーバー側の対応する項目のバージョン番号に一致する場合のみ成功します。アプリケーションに古いコピーがある場合に項目を更新または削除するには、その前にサーバーから最新バージョンを取得する必要があります。

次の C# サンプルコードでは、オブジェクト永続性属性と合わせて Book クラスを定義し、ProductCatalog テーブルにマッピングします。VersionNumber 属性が指定されたクラスの DynamoDBVersion プロパティには、バージョン番号が格納されます。

Example

```
[DynamoDBTable("ProductCatalog")]
  public class Book
  {
    [DynamoDBHashKey]   //Partition key
    public int Id { get; set; }
    [DynamoDBProperty]
    public string Title { get; set; }
    [DynamoDBProperty]
    public string ISBN { get; set; }
    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }
    [DynamoDBVersion]
    public int? VersionNumber { get; set; }
  }
```

> **Note**
>
> DynamoDBVersion 属性は、null が許容された数値プリミティブ型 (int? など) に対してのみ適用できます。

オプティミスティックロックは、DynamoDBContext オペレーションに対して次のような影響があります。

- DynamoDBContext は、新しい項目に対して初期バージョン番号 0 を割り当てます。既存の項目を取得し、1 つ以上のプロパティを更新して変更を保存しようとすると、保存オペレーションは、クライアント側とサーバー側のバージョン番号が一致する場合にのみ成功しま

す。DynamoDBContext によってバージョン番号が増加します。バージョン番号を設定する必要はありません。

- 次の C# サンプルコードに示すように、Delete メソッドでは、プライマリキーの値またはオブジェクトのいずれかをパラメータとして指定できるオーバーロードを使用できます。

  Example

```
DynamoDBContext context = new DynamoDBContext(client);
...
// Load a book.
Book book = context.Load<ProductCatalog>(111);
// Do other operations.
// Delete 1 - Pass in the book object.
context.Delete<ProductCatalog>(book);

// Delete 2 - Pass in the Id (primary key)
context.Delete<ProductCatalog>(222);
```

  パラメータとしてオブジェクトを指定した場合、オブジェクトのバージョンがサーバー側の対応する項目のバージョンと一致する場合のみ、削除が成功します。ただし、パラメータとしてプライマリキーの値を入力した場合には、DynamoDBContext はバージョン番号を認識せず、バージョンチェックを行わずに項目を削除します。

  オブジェクト永続性モデルのコードで、オプティミスティックロックを内部実装する場合は、DynamoDB の条件付き更新と条件付き削除用の API アクションが使用されることに注意してください。

楽観的ロックの無効化

オプティミスティックロックを無効にするには、SkipVersionCheck 設定プロパティを使用します。このプロパティは、DynamoDBContext の作成時に設定できます。この場合、このコンテキストを使用して作成したすべてのリクエストについて、オプティミスティックロックが無効になります。詳細については、「DynamoDBContext でのオプションパラメータの指定」を参照してください。

コンテキストレベルでプロパティを設定する代わりに、次の C# サンプルコードに示すように、特定のオペレーションに対するオプティミスティックロックを無効にすることができます。この例では、このコンテキストを使用して書籍項目を削除しています。Delete メソッドはオプションの SkipVersionCheck プロパティを true に設定し、バージョンチェックを無効にします。

Example

```
DynamoDBContext context = new DynamoDBContext(client);
// Load a book.
Book book = context.Load<ProductCatalog>(111);
...
// Delete the book.
context.Delete<Book>(book, new DynamoDBContextConfig { SkipVersionCheck = true });
```

AWS SDK for .NET のオブジェクト永続性モデルを使用した、DynamoDB での任意データのマッピング

サポート済みの .NET 型 (「サポートされているデータ型」を参照) に加えて、アプリケーション内にあり Amazon DynamoDB 型に直接マッピングされていない型を使用できます。オブジェクト永続性モデルでは、任意型から DynamoDB 型に (またはその反対に) データを変換できるコンバーターを用意することで、任意型のデータを格納できます。コンバーターコードによって、オブジェクトの保存およびロード中にデータが変換されます。

クライアント側で任意の型を作成できます。ただし、テーブルには、いずれかの DynamoDB 型を使用してデータが格納されます。また、クエリおよびスキャンの際のデータ比較は、DynamoDB に格納されているデータに対して行われます。

次の C# コード例では、Book、Id、Title、および ISBN プロパティを使用して Dimension クラスを定義しています。Dimension プロパティは、DimensionType、Height、および Width プロパティを記述する Thickness に含まれています。このコード例では、コンバーターメソッド ToEntry および FromEntry によって、DimensionType と DynamoDB の文字列型との間でデータが変換されます。たとえば、Book インスタンスを保存すると、コンバータによって、「8.5 x 11 x 05」などの書籍 Dimension 文字列が作成されます。書籍を取得すると、その文字列は DimensionType インスタンスに変換されます。

この例では、Book 型を ProductCatalog テーブルにマッピングしています。サンプルの Book インスタンスを保存し、取得し、寸法を更新し、更新された Book を再度保存しています。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
```

```
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class HighLevelMappingArbitraryData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                DynamoDBContext context = new DynamoDBContext(client);

                // 1. Create a book.
                DimensionType myBookDimensions = new DimensionType()
                {
                    Length = 8M,
                    Height = 11M,
                    Thickness = 0.5M
                };

                Book myBook = new Book
                {
                    Id = 501,
                    Title = "AWS SDK for .NET Object Persistence Model Handling
 Arbitrary Data",
                    ISBN = "999-9999999999",
                    BookAuthors = new List<string> { "Author 1", "Author 2" },
                    Dimensions = myBookDimensions
                };

                context.Save(myBook);

                // 2. Retrieve the book.
                Book bookRetrieved = context.Load<Book>(501);

                // 3. Update property (book dimensions).
                bookRetrieved.Dimensions.Height += 1;
                bookRetrieved.Dimensions.Length += 1;
                bookRetrieved.Dimensions.Thickness += 0.2M;
                // Update the book.
```

```
                context.Save(bookRetrieved);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }
    }
    [DynamoDBTable("ProductCatalog")]
    public class Book
    {
        [DynamoDBHashKey] //Partition key
        public int Id
        {
            get; set;
        }
        [DynamoDBProperty]
        public string Title
        {
            get; set;
        }
        [DynamoDBProperty]
        public string ISBN
        {
            get; set;
        }
        // Multi-valued (set type) attribute.
        [DynamoDBProperty("Authors")]
        public List<string> BookAuthors
        {
            get; set;
        }
        // Arbitrary type, with a converter to map it to DynamoDB type.
        [DynamoDBProperty(typeof(DimensionTypeConverter))]
        public DimensionType Dimensions
        {
            get; set;
        }
    }

    public class DimensionType
    {
```

```
        public decimal Length
        {
            get; set;
        }
        public decimal Height
        {
            get; set;
        }
        public decimal Thickness
        {
            get; set;
        }
    }

    // Converts the complex type DimensionType to string and vice-versa.
    public class DimensionTypeConverter : IPropertyConverter
    {
        public DynamoDBEntry ToEntry(object value)
        {
            DimensionType bookDimensions = value as DimensionType;
            if (bookDimensions == null) throw new ArgumentOutOfRangeException();

            string data = string.Format("{1}{0}{2}{0}{3}", " x ",
                                bookDimensions.Length, bookDimensions.Height,
bookDimensions.Thickness);

            DynamoDBEntry entry = new Primitive
            {
                Value = data
            };
            return entry;
        }

        public object FromEntry(DynamoDBEntry entry)
        {
            Primitive primitive = entry as Primitive;
            if (primitive == null || !(primitive.Value is String) ||
string.IsNullOrEmpty((string)primitive.Value))
                throw new ArgumentOutOfRangeException();

            string[] data = ((string)(primitive.Value)).Split(new string[] { " x " },
StringSplitOptions.None);
            if (data.Length != 3) throw new ArgumentOutOfRangeException();
```

```
            DimensionType complexData = new DimensionType
            {
                Length = Convert.ToDecimal(data[0]),
                Height = Convert.ToDecimal(data[1]),
                Thickness = Convert.ToDecimal(data[2])
            };
            return complexData;
        }
    }
}
```

このデベロッパーガイドのコード例の実行

AWS SDK では、次の言語で Amazon DynamoDB を幅広くサポートしています。

- Java:
- ブラウザの JavaScript
- .NET
- Node.js
- PHP
- Python *
- Ruby
- C++
- Go:
- Android
- iOS

この開発者ガイドのコード例は、以下のプログラミング言語を使用して DynamoDB オペレーションの詳細を示しています。

- Java コードの例
- .NET コード例

この演習を開始する前に、AWS アカウントを作成し、アクセスキーとシークレットキーを取得して、コンピュータで AWS Command Line Interface (AWS CLI) をセットアップします。詳細については、「」を参照してくださいDynamoDB (ウェブサービス) の設定

> **Note**
>
> DynamoDB のダウンロード可能バージョンを使用する場合は、AWS CLI を使用してテーブルとサンプルデータを作成する必要があります。また、各 `--endpoint-url` コマンドで AWS CLI パラメータも指定する必要があります。詳細については、「ローカルエンドポイントの設定」を参照してください。

DynamoDB でのコード例用のテーブルの作成とデータのロード

DynamoDB でのテーブルの作成、サンプルデータセットのロード、データに対するクエリの実行、およびデータの更新に関する基本については、以下を参照してください。

- ステップ 1: DynamoDB にテーブルを作成する
- ステップ 2: DynamoDB テーブルにデータを書き込む
- ステップ 3: DynamoDB テーブルからデータを読み込む
- ステップ 4: DynamoDB テーブルのデータを更新する

Java コードの例

トピック

- Java: AWS 認証情報の設定
- Java: AWS リージョンとエンドポイントの設定

この開発者ガイドには、Java コードスニペットとすぐに使用できるプログラムが含まれています。これらのコード例は、次のセクションで確認することができます。

- DynamoDB での項目と属性の操作
- DynamoDB でのテーブルとデータの操作
- DynamoDB のテーブルに対するクエリの実行
- DynamoDB でのテーブルのスキャン
- DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上
- Java 1.x: DynamoDBMapper
- DynamoDB Streams の変更データキャプチャ

Eclipse と AWS Toolkit for Eclipse を使用して、すぐに開始できます。フル機能の IDE に加えて、自動更新で AWS SDK for Java および AWS アプリケーションの構築用に事前設定されたテンプレートも取得できます。

Java サンプルコードを実行するには (Eclipse を使用)

1. Eclipse IDE をダウンロードし、インストールします。
2. のダウンロードおよびインストール。AWS Toolkit for Eclipse。
3. Eclipse を起動し、Eclipse メニューから、[File] (ファイル)、[New] (新規)、[Other] (その他) の順に選択します。
4. [Select a wizard] (ウィザードの選択) で、AWS、[AWS Java Project] (Java プロジェクト) の順に選択してから、[Next] (次へ) をクリックします。
5. [AWS Java の作成] で、次の操作を行います。

   a. [Project name] (プロジェクト名) にプロジェクトの名前を入力します。

   b. [Select Account] (アカウントの選択) リストから認証情報プロファイルを選択します。

      AWS Toolkit for Eclipse を初めて使用する場合、[AWS アカウントの設定] を選択して、AWS 認証情報を設定します。
6. [Finish] (完了) を選択してプロジェクトを作成します。
7. Eclipse メニューから [File] (ファイル)、[New] (新規)、[Class] (クラス) の順に選択します。
8. [Java Class] (Java クラス) の [Name] (名前) にクラスの名前を入力し (実行するコード例と同じ名前を使用)、[Finish] (完了) を選択してクラスを作成します。
9. ドキュメントページから Eclipse エディタにサンプルコードをコピーします。
10. コードを実行するには、Eclipse メニューの [Run] (実行) を選択します。

SDK for Java には、DynamoDB を操作するためにスレッドセーフなクライアントが用意されています。ベストプラクティスとして、ご利用のアプリケーションでクライアントを 1 つ作成し、そのクライアントをスレッド間で再利用します。

詳細については、AWS SDK for Java を参照してください。

> **Note**
>
> このガイドのサンプルコードは、最新バージョンの AWS SDK for Java で使用するためのものです。

> AWS Toolkit for Eclipse を使用している場合、SDK for Java の自動更新を設定することができます。これを Eclipse で行うには、[Preferences] (基本設定) に移動し、AWS Toolkit、AWS SDK for Java、[Download new SDKs automatically] (新しい SDK を自動的にダウンロード) の順に選択します。

Java: AWS 認証情報の設定

SDK for Java では、ランタイムにアプリケーションに AWS 認証情報を指定する必要があります。このガイドのコード例では、AWS 認証情報ファイルを使用していることを前提としています。詳細については、「AWS SDK for Java デベロッパーガイド」の「AWS 認証情報のセットアップ」を参照してください。

`~/.aws/credentials` という名前の AWS 認証情報ファイルの例を次に示します。ここで、チルダ文字 (~) はホームディレクトリを表します。

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Java: AWS リージョンとエンドポイントの設定

デフォルトでは、コード例は、米国西部 (オレゴン) リージョンの DynamoDB にアクセスします。このリージョンを変更するには、`AmazonDynamoDB` プロパティを変更します。

次のサンプルコードは、新しい `AmazonDynamoDB` をインスタンス化します。

```
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import com.amazonaws.regions.Regions;
...
// This client will default to US West (Oregon)
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
.withRegion(Regions.US_WEST_2)
.build();
```

`withRegion` メソッドを使用して、利用可能な任意のリージョンで、DynamoDB を対象としてコードを実行できます。リージョンのリストについては、「Amazon Web Services 全般のリファレンス」の「AWS リージョンとエンドポイント」を参照してください。

ローカルコンピュータで DynamoDB を使用してサンプルコードを実行する場合は、エンドポイントを設定する必要があります。

AWS SDKV1

```
AmazonDynamoDB client =
 AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
.build();
```

AWS SDK V2

```
DynamoDbClient client = DynamoDbClient.builder()
    .endpointOverride(URI.create("http://localhost:8000"))
    // The region is meaningless for local DynamoDb but required for client builder
 validation
    .region(Region.US_EAST_1)
    .credentialsProvider(StaticCredentialsProvider.create(
    AwsBasicCredentials.create("dummy-key", "dummy-secret")))
    .build();
```

.NET コード例

トピック

- .NET: AWS 認証情報の設定
- .NET: AWS リージョンとエンドポイントの設定

このガイドには、.NET コードスニペットとすぐに使用できるプログラムが含まれています。これらのコード例は、次のセクションで確認することができます。

- DynamoDB での項目と属性の操作
- DynamoDB でのテーブルとデータの操作
- DynamoDB のテーブルに対するクエリの実行
- DynamoDB でのテーブルのスキャン
- DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上
- DynamoDB での .NET ドキュメントの操作
- .NET オブジェクト永続性モデルと DynamoDB の使用
- DynamoDB Streams の変更データキャプチャ

AWS SDK for .NET と Toolkit for Visual Studio を使用すると、すぐに始めることができます。

.NET サンプルコードを実行するには (Visual Studio を使用)

1. Microsoft Visual Studio をダウンロードし、インストールします。
2. (オプション) Toolkit for Visual Studio をダウンロードしてインストールします。
3. AWS 認証情報をセットアップします。共有 AWS 認証情報ファイル (`~/.aws/credentials`) で認証情報プロファイルを設定します。詳細については、「AWS SDK for .NET デベロッパーガイド」の「AWS 認証情報の設定」を参照してください。
4. Visual Studio を起動します。[File] (ファイル)、[New] (新規)、[Project] (プロジェクト) の順に選択します。
5. [コンソールアプリ] を検索し、.NET をターゲットとする C# テンプレートを選択し、[次へ] を選択します。プロジェクト名と場所を設定し、[作成] を選択します。
6. プロジェクトに DynamoDB NuGet パッケージ用 AWS SDK を追加します。
   a. ソリューションエクスプローラーで、プロジェクトのコンテキスト (右クリック) メニューを開いて、[Manage NuGet Packages] (NuGet パッケージの管理) を選択します。
   b. NuGet パッケージマネージャーで、[Browse] (参照) を選択します。
   c. 検索ボックスに「**`AWSSDK.DynamoDBv2`**」と入力し、完了するまで待ちます。
   d. AWSSDK.DynamoDBv2、[Install] (インストール) の順に選択します。
7. Visual Studio プロジェクトで、`Program.cs` を開きます。内容を、実行するドキュメントページのコード例に置き換えます。
8. コードを実行するには、Visual Studio ツールバーの [Start] (開始) を選択します。

SDK for .NET には、DynamoDB を操作するためにスレッドセーフなクライアントが用意されています。ベストプラクティスとして、ご利用のアプリケーションでクライアントを 1 つ作成し、そのクライアントをスレッド間で再利用します。

詳細については、「AWS SDK for .NET」を参照してください。

> **Note**
>
> このガイドのサンプルコードは、最新バージョンの AWS SDK for .NET で使用するためのものです。

.NET: AWS 認証情報の設定

SDK for .NET では、ランタイムにアプリケーションに AWS 認証情報を指定する必要があります。このガイドのコード例は、SDK ストアを使用して AWS 認証情報ファイルを管理することを前提としています。詳細については、「AWS SDK for .NET デベロッパーガイド」の「SDK ストアの使用」を参照してください。SDK Store は、手動で管理された短期または長期の認証情報を保持しますが、これは推奨の方法ではなくなりました。アクセスを設定する推奨方法については、「AWS SDK for .NET デベロッパーガイド」の「IAM Identity Center を使用して AWS アクセスを設定する」を参照してください。

Toolkit for Visual Studio では、任意の数のアカウントの複数セットの認証情報がサポートされています。各セットはプロファイルと呼ばれています。Visual Studio では、プロジェクトの `App.config` ファイルにエントリを追加するため、アプリケーションはランタイム時に AWS 認証情報を見つけることができます。

次の例は、Toolkit for Visual Studio を使用した新しいプロジェクトの作成時に生成されたデフォルトの `App.config` ファイルを示しています。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
    <add key="AWSProfileName" value="default"/>
    <add key="AWSRegion" value="us-west-2" />
 </appSettings>
</configuration>
```

ランタイム時に、このプログラムは `AWSProfileName` エントリで指定されているように、AWS 認証情報の `default` セットを使用します。AWS 認証情報自体は、暗号化されたフォームで SDK ストアに保持されます。Toolkit for Visual Studio は、すべて Visual Studio 内からの認証情報を管理するためのグラフィカルユーザーインターフェイスを使用します。詳細については、「AWS Toolkit for Visual Studio ユーザーガイド」の「認証情報の指定」を参照してください。

> **Note**
>
> デフォルトでは、コード例は、米国西部 (オレゴン) リージョンの DynamoDB にアクセスします。リージョンを変更するには、App.config ファイルで `AWSRegion` エントリを変更します。`AWSRegion` を DynamoDB が利用可能なすべてのリージョンに設定します。リージョンのリストについては、「Amazon Web Services 全般のリファレンス」の「AWS リージョンとエンドポイント」を参照してください。

.NET: AWS リージョンとエンドポイントの設定

デフォルトでは、コード例は、米国西部 (オレゴン) リージョンの DynamoDB にアクセスします。リージョンを変更するには、AWSRegion ファイルで App.config エントリを変更します。または、AmazonDynamoDBClient プロパティを変更してリージョンを変更することもできます。

次のサンプルコードは、新しい AmazonDynamoDBClient をインスタンス化します。クライアントは、別のリージョンでコードが DynamoDB に対して実行するように変更されます。

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// This client will access the US East 1 region.
clientConfig.RegionEndpoint = RegionEndpoint.USEast1;
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

リージョンのリストについては、「Amazon Web Services 全般のリファレンス」の「AWS リージョンとエンドポイント」を参照してください。

ローカルコンピュータで DynamoDB を使用してサンプルコードを実行する場合は、エンドポイントを設定する必要があります。

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// Set the endpoint URL
clientConfig.ServiceURL = "http://localhost:8000";
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

DynamoDB 低レベル API

Amazon DynamoDB の低レベル API は、DynamoDB 用のプロトコルレベルのインターフェイスです。このレベルでは、すべての HTTP リクエストは、適切な形式で有効なデジタル署名がある必要があります。

AWS SDK は、低レベル DynamoDB API リクエストをユーザーに代わって作成し、DynamoDB からのレスポンスを処理します。これにより、低レベルの詳細ではなく、アプリケーションロジックに専念することができます。ただし、低レベル DynamoDB API の動作方法についての基本的な知識も役立ちます。

低レベル DynamoDB API の詳細については、Amazon DynamoDB API リファレンスを参照してください。

> **Note**
>
> DynamoDB Streams には、DynamoDB とは別に独自の低レベル API があり、AWS SDK で完全にサポートされています。
> 詳細については、「DynamoDB Streams の変更データキャプチャ」を参照してください。低レベル DynamoDB Streams API については、Amazon DynamoDB Streams API リファレンスを参照してください。

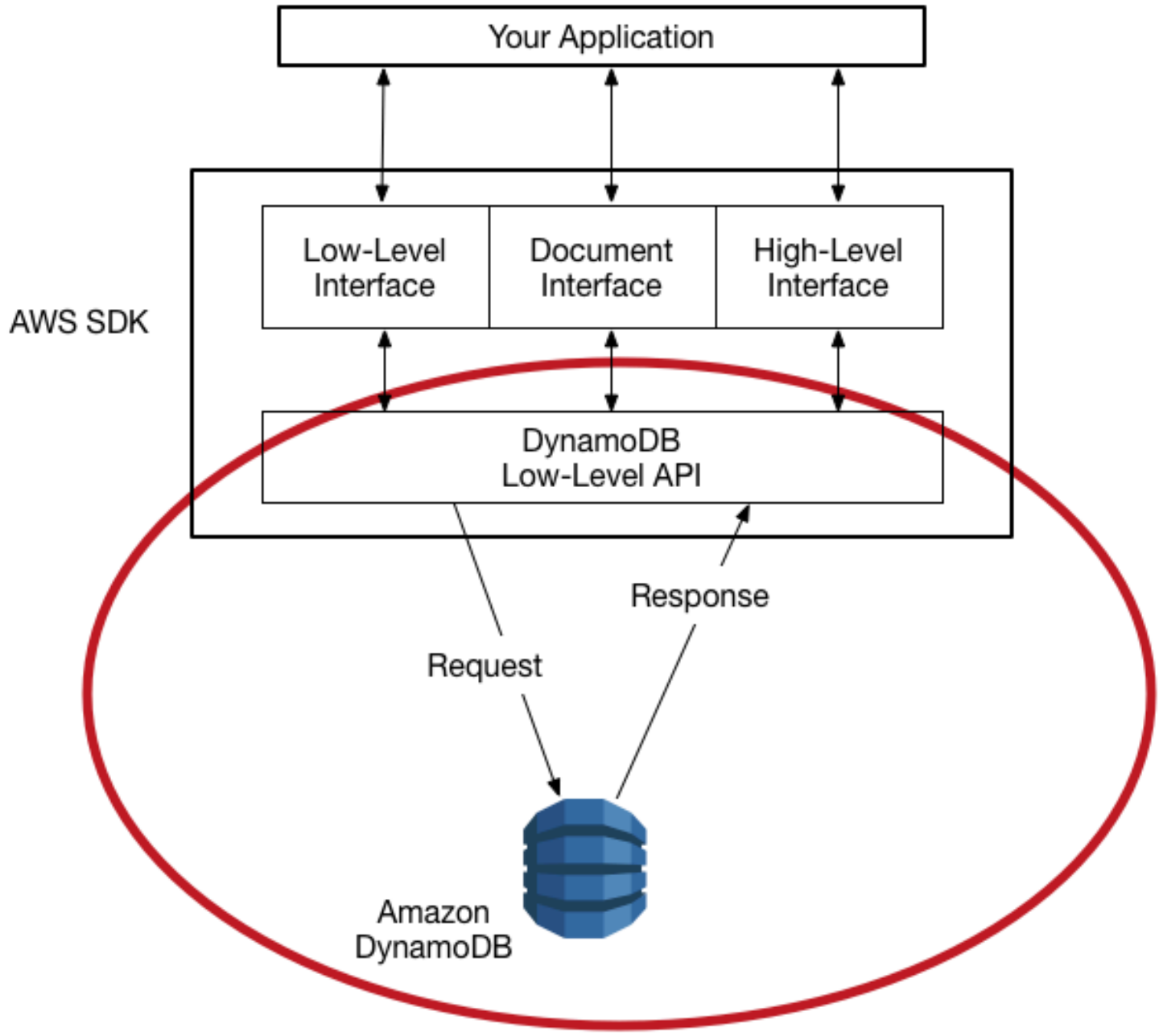
低レベル DynamoDB API は、ワイヤプロトコル形式として、JavaScript Object Notation (JSON) を使用しています。JSON では、データ値とデータ構造が両方同時にわかるように、データが階層で示されます。名前と値のペアは、`name:value` の形式で定義されます。データ階層は、名前と値のペアをブラケットで囲み、ネストする形で定義します。

DynamoDB は、ストレージ形式としてではなく、トランスポートプロトコルとしてのみ、JSON を使用しています。JSON を使用して AWS SDK が DynamoDB にデータを送信し、DynamoDB が JSON で応答します。DynamoDB は、JSON 形式でデータを永続的に保存しません。

> **Note**
>
> JSON の詳細については、`JSON.org` ウェブサイトで「JSON の入門」を参照してください。

トピック

- リクエストの形式
- レスポンスの形式
- データ型記述子
- 数値データ
- バイナリデータ



リクエストの形式

DynamoDB 低レベル API は、HTTP(S) POST リクエストを入力として受け付けます。AWS SDK はこれらのリクエストを作成します。

`Pets` という名のテーブルに、`AnimalType` (パーティションキー)、`Name` (ソートキー) によって構成されるキースキーマがあるとします。これらの属性はいずれも、`string` 型になります。`Pets` から項目を取得するために、AWS SDK は次のリクエストを作成します。

```
POST / HTTP/1.1
```

```
Host: dynamodb.<region>.<domain>;
Accept-Encoding: identity
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.0
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
 Signature=<Signature>
X-Amz-Date: <Date>
X-Amz-Target: DynamoDB_20120810.GetItem

{
    "TableName": "Pets",
    "Key": {
        "AnimalType": {"S": "Dog"},
        "Name": {"S": "Fido"}
    }
}
```

このリクエストに関して以下の点に注意してください。

- Authorization ヘッダーには、DynamoDB がリクエストを認証するのに必要な情報が含まれています。詳細は、「Amazon Web Services 全般のリファレンス」の「AWS API リクエストへの署名」と「署名バージョン 4 の署名プロセス」を参照してください。
- X-Amz-Target ヘッダーには、DynamoDB オペレーションの名前である GetItem が含まれます 。(これは、低レベル API バージョンと共に示されます。この場合は 20120810 となります。)
- リクエストのペイロード (本文) には、JSON 形式で、オペレーションのパラメーターが含まれます。GetItem オペレーションでは、パラメーターは TableName と Key です。

レスポンスの形式

リクエストを受け取ったら、DynamoDB が処理しレスポンスを返します。前の例に示したように、HTTP レスポンスペイロードには、オペレーションからの結果が含まれます。

```
HTTP/1.1 200 OK
x-amzn-RequestId: <RequestId>
x-amz-crc32: <Checksum>
Content-Type: application/x-amz-json-1.0
Content-Length: <PayloadSizeBytes>
Date: <Date>
{
    "Item": {
```

```
        "Age": {"N": "8"},
        "Colors": {
            "L": [
                {"S": "White"},
                {"S": "Brown"},
                {"S": "Black"}
            ]
        },
        "Name": {"S": "Fido"},
        "Vaccinations": {
            "M": {
                "Rabies": {
                    "L": [
                        {"S": "2009-03-17"},
                        {"S": "2011-09-21"},
                        {"S": "2014-07-08"}
                    ]
                },
                "Distemper": {"S": "2015-10-13"}
            }
        },
        "Breed": {"S": "Beagle"},
        "AnimalType": {"S": "Dog"}
    }
}
```

この時点で、AWS SDK は、さらに処理するためにアプリケーションに応答データを返します。

> **Note**
>
> リクエストを処理できない場合は、DynamoDB より HTTP エラーコードとメッセージが返ります。AWS SDK は、これらを例外形式でアプリケーションに伝達します。詳細については、「DynamoDB でのエラー処理」を参照してください。

データ型記述子

低レベル DynamoDB API のプロトコルは、各属性がデータ型記述子に伴われる必要があります。データ型記述子は、各属性を解釈する方法を DynamoDB に伝えるトークンです。

リクエストの形式とレスポンスの形式には、データ型記述子が使用されている例が示されています。`GetItem` リクエストでは、S キースキーマ属性 (`Pets` と `AnimalType`) に `Name` を `string` 型

で指定します。GetItem レスポンスには、string (S)、number (N)、map (M)、list (L) 型の属性を持つ、Pets 項目が含まれます。

DynamoDB データ型記述子の一覧を次に示します。

- **S** — 文字列
- **N** — 数値
- **B** — バイナリ
- **BOOL** — ブール
- **NULL** — Null
- **M** — マップ
- **L** — リスト
- **SS** — 文字列セット
- **NS** — 数値セット
- **BS** — バイナリセット

次の表は、各データ型記述子の正しい JSON 形式を示しています。数値は精度を維持するために文字列として表されますが、ブール値と null はネイティブの JSON 型を使用することに注意してください。

ディスクリプタ	JSON 形式	注意事項
S	`{"S": "Hello"}`	値は JSON 文字列です。
N	`{"N": "123.45"}`	値は文字列であり、JSON 数値ではありません。これにより、言語間で精度が維持されます。
B	`{"B": "dGhpcyBpcyBhIHRlc3Q="}`	値は base64 でエンコードされた文字列です。
BOOL	`{"BOOL": true}`	値は JSON ブール値 (`true` または `false`) で、文字列ではありません。
NULL	`{"NULL": true}`	値は null を示す JSON ブール値 `true` です。

ディスクリプタ	JSON 形式	注意事項
M	`{"M": {"Name": {"S": "Joe"}}}`	値は、属性名と値のペアの JSON オブジェクトです。
L	`{"L": [{"S": "Red"}, {"N": "5"}]}`	値は属性値の JSON 配列です。
SS	`{"SS": ["Red", "Blue"]}`	値は文字列の JSON 配列です。
NS	`{"NS": ["1", "2.5"]}`	値は数値文字列の JSON 配列です。
BS	`{"BS": ["U3Vubnk=", "UmFpbnk="]}`	値は base64 でエンコードされた文字列の JSON 配列です。

> **Note**
>
> DynamoDB データ型の詳細な説明については、「データ型」を参照してください。

数値データ

プログラミング言語により、提供される JSON のサポートのレベルが異なります。場合によっては、JSONドキュメントを検証し解析するにあたり、サードパーティーのライブラリを使用することもできます。

JSON Number 型に基づいて構築されたサードパーティーライブラリもあり、`int` や `long`、`double` など独自の型を提供しています。ただし、他のデータ型に正確にマッピングされない DynamoDB のネイティブ数値データ型が使用されるため、このようなデータ型の区別が競合の原因になる可能性があります。加えて、多くの JSON ライブラリでは固定精度の数値は処理されず、小数点を含む数字列は自動的に倍精度浮動小数点データ型であると推定されます。

これらの問題を解決するために、DynamoDB ではデータ損失のない単一の数値型が用意されています。誤って倍精度の値に暗黙的変換が行われないように、DynamoDB では数値のデータ転送には文字列が使用されます。この方法によって、01、2、03 などの値を適切な順序で配置するなど、適切な並べ替えセマンティクスを維持しながら、柔軟に属性値を更新することが可能になります。

数値の精度がアプリケーションにとって重要な場合は、数値を文字列に変換してから、DynamoDB に渡します。

バイナリデータ

DynamoDB ではバイナリ属性がサポートされています。ただし JSON では、ネイティブではバイナリデータのエンコードがサポートされていません。リクエストでバイナリデータを送信するには、base64 形式でエンコードする必要があります。DynamoDB はリクエストを受け取ると、base64 データをバイナリに復号します。

DynamoDB で使用される base64 エンコーディングスキームは、Internet Engineering Task Force (IETF) ウェブサイトの「RFC 4648」に記載されています。

Python と Boto3 による Amazon DynamoDB のプログラミング

このガイドは、Python で Amazon DynamoDB を使用したいと考えているプログラマーを対象としています。さまざまな抽象化レイヤー、設定管理、エラー処理、再試行ポリシーの制御、キープアライブの管理などについて説明します。

トピック

- Boto について
- Boto ドキュメントの使用
- クライアントとリソースの抽象化レイヤーを理解する
- テーブルリソース batch_writer の使用
- クライアントレイヤーとリソースレイヤーを調べるその他のコード例
- Client オブジェクトと Resource オブジェクトがセッションやスレッドとどのように相互作用するかを理解する
- Config オブジェクトのカスタマイズ
- エラー処理
- ログ記録
- イベントフック
- ページネーションとページネーター
- ウェイター

Boto について

Python から DynamoDB にアクセスするには、Python 用の公式 AWS SDK (一般に Boto3 と呼ばれる) を使用します。Boto (ボトと発音) という名前は、アマゾン川に自生する淡水イルカに由来してい

ます。Boto3 ライブラリは, ライブラリの 3 番目のメジャーバージョンであり、2015 年に初めてリリースされました。Boto3 ライブラリは、DynamoDB だけでなくすべての AWS サービスをサポートするため、非常に大きいです。このオリエンテーションは、DynamoDB に関連する Boto3 の部分のみを対象としています。

Boto は、GitHub でホストされるオープンソースプロジェクトとして AWS によって管理および公開されています。Botocore と Boto3 の 2 つのパッケージに分かれています。

- Botocore は低レベルの機能を備えています。Botocore には、クライアント、セッション、認証情報、設定、および例外クラスがあります。
- Boto3 は Botocore の上に構築されています。よりハイレベルで、より Python 的なインターフェイスを提供します。具体的には、DynamoDB テーブルをリソースとして公開し、低レベルのサービス指向のクライアントインターフェイスよりもシンプルで洗練されたインターフェイスを提供します。

これらのプロジェクトは GitHub でホストされているため、ソースコードを表示したり、未解決の問題を追跡したり、独自の問題を送信したりできます。

Boto ドキュメントの使用

以下のリソースを使って Boto ドキュメントを使い始めましょう。

- まず、パッケージインストールの確実な出発点となる「クイックスタート」セクションから始めてください。まだ Boto3 をインストールしていない場合は、そちらを参照してください (Boto3 は多くの場合、AWS Lambda などの AWS サービス内で自動的に利用可能になります)。
- その後、ドキュメントの DynamoDB ガイドに注目してください。テーブルの作成と削除、項目の操作、バッチ操作の実行、クエリの実行、スキャンの実行など、基本的な DynamoDB アクティビティを実行する方法を示しています。例ではリソースインターフェイスを使用しています。`boto3.resource('dynamodb')` が表示されれば、上位レベルのリソースインターフェイスを使用していることがわかります。
- ガイドを読み終えたら、DynamoDB リファレンスを確認できます。このランディングページには、使用できるクラスとメソッドがすべて記載されています。上部には `DynamoDB.Client` クラスが表示されます。これにより、コントロールプレーンとデータプレーンのすべての操作に低レベルでアクセスできます。一番下にある `DynamoDB.ServiceResource` クラスを見てください。これは上位レベルの Python インターフェイスです。これを使用すると、テーブルを作成したり、複数のテーブルにわたってバッチ操作を実行したり、テーブル固有のアクションのために `DynamoDB.ServiceResource.Table` インスタンスを取得したりできます。

クライアントとリソースの抽象化レイヤーを理解する

使用する 2 つのインターフェイスは、クライアントインターフェイスとリソースインターフェイスです。

- 低レベルのクライアントインターフェイスは、基盤となるサービス API に 1 対 1 で対応します。DynamoDB が提供するすべての API は、クライアントを通じて利用できます。つまり、クライアントインターフェイスは完全な機能を提供できますが、多くの場合、より冗長で使い方が複雑です。
- 上位レベルのリソースインターフェイスは、基盤となるサービス API と 1 対 1 で対応するわけではありません。ただし、`batch_writer` などのサービスへのアクセスが便利になります。

クライアントインターフェイスを使用して項目を挿入する例を次に示します。すべての値が、型を示すキー (文字列は 'S'、数値は 'N') と文字列としての値のマップとして渡されることに注目してください。これは DynamoDB JSON 形式と呼ばれます。

```
import boto3

dynamodb = boto3.client('dynamodb')

dynamodb.put_item(
    TableName='YourTableName',
    Item={
        'pk': {'S': 'id#1'},
        'sk': {'S': 'cart#123'},
        'name': {'S': 'SomeName'},
        'inventory': {'N': '500'},
        # ... more attributes ...
    }
)
```

リソースインターフェイスを使用した同じ `PutItem` 操作を次に示します。データ型の指定は暗黙的です。

```
import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')
```

```
table.put_item(
    Item={
        'pk': 'id#1',
        'sk': 'cart#123',
        'name': 'SomeName',
        'inventory': 500,
        # ... more attributes ...
    }
)
```

必要に応じて、boto3 に用意されている TypeSerializer および TypeDeserializer クラスを使用して、通常の JSON と DynamoDB JSON の間で変換できます。

```
def dynamo_to_python(dynamo_object: dict) -> dict:
    deserializer = TypeDeserializer()
    return {
        k: deserializer.deserialize(v)
        for k, v in dynamo_object.items()
    }

def python_to_dynamo(python_object: dict) -> dict:
    serializer = TypeSerializer()
    return {
        k: serializer.serialize(v)
        for k, v in python_object.items()
    }
```

クライアントインターフェイスを使用してクエリを実行する方法は次のとおりです。クエリは JSON コンストラクトとして表現されます。キーワードの競合が発生する可能性がある場合は、変数置換が必要な KeyConditionExpression 文字列を使用します。

```
import boto3

client = boto3.client('dynamodb')

# Construct the query
response = client.query(
    TableName='YourTableName',
    KeyConditionExpression='pk = :pk_val AND begins_with(sk, :sk_val)',
    FilterExpression='#name = :name_val',
    ExpressionAttributeValues={
```

```
        ':pk_val': {'S': 'id#1'},
        ':sk_val': {'S': 'cart#'},
        ':name_val': {'S': 'SomeName'},
    },
    ExpressionAttributeNames={
        '#name': 'name',
    }
)
```

リソースインターフェイスを使った同じクエリ操作を短くして簡略化できます。

```
import boto3
from boto3.dynamodb.conditions import Key, Attr

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

response = table.query(
    KeyConditionExpression=Key('pk').eq('id#1') & Key('sk').begins_with('cart#'),
    FilterExpression=Attr('name').eq('SomeName')
)
```

最後の例として、テーブルのおおよそのサイズ (テーブルに保持され、約 6 時間ごとに更新されるメタデータ) を取得したいとします。クライアントインターフェイスで、`describe_table()` 操作を行い、返された JSON 構造から回答を引き出します。

```
import boto3

dynamodb = boto3.client('dynamodb')

response = dynamodb.describe_table(TableName='YourTableName')
size = response['Table']['TableSizeBytes']
```

リソースインターフェイスでは、テーブルは describe 操作を暗黙的に実行し、データを属性として直接表示します。

```
import boto3

dynamodb = boto3.resource('dynamodb')
```

```
table = dynamodb.Table('YourTableName')
size = table.table_size_bytes
```

> **Note**
>
> クライアントインターフェイスとリソースインターフェイスのどちらを使用して開発するかを検討する際、リソースドキュメントではリソースインターフェイスに新しい機能が追加されないことに注意してください。「AWS Python SDK チームは boto3 のリソースインターフェイスに新しい機能を追加する予定はありません。既存のインターフェイスは boto3 のライフサイクル中も引き続き動作します。顧客はクライアントインターフェイスを通じて新しいサービス機能にアクセスできます。」

テーブルリソース batch_writer の使用

上位レベルのテーブルリソースでのみ利用できる便利な機能の 1 つは、batch_writer です。DynamoDB はバッチ書き込み操作をサポートしており、1 つのネットワークリクエストで最大 25 件の入力または削除操作が可能です。このようなバッチ処理は、ネットワークラウンドトリップを最小限に抑えることで効率性を高めます。

低レベルのクライアントライブラリでは、client.batch_write_item( ) 操作を使用してバッチを実行します。作業は手動で 25 個のバッチに分割する必要があります。各操作の後に、未処理の項目のリストを受け取るように要求する必要もあります (書き込み操作の中には成功するものもあれば、失敗するものもあります)。その後、それらの未処理項目を後の batch_write_item( ) 操作に再度渡す必要があります。大量のボイラープレートコードがあります。

Table.batch_writer メソッドは、オブジェクトをバッチで書き込むためのコンテキストマネージャーを作成します。一度に 1 つずつ項目を書き込んでいるように見えても、内部的には項目をバッファリングしてバッチで送信するインターフェイスを提供します。また、未処理の項目の再試行も暗黙的に処理されます。

```
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('YourTableName')

movies = # long list of movies in {'pk': 'val', 'sk': 'val', etc} format
with table.batch_writer() as writer:
    for movie in movies:
        writer.put_item(Item=movie)
```

クライアントレイヤーとリソースレイヤーを調べるその他のコード例

次のコードサンプルリポジトリを参照して、クライアントとリソースの両方を使用してさまざまな機能の使用法を調べることもできます。

- AWS 公式シングルアクションのコード例。
- AWS 公式のシナリオ指向コード例。
- コミュニティが管理するシングルアクションのコード例。

Client オブジェクトと Resource オブジェクトがセッションやスレッドとどのように相互作用するかを理解する

- Client は通常、いくつかの高度な機能を除いて、一度作成されるとスレッドセーフです。1 つの Client を作成し、複数のスレッドから同時に呼び出すことができます。これらの例外の詳細については、Boto3 ドキュメントの「Client に関するガイド」を参照してください。
- Resource はスレッドセーフではありません。1 つの Resource オブジェクトを複数のスレッドで使用することはできません。詳細については、Boto3 ドキュメントの「Resource のガイド」を参照してください。
- Session はスレッドセーフではありません。リスクは、Session から Client または Resource を作成することにあります。同じ Session から同時に Client または Resource を構築する 2 つのスレッドは、基盤となる SSL レイヤーで順序外のレスポンスやクラッシュを引き起こす可能性があります。Session から一度に 1 つずつ作成する場合は問題ありません。詳細については、Boto3 ドキュメントの「Session のガイド」を参照してください。

独自のセッションを作成しない場合、`boto3.resource()` と `boto3.client()` は共有デフォルト Session を使用します。これはシングルスレッドのコードでは問題ありませんが、マルチスレッドのコードでは、すべてのスレッドが同じ Session からオブジェクトを構築することになるため、先ほど説明した安全ではないパターンになります。

スレッド間で 1 つの Client を共有する

Client はスレッドセーフであるため、最もシンプルで効率的なアプローチは、メインスレッドで単一の Client を作成し、すべてのワーカーに渡すことです。これらはすべて 1 つの接続プールを共有します。

メインスレッドで 1 回作成され、すべてのスレッドと共有されます。

```
session = boto3.Session()
client = session.client("dynamodb")
```

各スレッドに独自の Resource を付与する

Resource はスレッドセーフではないため、各スレッドには独自の Resource が必要です。スレッド内に新しい Session を作成し、そこから Resource を作成するため、2 つのスレッドが同じ Session から同時に Resource を構築することはありません。

各スレッド内で実行されます。

```
session = boto3.Session()
dynamodb = session.resource("dynamodb")
```

Config オブジェクトのカスタマイズ

Client または Resource オブジェクトを作成するときに、オプションの名前付きパラメータを渡して動作をカスタマイズできます。config という名前のパラメータは、さまざまな機能をアンロックします。これは botocore.client.Config のインスタンスであり、Config のリファレンスドキュメントには、ユーザーが制御できるすべての情報が記載されています。設定ガイドには、概要がわかりやすく書かれています。

> **Note**
>
> これらの動作設定の多くは、Session レベル、AWS 構成ファイル内、または環境変数として変更できます。

タイムアウトの設定

カスタム設定の用途の 1 つは、ネットワークの動作を調整することです。

- connect_timeout (float または int) — 接続を試みたときにタイムアウト例外がスローされるまでの時間 (秒単位)。デフォルト値は 60 秒です。
- read_timeout (float または int) — 接続からの読み取りを試みたときにタイムアウト例外がスローされるまでの時間 (秒単位)。デフォルト値は 60 秒です。

DynamoDB では 60 秒のタイムアウトは過剰です。つまり、一時的なネットワークの不具合により、クライアントが再試行できるようになるまでに 1 分間の遅延が生じます。次のコードはタイムアウトを 10 秒に短縮します。

```
import boto3
from botocore.config import Config

my_config = Config(
   connect_timeout = 10.0,
   read_timeout = 10.0
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

タイムアウトの詳細については、「レイテンシーを考慮した DynamoDB アプリケーションのための AWS Java SDK HTTP リクエスト設定のチューニング」を参照してください。Java SDK には Python よりも多くのタイムアウト設定があることに注意してください。

キープアライブの設定

botocore 1.27.84 以降を使用している場合は、TCP キープアライブを制御することもできます。

- tcp_keepalive (bool) - `True` に設定された場合、新しい接続の作成時に使用される TCP キープアライブソケットオプションを有効にします (デフォルトは `False`)。これは botocore 1.27.84 以降でのみ使用可能です。

TCP キープアライブを `True` に設定すると、平均レイテンシーを減らすことができます。以下は、適切な botocore バージョンを使用している場合に TCP キープアライブを条件付きで true に設定するサンプルコードです。

```
import botocore
import boto3
from botocore.config import Config

required_version = "1.27.84"
current_version = botocore.__version__

my_config = Config(
   connect_timeout = 0.5,
   read_timeout = 0.5
)
```

```
if tuple(map(int, current_version.split("."))) >= tuple(map(int,
 required_version.split("."))):
    my_config = my_config.merge(Config(tcp_keepalive = True))

dynamodb = boto3.resource('dynamodb', config=my_config)
```

> **Note**
>
> TCP キープアライブは HTTP キープアライブとは異なります。TCP キープアライブでは、接続を維持してドロップを即座に検出するために、基盤となるオペレーティングシステムからソケット接続を介して小さなパケットが送信されます。HTTP キープアライブでは、基盤となるソケット上に構築された Web 接続が再利用されます。HTTP キープアライブは boto3 では常に有効になっています。

アイドル状態の接続を維持できる時間には制限があります。接続がアイドル状態で、次のリクエストではすでに確立されている接続を使いたい場合は、定期的に (たとえば 1 分ごとに) リクエストを送信することを検討してください。

再試行の設定

この設定では、希望する再試行動作を指定できる retries という辞書も使用できます。SDK がエラーを受信し、そのエラーが一時的なタイプの場合、SDK 内で再試行が行われます。エラーが内部で再試行された場合 (そして再試行によって最終的に成功の応答が得られた場合)、呼び出し側のコードから見ると、エラーは発生せず、レイテンシーがわずかに増加するだけです。指定できる値は次のとおりです。

- max_attempts — 1 回のリクエストで行われる最大再試行回数を表す整数。たとえば、この値を 2 に設定すると、最初のリクエストから最大 2 回リクエストが再試行されます。この値を 0 に設定すると、最初のリクエスト以降は再試行されません。
- total_max_attempts — 1 回のリクエストで行われる最大再試行合計回数を表す整数。これには最初のリクエストが含まれるため、値が 1 の場合はリクエストが再試行されないことを示します。`total_max_attempts` と `max_attempts` の両方が指定された場合、`total_max_attempts` が優先されます。`total_max_attempts` は、`AWS_MAX_ATTEMPTS` 環境変数と `max_attempts` 設定ファイルの値にマップされるので、`max_attempts` より優先されます。
- mode — botocore が使用すべき再試行モードの種類を表す文字列。次の値を指定できます。

- legacy — デフォルトモード。最初の再試行は 50 ms 待機し、基本係数 2 でエクスポネンシャルバックオフを使用します。DynamoDB の場合、最大試行回数は合計 10 回です (上記でオーバーライドされない限り)。

  > **Note**
  >
  > エクスポネンシャルバックオフでは、最後の試行ではほぼ 13 秒待機します。

- standard — 他の AWS SDK との整合性が高いことから、standard と名付けられています。最初の再試行時には、0 ミリ秒から 1,000 ミリ秒までのランダムな時間だけ待ちます。再試行が必要な場合は、0 ミリ秒から 1,000 ミリ秒までの別のランダムな時間を選択し、その値に 2 を掛けます。さらに再試行が必要な場合は、同じようにランダムな時間を選択し、これに 4 を掛け、試行を続けます。各待機時間の上限は 20 秒です。このモードは、検出された障害条件の数が `legacy` モードよりも多い場合に再試行を実行します。DynamoDB の場合、最大試行回数は合計 3 回です (上記でオーバーライドされない限り)。
- adaptive – standard モードのすべての機能を含みながら、クライアント側の自動スロットリングを含む実験的な再試行モード。適応型レート制限を使用すると、SDK は AWS のサービスの容量をより適切に処理するために、リクエストの送信速度を遅くすることができます。これは暫定モードであり、動作が変わる可能性があります。

これらの再試行モードの拡張定義は、再試行ガイドと SDK リファレンスの「再試行動作」トピックに記載されています。

次の例は、`legacy` 再試行ポリシーを明示的に使用して、合計リクエスト 3 回 (再試行 2 回) までにした例です。

```
import boto3
from botocore.config import Config

my_config = Config(
   connect_timeout = 1.0,
   read_timeout = 1.0,
   retries = {
     'mode': 'legacy',
     'total_max_attempts': 3
   }
)
dynamodb = boto3.resource('dynamodb', config=my_config)
```

DynamoDB は可用性が高く、レイテンシーが低いシステムであるため、組み込みの再試行ポリシーで許可されているよりも積極的に再試行の速度を上げる必要がある場合があります。boto3 に頼って暗黙的な再試行を行うのではなく、最大試行回数を 0 に設定し、自分で例外をキャッチし、必要に応じて独自のコードから再試行することで、独自の再試行ポリシーを実装できます。

独自の再試行ポリシーを管理する場合は、スロットルとエラーを区別しておきましょう。

- スロットル (`ProvisionedThroughputExceededException` または `ThrottlingException` で示される) は、DynamoDB テーブルまたはパーティションの読み取り容量または書き込み容量を超過したことを通知する正常なサービスを示します。1 ミリ秒が経過するごとに、読み取りまたは書き込み容量が少しずつ増えるため、すぐに (50 ミリ秒ごとなど) 再試行して、新しく解放された容量へのアクセスを試みることができます。スロットルについては、エクスポネンシャルバックオフは特に必要ありません。軽量で DynamoDB が返しやすく、リクエストごとの課金も発生しないためです。エクスポネンシャルバックオフでは、すでに最長の待機時間が過ぎたクライアントスレッドの遅延をさらに長くしていくため、統計的に p50 と p99 が外側に広がります。
- エラー (`InternalServerError` または `ServiceUnavailable` などによって示される) は、サービスに一時的な問題があることを示します。これはテーブル全体の場合もあれば、読み取り元または書き込み先のパーティションだけに関する場合もあります。エラーが発生した場合は、再試行の前に少し長く一時停止し (250 ミリ秒や 500 ミリ秒など)、ジッターを使って再試行をずらすことができます。

すべての AWS SDK のデフォルトの再試行動作に対する今後の変更については、AWS デベロッパーツールブログの「AWS SDK およびツールの更新された再試行動作の発表」を参照してください。Python に固有の変更の詳細については、GitHub ウェブサイトの「再試行動作更新の説明 (#4789)」を参照してください。

最大プール接続数の設定

最後に、この構成では接続プールのサイズを制御できます。

- max_pool_connections (int) — 接続プールに保持できる最大接続数。この値が設定されていない場合、デフォルト値の 10 が使用されます。

このオプションは、プールして再利用できるように保持する HTTP 接続の最大数を制御します。Client ごとに異なるプールが保持されます。Client のスレッドが接続を必要としており、プールが空の場合、新しい接続が作成されます。スレッドが使用後にプールへ接続を返そうとした際、プールがいっぱいであることがわかった場合、接続は閉じられます。1 つの Client に対して 10 を超える

スレッドが発生することが予想される場合は、スレッド数に合わせてプールサイズを増やすことを検討してください。

```
import boto3
from botocore.config import Config

my_config = Config(
   max_pool_connections = 20
)

session = boto3.Session()

# One client with a larger connection pool
client = session.client("dynamodb", config=my_config)
```

各 Resource にも接続プールがありますが、Resource オブジェクトはスレッドセーフではないため、スレッド間で共有しないでください。各スレッドに独自の Resource を付与します。

エラー処理

AWSBoto3 では サービスの例外がすべて静的に定義されているわけではありません。これは、AWS サービスのエラーや例外は大きく異なり、変更される可能性があるためです。Boto3 はすべてのサービス例外を ClientError としてラップし、詳細を構造化された JSON として公開します。例えば、エラーレスポンスは以下のような構造になっているかもしれません。

```
{
    'Error': {
        'Code': 'SomeServiceException',
        'Message': 'Details/context around the exception or error'
    },
    'ResponseMetadata': {
        'RequestId': '1234567890ABCDEF',
        'HostId': 'host ID data will appear here as a hash',
        'HTTPStatusCode': 400,
        'HTTPHeaders': {'header metadata key/values will appear here'},
        'RetryAttempts': 0
    }
}
```

次のコードは、すべての ClientError 例外をキャッチし、Error 内の Code の文字列値を調べて実行するアクションを決定します。

```
import botocore
import boto3

dynamodb = boto3.client('dynamodb')

try:
    response = dynamodb.put_item(...)

except botocore.exceptions.ClientError as err:
    print('Error Code: {}'.format(err.response['Error']['Code']))
    print('Error Message: {}'.format(err.response['Error']['Message']))
    print('Http Code: {}'.format(err.response['ResponseMetadata']['HTTPStatusCode']))
    print('Request ID: {}'.format(err.response['ResponseMetadata']['RequestId']))

    if err.response['Error']['Code'] in ('ProvisionedThroughputExceededException',
 'ThrottlingException'):
        print("Received a throttle")
    elif err.response['Error']['Code'] == 'InternalServerError':
        print("Received a server error")
    else:
        raise err
```

一部の (すべてではない) 例外コードは最上位クラスとして実装されています。これらを直接処理することもできます。Client インターフェイスを使用する場合、これらの例外はクライアントに動的に入力され、以下のようにクライアントインスタンスを使用してこれらの例外をキャッチします。

```
except ddb_client.exceptions.ProvisionedThroughputExceededException:
```

Resource インターフェイスを使用しているときには、以下のように、.meta.client を使用してリソースから基盤となる Client までトラバースして例外にアクセスする必要があります。

```
except ddb_resource.meta.client.exceptions.ProvisionedThroughputExceededException:
```

マテリアライズド例外タイプのリストを確認するには、リストを動的に生成できます。

```
ddb = boto3.client("dynamodb")
print([e for e in dir(ddb.exceptions) if e.endswith('Exception') or
 e.endswith('Error')])
```

条件式を使用して書き込み操作を行う場合、式が失敗した場合にその項目の値をエラーレスポンスで返すようにリクエストできます。

```
try:
    response = table.put_item(
        Item=item,
        ConditionExpression='attribute_not_exists(pk)',
        ReturnValuesOnConditionCheckFailure='ALL_OLD'
    )
except table.meta.client.exceptions.ConditionalCheckFailedException as e:
    print('Item already exists:', e.response['Item'])
```

エラー処理と例外について詳しくは、以下を参照してください。

- エラー処理技術の詳細については、エラー処理に関する boto3 ガイドを参照してください。
- DynamoDB 開発者ガイドのプログラミングエラーに関するセクションには、発生する可能性のあるエラーが記載されています。
- API リファレンスの「一般的なエラー」セクション。
- 各 API 操作のドキュメントには、呼び出しによって発生する可能性のあるエラー (例: BatchWriteItem) が記載されています。

ログ記録

boto3 ライブラリは Python の組み込みロギングモジュールと統合されており、セッション中に何が起こるかを追跡できます。ロギングレベルを制御するには、ロギングモジュールを設定します。

```
import logging

logging.basicConfig(level=logging.INFO)
```

これにより、INFO 以上のレベルのメッセージを記録するようにルートロガーが設定されます。レベルよりも重大度の低いロギングメッセージは無視されます。ロギングレベルには、DEBUG、INFO、WARNING、ERROR、および CRITICAL があります。デフォルトは WARNING です。

boto3 のロガーは階層構造になっています。ライブラリはいくつかの異なるロガーを使用しており、それぞれがライブラリの異なる部分に対応しています。それぞれの動作を個別に制御できます。

- boto3: boto3 モジュールのメインロガー。
- botocore: botocore パッケージのメインロガー。
- botocore.auth: リクエストの AWS 署名作成を記録するために使用されます。

- botocore.credentials: 認証情報の取得と更新のプロセスを記録するために使用されます。
- botocore.endpoint: ネットワーク経由で送信される前に、リクエストの作成を記録するために使用されます。
- botocore.hooks: ライブラリでトリガーされたイベントの記録に使用されます。
- botocore.loaders: AWS サービスモデルの一部がロードされたときの記録に使用されます。
- botocore.parsers: AWS サービスレスポンスを解析する前に記録するために使用されます。
- botocore.retryhandler: AWS サービスリクエストの再試行の処理を記録するために使用されます (レガシーモード)。
- botocore.retries.standard: AWS サービスリクエストの再試行の処理を記録するために使用されます (standard または adaptive モード)。
- botocore.utils: ライブラリ内のさまざまなアクティビティを記録するために使用されます。
- botocore.waiter: 特定の状態になるまで AWS サービスにポーリングするウェイターの機能を口グに記録するために使用されます。

他のライブラリもログに記録されます。内部的には、boto3 は HTTP 接続処理にサードパーティの urllib3 を使用しています。レイテンシーが重要な場合は、ログを監視して、urllib3 が新しい接続を確立したり、アイドル状態の接続を閉じたりするタイミングを確認することで、プールが十分に利用されているかどうかを確認できます。

- urllib3.connectionpool: 接続プールの処理イベントを記録するために使用されます。

以下のコードスニペットは、ほとんどのロギングを `INFO` に設定し、エンドポイントと接続プールのアクティビティの記録には `DEBUG` に設定しています。

```
import logging

logging.getLogger('boto3').setLevel(logging.INFO)
logging.getLogger('botocore').setLevel(logging.INFO)
logging.getLogger('botocore.endpoint').setLevel(logging.DEBUG)
logging.getLogger('urllib3.connectionpool').setLevel(logging.DEBUG)
```

イベントフック

Botocore は実行中のさまざまな段階でイベントを発生させます。これらのイベントのハンドラーを登録すると、イベントが発生するたびにハンドラーが呼び出されます。これにより、内部を変更しなくても botocore の動作を拡張できます。

例えば、アプリケーションの DynamoDB テーブルで PutItem 操作が呼び出されるたびに記録しておきたいとしましょう。'provide-client-params.dynamodb.PutItem' イベントに登録すると、関連する Session で PutItem 操作が呼び出されるたびにキャッチしてログに記録できます。例を示します。

```
import boto3
import botocore
import logging

def log_put_params(params, **kwargs):
    if 'TableName' in params and 'Item' in params:
        logging.info(f"PutItem on table {params['TableName']}: {params['Item']}")

logging.basicConfig(level=logging.INFO)

session = boto3.Session()
event_system = session.events

# Register our interest in hooking in when the parameters are provided to PutItem
event_system.register('provide-client-params.dynamodb.PutItem', log_put_params)

# Now, every time you use this session to put an item in DynamoDB,
# it will log the table name and item data.
dynamodb = session.resource('dynamodb')
table = dynamodb.Table('YourTableName')
table.put_item(
    Item={
        'pk': '123',
        'sk': 'cart#123',
        'item_data': 'YourItemData',
        # ... more attributes ...
    }
)
```

ハンドラー内で、パラメータをプログラムで操作して動作を変更することもできます。

```
params['TableName'] = "NewTableName"
```

イベントについて詳しくは、イベントに関する botocore ドキュメントと、イベントに関する boto3 ドキュメントを参照してください。

ページネーションとページネーター

Query や Scan などの一部のリクエストでは、1 回のリクエストで返されるデータのサイズが制限され、後続のページを取得するにはリクエストを繰り返し行う必要があります。

各ページで読み取られる項目の最大数は、limit パラメータを使用して制御できます。例えば、最後の 10 項目だけが必要な場合、limit を使用して最後の 10 項目だけを取得できます。制限は、フィルタリングが適用される前にテーブルから読み取られる量であることに注意してください。フィルタリング後に正確に 10 を指定する方法はありません。事前にフィルター処理される回数を制御し、実際に 10 個を取得した時点でクライアント側で確認することしかできません。制限にかかわらず、すべての応答の最大サイズは常に 1 MB です。

応答に LastEvaluatedKey が含まれている場合、それは件数またはサイズの制限に達したために応答が終了したことを示します。キーはレスポンスで評価された最後のキーです。この LastEvaluatedKey を取得して , フォローアップコールに ExclusiveStartKey として渡すと、その開始点から次のチャンクを読み取ることができます。LastEvaluatedKey が返されないということは、Query または Scan に一致するアイテムがもうないということです。

以下は、1 ページあたり最大 100 個の項目を読み込み、すべての項目が読み込まれるまでループする単純な例です (Resource インターフェイスを使用していますが、Client インターフェイスでも同じパターンです)。

```
import boto3
from boto3.dynamodb.conditions import Key

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('YourTableName')

query_params = {
    'KeyConditionExpression': Key('pk').eq('123') & Key('sk').gt(1000),
    'Limit': 100
}

while True:
    response = table.query(**query_params)

    # Process the items however you like
    for item in response['Items']:
        print(item)

    # No LastEvaluatedKey means no more items to retrieve
```

```
    if 'LastEvaluatedKey' not in response:
        break

    # If there are possibly more items, update the start key for the next page
    query_params['ExclusiveStartKey'] = response['LastEvaluatedKey']
```

便宜上、boto3 はページネーターを使ってこれを行うことができます。ただし、Client インターフェイスでのみ機能します。以下はページネーターを使うように書き直されたコードです。

```
import boto3

dynamodb = boto3.client('dynamodb')

paginator = dynamodb.get_paginator('query')

query_params = {
    'TableName': 'YourTableName',
    'KeyConditionExpression': 'pk = :pk_val AND sk > :sk_val',
    'ExpressionAttributeValues': {
        ':pk_val': {'S': '123'},
        ':sk_val': {'N': '1000'},
    },
    'Limit': 100
}

page_iterator = paginator.paginate(**query_params)

for page in page_iterator:
    # Process the items however you like
    for item in page['Items']:
        print(item)
```

詳細については、ページネーターに関するガイドと DynamoDB.Paginator.Query の API リファレンスを参照してください。

> **Note**
>
> ページネーターには、MaxItems、StartingToken、および PageSize という名前の独自の構成設定もあります。DynamoDB でページ分割を行う場合は、これらの設定は無視してください。

ウェイター

ウェイターは、処理が完了するのを待ってから次の処理を進めることができます。現在のところ、サポートしているのはテーブルが作成または削除されるのを待つことだけです。バックグラウンドでは、ウェイター操作は 20 秒ごとに最大 25 回までチェックを行います。これは自分で行うこともできますが、オートメーションを書くときはウェイターを使うのが便利です。

次のコードは、特定のテーブルが作成されるまで待機する方法を示しています。

```
# Create a table, wait until it exists, and print its ARN
response = client.create_table(...)
waiter = client.get_waiter('table_exists')
waiter.wait(TableName='YourTableName')
print('Table created:', response['TableDescription']['TableArn'])
```

詳細については、ウェイターに関するガイドとウェイターに関するリファレンスを参照してください。

JavaScript による Amazon DynamoDB のプログラミング

このガイドは、JavaScript で Amazon DynamoDB を使用したいと考えているプログラマーを対象としています。AWS SDK for JavaScript、利用可能な抽象化レイヤー、接続の設定、エラー処理、再試行ポリシーの定義、キープアライブの管理などについて説明します。

トピック

- AWS SDK for JavaScript について
- AWS SDK for JavaScript V3 を使用する
- JavaScript のドキュメントを参照する
- 抽象化レイヤー
- marshall ユーティリティ関数を使用する
- 項目の読み込み
- 条件付きの書き込み
- ページ分割
- 構成を指定する
- ウェイター

- エラー処理
- ログ記録
- 考慮事項

AWS SDK for JavaScript について

AWS SDK for JavaScript を使用すると、ブラウザスクリプトまたは Node.js のいずれかで AWS のサービス にアクセスできます。このドキュメントでは、最新バージョンの SDK (V3) を主に取り上げます。AWS SDK for JavaScript V3 は、AWS が管理するオープンソースのプロジェクトであり、GitHub でホストされています。Issue や Feature のリクエストは公開されており、GitHub リポジトリの [Issues] ページからアクセスできます。

JavaScript V2 は V3 と似ていますが、構文が違います。V3 の方がモジュール性が高いため、依存関係をより小さくして配布することが容易で、TypeScript のサポートも充実しています。最新バージョンの SDK を使用することをお勧めします。

AWS SDK for JavaScript V3 を使用する

Node Package Manager を使用して SDK を Node.js アプリケーションに追加できます。以下の例は、DynamoDB の操作に一般的に使われる SDK パッケージの追加方法を示しています。

- `npm install @aws-sdk/client-dynamodb`
- `npm install @aws-sdk/lib-dynamodb`
- `npm install @aws-sdk/util-dynamodb`

パッケージをインストールすると、package.json プロジェクトファイルの依存関係セクションに参照が追加されます。新しい ECMAScript モジュール構文を使用することもできます。これら 2 つのアプローチの詳細については、「考慮事項」セクションを参照してください。

JavaScript のドキュメントを参照する

まずは、JavaScript のドキュメントを確認しましょう。以下のリソースを参照してください。

- JavaScript の主要ドキュメントについては、「デベロッパーガイド」を参照してください。インストール手順は「Setting up」セクションに記載されています。
- API リファレンスドキュメントを参照し、使用可能なすべてのクラスとメソッドを確認してください。

- SDK for JavaScript は DynamoDB 以外にも数多くの AWS のサービス をサポートしています。以下の手順にそって、DynamoDB の特定の API カバレッジを調べてください。

  1. [Services] から [DynamoDB and Libraries] を選択します。これは、低レベルクライアントをドキュメント化したものです。
  2. lib-dynamodb を選択します。これは、高レベルクライアントをドキュメント化したものです。これら 2 つのクライアントは、2 つの異なる抽象化レイヤーを表しています。任意で選んで使用できます。抽象レイヤーの詳細については、以下のセクションを参照してください。

抽象化レイヤー

SDK for JavaScript V3 には、低レベルのクライアント (`DynamoDBClient`) と高レベルのクライアント (`DynamoDBDocumentClient`) があります。

トピック

- 低レベルクライアント (DynamoDBClient)
- 高レベルクライアント (DynamoDBDocumentClient)

低レベルクライアント (**`DynamoDBClient`**)

低レベルクライアントでは、基盤のワイヤプロトコルが別段抽象化されません。通信をあらゆる側面から完全に制御できますが、抽象化されていないため、項目定義の提供などの操作は、DynamoDB JSON 形式を使って行う必要があります。

以下の例に示すように、この形式ではデータ型を明示的に指定する必要があります。S は文字列値を示し、N は数値を示します。ネットワーク上の数値は、精度が損なわれないように、必ず数値型としてタグ付けされた文字列として送信されます。低レベルの API コールには、`PutItemCommand` や `GetItemCommand` などの命名規則があります。

次の例では、低レベルクライアントを使用し、DynamoDB JSON で `Item` を定義しています。

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
  const params = {
    TableName: "products",
```

```
    Item: {
      "id": { S: "Product01" },
      "description": { S: "Hiking Boots" },
      "category": { S: "footwear" },
      "sku": { S: "hiking-sku-01" },
      "size": { N: "9" }
    }
  };

  try {
    const data = await client.send(new PutItemCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}
addProduct();
```

高レベルクライアント (**DynamoDBDocumentClient**)

高レベルの DynamoDB ドキュメントクライアントには、データを手動でマーシャリングする必要がない、標準の JavaScript オブジェクトを使用して直接読み書きできるなど、便利な機能が組み込まれています。lib-dynamodb のドキュメントには、利点が一覧で紹介されています。

DynamoDBDocumentClient をインスタンス化するには、まず、低レベルの DynamoDBClient を構築し、それを DynamoDBDocumentClient でラップします。関数の命名規則は 2 つのパッケージ間で若干異なります。例えば、低レベルでは PutItemCommand を使用し、高レベルでは PutCommand を使用します。名前が違えば、両方の関数セットが同じコンテキストで共存可能です。つまり、同一スクリプト内で両方を組み合わせることができます。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, PutCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function addProduct() {
  const params = {
    TableName: "products",
    Item: {
      id: "Product01",
```

```
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
      size: 9,
    },
  };

  try {
    const data = await docClient.send(new PutCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

addProduct();
```

GetItem、Query、Scan などの API オペレーションを使用して項目を読み取る場合の使用パターンは同じです。

marshall ユーティリティ関数を使用する

低レベルクライアントを使用して、データ型を自分でマーシャリングまたはアンマーシャリングできます。ユーティリティパッケージ util-dynamodb には、JSON を受け入れて DynamoDB JSON を生成する marshall() ユーティリティ関数と、その逆を行う unmarshall() 関数があります。次の例では、低レベルクライアントを使用し、marshall() を呼び出してデータマーシャリングを処理しています。

```
const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");
const { marshall } = require("@aws-sdk/util-dynamodb");

const client = new DynamoDBClient({});

async function addProduct() {
  const params = {
    TableName: "products",
    Item: marshall({
      id: "Product01",
      description: "Hiking Boots",
      category: "footwear",
      sku: "hiking-sku-01",
      size: 9,
```

```
    }),
  };

  try {
    const data = await client.send(new PutItemCommand(params));
  } catch (error) {
    console.error("Error:", error);
  }
}
addProduct();
```

項目の読み込み

DynamoDB から項目を 1 つ読み込むには、GetItem API オペレーションを使用します。PutItem コマンドと同様に、低レベルのクライアントまたは高レベルのドキュメントクライアントのいずれかを選んで使用できます。以下の例は、高レベルのドキュメントクライアントを使用して項目を取得する方法を示しています。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, GetCommand } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function getProduct() {
  const params = {
    TableName: "products",
    Key: {
      id: "Product01",
    },
  };

  try {
    const data = await docClient.send(new GetCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

getProduct();
```

Query API オペレーションを使用して、複数の項目を読み込みます。低レベルのクライアントまたはドキュメントクライアントを使用できます。以下の例では、高レベルのドキュメントクライアントを使用しています。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  QueryCommand,
} = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({});

const docClient = DynamoDBDocumentClient.from(client);

async function productSearch() {
  const params = {
    TableName: "products",
    IndexName: "GSI1",
    KeyConditionExpression: "#category = :category and begins_with(#sku, :sku)",
    ExpressionAttributeNames: {
      "#category": "category",
      "#sku": "sku",
    },
    ExpressionAttributeValues: {
      ":category": "footwear",
      ":sku": "hiking",
    },
  };

  try {
    const data = await docClient.send(new QueryCommand(params));
    console.log('result : ' + JSON.stringify(data));
  } catch (error) {
    console.error("Error:", error);
  }
}

productSearch();
```

条件付きの書き込み

DynamoDB の書き込みオペレーションでは、論理条件式を指定できます。条件式が true と評価されないと、書き込みは続行されません。条件の評価が true にならなかった場合は、例外が生成されます。条件式では、該当する項目が既に存在するかどうかや、その属性が特定の制約に一致するかどうかを確認できます。

```
ConditionExpression = "version = :ver AND size(VideoClip) < :maxsize"
```

条件式が失敗した場合は、ReturnValuesOnConditionCheckFailure を使用して、条件を満たさなかった項目をエラーレスポンスに含めるようにリクエストできるため、問題の原因究明に役立ちます。詳細については、「Handle conditional write errors in high concurrency scenarios with Amazon DynamoDB」を参照してください。

```
try {
      const response = await client.send(new PutCommand({
          TableName: "YourTableName",
          Item: item,
          ConditionExpression: "attribute_not_exists(pk)",
          ReturnValuesOnConditionCheckFailure: "ALL_OLD"
      }));
  } catch (e) {
      if (e.name === 'ConditionalCheckFailedException') {
          console.log('Item already exists:', e.Item);
      } else {
          throw e;
      }
  }
```

JavaScript SDK V3 のその他の使用法を示す追加のコード例は、JavaScript SDK V3 ドキュメントと DynamoDB-SDK-Examples GitHub リポジトリでご覧ください。

ページ分割

トピック

- 便利な paginateScan メソッドを使用する

Scan や Query などの読み取りリクエストでは、データセット内の複数の項目が返される場合があります。Limit パラメータを指定して Scan や Query を実行した場合は、システムで多数の項目が

全部読み取られた後、一部のレスポンスが送信されます。追加の項目を取得するには、ページ分割が必要です。

システムは、1 回のリクエストにつき最大 1 MB のデータのみを読み取ります。Filter 式を含めた場合も、システムは最大 1 MB のデータをディスクから読み取りますが、その 1 MB の中から、指定したフィルターに一致した項目だけを返します。フィルターオペレーションでは 1 ページあたりに返される項目数が 0 個になる場合もありますが、それでも、さらにページ分割をしないと、最後まで検索できません。

データの取得を継続するには、レスポンスで LastEvaluatedKey を探し、それを後続のリクエストの ExclusiveStartKey パラメータに指定する必要があります。これが、以下の例に示すように、ブックマークの役割を果たします。

> **Note**
>
> この例では、1 回目の反復で null lastEvaluatedKey が ExclusiveStartKey として渡されています。これは許容されています。

LastEvaluatedKey を使用する例:

```
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScan() {
  let lastEvaluatedKey;
  let pageCount = 0;

  do {
    const params = {
      TableName: "products",
      ExclusiveStartKey: lastEvaluatedKey,
    };

    const response = await client.send(new ScanCommand(params));
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, response.Items);
    lastEvaluatedKey = response.LastEvaluatedKey;
  } while (lastEvaluatedKey);
}
```

```
paginatedScan().catch((err) => {
  console.error(err);
});
```

便利な paginateScan メソッドを使用する

SDK には、paginateScan と paginateQuery という便利なメソッドがあります。これらは、ページ分割を自動で行い、繰り返しのリクエストをバックグラウンドで処理してくれます。標準の Limit パラメータを使用して、1 回のリクエストで読み取る項目の最大数を指定します。

```
const { DynamoDBClient, paginateScan } = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({});

async function paginatedScanUsingPaginator() {
  const params = {
    TableName: "products",
    Limit: 100
  };

  const paginator = paginateScan({client}, params);

  let pageCount = 0;

  for await (const page of paginator) {
    pageCount++;
    console.log(`Page ${pageCount}, Items:`, page.Items);
  }
}

paginatedScanUsingPaginator().catch((err) => {
  console.error(err);
});
```

> **Note**
>
> テーブルが小さくない限り、テーブル全体のスキャンを定期的に実行することは、アクセスパターンとして推奨されません。

構成を指定する

トピック

- タイムアウトの設定
- キープアライブの設定
- 再試行の設定

DynamoDBClient を設定する際に、構成オブジェクトをコンストラクタに渡すことで、さまざまな構成オーバーライドを指定できます。例えば、接続先のリージョン (呼び出し側のコンテキストが把握していない場合) や、使用するエンドポイント URL を指定できます。開発目的で DynamoDB Local インスタンスをターゲットにする場合に便利です。

```
const client = new DynamoDBClient({
  region: "eu-west-1",
  endpoint: "http://localhost:8000",
});
```

タイムアウトの設定

DynamoDB では、クライアント/サーバー通信に HTTPS を使用します。HTTP レイヤーの一部の側面は、NodeHttpHandler オブジェクトを指定することで制御できます。例えば、主要なタイムアウト値である connectionTimeout や requestTimeout を調整できます。connectionTimeout は、クライアントが接続を試みる際の最長待機時間 (ミリ秒単位) です。この時間内に確立できない場合は、接続を断念します。

requestTimeout では、リクエストが送信されてからクライアントがレスポンスを待機する時間 (ミリ秒単位) を定義します。いずれもデフォルト値は 0 です。その場合、タイムアウトは無効になり、レスポンスが届かなければクライアントは制限なく待ち続けることになります。ネットワークに問題が発生した場合にリクエストがエラーになり、クライアントが新しいリクエストを開始できるように、妥当なタイムアウト値を設定しておく必要があります。例えば、10 秒です。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";

const requestHandler = new NodeHttpHandler({
  connectionTimeout: 10000,
  requestTimeout: 10000,
});
```

```
const client = new DynamoDBClient({
  requestHandler
});
```

> **Note**
>
> この例では Smithy をインポートしています。Smithy はサービスと SDK を定義するための言語で、AWS が管理し、オープンソースで公開しています。

タイムアウト値を設定するほかに、最大ソケット数を設定し、オリジンあたりの同時接続数を増やすことができます。デベロッパーガイドでは、`maxSockets` パラメータの設定について詳しく解説しています。アプリケーションの同時実行数 (処理中の同時リクエスト数) が定期的に 50 に近づくか超える場合は、この値を増やすことを検討してください。

キープアライブの設定

HTTPS を使用する場合、最初のリクエストでは、安全な接続を確立するために何往復かの通信が必要になります。HTTP キープアライブを使用すると、すでに確立された接続を後続のリクエストで再利用できるため、リクエストの効率が上がり、レイテンシーが短縮されます。JavaScript V3 では、HTTP キープアライブがデフォルトで有効になっています。

アイドル状態の接続を維持できる時間には制限があります。接続がアイドル状態になるが、すでに確立されている接続を次のリクエストで利用したい場合は、定期的に (たとえば 1 分ごとに) リクエストを送信することを検討してください。

> **Note**
>
> SDK の旧バージョン V2 では、キープアライブはデフォルトで無効になっているため、各接続は使用後すぐに切断されます。V2 を使用している場合は、この設定をオーバーライドできます。

再試行の設定

SDK がエラーレスポンスを受信し、そのエラーを SDK が再開可能であると判断した場合 (スロットリング例外や一時的なサービス例外など)、再試行されます。呼び出し側にはわからない形で行われますが、リクエスト成功までの時間が長引くという症状は現れます。

SDK for JavaScript V3 は、デフォルトではリクエストを合計 3 回行います。それで成功しなければそれ以上は再試行せず、呼び出し側のコンテキストにエラーを渡します。こうした再試行の回数と頻度を調整できます。

DynamoDBClient コンストラクタには、試行回数を制限する maxAttempts を設定できます。以下の例では、その値がデフォルトの 3 から合計 5 に引き上げられています。0 または 1 に設定した場合、自動再試行は不要であり、再開可能なエラーはキャッチブロック内で手動で処理するという意味になります。

```
const client = new DynamoDBClient({
  maxAttempts: 5,
});
```

また、再試行のタイミングも、カスタムの再試行戦略で制御できます。その場合は、util-retry ユーティリティパッケージをインポートし、現在の再試行が何回目かに応じて再試行の間隔を計算するカスタムのバックオフ関数を作成します。

以下の例では、1 回目の試行が失敗した場合は、15、30、90、360 ミリ秒と徐々に遅延を増やしながら最大 5 回試行するように指定しています。カスタムのバックオフ関数 calculateRetryBackoff は、再試行回数 (初回の再試行が 1 で、以降増分) に応じて遅延を計算し、該当するリクエストで待機するミリ秒数を返します。

```
const { ConfiguredRetryStrategy } = require("@aws-sdk/util-retry");

const calculateRetryBackoff = (attempt) => {
  const backoffTimes = [15, 30, 90, 360];
  return backoffTimes[attempt - 1] || 0;
};

const client = new DynamoDBClient({
  retryStrategy: new ConfiguredRetryStrategy(
    5, // max attempts.
    calculateRetryBackoff // backoff function.
  ),
});
```

すべての AWS SDK のデフォルトの再試行動作に対する今後の変更については、AWS デベロッパーツールブログの「AWS SDK およびツールの更新された再試行動作の発表」を参照してください。JavaScript v3 の正確な詳細については、GitHub ウェブサイトの「JavaScript v3 retry behavior update (issue #8037)」を参照してください。

ウェイター

DynamoDB クライアントには、便利なウェーター関数が 2 つあります。テーブルの作成、変更、削除時に、テーブルの変更が完了するまでコードの処理を待機させたい場合に、これらの関数を使用できます。例えば、テーブルをデプロイして waitUntilTableExists 関数を呼び出すと、テーブルが ACTIVE になるまでコードがブロックされます。ウェーターは内部的に 20 秒ごとに describe-table を実行し、DynamoDB サービスをポーリングします。

```
import {waitUntilTableExists, waitUntilTableNotExists} from "@aws-sdk/client-dynamodb";

… <create table details>

const results = await waitUntilTableExists({client: client, maxWaitTime: 180},
 {TableName: "products"});
if (results.state == 'SUCCESS') {
  return results.reason.Table
}
console.error(`${results.state} ${results.reason}`);
```

waitUntilTableExists 関数は、describe-table コマンドを実行でき、テーブルのステータスが ACTIVE と表示された場合にのみ制御を返します。そのため、waitUntilTableExists を使用して、テーブルの作成や、GSI インデックスの追加などの変更の完了を待つことができます。こうした変更には時間がかかり、終わってからテーブルが ACTIVE ステータスに戻ります。

エラー処理

ここで最初に紹介した数例では、すべてのエラーを網羅してキャッチしています。しかし、実際のアプリケーションでは、さまざまなエラータイプを見分け、より緻密なエラー処理を実装することが重要です。

DynamoDB エラーレスポンスには、エラーの名前を含むメタデータが含まれています。エラーをキャッチし、エラー条件の文字列名候補と照合して、処理方法を決定できます。サーバー側のエラーについては、@aws-sdk/client-dynamodb パッケージによってエクスポートされたエラータイプで instanceof 演算子を利用して、エラー処理を効率的に管理できます。

こうしたエラーは、再試行回数をすべて使い切って初めて露呈する点に注意が重要です。エラーが再試行され、最終的に呼び出しが成功した場合、コードの観点ではエラーは発生せず、レイテンシーが若干長引いただけになります。再試行は、スロットルリクエストやエラーリクエストなど、失敗したリクエストとして Amazon CloudWatch のグラフに表示されます。クライアントは再試行回数が最大

回数に達すると、例外を生成します。つまり、クライアントはそれ以上再試行しないということがわかります。

以下のスニペットは、エラーをキャッチし、返されたエラーのタイプに基づいてアクションを実行します。

```
import {
  ResourceNotFoundException,
  ProvisionedThroughputExceededException,
  DynamoDBServiceException,
} from "@aws-sdk/client-dynamodb";

try {
  await client.send(someCommand);
} catch (e) {
    if (e instanceof ResourceNotFoundException) {
      // Handle ResourceNotFoundException
    } else if (e instanceof ProvisionedThroughputExceededException) {
      // Handle ProvisionedThroughputExceededException
    } else if (e instanceof DynamoDBServiceException) {
      // Handle DynamoDBServiceException
    } else {
      // Other errors such as those from the SDK
      if (e.name === "TimeoutError") {
        // Handle SDK TimeoutError.
      } else {
        // Handle other errors.
      }
    }
}
```

一般的なエラー文字列については、「DynamoDB デベロッパーガイド」の「the section called “エラー処理”」を参照してください。特定の API コールで発生する可能性のある具体的なエラーについては、Query API のドキュメントなど、該当する API コールのドキュメントに記載されています。

エラーのメタデータには、エラーによって異なりますが、追加のプロパティが含まれます。TimeoutError の場合、以下に示すように、試行回数と totalRetryDelay がメタデータに含まれます。

```
{
  "name": "TimeoutError",
  "$metadata": {
```

```
    "attempts": 3,
    "totalRetryDelay": 199
  }
}
```

独自の再試行ポリシーを管理する場合は、スロットルとエラーを区別しておきましょう。

- スロットル (ProvisionedThroughputExceededException または ThrottlingException で示される) は、DynamoDB テーブルまたはパーティションの読み取り容量または書き込み容量を超過したことを通知する正常なサービスを示します。1 ミリ秒が経過するごとに、読み取りまたは書き込みの容量が少しずつ増えるため、すぐに (50 ミリ秒ごとなど) 再試行して、新しく解放された容量へのアクセスを試みることができます。

  スロットルについては、エクスポネンシャルバックオフは特に必要ありません。軽量で DynamoDB が返しやすく、リクエストごとの課金も発生しないためです。エクスポネンシャルバックオフでは、すでに最長の待機時間が過ぎたクライアントスレッドの遅延をさらに長くしていくため、統計的に p50 と p99 が外側に広がります。
- エラー (InternalServerError や ServiceUnavailable などで示される) が発生した場合は、サービス (テーブル全体、または読み取り/書き込み先のパーティションなど) に一時的な問題があります。エラーについては、再試行する前に比較的長く (250 ミリ秒や 500 ミリ秒など) 一時停止したり、ジッターを追加して再試行をずらしたりできます。

ログ記録

ログ記録を有効にすると、SDK による処理内容を詳しく把握できます。以下の例に示すように、DynamoDBClient にパラメータを設定できます。ステータスコードや消費容量などのメタデータを含む詳細なログ情報がコンソールに表示されます。コードをターミナルウィンドウでローカルに実行した場合は、そのウィンドウにログが表示されます。AWS Lambda でコードを実行した場合、Amazon CloudWatch Logs が設定されていれば、コンソール出力がそのログに書き込まれます。

```
const client = new DynamoDBClient({
  logger: console
});
```

また、内部 SDK アクティビティにフックして、特定のイベントが発生したときにカスタムのログ記録を実行することもできます。以下の例では、クライアントの middlewareStack を使用して、各リクエストを SDK から送信された時点でインターセプトし、発生時にログに記録します。

```
const client = new DynamoDBClient({});

client.middlewareStack.add(
  (next) => async (args) => {
    console.log("Sending request from AWS SDK", { request: args.request });
    return next(args);
  },
  {
    step: "build",
    name: "log-ddb-calls",
  }
);
```

MiddlewareStack は、SDK の動作を観察して制御するための強力なフックを提供します。詳細については、ブログ「Introducing Middleware Stack in Modular AWS SDK for JavaScript」を参照してください。

考慮事項

プロジェクトに AWS SDK for JavaScript を実装する際には、さらに以下の点を考慮する必要があります。

モジュールシステム

SDK は CommonJS と ES (ECMAScript) の 2 つのモジュールシステムに対応しています。CommonJS は `require` 関数を使用し、ES は `import` キーワードを使用します。

1. CommonJS – `const { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");`
2. ES (ECMAScript) – `import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";`

プロジェクトタイプは package.json ファイルの type セクションで指定され、使用するモジュールシステムを決定します。デフォルトは CommonJS です。`"type": "module"` を使用して、ES プロジェクトを指定します。CommonJS パッケージ形式を使用する既存の Node.JS プロジェクトがある場合でも、関数ファイルに .mjs 拡張子の付いた名前を指定することで、より新しい SDK V3 Import 構文を使用して関数を追加できます。そうすることで、コードファイルを ES (ECMAScript) として扱うことができます。

非同期オペレーション

コールバックとプロミスを使用して DynamoDB オペレーションの結果を処理するサンプルコードが多数あります。最近の JavaScript では、そのように複雑な処理は必要なく、デベロッパーはもっと簡潔で読みやすい async/await 構文を活用して非同期オペレーションを実行できます。

ウェブブラウザランタイム

React または React Native を使用してビルドしているウェブ開発者やモバイル開発者は、各自のプロジェクトで SDK for JavaScript を使用できます。SDK の旧バージョン V2 では、ウェブ開発者は https://sdk.amazonaws.com/js/ でホストされている SDK イメージを参照して、SDK 全体をブラウザにロードする必要がありました。

V3 では、必要な V3 クライアントモジュールと必要なすべての JavaScript 関数を Webpack を使用して単一の JavaScript ファイルにバンドルし、HTML ページの <head> のスクリプトタグに追加できます。詳しくは、SDK ドキュメントの「Getting started in a browser script」セクションで説明しています。

DAX データプレーンオペレーション

SDK for JavaScript V3 は、Amazon DynamoDB Accelerator (DAX) データプレーンオペレーションをサポートしています。

AWS SDK for Java 2.x を使用した DynamoDB のプログラミング

このプログラミングガイドは、Java で Amazon DynamoDB を使用することを検討しているプログラマーを対象としています。このガイドでは、抽象化レイヤー、設定管理、エラー処理、再試行ポリシーの制御、キープアライブの管理など、さまざまな概念について説明します。

トピック

- AWS SDK for Java 2.x について
- AWS SDK for Java 2.x の開始方法
- AWS SDK for Java 2.x ドキュメントのレビュー
- サポートされているインターフェイス
- その他のコード例
- 同期プログラミングと非同期プログラミング
- HTTP クライアント

- HTTP クライアントの設定
- エラー処理
- AWS リクエスト ID
- ログ記録
- Pagination (ページ分割)
- データクラス注釈
- ベストプラクティス

AWS SDK for Java 2.x について

DynamoDB には、公式の AWS SDK for Java を使用して Java からアクセスできます。SDK for Java には、1.x と 2.x の 2 つのバージョンがあります。1.x のサポートは 2025 年 12 月 31 日に終了しました。詳細については、AWS デベロッパーツールブログの「AWS SDK for Java v1.x のサポート終了のお知らせ」を参照してください。新規開発には、2018 年に最初にリリースされた 2.x を使用することを強くお勧めします。このガイドは 2.x のみを対象としており、SDK の DynamoDB との関連性がある部分のみを重点的に取り扱っています。

AWS SDK のメンテナンスとサポートの詳細については、「AWS SDKs and Tools Reference Guide」の「AWS SDK and Tools maintenance policy」 および 「AWS SDKs and Tools version support matrix」を参照してください。

AWS SDK for Java 2.x は、1.x コードベースより大幅に変更されています。SDK for Java 2.x は、Java 8 で導入されたノンブロッキング I/O などの最新の Java 機能をサポートしています。また、SDK for Java 2.x では、ネットワーク接続の柔軟性と構成可能性を高めるために、プラガブル HTTP クライアント実装のサポートも追加されています。

SDK for Java 1.x から SDK for Java 2.x への主な変更点は、パッケージ名の変更です。Java 1.x SDK は、`com.amazonaws` のパッケージ名を使用していますが、Java 2.x SDK は `software.amazon.awssdk` を使用します。同様に、Java 1.x SDK の Maven アーティファクトは `com.amazonaws` `groupId` を使用していますが、Java 2.x SDK のアーティファクトは `software.amazon.awssdk` `groupId` を使用します。

> **Important**
>
> AWS SDK for Java 1.x には、`com.amazonaws.dynamodbv2` という名前の DynamoDB パッケージがあります。パッケージ名の「v2」は、Java 2 (J2SE) 用であることを示すもの

ではありません。代わりに「v2」は、パッケージが低レベル API の元のバージョンではなく、DynamoDB 低レベル API の 2 番目のバージョンをサポートしていることを示します。

Java バージョンのサポート

AWS SDK for Java 2.x は、Java リリースの長期サポート (LTS) を完全にサポートします。

AWS SDK for Java 2.x の開始方法

次のチュートリアルでは、SDK for Java 2.x の依存関係を定義するために Apache Maven を使用する方法を説明しています。このチュートリアルでは、DynamoDB に接続して使用可能な DynamoDB テーブルを一覧表示するコードの記述方法も説明します。このガイドのチュートリアルは、「AWS SDK for Java 2.x デベロッパーガイド」の「Get started with the AWS SDK for Java 2.x」のチュートリアルに基づいています。このチュートリアルは、Amazon S3 の代わりに DynamoDB を呼び出すように編集されています。

このチュートリアルを完了するには、以下が必要です。

- ステップ 1: このチュートリアルのために設定する
- ステップ 2: プロジェクトを作成する
- ステップ 3: コードを記述する
- ステップ 4: アプリケーションをビルドして実行する

ステップ 1: このチュートリアルのために設定する

このチュートリアルを開始する前に、以下を実行する必要があります

- DynamoDB へのアクセス許可。
- AWS のサービス を使用して AWS アクセスポータル にアクセスするように設定された Java 開発環境。

このチュートリアル用にセットアップするには、「AWS SDK for Java 2.x デベロッパーガイド」の「Setup overview」の手順に従う必要があります。Java SDK のシングルサインオンアクセスを使用して開発環境を設定し、アクティブな AWS アクセスポータルセッションを確立した後、このチュートリアルのステップ 2 に進みます。

ステップ 2: プロジェクトを作成する

このチュートリアル用のプロジェクトを作成するには、プロジェクトの設定方法に関する入力を求める Maven コマンドを実行します。すべての入力を終えて確定すると、Maven は pom.xml ファイルとスタブ Java ファイルを作成して、プロジェクトの構築を完了します。

1. ターミナルまたはコマンドプロンプトウィンドウを開き、Desktop や Home フォルダなど、任意のディレクトリに移動します。
2. ターミナルに次のコマンドを入力して、Enter を押します。

```
mvn archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-app-quickstart \
   -DarchetypeVersion=2.22.0
```

3. 各プロンプトで、2 列目に記載されている値を入力します。

プロンプト	入力する値
Define value for property 'service':	dynamodb
Define value for property 'httpClient' :	apache-client
Define value for property 'nativeImage' :	false
Define value for property 'credentialProvider'	identity-center
Define value for property 'groupId':	org.example
Define value for property 'artifactId':	getstarted
Define value for property 'version' 1.0-SNAPSHOT:	*<Enter>*

プロンプト	入力する値
`Define value for property 'package' org.example:`	*<Enter>*

4. 最後の値を入力すると、Maven は選択した内容を一覧表示します。確認するには、Y と入力します。または、N と入力して値を再入力します。

Maven は、入力した artifactId 値に基づいて getstarted という名前が付けられたプロジェクトフォルダーを作成します。getstarted フォルダー内で、レビューできる README.md ファイル、pom.xml ファイル、および src ディレクトリを検索します。

Maven は以下のディレクトリーツリーを構築します。

```
getstarted
 ### README.md
 ### pom.xml
 ### src
     ### main
     #   ### java
     #   #   ### org
     #   #       ### example
     #   #           ### App.java
     #   #           ### DependencyFactory.java
     #   #           ### Handler.java
     #   ### resources
     #       ### simplelogger.properties
     ### test
         ### java
             ### org
                 ### example
                     ### HandlerTest.java

 10 directories, 7 files
```

以下は、pom.xml プロジェクトファイルの内容を示しています。

pom.xml

dependencyManagement セクションには AWS SDK for Java 2.x への依存関係が含まれており、dependencies セクションには DynamoDB に対する依存関係があります。このよ

うな依存関係を指定すると、Maven は関連する .jar ファイルを Java クラスパスに強制的に含めます。デフォルトでは、AWS SDK にはすべての AWS のサービス クラスは含まれません。DynamoDB で低レベルインターフェイスを使用する場合、dynamodb アーティファクトへの依存関係が必要です。高レベルインターフェイスを使用する場合は、dynamodb-enhanced アーティファクトへの依存関係が必要です。関連する依存関係が含まれていない場合、コードはコンパイルされません。プロジェクトでは Java 1.8 が使用されています。これは、maven.compiler.source、maven.compiler.target およびプロパティに 1.8 値があるためです。

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>

     <groupId>org.example</groupId>
     <artifactId>getstarted</artifactId>
     <version>1.0-SNAPSHOT</version>
     <packaging>jar</packaging>
     <properties>
         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
         <maven.compiler.source>1.8</maven.compiler.source>
         <maven.compiler.target>1.8</maven.compiler.target>
         <maven.shade.plugin.version>3.2.1</maven.shade.plugin.version>
         <maven.compiler.plugin.version>3.6.1</maven.compiler.plugin.version>
         <exec-maven-plugin.version>1.6.0</exec-maven-plugin.version>
         <aws.java.sdk.version>2.22.0</aws.java.sdk.version> <-------- SDK version
 picked up from archetype version.
         <slf4j.version>1.7.28</slf4j.version>
         <junit5.version>5.8.1</junit5.version>
     </properties>

     <dependencyManagement>
         <dependencies>
             <dependency>
                 <groupId>software.amazon.awssdk</groupId>
                 <artifactId>bom</artifactId>
                 <version>${aws.java.sdk.version}</version>
                 <type>pom</type>
                 <scope>import</scope>
             </dependency>
         </dependencies>
```

```
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>dynamodb</artifactId>  <-------- DynamoDB dependency
            <exclusions>
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>netty-nio-client</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>software.amazon.awssdk</groupId>
                    <artifactId>apache-client</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>sso</artifactId> <-------- Required for identity center
authentication.
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>ssooidc</artifactId> <-------- Required for identity center
authentication.
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>apache-client</artifactId> <-------- HTTP client specified.
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
```

```
            <version>${slf4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <!-- Needed to adapt Apache Commons Logging used by Apache HTTP Client to
Slf4j to avoid
        ClassNotFoundException: org.apache.commons.logging.impl.LogFactoryImpl during
runtime -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <!-- Test Dependencies -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit5.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven.compiler.plugin.version}</version>
            </plugin>
        </plugins>
    </build>

</project>
```

ステップ 3: コードを記述する

次のコードは Maven によって作成された App クラスを示しています。main メソッドはアプリケーションへのエントリポイントであり、Handler クラスのインスタンスを作成してその sendRequest メソッドを呼び出します。

App クラス

```
package org.example;
 import org.slf4j.Logger;
 import org.slf4j.LoggerFactory;

 public class App {
     private static final Logger logger = LoggerFactory.getLogger(App.class);

     public static void main(String... args) {
         logger.info("Application starts");

         Handler handler = new Handler();
         handler.sendRequest();

         logger.info("Application ends");
     }
 }
```

Maven が作成する DependencyFactory クラスには、DynamoDbClient インスタンスを構築して返す dynamoDbClient ファクトリメソッドが含まれています。DynamoDbClient インスタンスは Apache ベースの HTTP クライアントのインスタンスを使用します。これは、どの HTTP クライアントを使用するかを Maven が求めたときに apache-client が指定されたためです。

次のコードは DependencyFactory クラスを示しています。

DependencyFactory クラス

```
package org.example;

 import software.amazon.awssdk.http.apache.ApacheHttpClient;
 import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

 /**
  * The module containing all dependencies required by the {@link Handler}.
```

```
 */
public class DependencyFactory {

    private DependencyFactory() {}

    /**
     * @return an instance of DynamoDbClient
     */
    public static DynamoDbClient dynamoDbClient() {
        return DynamoDbClient.builder()
                        .httpClientBuilder(ApacheHttpClient.builder())
                        .build();
    }
}
```

Handler クラスにはプログラムのメインロジックが含まれています。Handler のインスタンスが App クラス内で作成されると、DependencyFactory は DynamoDbClient サービスクライアントを配置します。コードでは、DynamoDbClient インスタンスを使用して DynamoDB を呼び出します。

Maven は *TODO* コメント付きの次の Handler クラスを生成します。チュートリアルの次のステップでは、*TODO* コメントをコードに置き換えます。

Maven によって生成された **Handler** クラス

```
package org.example;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;


public class Handler {
    private final DynamoDbClient dynamoDbClient;

    public Handler() {
        dynamoDbClient = DependencyFactory.dynamoDbClient();
    }

    public void sendRequest() {
        // TODO: invoking the API calls using dynamoDbClient.
    }
}
```

ロジックを設定するには、Handler クラスのすべてのコンテンツを次のコードに置き換えます。sendRequest メソッドが入力され、必要なインポートが追加されます。

Handler クラス、実装済み

次のコードでは、DynamoDbClientインスタンスを使用して既存のテーブルのリストを取得します。特定のアカウントと AWS リージョン にテーブルが配置されている場合、コードは Logger インスタンスを使用してこのようなテーブルの名前を口グ記録します。

```
package org.example;

 import org.slf4j.Logger;
 import org.slf4j.LoggerFactory;
 import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
 import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;


 public class Handler {
     private final DynamoDbClient dynamoDbClient;

     public Handler() {
         dynamoDbClient = DependencyFactory.dynamoDbClient();
     }

     public void sendRequest() {
         Logger logger = LoggerFactory.getLogger(Handler.class);

         logger.info("calling the DynamoDB API to get a list of existing tables");
         ListTablesResponse response = dynamoDbClient.listTables();

         if (!response.hasTableNames()) {
             logger.info("No existing tables found for the configured account &
 region");
         } else {
             response.tableNames().forEach(tableName -> logger.info("Table: " +
 tableName));
         }
     }
 }
```

ステップ 4: アプリケーションをビルドして実行する

プロジェクトが作成され、完全な `Handler` クラスを含めたら、アプリケーションを構築して実行します。

1. アクティブな AWS IAM アイデンティティセンター セッションがあることを確認します。確認するには、AWS Command Line Interface (AWS CLI) コマンド `aws sts get-caller-identity` を実行してレスポンスを確認します。アクティブなセッションがない場合は、「Sign in using the AWS CLI」の手順を参照してください。
2. ターミナルまたはコマンドプロンプトウィンドウを開いて、プロジェクトディレクトリ `getstarted` に移動します。
3. 次のコマンドを使用して、プロジェクトを構築します。

```
mvn clean package
```

4. 次のコマンドを使用して、アプリケーションを実行します。

```
mvn exec:java -Dexec.mainClass="org.example.App"
```

Success

Maven プロジェクトがエラーなしで構築および実行された場合は、正常に完了しています! これで、Java 2.x 用 SDK を使用して最初の Java アプリケーションを正常に構築できました。

クリーンアップ

このチュートリアルで作成したリソースをクリーンアップするには、プロジェクトフォルダ `getstarted` を削除します。

AWS SDK for Java 2.x ドキュメントのレビュー

「https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/home.htmlAWS SDK for Java 2.x デベロッパーガイド」は、すべての AWS のサービス サービスにおける SDK のすべての側面について説明しています。以下のトピックを確認することをお勧めします。

- Migrate from version 1.x to 2.x — 1.x と 2.x の違いについて詳細に説明しています。このトピックでは、両方のメジャーバージョンを並行して使用する方法も説明しています。
- DynamoDB guide for Java 2.x SDK — テーブルの作成、項目の操作、項目の取得など、基本的な DynamoDB のオペレーションを実行する方法を説明しています。上記の例では、低レベルイン

ターフェイスを使用しています。「サポートされているインターフェイス」セクションで説明するとおり、Java には複数のインターフェイスがあります。

> **Tip**
>
> これらのトピックを確認したら、「AWS SDK for Java 2.x API Reference」をブックマークします。このガイドはすべての AWS のサービス を対象としており、主な API リファレンスとして使用することをお勧めします。

サポートされているインターフェイス

AWS SDK for Java 2.x は、必要な抽象化レベルに応じて、以下のインターフェイスをサポートします。

このセクションのトピック

- 低レベルインターフェイス
- 高レベルインターフェイス
- Document インターフェイス
- Query 例を使用したインターフェイスの比較

低レベルインターフェイス

低レベルインターフェイスは、基盤となるサービス API に 1 対 1 でマッピングされています。すべての DynamoDB API は、このインターフェイスを介して利用できます。つまり、低レベルのインターフェイスでは完全な機能を提供するものの、多くの場合、使い方はより冗長で複雑になります。例えば、文字列を保持するには .s() 関数を使用し、数値を保持するには .n() 関数を使用する必要があります。次の PutItem の例では、低レベルインターフェイスを使用して項目を挿入しています。

```
import org.slf4j.*;
import software.amazon.awssdk.http.crt.AwsCrtHttpClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class PutItem {
```

```
    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT = DynamoDbClient.create();
    private static final Logger LOGGER = LoggerFactory.getLogger(PutItem.class);

    private void putItem() {
        PutItemResponse response = DYNAMODB_CLIENT.putItem(PutItemRequest.builder()
                .item(Map.of(
                        "pk", AttributeValue.builder().s("123").build(),
                        "sk", AttributeValue.builder().s("cart#123").build(),
                        "item_data",
 AttributeValue.builder().s("YourItemData").build(),
                        "inventory", AttributeValue.builder().n("500").build()
                        // ... more attributes ...
                ))
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .tableName("YourTableName")
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
}
```

高レベルインターフェイス

AWS SDK for Java 2.x の高レベルインターフェイスは DynamoDB 拡張クライアントと呼ばれます。このインターフェイスでは、より慣用的なコード作成エクスペリエンスが提供されます。

拡張クライアントでは、クライアント側のデータクラスと、そのデータを保存するように設計された複数の DynamoDB テーブル間でマッピングできます。テーブルおよび対応するモデルクラスの間の関係をコードで定義します。これにより、SDK を利用してデータ型の操作を管理できます。拡張クライアントの詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「DynamoDB enhanced client API」を参照してください。

次の PutItem の例では、高レベルインターフェイスを使用してしています。この例では、YourItem という名前の DynamoDbBean が TableSchema を作成して、putItem() 呼び出しの入力として直接使用できるようにしています。

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
```

```
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
 ENHANCED_DYNAMODB_CLIENT.table("YourTableName", TableSchema.fromBean(YourItem.class));
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedClientPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourItem.class)
                .item(new YourItem("123", "cart#123", "YourItemData", 500))
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }

    @DynamoDbBean
    public static class YourItem {

        public YourItem() {}

        public YourItem(String pk, String sk, String itemData, int inventory) {
            this.pk = pk;
            this.sk = sk;
            this.itemData = itemData;
            this.inventory = inventory;
        }

        private String pk;
        private String sk;
        private String itemData;

        private int inventory;

        @DynamoDbPartitionKey
        public void setPk(String pk) {
            this.pk = pk;
        }
```

```
        public String getPk() {
            return pk;
        }

        @DynamoDbSortKey
        public void setSk(String sk) {
            this.sk = sk;
        }

        public String getSk() {
            return sk;
        }

        public void setItemData(String itemData) {
            this.itemData = itemData;
        }

        public String getItemData() {
            return itemData;
        }

        public void setInventory(int inventory) {
            this.inventory = inventory;
        }

        public int getInventory() {
            return inventory;
        }
    }
}
```

AWS SDK for Java 1.x には独自の高レベルインターフェイスがあり、メインクラス DynamoDBMapper で頻繁に参照されます。AWS SDK for Java 2.x は software.amazon.awssdk.enhanced.dynamodb という名前の別のパッケージ (と Maven アーティファクト) で公開されています。Java 2.x SDK は、メインクラス DynamoDbEnhancedClient で頻繁に参照されます。

イミュータブルデータクラスを使用する高レベルインターフェイス

DynamoDB 拡張クライアント API のマッピング機能は、イミュータブルデータクラスで動作します。不変クラスにはゲッターしかなく、SDK がクラスのインスタンスを作成するために使用するビルダークラスが必要です。Java のイミュータブルクラスは、開発者が副作用のないクラスを作

成するために使用できる一般的なスタイルです。これらのクラスの動作は、複雑なマルチスレッドアプリケーションでより予測可能になります。「High-level interface example」で説明したとおり、@DynamoDbBean アノテーションを使用する代わりに、イミュータブルクラスは Builder クラスを入力として受け取る @DynamoDbImmutable アノテーションを使用します。

次の例では、DynamoDbEnhancedClientImmutablePutItem Builder クラスを入力としてテーブルスキーマを作成します。この例では次に、スキーマを PutItem API コールの入力として使用します。

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientImmutablePutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourImmutableItem>
 DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.fromImmutableClass(YourImmutableItem.class));
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedClientImmutablePutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourImmutableItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableItem.class)
                .item(YourImmutableItem.builder()
                                    .pk("123")
                                    .sk("cart#123")
                                    .itemData("YourItemData")
                                    .inventory(500)
                                    .build())
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
}
```

次の例は、イミュータブルデータクラスを示しています。

```
@DynamoDbImmutable(builder = YourImmutableItem.YourImmutableItemBuilder.class)
```

```
class YourImmutableItem {
    private final String pk;
    private final String sk;
    private final String itemData;
    private final int inventory;
    public YourImmutableItem(YourImmutableItemBuilder builder) {
        this.pk = builder.pk;
        this.sk = builder.sk;
        this.itemData = builder.itemData;
        this.inventory = builder.inventory;
    }

    public static YourImmutableItemBuilder builder() { return new
 YourImmutableItemBuilder(); }

    @DynamoDbPartitionKey
    public String getPk() {
        return pk;
    }

    @DynamoDbSortKey
    public String getSk() {
        return sk;
    }

    public String getItemData() {
        return itemData;
    }

    public int getInventory() {
        return inventory;
    }

    static final class YourImmutableItemBuilder {
        private String pk;
        private String sk;
        private String itemData;
        private int inventory;

        private YourImmutableItemBuilder() {}

        public YourImmutableItemBuilder pk(String pk) { this.pk = pk; return this; }
        public YourImmutableItemBuilder sk(String sk) { this.sk = sk; return this; }
```

```
        public YourImmutableItemBuilder itemData(String itemData) { this.itemData =
 itemData; return this; }
        public YourImmutableItemBuilder inventory(int inventory) { this.inventory =
 inventory; return this; }

        public YourImmutableItem build() { return new YourImmutableItem(this); }
    }
}
```

イミュータブルデータクラスとサードパーティーのボイラープレート生成ライブラリを使用する高レベルインターフェイス

前のセクションで説明したイミュータブルデータクラスの例では、いくつかのボイラープレートコードが必要です。例えば、Builder クラス以外にも、データクラスには getter ロジックと setter ロジックがあります。Project Lombok などのサードパーティーライブラリは、このようなタイプのボイラープレートコードの生成に役立ちます。ボイラープレートコードの大部分を低減すると、イミュータブルデータクラスと AWS SDK の使用に必要なコードの量を制限できます。これにより、コードの生産性と可読性がさらに向上します。詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「Use third-party libraries, such as Lombok」を参照してください。

次の例は、Project Lombok が DynamoDB 拡張クライアント API を使用するために必要なコードをどのように簡略化できるかを示しています。

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedClientImmutableLombokPutItem {

    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<YourImmutableLombokItem>
 DYNAMODB_TABLE = ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.fromImmutableClass(YourImmutableLombokItem.class));
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedClientImmutableLombokPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<YourImmutableLombokItem> response =
 DYNAMODB_TABLE.putItemWithResponse(PutItemEnhancedRequest.builder(YourImmutableLombokItem.clas
                .item(YourImmutableLombokItem.builder()
```

```
                        .pk("123")
                        .sk("cart#123")
                        .itemData("YourItemData")
                        .inventory(500)
                        .build())
                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                .build());
        LOGGER.info("PutItem call consumed [" +
 response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }
}
```

次の例は、イミュータブルデータクラスのイミュータブルデータオブジェクトを示しています。

```
import lombok.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;

@Builder
@DynamoDbImmutable(builder =
 YourImmutableLombokItem.YourImmutableLombokItemBuilder.class)
@Value
public class YourImmutableLombokItem {

    @Getter(onMethod_=@DynamoDbPartitionKey)
    String pk;
    @Getter(onMethod_=@DynamoDbSortKey)
    String sk;
    String itemData;
    int inventory;
}
```

`YourImmutableLombokItem` クラスは Project Lombok と AWS SDK が提供する以下のアノテーションを使用します。

- @Builder — Project Lombok が提供するデータクラス用の複雑な Builder API を生成します。
- @DynamoDbImmutable — `DynamoDbImmutable` クラスを、AWS SDK が提供する DynamoDB マッピング可能なエンティティのアノテーションとして識別します。
- @Value – `@Data` のイミュータブルバリアント。デフォルトで、すべてのフィールドはプライベートかつ最終的なものになり、setters は生成されません。Project Lombok はこのアノテーションを提供します。

Document インターフェイス

AWS SDK for Java 2.x Document インターフェイスでは、データ型記述子を指定する必要がありません。データ型は、データ自体のセマンティクスによって暗示されています。この Document インターフェイスは AWS SDK for Java 1.x の Document インターフェイスに似ていますが、インターフェイスは再設計されています。

次の Document interface example は、この Document インターフェイスを使用した `PutItem` コールの表現を示しています。この例では EnhancedDocument も使用しています。Enhanced Document API を使用して DynamoDB テーブルに対してコマンドを実行するには、まずテーブルを Document テーブルスキーマに関連付けて、`DynamoDBTable` リソースオブジェクトを作成する必要があります。Document テーブルスキーマの Builder には、プライマリインデックスキーと属性コンバーターのプロバイダーが必要です。

デフォルトタイプの Document 属性の変換には `AttributeConverterProvider.defaultProvider()` を使用できます。全体的なデフォルト動作は、カスタム `AttributeConverterProvider` 実装で変更できます。また、1 つの属性のコンバーターを変更することもできます。「AWS SDK および ツールリファレンスガイド」には、カスタムコンバーターの使用方法の詳細と例が提供されています。コンバーターは主に、デフォルトコンバーターがないドメインクラスの属性に使用します。カスタムコンバータを使用すると、DynamoDB への書き込みまたは読み取りに必要な情報を SDK に提供できます。

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document.EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.ReturnConsumedCapacity;

public class DynamoDbEnhancedDocumentClientPutItem {
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
            ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.documentSchemaBuilder()

 .addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk", AttributeValueType.S)
                            .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
 AttributeValueType.S)

 .attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                            .build());
```

```
    private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientPutItem.class);

    private void putItem() {
        PutItemEnhancedResponse<EnhancedDocument> response =
DYNAMODB_TABLE.putItemWithResponse(
                        PutItemEnhancedRequest.builder(EnhancedDocument.class)
                                .item(
                                    EnhancedDocument.builder()

.attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                                            .putString("pk", "123")
                                            .putString("sk", "cart#123")
                                            .putString("item_data", "YourItemData")
                                            .putNumber("inventory", 500)
                                            .build())
                                .returnConsumedCapacity(ReturnConsumedCapacity.TOTAL)
                                .build());
        LOGGER.info("PutItem call consumed [" +
response.consumedCapacity().capacityUnits() + "] Write Capacity Unites (WCU)");
    }

}
```

JSON ドキュメントとネイティブの Amazon DynamoDB のデータ型との間の変換を行うには、以下のユーティリティメソッドを使用します。

- `EnhancedDocument.fromJson(String json)` — JSON 文字列から新しい EnhancedDocument インスタンスを作成します。
- `EnhancedDocument.toJson()` — Document の JSON 文字列表現を作成して、その他の JSON オブジェクトと同様にアプリケーションで使用できるようにします。

`Query` 例を使用したインターフェイスの比較

このセクションでは、さまざまなインターフェイスにおける、同じ `Query` 呼び出しの表現について説明します。これらのクエリの結果を微調整する場合は、以下の点に注意してください。

- DynamoDB では、単一の特定のパーティションキー値をターゲットにするため、パーティションキーは完全に指定する必要があります。

- ソートキーには begins_with を使用したキー条件式があるため、カートの商品のみがこのクエリの対象となります。
- limit() メソッドは、返される項目を最大 100 個に制限します。
- scanIndexForward を false に設定すると、結果は UTF-8 バイトの逆順で返されます。つまり通常、番号が最も小さいカートの商品が最初に返されます。scanIndexForward を false に設定すると、この順序が逆になり、番号が最も大きいカートの商品が最初に返されます。
- フィルターは条件に一致しない結果をすべて削除します。項目がフィルターに一致するかどうかにかかわらず、フィルターが適用されるデータは読み取りキャパシティを消費します。

Example低レベルインターフェイスを使用した Query

次の例では、keyConditionExpression を使用して、YourTableName という名前のテーブルをクエリします。これにより、特定のプレフィックス値で始まる特定のパーティションキー値とソートキー値にクエリが制限されます。このようなキーの条件は、DynamoDB から読み取られるデータ量を制限します。このクエリは最後に、filterExpression を使用して DynamoDB から取得したデータにフィルターを適用します。

```
import org.slf4j.*;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;

import java.util.Map;

public class Query {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbClient DYNAMODB_CLIENT =
 DynamoDbClient.builder().build();
    private static final Logger LOGGER = LoggerFactory.getLogger(Query.class);

    private static void query() {
        QueryResponse response = DYNAMODB_CLIENT.query(QueryRequest.builder()
                .expressionAttributeNames(Map.of("#name", "name"))
                .expressionAttributeValues(Map.of(
                    ":pk_val", AttributeValue.fromS("id#1"),
                    ":sk_val", AttributeValue.fromS("cart#"),
                    ":name_val", AttributeValue.fromS("SomeName")))
                .filterExpression("#name = :name_val")
                .keyConditionExpression("pk = :pk_val AND begins_with(sk, :sk_val)")
```

```
                .limit(100)
                .scanIndexForward(false)
                .tableName("YourTableName")
                .build());

        LOGGER.info("nr of items: " + response.count());
        LOGGER.info("First item pk: " + response.items().get(0).get("pk"));
        LOGGER.info("First item sk: " + response.items().get(0).get("sk"));
    }
}
```

Example Document インターフェイスを使用した **Query**

次の例では、Document インターフェイスを使用して、YourTableName という名前のテーブルをクエリします。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.document.EnhancedDocument;
import software.amazon.awssdk.enhanced.dynamodb.model.*;

import java.util.Map;

public class DynamoDbEnhancedDocumentClientQuery {

    // Create a DynamoDB client with the default settings connected to the DynamoDB
    // endpoint in the default region based on the default credentials provider chain.
    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
    private static final DynamoDbTable<EnhancedDocument> DYNAMODB_TABLE =
            ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
 TableSchema.documentSchemaBuilder()
                    .addIndexPartitionKey(TableMetadata.primaryIndexName(),"pk",
 AttributeValueType.S)
                    .addIndexSortKey(TableMetadata.primaryIndexName(), "sk",
 AttributeValueType.S)

 .attributeConverterProviders(AttributeConverterProvider.defaultProvider())
                    .build());
    private static final Logger LOGGER =
 LoggerFactory.getLogger(DynamoDbEnhancedDocumentClientQuery.class);
```

```
    private void query() {
        PageIterable<EnhancedDocument> response =
 DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
                .filterExpression(Expression.builder()
                        .expression("#name = :name_val")
                        .expressionNames(Map.of("#name", "name"))
                        .expressionValues(Map.of(":name_val",
 AttributeValue.fromS("SomeName")))
                        .build())
                .limit(100)
                .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
                        .partitionValue("id#1")
                        .sortValue("cart#")
                        .build()))
                .scanIndexForward(false)
                .build());

        LOGGER.info("nr of items: " + response.items().stream().count());
        LOGGER.info("First item pk: " +
 response.items().iterator().next().getString("pk"));
        LOGGER.info("First item sk: " +
 response.items().iterator().next().getString("sk"));

    }
}
```

Example高レベルインターフェイスを使用した Query

次の例では、DynamoDB Enhanced Client API を使用して、YourTableName という名前のテーブルをクエリします。

```
import org.slf4j.*;
import software.amazon.awssdk.enhanced.dynamodb.*;
import software.amazon.awssdk.enhanced.dynamodb.mapper.annotations.*;
import software.amazon.awssdk.enhanced.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

import java.util.Map;

public class DynamoDbEnhancedClientQuery {

    private static final DynamoDbEnhancedClient ENHANCED_DYNAMODB_CLIENT =
 DynamoDbEnhancedClient.builder().build();
```

```
   private static final DynamoDbTable<YourItem> DYNAMODB_TABLE =
ENHANCED_DYNAMODB_CLIENT.table("YourTableName",
TableSchema.fromBean(DynamoDbEnhancedClientQuery.YourItem.class));
   private static final Logger LOGGER =
LoggerFactory.getLogger(DynamoDbEnhancedClientQuery.class);

   private void query() {
       PageIterable<YourItem> response =
DYNAMODB_TABLE.query(QueryEnhancedRequest.builder()
               .filterExpression(Expression.builder()
                       .expression("#name = :name_val")
                       .expressionNames(Map.of("#name", "name"))
                       .expressionValues(Map.of(":name_val",
AttributeValue.fromS("SomeName")))
                       .build())
               .limit(100)
               .queryConditional(QueryConditional.sortBeginsWith(Key.builder()
                       .partitionValue("id#1")
                       .sortValue("cart#")
                       .build()))
               .scanIndexForward(false)
               .build());

       LOGGER.info("nr of items: " + response.items().stream().count());
       LOGGER.info("First item pk: " + response.items().iterator().next().getPk());
       LOGGER.info("First item sk: " + response.items().iterator().next().getSk());
   }

   @DynamoDbBean
   public static class YourItem {

       public YourItem() {}

       public YourItem(String pk, String sk, String name) {
           this.pk = pk;
           this.sk = sk;
           this.name = name;
       }

       private String pk;
       private String sk;
       private String name;

       @DynamoDbPartitionKey
```

```
        public void setPk(String pk) {
            this.pk = pk;
        }

        public String getPk() {
            return pk;
        }

        @DynamoDbSortKey
        public void setSk(String sk) {
            this.sk = sk;
        }

        public String getSk() {
            return sk;
        }

        public void setName(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }
    }
}
```

イミュータブルデータクラスを使用する高レベルインターフェイス

高レベルのイミュータブルデータクラスを使用して Query を実行する場合、YourItem または YourImmutableItem エンティティクラスの構築を除き、コードは高レベルインターフェイスの例と同じです。詳細については、「PutItem」の例を参照してください。

イミュータブルデータクラスとサードパーティーのボイラープレート生成ライブラリを使用する高レベルインターフェイス

高レベルのイミュータブルデータクラスを使用して Query を実行する場合、YourItem または YourImmutableLombokItem エンティティクラスの構築を除き、コードは高レベルインターフェイスの例と同じです。詳細については、「PutItem」の例を参照してください。

その他のコード例

SDK for Java 2.x で DynamoDB を使用する場合のその他の例については、以下のコードサンプルリポジトリを参照してください。

- AWS 公式シングルアクションのコード例
- コミュニティが管理するシングルアクションのコード例
- AWS 公式のシナリオ指向コード例

同期プログラミングと非同期プログラミング

AWS SDK for Java 2.x は、DynamoDB などの AWS のサービス に同期クライアントと非同期クライアントの両方を提供します。

`DynamoDbClient` クラスと `DynamoDbEnhancedClient` クラスは、クライアントがサービスから応答を受信するまでスレッドの実行をブロックする同期メソッドを提供します。非同期のオペレーションが必要ない場合、このクライアントは DynamoDB を操作する最も簡単な方法です。

`DynamoDbAsyncClient` クラスと `DynamoDbEnhancedAsyncClient` クラスは、直ちに呼び出し元に戻り、応答を待たずに呼び出し元のスレッドに制御を返す非同期メソッドを提供します。ノンブロッキングクライアントの場合、少数のスレッド間で優れた同時実行が可能で、最小限のコンピューティングリソースで I/O リクエストを効率的に処理できるという利点があります。これにより、スループットと即応性が向上します。

AWS SDK for Java 2.x はノンブロッキング I/O のネイティブサポートを使用しています。AWS SDK for Java 1.x ではノンブロッキング I/O をシミュレートする必要がありました。

非同期メソッドは応答が利用できるようになる前に呼び出し元に戻るため、応答の準備ができたら応答を取得する手段を講じる必要があります。AWS SDK for Java のこれらのメソッドは、その後の非同期オペレーションの結果を含む `CompletableFuture` オブジェクトを返します。このような `CompletableFuture` オブジェクトに対して `get()` または `join()` を呼び出すと、結果が利用できるようになるまでコードはブロックされます。リクエストと同時にこれらの呼び出しを行う場合、動作は通常の同期呼び出しと同様になります。

非同期プログラミングの詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「Use asynchronous programming」を参照してください。

HTTP クライアント

すべてのクライアントをサポートするための AWS のサービス との通信を処理する HTTP クライアントがあります。アプリケーションに最も適切な特性を持つ HTTP クライアントを選択して、代替 HTTP クライアントをプラグインできます。比較的軽量なクライアントも、設定オプションが豊富にあるクライアントもあります。

HTTP クライアントによっては、同期使用のみをサポートするものもあれば、非同期使用のみをサポートするものもあります。ワークロードに最適な HTTP クライアントの選択に役立つフローチャートについては、「AWS SDK for Java 2.x デベロッパーガイド」の「HTTP client recommendations」を参照してください。

利用できる HTTP クライアントのいくつかを次の一覧にまとめています。

トピック

- Apache ベースの HTTP クライアント
- URLConnection ベースの HTTP クライアント
- Netty ベースの HTTP クライアント
- AWS CRT ベースの HTTP クライアント

Apache ベースの HTTP クライアント

`ApacheHttpClient` クラスは同期サービスクライアントをサポートします。これは同期でのデフォルトの HTTP クライアントです。`ApacheHttpClient` クラスの設定については、「AWS SDK for Java 2.x デベロッパーガイド」の「Configure the Apache-based HTTP client」を参照してください。

`URLConnection` ベースの HTTP クライアント

`UrlConnectionHttpClient` クラスは、同期クライアント向けのもう 1 つのオプションです。Apache ベースの HTTP クライアントよりもロード時間がかからないとはいえ、機能は限られています。`UrlConnectionHttpClient` クラスの設定については、「AWS SDK for Java 2.x デベロッパーガイド」の「Configure the URLConnection-based HTTP client」を参照してください。

Netty ベースの HTTP クライアント

`NettyNioAsyncHttpClient` クラスは非同期クライアントをサポートします。これは非同期でのデフォルトの選択肢です。`NettyNioAsyncHttpClient` クラスの設定については、「AWS SDK

for Java 2.x デベロッパーガイド」の「Configure the Netty-based HTTP client」を参照してください。

AWS CRT ベースの HTTP クライアント

同期クライアントと非同期クライアントの両方をサポートするオプションには、AWS Common Runtime (CRT) ライブラリに新たに追加された `AwsCrtHttpClient` クラスと `AwsCrtAsyncHttpClient` クラスがあります。

`AwsCrtHttpClient` クラスと `AwsCrtAsyncHttpClient` クラスの設定については、「AWS SDK for Java 2.x デベロッパーガイド」の「Configure the AWS CRT-based HTTP clients」を参照してください。

AWS CRT ベースの HTTP クライアントの概要については、「AWS Developer Tools Blog」の「Announcing availability of the AWS CRT HTTP Client in the AWS SDK for Java 2.x」を参照してください。

HTTP クライアントの設定

クライアントを設定する際には、次のとおりのさまざまな設定オプションを指定できます。

- API コールのさまざまな側面でのタイムアウトの設定。
- TCP キープアライブの有効化。
- エラー発生時の再試行ポリシーの制御。
- 実行インターセプターインスタンスが変更できる実行属性の指定。実行インターセプターは、API リクエストと応答の実行をインターセプトするコードを作成できます。これにより、メトリクスの公開やリクエストの変更などのタスクを進行中に実行できます。
- HTTP ヘッダーの追加や操作。
- クライアント側のパフォーマンスメトリクスの追跡を有効にできます。この機能を使用すると、アプリケーション内のサービスクライアントに関するメトリクスを収集し、Amazon CloudWatch で出力を分析できます。
- 非同期の再試行やタイムアウトタスクなどのタスクのスケジュール設定に使用する代替エグゼキュータサービスを指定できます。

設定を制御するには、`ClientOverrideConfiguration` オブジェクトをサービスクライアントの `Builder` クラスに提供します。これについては、次のセクションのコードサンプルのいくつかで説明します。

ClientOverrideConfiguration では、標準的な設定のオプションが提供されます。プラガブル HTTP クライアントによっては、実装に応じた設定もできます。

このセクションのトピック

- タイムアウト設定
- RetryMode
- DefaultsMode
- キープアライブの設定
- 最大接続数と同時実行数

タイムアウト設定

クライアントの設定を調整して、サービスの呼び出しに関連するタイムアウトを制御できます。DynamoDB は、その他の AWS のサービス サービスと比較してレイテンシーが低減します。このため、ネットワークに問題が発生した場合のフェイルファストを実現するように、これらのプロパティを調整してタイムアウト値を低く (4 ~ 10 秒の間) 設定することができます。

DynamoDB クライアントで ClientOverrideConfiguration を使用するか、基盤となる HTTP クライアント実装で詳細な設定オプションを変更することによって、レイテンシーに関する動作をカスタマイズできます。

以下の影響を左右するプロパティを設定するには、ClientOverrideConfiguration を使用できます。

- apiCallAttemptTimeout — HTTP リクエストの 1 回の試行が完了するまで、諦めてタイムアウトになるまで待機する時間。
- apiCallTimeout – クライアントで API コールを完了するまでに許容される時間。これには、再試行を含むすべての HTTP リクエストで設定されるリクエストハンドラーの実行が含まれます。

接続タイムアウトやソケットタイムアウトなど、タイムアウトオプションによっては、AWS SDK for Java 2.x は、デフォルト値を提供します。SDK は、API コールのタイムアウトや個別の API コールの試行タイムアウトのデフォルト値は提供していません。これらのタイムアウトが ClientOverrideConfiguration で設定されていない場合、SDK はソケットタイムアウト値を apiCallAttemptTimeout として効果的に使用し、apiCallTimeout を強制しません。ソケットのタイムアウトのデフォルト値は 30 秒です。

詳細と例については、「AWS SDK for Java 2.x デベロッパーガイド」の「タイムアウト」を参照してください。

タイムアウトを極端に低く設定しないでください。レイテンシーが高くなるときにすべてのリクエストが失敗するリスクがあります。より良いパターンは、リクエストヘッジングを使用して、最初のリクエストに時間がかかりすぎる場合に 2 番目の同時リクエストを開始することです。リクエストヘッジングがテールレイテンシーを低減する方法の例については、AWS データベースブログの「Global Payments Inc. が DynamoDB によるリクエストヘッジングを使用してテールレイテンシーを改善した方法」を参照してください。

RetryMode

タイムアウト設定に関して考慮すべきもう 1 つの設定は、`RetryMode` 設定オブジェクトです。この設定オブジェクトには、再試行動作のコレクションが含まれています。

SDK for Java 2.x は、次の再試行モードをサポートしています。

- `legacy` — 明示的に変更しない場合のデフォルトの再試行モード。この再試行モードは Java SDK 固有のものです。このモードでは、最大 3 回まで再試行でき、DynamoDB などのサービスの場合は最大 8 回まで再試行できます。
- `standard` — 他の AWS SDK との整合性が高いことから、「standard」と名付けられています。再試行するたびに、このモードは 0 ミリ秒から指数関数的に増加する上限までのランダムな時間を待機します。このアプローチはフルジッターと呼ばれ、固定遅延を使用するのではなく、各待機を完全にランダム化します。DynamoDB の場合、その上限は 25 ミリ秒の基本遅延から始まり、それ以降の再試行ごとに 2 倍になり、最大 20 秒に制限されます。このモードは、検出された障害条件の数が `legacy` モードよりも多い場合に再試行を実行します。DynamoDB の場合、the maximum number of attempts でオーバーライドしない限り、最大合計 9 回の試行が実行されます。
- `adaptive` — `standard` モードに基づいて構築されています。成功率を最大化するために AWS リクエスト率を動的に制限します。これにより、リクエストのレイテンシーが増大する場合があります。予測可能なレイテンシーが重要な場合は、アダプティブ再試行モードはお勧めしません。

このような再試行モードの詳細な定義については、「AWS SDK とツールのリファレンスガイド」の「再試行動作」トピックを参照してください。

すべての AWS SDK のデフォルトの再試行動作に対する今後の変更については、AWS デベロッパーツールブログの「AWS SDK およびツールの更新された再試行動作の発表」を参照してくださ

い。SDK for Java 2.x の正確な詳細については、GitHub ウェブサイトの「再試行動作の更新 (ディスカッション #6984)」を参照してください。

再試行戦略

各再試行モードは、再試行戦略によって実装されます。再試行戦略 API (`software.amazon.awssdk.retries` パッケージ) は、古い再試行ポリシー API (`software.amazon.awssdk.core.retry` の `RetryPolicy` および `RetryCondition`) よりも優先されます。既存の `RetryPolicy` 設定は引き続き機能します。SDK はそれらを再試行戦略に適応させますが、新しいコードでは `RetryStrategy` を使用する必要があります。AWS SDK for Java 2.x には、次の 3 つの組み込み戦略があります。

- `StandardRetryStrategy` – ほとんどのユースケースに推奨される戦略。
- `LegacyRetryStrategy` – 戦略を指定しない場合のデフォルトの戦略。スロットリングの例外と非スロットリングの例外は異なる方法で扱われます。
- `AdaptiveRetryStrategy` – リソースに制約のあるユースケースの場合。クライアント側のレート制限が追加されます。また、クライアントが 1 つのリソースに対して動作することを前提としています。

詳細については、「Configure retry behavior in the AWS SDK for Java 2.x SDK」を参照してください。

クライアントでスロットリング例外や一時的なサーバーエラーなどの再試行できるエラーが発生すると、SDK は自動的にリクエストを再試行します。再試行戦略を次のようにカスタマイズすることで、これらの再試行の回数と速度を制御できます。

- `maxAttempts` – リクエストが失敗したと見なされるまでの最大試行回数 (最初の試行と再試行)。DynamoDB クライアントの場合、デフォルトはすべての戦略で 9 回の試行です。
- `backoffStrategy` – 再試行間の遅延を決定する `BackoffStrategy`。デフォルトでは、DynamoDB クライアントは基本遅延 25 ミリ秒、最大遅延 20 秒の `BackoffStrategy.exponentialDelay` を使用します。
- `retryOnException` – SDK の再試行可能な例外のデフォルトセットに加えて、再試行をトリガーする例外タイプをセットに追加します。

次のコードは、DynamoDB クライアントを最大 6 回の試行 (最初の試行と 5 回の再試行) にカスタマイズした標準の再試行戦略と、100 ミリ秒から開始し 1 秒に制限されるエクスポネンシャルバックオフで設定します。

```
BackoffStrategy backoffStrategy =
    BackoffStrategy.exponentialDelay(Duration.ofMillis(100), Duration.ofSeconds(1));

StandardRetryStrategy retryStrategy = AwsRetryStrategy.standardRetryStrategy()
    .toBuilder()
    .maxAttempts(6)
    .backoffStrategy(backoffStrategy)
    .build();

DynamoDbClient client = DynamoDbClient.builder()
    .overrideConfiguration(o -> o.retryStrategy(retryStrategy))
    .build();
```

DefaultsMode

`ClientOverrideConfiguration` と `RetryMode` が管理しないタイムアウトプロパティは、通常、`DefaultsMode` を指定することで暗黙的に設定されます。

AWS SDK for Java 2.x (バージョン 2.17.102 以降) では、`DefaultsMode` のサポートが導入されました。この機能は、HTTP 通信設定、再試行動作、サービスのリージョンエンドポイント設定、他の SDK 関連の構成など、一般的で設定可能な設定のデフォルト値のセットを提供します。この機能を使用すると、一般的な使用シナリオに合わせて新しい設定のデフォルトを指定できます。

デフォルトモードはすべての AWS SDK で標準化されています。SDK for Java 2.x は、次のデフォルトモードをサポートしています。

- `legacy` — AWS SDK によって異なり、`DefaultsMode` が確立される前に存在していたデフォルト設定を使用します。
- `standard` — ほとんどのシナリオで、最適化されていないデフォルト設定を提供します。
- `in-region` — 標準モードに基づいて構築されており、同じ AWS リージョン 内から AWS のサービス を呼び出すアプリケーションにカスタマイズされた最適化が含まれています。
- `cross-region` — 標準モードに基づいて構築されており、異なるリージョンの AWS のサービスを呼び出すアプリケーション向けにタイムアウトを長く指定した設定が含まれています。
- `mobile` — 標準モードに基づいて構築されており、レイテンシーの長いモバイルアプリケーション向けにカスタマイズした、タイムアウト時間を長く指定した設定が含まれています。
- `auto` – 標準モードに基づいて構築されており、実験的な機能が含まれています。SDK はランタイム環境を検出して適切な設定を自動的に決定しようとします。自動検出はヒューリスティックベースで、100% の精度が得られるとは限りません。ランタイム環境が特定できない場合は、標準モー

ドを使用します。自動検出を行うと、インスタンスのメタデータとユーザデータがクエリされる可能性があり、これによりレイテンシーが発生する場合があります。起動時のレイテンシーがアプリケーションにとって最重要な場合は、代わりに明示的な DefaultsMode を選択することをおすすめします。

デフォルトモードは次の方法で設定できます。

- AwsClientBuilder.Builder#defaultsMode(DefaultsMode) を介した直接クライアントでの設定。
- defaults_mode プロファイルファイルプロパティを使用した設定プロファイルでの設定。
- aws.defaultsMode システムプロパティを使用したグローバル設定。
- AWS_DEFAULTS_MODE 環境変数を使用したグローバル設定。

> **Note**
>
> legacy 以外のモードでは、ベストプラクティスの進化に応じて、提供されるデフォルト値が変更される可能性があります。そのため、legacy 以外のモードを使用している場合は、SDK をアップグレードする際にテストを実施することをお勧めします。

「AWS SDK とツールのリファレンスガイド」の「スマート設定デフォルト」には、さまざまなデフォルトモードでの設定プロパティとデフォルト値の一覧が記載されています。

デフォルトのモード値は、アプリケーションの特性と、アプリケーションが操作する AWS のサービス に基づいて選択します。

デフォルト値は、幅広い AWS のサービス が選択されることを考慮して設定されています。DynamoDB テーブルとアプリケーションの両方が単一のリージョンにデプロイされる典型的な DynamoDB デプロイの場合、standard デフォルトモードの中で最も関連性があるのは in-region のデフォルトモードです。

Example低レイテンシーの呼び出し向けに調整した DynamoDB SDK クライアントの設定

次の例では、低レイテンシーが期待される DynamoDB 呼び出し向けにタイムアウトをより低い値に調整します。

```
DynamoDbAsyncClient asyncClient = DynamoDbAsyncClient.builder()
    .defaultsMode(DefaultsMode.IN_REGION)
```

```
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder())
    .overrideConfiguration(ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ofSeconds(3))
        .apiCallAttemptTimeout(Duration.ofMillis(500))
        .build())
    .build();
```

個別の HTTP クライアント実装により、タイムアウトと接続使用の動作はさらに詳細に制御できます。例えば、AWS CRT ベースのクライアントの場合、ConnectionHealthConfiguration を有効にすると、クライアントは使用される接続のヘルスをアクティブにモニタリングできます。詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「Advanced configuration of AWS CRT-based HTTP clients」を参照してください。

キープアライブの設定

キープアライブを有効にすると、接続を再利用することでレイテンシーを短縮できます。キープアライブには、HTTP キープアライブと TCP キープアライブの 2 種類があります。

- HTTP キープアライブは、その後のリクエストで接続を再利用できるように、クライアントとサーバー間の HTTPS 接続の維持を試行します。これにより、重い HTTPS 認証がその後のリクエストでスキップされます。HTTP キープアライブはすべてのクライアントでデフォルトで有効になっています。
- TCP キープアライブは、基盤となるオペレーティングシステムに対して、ソケット接続を介して小型のパケットを送信するようにリクエストします。これにより、ソケットが維持されることがさらに保証され、ドロップを即座に検出できます。これにより、ドロップしたソケットの使用にその後のリクエストが時間を費やす必要がなくなります。TCP キープアライブは、デフォルトではすべてのクライアントで無効になっています。以下のコードサンプルは、各 HTTP クライアントでこれを有効化する方法を示しています。CRT ベース以外のすべての HTTP クライアントの場合、キープアライブを有効にすると、実際のキープアライブメカニズムはオペレーティングシステムに依存します。このため、タイムアウトやパケット数など、追加の TCP キープアライブ値をオペレーティングシステムで設定する必要があります。これを実行するには、Linux または macOS では `sysctl` を使用し、Windows ではレジストリ値を使用します。

Example Apache ベースの HTTP クライアントで TCP キープアライブを有効にするには

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(ApacheHttpClient.builder().tcpKeepAlive(true))
    .build();
```

URLConnection ベースの HTTP クライアント

URLConnection ベースの HTTP クライアント HttpURLConnection を使用する同期クライアントには、キープアライブを有効にするメカニズムはありません。

Example Netty ベースの HTTP クライアントで TCP キープアライブを有効にするには

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(NettyNioAsyncHttpClient.builder().tcpKeepAlive(true))
    .build();
```

Example AWS CRT ベースの HTTP クライアントで TCP キープアライブを有効にするには

AWS CRT ベースの HTTP クライアントを使用すると、TCP キープアライブを有効にして、実行時間を制御できます。

```
DynamoDbClient client = DynamoDbClient.builder()
    .httpClientBuilder(AwsCrtHttpClient.builder()
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
        .build()))
    .build();
```

非同期の DynamoDB クライアントを使用する場合は、次のコードのとおり TCP キープアライブを有効にすることができます。

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
    .httpClientBuilder(AwsCrtAsyncHttpClient.builder()
    .tcpKeepAliveConfiguration(TcpKeepAliveConfiguration.builder()
        .keepAliveInterval(Duration.ofSeconds(50))
        .keepAliveTimeout(Duration.ofSeconds(5))
        .build()))
    .build();
```

最大接続数と同時実行数

基本 URLConnection を除くすべての HTTP エンジンは、許可される同時リクエストの数を制御するために、maxConnections() (同期の場合) または maxConcurrency() (非同期の場合) を提供します。デフォルトは 50 です。同じクライアントインスタンスに対して高い同時実行性が予想される場合は、この値を引き上げることができます。

```
ApacheHttpClient.Builder httpClientBuilder =
  ApacheHttpClient.builder()
                  .maxConnections(100); // Set to 100 from this synchronous client

NettyNioAsyncHttpClient.Builder nettyClientBuilder =
  NettyNioAsyncHttpClient.builder()
                         .maxConcurrency(200); // Async allows higher throughput
```

エラー処理

例外処理については、AWS SDK for Java 2.x ではランタイム (非チェック) 例外を使用します。

すべての SDK 例外をカバーする基本となる例外は、Java 非チェック RuntimeException の拡張である SdkServiceException です。これをキャッチすると、SDK からスローされるすべての例外をキャッチできます。

SdkServiceException には、AwsServiceException というサブクラスがあります。このサブクラスは、AWS のサービス との間の通信に問題があることを示します。これには、DynamoDB との通信の問題を示す DynamoDbException というサブクラスがあります。これをキャッチすると、DynamoDB に関連するすべての例外はキャッチできますが、他の SDK 例外はキャッチできません。

DynamoDbException には、さらに具体的な例外タイプもあります。このような例外タイプの中には、TableAlreadyExistsException などのコントロールプレーンの操作に適用されるものもあります。データプレーン操作に適用されるものもあります。一般的なデータプレーン例外の例は、以下のとおりです。

- ConditionalCheckFailedException — リクエストに false と評価される条件を指定しました。たとえば、項目に条件付き更新を実行しようとしたかもしれませんが、属性の実際の値は、条件の予期される値と一致しませんでした。この方法で失敗したリクエストは再試行されません。

その他の状況については、特定の例外は定義されていません。例えば、リクエストがスロットリングすると、特定の ProvisionedThroughputExceededException がスローされ、その他の場合にはより一般的な DynamoDbException がスローされる可能性があります。いずれの場合も、isThrottlingException() が true を返したかを確認することで、例外がスロットリングによって引き起こされたかどうかを判断できます。

アプリケーションのニーズに応じて、すべての AwsServiceException インスタンスまたは DynamoDbException インスタンスをキャッチできます。ただし多くの場合、状況に応じて異なる

動作が必要になります。条件チェックの失敗を処理するロジックは、スロットリングを処理するロジックとは異なります。どの例外パスに対処するかを定義して、代替パスは必ずテストしてください。テストの実施は、関連するすべてのシナリオに確実に対処できるようにするうえで役立ちます。

よく発生するエラーのリストについては、「DynamoDB でのエラー処理」を参照してください。また、「Amazon DynamoDB API Reference」の「Common Errors」も参照してください。この API Reference には、`Query` オペレーションなど、各 API オペレーションで発生する可能性のある厳密なエラーも記載されています。例外処理の詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「Exception handling for the AWS SDK for Java 2.x」を参照してください。

AWS リクエスト ID

各リクエストにはリクエスト ID が含まれており、AWS サポート と連携して問題を診断する場合に取得すると便利です。`SdkServiceException` から派生した各例外には、リクエスト ID を取得できる `requestId()` メソッドがあります。

ログ記録

SDK が提供するログ記録を使用すると、クライアントライブラリから重要なメッセージをキャッチする場合や、より詳細なデバッグを行う場合の両方で役に立ちます。ロガーは階層化されており、SDK はルートロガーとして `software.amazon.awssdk` を使用します。レベルは、`TRACE`、`DEBUG`、`INFO`、`WARN`、`ERROR`、`ALL`、または `OFF` のいずれかを使用して設定できます。設定したレベルは、そのロガーとロガー階層の下位レベルに適用されます。

AWS SDK for Java 2.x は、ログ記録に Simple Logging Façade for Java (SLF4J) を使用します。これは他のロガーの抽象化レイヤーとして機能し、これを使用して希望するロガーをプラグインできます。ロガーの接続方法については、「SLF4J ユーザーマニュアル」を参照してください。

各ロガーには特定の動作があります。Log4j 2.x ロガーは、デフォルトで、ログイベントを `System.out` に追加する `ConsoleAppender` を作成し、デフォルトで `ERROR` ログレベルを設定します。

SLF4J に含まれる SimpleLogger ロガーは、デフォルトで `System.err` を出力し、ログレベルはデフォルトで `INFO` になります。

出力量を制限しながら、SDK のクライアントライブラリから重要なメッセージをキャッチできるように、本番環境のデプロイでは `software.amazon.awssdk` のレベルを `WARN` に設定することをお勧めします。

SLF4J がクラスパス上でサポートされているロガーを見つけられない場合 (SLF4J バインディングがない場合)、SLF4J はデフォルトで no operation implementation になります。この実装により、SLF4J がクラスパス上でロガー実装を見つけられなかったことを説明するメッセージが System.err に記録されます。このような状況を防ぐには、ロガー実装を追加する必要があります。これを実行するには、Apache Maven の pom.xml に org.slf4j.slf4j-simple や org.apache.logging.log4j.log4j-slf4j2-imp などのアーティファクトに対する依存関係を追加します。

アプリケーション構成へのログ記録依存関係の追加など、SDK でのログ記録の構成方法については、「AWS SDK for Java デベロッパーガイド」の「Logging with the SDK for Java 2.x」を参照してください。

Log4j2.xml ファイルの次の設定では、Apache Log4j 2 ロガーを使用している場合にログ記録の動作を調整する方法を示しています。この設定では、ルートロガーレベルは WARN に設定しています。このログレベルは、software.amazon.awssdk ロガーなど、階層内のすべてのロガーに継承されます。

デフォルトでは、出力は System.out に送信されます。次の例では、デフォルトの出力 Log4j アペンダを上書きして、カスタマイズした Log4j PatternLayout を適用します。

Log4j2.xml 設定ファイルの例

次の設定では、すべてのロガー階層のコンソールに ERROR レベルと WARN レベルのメッセージがログ記録されます。

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="WARN">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
  </Loggers>
</Configuration>
```

AWS リクエスト ID のログ記録

何らかの問題が発生した場合、例外内でリクエスト ID を検索できます。ただし、例外を生成していないリクエストのリクエスト ID が必要な場合は、ログ記録を使用できます。

リクエスト ID は、DEBUG ロガーによって software.amazon.awssdk.request レベルで出力されます。次の例は、前の configuration example を拡張して、ルートロガーのレベルを ERROR、software.amazon.awssdk のレベルを WARN、software.amazon.awssdk.request のレベルを DEBUG に維持します。このようなレベルを設定すると、リクエスト ID や、エンドポイント、ステータスコードなどのその他のリクエストに関する詳細を取得するうえで役立ちます。

```
<Configuration status="WARN">
  <Appenders>
    <Console name="ConsoleAppender" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{YYYY-MM-dd HH:mm:ss} [%t] %-5p %c:%L - %m%n" />
    </Console>
  </Appenders>

  <Loggers>
    <Root level="ERROR">
      <AppenderRef ref="ConsoleAppender"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

ログ出力の例を次に示します。

```
2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Sending Request:
 DefaultSdkHttpFullRequest(httpMethod=POST, protocol=https, host=dynamodb.us-
east-1.amazonaws.com, encodedPath=/, headers=[amz-sdk-invocation-id, Content-Length,
 Content-Type, User-Agent, X-Amz-Target], queryParameters=[])
 2022-09-23 16:02:08 [main] DEBUG software.amazon.awssdk.request:85 - Received
 successful response: 200, Request ID:
 QS9DUMME2NHEDH8TGT9N5V53OJVV4KQNSO5AEMVJF66Q9ASUAAJG, Extended Request ID: not
 available
```

Pagination (ページ分割)

Query や Scan など、リクエストによっては、1 回のリクエストで返されるデータのサイズが制限され、後続のページを取得するにはリクエストを繰り返し行う必要がある場合があります。

各ページで読み取られる項目の最大数は、Limit パラメータを使用して制御できます。Limit パラメータを使用すると、例えば、10 項目のみを取得することができます。この制限は、フィルタリングが適用される前にテーブルから読み込む項目の上限数を指定します。フィルタリングの結果を 10 項目に正確に指定する方法はありません。事前にフィルタリング済みの数を制御し、実際に 10 項目を取得した場合にのみクライアント側でチェックできます。制限にかかわらず、すべての応答の最大サイズは常に 1 MB です。

API レスポンスには LastEvaluatedKey が含まれる場合があります。これは、制限数または制限サイズに達したためにレスポンスが終了したことを示します。このキーは、応答のために評価された最後のキーです。API を直接操作して、この LastEvaluatedKey を取得し、それを ExclusiveStartKey として後続の呼び出しに渡し、その開始点から次のチャンクを読み取ることができます。LastEvaluatedKey が返されない場合は、Query または Scan API コールに一致する項目がもうなかったことを意味します。

次の例では、低レベルインターフェイスを使用しており、keyConditionExpression パラメータに基づいて項目が 100 に制限されます。

```
QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
        .expressionAttributeValues(Map.of(
                ":pk_val", AttributeValue.fromS("123"),
                ":sk_val", AttributeValue.fromN("1000")))
        .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
        .limit(100)
        .tableName(TABLE_NAME);

while (true) {
    QueryResponse queryResponse = DYNAMODB_CLIENT.query(queryRequestBuilder.build());

    queryResponse.items().forEach(item -> {
        LOGGER.info("item PK: [" + item.get("pk") + "] and SK: [" + item.get("sk") +
 "]");
    });

    if (!queryResponse.hasLastEvaluatedKey()) {
        break;
    }
```

```
    queryRequestBuilder.exclusiveStartKey(queryResponse.lastEvaluatedKey());
}
```

AWS SDK for Java 2.x では、複数のサービスコールを実行して、結果の次のページを自動的に取得する自動ページ分割メソッドを提供し、DynamoDB の操作を簡素化します。これによりコードが簡素化されます。ただし、ページを手動で読み取ることで維持していたリソース使用量がある程度制御できなくなります。

`QueryPaginator` や `ScanPaginator` など、DynamoDB クライアントで利用可能な `Iterable` メソッドを使用すると、SDK がページ分割を処理します。このようなメソッドの戻り値の型はカスタム iterable で、これを使用してすべてのページを反復処理できます。SDK は内部でサービス呼び出しを処理します。Java Stream API を使用すると、次の例に示すとおり `QueryPaginator` の結果を処理できます。

```
QueryIterable results = DYNAMODB_CLIENT.queryPaginator(QueryRequest.builder()
        .expressionAttributeValues(Map.of(
            ":pk_val", AttributeValue.fromS("123"),
            ":sk_val", AttributeValue.fromN("1000")))
        .keyConditionExpression("pk = :pk_val AND sk > :sk_val")
        .limit(100)
        .tableName("YourTableName")
        .build());

results.items().stream()
    .forEach(item -> System.out.println(item.get("itemData")));
```

データクラス注釈

Java SDK は、データクラスの属性に追加できるいくつかのアノテーションを提供します。このようなアノテーションは、SDK が属性を処理する方法に影響します。アノテーションを追加すると、属性を暗黙的なアトミックカウンタとして動作させたり、自動生成したタイムスタンプ値を維持したり、項目のバージョン番号を追跡したりできます。詳細については、「データクラスのアノテーション」を参照してください。

ベストプラクティス

Java SDK のベストプラクティスの詳細については、「AWS SDK for Java 2.x デベロッパーガイド」の「AWS SDK for Java 2.x を使用するためのベストプラクティス」を参照してください。

DynamoDB でのエラー処理

このセクションでは、ランタイムエラーとその処理方法について説明します。同時に、Amazon DynamoDB に特有のエラーメッセージとコードについて説明します。AWS のすべてのサービスに共通するエラーのリストについては、「アクセス管理」を参照してください。

トピック

- エラーコンポーネント
- トランザクションエラー
- エラーメッセージおよびコード
- アプリケーションのエラー処理
- エラーの再試行とエクスポネンシャルバックオフ
- バッチオペレーションとエラー処理

エラーコンポーネント

プログラムがリクエストを送信すると、DynamoDB はその処理を実行するよう試みます。リクエストが成功した場合、DynamoDB はそのオペレーションが出力した結果とともに、 HTTP の成功ステータスコード (`200 OK`) を返します。

リクエストが正常に行われなかった場合、DynamoDB はエラーを返します。それぞれのエラーには、次の三つのコンポーネントがあります:

- HTTP ステータスコード (`400` など)。
- 例外の名前 (`ResourceNotFoundException` など)。
- エラーメッセージ (`Requested resource not found: Table: tablename not found` など)。

AWS SDK によりアプリケーションにエラーが伝達されるため、適切なアクションを実行できます。たとえば、Java プログラムでは、 `try-catch` を処理する `ResourceNotFoundException` ロジックを記述できます。

AWS SDK を使用していない場合は、DynamoDB からの低レベルのレスポンスの内容を、ユーザー側で解析する必要があります。以下に、そのようなレスポンスの例を示します。

```
HTTP/1.1 400 Bad Request
```

```
x-amzn-RequestId: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNSO5AEMF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 240
Date: Thu, 15 Mar 2012 23:56:23 GMT

{"__type":"com.amazonaws.dynamodb.v20120810#ResourceNotFoundException",
"message":"Requested resource not found: Table: tablename not found"}
```

トランザクションエラー

トランザクションエラーの詳細については、「DynamoDB でのトランザクション競合の処理」を参照してください。

エラーメッセージおよびコード

以下に、DynamoDB によって返される例外のリストを、HTTP ステータスコードでグループ分けして示します。再試行してもいいですかがはいであれば、同じリクエストを再度送信できます。再試行してもいいですかがいいえであれば、新しいリクエストを送信する前に、クライアント側で問題を修正する必要があります。

HTTP ステータスコード 400

HTTP ステータスコード400は、認証の失敗、必須パラメータの欠落、またはテーブルにプロビジョニングされているスループットの超過などのリクエストに関連した問題があることを示しています。リクエストを再度送信する前に、アプリケーションで問題を修正する必要があります。

AccessDeniedException

メッセージ: アクセスが拒否されました。

クライアントがリクエストに正しく署名しませんでした。AWS SDK を使用している場合には、リクエストへの署名が自動的に実行されます。使用していない場合は、「AWS 全般のリファレンス」の「署名バージョン 4 の署名プロセス」に進んでください。

再試行してもいいですか。いいえ

ConditionalCheckFailedException

メッセージ: 条件付きリクエストが失敗しました。

false と評価された条件を指定しました。たとえば、項目に条件付き更新を実行しようとしたかもしれませんが、属性の実際の値は、条件の予期される値と一致しませんでした。

再試行してもいいですか。いいえ

IncompleteSignatureException

メッセージ: リクエストの署名が AWS スタンダードに適合しません。

リクエストの署名に、必要なすべての要素が含まれていませんでした。AWS SDK を使用している場合には、リクエストへの署名が自動的に実行されます。使用していない場合は、「AWS 全般のリファレンス」の「署名バージョン 4 の署名プロセス」に進んでください。

再試行してもいいですか。いいえ

ItemCollectionSizeLimitExceededException

メッセージ: コレクションサイズが超過しました。

ローカルセカンダリインデックスがあるテーブルにおいて、同じパーティションキー値を持つ項目のグループが占めるサイズが、最大 10 GB の上限を超過しました。項目コレクションの詳細については、「ローカルセカンダリインデックス内の項目コレクション」を参照してください。

再試行してもいいですか。はい

LimitExceededException

メッセージ: 特定の受信者に対するオペレーションが多すぎます。

同時オペレーションのコントロールプレーンが多すぎます。CREATING、DELETING、または UPDATING の状態のテーブルやインデックスの累積数が、500 を超えることはできません。

再試行してもいいですか。はい

MissingAuthenticationTokenException

メッセージ: リクエストには、有効な (登録済みメンバー) AWS アクセスキー ID が含まれている必要があります。

リクエストに、必要な認可ヘッダーが含まれていないか、または正しい形式ではありません。「」を参照してくださいDynamoDB 低レベル API

再試行してもいいですか。いいえ

ProvisionedThroughputExceededException

メッセージ: 1 つのテーブルまたは 1 つ以上のグローバルセカンダリインデックスで、プロビジョン済みスループットが、許容されている最大量を超えました。プロビジョンドスループットと消費スループットのパフォーマンスメトリクスを表示するには、「Amazon CloudWatch コンソール」を参照してください。

エラーには、スロットリングが発生した理由に関する特定のコンテキストを提供する ThrottlingReason フィールドのリストが含まれ、形式は ResourceType+OperationType +LimitType (例: TableReadProvisionedThroughputExceeded) と、影響を受けるリソースの ARN です。これにより、スロットリング対象のリソース (テーブルまたはインデックス)、スロットリングをトリガーしたオペレーションタイプ (読み取りまたは書き込み)、および超過した特定の制限 (この場合はプロビジョンドキャパシティ) を特定できます。スロットリングの問題の診断と解決の詳細については、「スロットリングの診断」を参照してください。

例: リクエストの頻度が多すぎます。DynamoDB 用の AWS SDK では、この例外を受け取ったリクエストを自動的に再試行します。リクエストは最終的に成功しますが、再試行キューが大きすぎて終了しない場合もあります。エラーの再試行とエクスポネンシャルバックオフ を使用して、リクエストの頻度を少なくします。

再試行してもいいですか。はい

ReplicatedWriteConflictException

メッセージ: このリクエスト内の 1 つ以上の項目が、別のリージョンのリクエストによって変更されています。

例: マルチリージョンの強力な整合性 (MRSC) グローバルテーブル内の項目に対して書き込み操作がリクエストされましたが、現在別のリージョンのリクエストによって変更されています。

再試行してもいいですか。はい

RequestLimitExceeded

メッセージ: スループットがアカウントの現在のスループット制限を超えています。制限の引き上げをリクエストするには、https://aws.amazon.com/support で AWS サポートにお問い合わせください。

エラーには、スロットリングが発生した理由に関する特定のコンテキストを提供する ThrottlingReason フィールドのリストが含まれ、形式は ResourceType+OperationType+LimitType (例: TableWriteAccountLimitExceeded または IndexReadAccountLimitExceeded) と、影響を受けるリソースの ARN です。これにより、スロットリング対象のリソース (テーブルまたはインデックス)、スロットリングをトリガーしたオペレーションタイプ (読み取りまたは書き込み)、アカウントレベルのサービスクォータを超えたことを確認できます。スロットリングの問題の診断と解決の詳細については、「スロットリングの診断」を参照してください。

例: オンデマンドリクエストの速度が、許容されているアカウントスループットを超えていて、これ以上テーブルをスケールできません。

再試行してもいいですか。はい

ResourceInUseException

メッセージ: 変更しようとしているリソースは使用中です。

例: 既存のテーブルを再作成しようとしたか、CREATING 状態にあるテーブルを削除しようとしました。

再試行してもいいですか。いいえ

ResourceNotFoundException

メッセージ: リクエストされたリソースは存在しません。

例: リクエストされたテーブルが存在しないか、ごく初期の CREATING 状態にあります。

再試行してもいいですか。いいえ

ThrottlingException

メッセージ: リクエストの速度が、許容されているスループットを超えています。

この例外は、AmazonServiceException レスポンスとして、TROTTLING_EXCEPTION ステータスコードとともに返されます。この例外は、コントロールプレーン API オペレーションの実行が速すぎる場合に返される可能性があります。

オンデマンドモードを使用するテーブルの場合、リクエストレートが高すぎると、データプレーン API オペレーションでこの例外が返されることがあります。オンデマンドスケーリングの詳細については、「初期スループットとスケーリングのプロパティ」を参照してください。

エラーには、スロットリングが発生した理由に関する特定のコンテキストを提供する ThrottlingReason フィールドのリストが含まれ、形式は ResourceType+OperationType+LimitType (例: TableReadKeyRangeThroughputExceeded または IndexWriteMaxOnDemandThroughputExceeded) と、影響を受けるリソースの ARN です。この情報は、スロットリング対象のリソース (テーブルまたはインデックス)、スロットリングをトリガーしたオペレーションタイプ (読み取りまたは書き込み)、および超過した特定の制限 (パーティション制限またはオンデマンドの最大スループット) を特定するのに役立ちます。スロットリングの問題の診断と解決の詳細については、「スロットリングの診断」を参照してください。

再試行してもいいですか。はい

UnrecognizedClientException

メッセージ: アクセスキー ID またはセキュリティトークンが無効です。

リクエスト署名が間違っています。最も可能性の高い原因は、AWS アクセスキー ID またはシークレットキーが無効であることです。

再試行してもいいですか。はい

ValidationException

メッセージ: 発生した特定のエラーにより異なります

このエラーは、必須パラメータが指定されていない、値が範囲外である、データ型が一致しない、などいくつかの理由で発生します。エラーメッセージに、エラーを引き起こしたリクエストの特定部分に関する詳細が含まれています。

再試行してもいいですか。いいえ

HTTP ステータスコード 5xx

HTTP ステータスコード 5xx は、AWS で解決する必要のある問題を示しています。これは一時的なエラーかもしれず、その場合はリクエストを再試行することで成功する場合があります。それ以外の

場合、サービスにオペレーション上の問題があるかどうかを確認するために、AWS Service Health Dashboard を参照してください。

詳細については、「Amazon DynamoDB の HTTP 5xx エラーを解決する方法を教えてください。」を参照してください。

InternalServerError (HTTP 500)

DynamoDB はリクエストを処理できませんでした。

再試行してもいいですか。はい

> **Note**
>
> 項目の操作中に内部サーバーエラーが発生することがあります。これはテーブルの存続期間中に発生すると予想されます。失敗したリクエストは速やかに再試行できます。書き込みオペレーションでステータスコード 500 を受け取った場合、オペレーションは成功した可能性も、失敗した可能性もあります。書き込みオペレーションが `TransactWriteItem` リクエストである場合、オペレーションを再試行できる状態です。書き込みオペレーションが単一項目の書き込み要求 (`PutItem`、`UpdateItem`、または `DeleteItem` など) である場合、アプリケーションはオペレーションを再試行する前に項目の状態を読み取り、DynamoDB 条件式 CLI の例 を使用して、前のオペレーションが成功したか失敗したかにかかわらず、再試行後も項目が正しい状態のままであることを確認します。冪等性が書き込みオペレーションの要件である場合は、`TransactWriteItem` を使用してください。これは、同じアクションを実行する複数の試行を明確にするために自動的に `ClientRequestToken` を指定することにより、冪等リクエストをサポートします。

ServiceUnavailable (HTTP 503)

DynamoDB は現在利用できません。(これは一時的な状態です。)

再試行してもいいですか。はい

アプリケーションのエラー処理

アプリケーションをスムーズに実行するには、エラーを見つけ、エラーに対応するロジックを組み込む必要があります。一般的な方法には、`try-catch`ブロックや`if-then`ステートメントの使用などがあります。

AWS SDK は独自に再試行とエラーチェックを実行します。AWS SDK の使用中にエラーが発生した場合は、エラーコードと説明が問題のトラブルシューティングに役立ちます。

また、レスポンスに`Request ID`が表示されます。`Request ID` は、AWS サポートを使用して問題を診断することが必要な場合に便利です。

エラーの再試行とエクスポネンシャルバックオフ

特定のリクエストの処理中には、DNS サーバー、スイッチ、ロードバランサーなど、ネットワーク上のさまざまなコンポーネントが原因でエラーが発生する可能性があります。ネットワーク環境でこれらのエラー応答を処理する通常の方法は、クライアントアプリケーションで再試行を実装することです。この手法により、アプリケーションの信頼性が向上します。

各 AWS SDK は、自動的に再試行ロジックを実装しています。必要に応じて再試行パラメータを変更できます。たとえば、エラーが発生したら再試行が許可されない、Fail-Fast 方式を要求する Java アプリケーションについて考えてみましょう。AWS SDK for Java では、`ClientConfiguration` クラスを使用して `maxErrorRetry` の値を `0` に設定することで、再試行を無効にできます。詳細については、AWS SDK ドキュメントのご使用のプログラミング言語の項目を参照してください。

AWS SDK を使用していない場合は、サーバーエラー (5xx) を受け取る元のリクエストを再試行する必要があります。ただし、クライアントエラー (4xx、`ThrottlingException` または `ProvisionedThroughputExceededException` 以外) は、再試行する前にリクエスト自体を修正して問題を解決する必要があることを示しています。特定のスロットリングシナリオに対処するための詳細な推奨事項については、「DynamoDB スロットリングのトラブルシューティング」セクションを参照してください。

簡単な再試行に加えて、各 AWS SDK はエクスポネンシャルバックオフアルゴリズムを実装し、フロー制御を改善します。エクスポネンシャルバックオフは、再試行間の待機時間を累進的に長くして、連続的なエラー応答を受信するという概念に基づいています。たとえば、1 回目の再試行の前に最大 50 ミリ秒、2 回目の前に最大 100 ミリ秒、3 回目の前に最大 200 ミリ秒のようになります。ただし 1 分を経過してもリクエストが成功しない場合は、問題の原因はリクエストの速度ではなく、プロビジョニングしたスループットをリクエストのサイズが超えたためである可能性があります。1

分程度で再試行が停止するように最大回数を設定します。リクエストが失敗した場合は、プロビジョニングされたスループットのオプションを調べてください。

> **Note**
>
> AWS SDK には自動再試行ロジックとエクスポネンシャルバックオフが実装されています。

ほとんどのエクスポネンシャルバックオフアルゴリズムは、連続した衝突を防ぐためにジッター (ランダム化された遅延) を使用します。この場合は、そうした衝突を回避しようとしていないので、乱数を使用する必要はありません。ただし、同時クライアントを使用する場合、ジッターはリクエストをより速く成功させるのに役立ちます。詳細については、「エクスポネンシャルバックオフとジッター」に関するブログ投稿を参照してください。

バッチオペレーションとエラー処理

DynamoDB の低レベル API では、読み込み/書き込みのバッチオペレーションがサポートされています。`BatchGetItem` は 1 つ以上のテーブルからの項目の読み込みを処理し、`BatchWriteItem` は、1 つ以上のテーブルでの項目の入力や削除を処理します。これらのバッチオペレーションは、DynamoDB のバッチではない他のオペレーションのラッパーとして実装されています。つまり、`BatchGetItem` は、バッチの項目ごとに `GetItem` を一度呼び出します。同様に、`BatchWriteItem` は、`DeleteItem` または `PutItem` を必要に応じてバッチの項目ごとに呼び出します。

バッチオペレーションではバッチの個々のリクエストのエラーが許容されます。たとえば、5 つの項目を読み込む `BatchGetItem` リクエストの場合を考えます。基礎となる `GetItem` リクエストの一部が失敗した場合でも、`BatchGetItem` オペレーション全体が失敗することはありません。ただし、5 つのオペレーションにすべて失敗する場合は、`BatchGetItem` 全体が失敗します。

バッチオペレーションでは、失敗した個々のリクエストについて情報が返されるため、問題を診断し、オペレーションを再試行できます。`BatchGetItem` の場合、問題のあるテーブルとプライマリキーがレスポンスの `UnprocessedKeys` 値で返されます。`BatchWriteItem` の場合、同様の情報は `UnprocessedItems` で返されます。

読み込みまたは書き込みの失敗の原因として最も可能性が高いのは、スロットリングです。`BatchGetItem` の場合、バッチリクエストの 1 つ以上のテーブルに、オペレーションをサポートするための十分なプロビジョンド読み込みキャパシティーがなくなります。`BatchWriteItem` の場合、1 つ以上のテーブルに、十分なプロビジョンド書き込みキャパシティーがなくなります。

DynamoDB によって未処理の項目が返された場合は、それらの項目に対してバッチオペレーションを再試行する必要があります。ただし、エクスポネンシャルバックオフアルゴリズムを使用することを強くお勧めします。すぐにバッチオペレーションを再試行した場合、基礎となる読み込みまたは書き込みリクエストはやはり、個々のテーブルに対する帯域幅調整により失敗することがあります。エクスポネンシャルバックオフアルゴリズムを使用してバッチオペレーションを遅らせた場合は、バッチの個々のリクエストが成功する可能性がはるかに高くなります。

> **Note**
>
> 一部の AWS SDK では、未処理の項目の再試行を自動的に処理する高レベルのクライアントが提供されているため、この再試行ロジックを自分で実装する必要はありません。例えば、次のようになります。
>
> - Java – AWS SDK for Java v2 の DynamoDB 拡張クライアントと v1 の DynamoDBMapper はどちらも、バッチオペレーションを実行するときに未処理の項目を自動的に再試行します。
> - Python – boto3 Table リソース `batch_writer` は、バッチ書き込みオペレーションの未処理の項目の再試行を暗黙的に処理します。詳細については、「the section called “batch_writer の使用”」を参照してください。
>
> この動作を提供しない低レベルのクライアントまたは SDK を使用している場合は、上記のように再試行ロジックを自分で実装する必要があります。

AWS SDK で DynamoDB を使用する

AWS Software Development Kit (SDK) は、多くの一般的なプログラミング言語で使用できます。各 SDK には、デベロッパーが好みの言語でアプリケーションを簡単に構築できるようになる API、コード例、およびドキュメントが提供されています。

SDK ドキュメント	コードの例
AWS SDK for C++	AWS SDK for C++ コードの例
AWS CLI	AWS CLI コードの例
AWS SDK for Go	AWS SDK for Go コードの例

SDK ドキュメント	コードの例
AWS SDK for Java	AWS SDK for Java コードの例
AWS SDK for JavaScript	AWS SDK for JavaScript コードの例
AWS SDK for Kotlin	AWS SDK for Kotlin コードの例
AWS SDK for .NET	AWS SDK for .NET コードの例
AWS SDK for PHP	AWS SDK for PHP コードの例
AWS Tools for PowerShell	AWS Tools for PowerShell コードの例
AWS SDK for Python v4	AWS SDK for Python v4 のコード例
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) コードの例
AWS SDK for Ruby	AWS SDK for Ruby コードの例
AWS SDK for Rust	AWS SDK for Rust コードの例
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP コードの例
AWS SDK for Swift	AWS SDK for Swift コードの例

DynamoDB に固有の例については、「AWS SDK を使用した DynamoDB のコード例」を参照してください。

> **可用性の例**
>
> 必要なものが見つからなかった場合。このページの下側にある [Provide feedback] リンクから、コードの例をリクエストしてください。

テーブル、項目、クエリ、スキャン、およびインデックスの使用

このセクションでは、Amazon DynamoDB でのテーブル、項目、クエリ、その他の操作に関する詳細を説明します。

トピック

- DynamoDB でのテーブルとデータの操作
- グローバルテーブル - マルチアクティブ、マルチリージョンレプリケーション
- DynamoDB での項目と属性の操作
- DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上
- DynamoDB でのベクトルインデックスの使用
- DynamoDB トランザクションで複雑なワークフローを管理する
- Amazon DynamoDB の変更データキャプチャ

DynamoDB でのテーブルとデータの操作

このセクションでは、AWS Command Line Interface (AWS CLI) および AWS SDK を使用して Amazon DynamoDB でテーブルを作成、更新、削除する方法を説明します。

> **Note**
>
> このタスクは AWS マネジメントコンソール を使用して実行することもできます。詳細については、「コンソールを使用する」を参照してください。

このセクションでは、DynamoDB Auto Scaling や手動設定のプロビジョンされたスループットを使用し、スループット容量の詳細についても示します。

トピック

- DynamoDB テーブルの基本的なオペレーション
- DynamoDB でテーブルクラスを選択する場合の考慮事項
- DynamoDB でのリソースへのタグとラベルの追加

DynamoDB テーブルの基本的なオペレーション

他のデータベース管理システムと同様、Amazon DynamoDB はデータをテーブルに保存します。いくつかの基本的なオペレーションでテーブルを管理できます。

トピック

- テーブルの作成
- 表の説明
- テーブルの更新
- テーブルの削除
- 削除保護の使用
- テーブル名のリスト化
- プロビジョニングされたスループットクォータの説明

テーブルの作成

`CreateTable` オペレーションを使用して、Amazon DynamoDB でテーブルを作成します。テーブルを作成するために、以下の情報を指定する必要があります。

- [テーブル名] - 。名前は DynamoDB 命名規則に従う必要があります。また、現在の AWS アカウントとリージョンで一意である必要があります。たとえば、米国東部 (バージニア北部) に `People` テーブルを作成し、欧州 (アイルランド) に別の `People` テーブルを作成できます。ただし、これらの 2 つのテーブルは互いにまったく異なっています。詳細については、「Amazon DynamoDB でサポートされるデータ型と命名規則」を参照してください。
- プライマリキー。プライマリキーは 1 つの属性 (パーティションキー) または 2 つの属性で構成できます (パーティションキーとソートキー)。属性名、データタイプ、各属性のロール (パーティションキーでは `HASH`、ソートキーでは `RANGE`) を指定する必要があります。詳細については、「プライマリキー」を参照してください。
- スループット設定 (プロビジョニングされたテーブルの場合)。プロビジョニングモードを使用している場合、最初の読み取りと書き込みのスループット設定をテーブルに指定する必要があります。これらの設定は後から変更できます。また、DynamoDB Auto Scaling を有効化して設定を管理することもできます。詳細については、「DynamoDB プロビジョンドキャパシティモード」および「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

例 1: オンデマンドテーブルを作成する

オンデマンドモードを使用して同じテーブル Music を作成するには

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode=PAY_PER_REQUEST
```

CreateTable オペレーションは、以下のようにテーブルにメタデータを返します。

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 0,
            "ReadCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        },
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
```

```
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397468.348
    }
}
```

> **Important**
>
> オンデマンドテーブルで DescribeTable を呼び出すと、読み込みキャパシティユニットと書き込みキャパシティユニットが 0 に設定されます。

例 2: プロビジョン済みテーブルを作成する

以下の AWS CLI サンプルは、テーブル (Music) の作成方法を示しています。プライマリキーは、Artist (パーティションキー) と SongTitle (ソートキー) で構成されており、データ型はそれぞれ String です。このテーブルの最大スループットは、読み取りキャパシティーユニット数が 10、書き込みキャパシティーユニット数が 5 です。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

CreateTable オペレーションは、以下のようにテーブルにメタデータを返します。

```
{
    "TableDescription": {
```

```
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397215.37
    }
}
```

TableStatus エレメントは、テーブルの現在の状態を示します（CREATING）。ReadCapacityUnits および WriteCapacityUnits に指定した値によって、テーブルの作成にはしばらく時間がかかる場合があります。これらの値を大きくすると、DynamoDB はより多くのリソースをテーブルに割り当てなければならなくなります。

例 3: DynamoDB Standard-Infrequent Access テーブルクラスを使用してテーブルを作成する

DynamoDB Standard-Infrequent Access テーブルクラスを使用して同じ `Music` テーブルを作成するには

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

`CreateTable` オペレーションは、以下のようにテーブルにメタデータを返します。

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "TableClassSummary": {
            "LastUpdateDateTime": 1542397215.37,
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
```

```
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397215.37
    }
}
```

表の説明

テーブルの詳細を表示するには、DescribeTable オペレーションを使用します。テーブル名を入力する必要があります。DescribeTable の出力は CreateTable の出力と同じ形式です。このメタデータには、テーブル作成時のタイムスタンプ、キースキーマ、プロビジョンドスループット設定、推定サイズ、既存のすべてのセカンダリインデックスが含まれています。

> **Important**
>
> オンデマンドテーブルで DescribeTable を呼び出すと、読み込みキャパシティーユニットと書き込みキャパシティーユニットが 0 に設定されます。

Example

```
aws dynamodb describe-table --table-name Music
```

TableStatus が CREATING から ACTIVE に変わると、テーブルは使用できる状態になります。

> **Note**
>
> DescribeTable のリクエスト直後に CreateTable リクエストを発行した場合、DynamoDB によってエラー (ResourceNotFoundException) が返されることがあります。DescribeTable で結果整合性のあるクエリが使用されており、テーブルの

メタデータがその時点で使用できない可能性があるためです。数秒間待ってから、再び DescribeTable リクエストを試してみてください。
請求目的上、DynamoDB ストレージコストには、100 バイトの項目あたりのオーバーヘッドが含まれます。(詳細は、DynamoDB 料金表を参照してください。) 各項目の余分な 100 バイトは、容量単位の計算または DescribeTable オペレーションでは使用されません。

テーブルの更新

UpdateTable オペレーションを使用すると、以下のいずれかを実行できます。

- テーブルのプロビジョニングされたスループット設定を変更します (プロビジョニングモードのテーブルの場合)。
- テーブルの読み取り/書き込みキャパシティーモードを変更します。
- テーブルでグローバルセカンダリインデックスを操作します (「DynamoDB のグローバルセカンダリインデックスの使用」を参照)。
- テーブルの DynamoDB Streams を有効または無効にできます (DynamoDB Streams の変更データキャプチャ を参照)。

Example

次の AWS CLI の例では、テーブルのプロビジョニングされたスループット設定を変更する方法を示します。

```
aws dynamodb update-table --table-name Music \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10
```

> **Note**
>
> UpdateTable リクエストを発行すると、テーブルのステータスが AVAILABLE から UPDATING に変わります。テーブルは UPDATING 中も全面的に利用できます。この処理が完了すると、テーブルのステータスが UPDATING から AVAILABLE に変わります。

Example

次の AWS CLI の例では、テーブルの読み取り/書き込みキャパシティーモードをオンデマンドモードに変更する方法を示します。

```
aws dynamodb update-table --table-name Music \
    --billing-mode PAY_PER_REQUEST
```

テーブルの削除

DeleteTable オペレーションで未使用のテーブルを削除できます。テーブルの削除は回復不可能な操作です。AWS マネジメントコンソール を使用してテーブルを削除するには、「the section called "ステップ 6: (オプション) クリーンアップする"」を参照してください。

Example

次の AWS CLI の例で、テーブルを削除する方法について説明します。

```
aws dynamodb delete-table --table-name Music
```

DeleteTable リクエストを発行すると、テーブルのステータスが ACTIVE から DELETING に変わります。使用しているリソース (テーブルに保存されたデータ、ストリーム、テーブルのインデックスなど) によって、テーブルの削除には時間がかかる場合があります。

DeleteTable オペレーションが完了すると、テーブルは DynamoDB に存在しなくなります。

削除保護の使用

削除保護プロパティを使用すると、テーブルを誤って削除しないように保護できます。テーブルに対してこのプロパティを有効にすると、管理者が通常のテーブル管理オペレーションを行うときにテーブルが誤って削除されるのを防ぐことができます。これにより、通常業務が中断されるのを防ぐことができます。

テーブル所有者または権限のある管理者が、各テーブルの削除保護プロパティを制御します。すべてのテーブルで削除保護プロパティは、デフォルトでオフに設定されています。これには、グローバルレプリカやバックアップから復元されたテーブルが含まれます。テーブルの削除保護が無効になっている場合、Identity and Access Management (IAM) ポリシーによって承認されたすべてのユーザーがそのテーブルを削除できます。テーブルの削除保護が有効になっているときは、誰も削除できません。

この設定を変更するには、表の[Additional settings] (追加設定) に移動し、[Deletion Protection] (削除保護)パネルに移動して、[Enable delete protection] (削除保護を有効にする) を選択します。

削除保護プロパティは、DynamoDB コンソール、API、CLI/SDK、および CloudFormation でサポートされています。CreateTable API は、テーブル作成時の削除保護プロパティをサポートしてお

り、UpdateTable API は既存のテーブルに対する削除保護プロパティの変更をサポートしています。

> **Note**
>
> - AWS アカウントを削除しても、テーブルを含むそのアカウントのすべてのデータは 90 日以内に削除されます。
> - DynamoDB は、テーブルを暗号化したカスタマーマネージドキーにアクセスできない場合、テーブルをアーカイブします。アーカイブには、テーブルのバックアップの作成と元のテーブルの削除が含まれます。

テーブル名のリスト化

ListTables オペレーションは、現在の AWS アカウントやリージョンの DynamoDB テーブルの名前を返します。

Example

次の AWS CLI の例は、DynamoDB テーブル名をリストする方法を示しています。

```
aws dynamodb list-tables
```

プロビジョニングされたスループットクォータの説明

DescribeLimits オペレーションは、現在の AWS アカウントやリージョンの現在の読み取りおよび書き込みキャパシティーのクォータを返します。

Example

次の AWS CLI の例では、現在のプロビジョニングされたスループットクォータを記述する方法を示します。

```
aws dynamodb describe-limits
```

出力は、現在の AWS アカウントやリージョンの読み取りおよび書き込みキャパシティーユニットの上限クォータを返します。

これらのクォータの詳細およびクォータの引き上げをリクエストする方法については、「スループットのデフォルトクォータ」を参照してください。

DynamoDB でテーブルクラスを選択する場合の考慮事項

DynamoDB には、コストの最適化に役立つように設計された 2 つのテーブルクラスが用意されています。DynamoDB 標準テーブルクラスがデフォルトで、大半のワークロードで推奨されています。DynamoDB Standard-Infrequent Access (DynamoDB 標準-IA) テーブルクラスは、ストレージが主要なコストとなるテーブル用に最適化されています。例えば、アプリケーションログ、古いソーシャルメディアの投稿、E コマースの注文履歴、過去のゲーム実績など、アクセス頻度の低いデータを格納するテーブルは、標準-IA テーブルクラスの候補として適しています。

すべての DynamoDB テーブルは、テーブルクラスに関連付けられます。テーブルに関連付けられたすべてのセカンダリインデックスは、同じテーブルクラスを使用します。テーブル (デフォルトでは DynamoDB 標準) の作成時にテーブルクラスを設定できます。また、AWS マネジメントコンソール、AWS CLI、または AWS SDK を使用して、既存のテーブルのテーブルクラスを更新できます。DynamoDB は、シングルリージョンテーブル (グローバルテーブルではないテーブル) に対して AWS CloudFormation を使用したテーブルクラスの管理もサポートしています。テーブルクラスごとに、データストレージ、読み取りおよび書き込みリクエストに対して異なる料金が設定されています。テーブルのテーブルクラスを選択する場合、次の点に注意してください。

- DynamoDB 標準テーブルクラスは、DynamoDB Standard-Infrequent Access (DynamoDB 標準 – IA) よりもスループットコストが低く、スループットが主要なコストであるテーブルで最もコスト効率の高いオプションです。
- DynamoDB Standard-Infrequent Access (DynamoDB 標準 – IA) テーブルクラスは、DynamoDB 標準よりも低いストレージコストを提供し、ストレージが主要なコストであるテーブルで最もコスト効率の高いオプションです。ストレージが DynamoDB 標準テーブルクラスを使用するテーブルのスループット (読み取りと書き込み) コストの 50% を超える場合、DynamoDB Standard-Infrequent Access (DynamoDB 標準 – IA) テーブルクラスを使用すると、テーブルの総コストを削減できます。
- DynamoDB 標準 IA テーブルは、DynamoDB 標準テーブルと同じパフォーマンス、耐久性、可用性を提供します。
- DynamoDB 標準テーブルクラスと DynamoDB 標準 — IA テーブルクラスを入れ替えても、アプリケーションコードを変更する必要はありません。テーブルで使用するテーブルクラスに関係なく、同じ DynamoDB API とサービスエンドポイントを使用します。
- DynamoDB 標準 IA テーブルは、Auto Scaling、オンデマンドモード、有効期限 (TTL)、オンデマンドバックアップ、ポイントインタイムリカバリ (PITR)、グローバルセカンダリインデックスなど、既存の DynamoDB に備わる機能のすべてと互換性があります。

テーブルにとって最も費用対効果の高いテーブルクラスは、そのテーブルの予想されるストレージとスループットの使用パターンによって異なります。テーブルのストレージとスループットのコストおよび使用状況の履歴は、AWS コストと使用状況レポート、および AWS Cost Explorer から確認できます。この履歴データを使用して、テーブルに対して最も費用対効果の高いテーブルクラスを特定します。AWS コストと使用状況レポート、および AWS Cost Explorer の使用の詳細については、AWS Billing and Cost Management のドキュメントを参照してください。テーブルクラスの料金の詳細については、Amazon DynamoDB Pricing (Amazon DynamoDB の料金表) を参照してください。

> **Note**
>
> テーブルクラスの更新はバックグラウンドで処理されます。テーブルクラスの更新中も、通常どおりテーブルにアクセスできます。テーブルクラスを更新する時間は、テーブルトラフィック、ストレージサイズ、およびその他の関連する変数によって異なります。30 日間の追跡期間では、最大 2 つのテーブルクラスの更新が可能です。

DynamoDB でのリソースへのタグとラベルの追加

Amazon DynamoDB リソースにラベルを付けるには、タグを使用します。タグを使用すると、リソースを目的、所有者、環境、その他の条件などさまざまな方法で分類することができます。タグは、以下のことに役立ちます。

- リソースに割り当てたタグに基づいてリソースをすばやく特定します。
- タグ別に分類された AWS 請求を表示する。

> **Note**
>
> タグが付けられたテーブルに関連するローカルセカンダリインデックス (LSI) およびグローバルセカンダリインデックス (GSI) は、自動的に同じタグでラベルが付けられます。現在のところ、DynamoDB Streams の使用にタグを付けることはできません。

タグ付けは、Amazon EC2、Amazon S3、DynamoDB などの AWS のサービスでサポートされています。効率的なタグ付けを行うと、特定のタグを持つサービス間でレポートを作成でき、コストインサイトを得ることができます。

タグ付けの使用を開始するには、次のことを行います。

1. DynamoDB でのタグ付けの制限 について理解します。
2. DynamoDB でのタグ付けのリソース を使用してタグを作成します。
3. DynamoDB タグを使用して、コスト配分レポートを作成する を使用して、アクティブなタグごとに AWS のコストを追跡します。

最後に、最適のタグ付け手法に従うことをお勧めします。詳細については、「AWS tagging strategies (タグ付け戦略)」を参照してください。

DynamoDB でのタグ付けの制限

タグはそれぞれ、1 つのキーと 1 つの値で構成されており、どちらもお客様側が定義します。以下の制限が適用されます。

- 各 DynamoDB テーブルは同じキーを含む 1 つのタグのみを持つことができます。既存のタグ (同じキー) を追加しようとすると、既存のタグの値は新しい値に更新されます。
- タグのキーと値は大文字と小文字が区別されます。
- キーの最大長は Unicode 文字で 128 文字です。
- 値の最大長は Unicode 文字で 256 文字です。
- 使用できる文字は、文字、ホワイトスペース、数字、および特殊文字 (+ - = . _ : /) です。
- リソースあたりのタグの最大数は 50 です。
- テーブル内のすべてのタグにサポートされる最大サイズは 10 KB です。
- AWS 割り当てタグの名前と値には、`aws:` プレフィックスが自動的に割り当てられます。これをユーザーが割り当てることはできません。AWS 割り当てタグの名前は、タグ制限数の 50 または最大サイズ制限の 10 KB にカウントされません。ユーザー側で割り当てたタグ名は、`user:` というプレフィックスを付けてコスト配分レポートに表示されます。
- 過去にさかのぼってタグを適用することはできません。

DynamoDB でのタグ付けのリソース

Amazon DynamoDB コンソールまたは AWS Command Line Interface (AWS CLI) を使用して、タグを追加、リスト、編集、または削除できます。次に、これらのユーザー定義タグをアクティブ化し、コスト配分の追跡のため、AWS Billing and Cost Management コンソールに表示できます。詳細については、「DynamoDB タグを使用して、コスト配分レポートを作成する」を参照してください。

一括編集の場合は、AWS マネジメントコンソール のタグエディタを使用することもできます。詳細については、「タグエディタの使用」を参照してください。

代わりに DynamoDB API を使用するには、Amazon DynamoDB API リファレンスで以下のオペレーションを参照してください。

- TagResource
- UntagResource
- ListTagsOfResource

トピック

- タグでフィルターする権限の設定
- 新しいテーブルまたは既存のテーブルへのタグの追加 (AWS マネジメントコンソール)
- 新しいテーブルまたは既存のテーブルへのタグの追加 (AWS CLI)

タグでフィルターする権限の設定

タグを使用して DynamoDB コンソールでテーブルリストをフィルタリングするには、ユーザーのポリシーに以下のオペレーションへのアクセス権が含まれていることを確認します。

- `tag:GetTagKeys`
- `tag:GetTagValues`

これらのオペレーションにアクセスするには、以下の手順に従って、新しい IAM ポリシーをユーザーにアタッチします。

1. 管理者ユーザーで IAM コンソールに移動します。
2. 左のナビゲーションメニューで [Policies (ポリシー)] を選択します。
3. [Create policy (ポリシーの作成)] を選択します。
4. 以下のポリシーを JSON エディタに貼り付けます。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "tag:GetTagKeys",
                "tag:GetTagValues"
            ],
            "Resource": "*"
        }
    ]
}
```

5. ウィザードを完了し、ポリシーに名前を割り当てます (例:TagKeysAndValuesReadAccess)。
6. 左のナビゲーションメニューで [Users (ユーザー)] を選択します。
7. リストから、DynamoDB コンソールにアクセスする通常のユーザーを選択します。
8. [Add permissions (アクセス許可の追加)] を選択します。
9. [Attach existing policies directly (既存のポリシーを直接アタッチ)] を選択します。
10. リストから、前に作成したポリシーを選択します。
11. ウィザードを終了します。

新しいテーブルまたは既存のテーブルへのタグの追加 (AWS マネジメントコンソール)

新しいタグを作成するときや、既存のテーブルのタグを追加、編集、削除するときは、DynamoDB コンソールを使用してタグを新しいテーブルに追加できます。

作成時にリソースにタグを付けるには (コンソール)

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインで [テーブル] を選択して、[テーブルの作成] を選択します。
3. [Create DynamoDB table (DynamoDB テーブルの作成)] ページで、名前とプライマリキーを指定します。[タグ] セクションで、[新しいタグを追加] を選択し、使用するタグを入力します。

   タグ構造の詳細については、「DynamoDB でのタグ付けの制限」を参照してください。

   テーブル作成の詳細については、「DynamoDB テーブルの基本的なオペレーション」を参照してください。

既存のリソースにタグを付けるには (コンソール)

https://console.aws.amazon.com/dynamodb/ で DynamoDB コンソールを開きます。

1. ナビゲーションペインで [テーブル] を選択します。
2. リストでテーブルを選択し、[追加設定] タブを選択します。ページの下部にある [タグ] セクションで、タグを追加、編集、削除できます。

新しいテーブルまたは既存のテーブルへのタグの追加 (AWS CLI)

次の例は、テーブルとインデックスを作成するとき、および既存のリソースへのタグを付けるときに、AWS CLI を使用してタグを指定する方法を示しています。

作成時にリソースにタグを付けるには (AWS CLI)

- 次の例では、新しい `Movies` テーブルを作成し、値が `Owner` の `blueTeam` タグを追加します。

```
aws dynamodb create-table \
    --table-name Movies \
    --attribute-definitions AttributeName=Title,AttributeType=S \
    --key-schema AttributeName=Title,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

既存のリソースにタグを付けるには (AWS CLI)

- 次の例では、`Owner` テーブル用に、値が `blueTeam` の `Movies` タグを追加します。

```
aws dynamodb tag-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies \
    --tags Key=Owner,Value=blueTeam
```

テーブルのすべてのタグを一覧表示するには (AWS CLI)

- 次の例では、`Movies` テーブルに関連付けられたすべてのタグを一覧表示します。

```
aws dynamodb list-tags-of-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies
```

DynamoDB タグを使用して、コスト配分レポートを作成する

AWS はタグを使用して、コスト配分レポートでリソースコストを分類します。AWS には 2 つのタイプのコスト配分タグがあります。

- AWS で生成されたタグ。AWS はユーザーのためにこのタグを定義、作成、適用します。
- ユーザー定義のタグ。これらのタグを定義、作成、適用します。

Cost Explorer またはコスト配分レポートで使用するには、事前に両方のタイプのタグを別々にアクティブ化しておく必要があります。

AWS で生成されたタグをアクティブ化するには:

1. AWS マネジメントコンソール にサインインし、https://console.aws.amazon.com/billing/home#/ で請求およびコスト管理コンソールを開きます。
2. ナビゲーションペインで、[Cost Allocation Tags] (コスト配分タグ) を選択します。
3. [AWS-Generated Cost Allocation Tags] で、[Activate] を選択します。

ユーザー定義タグをアクティブにするには:

1. AWS マネジメントコンソール にサインインし、https://console.aws.amazon.com/billing/home#/ で請求およびコスト管理コンソールを開きます。
2. ナビゲーションペインで、[Cost Allocation Tags] (コスト配分タグ) を選択します。
3. [User-Defined Cost Allocation Tags] (ユーザー定義のコスト配分タグ) で、[Activate] (アクティブ化) を選択します。

タグを作成してアクティブ化すると、AWS は、アクティブなタグごとにグループ化された使用量とコストを含むコスト配分レポートを生成します。コスト配分レポートには、ご利用の AWS のサービスのコストすべてが請求期間別に出力されます。タグ付きとタグなしのどちらのリソースもこのレポートに出力されるので、リソース別の請求額を明確に分類できます。

> **Note**
>
> 現時点では、DynamoDB から転送されたデータは、コスト配分レポートのタグによって分類されません。

詳細については、「Using cost allocation tags」(コスト配分タグの使用) を参照してください。

グローバルテーブル - マルチアクティブ、マルチリージョンレプリケーション

Amazon DynamoDB グローバルテーブルは、フルマネージドで、マルチリージョン、マルチアクティブなデータベース機能で、グローバルにスケールされたアプリケーション向けに、使いやすいレプリケーションと高速なローカルの読み込みと書き込みのパフォーマンスを提供します。

グローバルテーブルは、独自のレプリケーションソリューションを構築および維持することなく、AWS リージョン間、オプションで AWS アカウント間で DynamoDB テーブルデータを自動的にレプリケートします。グローバルテーブルは、マルチリージョンデプロイを通じてビジネス継続性と高可用性を必要とするアプリケーションに最適です。グローバルテーブルのレプリカは、読み取りと書き込みの両方を処理できます。アプリケーションは、リージョンでアプリケーション処理が中断された場合にトラフィックを別のリージョンに移行することで、目標復旧時点 (RPO) を低くまたはゼロにして高い回復力を実現できます。グローバルテーブルは、DynamoDB が利用可能なすべてのリージョンで使用できます。

整合性モード

グローバルテーブルを作成するときに、その整合性モードを設定できます。グローバルテーブルは、マルチリージョンの結果整合性 (MREC) およびマルチリージョンの強力な整合性 (MRSC) の 2 つの整合性モードをサポートしています。

グローバルテーブルの作成時に整合性モードを指定しない場合、グローバルテーブルはデフォルトでマルチリージョンの結果整合性 (MREC) になります。グローバルテーブルには、異なる整合性モードで設定されたレプリカを含めることはできません。グローバルテーブルの整合性モードを作成後に変更することはできません。

アカウント設定

DynamoDB は現在、それぞれ異なるアーキテクチャパターン用に設計された 2 つのグローバルテーブルモデルをサポートしています。

- 同一アカウントグローバルテーブル – すべてのレプリカは 1 つの AWS アカウント内で作成および管理されます。
- マルチアカウントグローバルテーブル – レプリカは、共有レプリケーショングループに参加している間、複数の AWS アカウントにデプロイされます。

同一アカウントモデルとマルチアカウントモデルの両方で、マルチリージョン書き込み、非同期レプリケーション、最後の書き込み優先の競合解決、および同じ請求モデルがサポートされます。ただし、アカウント、アクセス許可、暗号化、テーブルガバナンスの管理方法は異なります。

MRSC 用に設定されたグローバルテーブルは、同じアカウント設定のみをサポートします。

AWS マネジメントコンソールを使用してグローバルテーブルを設定できます。グローバルテーブルは、既存の DynamoDB API を使用してテーブルにデータを読み書きするため、アプリケーションの変更は必要ありません。プロビジョニングまたは使用したリソースに対してのみ料金が発生します。前払い料金や契約はありません。

同一アカウントおよびマルチアカウントグローバルテーブルの比較

プロパティ	同一アカウントグローバルテーブル	マルチアカウントグローバルテーブル
主なユースケース	単一 AWS アカウント内のアプリケーションのマルチリージョン回復性	異なるチーム、異なるビジネスユニット、またはアカウント間の強力なセキュリティ境界が所有するアプリケーションのための、マルチリージョン、マルチアカウントレプリケーション
アカウントモデル:	すべてのレプリカは 1 つの AWS アカウントで作成および管理されます	同じデプロイ内の複数の AWS アカウントに作成されたレプリカ
リソース所有権	1 つのアカウントがテーブルとすべてのレプリカを所有します	各アカウントはローカルレプリカを所有し、レプリケーショングループはアカウントにまたがります
サポートされているバージョン	グローバルテーブルバージョン 2019.11.21 (現行) およびバージョン 2017.11.29 (レガシー)	グローバルテーブルバージョン 2019.11.21 (現行)

プロパティ	同一アカウントグローバルテーブル	マルチアカウントグローバルテーブル
コントロールプレーンオペレーション	テーブル所有者アカウントを使用してレプリカを作成、変更、削除します	分散コントロールプレーンオペレーション: アカウントがレプリケーショングループに参加または退出します
データプレーンオペレーション	リージョンあたりの標準 DynamoDB エンドポイント	アカウント/リージョンあたりのデータプレーンアクセス、レプリケーショングループ経由のルーティング
セキュリティ境界	単一の IAM と KMS の境界	アカウントごとに異なる IAM、KMS、請求、CloudTrail、ガバナンス
最適な用途	テーブルの一元的な所有権を持つ組織	フェデレーティッドチーム、ガバナンスの境界、またはマルチアカウント設定を持つ組織

トピック

- グローバルテーブルの重要な概念
- DynamoDB 同一アカウントグローバルテーブル
- DynamoDB マルチアカウントグローバルテーブル
- Amazon DynamoDB のグローバルテーブルの請求について
- DynamoDB グローバルテーブルのバージョン
- グローバルテーブルのベストプラクティス

グローバルテーブルの重要な概念

以下のセクションでは、Amazon DynamoDB におけるグローバルテーブルの概念および動作について説明します

概念

グローバルテーブルは、AWS リージョン間でテーブルデータをレプリケートする DynamoDB 機能です。

レプリカテーブル (または、レプリカ) は、グローバルテーブルの一環として機能する DynamoDB テーブルです。グローバルテーブルは、異なる AWS リージョンにまたがる 2 つ以上のレプリカテーブルで構成されます。各グローバルテーブルは、AWS リージョンごとに 1 つのレプリカのみを持つことができます。グローバルテーブル内のすべてのレプリカは、同じテーブル名、プライマリキースキーマ、および項目データを共有します。

アプリケーションが 1 つのリージョンのレプリカにデータを書き込むと、DynamoDB は自動的にその書き込みをグローバルテーブル内の他のすべてのレプリカにレプリケートします。グローバルテーブルの詳しい使用方法については、「チュートリアル: グローバルテーブルの作成」または「チュートリアル: マルチアカウントグローバルテーブルの作成」を参照してください。

バージョン

DynamoDB グローバルテーブルには、グローバルテーブルバージョン 2019.11.21 (現行) と グローバルテーブルバージョン 2017.11.29 (レガシー) の 2 つのバージョンがあります。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用する必要があります。このドキュメントセクションの情報は、バージョン 2019.11.21 (現行) 用です。詳細については、「グローバルテーブルのバージョンを確認するグローバルテーブルのバージョンを確認する」を参照してください。

可用性

グローバルテーブルは、マルチリージョンの高可用性アーキテクチャの実装を容易にすることで、ビジネス継続性の向上に役立ちます。1 つの AWS リージョンのワークロードに障害が発生した場合、アプリケーショントラフィックを別のリージョンに移行し、同じグローバルテーブル内の別のレプリカテーブルに対して読み取りと書き込みを実行できます。

グローバルテーブル内の各レプリカテーブルは、単一リージョンの DynamoDB テーブルと同じ耐久性と可用性を提供します。グローバルテーブルは 99.999% の可用性サービスレベルアグリーメント (SLA) を提供しますが、単一リージョンテーブルでは 99.99% です。

フォールトインジェクションテスト

MREC と MRSC のグローバルテーブルはどちらも、制御されたフォールトインジェクション実験を実行してアプリケーションの耐障害性を向上させるためのフルマネージドサービスである AWS

Fault Injection Service (AWS FIS) と統合されています。AWS FIS を使用すると、次のことができます。

- 特定の障害シナリオを定義する実験テンプレートを作成します。
- アプリケーションの耐障害性を検証するため、リージョンの分離 (つまり、選択したレプリカとのレプリケーションを一時停止すること) をシミュレートして障害を注入し、エラー処理、復旧メカニズム、1 つの AWS リージョンで中断が発生した場合のマルチリージョントラフィックシフトの動作をテストします。

例えば、米国東部 (バージニア北部)、米国東部 (オハイオ)、米国西部 (オレゴン) にレプリカがあるグローバルテーブルでは、米国東部 (バージニア北部) と米国西部 (オレゴン) が通常のオペレーションを継続している間に、米国東部 (オハイオ) で実験を実行してリージョンの分離をテストできます。この制御されたテストは、本番稼働用ワークロードに影響を与える前に潜在的な問題を特定して解決するのに役立ちます。

AWS FIS でサポートされているアクションの完全なリストについては、「AWS FIS ユーザーガイド」の「アクションターゲット」を参照してください。また、リージョン間の DynamoDB レプリケーションを一時停止するには、「クロスリージョン接続」を参照してください。

AWS FIS で使用できる Amazon DynamoDB グローバルテーブルアクションの詳細については、「AWS FIS ユーザーガイド」の「DynamoDB グローバルテーブルアクションリファレンス」を参照してください。

フォールトインジェクション実験の実行を開始するには、「AWS FIS ユーザーガイド」の「AWS FIS 実験の計画」を参照してください。

> **Note**
>
> MRSC での AWS FIS 実験中は、結果整合性のある読み込みが許可されますが、MREC と同様に請求モードの変更やテーブルスループットの設定などのテーブル設定の更新は許可されません。エラーコードの詳細については、CloudWatch メトリクス `FaultInjectionServiceInducedErrors` を確認してください。

有効期限 (TTL)

MREC 用に設定されたグローバルテーブルは有効期限 (TTL) 削除の設定をサポートしています。TTL 設定は、グローバルテーブル内のすべてのレプリカに対して自動的に同期されます。TTL がリー

ジョン内のレプリカから項目を削除すると、削除はグローバルテーブル内の他のすべてのレプリカにレプリケートされます。TTL は書き込み容量を消費しないため、削除が発生したリージョンでは TTL 削除に対して課金されません。ただし、グローバルテーブルのレプリカが存在する他のリージョンでレプリケートされた削除に対しては課金されます。

TTL 削除レプリケーションは、削除がレプリケートされるレプリカの書き込み容量を消費します。プロビジョンドキャパシティ用に設定されたレプリカは、書き込みスループットと TTL 削除スループットの組み合わせがプロビジョンド書き込みキャパシティを超える場合、リクエストをスロットリングすることがあります。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、有効期限 (TTL) 削除の設定をサポートしていません。

Streams

マルチリージョンの結果整合性 (MREC) 用に設定されたグローバルテーブルは、レプリカテーブルの DynamoDB Stream からこれらの変更を読み取り、その変更を他のすべてのレプリカテーブルに適用することで、変更をレプリケートします。したがって、Streams は MREC グローバルテーブル内のすべてのレプリカでデフォルトで有効になっており、それらのレプリカで無効にすることはできません。MREC レプリケーションプロセスでは、短期間に複数の変更を 1 つのレプリケートされた書き込みに結合する場合があり、各レプリカの Stream にわずかに異なるレコードが含まれる場合があります。MREC レプリカのストリームレコードは、同じ項目に対するすべての変更の順序を維持しますが、異なる項目に対する変更の相対的な順序はレプリカによって異なる場合があります。

グローバルテーブル内の他のリージョンではなく、特定のリージョンで発生した変更について Streams レコードを処理するアプリケーションを作成する場合は、各項目にその項目の変更が発生したリージョンを定義する属性を追加できます。この属性を使用して、特定のリージョンの変更に対してのみ Lambda 関数を呼び出す Lambda イベントフィルターの使用など、他のリージョンで発生した変更について Streams レコードをフィルタリングできます。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、レプリケーションに DynamoDB Streams を使用しないため、MRSC レプリカでは Streams はデフォルトで有効になっていません。MRSC レプリカで Streams を有効にできます。MRSC レプリカの Streams レコードは、Stream レコードの順序付けを含め、すべてのレプリカで同じです。

取引

MREC 用に設定されたグローバルテーブルでは、DynamoDB トランザクションオペレーション (`TransactWriteItems` および `TransactGetItems`) は、操作が呼び出されたリージョン内でのみ

アトミックです。トランザクション書き込みはリージョン間でユニットとしてレプリケートされません。つまり、特定の時点で、トランザクション内の書き込みの一部のみが、他のレプリカでの読み取り操作によって返される可能性があります。

例えば、米国東部 (オハイオ) および米国西部 (オレゴン) リージョンにレプリカを持つグローバルテーブルがあり、米国東部 (オハイオ) リージョンで `TransactWriteItems` オペレーションを実行すると、変更がレプリケートされたときに米国西部 (オレゴン) では部分的に完了したトランザクションが観察されることがあります。変更は、ソースリージョンでコミットされた後にのみ、他のリージョンにレプリケートされます。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルはトランザクションオペレーションをサポートしておらず、これらのオペレーションが MRSC レプリカで呼び出されるとエラーが返されます。

読み取りおよび書き込みスループット

プロビジョニングモード

レプリケーションは書き込み容量を消費します。プロビジョンドキャパシティ用に設定されたレプリカは、アプリケーションの書き込みスループットとレプリケーション書き込みスループットの組み合わせがプロビジョンド書き込みキャパシティを超えると、リクエストをスロットリングすることがあります。プロビジョニングモードを使用するグローバルテーブルの場合、読み取り容量と書き込み容量の両方の自動スケーリング設定がレプリカ間で同期されます。

レプリカレベルで `ProvisionedThroughputOverride` パラメータを使用して、グローバルテーブル内の各レプリカの読み取り容量設定を個別に設定できます。デフォルトでは、プロビジョニングされた読み取り容量への変更は、グローバルテーブル内のすべてのレプリカに適用されます。グローバルテーブルに新しいレプリカを追加する場合、レプリカレベルのオーバーライドが明示的に指定されていない限り、ソーステーブルまたはレプリカの読み取り容量が初期値として使用されます。

オンデマンドモード

オンデマンドモードに設定されたグローバルテーブルの場合、書き込み容量はすべてのレプリカ間で自動的に同期されます。DynamoDB はトラフィックに基づいて容量を自動的に調整するため、レプリカ固有の読み取りまたは書き込み容量設定を管理する必要はありません。

グローバルテーブルのモニタリング

マルチリージョンの結果整合性 (MREC) 用に設定されたグローバルテーブルは `ReplicationLatency` メトリクスを CloudWatch に発行します。このメトリクスは、項目がレプ

リカテーブルに書き込まれてから、その項目がグローバルテーブルの別のレプリカに表示されるまでの経過時間を追跡します。ReplicationLatency はミリ秒単位で表し、グローバルテーブル内のすべての送信元と送信先のリージョンペアに対して出力されます。

一般的な ReplicationLatency 値は、選択した AWS リージョン間の距離と、ワークロードタイプやスループットなどの他の変数によって異なります。例えば、米国西部 (北カリフォルニア) (us-west-1) リージョンのソースレプリカは、アフリカ (ケープタウン) (af-south-1) リージョンと比較して、米国西部 (オレゴン) (us-west-2) リージョンへの ReplicationLatency が低くなります。

ReplicationLatency の値が上昇している場合、1 つのレプリカからの更新が他のレプリカテーブルにタイムリーに伝播されていないことを示している可能性があります。この場合、アプリケーションの読み込みおよび書き込みアクティビティを別の AWS リージョンに一時的にリダイレクトすることができます。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、ReplicationLatency メトリクスを発行しません。

グローバルテーブルの管理に関する考慮事項

新しいグローバルテーブルレプリカの追加に使用したテーブルは、新しいレプリカが作成されてから 24 時間が経過するまで削除できません。

グローバルテーブルレプリカを含む AWS リージョンを無効にすると、それらのレプリカは、リージョンが無効になってから 20 時間後に単一リージョンテーブルに永続的に変換されます。

DynamoDB 同一アカウントグローバルテーブル

同一アカウントグローバルテーブルは、1 つの AWS アカウント内の AWS リージョン間で DynamoDB テーブルデータを自動的にレプリケートします。同一アカウントグローバルテーブルは、すべてのレプリカが同じアカウントの境界、所有権、アクセス許可モデルを共有するため、マルチリージョンアプリケーションを実行するための最もシンプルなモデルを提供します。レプリカテーブル用の AWS リージョンを選択すると、グローバルテーブルはすべてのレプリケーションを自動的に処理します。グローバルテーブルは、DynamoDB が利用可能なすべてのリージョンで使用できます。

同一アカウントグローバルテーブルには以下の利点があります。

- 選択した AWS リージョン間で DynamoDB テーブルデータを自動的にレプリケートして、ユーザーに近いデータを配置する

- リージョンの分離またはパフォーマンス低下時にアプリケーションの可用性を高める
- 組み込みの競合の解決を使用して、アプリケーションのビジネスロジックに集中できるようにする
- 同一アカウントグローバルテーブルを作成するときは、マルチリージョンの結果整合性 (MREC) または マルチリージョンの強力な整合性 (MRSC) のいずれかを選択できます。

トピック

- DynamoDB グローバルテーブルの仕組み
- チュートリアル: グローバルテーブルの作成
- DynamoDB グローバルテーブルのセキュリティ

DynamoDB グローバルテーブルの仕組み

以下のセクションでは、Amazon DynamoDB におけるグローバルテーブルの概念および動作について説明します。

概念

グローバルテーブルは、AWS リージョン間でテーブルデータをレプリケートする DynamoDB 機能です。

レプリカテーブル (または、レプリカ) は、グローバルテーブルの一環として機能する DynamoDB テーブルです。グローバルテーブルは、異なる AWS リージョンにまたがる 2 つ以上のレプリカテーブルで構成されます。各グローバルテーブルは、AWS リージョンごとに 1 つのレプリカのみを持つことができます。グローバルテーブル内のすべてのレプリカは、同じテーブル名、プライマリキースキーマ、および項目データを共有します。

アプリケーションが 1 つのリージョンのレプリカにデータを書き込むと、DynamoDB は自動的にその書き込みをグローバルテーブル内の他のすべてのレプリカにレプリケートします。グローバルテーブルの詳しい使用方法については、チュートリアル: グローバルテーブルの作成 を参照してください。

バージョン

DynamoDB グローバルテーブルには、バージョン 2019.11.21 (現行) とバージョン 2017.11.29 (レガシー) の 2 つのバージョンがあります。可能な限り、バージョン 2019.11.21 (現行) を使用する必要があります。このドキュメントセクションの情報は、バージョン 2019.11.21 (現行) 用です。詳細については、「グローバルテーブルのバージョンを確認する」を参照してください。

可用性

グローバルテーブルは、マルチリージョンの高可用性アーキテクチャの実装を容易にすることで、ビジネス継続性の向上に役立ちます。1 つの AWS リージョンのワークロードに障害が発生した場合、アプリケーショントラフィックを別のリージョンに移行し、同じグローバルテーブル内の別のレプリカテーブルに対して読み取りと書き込みを実行できます。

グローバルテーブル内の各レプリカテーブルは、単一リージョンの DynamoDB テーブルと同じ耐久性と可用性を提供します。グローバルテーブルは 99.999% の可用性サービスレベルアグリーメント (SLA) を提供しますが、単一リージョンテーブルでは 99.99% です。

整合性モード

グローバルテーブルを作成するときに、その整合性モードを設定できます。グローバルテーブルは、マルチリージョンの結果整合性 (MREC) およびマルチリージョンの強力な整合性 (MRSC) の 2 つの整合性モードをサポートしています。

グローバルテーブルの作成時に整合性モードを指定しない場合、グローバルテーブルはデフォルトでマルチリージョンの結果整合性 (MREC) になります。グローバルテーブルには、異なる整合性モードで設定されたレプリカを含めることはできません。グローバルテーブルの整合性モードを作成後に変更することはできません。

マルチリージョンの結果整合性 (MREC)

マルチリージョンの結果整合性 (MREC) は、グローバルテーブルのデフォルトの整合性モードです。MREC グローバルテーブルレプリカの項目の変更は、通常は 1 秒以内で他のすべてのレプリカに非同期でレプリケートされます。万一、MREC グローバルテーブルのレプリカで分離または障害が発生した場合、他のリージョンにまだレプリケートされていないデータは、レプリカが正常になるとレプリケートされます。

同じ項目が複数のリージョンで同時に変更された場合、DynamoDB は項目ごとに最新の内部タイムスタンプを使用して競合を解決します。これは「最終書き込み者優先」競合解決方法と呼ばれます。最終的に、項目はすべてのレプリカで最後の書き込みによって作成されたバージョンに収束します。

強力な整合性のある読み込みオペレーションは、その項目が読み取りが発生したリージョンで最後に更新された場合は項目の最新バージョンを返しますが、項目が別のリージョンで最後に更新された場合は古いデータを返す可能性があります。条件付き書き込みは、リージョン内の項目のバージョンに対して条件式を評価します。

MREC グローバルテーブルを作成するには、既存の DynamoDB テーブルにレプリカを追加します。レプリカを追加しても、既存の単一リージョン DynamoDB テーブルまたはグローバルテーブルレプ

リカのパフォーマンスには影響しません。MREC グローバルテーブルにレプリカを追加して、データがレプリケートされるリージョンの数を拡張したり、不要になったレプリカを MREC グローバルテーブルから削除したりできます。MREC グローバルテーブルは、DynamoDB が利用可能な任意のリージョンにレプリカを持つことができ、AWS パーティション内のリージョンと同じ数のレプリカを持つことができます。

マルチリージョンの強力な整合性 (MRSC)

グローバルテーブルを作成する際に、マルチリージョンの強力な整合性 (MRSC) モードを設定できます。MRSC グローバルテーブルレプリカの項目の変更は、書き込み操作が正常なレスポンスを返す前に、少なくとも 1 つの他のリージョンに同期的にレプリケートされます。MRSC レプリカに対する強力な整合性のある読み込み操作は、常に項目の最新バージョンを返します。条件付き書き込みは、常に条件式を項目の最新バージョンと照合して評価します。

MRSC グローバルテーブルは、厳密に 3 つのリージョンにデプロイする必要があります。MRSC グローバルテーブルは、3 つのレプリカ、または 2 つのレプリカと 1 つの監視で設定できます。監視は、グローバルテーブルレプリカに書き込まれたデータを含む MRSC グローバルテーブルのコンポーネントであり、MRSC の可用性アーキテクチャをサポートしながら、完全なレプリカに代わるオプションを提供します。監視に対して読み取りまたは書き込み操作を実行することはできません。監視は、2 つのレプリカとは異なるリージョンにあります。MRSC グローバルテーブルを作成する際、MRSC テーブルの作成時にレプリカと監視デプロイの両方のリージョンを選択します。MRSC グローバルテーブルに監視が設定されているかどうか、どのリージョンに設定されているかは、`DescribeTable` API の出力から判断できます。監視は DynamoDB によって所有および管理され、設定されているリージョンの AWS アカウントには表示されません。

MRSC グローバルテーブルは、米国東部 (バージニア北部)、米国東部 (オハイオ)、米国西部 (オレゴン)、カナダ (中部)、欧州 (アイルランド)、欧州 (ロンドン)、欧州 (パリ)、欧州 (フランクフルト)、欧州 (ストックホルム)、欧州 (スペイン)、アジアパシフィック (東京)、アジアパシフィック (ソウル)、アジアパシフィック (大阪)、アジアパシフィック (ムンバイ)、およびアジアパシフィック (シンガポール)の各 AWS リージョンで利用できます。MRSC グローバルテーブルには、異なる大陸のリージョンを含め、これらの中から任意の 3 つのリージョンを組み合わせたレプリカを含めることができます。MRSC グローバルテーブルの場合、書き込みおよび強力な整合性のある読み込みのレイテンシーは、選択するリージョンによって異なります。これらのオペレーションにはクロスリージョン通信が必要となるためです。参加リージョン間のラウンドトリップレイテンシーが大きくなると、リクエストレイテンシーが増加する可能性があります。通常時とフェイルオーバー時の両方でアプリケーションのレイテンシー要件を満たしているかどうかを確認するために、対象のリージョンでワークロードをテストすることをお勧めします。

MRSC グローバルテーブルを作成するには、データを含まない既存の DynamoDB テーブルに 1 つのレプリカと監視または 2 つのレプリカを追加します。既存の単一リージョンテーブルを MRSC グローバルテーブルに変換する場合は、テーブルが空であることを確認する必要があります。単一リージョンテーブルを既存の項目を持つ MRSC グローバルテーブルに変換することはサポートされていません。変換プロセス中にデータがテーブルに書き込まれないよう確認してください。既存の MRSC グローバルテーブルに追加のレプリカを追加することはできません。MRSC グローバルテーブルから単一のレプリカまたは監視を削除することはできません。MRSC グローバルテーブルから 2 つのレプリカを削除するか、1 つのレプリカと監視を削除して、残りのレプリカを単一リージョンの DynamoDB テーブルに変換できます。

別のリージョンで既に変更されている項目を変更しようとすると、書き込み操作は `ReplicatedWriteConflictException` で失敗します。`ReplicatedWriteConflictException` で失敗した書き込みは再試行でき、項目が別のリージョンで変更されなくなった場合に成功します。

MRSC グローバルテーブルには、次の考慮事項が適用されます。

- MRSC グローバルテーブルでは、有効期限 (TTL) はサポートされていません。
- ローカルセカンダリインデックス (LSI) は、MRSC グローバルテーブルではサポートされていません。
- CloudWatch Contributor Insights の情報は、オペレーションが発生したリージョンについてのみ報告されます。

整合性モードの選択

マルチリージョン整合性モードを選択するための重要な基準は、アプリケーションが低レイテンシーの書き込みと強力な整合性のある読み込みを優先するか、グローバルな強力な整合性を優先するかです。

MREC グローバルテーブルは、MRSC グローバルテーブルと比較して書き込みレイテンシーが低く、強力な整合性のある読み込みレイテンシーを持ちます。MREC グローバルテーブルには、レプリカ間のレプリケーション遅延と等しい目標復旧時点 (RPO) があり、レプリカのリージョンに応じて通常は数秒です。

次の場合は、MREC モードを使用する必要があります。

- アプリケーションは、データが別のリージョンで更新された場合に、強力な整合性のある読み込みオペレーションから返される古いデータを許容できます。

- マルチリージョンの読み取り整合性よりも、書き込みレイテンシーが低く、強力な整合性のある読み込みレイテンシーを優先します。
- マルチリージョンの高可用性戦略では、0 を超える RPO を許容できます。

MRSC グローバルテーブルは、MREC グローバルテーブルと比較して書き込みレイテンシーが高く、強力な整合性のある読み込みレイテンシーを持ちます。MRSC グローバルテーブルは、目標復旧時点 (RPO) ゼロをサポートします。

MRSC モードは、次の場合に使用します。

- 複数のリージョン間で強力な整合性のある読み込みが必要です。
- グローバル読み取り整合性は、書き込みレイテンシーの短縮よりも優先されます。
- マルチリージョンの高可用性戦略では、RPO を 0 にする必要があります。

グローバルテーブルのモニタリング

マルチリージョンの結果整合性 (MREC) 用に設定されたグローバルテーブルは `ReplicationLatency` メトリクスを CloudWatch に発行します。このメトリクスは、項目がレプリカテーブルに書き込まれてから、その項目がグローバルテーブルの別のレプリカに表示されるまでの経過時間を追跡します。`ReplicationLatency` はミリ秒単位で表し、グローバルテーブル内のすべての送信元と送信先のリージョンペアに対して出力されます。

一般的な `ReplicationLatency` 値は、選択した AWS リージョン間の距離と、ワークロードタイプやスループットなどの他の変数によって異なります。例えば、米国西部 (北カリフォルニア) (us-west-1) リージョンのソースレプリカは、アフリカ (ケープタウン) (af-south-1) リージョンと比較して、米国西部 (オレゴン) (us-west-2) リージョンへの `ReplicationLatency` が低くなります。

`ReplicationLatency` の値が上昇している場合、1 つのレプリカからの更新が他のレプリカテーブルにタイムリーに伝播されていないことを示している可能性があります。この場合、アプリケーションの読み込みおよび書き込みアクティビティを別の AWS リージョンに一時的にリダイレクトすることができます。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、`ReplicationLatency` メトリクスを発行しません。

フォールトインジェクションテスト

MREC と MRSC のグローバルテーブルはどちらも、制御されたフォールトインジェクション実験を実行してアプリケーションの耐障害性を向上させるためのフルマネージドサービスである AWS

Fault Injection Service (AWS FIS) と統合されています。AWS FIS を使用すると、次のことができます。

- 特定の障害シナリオを定義する実験テンプレートを作成します。
- アプリケーションの耐障害性を検証するため、リージョンの分離 (つまり、選択したレプリカとのレプリケーションを一時停止すること) をシミュレートして障害を注入し、エラー処理、復旧メカニズム、1 つの AWS リージョンで中断が発生した場合のマルチリージョントラフィックシフトの動作をテストします。

例えば、米国東部 (バージニア北部)、米国東部 (オハイオ)、米国西部 (オレゴン) にレプリカがあるグローバルテーブルでは、米国東部 (バージニア北部) と米国西部 (オレゴン) が通常のオペレーションを継続している間に、米国東部 (オハイオ) で実験を実行してリージョンの分離をテストできます。この制御されたテストは、本番稼働用ワークロードに影響を与える前に潜在的な問題を特定して解決するのに役立ちます。

AWS FIS でサポートされているアクションの完全なリストについては、「AWS FIS ユーザーガイド」の「アクションターゲット」を参照してください。また、リージョン間の DynamoDB レプリケーションを一時停止するには、「クロスリージョン接続」を参照してください。

AWS FIS で使用できる Amazon DynamoDB グローバルテーブルアクションの詳細については、「AWS FIS ユーザーガイド」の「DynamoDB グローバルテーブルアクションリファレンス」を参照してください。

フォールトインジェクション実験の実行を開始するには、「AWS FIS ユーザーガイド」の「AWS FIS 実験の計画」を参照してください。

> **Note**
>
> MRSC での AWS FIS 実験中は、結果整合性のある読み込みが許可されますが、MREC と同様に請求モードの変更やテーブルスループットの設定などのテーブル設定の更新は許可されません。エラーコードの詳細については、CloudWatch メトリクス `FaultInjectionServiceInducedErrors` を確認してください。

有効期限 (TTL)

MREC 用に設定されたグローバルテーブルは有効期限 (TTL) 削除の設定をサポートしています。TTL 設定は、グローバルテーブル内のすべてのレプリカに対して自動的に同期されます。TTL がリー

ジョン内のレプリカから項目を削除すると、削除はグローバルテーブル内の他のすべてのレプリカにレプリケートされます。TTL は書き込み容量を消費しないため、削除が発生したリージョンでは TTL 削除に対して課金されません。ただし、グローバルテーブルのレプリカが存在する他のリージョンでレプリケートされた削除に対しては課金されます。

TTL 削除レプリケーションは、削除がレプリケートされるレプリカの書き込み容量を消費します。プロビジョンドキャパシティ用に設定されたレプリカは、書き込みスループットと TTL 削除スループットの組み合わせがプロビジョンド書き込みキャパシティを超える場合、リクエストをスロットリングすることがあります。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、有効期限 (TTL) 削除の設定をサポートしていません。

Streams

マルチリージョンの結果整合性 (MREC) 用に設定されたグローバルテーブルは、レプリカテーブルの DynamoDB Stream からこれらの変更を読み取り、その変更を他のすべてのレプリカテーブルに適用することで、変更をレプリケートします。したがって、Streams は MREC グローバルテーブル内のすべてのレプリカでデフォルトで有効になっており、それらのレプリカで無効にすることはできません。MREC レプリケーションプロセスでは、短期間に複数の変更を 1 つのレプリケートされた書き込みに結合する場合があり、各レプリカの Stream にわずかに異なるレコードが含まれる場合があります。MREC レプリカの Streams レコードは、常に項目ごとに順序付けられますが、項目間の順序はレプリカによって異なる場合があります。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、レプリケーションに DynamoDB Streams を使用しないため、MRSC レプリカでは Streams はデフォルトで有効になっていません。MRSC レプリカで Streams を有効にできます。MRSC レプリカの Streams レコードは、Stream レコードの順序付けを含め、すべてのレプリカで同じです。

グローバルテーブル内の他のリージョンではなく、特定のリージョンで発生した変更について Streams レコードを処理するアプリケーションを作成する場合は、各項目にその項目の変更が発生したリージョンを定義する属性を追加できます。この属性を使用して、特定のリージョンの変更に対してのみ Lambda 関数を呼び出す Lambda イベントフィルターの使用など、他のリージョンで発生した変更について Streams レコードをフィルタリングできます。

取引

MREC 用に設定されたグローバルテーブルでは、DynamoDB トランザクションオペレーション (`TransactWriteItems` および `TransactGetItems`) は、操作が呼び出されたリージョン内でのみアトミックです。トランザクション書き込みはリージョン間でユニットとしてレプリケートされませ

ん。つまり、特定の時点で、トランザクション内の書き込みの一部のみが、他のレプリカでの読み取り操作によって返される可能性があります。

例えば、米国東部 (オハイオ) および米国西部 (オレゴン) リージョンにレプリカを持つグローバルテーブルがあり、米国東部 (オハイオ) リージョンで `TransactWriteItems` オペレーションを実行すると、変更がレプリケートされたときに米国西部 (オレゴン) では部分的に完了したトランザクションが観察されることがあります。変更は、ソースリージョンでコミットされた後にのみ、他のリージョンにレプリケートされます。

マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルはトランザクションオペレーションをサポートしておらず、これらのオペレーションが MRSC レプリカで呼び出されるとエラーが返されます。

読み取りおよび書き込みスループット

プロビジョニングモード

レプリケーションは書き込み容量を消費します。プロビジョンドキャパシティ用に設定されたレプリカは、アプリケーションの書き込みスループットとレプリケーション書き込みスループットの組み合わせがプロビジョンド書き込みキャパシティを超えると、リクエストをスロットリングすることがあります。プロビジョニングモードを使用するグローバルテーブルの場合、読み取り容量と書き込み容量の両方の自動スケーリング設定がレプリカ間で同期されます。

レプリカレベルで `ProvisionedThroughputOverride` パラメータを使用して、グローバルテーブル内の各レプリカの読み取り容量設定を個別に設定できます。デフォルトでは、プロビジョニングされた読み取り容量への変更は、グローバルテーブル内のすべてのレプリカに適用されます。グローバルテーブルに新しいレプリカを追加する場合、レプリカレベルのオーバーライドが明示的に指定されていない限り、ソーステーブルまたはレプリカの読み取り容量が初期値として使用されます。

オンデマンドモード

オンデマンドモードに設定されたグローバルテーブルの場合、書き込み容量はすべてのレプリカ間で自動的に同期されます。DynamoDB はトラフィックに基づいて容量を自動的に調整するため、レプリカ固有の読み取りまたは書き込み容量設定を管理する必要はありません。

設定の同期

DynamoDB グローバルテーブルの設定は、テーブルの動作とレプリケーションのさまざまな側面を制御する設定パラメータです。これらの設定は DynamoDB コントロールプレーン API を通じて管理され、グローバルテーブルを作成または変更するときに設定できます。グローバルテーブルは、すべ

てのレプリカで特定の設定を自動的に同期して一貫性を維持し、リージョン固有の最適化に柔軟性を持たせます。どの設定が同期され、どのように動作するかを理解することは、グローバルテーブルを効果的に設定するのに役立ちます。設定は、レプリカ間でどのように同期されるかに基づいて、3 つの主要なカテゴリに分類されます。

次の設定は常にグローバルテーブルのレプリカ間で同期されます。

- キャパシティモード (プロビジョンドキャパシティまたはオンデマンド)
- テーブルのプロビジョニングされた書き込み容量
- テーブル書き込みの自動スケーリング
- キースキーマの属性定義
- グローバルセカンダリインデックス (GSI) 定義
- GSI のプロビジョニングされた書き込みキャパシティ
- GSI の書き込み自動スケーリング
- サーバー側の暗号化 (SSE)
- MREC モードでの Streams 定義
- 有効期限 (TTL)
- ウォームスループット
- オンデマンドの最大書き込みスループット

以下の設定はレプリカ間で同期されますが、レプリカごとに上書きできます。

- テーブルのプロビジョニングされた読み込みキャパシティ
- テーブル読み取りの自動スケーリング
- GSI のプロビジョニングされた読み込み容量キャパシティ
- GSI の読み取り自動スケーリング
- テーブルクラス
- オンデマンドの最大読み取りスループット

> **Note**
>
> 上書き可能な設定値は、設定が他のレプリカで変更されると変更されます。例えば、米国東部 (バージニア北部) と米国西部 (オレゴン) にレプリカを持つ MREC グローバルテーブルが

あるとします。米国東部 (バージニア北部) レプリカのプロビジョニングされた読み込みスループットは 200 RCU に設定されています。米国西部 (オレゴン) のレプリカでは、プロビジョニングされた読み込みスループットオーバーライドが 100 RCU に設定されています。米国東部 (バージニア北部) レプリカのプロビジョニングされた読み込みスループット設定を 200 RCU から 300 RCU に更新すると、新しいプロビジョニングされた読み込みスループット値が米国西部 (オレゴン) のレプリカにも適用されます。これにより、米国西部 (オレゴン) レプリカのプロビジョニングされた読み込みスループット設定が、オーバーライドされた値である 100 RCU から新しい値である 300 RCU に変更されます。

以下の設定はレプリカ間で同期されません。

- 削除保護
- ポイントインタイムリカバリ
- タグ
- テーブルの CloudWatch Contributor Insights の有効化
- GSI の CloudWatch Contributor Insights の有効化
- Kinesis Data Streams 定義
- リソースポリシー
- MRSC モードでの Streams 定義

他のすべての設定はレプリカ間で同期されません。

DynamoDB Accelerator (DAX)

グローバルテーブルレプリカへの書き込みは DynamoDB Accelerator (DAX) をバイパスし、DynamoDB を直接更新します。その結果、書き込みが DAX キャッシュを更新しないため、DAX キャッシュが古くなる可能性があります。グローバルテーブルレプリカ用に設定された DAX キャッシュは、キャッシュ TTL の有効期限が切れた場合にのみ更新されます。

グローバルテーブルの管理に関する考慮事項

新しいグローバルテーブルレプリカの追加に使用したテーブルは、新しいレプリカが作成されてから 24 時間が経過するまで削除できません。

グローバルテーブルレプリカを含む AWS リージョンを無効にすると、それらのレプリカは、リージョンが無効になってから 20 時間後に単一リージョンテーブルに永続的に変換されます。

チュートリアル: グローバルテーブルの作成

このセクションでは、任意の整合性モード用に設定された DynamoDB グローバルテーブルを作成するための手順を順を追って説明します。アプリケーションの要件に基づいて、マルチリージョンの結果整合性 (MREC) モードまたはマルチリージョンの強力な整合性 (MRSC) モードのいずれかを選択します。

MREC グローバルテーブルは、書き込みレイテンシーを低くし、AWS リージョン全体で結果整合性を実現します。MRSC グローバルテーブルは、MREC よりも書き込みレイテンシーがわずかに高いリージョン間で強力な整合性のある読み込みを提供します。データ整合性、レイテンシー、可用性に関するアプリケーションのニーズに最適な整合性モードを選択します。

トピック

- MREC 用に設定されたグローバルテーブルの作成
- MRSC 用に設定されたグローバルテーブルの作成

MREC 用に設定されたグローバルテーブルの作成

このセクションでは、マルチリージョンの結果整合性 (MREC) モードでグローバルテーブルを作成する方法を示します。MREC はグローバルテーブルのデフォルトの整合性モードであり、AWS リージョン全体で非同期レプリケーションによる低レイテンシーの書き込みを提供します。1 つのリージョンの項目に加えられた変更は、通常、1 秒以内に他のすべてのリージョンにレプリケートされます。これにより、MREC は低い書き込みレイテンシーを優先し、異なるリージョンがわずかに異なるバージョンのデータを返す短い期間を許容できるアプリケーションに最適です。

DynamoDB が利用可能な任意の AWS リージョンでレプリカを含む MREC グローバルテーブルを作成し、レプリカをいつでも追加または削除できます。次の例は、複数のリージョンにレプリカを持つ MREC グローバルテーブルを作成する方法を示しています。

DynamoDB コンソールを使用した MREC グローバルテーブルの作成

AWS マネジメントコンソール を使用してグローバルテーブルを作成するには、次の手順に従います。以下の例では、レプリカテーブルを持つグローバルテーブルを米国およびヨーロッパに作成します。

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. この例では、ナビゲーションバーのリージョンセレクターから [米国東部 (オハイオ)] を選択します。

3. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
4. [Create Table] (テーブルの作成) を選択します。
5. [テーブルの作成] ページで、次の操作を行います。

   a. [テーブル名] に **`Music`** と入力します。

   b. [パーティションキー] に **`Artist`** と入力します。

   c. [ソートキー] に **`SongTitle`** と入力します。

   d. 他のデフォルトの設定はそのままにして、[テーブルの作成] を選択します。

      この新しいテーブルは、新しいグローバルテーブルの最初のレプリカテーブルとして機能します。これは、後で追加する他のレプリカテーブルのプロトタイプです。

6. テーブルがアクティブになった後、次の操作を行います。

   a. テーブルのリストから [ミュージック] テーブルを選択します。

   b. [グローバルテーブル] タブを選択します。

   c. レプリカの作成を選択します。

7. [利用可能なレプリケーションリージョン] ドロップダウンリストから、[米国西部 (オレゴン) us-west-2] を選択します。

   コンソールは、選択したリージョンに同一名のテーブルが存在しないことを確認します。同一名のテーブルが存在する場合は、そのリージョンで新しいレプリカテーブルを作成する前に既存のテーブルを削除する必要があります。

8. レプリカの作成を選択します。これで、米国西部 (オレゴン) us-west-2 リージョンでテーブル作成プロセスが開始されます。

   [Music] テーブル (およびその他すべてのレプリカテーブル) の [グローバルテーブル] タブに、テーブルが複数のリージョンでレプリケートされたことが示されます。

9. 前のステップを繰り返して別のリージョンを追加しますが、リージョンとして [欧州 (フランクフルト) eu-central-1] を選択します。
10. レプリケーションをテストするには、次の操作を行います。

    a. 米国東部 (オハイオ) リージョンでAWS マネジメントコンソールを使用していることを確認します。

    b. [テーブルアイテムの探索] を選択します。

    c. [項目を作成] を選択します。

d. [アーティスト] に「**item_1**」、[曲名] に「**Song Value 1**」と入力します。

e. [項目を作成] を選択します。

11. 他のリージョンに切り替えてレプリケーションを確認します。

a. 右上隅のリージョンセレクターから、[欧州 (フランクフルト)] を選択します。

b. [ミュージック] テーブルに、作成した項目が含まれていることを確認します。

c. [米国西部 (オレゴン)] で検証を繰り返します。

AWS CLI または Java を使用した MREC グローバルテーブルの作成

CLI

次のコード例は、最終的な一貫性を備えたマルチリージョンレプリケーション (MREC) を使用して DynamoDB グローバルテーブルを管理する方法を示しています。

- マルチリージョンレプリケーション (MREC)を備えたテーブルを作成します。
- レプリカテーブルの項目を配置および取得します。
- レプリカを一つずつ削除します。
- テーブルを削除してクリーンアップします。

AWS CLIBash スクリプトを使用した

マルチリージョンレプリケーションを備えたテーブルを作成します。

```
# Step 1: Create a new table (MusicTable) in US East (Ohio), with DynamoDB
 Streams enabled (NEW_AND_OLD_IMAGES)
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2

# Step 2: Create an identical MusicTable table in US East (N. Virginia)
```

```
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2

# Step 3: Create a table in Europe (Ireland)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2
```

マルチリージョンテーブルについて説明します。

```
# Step 4: View the list of replicas created using describe-table
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

レプリカテーブルに項目を配置します。

```
# Step 5: To verify that replication is working, add a new item to the Music
 table in US East (Ohio)
```

```
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

レプリカテーブルから項目を取得します。

```
# Step 6: Wait for a few seconds, and then check to see whether the item has
 been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1

aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

レプリカを削除します。

```
# Step 7: Delete the replica table in Europe (Ireland) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2

# Delete the replica table in US East (N. Virginia) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
```

```
    {
      "Delete": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2
```

テーブルを削除してクリーンアップします。

```
# Clean up: Delete the primary table
aws dynamodb delete-table --table-name MusicTable --region us-east-2

echo "Global table demonstration complete."
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateTable

Java

次のコード例は、複数のリージョンにまたがるレプリカを含む DynamoDB グローバルテーブルを作成して管理する方法を示しています。

- グローバルセカンダリインデックスおよび DynamoDB Streams を含むテーブルを作成します。
- 異なるリージョンにレプリカを追加して、グローバルテーブルを作成します。
- グローバルテーブルからレプリカを削除します。
- テスト項目を追加して、リージョン間のレプリケーションを検証します。

- グローバルテーブル設定とレプリカのステータスについて説明します。

SDK for Java 2.x

AWS SDK for Java 2.x を使用して、グローバルセカンダリインデックスおよび DynamoDB Streams を含むテーブルを作成します。

```
   public static CreateTableResponse createTableWithGSI(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
indexName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (indexName == null || indexName.trim().isEmpty()) {
           throw new IllegalArgumentException("Index name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating table: " + tableName + " with GSI: " +
indexName);

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
                       .attributeType(ScalarAttributeType.S)
                       .build(),
                   AttributeDefinition.builder()
                       .attributeName("SongTitle")
                       .attributeType(ScalarAttributeType.S)
                       .build())
               .keySchema(
                   KeySchemaElement.builder()
                       .attributeName("Artist")
                       .keyType(KeyType.HASH)
                       .build(),
```

```
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .globalSecondaryIndexes(GlobalSecondaryIndex.builder()
                    .indexName(indexName)
                    .keySchema(KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.HASH)
                        .build())
                    .projection(

Projection.builder().projectionType(ProjectionType.ALL).build())
                    .build())
                .streamSpecification(StreamSpecification.builder()
                    .streamEnabled(true)
                    .streamViewType(StreamViewType.NEW_AND_OLD_IMAGES)
                    .build())
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、テーブルがアクティブになるまで待ちます。

```
    public static void waitForTableActive(final DynamoDbClient dynamoDbClient,
final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
```

```
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Waiting for table to become active: " + tableName);

            try (DynamoDbWaiter waiter =
                DynamoDbWaiter.builder().client(dynamoDbClient).build()) {
                DescribeTableRequest request =
                    DescribeTableRequest.builder().tableName(tableName).build();

                waiter.waitUntilTableExists(request);
                LOGGER.info("Table is now active: " + tableName);
            }

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to wait for table to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、レプリカを追加して、グローバルテーブルを作成または拡張します。

```
    public static UpdateTableResponse addReplica(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final String indexName,
        final Long readCapacity) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
```

```
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (indexName == null || indexName.trim().isEmpty()) {
            throw new IllegalArgumentException("Index name cannot be null or
empty");
        }
        if (readCapacity == null || readCapacity <= 0) {
            throw new IllegalArgumentException("Read capacity must be a positive
number");
        }

        try {
            LOGGER.info("Adding replica in region: " + replicaRegion.id() + " for
table: " + tableName);

            // Create a ReplicationGroupUpdate for adding a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .create(builder -> builder.regionName(replicaRegion.id())
                    .globalSecondaryIndexes(ReplicaGlobalSecondaryIndex.builder()
                        .indexName(indexName)

.provisionedThroughputOverride(ProvisionedThroughputOverride.builder()
                            .readCapacityUnits(readCapacity)
                            .build())
                        .build())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica addition initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
```

```
            LOGGER.severe("Failed to add replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用してグローバルテーブルからレプリカを削除します。

```
    public static UpdateTableResponse removeReplica(
        final DynamoDbClient dynamoDbClient, final String tableName, final Region
replicaRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }

        try {
            LOGGER.info("Removing replica in region: " + replicaRegion.id() + "
for table: " + tableName);

            // Create a ReplicationGroupUpdate for removing a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .delete(builder ->
builder.regionName(replicaRegion.id()).build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
```

```
            LOGGER.info("Replica removal initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to remove replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用してテスト項目を追加して、レプリケーションを検証します。

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to table: " + tableName);

            Map<String,
software.amazon.awssdk.services.dynamodb.model.AttributeValue> item = new
HashMap<>();
            item.put(
                "Artist",
```

```
software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(artist)
                    .build());
            item.put(
                "SongTitle",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(songTitle)
                    .build());

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、グローバルテーブル設定とレプリカを記述します。

```
    public static DescribeTableResponse describeTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing table: " + tableName);
```

```
            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用したグローバルテーブル操作の完全な例。

```
    public static void exampleUsage(final Region sourceRegion, final Region
replicaRegion) {

        String tableName = "Music";
        String indexName = "SongTitleIndex";
        Long readCapacity = 15L;

        // Create DynamoDB client for the source region
        try (DynamoDbClient dynamoDbClient =
```

```
            DynamoDbClient.builder().region(sourceRegion).build()) {

            try {
                // Step 1: Create the initial table with GSI and streams
                LOGGER.info("Step 1: Creating table in source region: " +
sourceRegion.id());
                createTableWithGSI(dynamoDbClient, tableName, indexName);

                // Step 2: Wait for table to become active
                LOGGER.info("Step 2: Waiting for table to become active");
                waitForTableActive(dynamoDbClient, tableName);

                // Step 3: Add replica in destination region
                LOGGER.info("Step 3: Adding replica in region: " +
replicaRegion.id());
                addReplica(dynamoDbClient, tableName, replicaRegion, indexName,
readCapacity);

                // Step 4: Wait a moment for replica creation to start
                Thread.sleep(5000);

                // Step 5: Describe table to view replica information
                LOGGER.info("Step 5: Describing table to view replicas");
                describeTable(dynamoDbClient, tableName);

                // Step 6: Add a test item to verify replication
                LOGGER.info("Step 6: Adding test item to verify replication");
                putTestItem(dynamoDbClient, tableName, "TestArtist", "TestSong");

                LOGGER.info("Global table setup completed successfully!");
                LOGGER.info("You can verify replication by checking the item in
region: " + replicaRegion.id());

                // Step 7: Remove replica and clean up table
                LOGGER.info("Step 7: Removing replica from region: " +
replicaRegion.id());
                removeReplica(dynamoDbClient, tableName, replicaRegion);
                DeleteTableResponse deleteTableResponse =
dynamoDbClient.deleteTable(
                    DeleteTableRequest.builder().tableName(tableName).build());
                LOGGER.info("MREC global table demonstration completed
successfully!");

            } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
                throw new RuntimeException("Thread was interrupted", e);
            } catch (DynamoDbException e) {
                LOGGER.severe("DynamoDB operation failed: " + e.getMessage());
                throw e;
            }
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - CreateTable
  - DescribeTable
  - PutItem
  - UpdateTable

MRSC 用に設定されたグローバルテーブルの作成

このセクションでは、マルチリージョンの強力な整合性 (MRSC) グローバルテーブルを作成する方法について説明します。MRSC グローバルテーブルは、リージョン間で項目の変更を同期的にレプリケートし、レプリカに対する強力な整合性のある読み込みオペレーションが常に項目の最新バージョンを返すようにします。単一リージョンテーブルを MRSC グローバルテーブルに変換する場合は、テーブルが空であることを確認する必要があります。単一リージョンテーブルを既存の項目を持つ MRSC グローバルテーブルに変換することはサポートされていません。変換プロセス中にデータがテーブルに書き込まれないよう確認してください。

MRSC グローバルテーブルは、3 つのレプリカ、または 2 つのレプリカと 1 つの監視で設定できます。MRSC グローバルテーブルを作成するときに、レプリカとオプションの監視がデプロイされるリージョンを選択します。次の例では、米国東部 (バージニア北部) および米国東部 (オハイオ) リージョンにレプリカを持つ MRSC グローバルテーブルを作成し、米国西部 (オレゴン) リージョンに監視を配置します。

> **Note**
>
> グローバルテーブルを作成する前に、Service Quota のスループット制限がすべてのターゲットリージョンで一貫していることを確認してください。これは、グローバルテーブルの

作成に必要です。グローバルテーブルのスループット制限の詳細については、「グローバルテーブルのクォータ」を参照してください。

DynamoDB コンソールを使用した MRSC グローバルテーブルの作成

AWS マネジメントコンソール を使用してグローバルテーブルを作成するには、次の手順に従います。

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションバーのリージョンセレクターで、MRSC を使用したグローバルテーブルがサポートされているリージョン (**us-east-2** など) を選択します。
3. ナビゲーションペインで、[Table] (テーブル) を選択します。
4. [テーブルの作成] を選択します。
5. [テーブルの作成] ページで、次の操作を行います。
   a. [テーブル名] に「**Music**」と入力します。
   b. [パーティションキー] に「**Artist**」と入力し、デフォルトの [文字列] タイプのままにします。
   c. [ソートキー] に「**SongTitle**」と入力し、デフォルトの [文字列] タイプのままにします。
   d. 他のデフォルトの設定はそのままにして、[テーブルの作成] を選択します

      この新しいテーブルは、新しいグローバルテーブルの最初のレプリカテーブルとして機能します。これは、後で追加する他のレプリカテーブルのプロトタイプです。
6. テーブルがアクティブになるまで待ってから、テーブルリストから選択します。
7. [グローバルテーブル] タブを選択して、[レプリカを作成] を選択します。
8. [レプリカの作成] ページで、次の操作を行います。
   a. [マルチリージョン整合性] で、[強力な整合性] を選択します。
   b. [レプリケーションリージョン 1] で、「**US East (N. Virginia) us-east-1**」を選択します。
   c. [レプリケーションリージョン 2] で、「**US West (Oregon) us-west-2**」を選択します。
   d. 米国西部 (オレゴン) リージョンの [監視として設定する] を確認します。

e. [レプリカの作成] を選択します。

9. レプリカと監視の作成プロセスが完了するまで待ちます。テーブルが使用可能になると、レプリカのステータスは [アクティブ] と表示されます。

AWS CLI または Java を使用した MRSC グローバルテーブルの作成

開始する前に、監視リージョンを持つ MRSC グローバルテーブルを作成するために必要なアクセス許可が IAM プリンシパルにあることを確認してください。

次のサンプル IAM ポリシーでは、米国東部 (オハイオ) に DynamoDB テーブル (`MusicTable`) を作成し、米国東部 (バージニア北部) にレプリカ、米国西部 (オレゴン) に監視リージョンを作成できます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:CreateTableReplica",
                "dynamodb:CreateGlobalTableWitness",
                "dynamodb:DescribeTable",
                "dynamodb:UpdateTable",
                "dynamodb:DeleteTable",
                "dynamodb:DeleteTableReplica",
                "dynamodb:DeleteGlobalTableWitness",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MusicTable",
                "arn:aws:dynamodb:us-east-2:123456789012:table/MusicTable",
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicTable"
```

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "replication.dynamodb.amazonaws.com"
                }
            }
        }
    ]
}
```

次のコード例は、マルチリージョンの強力な整合性 (MRSC) を備えた DynamoDB グローバルテーブルを作成および管理する方法を示しています。

- マルチリージョンの強力な整合性を備えたテーブルを作成します。
- MRSC 設定とレプリカのステータスを確認します。
- 即時読み取りでリージョン間の強力な整合性をテストします。
- MRSC 保証を使用して条件付き書き込みを実行します。
- MRSC グローバルテーブルリソースをクリーンアップします。

Bash

AWS CLIBash スクリプトを使用した

マルチリージョンの強力な整合性を備えたテーブルを作成します。

```
# Step 1: Create a new table in us-east-2 (primary region for MRSC)
# Note: Table must be empty when enabling MRSC
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
```

```
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --region us-east-2

# Wait for table to become active
aws dynamodb wait table-exists --table-name MusicTable --region us-east-2

# Step 2: Add replica and witness with Multi-Region Strong Consistency
# MRSC requires exactly three replicas in supported regions
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Create": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Create": {"RegionName": "us-west-2"}}]' \
    --multi-region-consistency STRONG \
    --region us-east-2
```

MRSC 設定とレプリカのステータスを確認します。

```
# Verify the global table configuration and MRSC setting
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

リージョン間の即時読み取りの強力な整合性をテストします。

```
# Write an item to the primary region
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"},"Album":
 {"S":"The Beatles 1967-1970"},"Year": {"N":"1968"}}' \
    --region us-east-2

# Read the item from replica region to verify strong consistency (cannot read or
 write to witness)
# No wait time needed - MRSC provides immediate consistency
echo "Reading from us-east-1 (immediate consistency):"
aws dynamodb get-item \
    --table-name MusicTable \
```

```
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --consistent-read \
    --region us-east-1
```

MRSC 保証を使用して条件付き書き込みを実行します。

```
# Perform a conditional update from a different region
# This demonstrates that conditions work consistently across all regions
aws dynamodb update-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --update-expression "SET #rating = :rating" \
    --condition-expression "attribute_exists(Artist)" \
    --expression-attribute-names '{"#rating": "Rating"}' \
    --expression-attribute-values '{":rating": {"N":"5"}}' \
    --region us-east-1
```

MRSC グローバルテーブルリソースをクリーンアップします。

```
# Remove replica tables (must be done before deleting the primary table)
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Delete": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Delete": {"RegionName": "us-west-2"}}]' \
    --region us-east-2

# Wait for replicas to be deleted
echo "Waiting for replicas to be deleted..."
sleep 30

# Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteTable

- DescribeTable
- GetItem
- PutItem
- UpdateItem
- UpdateTable

Java

SDK for Java 2.x

MRSC 変換の準備ができたリージョンテーブルを AWS SDK for Java 2.x を使用して作成します。

```
   public static CreateTableResponse createRegionalTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating regional table: " + tableName + " (must be
empty for MRSC)");

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
                       .attributeType(ScalarAttributeType.S)
                       .build(),
                   AttributeDefinition.builder()
                       .attributeName("SongTitle")
                       .attributeType(ScalarAttributeType.S)
                       .build())
               .keySchema(
                   KeySchemaElement.builder()
```

```
                        .attributeName("Artist")
                        .keyType(KeyType.HASH)
                        .build(),
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Regional table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create regional table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to create regional table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、リージョンテーブルをレプリカと監視を備えた MRSC に変換します。

```
    public static UpdateTableResponse convertToMRSCWithWitness(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
```

```
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Converting table to MRSC with replica in " +
replicaRegion.id() + " and witness in "
                + witnessRegion.id());

            // Create replica update using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .create(CreateReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Create witness update
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .create(CreateGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .multiRegionConsistency(MultiRegionConsistency.STRONG)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC conversion initiated. Status: "
                + response.tableDescription().tableStatus());
            LOGGER.info("UpdateTableResponse full object: " + response);
            return response;

        } catch (DynamoDbException e) {
```

```
            LOGGER.severe("Failed to convert table to MRSC: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to convert table to MRSC: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC のグローバルテーブル設定を記述します。

```
    public static DescribeTableResponse describeMRSCTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing MRSC global table: " + tableName);

            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            LOGGER.info("Multi-region consistency: " +
response.table().multiRegionConsistency());

            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
```

```
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            if (response.table().globalTableWitnesses() != null
                && !response.table().globalTableWitnesses().isEmpty()) {
                LOGGER.info("Number of witnesses: "
                    + response.table().globalTableWitnesses().size());
                response.table()
                    .globalTableWitnesses()
                    .forEach(witness -> LOGGER.info(
                        "Witness region: " + witness.regionName() + ", Status: "
+ witness.witnessStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Table not found: " + tableName)
                .cause(e)
                .build();
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to describe table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC の強力な整合性を検証するためのテスト項目を追加します。

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
```

```
        final String album,
        final String year) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to MRSC global table: " + tableName);

            Map<String, AttributeValue> item = new HashMap<>();
            item.put("Artist", AttributeValue.builder().s(artist).build());
            item.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            if (album != null && !album.trim().isEmpty()) {
                item.put("Album", AttributeValue.builder().s(album).build());
            }
            if (year != null && !year.trim().isEmpty()) {
                item.put("Year", AttributeValue.builder().n(year).build());
            }

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully with strong consistency");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw DynamoDbException.builder()
```

```
                .message("Failed to add test item to table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカから一貫した読み取りを持つ項目を読み取ります。

```
    public static GetItemResponse getItemWithConsistentRead(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Reading item from MRSC global table with consistent
read: " + tableName);

            Map<String, AttributeValue> key = new HashMap<>();
            key.put("Artist", AttributeValue.builder().s(artist).build());
            key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            GetItemRequest getItemRequest = GetItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .consistentRead(true)
                .build();
```

```
            GetItemResponse response = dynamoDbClient.getItem(getItemRequest);

            if (response.hasItem()) {
                LOGGER.info("Item found with strong consistency - no wait time
needed");
            } else {
                LOGGER.info("Item not found");
            }

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to read item from table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to read item from table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC 保証を使用する条件付きの更新を実行します。

```
    public static UpdateItemResponse performConditionalUpdate(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String rating) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
```

```
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }
        if (rating == null || rating.trim().isEmpty()) {
            throw new IllegalArgumentException("Rating cannot be null or empty");
        }

        try {
            LOGGER.info("Performing conditional update on MRSC global table: " +
tableName);

            Map<String, AttributeValue> key = new HashMap<>();
            key.put("Artist", AttributeValue.builder().s(artist).build());
            key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.put("#rating", "Rating");

            Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
            expressionAttributeValues.put(
                ":rating", AttributeValue.builder().n(rating).build());

            UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #rating = :rating")
                .conditionExpression("attribute_exists(Artist)")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

            UpdateItemResponse response =
dynamoDbClient.updateItem(updateItemRequest);
            LOGGER.info("Conditional update successful - demonstrates strong
consistency");

            return response;

        } catch (ConditionalCheckFailedException e) {
            LOGGER.warning("Conditional check failed: " + e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
```

```
            LOGGER.severe("Failed to perform conditional update: " + tableName +
" - " + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to perform conditional update: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカと監視がアクティブになるまで待ちます。

```
    public static void waitForMRSCReplicasActive(
        final DynamoDbClient dynamoDbClient, final String tableName, final int
maxWaitTimeSeconds)
        throws InterruptedException {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (maxWaitTimeSeconds <= 0) {
            throw new IllegalArgumentException("Max wait time must be positive");
        }

        try {
            LOGGER.info("Waiting for MRSC replicas and witnesses to become
active: " + tableName);

            final long startTime = System.currentTimeMillis();
            final long maxWaitTimeMillis = maxWaitTimeSeconds * 1000L;
            int backoffSeconds = 5; // Start with 5 second intervals
            final int maxBackoffSeconds = 30; // Cap at 30 seconds

            while (System.currentTimeMillis() - startTime < maxWaitTimeMillis) {
                DescribeTableResponse response =
describeMRSCTable(dynamoDbClient, tableName);

                boolean allActive = true;
```

```
                StringBuilder statusReport = new StringBuilder();

                if (response.table().multiRegionConsistency() == null
                    || !MultiRegionConsistency.STRONG
                        .toString()

.equals(response.table().multiRegionConsistency().toString())) {
                    allActive = false;
                    statusReport
                        .append("MultiRegionConsistency: ")
                        .append(response.table().multiRegionConsistency())
                        .append(" ");
                }
                if (response.table().replicas() == null
                    || response.table().replicas().isEmpty()) {
                    allActive = false;
                    statusReport.append("No replicas found. ");
                }
                if (response.table().globalTableWitnesses() == null
                    || response.table().globalTableWitnesses().isEmpty()) {
                    allActive = false;
                    statusReport.append("No witnesses found. ");
                }

                // Check table status
                if (!"ACTIVE".equals(response.table().tableStatus().toString()))
{
                    allActive = false;
                    statusReport
                        .append("Table: ")
                        .append(response.table().tableStatus())
                        .append(" ");
                }

                // Check replica status
                if (response.table().replicas() != null) {
                    for (var replica : response.table().replicas()) {
                        if (!"ACTIVE".equals(replica.replicaStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Replica(")
                                .append(replica.regionName())
                                .append("): ")
```

```
                                .append(replica.replicaStatus())
                                .append(" ");
                        }
                    }
                }

                // Check witness status
                if (response.table().globalTableWitnesses() != null) {
                    for (var witness : response.table().globalTableWitnesses()) {
                        if (!"ACTIVE".equals(witness.witnessStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Witness(")
                                .append(witness.regionName())
                                .append("): ")
                                .append(witness.witnessStatus())
                                .append(" ");
                        }
                    }
                }

                if (allActive) {
                    LOGGER.info("All MRSC replicas and witnesses are now active:
" + tableName);
                    return;
                }

                LOGGER.info("Waiting for MRSC components to become active.
Status: " + statusReport.toString());
                LOGGER.info("Next check in " + backoffSeconds + " seconds...");

                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, maxBackoffSeconds);
            }

            throw DynamoDbException.builder()
                .message("Timeout waiting for MRSC replicas to become active
after " + maxWaitTimeSeconds + " seconds")
                .build();

        } catch (DynamoDbException | InterruptedException e) {
```

```
            LOGGER.severe("Failed to wait for MRSC replicas to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカと監視をクリーンアップします。

```
    public static UpdateTableResponse cleanupMRSCReplicas(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Cleaning up MRSC replicas and witnesses for table: " +
tableName);

            // Remove replica using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .delete(DeleteReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Remove witness
```

```
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .delete(DeleteGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC cleanup initiated - removing replica and witness.
Response: " + response);

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to cleanup MRSC replicas: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to cleanup MRSC replicas: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC ワークフローのデモンストレーションを完了します。

```
    public static void demonstrateCompleteMRSCWorkflow(
        final DynamoDbClient primaryClient,
        final DynamoDbClient replicaClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion)
        throws InterruptedException {

        if (primaryClient == null) {
```

```
            throw new IllegalArgumentException("Primary DynamoDB client cannot be
null");
        }
        if (replicaClient == null) {
            throw new IllegalArgumentException("Replica DynamoDB client cannot be
null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("=== Starting Complete MRSC Workflow Demonstration ===");

            // Step 1: Create an empty single-Region table
            LOGGER.info("Step 1: Creating empty single-Region table");
            createRegionalTable(primaryClient, tableName);

            // Use the existing GlobalTableOperations method for basic table
waiting
            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 2: Convert to MRSC with replica and witness
            LOGGER.info("Step 2: Converting to MRSC with replica and witness");
            convertToMRSCWithWitness(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for MRSC conversion to complete using MRSC-specific waiter
            LOGGER.info("Waiting for MRSC conversion to complete...");
            waitForMRSCReplicasActive(primaryClient, tableName);

            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);
```

```
            // Step 3: Verify MRSC configuration
            LOGGER.info("Step 3: Verifying MRSC configuration");
            describeMRSCTable(primaryClient, tableName);

            // Step 4: Test strong consistency with data operations
            LOGGER.info("Step 4: Testing strong consistency with data
operations");

            // Add test item to primary region
            putTestItem(primaryClient, tableName, "The Beatles", "Hey Jude", "The
Beatles 1967-1970", "1968");

            // Immediately read from replica region (no wait needed with MRSC)
            LOGGER.info("Reading from replica region immediately (strong
consistency):");
            GetItemResponse getResponse =
                getItemWithConsistentRead(replicaClient, tableName, "The
Beatles", "Hey Jude");

            if (getResponse.hasItem()) {
                LOGGER.info("# Strong consistency verified - item immediately
available in replica region");
            } else {
                LOGGER.warning("# Item not found in replica region");
            }

            // Test conditional update from replica region
            LOGGER.info("Testing conditional update from replica region:");
            performConditionalUpdate(replicaClient, tableName, "The Beatles",
"Hey Jude", "5");
            LOGGER.info("# Conditional update successful - demonstrates strong
consistency");

            // Step 5: Cleanup
            LOGGER.info("Step 5: Cleaning up resources");
            cleanupMRSCReplicas(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for cleanup to complete using basic table waiter
            LOGGER.info("Waiting for replica cleanup to complete...");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // "Halt" until replica/witness cleanup is complete
```

```
            DescribeTableResponse cleanupVerification =
describeMRSCTable(primaryClient, tableName);
            int backoffSeconds = 5; // Start with 5 second intervals
            while (cleanupVerification.table().multiRegionConsistency() != null)
{
                LOGGER.info("Waiting additional time (" + backoffSeconds + "
seconds) for MRSC cleanup to complete...");
                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, 30);
                cleanupVerification = describeMRSCTable(primaryClient,
tableName);
            }

            // Delete the primary table
            deleteTable(primaryClient, tableName);

            LOGGER.info("=== MRSC Workflow Demonstration Complete ===");
            LOGGER.info("");
            LOGGER.info("Key benefits of Multi-Region Strong Consistency
(MRSC):");
            LOGGER.info("- Immediate consistency across all regions (no eventual
consistency delays)");
            LOGGER.info("- Simplified application logic (no need to handle
eventual consistency)");
            LOGGER.info("- Support for conditional writes and transactions across
regions");
            LOGGER.info("- Consistent read operations from any region without
waiting");

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("MRSC workflow failed: " + e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - CreateTable
  - DeleteTable

- DescribeTable
- GetItem
- PutItem
- UpdateItem
- UpdateTable

DynamoDB グローバルテーブルのセキュリティ

グローバルテーブルレプリカは DynamoDB テーブルであるため、AWS Identity and Access Management (IAM) ID ポリシーやリソースベースのポリシーなど、単一リージョンテーブルに対して行うレプリカへのアクセスを制御するのと同じ方法を使用します。

このトピックでは、IAM アクセス許可と AWS Key Management Service (AWS KMS) 暗号化を使用して DynamoDB グローバルテーブルを保護する方法について説明します。クロスリージョンレプリケーションと自動スケーリングを許可するサービスリンクロール (SLR)、グローバルテーブル の作成、更新、削除に必要な IAM アクセス許可、マルチリージョンの最終的な一貫性 (MREC) テーブルとマルチリージョンの強力な整合性 (MRSC) テーブルの違いについて説明します。また、クロスリージョンレプリケーションを安全に管理するための AWS KMS 暗号化キーについても説明します。

グローバルテーブルのサービスリンクロール

DynamoDB グローバルテーブルは、クロスリージョンレプリケーションと自動スケーリング機能を管理するために、サービスリンクロール (SLR) に依存しています。

これらのロールは、AWS アカウントごとに 1 回だけ設定する必要があります。作成後、同じロールがアカウント内のすべてのグローバルテーブルに機能します。サービスリンクロールの詳細については、「IAM ユーザーガイド」の「サービスリンクロールの使用」を参照してください。

レプリケーションサービスにリンクされたロール

最初のグローバルテーブルを作成するとき、Amazon DynamoDB によって、`AWSServiceRoleForDynamoDBReplication` サービスリンクロール (SLR) が自動的に作成されます。このロールは、クロスリージョンレプリケーションを管理します。

リソースベースのポリシーをレプリカに適用する際は、`AWSServiceRoleForDynamoDBReplicationPolicy` で定義されている SLR プリンシパルへのアクセス許可を拒否しないでください。拒否すると、レプリケーションが中断されます。必要な

SLR アクセス許可を拒否すると、影響を受けるレプリカとの間のレプリケーションは停止し、レプリカテーブルのステータスは「REPLICATION_NOT_AUTHORIZED」に変わります。

- マルチリージョンの結果整合性 (MREC) のグローバルテーブルについて、レプリカが 20 時間以上「REPLICATION_NOT_AUTHORIZED」ステータスのままである場合、レプリカは 1 つのリージョンの DynamoDB テーブルに変換され、元に戻せなくなります。
- マルチリージョンの強力な整合性 (MRSC) のグローバルテーブルの場合、必要なアクセス許可を拒否すると、書き込みおよび強力な整合性のある読み込みオペレーションが AccessDeniedException になります。レプリカが 7 日以上「REPLICATION_NOT_AUTHORIZED」ステータスのままである場合、レプリカは永続的にアクセスできなくなり、書き込みオペレーションと強力な整合性のある読み込みオペレーションはエラーで失敗し続けます。レプリカの削除などの一部の管理オペレーションは成功します。

自動スケーリングサービスにリンクされたロール

プロビジョンドキャパシティモードのグローバルテーブルを設定する場合は、グローバルテーブルに自動スケーリングを設定する必要があります。DynamoDB 自動スケーリングは、AWS Application Auto Scaling Service を使用して、グローバルテーブルレプリカのプロビジョニングされたスループットキャパシティを動的に調整します。Application Auto Scaling サービスは、AWSServiceRoleForApplicationAutoScaling_DynamoDBTable という名前のサービスにリンクされたロール (SLR) を作成します。このサービスにリンクされたロールは、DynamoDB テーブルの自動スケーリングを初めて設定するときに、AWS アカウントで自動的に作成されます。これにより、Application Auto Scaling はプロビジョニングされたテーブル容量を管理でき、CloudWatch アラームを作成できます。

リソースベースのポリシーをレプリカに適用する際は、AWSApplicationAutoscalingDynamoDBTablePolicy で定義されている Application Auto Scaling SLR プリンシパルへのアクセス許可を拒否しないでください。拒否すると、自動スケーリング機能が中断されます。

サービスにリンクされたロールの IAM ポリシーの例

以下の条件の IAM ポリシーは、DynamoDB レプリケーション SLR および AWS 自動スケーリング SLR に必要なアクセス許可には影響しません。この条件は、レプリケーションや自動スケーリングが意図せずに中断されないように、制限の広いポリシーに追加できます。

拒否ポリシーから必要な SLR アクセス許可を除外する

次の例は、サービスにリンクされたロールプリンシパルを拒否ステートメントから除外する方法を示しています。

```
"Condition": {
    "StringNotEquals": {
        "aws:PrincipalArn": [
            "arn:aws::iam::111122223333:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication",
            "arn:aws::iam::111122223333:role/aws-service-role/dynamodb.application-
autoscaling.amazonaws.com/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable"
        ]
    }
}
```

グローバルテーブルで AWS IAM を使用する方法

次のセクションでは、さまざまなグローバルテーブルのオペレーションに必要なアクセス許可について説明し、ユーザーとアプリケーションに適切なアクセスを設定するのに役立つポリシーの例を示します。

> **Note**
>
> 説明されているすべてのアクセス許可は、影響を受けるリージョンの特定のテーブルリソース ARN に適用する必要があります。テーブルリソース ARN は `arn:aws:dynamodb:region:account-id:table/table-name` の形式に従います。ここでは、実際のリージョン、アカウント ID、テーブル名の値を指定する必要があります。

トピック

- グローバルテーブルの作成とレプリカの追加
- グローバルテーブルの更新
- グローバルテーブルとレプリカの削除

グローバルテーブルの作成とレプリカの追加

DynamoDB グローバルテーブルは、マルチリージョンの結果整合性 (MREC) およびマルチリージョンの強力な整合性 (MRSC) の 2 つの整合性モードをサポートしています。MREC グローバルテーブ

ルは、任意の数のリージョンにまたがって複数のレプリカを持つことができ、結果整合性を提供します。MRSC グローバルテーブルは、3 つのリージョン (3 つのレプリカまたは 2 つのレプリカと 1 つの監視) のみを必要とし、目標復旧時点 (RPO) との強力な整合性を提供します。

グローバルテーブルの作成に必要なアクセス許可は、監視の有無にかかわらず、グローバルテーブルを作成するかどうかによって異なります。

グローバルテーブルを作成するためのアクセス許可

次のアクセス許可は、最初のグローバルテーブルの作成と、後でレプリカを追加するために必要です。これらのアクセス許可は、マルチリージョンの結果整合性 (MREC) とマルチリージョンの強力な整合性 (MRSC) の両方のグローバルテーブルに適用されます。

- グローバルテーブルには、DynamoDB が AWSServiceRoleForDynamoDBReplication サービスリンクロール (SLR) を通じて管理するクロスリージョンレプリケーションが必要です。次のアクセス許可により、DynamoDB はグローバルテーブルを初めて作成するときに、このロールを自動的に作成できます。
  - iam:CreateServiceLinkedRole
- UpdateTable API を使用してグローバルテーブルを作成するか、レプリカを追加するには、ソーステーブルリソースに対して次のアクセス許可が必要です。
  - dynamodb:UpdateTable
- レプリカを追加するには、そのリージョンのテーブルリソースに対して次のアクセス許可が必要です。
  - dynamodb:CreateTable
  - dynamodb:CreateTableReplica
  - dynamodb:Query
  - dynamodb:Scan
  - dynamodb:UpdateItem
  - dynamodb:PutItem
  - dynamodb:GetItem
  - dynamodb:DeleteItem
  - dynamodb:BatchWriteItem

監視を使用した MRSC グローバルテーブルの追加アクセス許可

監視リージョンを使用してマルチリージョンの強力な整合性 (MRSC) グローバルテーブルを作成する場合、すべての参加リージョン (レプリカリージョンと監視リージョンの両方を含む) のテーブルリソースに対して次のアクセス許可が必要です。

- `dynamodb:CreateGlobalTableWitness`

グローバルテーブルの作成のための IAM ポリシーの例

3 つのリージョンにわたる MREC または MRSC グローバルテーブルの作成

次のアイデンティティベースのポリシーでは、必要な DynamoDB レプリケーションのサービスリンクロールの作成など、3 つのリージョンにまたがって「users」という名前の MREC または MRSC グローバルテーブルを作成できます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingUsersGlobalTable",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users",
        "arn:aws:dynamodb:us-west-2:123456789012:table/users"
      ]
    },
```

```
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

MREC または MRSC グローバルテーブルの作成を特定のリージョンに制限する

次のアイデンティティベースのポリシーでは、必要な DynamoDB レプリケーションサービスリンクロールの作成など、aws:RequestedRegion 条件キーを使用して、特定のリージョン間で DynamoDB グローバルテーブルのレプリカを作成できます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAddingReplicasToSourceTable",
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-1"
          ]
        }
      }
    },
    {
```

```
      "Sid": "AllowCreatingReplicas",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-2",
            "us-west-2"
          ]
        }
      }
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

監視付き MRSC グローバルテーブルの作成

次のアイデンティティベースのポリシーでは、us-east-1 と us-east-2 のレプリカと us-west-2 の監視のある「users」という名前の DynamoDB MRSC グローバルテーブルを作成できます。これには、必要な DynamoDB レプリケーションサービスリンクロールの作成が含まれます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingUsersGlobalTableWithWitness",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:CreateGlobalTableWitness",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users"
      ]
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
```

```
  ]
}
```

MRSC 監視の作成を特定のリージョンに制限する

このアイデンティティベースのポリシーでは、必要な DynamoDB レプリケーションサービスリンクロールの作成など、aws:RequestedRegion 条件キーとすべてのリージョン間で制限されていない監視の作成を使用して、特定のリージョン間で MRSC グローバルテーブルとレプリカを作成できます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCreatingReplicas",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:CreateTableReplica",
        "dynamodb:UpdateTable",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "us-east-1",
            "us-east-2"
          ]
        }
      }
    },
    {
```

```
      "Sid": "AllowCreatingWitness",
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateGlobalTableWitness"
      ],
      "Resource": "*"
    },
    {
      "Sid": "AllowCreatingSLR",
      "Effect": "Allow",
      "Action": [
        "iam:CreateServiceLinkedRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/aws-service-role/
replication.dynamodb.amazonaws.com/AWSServiceRoleForDynamoDBReplication"
      ]
    }
  ]
}
```

グローバルテーブルの更新

UpdateTable API を使用して既存のグローバルテーブルのレプリカ設定を変更するには、API コールを行うリージョンのテーブルリソースに対して次のアクセス許可が必要です。

- dynamodb:UpdateTable

自動スケーリングポリシーや有効期限設定など、他のグローバルテーブル設定も更新できます。これらの追加の更新オペレーションには、次のアクセス許可が必要です。

- UpdateTableReplicaAutoScaling API を使用してレプリカの自動スケーリングポリシーを更新するには、レプリカを含むすべてのリージョンのテーブルリソースに対して、次のアクセス許可が必要です。
  - application-autoscaling:DeleteScalingPolicy
  - application-autoscaling:DeleteScheduledAction
  - application-autoscaling:DeregisterScalableTarget
  - application-autoscaling:DescribeScalableTargets
  - application-autoscaling:DescribeScalingActivities

  - application-autoscaling:DescribeScalingPolicies
  - application-autoscaling:DescribeScheduledActions
  - application-autoscaling:PutScalingPolicy
  - application-autoscaling:PutScheduledAction
  - application-autoscaling:RegisterScalableTarget
- UpdateTimeToLive API を使用して有効期限設定を更新するには、レプリカを含むすべてのリージョンのテーブルリソースに対して次のアクセス許可が必要です。
  - dynamodb:UpdateTimeToLive

  有効期限 (TTL) は、マルチリージョンの結果整合性 (MREC) で設定されたグローバルテーブルでのみサポートされることに注意してください。グローバルテーブルの仕組みに関する詳細については、「DynamoDB グローバルテーブルの仕組み」を参照してください。

グローバルテーブルとレプリカの削除

グローバルテーブルを削除するには、すべてのレプリカを削除する必要があります。このオペレーションに必要なアクセス許可は、監視リージョンの有無にかかわらず、グローバルテーブルを削除するかどうかによって異なります。

グローバルテーブルとレプリカを削除するアクセス許可

個々のレプリカを削除したり、グローバルテーブルを完全に削除したりするには、次のアクセス許可が必要です。グローバルテーブル設定を削除すると、異なるリージョンのテーブル間のレプリケーション関係のみが削除されます。最後に残っているリージョンの基盤となる DynamoDB テーブルは削除されません。最後のリージョンのテーブルは、同じデータと設定を持つ標準の DynamoDB テーブルとして引き続き存在し続けます。これらのアクセス許可は、マルチリージョンの結果整合性 (MREC) とマルチリージョンの強力な整合性 (MRSC) の両方のグローバルテーブルに適用されます。

- UpdateTable API を使用して既存のグローバルテーブルからレプリカを削除するには、API コールを行う元のリージョンのテーブルリソースに対して次のアクセス許可が必要です。
  - dynamodb:UpdateTable
- レプリカを削除する各リージョンのテーブルリソースには、次のアクセス許可が必要です。
  - dynamodb:DeleteTable
  - dynamodb:DeleteTableReplica

監視を使用した MRSC グローバルテーブルの追加アクセス許可

監視を使用するマルチリージョンの強力な整合性 (MRSC) グローバルテーブルを削除する場合、すべての参加リージョン (レプリカリージョンと監視リージョンの両方を含む) のテーブルリソースに対して次のアクセス許可が必要です。

- `dynamodb:DeleteGlobalTableWitness`

グローバルテーブルレプリカを削除する IAM ポリシーの例

グローバルテーブルレプリカの削除

このアイデンティティベースのポリシーでは、「users」という名前の DynamoDB グローバルテーブルとそのレプリカを 3 つのリージョン間で削除できます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable",
        "dynamodb:DeleteTable",
        "dynamodb:DeleteTableReplica"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users",
        "arn:aws:dynamodb:us-west-2:123456789012:table/users"
      ]
    }
  ]
}
```

監視を使用する MRSC グローバルテーブルの削除

このアイデンティティベースのポリシーでは、「users」という名前の MRSC グローバルテーブルのレプリカと監視を削除できます。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateTable",
        "dynamodb:DeleteTable",
        "dynamodb:DeleteTableReplica",
        "dynamodb:DeleteGlobalTableWitness"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/users",
        "arn:aws:dynamodb:us-east-2:123456789012:table/users"
      ]
    }
  ]
}
```

グローバルテーブルで AWS KMS を使用する方法

すべての DynamoDB テーブルと同様に、グローバルテーブルレプリカは常に AWS Key Management Service (AWS KMS) に保存されている暗号化キーを使用して保管中のデータを暗号化します。

グローバルテーブル内のすべてのレプリカは、同じタイプの KMS キー (AWS 所有キー、AWS マネージドキー、またはカスタマーマネージドキー) で設定する必要があります。

> **Important**
>
> DynamoDB では、レプリカを削除するには、レプリカの暗号化キーにアクセスする必要があります。レプリカの暗号化に使用されるカスタマーマネージドキーを無効化または削除する場合、まずレプリカを削除し、残りのレプリカのいずれかのテーブルステータスが「ACTIVE」に変わるのを待ってから、キーを無効化または削除する必要があります。

MREC (マルチリージョンの結果整合性)

レプリカの暗号化に使用されるカスタマーマネージドキーへの DynamoDB のアクセスを無効化または取り消すと、レプリカとの間のレプリケーションは停止します。レプリカのステータスが INACCESSIBLE_ENCRYPTION_CREDENTIALS に変わります。MREC グローバルテーブルのレプリカが 20 時間以上「INACCESSIBLE_ENCRYPTION_CREDENTIALS」ステータスのままである場合、レプリカは単一リージョンの DynamoDB テーブルに変換され、元に戻せなくなります。

MRSC (マルチリージョンの強力な整合性)

レプリカの暗号化に使用されるカスタマーマネージドキーへの DynamoDB のアクセスを無効化または取り消すと、レプリカとの間のレプリケーションは停止します。レプリカへの書き込みまたは強力な整合性のある読み込みを実行しようとすると、エラーが返され、レプリカのステータスは INACCESSIBLE_ENCRYPTION_CREDENTIALS に変わります。MRSC グローバルテーブルのレプリカが 7 日以上「INACCESSIBLE_ENCRYPTION_CREDENTIALS」ステータスのままである場合、取り消された特定のアクセス許可に応じて、レプリカはアーカイブされるか、永続的にアクセスできなくなります。

DynamoDB マルチアカウントグローバルテーブル

マルチアカウントグローバルテーブルは、DynamoDB テーブルデータを複数の AWS リージョンと複数の AWS アカウントに自動的にレプリケートし、耐障害性を向上させ、アカウントレベルでワークロードを分離し、個別のセキュリティとガバナンスのコントロールを適用します。各レプリカテーブルは個別の AWS アカウントに存在するため、リージョンレベルとアカウントレベルの両方で障害分離できます。レプリカを AWS 組織構造に合わせて配置することもできます。マルチアカウントグローバルテーブルでは、同一アカウントグローバルテーブルと比較して、分離性、ガバナンス、およびセキュリティ上の利点がさらに高まります。

マルチアカウントグローバルテーブルには以下の利点があります。

- 選択した AWS アカウントとリージョン間で DynamoDB テーブルデータを自動的にレプリケートします
- 異なるポリシー、ガードレール、コンプライアンスの境界を持つアカウント間でデータをレプリケートすることで、セキュリティとガバナンスを強化します
- レプリカを別々の AWS アカウントに配置することで、運用の耐障害性とアカウントレベルの障害分離を改善します
- マルチアカウント戦略を使用する場合、ビジネスユニットまたは所有権ごとにワークロードを調整します

- 各レプリカをそれぞれの AWS アカウントに請求することで、コスト帰属を簡素化します

詳細については、「複数の AWS アカウントを使用するベネフィット」を参照してください。ワークロードにマルチアカウントレプリケーションが必要ない場合、またはローカルオーバーライドによるレプリカ管理を簡素化したい場合は、引き続き同じアカウントのグローバルテーブルを使用できます。

マルチリージョンの結果整合性 (MREC) を使用してマルチアカウントグローバルテーブルを設定できます。マルチリージョンの強力な整合性 (MRSC) 用に設定されたグローバルテーブルは、マルチアカウントモデルをサポートしていません。

トピック

- DynamoDB グローバルテーブルの仕組み
- チュートリアル: マルチアカウントグローバルテーブルの作成
- DynamoDB グローバルテーブルのセキュリティ

DynamoDB グローバルテーブルの仕組み

マルチアカウントグローバルテーブルは、DynamoDB グローバルテーブルのフルマネージド、サーバーレス、マルチリージョン、マルチアクティブ機能を複数の AWS アカウントに拡張します。マルチアカウントグローバルテーブルは、AWS リージョンとアカウント間でデータをレプリケートし、同一アカウントグローバルテーブルと同じアクティブ/アクティブ機能を提供します。いずれかのレプリカに書き込むと、DynamoDB は他のすべてのレプリカにデータをレプリケートします。

同一アカウントグローバルテーブルとの主な違いは次のとおりです。

- マルチアカウントレプリケーションは、マルチリージョンの結果整合性 (MREC) グローバルテーブルでサポートされています。
- レプリカを追加できるのは、単一リージョンテーブルから始める場合のみです。既存の同一アカウントグローバルテーブルをマルチアカウント設定に変換することはサポートされていません。移行するには、新しいマルチアカウントグローバルテーブルを作成する前に、既存のレプリカを削除して単一リージョンテーブルに戻す必要があります。
- 各レプリカは個別の AWS アカウントに存在する必要があります。N 個のレプリカを持つマルチアカウントグローバルテーブルの場合、N 個のアカウントが必要です。
- マルチアカウントグローバルテーブルは、デフォルトですべてのレプリカにわたって統合されたテーブル設定を使用します。すべてのレプリカは自動的に同じ設定 (スループットモード、TTL な

ど) を共有しますが、同一アカウントグローバルテーブルとは異なり、これらの設定はレプリカごとに上書きすることはできません。

- 顧客は、リソースポリシーで DynamoDB グローバルテーブルサービスプリンシパルにレプリケーション許可を提供する必要があります。

マルチアカウントグローバルテーブルは、同一アカウントグローバルテーブルと同じ基盤となるレプリケーションテクノロジーを使用します。テーブル設定はすべてのリージョンレプリカに自動的にレプリケートされ、顧客はレプリカごとに設定を上書きまたはカスタマイズすることはできません。これにより、同じグローバルテーブルに参加している複数の AWS アカウント間で一貫した設定と予測可能な動作が保証されます。

DynamoDB グローバルテーブルの設定は、テーブルの動作と、リージョン間でのデータのレプリケート方法を定義します。これらの設定は、テーブルの作成時または新しいリージョンレプリカの追加時に DynamoDB コントロールプレーン API を介して設定されます。

マルチアカウントグローバルテーブルを作成する場合、顧客はリージョンレプリカごとに `GlobalTableSettingsReplicationMode = ENABLED` を設定する必要があります。これにより、1 つのリージョンで行われた設定変更は、グローバルテーブルに参加する他のすべてのリージョンに自動的に伝播されます。

テーブルの作成後に設定のレプリケーションを有効にできます。これは、テーブルがもともとリージョンテーブルとして作成され、後でマルチアカウントグローバルテーブルにアップグレードされるシナリオをサポートします。

同期される設定

次のテーブル設定は、マルチアカウントグローバルテーブル内のすべてのレプリカ間で常に同期されます。

> **Note**
>
> 同一アカウントグローバルテーブルとは異なり、マルチアカウントグローバルテーブルでは、これらの設定をリージョンごとに上書きすることはできません。唯一の例外は、読み取り自動スケーリングポリシー (テーブルと GSI) は個別の外部リソースであるため、上書きが許可されることです。

- キャパシティモード (プロビジョンドキャパシティまたはオンデマンド)

- テーブルのプロビジョニングされた読み込みおよび書き込みキャパシティ
- テーブルの読み込みおよび書き込みの自動スケーリング
- ローカルセカンダリインデックス (LSI) 定義
- グローバルセカンダリインデックス (GSI) 定義
- GSI プロビジョニングされた読み込みおよび書き込みキャパシティ
- GSI の読み込みおよび書き込みの自動スケーリング
- MREC モードでの Streams 定義
- 有効期限 (TTL)
- ウォームスループット
- オンデマンドの最大読み取りおよび書き込みスループット

同期されない設定

以下の設定はレプリカ間で同期されないため、リージョンごとにレプリカテーブルごとに個別に設定する必要があります。

- テーブルクラス
- サーバー側の暗号化 (SSE)
- ポイントインタイムリカバリ
- サーバー側の暗号化 (SSE) KMS キー ID
- 削除保護
- Kinesis Data Streams (KDSD)
- タグ
- リソースポリシー
- テーブル Cloudwatch-Contributor Insights (CCI)
- GSI Cloudwatch-Contributor Insights (CCI)

モニタリング

マルチリージョンの結果整合性 (MREC) 用に設定されたグローバルテーブルは `ReplicationLatency` メトリクスを CloudWatch に発行します。このメトリクスは、項目がレプリカテーブルに書き込まれてから、その項目がグローバルテーブルの別のレプリカに表示されるまで

の経過時間を追跡します。ReplicationLatency はミリ秒単位で表し、グローバルテーブル内のすべての送信元と送信先のリージョンペアに対して出力されます。

一般的な ReplicationLatency 値は、選択した AWS リージョン間の距離と、ワークロードタイプやスループットなどの他の変数によって異なります。例えば、米国西部 (北カリフォルニア) (us-west-1) リージョンのソースレプリカは、アフリカ (ケープタウン) (af-south-1) リージョンと比較して、米国西部 (オレゴン) (us-west-2) リージョンへの ReplicationLatency が低くなります。

ReplicationLatency の値が上昇している場合、1 つのレプリカからの更新が他のレプリカテーブルにタイムリーに伝播されていないことを示している可能性があります。この場合、アプリケーションの読み込みおよび書き込みアクティビティを別の AWS リージョンに一時的にリダイレクトすることができます。

マルチアカウントグローバルテーブルでのレプリケーションレイテンシーの問題の処理

レプリカテーブルで顧客に起因する問題が原因で ReplicationLatency が 3 時間を超える場合、DynamoDB は顧客に根本的な問題に対処するように求める通知を送信します。レプリケーションを妨げる可能性がある一般的な顧客に起因する問題には、次のようなものがあります。

- レプリカテーブルのリソースポリシーから必要なアクセス許可を削除する
- マルチアカウントグローバルテーブルのレプリカをホストする AWS リージョンをオプトアウトする
- データの復号に必要なテーブルの AWS KMS キーアクセス許可を拒否する

DynamoDB は、レプリケーションレイテンシーが増加してから 3 時間以内に最初の通知を送信し、問題が解決されない場合は 20 時間後に 2 回目の通知を送信します。必要な時間枠内に問題が修正されない場合、DynamoDB はグローバルテーブルからレプリカの関連付けを自動的に解除します。その後、影響を受けるレプリカはリージョンテーブルに変換されます。

チュートリアル: マルチアカウントグローバルテーブルの作成

このセクションでは、複数の AWS アカウントにまたがる DynamoDB グローバルテーブルを作成するための段階的な手順について説明します。

DynamoDB コンソールを使用してマルチアカウントグローバルテーブルを作成する

AWS マネジメントコンソール を使用してマルチアカウントグローバルテーブルを作成するには、次の手順に従います。以下の例では、レプリカテーブルを持つグローバルテーブルを米国に作成します。

1. AWS マネジメントコンソール にサインインして、最初のアカウント (例: *111122223333*) の DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. この例では、ナビゲーションバーのリージョンセレクターから [米国東部 (オハイオ)] を選択します。
3. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
4. [Create Table] (テーブルの作成) を選択します。
5. [テーブルの作成] ページで、次の操作を行います。

   a. [テーブル名] に **MusicTable** と入力します。

   b. [パーティションキー] に **Artist** と入力します。

   c. [ソートキー] に **SongTitle** と入力します。

   d. 他のデフォルトの設定はそのままにして、[テーブルの作成] を選択します。
6. 次のリソースポリシーをテーブルに追加します

   JSON

```
{
"Version":"2012-10-17",
"Statement": [
    {
        "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
        "Effect": "Allow",
        "Action": [
            "dynamodb:ReadDataForReplication",
            "dynamodb:WriteDataForReplication",
            "dynamodb:ReplicateSettings"
        ],
        "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable",
        "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": ["444455556666","111122223333"],
                "aws:SourceArn": [
                    "arn:aws:dynamodb:us-east-1:444455556666:table/
MusicTable",
                    "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable"
                ]
```

```
            }
        }
    },
    {
        "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
        "Effect": "Allow",
        "Action": [
            "dynamodb:AssociateTableReplica"
        ],
        "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable",
        "Principal": {"AWS": ["444455556666"]}
    }
]
}
```

7. この新しいテーブルは、新しいグローバルテーブルの最初のレプリカテーブルとして機能します。これは、後で追加する他のレプリカテーブルのプロトタイプです。
8. テーブルが [アクティブ] になるまで待ちます。新しく作成したテーブルについては、[グローバルテーブル] タブから [設定のレプリケーション] に移動し、[有効化] を選択します。
9. このアカウント (ここでは *111122223333*) からログアウトします。
10. AWS マネジメントコンソール にサインインし、2 番目のアカウント (例: *444455556666*) の DynamoDB コンソール https://console.aws.amazon.com/dynamodb/ を開きます。
11. この例では、ナビゲーションバーのリージョンセレクターから [米国東部 (バージニア北部)] を選択します。
12. コンソールは、選択したリージョンに同一名のテーブルが存在しないことを確認します。同一名のテーブルが存在する場合は、そのリージョンで新しいレプリカテーブルを作成する前に既存のテーブルを削除する必要があります。
13. [テーブルの作成] の近くのドロップダウンで、[別のアカウントから作成] を選択します。
14. [別のアカウントからテーブルを作成する] ページで次の操作を行います。

    a. ソーステーブルのテーブル ARN として **arn:aws:dynamodb:us-east-2:*111122223333*:table/MusicTable** を追加します。

    b. [レプリカテーブル ARN] で、ソーステーブルの ARN を再度追加します (**arn:aws:dynamodb:us-east-2:*111122223333*:table/MusicTable**)。マルチアカウントグローバルテーブルの一部として複数のレプリカが既に存在する場合は、既存のすべてのレプリカを ReplicaTableARN に追加する必要があります。

c. [リソースベースのポリシー] セクションで、このアカウント (*444455556666*) のテーブルに次のリソースポリシーを追加します。マルチアカウントグローバルテーブルのレプリケーションは双方向であるため、各レプリカテーブルは DynamoDB レプリケーションサービスおよび他の参加アカウントに、レプリケーションに必要なアクセス許可を付与する必要があります。これは、最初のアカウントで追加したポリシーをミラーリングしたもので、アカウント参照が逆になっています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:444455556666:table/
MusicTable",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": ["444455556666","111122223333"],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:us-east-1:444455556666:table/
MusicTable",
                        "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable"
                    ]
                }
            }
        }
    ]
}
```

d. 他のデフォルトの設定はそのままにして、[送信] を選択します。

15. Music テーブル (およびその他すべてのレプリカテーブル) の [グローバルテーブル] タブに、テーブルが複数のリージョンでレプリケートされたことが示されます。

16. レプリケーションをテストするには、次の操作を行います。

a. このテーブルのレプリカが存在する任意のリージョンを使用できます。
b. [テーブルアイテムの探索] を選択します。
c. [項目を作成] を選択します。
d. [アーティスト] に「**item_1**」、[曲名] に「**Song Value 1**」と入力します。
e. [項目を作成] を選択します。
f. 他のリージョンに切り替えてレプリケーションを確認します。
g. [ミュージック] テーブルに、作成した項目が含まれていることを確認します。

AWS CLI を使用してマルチアカウントグローバルテーブルを作成する

次の例は、AWS CLI を使用してマルチアカウントグローバルテーブルを作成する方法を示しています。これらの例は、クロスアカウントレプリケーションを設定するための完全なワークフローを示しています。

CLI

クロスアカウントレプリケーションでマルチアカウントグローバルテーブルを作成するには、次の AWS CLI コマンドを使用します。

```
# STEP 1: Setting resource policy for the table in account 111122223333

cat > /tmp/source-resource-policy.json << 'EOF'
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": ["444455556666","111122223333"],
                    "aws:SourceArn": [
```

```
                        "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
                        "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable",
            "Principal": {"AWS": ["444455556666"]}
        }
    ]
}
EOF

# Step 2: Create a new table (MusicTable) in US East (Ohio),
#   with DynamoDB Streams enabled (NEW_AND_OLD_IMAGES),
#   and Settings Replication ENABLED on the account 111122223333

aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --global-table-settings-replication-mode ENABLED \
    --resource-policy file:///tmp/source-resource-policy.json \
    --region us-east-2


# Step 3: Creating replica table in account 444455556666

# Resource policy for account 444455556666
cat > /tmp/dest-resource-policy.json << 'EOF'
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": ["444455556666","111122223333"],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:us-east-1:444455556666:table/MusicTable",
                        "arn:aws:dynamodb:us-east-2:111122223333:table/MusicTable"
                    ]
                }
            }
        }
    ]
}
EOF

# Execute the replica table creation
aws dynamodb create-table \
    --table-name MusicTable \
    --global-table-source-arn "arn:aws:dynamodb:us-east-2:111122223333:table/
MusicTable" \
    --resource-policy file:///tmp/dest-resource-policy.json \
    --global-table-settings-replication-mode ENABLED \
    --region us-east-1

# Step 4: View the list of replicas created using describe-table
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistency,R
{Region:RegionName,Status:ReplicaStatus}}'

# Step 5: To verify that replication is working, add a new item to the Music table
 in US East (Ohio)
```

```
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2

# Step 6: Wait for a few seconds, and then check to see whether the item has been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1

aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2

# Step 7: Delete the replica table in US East (N. Virginia) Region
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-1

# Clean up: Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

DynamoDB グローバルテーブルのセキュリティ

グローバルテーブルレプリカは DynamoDB テーブルであるため、AWS Identity and Access Management (IAM) ID ポリシーやリソースベースのポリシーなど、単一リージョンテーブルに対して行うレプリカへのアクセスを制御するのと同じ方法を使用します。このトピックでは、IAM アクセス許可と AWS Key Management Service (AWS KMS) 暗号化を使用して DynamoDB マルチアカウントグローバルテーブルを保護する方法について説明します。クロスリージョンクロスアカウントレプリケーションと自動スケーリングを許可するリソースベースのポリシーとサービスリンクロール (SLR) について、また、マルチリージョンの結果整合性 (MREC) テーブルに対して、グローバルテーブルの作成、更新、削除に必要な IAM アクセス許可について説明します。また、クロスリージョンレプリケーションを安全に管理するための AWS KMS 暗号化キーについても説明します。

クロスアカウントおよびクロスリージョンテーブルレプリケーションを確立するために必要なリソースベースのポリシーとアクセス許可に関する詳細情報を提供します。このセキュリティモデルを理解することは、安全なクロスアカウントデータレプリケーションソリューションを実装する必要があるお客様にとって不可欠です。

レプリケーションのサービスプリンシパル承認

DynamoDB のマルチアカウントグローバルテーブルは、レプリケーションがアカウントの境界を越えて実行されるため、個別の承認アプローチを使用します。これは、DynamoDB のレプリケーションサービスプリンシパルを使用して行われます (`replication.dynamodb.amazonaws.com`)。各参加アカウントは、レプリカテーブルのリソースポリシーでそのプリンシパルを明示的に許可し、`aws:SourceAccount`、`aws:SourceArn` などのキーのソースコンテキスト条件によって特定のレプリカに制限できるアクセス許可を付与する必要があります。詳細については、「AWS グローバル条件キー」を参照してください。アクセス許可は双方向です。つまり、レプリケーションを特定のレプリカのペア間で確立する前に、すべてのレプリカが相互にアクセス許可を明示的に付与する必要があります。

クロスアカウントレプリケーションには、次のサービスプリンシパルのアクセス許可が不可欠です。

- `dynamodb:ReadDataForReplication` は、レプリケーション目的でデータを読み取る機能を付与します。このアクセス許可により、1 つのレプリカの変更を読み取って他のレプリカに伝播できます。
- `dynamodb:WriteDataForReplication` は、レプリケート先テーブルへのレプリケートデータの書き込みを許可します。このアクセス許可により、グローバルテーブル内のすべてのレプリカ間で変更を同期できます。
- `dynamodb:ReplicateSettings` を使用すると、レプリカ間でテーブル設定を同期できるため、すべての参加テーブルで一貫した設定が可能になります。

各レプリカは、上記のアクセス許可を他のすべてのレプリカとそれ自体に付与する必要があります。つまり、ソースコンテキスト条件には、グローバルテーブルを構成するレプリカの完全なセットが含まれている必要があります。これらのアクセス許可は、新しいレプリカがマルチアカウントグローバルテーブルに追加されるたびに検証されます。これにより、レプリケーションオペレーションが承認された DynamoDB サービスによってのみ実行され、目的のテーブル間でのみ実行されることが検証されます。

マルチアカウントグローバルテーブルのサービスリンクロール

DynamoDB マルチアカウントグローバルテーブルは、すべてのレプリカ間で設定をレプリケートするため、各レプリカは一貫したスループットで同じようにセットアップされ、シームレスなフェイルオーバーエクスペリエンスを提供します。設定のレプリケーションは、サービスプリンシパルに対する `ReplicateSettings` アクセス許可によって制御されますが、DynamoDB は特定のクロスアカウントクロスリージョンレプリケーションと自動スケーリング機能の管理のために、サービスにリンクされたロール (SLR) も利用します。これらのロールは、AWS アカウントごとに 1 回だけ設定されます。作成後、同じロールがアカウント内のすべてのグローバルテーブルに機能します。サービスリンクロールの詳細については、IAM ユーザーガイドの「サービスリンクロールの使用」を参照してください。

設定管理サービスにリンクされたロール

Amazon DynamoDB は、アカウントで最初のマルチアカウントグローバルテーブルレプリカを作成すると、AWSServiceRoleForDynamoDBGlobalTableSettingsManagement サービスリンクロール (SLR) を自動的に作成します。このロールは、クロスアカウントクロスリージョンの設定のレプリケーションを管理します。

リソースベースのポリシーをレプリカに適用する場合は、`AWSServiceRoleForDynamoDBGlobalTableSettingsManagement` で定義されている SLR プリンシパルのアクセス許可をいずれも拒否しないことを確認してください。拒否すると、設定管理が妨げられ、レプリカ間または GSI 間でスループットが一致しない場合にレプリケーションに支障をきたす可能性があります。必要な SLR アクセス許可を拒否すると、影響を受けるレプリカとの間のレプリケーションが停止し、レプリカテーブルのステータスは「`REPLICATION_NOT_AUTHORIZED`」に変わります。マルチアカウントグローバルテーブルについて、レプリカが 20 時間以上「`REPLICATION_NOT_AUTHORIZED`」ステータスのままである場合、レプリカは 1 つのリージョンの DynamoDB テーブルに変換され、元に戻せなくなります。SLR には、以下のアクセス許可があります。

- `application-autoscaling:DeleteScalingPolicy`
- `application-autoscaling:DescribeScalableTargets`
- `application-autoscaling:DescribeScalingPolicies`
- `application-autoscaling:DeregisterScalableTarget`
- `application-autoscaling:PutScalingPolicy`
- `application-autoscaling:RegisterScalableTarget`

自動スケーリングサービスにリンクされたロール

プロビジョンドキャパシティモードのグローバルテーブルを設定する場合は、グローバルテーブルに自動スケーリングを設定する必要があります。DynamoDB 自動スケーリングは、AWS Application Auto Scaling Service を使用して、グローバルテーブルレプリカのプロビジョニングされたスループットキャパシティを動的に調整します。Application Auto Scaling サービスは、AWSServiceRoleForApplicationAutoScaling_DynamoDBTable という名前のサービスにリンクされたロール (SLR) を作成します。このサービスにリンクされたロールは、DynamoDB テーブルの自動スケーリングを初めて設定するときに、AWS アカウントで自動的に作成されます。これにより、Application Auto Scaling はプロビジョニングされたテーブル容量を管理でき、CloudWatch アラームを作成できます。

リソースベースのポリシーをレプリカに適用する際は、AWSApplicationAutoscalingDynamoDBTablePolicy で定義されている Application Auto Scaling SLR プリンシパルへのアクセス許可を拒否していないことを確認してください。拒否すると、自動スケーリング機能が中断されます。

グローバルテーブルで AWS IAM を使用する方法

次のセクションでは、さまざまなグローバルテーブルのオペレーションに必要なアクセス許可について説明し、ユーザーとアプリケーションに適切なアクセスを設定するのに役立つポリシーの例を示します。

> **Note**
>
> 説明されているすべてのアクセス許可は、影響を受けるリージョンの特定のテーブルリソース ARN に適用する必要があります。テーブルリソース ARN は `arn:aws:dynamodb:region:account-id:table/table-name` の形式に従います。ここでは、実際のリージョン、アカウント ID、テーブル名の値を指定する必要があります。

以下のセクションでは、順を追って以下のトピックを取り上げます。

- マルチアカウントグローバルテーブルの作成とレプリカの追加
- マルチアカウントグローバルテーブルの更新
- グローバルテーブルとレプリカの削除

グローバルテーブルの作成とレプリカの追加

グローバルテーブルを作成するためのアクセス許可

リージョンテーブルに新しいレプリカを追加してマルチアカウントグローバルテーブルを作成する場合、または既存のマルチアカウントグローバルテーブルに追加する場合、アクションを実行する IAM プリンシパルは既存のすべてのメンバーによって承認される必要があります。既存のメンバーはすべて、レプリカを正常に追加するために、テーブルポリシーで次のアクセス許可を付与する必要があります。

- `dynamodb:AssociateTableReplica` - このアクセス許可により、テーブルをグローバルテーブル設定に結合できます。これは、レプリケーション関係の初期確立を可能にする基本的なアクセス許可です。

この正確なコントロールにより、承認されたアカウントのみがグローバルテーブルのセットアップに参加できるようになります。

グローバルテーブルの作成のための IAM ポリシーの例

2 つのレプリカ設定の IAM ポリシーの例

マルチアカウントグローバルテーブルのセットアップは、安全なレプリケーションを提供する特定の認可フローに従います。顧客が 2 つのレプリカを持つグローバルテーブルを確立するという実践的なシナリオを順に見ていくことで、これが実際にどのように機能するかを確認してみましょう。最初のレプリカ (ReplicaA) は ap-east-1 リージョンのアカウント A に存在し、2 番目のレプリカ (ReplicaB) は eu-south-1 リージョンのアカウント B にあります。

- ソースアカウント (アカウント A) では、プロセスはプライマリレプリカテーブルの作成から始まります。アカウント管理者は、関連付けを実行するために必要なアクセス許可を送信先アカウント (アカウント B) に明示的に付与するリソースベースのポリシーをこのテーブルにアタッチする必要があります。このポリシーは、DynamoDB レプリケーションサービスが重要なレプリケーションアクションを実行することも許可します。
- 送信先アカウント (アカウント B) は、レプリカの作成時に対応するリソースベースのポリシーをアタッチし、レプリカの作成に使用するソーステーブル ARN を参照することで、同様のプロセスに従います。このポリシーは、アカウント A によって付与されたアクセス許可をミラーリングし、信頼された双方向の関係を作成します。レプリケーションを確立する前に、DynamoDB はこれらのクロスアカウントアクセス許可を検証して、適切な承認が設定されていることを確認します。

このセットアップを確立するには。

- アカウント A の管理者は、まずリソースベースのポリシーを ReplicaA にアタッチする必要があります。このポリシーは、アカウント B と DynamoDB レプリケーションサービスに必要なアクセス許可を明示的に付与します。
- 同様に、アカウント B の管理者は、一致するポリシーを ReplicaB にアタッチする必要があります。アカウント参照を逆にして、レプリカ A をソーステーブルとして参照するレプリカ B を作成するためのテーブル作成呼び出しで、対応するアクセス許可をアカウント A に付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
```

```
            "Principal": {"AWS": ["444455556666"]}
        }
    ]
}
```

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        }
    ]
}
```

3 つのレプリカ設定の IAM ポリシーの例

この設定では、アカウント A、アカウント B、アカウント C に、それぞれ 3 つのレプリカ ReplicaA、ReplicaB、および ReplicaC があります。ReplicaA は最初のレプリカであり、リージョンテーブルとして開始され、その後に ReplicaB と ReplicaC が追加されます。

- アカウント A の管理者は、まず、すべてのメンバーとのレプリケーションを許可し、アカウント B とアカウント C の IAM プリンシパルがレプリカを追加できるように、リソースベースのポリシーを ReplicaA にアタッチする必要があります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666",
 "123456789012" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
                        "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
            ],
            "Resource": "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
            "Principal": { "AWS": [ "444455556666", "123456789012" ] }
        }
    ]
```

```
}
```

- アカウント B の管理者は、ReplicaA をソースとしてポイントするレプリカ (ReplicaB) を追加する必要があります。ReplicaB には、すべてのメンバー間のレプリケーションを許可し、アカウント C にレプリカの追加を許可する次のポリシーがあります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666",
 "123456789012" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
                        "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC"
                    ]
                }
            }
        },
        {
            "Sid": "AllowTrustedAccountsToJoinThisGlobalTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:AssociateTableReplica"
```

```
            ],
            "Resource": "arn:aws:dynamodb:eu-south-1:444455556666:table/
ReplicaB",
            "Principal": { "AWS": [ "123456789012" ] }
        }
    ]
}
```

- 最後に、アカウント C の管理者は、すべてのメンバー間のレプリケーションアクセス許可を許可する次のポリシーを使用してレプリカを作成します。ポリシーにより、これ以上のレプリカの追加は許可されません。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBActionsNeededForSteadyStateReplication",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ReadDataForReplication",
                "dynamodb:WriteDataForReplication",
                "dynamodb:ReplicateSettings"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/ReplicaC",
            "Principal": {"Service": ["replication.dynamodb.amazonaws.com"]},
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": [ "111122223333", "444455556666" ],
                    "aws:SourceArn": [
                        "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                        "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                    ]
                }
            }
        }
    ]
}
```

マルチアカウントグローバルテーブルの更新

UpdateTable API を使用して既存のグローバルテーブルのレプリカ設定を変更するには、API コールを行うリージョンのテーブルリソースに対して次のアクセス許可が必要です。`dynamodb:UpdateTable`

自動スケーリングポリシーや有効期限設定など、他のグローバルテーブル設定も更新できます。これらの追加の更新オペレーションには、次のアクセス許可が必要です。

`UpdateTimeToLive` API を使用して有効期限設定を更新するには、レプリカを含むすべてのリージョンのテーブルリソースに対して次のアクセス許可が必要です。`dynamodb:UpdateTimeToLive`

`UpdateTableReplicaAutoScaling` API を使用してレプリカの自動スケーリングポリシーを更新するには、レプリカを含むすべてのリージョンのテーブルリソースに対して、次のアクセス許可が必要です。

- `application-autoscaling:DeleteScalingPolicy`
- `application-autoscaling:DeleteScheduledAction`
- `application-autoscaling:DeregisterScalableTarget`
- `application-autoscaling:DescribeScalableTargets`
- `application-autoscaling:DescribeScalingActivities`
- `application-autoscaling:DescribeScalingPolicies`
- `application-autoscaling:DescribeScheduledActions`
- `application-autoscaling:PutScalingPolicy`
- `application-autoscaling:PutScheduledAction`
- `application-autoscaling:RegisterScalableTarget`

> **Note**
>
> テーブルの更新を成功させるには、すべてのレプリカリージョンとアカウントで `dynamodb:ReplicateSettings` アクセス許可を付与する必要があります。いずれかのレプリカがマルチアカウントグローバルテーブル内のいずれかのレプリカに設定をレプリケートするアクセス許可を付与していない場合、アクセス許可が修正されるまで、すべてのレプリカのすべての更新オペレーションは `AccessDeniedException` で失敗します。

グローバルテーブルとレプリカの削除

グローバルテーブルを削除するには、すべてのレプリカを削除する必要があります。同一アカウントグローバルテーブルとは異なり、UpdateTable を使用してリモートリージョン内のレプリカテーブルを削除することはできません。各レプリカはそのテーブルを制御するアカウントから DeleteTable API を介して削除する必要があります。

グローバルテーブルとレプリカを削除するアクセス許可

個々のレプリカを削除したり、グローバルテーブルを完全に削除したりするには、次のアクセス許可が必要です。グローバルテーブル設定を削除すると、異なるリージョンのテーブル間のレプリケーション関係のみが削除されます。最後に残っているリージョンの基盤となる DynamoDB テーブルは削除されません。最後のリージョンのテーブルは、同じデータと設定を持つ標準の DynamoDB テーブルとして引き続き存在し続けます。

レプリカを削除する各リージョンのテーブルリソースには、次のアクセス許可が必要です。

- dynamodb:DeleteTable
- dynamodb:DeleteTableReplica

グローバルテーブルで AWS KMS を使用する方法

すべての DynamoDB テーブルと同様に、グローバルテーブルレプリカは常に AWS Key Management Service (AWS KMS) に保存されている暗号化キーを使用して保管中のデータを暗号化します。

> **Note**
>
> 同一アカウントグローバルテーブルとは異なり、マルチアカウントグローバルテーブル内の異なるレプリカは、異なるタイプの AWS KMS キー (AWS 所有キー、またはカスタマーマネージドキー) を使用して設定できます。マルチアカウントグローバルテーブルは AWS マネージドキーをサポートしていません。

CMK を使用するマルチアカウントグローバルテーブルでは、各レプリカのキーポリシーで、レプリケーションと設定管理のためのキーにアクセスするためのアクセス許可を DynamoDB レプリケーションサービスプリンシパル (replication.dynamodb.amazonaws.com) に付与する必要があります。以下のアクセス権限が必要です。

- kms:Decrypt
- kms:ReEncrypt*
- kms:GenerateDataKey*
- kms:DescribeKey

重要

DynamoDB では、レプリカを削除するには、レプリカの暗号化キーにアクセスする必要があります。レプリカを削除するため、レプリカの暗号化に使用されるカスタマーマネージドキーを無効化または削除する場合、まずレプリカを削除し、残りのレプリカの 1 つで記述を呼び出してレプリケーショングループからテーブルが削除されるまで待機してから、キーを無効化または削除する必要があります。

レプリカの暗号化に使用されるカスタマーマネージドキーへの DynamoDB のアクセスを無効化または取り消すと、レプリカとの間のレプリケーションは停止し、レプリカのステータスは「INACCESSIBLE_ENCRYPTION_CREDENTIALS」に変わります。レプリカが 20 時間以上「INACCESSIBLE_ENCRYPTION_CREDENTIALS」ステータスのままである場合、レプリカは単一リージョンの DynamoDB テーブルに変換され、元に戻せなくなります。

AWS KMS ポリシーの例

この AWS KMS ポリシーにより、DynamoDB はレプリカ A と B 間のレプリケーションのために両方の AWS KMS キーにアクセスできます。各アカウントの DynamoDB レプリカにアタッチされた AWS KMS キーは、次のポリシーで更新する必要があります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": { "Service": "replication.dynamodb.amazonaws.com" },
        "Action": [
            "kms:Decrypt",
            "kms:ReEncrypt*",
            "kms:GenerateDataKey*",
            "kms:DescribeKey"
```

```
        ],
        "Resource": "*",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": [ "111122223333", "444455556666" ],
                "aws:SourceArn": [
                    "arn:aws:dynamodb:ap-east-1:111122223333:table/ReplicaA",
                    "arn:aws:dynamodb:eu-south-1:444455556666:table/ReplicaB"
                ]
            }
        }
      }
    ]
}
```

Amazon DynamoDB のグローバルテーブルの請求について

このガイドでは、グローバルテーブルに対する DynamoDB の請求の仕組みを説明し、グローバルテーブルのコストに寄与するコンポーネントを実用的な例を含めて特定します。

Amazon DynamoDB グローバルテーブルは、完全マネージド型の、サーバーレス、マルチリージョン、マルチアクティブデータベースです。グローバルテーブルは、99.999% の可用性を提供するように設計されており、アプリケーションの耐障害性の向上、事業継続性の向上を実現します。グローバルテーブルは、選択した AWS リージョン間で DynamoDB テーブルを自動的にレプリケートするため、ローカルな読み取りおよび書き込みパフォーマンスを高速化できます。

仕組み

グローバルテーブルの請求モデルは、単一リージョンの DynamoDB テーブルとは異なります。単一リージョンの DynamoDB テーブルの書き込みオペレーションは、次の単位を使用して請求されます。

- オンデマンドキャパシティモードの場合は、書き込みリクエストユニット (WRU)。最大 1 KB の書き込みごとに 1 WRU が課金されます。
- プロビジョンドキャパシティモードの場合は、書き込みキャパシティユニット (WCU)。1 WCU で、1 秒あたり最大 1 KB の書き込みが 1 回提供されます。

既存の単一リージョンのテーブルにレプリカテーブルを追加してグローバルテーブルを作成すると、その単一リージョンテーブルはレプリカテーブルになります。つまり、テーブルへの書き込みの請求

に使用されるユニットも変更されます。レプリカテーブルへの書き込みオペレーションは、次の単位を使用して請求されます。

- オンデマンドキャパシティモードの場合は、レプリケート書き込みリクエストユニット (rWRU)。最大 1KB の書き込みごとにレプリカテーブルあたり 1 rWRU が課金されます。
- プロビジョンドキャパシティモードの場合は、レプリケート書き込みキャパシティユニット (rWCU)。レプリカテーブルごとに 1 WCU で、1 秒あたり最大 1 KB の書き込みが 1 回提供されます。

グローバルセカンダリインデックス (GSI) の更新は、GSI のベーステーブルがレプリカテーブルであっても、単一リージョンの DynamoDB テーブルと同じユニットを使用して請求されます。GSI の更新オペレーションは、次の単位を使用して請求されます。

- オンデマンドキャパシティモードの場合は、書き込みリクエストユニット (WRU)。最大 1 KB の書き込みごとに 1 WRU が課金されます。
- プロビジョンドキャパシティモードの場合は、書き込みキャパシティユニット (WCU)。1 WCU で、1 秒あたり最大 1 KB の書き込みが 1 回提供されます。

レプリケート書き込みユニット (rWCU および rWRU) の料金は、単一リージョンの書き込みユニット (WCU および WRU) と同じです。レプリケート書き込み (rWCU または rWRU) の料金は、グローバルテーブルのレプリカテーブルを含むすべてのリージョンで発生します。DynamoDB では、グローバルテーブルのリージョン間でデータをレプリケートする場合、リージョン間のデータ転送料金は発生しません。

単一リージョンテーブルとレプリカテーブルからの読み取りオペレーションでは、次の単位を使用します。

- オンデマンドキャパシティモードの場合は、読み込みリクエストユニット (RRU)。最大 4 KB の強力な整合性のある読み込みごとに 1 RRU が課金されます。
- プロビジョニングされたテーブルの場合は、読み込みキャパシティユニット (RCU)。1 RCU で、1 秒あたり最大 4 KB の強力な整合性のある読み込みが 1 回提供されます。

整合性モードと請求

書き込みオペレーションの請求に使用されるレプリケートされた書き込みユニット (rWCU と rWRU) は、マルチリージョンの強力な整合性 (MRSC) モードとマルチリージョンの結果整合性 (MREC) モードの両方で同一です。監視で設定されたマルチリージョンの強力な整合性 (MRSC) モードを使

用するグローバルテーブルでは、レプリケートされた書き込みユニットコスト (rWCU と rWRU)、ストレージコスト、または監視へのレプリケーションのデータ転送コストは発生しません。

DynamoDB グローバルテーブルの請求例

グローバルテーブル書き込みリクエストの請求の実際の仕組みについては、複数日間のシナリオ例を順を追って説明します (この例では書き込みリクエストのみを考慮し、レプリカを追加するときに発生するテーブルの復元料金は含まれていないことに注意してください)。

1 日目 - 単一リージョンテーブル: us-west-2 リージョンに Table_A という単一リージョンのオンデマンド DynamoDB テーブルがあります。100 個の 1 KB 項目を Table_A に書き込みます。単一リージョンの書き込みオペレーションの場合、書き込み 1 KB あたり 1 書き込みリクエストユニット (WRU) が課金されます。1 日目の料金は次のとおりです。

- us-west-2 リージョンで単一リージョン書き込み用に 100 WRU

1 日目に請求される合計リクエストユニット: 100 WRU。

2 日目 - グローバルテーブルの作成: us-east-2 リージョンに Table_A のレプリカを追加してグローバルテーブルを作成します。Table_A は、us-west-2 リージョンと us-east-2 リージョンの 2 つのレプリカテーブルを持つグローバルテーブルになりました。us-west-2 リージョンのレプリカテーブルに 150 個の 1 KB 項目を書き込みます。2 日目の料金は次のとおりです。

- us-west-2 リージョンでレプリケートされた書き込み用に 150 rWRU
- us-east-2 リージョンでレプリケートされた書き込み用に 150 rWRU

2 日目に請求されるリクエストユニットの合計: 300 rWRU。

3 日目 - グローバルセカンダリインデックスの追加: us-east-2 リージョンのレプリカテーブルに、ベース (レプリカ) テーブルからすべての属性を射影するグローバルセカンダリインデックス (GSI) を追加します。グローバルテーブルは、us-west-2 リージョンのレプリカテーブルに GSI を自動的に作成します。us-west-2 リージョンのレプリカテーブルに 200 個の新しい 1 KB レコードを書き込みます。3 日目の料金は次のとおりです。

- • us-west-2 リージョンでレプリケートされた書き込み用に 200 rWRU
- • us-west-2 リージョンで GSI 更新用に 200 WRU
- • us-east-2 リージョンでレプリケートされた書き込み用に 200 rWRU
- • us-east-2 リージョンで GSI 更新用に 200 WRU

3 日目に請求される書き込みリクエストユニットの合計: 400 WRU と 400 rWRU。

3 日間すべての書き込みユニット合計で、500 WRU (1 日目に 100 WRU + 3 日目に 400 WRU) と 700 rWRU (2 日目に 300 rWRU + 3 日目に 400 rWRU) の料金が発生します。

まとめとして、レプリカテーブルの書き込みオペレーションは、レプリカテーブルを含むすべてのリージョンで、レプリケートされた書き込みユニットとして請求されます。グローバルセカンダリインデックスがある場合、GSI を含むすべてのリージョン (グローバルテーブルではレプリカテーブルを含むすべてのリージョン) で GSI の更新に対して書き込みユニットが課金されます。

DynamoDB グローバルテーブルのバージョン

DynamoDB グローバルテーブルには、グローバルテーブルバージョン 2019.11.21 (現行) とグローバルテーブルバージョン 2017.11.29 (レガシー) の 2 つのバージョンがあります。グローバルテーブルバージョン 2019.11.21 (現行) を使用することをお勧めします。これは、バージョン 2017.11.29 (レガシー) と比較して、使いやすく、より多くのリージョンでサポートされ、ほとんどのワークロードでコストが低いためです。

グローバルテーブルのバージョンを確認する

AWS CLI を使用したバージョンの確認

バージョン 2019.11.21 (現行) グローバルテーブルレプリカの特定

テーブルがグローバルテーブルバージョン 2019.11.21 (現行) レプリカかどうかを判断するには、テーブルの `describe-table` コマンドを呼び出します。出力に「2019.11.21」の値を持つ `GlobalTableVersion` 属性が含まれている場合、テーブルはバージョン 2019.11.21 (現行) グローバルテーブルレプリカです。

`describe-table` の CLI コマンドの例:

```
aws dynamodb describe-table \
--table-name users \
--region us-east-2
```

(省略された) 出力には、値が「2019.11.21」の `GlobalTableVersion` 属性が含まれているため、このテーブルはバージョン 2019.11.21 (現行) グローバルテーブルレプリカです。

```
{
```

```
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "id",
                "AttributeType": "S"
            },
            {
                "AttributeName": "name",
                "AttributeType": "S"
            }
        ],
        "TableName": "users",
        ...
        "GlobalTableVersion": "2019.11.21",
        "Replicas": [
            {
                "RegionName": "us-west-2",
                "ReplicaStatus": "ACTIVE",
            }
        ],
        ...
    }
}
```

バージョン 2017.11.29 (レガシー) グローバルテーブルレプリカの識別

グローバルテーブルバージョン 2017.11.29 (レガシー) では、グローバルテーブル管理専用のコマンドセットが使用されます。テーブルがグローバルテーブルバージョン 2017.11.29 (レガシー) レプリカであるかどうかを確認するには、テーブルの `describe-global-table` コマンドを呼び出します。正常なレスポンスを受け取った場合、テーブルはバージョン 2017.11.29 (レガシー) グローバルテーブルレプリカです。`describe-global-table` コマンドが `GlobalTableNotFoundException` エラーを返した場合、テーブルはバージョン 2017.11.29 (レガシー) レプリカではありません。

`describe-global-table` の CLI コマンドの例:

```
aws dynamodb describe-global-table \
--global-table-name users \
--region us-east-2
```

コマンドは正常なレスポンスを返すため、このテーブルはバージョン 2017.11.29 (レガシー) グローバルテーブルレプリカです。

```
{
    "GlobalTableDescription": {
        "ReplicationGroup": [
            {
                "RegionName": "us-west-2"
            },
            {
                "RegionName": "us-east-2"
            }
        ],
        "GlobalTableArn": "arn:aws:dynamodb::123456789012:global-table/users",
        "CreationDateTime": "2025-06-10T13:55:53.630000-04:00",
        "GlobalTableStatus": "ACTIVE",
        "GlobalTableName": "users"
    }
}
```


DynamoDB コンソールを使用したバージョンの確認

グローバルテーブルレプリカのバージョンを特定するには、以下を実行します。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/home) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. グローバルテーブルのバージョンを識別するテーブルを選択します。
4. [グローバルテーブル] タブを選択します。

   [概要] セクションには、使用中のグローバルテーブルのバージョンが表示されます。

レガシーバージョンと現行バージョンの動作の違い

以下のリストは、グローバルテーブルのレガシーバージョンと現行バージョンの動作の違いを示しています。

- いくつかの DynamoDB オペレーションにおいて、バージョン 2019.11.21 (現行) は、バージョン 2017.11.29 (レガシー) よりも書き込みキャパシティの消費が少ないため、ほとんどのユーザーにとってコスト効率が向上します。DynamoDB オペレーションの違いは次のとおりです。
  - あるリージョンの 1KB アイテムに対して PutItem を呼び出し、他のリージョンにレプリケートする場合、2017.11.29 (レガシー) ではリージョンごとに 2 rWRU が必要ですが、2019.11.21 (現行) では 1 rWRU のみ必要です。

- 1KB アイテムに対して UpdateItem を呼び出す場合、2017.11.29 (レガシー) ではソースリージョンで 2 rWRU、送信先リージョンで 1 rWRU が必要ですが、2019.11.21 (現行) ではソースリージョンおよび送信先リージョンでそれぞれ 1 rWRU のみ必要です。
- 1KB アイテムに対して DeleteItem を呼び出す場合、2017.11.29 (レガシー) ではソースリージョンで 1 rWRU、送信先リージョンで 2 rWRU が必要ですが、2019.11.21 (現行) ではソースリージョンおよび送信先リージョンでそれぞれ 1 rWRU のみ必要です。

次の表は、2 つのリージョンの 1 KB 項目に対する 2017.11.29 (レガシー) テーブルと 2019.11.21 (現行) テーブルの rWRU 消費量を示しています。

Operation	2017.11.29 (レガシー)	2019.11.21 (現行)	削減量
PutItem	4 rWRU	2 rWRU	50%
UpdateItem	3 rWRU	2 rWRU	33%
DeleteItem	3 rWRU	2 rWRU	33%

- バージョン 2017.11.29 (レガシー) は、11 か所の AWS リージョンでのみ利用可能です。一方、バージョン 2019.11.21 (現行) は、すべての AWS リージョンで使用可能です。
- バージョン 2017.11.29 (レガシー) グローバルテーブルを作成するには、まず空のリージョンテーブルのセットを作成し、次に CreateGlobalTable API を呼び出してグローバルテーブルを作成します。バージョン 2019.11.21 (現行) グローバルテーブルを作成するには、UpdateTable API を呼び出して、既存のリージョンテーブルにレプリカを追加します。
- バージョン 2017.11.29 (レガシー) では、新しいリージョンにレプリカを追加する前 (作成時を含む) に、テーブル内のすべてのレプリカを空にする必要があります。バージョン 2019.11.21 (現行) では、データが既に含まれているリージョンのテーブルに対してレプリカを追加および削除できます。
- バージョン 2017.11.29 (レガシー) では、以下の専用コントロールプレーン API のセットを使用してレプリカを管理します。
  - CreateGlobalTable
  - DescribeGlobalTable
  - DescribeGlobalTableSettings
  - ListGlobalTables
  - UpdateGlobalTable

- UpdateGlobalTableSettings

バージョン 2019.11.21 (現行) は、DescribeTable API および UpdateTable API を使用してレプリカを管理します。

- バージョン 2017.11.29 (レガシー) は、書き込みごとに 2 つの DynamoDB Streams レコードを発行します。バージョン 2019.11.21 (現行) は、書き込みごとに 1 つの DynamoDB Streams レコードのみを発行します。
- バージョン 2017.11.29 (レガシー) は、`aws:rep:deleting`、`aws:rep:updateregion`、`aws:rep:updatetime` の各属性を入力および更新します。バージョン 2019.11.21 (現行) は、これらの属性を入力または更新しません。
- バージョン 2017.11.29 (レガシー) は、レプリカ間で DynamoDB での Time to Live (TTL) の使用 設定を同期しません。バージョン 2019.11.21 (現行) は、レプリカ間で TTL 設定を同期します。
- バージョン 2017.11.29 (レガシー) は、TTL 削除を他のレプリカにレプリケートしません。バージョン 2019.11.21 (現行) は、TTL 削除をすべてのレプリカにレプリケートします。
- バージョン 2017.11.29 (レガシー) は、レプリカ間で自動スケーリング設定を同期しません。バージョン 2019.11.21 (現行) は、レプリカ間で自動スケーリング設定を同期します。
- バージョン 2017.11.29 (レガシー) は、レプリカ間でグローバルセカンダリインデックス (GSI) 設定を同期しません。バージョン 2019.11.21 (現行) は、レプリカ間で GSI 設定を同期します。
- バージョン 2017.11.29 (レガシー) は、レプリカ間で保管中の暗号化設定を同期しません。バージョン 2019.11.21 (現行) は、レプリカ間で保管中の暗号化設定を同期します。
- バージョン 2017.11.29 (レガシー) は `PendingReplicationCount` メトリクスを発行します。バージョン 2019.11.21 (現行) は同メトリクスを発行しません。

現行バージョンへのアップグレード

グローバルテーブルのアップグレードに必要なアクセス許可

バージョン 2019.11.21 (現行) にアップグレードするには、レプリカがあるすべてのリージョンで `dynamodb:UpdateGlobalTableversion` アクセス許可が必要です。DynamoDB コンソールへのアクセス許可、およびテーブル表示に必要なアクセス許可に加えて、このアクセス許可が必要です。

次の IAM ポリシーは、グローバルテーブルをバージョン 2019.11.21 (現行) にアップグレードするアクセス許可を付与します。

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:UpdateGlobalTableversion",
            "Resource": "*"
        }
    ]
}
```

次の IAM ポリシーは、2 つのリージョンにレプリカがある Music グローバルテーブルのみをバージョン 2019.11.21 (現行) にアップグレードするアクセス許可を付与します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:UpdateGlobalTableversion",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Music",
                "arn:aws:dynamodb:ap-southeast-1:123456789012:table/Music",
                "arn:aws:dynamodb:us-east-2:123456789012:table/Music"
            ]
        }
    ]
}
```

アップグレード中の注意点

- すべてのグローバルテーブルレプリカは、アップグレード中も読み取りと書き込みのトラフィックを処理します。
- テーブルのサイズとレプリカの数にもよりますが、アップグレードプロセスには、数分から数時間かかります。
- アップグレードプロセス中、TableStatus の値は ACTIVE から UPDATING に変わります。テーブルのステータスを表示するには、DescribeTable API を呼び出すか、DynamoDB コンソールの [テーブル] ビューを使用します。
- テーブルのアップグレード中、自動スケーリングはグローバルテーブルのプロビジョニング済のキャパシティ設定を調整しません。アップグレード中は、テーブルをオンデマンドキャパシティモードに設定することを強くお勧めします。

- アップグレード中に自動スケーリングでプロビジョニングされた容量を使用する場合は、アップグレード中に予想されるトラフィックの増加に対応しスロットリングを回避するために、ポリシーの最小読み取りスループットと最小書き込みスループットを増やす必要があります。
- `ReplicationLatency` メトリクスは、アップグレードプロセス中にレイテンシーのスパイクを一時的に報告したり、メトリクスデータのレポートを停止したりする場合があります。詳細については、「the section called “ReplicationLatency”」を参照してください。
- アップグレードプロセスが完了すると、テーブルのステータスは `ACTIVE` に変わります。

アップグレード前、アップグレード中、アップグレード後の DynamoDB Streams の動作

Operation	レプリカリージョン	アップグレード前の動作	アップグレード中の動作	アップグレード後の動作
Put または Update	ソース	タイムスタンプの生成は UpdateItem を使用して行われます。	タイムスタンプの生成は PutItem を使用して行われます。	カスタマー表示タイムスタンプは生成されません。
		2 つの Streams レコードが生成されます。最初のレコードには、お客様が書き込んだ属性が含まれます。2 つ目のレコードには、`aws:rep:*` 属性が含まれます。	2 つの Streams レコードが生成されます。最初のレコードには、お客様が書き込んだ属性が含まれます。2 つ目のレコードには、`aws:rep:*` 属性が含まれます。	お客様が作成した属性を含む 1 つの Streams レコードが生成されます。
		お客様による書き込みごとに 2 つの rWCU が消費されます。	お客様による書き込みごとに 2 つの rWCU が消費されます。	お客様による書き込みごとに 1 つの rWCU が消費されます。

Operation	レプリカリージョン	アップグレード前の動作	アップグレード中の動作	アップグレード後の動作
		ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスは、CloudWatch に公開されます。
	送信先	レプリケーションは PutItem を使用して行われます。	レプリケーションは PutItem を使用して行われます。	レプリケーションは PutItem を使用して行われます。
		お客様が作成した属性と aws:rep:* 属性の両方を含む 1 つの Streams レコードが生成されます。	お客様が作成した属性と aws:rep:* 属性の両方を含む 1 つの Streams レコードが生成されます。	お客様が作成した属性のみを含みレプリケーション属性を含まない 1 つの Streams レコードが生成されます。
		アイテムが送信先のリージョンに存在する場合、1 rWCU が消費されます。アイテムが送信先のリージョンに存在しない場合、2 rWCU が消費されます。	アイテムが送信先のリージョンに存在する場合、1 rWCU が消費されます。アイテムが送信先のリージョンに存在しない場合、2 rWCU が消費されます。	お客様による書き込みごとに 1 つの rWCU が消費されます。

Operation	レプリカリージョン	アップグレード前の動作	アップグレード中の動作	アップグレード後の動作
		ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスは、CloudWatch に公開されます。
削除*	ソース	DeleteItem を使用して、より小さいタイムスタンプを持つすべてのアイテムを削除します。	DeleteItem を使用して、より小さいタイムスタンプを持つすべてのアイテムを削除します。	DeleteItem を使用して、より小さいタイムスタンプを持つすべてのアイテムを削除します。
		お客様が作成した属性と aws:rep:* 属性の両方を含む 1 つの Streams レコードが生成されます。	お客様が作成した属性と aws:rep:* 属性の両方を含む 1 つの Streams レコードが生成されます。	お客様が作成した属性を含む 1 つの Streams レコードが生成されます。
		お客様による削除ごとに 1 rWCU が消費されます。	お客様による削除ごとに 1 rWCU が消費されます。	お客様による削除ごとに 1 rWCU が消費されます。

Operation	レプリカリージョン	アップグレード前の動作	アップグレード中の動作	アップグレード後の動作
		ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスおよび PendingReplicationCount メトリクスは、CloudWatch に公開されます。	ReplicationLatency メトリクスは、CloudWatch に公開されます。
	送信先	2 つのフェーズでの削除が行われます。 • フェーズ 1 では、UpdateItem によって削除フラグが設定されます。 • フェーズ 2 では、DeleteItem によってアイテムが削除されます。	DeleteItem を使用してアイテムを削除します。	DeleteItem を使用してアイテムを削除します。

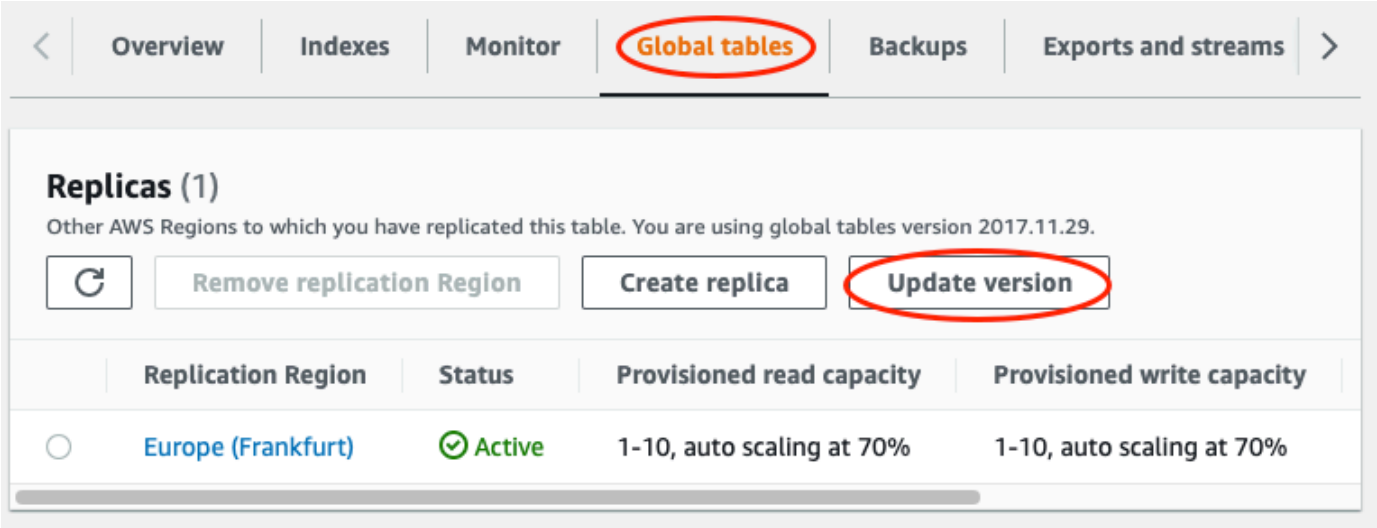
Operation	レプリカリージョン	アップグレード前の動作	アップグレード中の動作	アップグレード後の動作
		2 つの Streams レコードが生成されます。1 つ目のレコードには、`aws:rep:deleting` フィールドへの変更が含まれます。2 つ目のレコードには、お客様が作成した属性と `aws:rep:*` 属性が含まれます。	お客様が作成した属性を含む 1 つの Stream レコードが生成されます。	お客様が作成した属性を含む 1 つの Stream レコードが生成されます。
		お客様による削除ごとに 2 つの rWCU が消費されます。	お客様による削除ごとに 1 rWCU が消費されます。	お客様による削除ごとに 1 rWCU が消費されます。
		`ReplicationLatency` メトリクスおよび `PendingReplicationCount` メトリクスは、CloudWatch に公開されます。	`ReplicationLatency` メトリクスは、CloudWatch に公開されます。	`ReplicationLatency` メトリクスは、CloudWatch に公開されます。

バージョン 2019.11.21 (現行) へのアップグレード

AWS マネジメントコンソール を使用して DynamoDB グローバルテーブルのバージョンをアップグレードするには、次の手順に従います。

グローバルテーブルをバージョン 2019.11.21 (現行) にアップグレードするには

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/home) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択し、バージョン 2019.11.21 (現行) にアップグレードするグローバルテーブルを選択します。
3. [グローバルテーブル] タブを選択します。
4. [Update version (バージョンの更新)] を選択します。



5. 新しい要件を読んで同意してから、[バージョンを更新] を選択します。
6. アップグレードプロセスが完了すると、コンソールに表示されるグローバルテーブルのバージョンが 2019.11.21 に変更されます。

グローバルテーブルのベストプラクティス

以下のセクションでは、グローバルテーブルをデプロイして使用するためのベストプラクティスについて説明します。

バージョン

DynamoDB グローバルテーブルには、バージョン 2019.11.21 (現行) とバージョン 2017.11.29 (レガシー) の 2 つのバージョンがあります。可能な限り、バージョン 2019.11.21 (現行) を使用する必要があります。

削除保護

誤って削除されないように保護するグローバルテーブルレプリカの削除保護を有効にする必要があります。レプリカごとに削除保護を有効にする必要があります。

AWS CloudFormation の使用

CloudFormation は現在、スタック間のグローバルテーブルなどのマルチリージョンリソースの調整をサポートしていません。別のリージョンのスタックでグローバルテーブルの各レプリカを定義すると、レプリカの更新の実行時にスタック間で検出されたドリフトが原因でエラーが発生します。この問題を回避するには、グローバルテーブルをデプロイするためのリファレンスリージョンとして 1 つのリージョンを選択し、そのリージョンのスタックでグローバルテーブルのすべてのレプリカを定義する必要があります。

> **Important**
>
> タイプ `AWS::DynamoDB::Table` のリソースは、テンプレート内でタイプを変更してタイプ `AWS::DynamoDB::GlobalTable` のリソースに変換できません。CloudFormation リソースタイプを変更して単一リージョンテーブルをグローバルテーブルに変換しようとすると、DynamoDB テーブルが削除される可能性があります。

`AWS::DynamoDB::GlobalTable` リソースを使用して、単一のリージョンでテーブルを作成できます。このテーブルは、他の単一リージョンテーブルと同様にデプロイされます。後でスタックを更新して他のリージョンをリソースに追加すると、レプリカがテーブルに追加され、グローバルテーブルに安全に変換されます。

既存の `AWS::DynamoDB::Table` リソースを `AWS::DynamoDB::GlobalTable` リソースに変換する場合、リソースタイプを変換するための推奨ステップは次のとおりです。

1. `AWS::DynamoDB::Table` 削除ポリシーを保持するように設定します。
2. テーブルをスタック定義から削除します。
3. AWS コンソールの単一リージョンテーブルにレプリカを追加し、グローバルテーブルに変換します。
4. 新しいグローバルテーブルを新しい `AWS::DynamoDB::GlobalTable` リソースとしてスタックにインポートします。

> **Note**
>
> 新しいリージョンのレプリカをグローバルテーブルに追加すると、CloudFormation テンプレートで指定された容量値に関係なく、自動スケーリングの最大容量を含むソースリージョンのプロビジョニングされた容量設定が、新しいレプリカの作成時に適用されます。ソースリージョンの最大容量がターゲットリージョンのテーブルレベルのスループットクォータを超える場合、レプリカの作成は失敗し、`insufficient TableMaxReadCapacityUnits limits` のようなエラーが発生します。レプリカを作成したら、読み取り容量設定を個別に調整できます。この調整は他のリージョンには転送されません。グローバルテーブルのテーブルレベルでのスループットクォータの詳細については、「the section called “グローバルテーブル”」を参照してください。

バックアップとポイントインタイムリカバリ

グローバルテーブル内の 1 つのレプリカの自動バックアップとポイントインタイムリカバリ (PITR) を有効にすることで、ディザスタリカバリの目標を達成できる場合があります。AWS-Backup で作成されたレプリカバックアップは、リージョン間で自動的にレプリケートできるため、耐障害性が向上します。バックアップと PITR 有効化戦略を選択する際は、マルチリージョンの高可用性の観点からディザスタリカバリ計画の目標を検討してください。

マルチリージョンの高可用性のための設計

グローバルテーブルのデプロイに関する規範的ガイダンスについては、「DynamoDB グローバルテーブル設計のベストプラクティス」を参照してください。

DynamoDB での項目と属性の操作

Amazon DynamoDB では、項目は属性の集まりです。各属性には名前と値があります。属性値はスカラー型、セット型、ドキュメント型のいずれかです。詳細については、「Amazon DynamoDB: 仕組み」を参照してください。

DynamoDB では、作成、読み込み、更新、および削除 (CRUD) の 4 つの基本的な操作機能を使用できます。これらの操作はすべてアトミックです。

- `PutItem` — 項目を作成します。
- `GetItem` — 項目を読み込みます。
- `UpdateItem` — 項目を更新します。

- `DeleteItem` — 項目を削除します。

これらの各オペレーションでは、作業対象の項目のプライマリキーを指定する必要があります。たとえば、`GetItem` を使用して項目を読み込むには、その項目のパーティションキーとソートキー (該当する場合) を指定する必要があります。

4 つの基本的な CRUD オペレーションに加えて、DynamoDB は以下も提供します。

- `BatchGetItem` — 1 つ以上のテーブルから最大 100 個の項目を読み込みます。
- `BatchWriteItem` — 1 つ以上のテーブルから最大 25 個の項目を作成または削除します。

これらのバッチ操作は、複数の CRUD オペレーションを単一のリクエストにまとめます。さらに、応答のレイテンシーを最小限に抑えるため、バッチオペレーションは項目を並列で読み書きします。

このセクションには、これらのオペレーションを使用する方法の説明、および、条件付き更新やアトミックカウンターなどの関連するトピックが含まれています。このセクションには、AWS SDK を使用するサンプルコードも含まれています。

トピック

- DynamoDB 項目のサイズと形式
- 項目の読み込み
- 項目を書き込みます
- 戻り値
- バッチオペレーション
- アトミックカウンタ
- 条件付きの書き込み
- DynamoDB での式の使用
- DynamoDB での Time to Live (TTL) の使用
- DynamoDB のテーブルに対するクエリの実行
- DynamoDB でのテーブルのスキャン
- PartiQL: Amazon DynamoDB 用の SQL 互換クエリ言語
- 項目の操作: Java
- 項目の操作: .NET

DynamoDB 項目のサイズと形式

DynamoDB テーブルはプライマリキーを除いてスキーマレスです。そのため、テーブルの項目の属性、サイズ、データ型はすべて異なる場合があります。

項目の合計サイズは、属性名と属性値の文字列の長さの合計、および以下に説明するように該当するオーバーヘッドが追加されます。次のガイドラインを使用して属性サイズを予測することができます。

- 文字列は、UTF-8 バイナリエンコードの Unicode です。文字列のサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (UTF-8 でエンコードされたバイト数) です。
- 数値は、有効桁数が最大 38 の可変長です。先頭と末尾の 0 は切り捨てられます。数値のおおよそのサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (有効桁数 2 あたり 1 バイト) + (1 バイト) です。
- バイナリ値を DynamoDB に送信するには base64 形式でエンコードする必要がありますが、サイズの計算には値の実際のバイト長が使用されます。バイナリ属性のサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (raw バイト数) です。
- null 属性または Boolean 属性のサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (1 バイト) です。
- `List` 型または `Map` 型の属性は、その内容にかかわらず、余分な 3 バイトが必要です。`List` または `Map` のサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (入れ子要素のサイズの合計) + (3 バイト) です。空の `List` または `Map` のサイズは、(属性名の UTF-8 でエンコードされたバイト数) + (3 バイト) です。
- `List` または `Map` の各要素には、余分な 1 バイトが必要です。
- セット (文字列セット、数値セット、またはバイナリセット) では、要素のサイズを超える追加のオーバーヘッドは必要ありません。セットのサイズは、(属性名の UTF-8 でエンコードされたバイトの数) + 合計 (各要素の値のサイズ) です。各要素値は、対応するスカラータイプと同じルールを使用して、そのタイプに応じてサイズが設定されますが、属性名は含まれません。文字列要素は (UTF-8 でエンコードされたバイト数)、数値要素は (有効桁数 2 あたり 1 バイト) + (1 バイト)、バイナリ要素は (raw バイト数) です。

> **Note**
>
> 属性名は長いものよりも短いものにすることをお勧めします。これにより、必要なストレージの量を減らすことができ、使用する RCU/WCU の量を減らすこともできます。

ストレージの請求において、各項目には、有効にした機能に応じて、項目あたりのストレージオーバーヘッドが含まれます。

- DynamoDB のすべての項目は、項目ごとに 100 バイトのストレージオーバーヘッドを必要とします。
- 一部の DynamoDB 機能 (グローバルテーブル、トランザクション、DynamoDB を使用した Kinesis Data Streams の変更データキャプチャ) では、これらの機能を有効にすることでシステムが作成した属性を考慮するために、追加のストレージオーバーヘッドが必要になります。例えば、グローバルテーブルでは、48 バイトのストレージオーバーヘッドがさらに必要になります。

項目の読み込み

DynamoDB テーブルから項目を読み込むには、`GetItem` オペレーションを使用します。必要な項目のプライマリキーと共にテーブルの名前を指定する必要があります。

Example

次の AWS CLI の例は、`ProductCatalog` テーブルから項目を読み込む方法を示しています。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}'
```

> **Note**
>
> `GetItem` では、プライマリキーの一部だけではなく全体を指定する必要があります。たとえば、テーブルに複合プライマリキー (パーティションキーおよびソートキー) がある場合、パーティションキーおよびソートキーの値を指定する必要があります。

デフォルトでは、`GetItem` リクエストは結果的に整合性のある読み込みを行います。代わりに、`ConsistentRead` パラメータを使用して、強い整合性のある読み込みをリクエストできます。(これは、追加の読み込みキャパシティーユニットを消費しますが、項目の最新バージョンを返します)。

`GetItem` は、項目のすべての属性を返します。一部の属性のみが返されるように、プロジェクション式を使用できます。詳細については、「DynamoDB でのプロジェクション式の使用」を参照してください。

GetItem で消費される読み込みキャパシティーユニットの数を返すには、ReturnConsumedCapacity パラメータを TOTAL に設定します。

Example

次の AWS Command Line Interface (AWS CLI) の例は、オプションの GetItem パラメータを示しています。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --consistent-read \
    --projection-expression "Description, Price, RelatedItems" \
    --return-consumed-capacity TOTAL
```

項目を書き込みます

DynamoDB テーブルの項目を作成、更新、または削除するには、次のオペレーションのいずれかを使用します。

- PutItem
- UpdateItem
- DeleteItem

これらの各オペレーションで、プライマリキーの一部の属性ではなくすべての属性を指定する必要があります。たとえば、テーブルに複合プライマリキー (パーティションキーおよびソートキー) がある場合、パーティションキーおよびソートキーの値を提供する必要があります。

これらのオペレーションのいずれかで消費される書き込みキャパシティーユニットの数を返すには、ReturnConsumedCapacity パラメータを次のいずれかに設定します。

- TOTAL — 消費された書き込み容量単位の総数を返します。
- INDEXES — 消費された書き込み容量単位の総数とともに、テーブルおよびオペレーションに影響を受けたセカンダリインデックスの小計を返します。
- NONE — 書き込み容量の詳細は返されません。(これがデフォルトです)

PutItem

PutItem は新しい項目を作成します。同じキーを持つ項目がテーブルにすでに存在する場合は、新しい項目に置き換えられます。

Example

Thread テーブルに新しい項目を書き込みます。Thread のプライマリキーは、ForumName (パーティションキー) と Subject (ソートキー) で構成されます。

```
aws dynamodb put-item \
    --table-name Thread \
    --item file://item.json
```

--item の引数は、ファイル item.json に保存されます。

```
{
    "ForumName": {"S": "Amazon DynamoDB"},
    "Subject": {"S": "New discussion thread"},
    "Message": {"S": "First post in this thread"},
    "LastPostedBy": {"S": "fred@example.com"},
    "LastPostDateTime": {"S": "201603190422"}
}
```

UpdateItem

指定されたキーを持つ項目が存在しない場合は、UpdateItem により新しい項目が作成されます。または、既存の項目の属性が変更されます。

更新式を使用して、変更する属性と新しい値を指定します。詳細については、「DynamoDB での更新式の使用」を参照してください。

更新式では、実際の値にプレースホルダーとして式の属性値を使用します。詳細については、「DynamoDB での式属性値の使用」を参照してください。

Example

Thread 項目のさまざまな属性を変更します。オプションの ReturnValues パラメータは、更新後に表示されるように項目を表示します。詳細については、「戻り値」を参照してください。

```
aws dynamodb update-item \
    --table-name Thread \
```

```
    --key file://key.json \
    --update-expression "SET Answered = :zero, Replies = :zero, LastPostedBy
 = :lastpostedby" \
    --expression-attribute-values file://expression-attribute-values.json \
    --return-values ALL_NEW
```

--key の引数は、ファイル key.json に保存されます。

```
{
    "ForumName": {"S": "Amazon DynamoDB"},
    "Subject": {"S": "New discussion thread"}
}
```

--expression-attribute-values の引数は、ファイル expression-attribute-values.json に保存されます。

```
{
    ":zero": {"N":"0"},
    ":lastpostedby": {"S":"barney@example.com"}
}
```

DeleteItem

DeleteItem は指定されたキーを持つ項目を削除します。

Example

次の AWS CLI の例で、Thread 項目を削除する方法について説明します。

```
aws dynamodb delete-item \
    --table-name Thread \
    --key file://key.json
```

戻り値

場合によっては、ある属性値を変更する前後にその属性値が表示されるように DynamoDB で返すこともできます。PutItem、UpdateItem、および DeleteItem オペレーションには、ReturnValues パラメータがあり、これを使用することで、属性の変更前後に、その属性値を返すことができます。

ReturnValues のデフォルトの値は NONE で、DynamoDB は変更された属性の情報を返しません。

以下は、DynamoDB API オペレーションごとに整理された ReturnValues のその他の有効な設定です。

PutItem

- ReturnValues: ALL_OLD
  - 既存の項目に上書きする場合、ALL_OLD は上書きの前に表示されたように、項目全体を返します。
  - 存在しない項目を書き込んだ場合は、ALL_OLD による影響はありません。

UpdateItem

UpdateItem の最も一般的な使用方法は、既存の項目の更新です。ただし、UpdateItem は実際にはアップサートを実行します。つまり、項目が存在しない場合は、自動的に作成します。

- ReturnValues: ALL_OLD
  - 既存の項目を更新する場合、ALL_OLD は更新前に表示されたように、項目全体を返します。
  - 存在しない項目を更新 (アップサート) すると、ALL_OLD による影響はありません。
- ReturnValues: ALL_NEW
  - 既存の項目を更新する場合、ALL_NEW は更新後に表示されるように、項目全体が返されます。
  - 存在しない項目を更新 (アップサート) すると、ALL_NEW は項目全体を返します。
- ReturnValues: UPDATED_OLD
  - 既存の項目を更新した場合、UPDATED_OLD は、更新前に表示されたように、更新された属性だけを返します。
  - 存在しない項目を更新 (アップサート) すると、UPDATED_OLD による影響はありません。
- ReturnValues: UPDATED_NEW
  - 既存の項目を更新した場合、UPDATED_NEW は、更新後に表示されるように、影響のある属性だけを返します。
  - 存在しない項目を更新 (アップサート) する場合、UPDATED_NEW は更新後に表示される、更新された属性だけを返します。

DeleteItem

- ReturnValues: ALL_OLD

- 既存の項目を削除すると、ALL_OLD は削除の前に表示されたように、項目全体を返します。
- 存在しない項目を削除すると、ALL_OLD はデータを返しません。

バッチオペレーション

複数の項目の読み込みや書き込みを必要とするアプリケーションのために、DynamoDB は BatchGetItem および BatchWriteItem オペレーションを提供します。これらのオペレーションを使用すると、アプリケーションから DynamoDB へのネットワークラウンドトリップの数を減らすことができます。さらに、DynamoDB は個別の読み込みまたは書き込みオペレーションを並行して実行します。同時実行またはスレッディングの管理をする必要がないので、アプリケーションにはこの並列処理が役立ちます。

バッチオペレーションは、基本的に複数の読み込みまたは書き込みリクエストをまとめます。たとえば、BatchGetItem リクエストに 5 つの項目が含まれている場合、DynamoDB によって 5 回の GetItem オペレーションが実行されます。同様に、BatchWriteItem リクエストに 2 つの PUT リクエストと 4 つの DELETE リクエストが含まれている場合、DynamoDB によって 2 つの PutItem リクエストと 4 つの DeleteItem リクエストが実行されます。

一般的に、バッチのすべてのリクエストが失敗しない限り、バッチオペレーションは失敗しません。たとえば、BatchGetItem オペレーションを実行する際、バッチの個々の GetItem リクエストが失敗したとします。この場合、BatchGetItem は失敗した GetItem リクエストからキーとデータを返します。バッチのその他の GetItem リクエストは影響を受けません。

BatchGetItem

1 回の BatchGetItem オペレーションには、最大 100 の個々の GetItem リクエストが含まれていて、最大 16 MB のデータを取得できます。さらに、BatchGetItem オペレーションで、複数のテーブルから項目を取得できます。

Example

一部の属性のみが返されるようにプロジェクション式を使用して Thread テーブルから 2 つの項目を取得します。

```
aws dynamodb batch-get-item \
    --request-items file://request-items.json
```

--request-items の引数は、ファイル request-items.json に保存されます。

```
{
    "Thread": {
        "Keys": [
            {
                "ForumName":{"S": "Amazon DynamoDB"},
                "Subject":{"S": "DynamoDB Thread 1"}
            },
            {
                "ForumName":{"S": "Amazon S3"},
                "Subject":{"S": "S3 Thread 1"}
            }
        ],
        "ProjectionExpression":"ForumName, Subject, LastPostedDateTime, Replies"
    }
}
```

BatchWriteItem

BatchWriteItem オペレーションには最大 25 の個々の PutItem リクエストと DeleteItem リクエストを含むことができ、最大 16 MB のデータを書き込めます。(個々の項目の最大サイズは 400 KB です。) さらに、BatchWriteItem オペレーションで、複数のテーブルの項目を入力したり削除したりできます。

> **Note**
>
> BatchWriteItem では UpdateItem リクエストはサポートされません。

Example

ProductCatalog テーブルに 2 つの項目を書き込みます。

```
aws dynamodb batch-write-item \
    --request-items file://request-items.json
```

--request-items の引数は、ファイル request-items.json に保存されます。

```
{
    "ProductCatalog": [
        {
```

```
            "PutRequest": {
                "Item": {
                    "Id": { "N": "601" },
                    "Description": { "S": "Snowboard" },
                    "QuantityOnHand": { "N": "5" },
                    "Price": { "N": "100" }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": { "N": "602" },
                    "Description": { "S": "Snow shovel" }
                }
            }
        }
    ]
}
```

アトミックカウンタ

この UpdateItem オペレーションを使用して、アトミックカウンター (他の書き込みリクエストに干渉することなく無条件に増分される数値属性) を実装できます。(すべての書き込みリクエストは、受信された順に適用されます) 。アトミックカウンターでは、更新はべき等ではありません。つまり、UpdateItem を呼び出すたびに数値はインクリメントまたはデクリメントされます。アトミックカウンタの更新に使用されるインクリメント値が正の場合、オーバーカウントが発生する可能性があります。インクリメント値が負の場合、アンダーカウントの原因となります。

ウェブサイトの訪問者数を追跡するためにアトミックカウンターを使用できます。この場合、アプリケーションでは、現在値に関係なく、数値はインクリメントされます。UpdateItem オペレーションが失敗した場合、アプリケーションはオペレーションを再試行します。これには、カウンターを 2 度更新する恐れがありますが、ウェブサイトの訪問者数のカウントに多少の誤差があっても許容できるでしょう。

アトミックカウンターはカウントの誤差が許容されない場合にはふさわしくありません (銀行業務用のアプリケーションなど)。この場合は、アトミックカウンターの代わりに条件付き更新を使用する方が安全です。

詳細については、「数値属性の増減」を参照してください。

Example

次の AWS CLI の例では、商品の Price が 5 でインクリメントされます。この例では、カウンタが更新される前に項目が存在することがわかっていました。(UpdateItem は冪等性ではないので、Price はこのコードを実行するたびに増えていきます)。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id": { "N": "601" }}' \
    --update-expression "SET Price = Price + :incr" \
    --expression-attribute-values '{":incr":{"N":"5"}}' \
    --return-values UPDATED_NEW
```

条件付きの書き込み

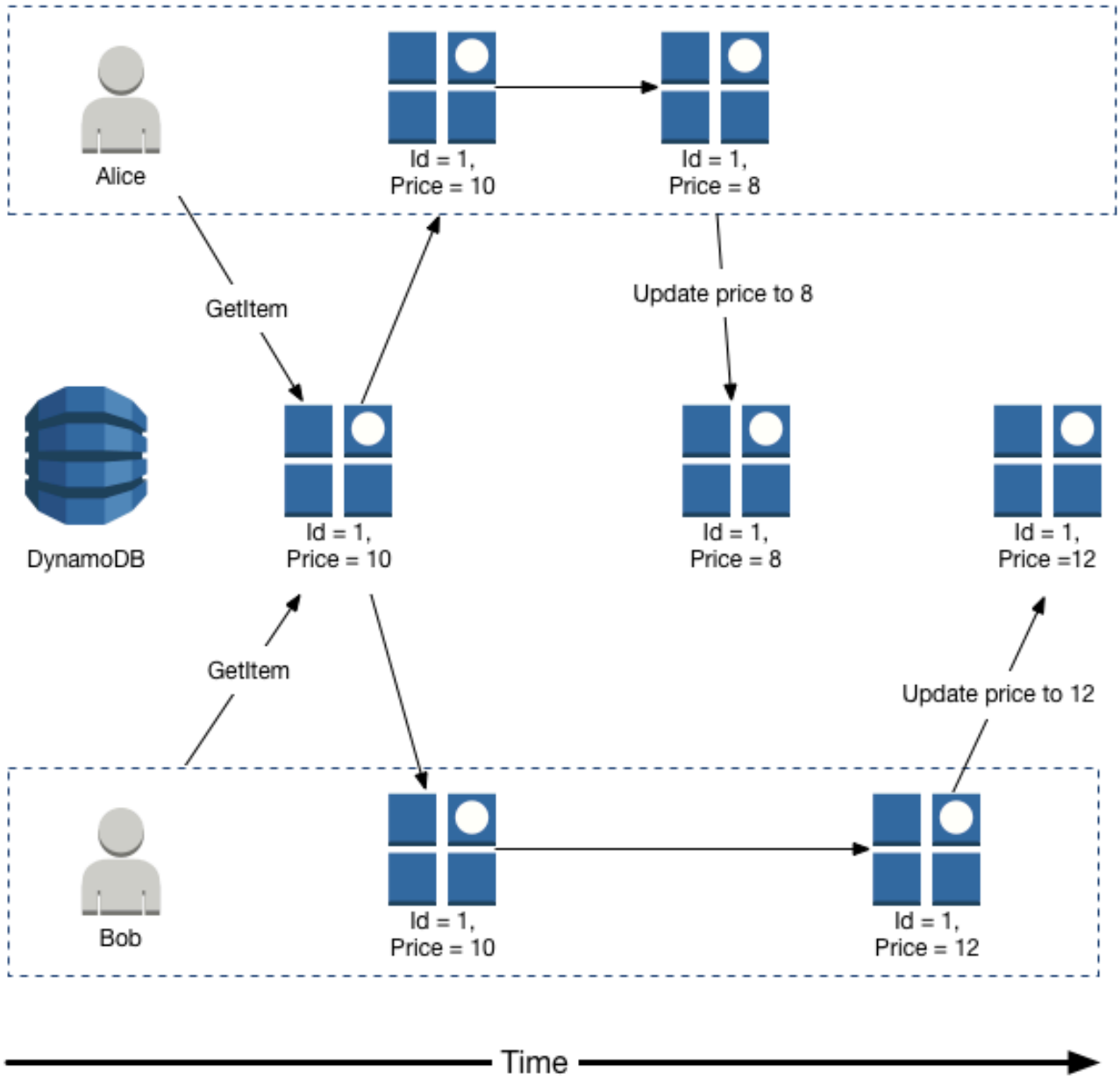
デフォルトでは、DynamoDB 書き込みオペレーション (PutItem、DeleteItem) は無条件です。つまり、これらの各オペレーションでは、指定されたプライマリキーを持つ既存の項目が上書きされます。

DynamoDB はオプションでこれらのオペレーションの条件付き書き込みをサポートしています。条件付き書き込みが成功するのは、項目の属性が 1 つ以上の想定条件を満たす場合のみです。それ以外の場合は、エラーが返されます。

条件付き書き込みでは、その条件について項目の最新更新バージョンと照合します。なお、項目が以前に存在しなかった場合や、その項目に対して最後に成功した操作が削除であった場合、条件付き書き込みでは以前の項目は検出されません。

条件付き書き込みは多くの状況で役立ちます。たとえば、同じプライマリキーを持つ既存の項目がない場合にのみ、PutItem オペレーションが成功するようにできます。または、属性の 1 つに特定の値がある場合に UpdateItem オペレーションが項目を変更することを防ぐことができます。

条件付き書き込みは、複数のユーザーが同じ項目を変更しようとする場合に役立ちます。2 人のユーザー (Alice と Bob) が DynamoDB テーブルから同じ項目を処理している以下の図を考慮します。



Alice が AWS CLI を使用して `Price` 属性を 8 に更新するとします。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --update-expression "SET Price = :newval" \
```

```
    --expression-attribute-values file://expression-attribute-values.json
```

--expression-attribute-values の引数は、ファイル expression-attribute-values.json に保存されます。

```
{
    ":newval":{"N":"8"}
}
```

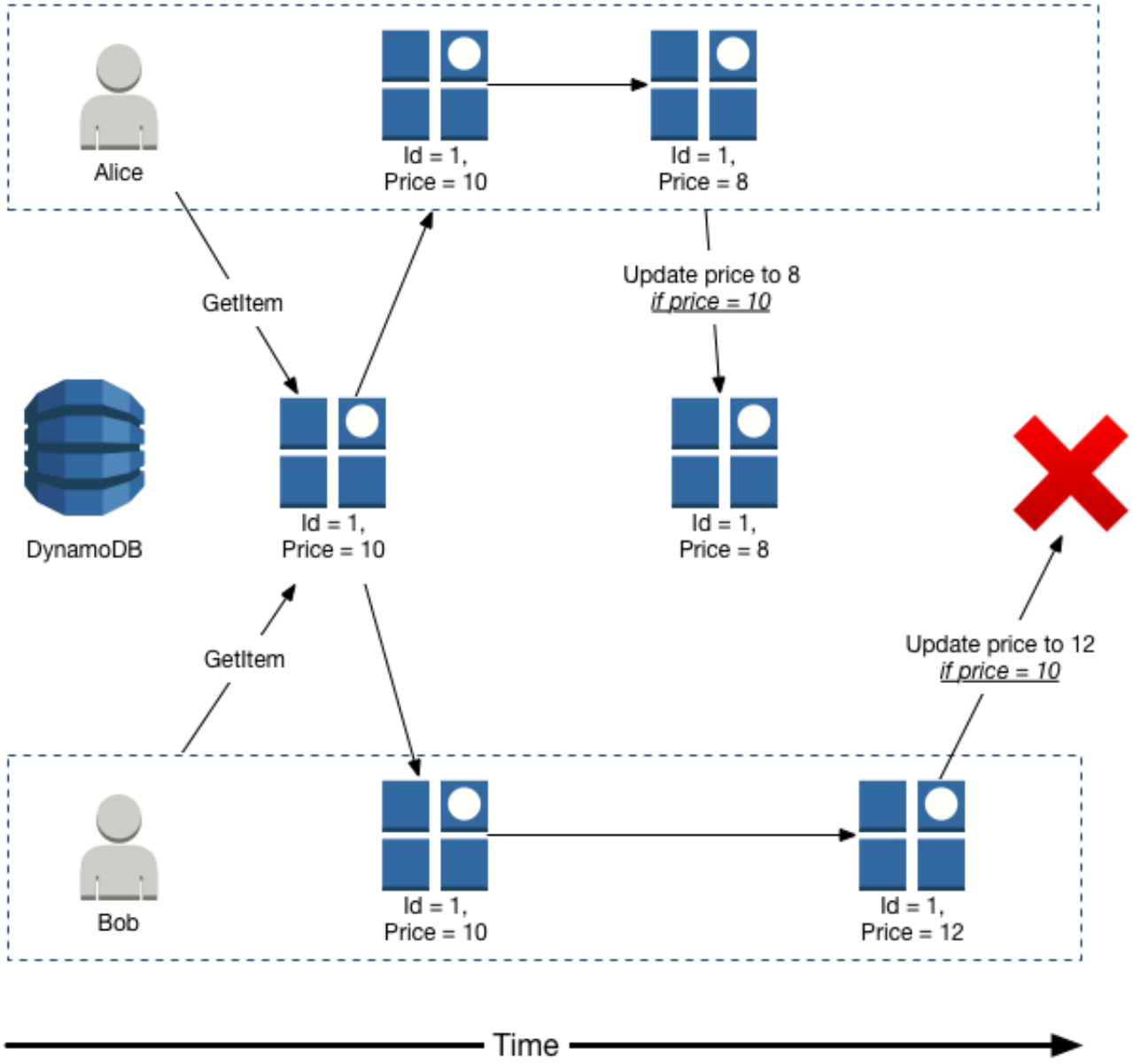
次に、Bob が後で同様の UpdateItem リクエストを発行しますが、Price を 12 に変更します。Bob には、--expression-attribute-values パラメータは次のように見えます。

```
{
    ":newval":{"N":"12"}
}
```

Bob のリクエストは成功しますが、その前の Alice の更新は失われます。

条件付き PutItem、DeleteItem、または UpdateItem をリクエストするには、条件式を指定します。条件式は、属性名、条件付き演算子および組み込み関数を含む文字列です。式全体の評価が true になる必要があります。それ以外の場合は、このオペレーションは失敗します。

次は、条件付き書き込みにより Alice の更新が上書きされるのを防ぐ方法について、以下の図を考慮します。



Alice はまず `Price` を 8 に更新を試行しますが、現在の `Price` が 10 である場合にのみ、という条件でそうします。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"1"}}' \
    --update-expression "SET Price = :newval" \
```

```
    --condition-expression "Price = :currval" \
    --expression-attribute-values file://expression-attribute-values.json
```

`--expression-attribute-values` の引数は、ファイル `expression-attribute-values.json` に保存されます。

```
{
    ":newval":{"N":"8"},
    ":currval":{"N":"10"}
}
```

条件が true に評価されるため、Alice の更新は成功します。

次に、Bob は、`Price` を 12 に 更新しますが、現在の `Price` が 10 である場合にのみ、という条件でそうします。Bob には、`--expression-attribute-values` パラメータは次のように見えます。

```
{
    ":newval":{"N":"12"},
    ":currval":{"N":"10"}
}
```

Alice がすでに `Price` を 8 に変更していたので、条件式は false 評価され Bob の更新は失敗します。

詳細については、「DynamoDB 条件式 CLI の例」を参照してください。

条件付き書き込みの冪等性

条件付き書き込みは、条件チェックが更新される同じ属性で行われる場合に、べき等性が保たれます。つまり、リクエスト時に項目の特定の属性値が想定されたものである場合にのみ、DynamoDB によって指定された書き込みリクエストが実行されます。

たとえば、`UpdateItem` リクエストを発行し、項目の `Price` を、現在の `Price` が 20 である場合のみ、3 増加させるとします。リクエストを送信した後、その結果を得る前にネットワークエラーが発生したため、リクエストが成功したかどうか不明です。この条件付き書き込みはべき等のオペレーションであるため、同じ `UpdateItem` リクエストを再試行できます。すると、DynamoDB は、現在の `Price` が 20 である場合のみ項目を更新します。

失敗した条件付き書き込みの項目属性を返す

条件付き PutItem、UpdateItem、または DeleteItem オペレーションで条件が満たされない場合は、ConditionalCheckFailedException が返されます。デフォルトでは、この例外には、条件で false と評価される原因となった項目の属性は含まれません。

DynamoDB がこれらの属性を返すようにするには、ReturnValuesOnConditionCheckFailure パラメータを ALL_OLD に設定します。条件が失敗すると、ConditionalCheckFailedException には既存の項目の属性値が含まれます。これにより、個別の読み取りリクエストを発行することなく、項目の現在の状態を調べることができます。例えば、書き込みが失敗する原因となった値を特定できます。デフォルト値は NONE です。

条件付き書き込みで消費されるキャパシティユニット

条件付き書き込み中に ConditionExpression が false と評価された場合でも、DynamoDB はテーブルの書き込みキャパシティを消費します。消費量は、既存の項目のサイズ (または最低 1) によって異なります。例えば、既存の項目が 300 KB で、作成または更新しようとしている新しい項目が 310 KB の場合、消費される書き込みキャパシティーユニットは、条件が満たされないかったら 300、満たされたら 310 になります。これが新しい項目 (既存の項目なし) の場合、消費される書き込みキャパシティーユニットは条件が満たされれば 1、条件が満たされなかったら 310 になります。

> **Note**
>
> 書き込みオペレーションでは、書き込みキャパシティーユニットのみが消費されます。読み込みキャパシティーユニットは消費されません。

失敗した条件付き書き込みは ConditionalCheckFailedException を返します。これが起きると、消費された書き込みキャパシティに関する情報はレスポンスで返されません。

条件付き書き込みの際に消費された書き込みキャパシティーユニットの数を返すには、ReturnConsumedCapacity パラメータを使用します。

- TOTAL — 消費された書き込み容量単位の総数を返します。
- INDEXES — 消費された書き込み容量単位の総数とともに、テーブルおよびオペレーションに影響を受けたセカンダリインデックスの小計を返します。
- NONE — 書き込み容量の詳細は返されません。(これがデフォルトです)

> **Note**
>
> グローバルセカンダリインデックスとは異なり、ローカルセカンダリインデックスは、プロビジョンドスループット性能をそのテーブルと共有します。ローカルセカンダリインデックスでの読み込みと書き込みのアクティビティは、テーブルからプロビジョンドスループット性能を消費します。

DynamoDB での式の使用

Amazon DynamoDB では、式を使用して、項目から読み取る属性の指定、条件が満たされたときのデータの書き込み、項目の更新方法の指定、クエリの定義、クエリ結果のフィルタリングを行うことができます。

このテーブルでは、式の基本的な文法と利用可能な式の種類について説明します。

式タイプ	説明
プロジェクション式	プロジェクション式は、GetItem、Query、Scan などのオペレーションを使用するときに項目から取得する属性を特定します。
条件式	条件式は、PutItem、UpdateItem、DeleteItem の各オペレーションを使用したときに、どの項目を変更すべきかを決定します。
更新式	更新式は、UpdateItem で項目の属性を変更する方法を指定します。例えば、スカラー値を設定したり、リストまたはマップから要素を削除したりします。
キー条件式	キー条件式は、クエリがテーブルやインデックスから読み取る項目を決定します。
フィルター式	フィルター式は、クエリ結果のうち、どの項目を返すべきかを決定します。他のすべての結果は破棄されます。

式の構文と各タイプの式の詳細については、以下のセクションを参照してください。

トピック

- DynamoDB で式を使用する場合の項目属性の参照
- DynamoDB の式の属性名 (エイリアス)
- DynamoDB での式属性値の使用
- DynamoDB でのプロジェクション式の使用
- DynamoDB での更新式の使用
- DynamoDB の条件式とフィルター式、演算子、関数
- DynamoDB 条件式 CLI の例

> **Note**
>
> 下位互換性のために、DynamoDB は式を使用しない条件パラメータもサポートします。詳細については、「レガシー DynamoDB 条件パラメータ」を参照してください。
> 新しいアプリケーションでは、レガシーパラメータではなく式を使用する必要があります。

DynamoDB で式を使用する場合の項目属性の参照

このセクションでは、Amazon DynamoDB で式の項目属性を参照する方法を説明します。複数のリストやマップ内で深くネストされている場合でも、属性を使用できます。

トピック

- 最上位属性
- 入れ子の属性
- ドキュメントパス

サンプル項目: ProductCatalog

このページの例では、`ProductCatalog` テーブルで次のサンプル項目を使用しています (このテーブルについては、「DynamoDB で使用するテーブルとデータの例」で説明されています)。

```
{
```

```
    "Id": 123,
    "Title": "Bicycle 123",
    "Description": "123 description",
    "BicycleType": "Hybrid",
    "Brand": "Brand-Company C",
    "Price": 500,
    "Color": ["Red", "Black"],
    "ProductCategory": "Bicycle",
    "InStock": true,
    "QuantityOnHand": null,
    "RelatedItems": [
        341,
        472,
        649
    ],
    "Pictures": {
        "FrontView": "http://example.com/products/123_front.jpg",
        "RearView": "http://example.com/products/123_rear.jpg",
        "SideView": "http://example.com/products/123_left_side.jpg"
    },
    "ProductReviews": {
     "FiveStar": [
       "Excellent! Can't recommend it highly enough! Buy it!",
       "Do yourself a favor and buy this."
     ],
     "OneStar": [
       "Terrible product! Do not buy this."
     ]
    },
    "Comment": "This product sells out quickly during the summer",
    "Safety.Warning": "Always wear a helmet"
}
```

次の点に注意してください。

- パーティションキー値 (Id) は 123 です。ソートキーはありません。
- ほとんどの属性に、String、Number、Boolean、Null などのスカラーデータ型があります。
- 1 つの属性 (Color) は String Set です。
- 次の属性はドキュメントデータ型です。
  - RelatedItems のリスト。各要素は関連製品の Id です。
  - Pictures のマップ。各要素は対応するイメージファイルの URL と、写真の短い説明です。

- ProductReviews のマップ。各要素は、レーティングと、そのレーティングに対応する評価のリストを表します。最初に、このマップには 5 つ星と 1 つ星の評価が入力されます。

最上位属性

属性が別の属性に組み込まれていない場合、最上位属性と呼ばれます。ProductCatalog 項目の最上位属性は次のようになります。

- Id
- Title
- Description
- BicycleType
- Brand
- Price
- Color
- ProductCategory
- InStock
- QuantityOnHand
- RelatedItems
- Pictures
- ProductReviews
- Comment
- Safety.Warning

Color (リスト)、RelatedItems (リスト)、Pictures (マップ)、および ProductReviews (マップ) を除くすべての最上位属性はスカラーです。

入れ子の属性

属性が別の属性に組み込まれている場合、入れ子の属性と呼ばれます。入れ子の属性にアクセスするには、間接参照演算子を使用します。

- [n] — リストの要素

- . (ドット) — マップの要素

リスト要素へのアクセス

リスト要素の間接参照演算子は [n] で、n は要素数です。リストの要素はゼロベースであるため、[0] はリスト内の最初の要素、[1] は 2 番目の要素、という順番で表されます。次に例を示します。

- `MyList[0]`
- `AnotherList[12]`
- `ThisList[5][11]`

要素 `ThisList[5]` は、それ自体がネストされたリストです。したがって、`ThisList[5][11]` は、そのリストの 12 番目の要素を参照します。

角括弧内の数は、負以外の整数である必要があります。そのため、次の式は無効です。

- `MyList[-1]`
- `MyList[0.4]`

マップ要素へのアクセス

マップ要素の間接参照演算子は . (ドット) です。マップの要素間の区切り文字として、ドットを使用します。

- `MyMap.nestedField`
- `MyMap.nestedField.deeplyNestedField`

ドキュメントパス

式では、ドキュメントパスを使用して、属性の場所を DynamoDB に伝えます。最上位属性の場合、ドキュメントパスは単純に属性名になります。ネストされた属性の場合は、間接参照演算子を使用してドキュメントパスを構築します。

ドキュメントパスのいくつかの例を次に示します。(DynamoDB で式を使用する場合の項目属性の参照 に示された項目を参照してください)

- 最上位のスカラー属性。

`Description`

- 最上位のリスト属性。(これはいくつかの要素だけではなく、すべてを示すリストを返します)

`RelatedItems`

- `RelatedItems` リストの 3 番目の要素 (このリスト要素はゼロベースであることに注意してください)。

`RelatedItems[2]`

- 製品の正面の写真。

`Pictures.FrontView`

- すべての 5 つ星の評価。

`ProductReviews.FiveStar`

- 最初の 5 つ星の評価。

`ProductReviews.FiveStar[0]`

> **Note**
>
> ドキュメントパスの最大深度は 32 です。したがって、任意のパスの間接参照演算子の数はこの制限を超えることはできません。

次の要件を満たしている限り、ドキュメントパスには任意の属性名を使用できます。

- 最初の文字は `a-z`、`A-Z`、または `0-9`
- 2 番目の文字 (存在する場合) は `a-z`、`A-Z`

> **Note**
>
> 属性名がこの要件を満たさない場合は、式の属性名をプレースホルダーとして定義する必要があります。

詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

DynamoDB の式の属性名 (エイリアス)

式の属性名は、実際の属性名の代わりとして Amazon DynamoDB の式で使用するエイリアス (またはプレースホルダー) です。式の属性名は、シャープ記号 (#) で始める必要があり、1 つ以上の英数字が続きます。アンダースコア (_) 文字も使用できます。

このセクションでは、式の属性名を使用する必要があるいくつかの状況について説明します。

> **Note**
>
> このセクションの例では AWS Command Line Interface (AWS CLI) を使用します。

トピック

- 予約語
- 特殊文字を含む属性名
- 入れ子の属性
- 属性名を繰り返し参照する

予約語

DynamoDB 予約語と競合する属性名を含む式を書く必要が生じることもあります。予約語の一覧については、「DynamoDB の予約語」を参照してください。

たとえば、AWS CLI は予約語であるため、次の `COMMENT` の例は失敗します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "Comment"
```

この問題に対処するには、`Comment` を、`#c` などの式の属性名で置き換えることができます。`#` (シャープ記号) は必須であり、これが属性名のプレースホルダーであることを示します。AWS CLI の例は次のようになります。

```
aws dynamodb get-item \
     --table-name ProductCatalog \
```

```
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#c" \
    --expression-attribute-names '{"#c":"Comment"}'
```

> **Note**
>
> 属性名が数値で始まるか、スペースまたは予約語を含む場合、式の属性名を使用して式のその属性名を置き換える必要があります。

特殊文字を含む属性名

式では、ドット (".") はドキュメントパスの区切り文字として解釈されます。ただし、DynamoDB では属性名の一部としてドット文字や、ハイフン (「-」) などの特殊文字を使用することもできます。これは、あいまいな意味を持つことがあります。例として、ProductCatalog 項目から Safety.Warning 属性を取得するとします (DynamoDB で式を使用する場合の項目属性の参照 を参照してください)。

プロジェクション式を使用して、Safety.Warning にアクセスするとします。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "Safety.Warning"
```

DynamoDB は、予想される文字列 ("Always wear a helmet") ではなく空の結果を返します。これは、DynamoDB が式のドットをドキュメントパスの区切り文字として解釈するためです。この場合、#sw の置換として式の属性名 (Safety.Warning など) を定義する必要があります。その後、次のプロジェクション式を使用します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#sw" \
    --expression-attribute-names '{"#sw":"Safety.Warning"}'
```

次に、DynamoDB が正しい結果を返します。

> **Note**
>
> 属性名にドット (「.」) またはハイフン (「-」) が含まれている場合は、式の属性名を使用して、式でその属性名を置き換える必要があります。

入れ子の属性

ネストされた属性 ProductReviews.OneStar にアクセスするとします。式の属性名において、DynamoDB はドット (".") を属性名内の文字として扱います。ネストされた属性を参照するには、ドキュメントパス内の要素ごとに式の属性名を定義します。

- #pr – ProductReviews
- #1star – OneStar

その後、プロジェクション式として #pr.#1star を使用します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#pr.#1star"  \
    --expression-attribute-names '{"#pr":"ProductReviews", "#1star":"OneStar"}'
```

次に、DynamoDB が正しい結果を返します。

属性名を繰り返し参照する

式の属性名は、同じ属性名を繰り返し参照する必要がある場合に役立ちます。たとえば、ProductCatalog 項目からいくつかの評価を取得する次の式を考えてみます。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "ProductReviews.FiveStar, ProductReviews.ThreeStar,
 ProductReviews.OneStar"
```

これをより簡潔にするために、ProductReviews を、#pr などの式の属性名で置き換えることができます。変更された式は次のようになります。

- #pr.FiveStar, #pr.ThreeStar, #pr.OneStar

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"123"}}' \
    --projection-expression "#pr.FiveStar, #pr.ThreeStar, #pr.OneStar" \
    --expression-attribute-names '{"#pr":"ProductReviews"}'
```

式属性名を定義した場合、式全体で一貫して使用する必要があります。さらに、# 記号を省略することはできません。

DynamoDB での式属性値の使用

Amazon DynamoDB の式の属性値は変数として機能します。式の属性値は、比較する実際の値 (ランタイムまでは不明である可能性のある値) の代わりになります。式の属性値はコロン (:) で始まり、1 つ以上の英数字が続きます。

たとえば、ProductCatalog で入手でき、費用が Black 以下の 500 項目のすべてを返すとします。この Scan (AWS Command Line Interface) の例のように AWS CLI オペレーションでフィルタ式を使用できます。

```
aws dynamodb scan \
    --table-name ProductCatalog \
    --filter-expression "contains(Color, :c) and Price <= :p" \
    --expression-attribute-values file://values.json
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
    ":c": { "S": "Black" },
    ":p": { "N": "500" }
}
```

式属性値を定義した場合、式全体で一貫して使用する必要があります。さらに、: 記号を省略することはできません。

式の属性値は、主な条件式、条件式、更新式、およびフィルター式で使用されます。

DynamoDB でのプロジェクション式の使用

テーブルからデータを読み込むには、次のような GetItem、Query、または Scan のオペレーションを使用します。Amazon DynamoDB は、デフォルトですべての項目属性を返します。すべての属性ではなく、一部の属性のみを取得するには、プロジェクション式を使用します。

プロジェクション式は、任意の属性を識別する文字列です。1 つの属性を取得するには、名前を指定します。複数の属性の場合、名前をカンマで区切る必要があります。

ProductCatalog からの DynamoDB で式を使用する場合の項目属性の参照 項目に基づく、プロジェクション式のいくつかの例を次に示します。

- 1 つの最上位属性。

  ```
  Title
  ```

- 3 つの最上位属性。DynamoDB は Color セット全体を取得します。

  ```
  Title, Price, Color
  ```

- 4 つの最上位属性。DynamoDB は RelatedItems および ProductReviews のコンテンツ全体を返します。

  ```
  Title, Description, RelatedItems, ProductReviews
  ```

> **Note**
>
> プロジェクション式は、プロビジョンドスループットの消費には影響しません。DynamoDB は、アプリケーションに返すデータの量ではなく、項目のサイズに基づいて、キャパシティユニットの消費数を決定します。

予約語と特殊文字

DynamoDB には予約語と特殊文字があります。DynamoDB では、これらの予約語と特殊文字を名前に使用できますが、お勧めしません。これらの名前を式で使用するたびに、代わりにエイリアスを使用する必要があるためです。詳細な一覧については、「DynamoDB の予約語」を参照してください。

次の場合は、実際の名前の代わりに式の属性名を使用する必要があります。

- 属性名が DynamoDB の予約語リストにある場合。
- 最初の文字が a-z または A-Z で、2 番目の文字 (存在する場合) が a-Z、A-Z、0-9 のいずれかでなければならないという要件を属性名が満たしていない場合。
- 属性名に、# (ハッシュ) または : (コロン) が含まれている場合。

以下の AWS CLI の例に示しているのは、GetItem オペレーションでプロジェクション式を使用する方法です。この式は最上位スカラー射影内の属性 (Description)、リスト内の最初の要素 (RelatedItems[0])、およびマップ内に入れ子にするリスト (ProductReviews.FiveStar) を取得します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id": { "N": "123" }}' \
    --projection-expression "Description, RelatedItems[0], ProductReviews.FiveStar"
```

この例では、次の JSON が返されます。

```
{
    "Item": {
        "Description": {
            "S": "123 description"
        },
        "ProductReviews": {
            "M": {
                "FiveStar": {
                    "L": [
                        {
                            "S": "Excellent! Can't recommend it highly enough! Buy it!"
                        },
                        {
                            "S": "Do yourself a favor and buy this."
                        }
                    ]
                }
            }
        },
        "RelatedItems": {
            "L": [
                {
                    "N": "341"
                }
            ]
        }
    }
}
```

DynamoDB での更新式の使用

UpdateItem オペレーションは、既存項目を更新します。また、存在しない場合は新しい項目をテーブルに追加します。更新する項目のキーを指定する必要があります。また、変更する属性を示す更新式と、その式に割り当てる値も指定する必要があります。

更新式は、UpdateItem が項目の属性を変更する方法を指定します。たとえば、スカラー値を設定したり、リストまたはマップから要素を削除したりします。

更新式の構文の概要を次に示します。

```
update-expression ::=
    [ SET action [, action] ... ]
    [ REMOVE action [, action] ...]
    [ ADD action [, action] ... ]
    [ DELETE action [, action] ...]
```

更新式は、1 つ以上の句で構成されます。各句は、SET、REMOVE、ADD、または DELETE キーワードで始まります。これらのいずれの句も、任意の順序で更新式に含めることができます。ただし、各アクションキーワードは 1 回のみ表示できます。

各句内には、カンマで区切った 1 つ以上のアクションがあります。各アクションはデータ変更を表します。

> **Note**
>
> 更新式に複数のアクションが含まれている場合、DynamoDB は更新前と同様に、項目の属性値に対して各アクションを評価します。アクションは左から右に連続して適用されないため、アクションの右側のオペランドは常に属性の更新前の値を参照します。同じ式の別のアクションがその属性を変更する場合でも同じです。
> 例えば、項目が {"id": "1", "a": 1, "b": 2, "c": 3} で、次の更新式を実行するとします。
>
> ```
> REMOVE a SET b = a, c = b
> ```
>
> 結果は、{"id": "1", "b": 1, "c": 2} です。b = a と c = b の両方は元の値である a と b (1 と 2) を使用し、a が削除されます。

このセクションの例は、ProductCatalog の DynamoDB でのプロジェクション式の使用 項目に基づいています。

以下のトピックでは、SET アクションのさまざまなユースケースについて説明します。

トピック

- SET — 項目属性の変更または追加
- REMOVE — 項目から属性を削除
- ADD — 数値とセットの更新
- DELETE — セットから要素を削除
- 複数の更新式を使用する

SET — 項目属性の変更または追加

1 つ以上の属性を項目に追加するには、更新式で SET アクションを使用します。これらのいずれかの属性が既に存在する場合、新しい値で上書きされます。既存の属性の上書きを回避するには、SET で if_not_exists 関数を使用できます。if_not_exists 関数は SET アクションに固有で、更新式でのみ使用できます。

SET を使用してリスト要素を更新すると、その要素のコンテンツは、指定した新しいデータで置き換えられます。要素が既に存在していない場合、SET はリストの末尾に新しい要素を追加します。

1 つの SET オペレーションに複数の要素を追加すると、要素は要素番号で順にソートされます。

SET を使用して、Number 型である属性を増減することもできます。複数の SET アクションを実行するには、オペレーションをカンマで区切ります。

次の構文の概要について説明します。

- *path* 要素は、項目へのドキュメントパスです。
- *operand* 要素は、項目へのドキュメントパスまたは関数とすることができます。

```
set-action ::=
    path = value

value ::=
    operand
```

```
    | operand '+' operand
    | operand '-' operand

operand ::=
    path | function

function ::=
    if_not_exists (path, value)
```

項目の指定したパスに属性が含まれていない場合、if_not_exists は value に評価されます。それ以外の場合は、path に評価されます。

次の PutItem オペレーションは、例で参照するサンプル項目を作成します。

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json
```

--item の引数は、ファイル item.json に保存されます。(分かりやすいように、いくつかの項目属性のみが使用されます)

```
{
    "Id": {"N": "789"},
    "ProductCategory": {"S": "Home Improvement"},
    "Price": {"N": "52"},
    "InStock": {"BOOL": true},
    "Brand": {"S": "Acme"}
}
```

トピック

- 属性の変更
- リストおよびマップの追加
- リストに要素を追加
- ネストされたマップ属性の追加
- 数値属性の増減
- リストに要素を追加
- 既存の属性の上書きを防止

属性の変更

Example

ProductCategory 属性および Price 属性を更新します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET ProductCategory = :c, Price = :p" \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
    ":c": { "S": "Hardware" },
    ":p": { "N": "60" }
}
```

> **Note**
>
> UpdateItem オペレーションでは、--return-values ALL_NEW を指定すると、DynamoDB は更新後に表示される項目を返します。

リストおよびマップの追加

Example

新しいリストおよび新しいマップを追加します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems = :ri, ProductReviews = :pr" \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
```

```
    ":ri": {
        "L": [
            { "S": "Hammer" }
        ]
    },
    ":pr": {
        "M": {
            "FiveStar": {
                "L": [
                    { "S": "Best product ever!" }
                ]
            }
        }
    }
}
```

リストに要素を追加

Example

新しい属性を RelatedItems リストに追加します。リストの要素はゼロベースであるため、[0] はリスト内の最初の要素、[1] は 2 番目の要素、という順番で表されることに注意してください。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems[1] = :ri" \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
    ":ri": { "S": "Nails" }
}
```

> **Note**
>
> SET を使用してリスト要素を更新すると、その要素のコンテンツは、指定した新しいデータで置き換えられます。要素が既に存在していない場合、SET はリストの末尾に新しい要素を追加します。

1 つの SET オペレーションに複数の要素を追加すると、要素は要素番号で順にソートされます。

ネストされたマップ属性の追加

Example

ネストされたマップ属性を追加します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #pr.#5star[1] = :r5, #pr.#3star = :r3" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_NEW
```

`--expression-attribute-names` の引数は、ファイル `names.json` に保存されます。

```
{
    "#pr": "ProductReviews",
    "#5star": "FiveStar",
    "#3star": "ThreeStar"
}
```

`--expression-attribute-values` の引数は、ファイル `values.json` に保存されます。

```
{
    ":r5": { "S": "Very happy with my purchase" },
    ":r3": {
        "L": [
            { "S": "Just OK - not that great" }
        ]
    }
}
```

Important

親マップが存在しない場合、ネストされたマップ属性を更新することはできません。親マップ (`ProductReviews.FiveStar`) が存在しない場合にネストされた属性

(ProductReviews など) を更新しようとすると、DynamoDB は「更新式で指定されたドキュメントパスは更新に無効です」というメッセージで ValidationException を返します。
ネストされたマップ属性を後で更新する項目を作成する場合は、親属性の空のマップを初期化します。例えば、次のようになります。

```
{
    "Id": {"N": "789"},
    "ProductReviews": {"M": {}},
    "Metadata": {"M": {}}
}
```

これにより、ProductReviews.FiveStar のようなネストされた属性を エラーなしで更新できます。

数値属性の増減

既存の数値属性は増減することができます。これを行うには、+ (プラス) および - (マイナス) 演算子を使用します。

Example

項目の Price を下げます。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET Price = Price - :p" \
    --expression-attribute-values '{":p": {"N":"15"}}' \
    --return-values ALL_NEW
```

Price を上げるために、更新式で + 演算子を使用します。

リストに要素を追加

リストの最後に要素を追加できます。これを行うには、SET 関数の list_appendを使用します。(関数名では、大文字と小文字が区別されます)。list_append 関数は SET アクションに固有で、更新式でのみ使用できます。構文は次のとおりです。

- list_append (*list1*, *list2*)

この関数は、入力として 2 つのリストを受け取り、*list2* のすべての要素を *list1* に追加します。

Example

リストに要素を追加 で、RelatedItems リストを追加し Hammer と Nails の 2 つの要素を追加します。次に、RelatedItems の末尾にさらに 2 つの要素を追加します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #ri = list_append(#ri, :vals)" \
    --expression-attribute-names '{"#ri": "RelatedItems"}' \
    --expression-attribute-values file://values.json  \
    --return-values ALL_NEW
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
    ":vals": {
        "L": [
            { "S": "Screwdriver" },
            {"S": "Hacksaw" }
        ]
    }
}
```

最後に、RelatedItems の先頭にもう 1 つの要素を追加します。これを実行するには、list_append 要素の順序を入れ替えます。(list_append は 2 つのリストを入力し、1 番目のリストに 2 番目のリストを追加することに注意してください)。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET #ri = list_append(:vals, #ri)" \
    --expression-attribute-names '{"#ri": "RelatedItems"}' \
    --expression-attribute-values '{":vals": {"L": [ { "S": "Chisel" }]}}' \
    --return-values ALL_NEW
```

生成される RelatedItems 属性には次の順序で 5 つの要素が含まれます。Chisel、Hammer、Nails、Screwdriver、 Hacksaw。

既存の属性の上書きを防止

Example

項目にすでに Price 属性がない場合にのみ、項目の Price を設定します (Price がすでに存在する場合は何も起こりません)。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET Price = if_not_exists(Price, :p)" \
    --expression-attribute-values '{":p": {"N": "100"}}' \
    --return-values ALL_NEW
```

REMOVE — 項目から属性を削除

1 つ以上の属性を Amazon DynamoDB の項目から削除するには、更新式で REMOVE アクションを使用します。複数の REMOVE アクションを実行するには、オペレーションをカンマで区切ります。

更新式の REMOVE の構文の概要を次に示します。唯一のオペランドは、削除する属性のドキュメントパスです。

```
remove-action ::=
    path
```

Example

項目から属性を削除します。(属性が存在しない場合は、何も起こりません)。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "REMOVE Brand, InStock, QuantityOnHand" \
    --return-values ALL_NEW
```

リストから要素を削除

REMOVE を使用して、リストから個別要素を削除できます。

Example

リストに要素を追加 で、5 つの要素を含むようにリスト属性 (RelatedItems) を変更します。

- [0]—Chisel
- [1]—Hammer
- [2]—Nails
- [3]—Screwdriver
- [4]—Hacksaw

次の AWS Command Line Interface (AWS CLI) の例では、リストから Hammer と Nails を削除します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "REMOVE RelatedItems[1], RelatedItems[2]" \
    --return-values ALL_NEW
```

Hammer と Nails を削除すると、残りの要素は変わります。リストには以下が含まれます。

- [0]—Chisel
- [1]—Screwdriver
- [2]—Hacksaw

ADD — 数値とセットの更新

> **Note**
>
> 一般的に、べき等なオペレーションを確保するために ADD ではなく SET を使用することをお勧めします。

項目に新しい属性および値を追加するには、更新式で ADD アクションを使用します。

属性が既に存在する場合、ADD の動作は属性のデータ型によって決まります。

- 属性が数値で、追加する値も数値である場合、値は既存の属性に数学的に追加されます (値が負の数値である場合は、既存の属性から減算されます)。
- 属性が設定され、追加する値も設定された場合、値は既存のセットに付加されます。

> **Note**
>
> ADD アクションでは、数値とセットデータ型のみがサポートされます。

複数の ADD アクションを実行するには、オペレーションをカンマで区切ります。

次の構文の概要について説明します。

- *path* 要素は、属性へのドキュメントパスです。属性は Number またはセットデータ型である必要があります。
- *value* 要素は、属性に追加する数値 (Number データ型の場合)、または属性に付加するセット (セット型の場合) です。

```
add-action ::=
    path value
```

以下のトピックでは、ADD アクションのさまざまなユースケースについて説明します。

トピック

- 数値の追加
- セットに要素を追加

数値の追加

QuantityOnHand 属性が存在しないと想定します。次の AWS CLI の例では、QuantityOnHand を 5 に設定します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD QuantityOnHand :q" \
    --expression-attribute-values '{":q": {"N": "5"}}' \
    --return-values ALL_NEW
```

これで QuantityOnHand が存在するようになったので、QuantityOnHand が毎回 5 増分するように例を再実行できます。

セットに要素を追加

Color 属性が存在しないと想定します。次の AWS CLI の例では、Color を 2 つの要素を持つ文字列セットに設定します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD Color :c" \
    --expression-attribute-values '{":c": {"SS":["Orange", "Purple"]}}' \
    --return-values ALL_NEW
```

これで Color が存在するので、さらに要素を追加できます。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "ADD Color :c" \
    --expression-attribute-values '{":c": {"SS":["Yellow", "Green", "Blue"]}}' \
    --return-values ALL_NEW
```

DELETE — セットから要素を削除

> **Important**
>
> DELETE は Set データ型のみをサポートします。

1 つ以上の要素をセットから削除するには、更新式で DELETE アクションを使用します。複数の DELETE アクションを実行するには、オペレーションをカンマで区切ります。

次の構文の概要について説明します。

- *path* 要素は、属性へのドキュメントパスです。属性はセットデータ型である必要があります。
- *#####*は *path* から削除する 1 つ以上の要素です。*#####*はセット型として指定する必要があります。

```
delete-action ::=
    path subset
```

Example

セットに要素を追加 で、Color 文字列セットを作成します。この例では、セットから要素の一部を削除します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "DELETE Color :p" \
    --expression-attribute-values '{":p": {"SS": ["Yellow", "Purple"]}}' \
    --return-values ALL_NEW
```

複数の更新式を使用する

単一の更新式で複数のアクションを使用できます。すべての属性参照は、いずれかのアクションが適用される前に、項目の状態に対して解決されます。

Example

項目 {"id": "1", "a": 1, "b": 2, "c": 3} を指定すると、次の式は a を削除し、b と c の値をシフトします。

```
aws dynamodb update-item \
    --table-name test \
    --key '{"id":{"S":"1"}}' \
    --update-expression "REMOVE a SET b = a, c = b" \
    --return-values ALL_NEW
```

結果は、{"id": "1", "b": 1, "c": 2} です。同じ式で a が削除され、b が再割り当てされても、両方の参照は元の値に解決されます。

Example

属性の値を変更して別の属性を完全に削除したい場合は、SET アクションと REMOVE アクションを 1 つのステートメントで使用できます。このオペレーションでは、Price 値を 15 に減らし、項目から InStock 属性も削除します。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
```

```
    --update-expression "SET Price = Price - :p REMOVE InStock" \
    --expression-attribute-values '{":p": {"N":"15"}}' \
    --return-values ALL_NEW
```

Example

リストに追加しつつ別の属性の値も変更したい場合は、1 つのステートメントで 2 つの SET アクションを使用できます。このオペレーションでは、RelatedItems リスト属性に「Nails」が追加され、Price 値も 21 に設定されます。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"789"}}' \
    --update-expression "SET RelatedItems[1] = :newValue, Price = :newPrice" \
    --expression-attribute-values '{":newValue": {"S":"Nails"}, ":newPrice":
 {"N":"21"}}'  \
    --return-values ALL_NEW
```

DynamoDB の条件式とフィルター式、演算子、関数

Amazon DynamoDB テーブルのデータを操作するには、PutItem、UpdateItem、DeleteItem の各オペレーションを使用します。これらのデータ操作オペレーションでは、どの項目を修正する必要があるかを判断するために、条件式を指定できます。条件式が true と評価される場合、オペレーションは成功します。それ以外の場合は、このオペレーションは失敗します。

このセクションでは、Amazon DynamoDB の書き込みフィルター式と条件式の組み込み関数およびキーワードについて説明します。DynamoDB で使用する関数とプログラミングの詳細については、「DynamoDB と AWS SDK を使用したプログラミング」と「DynamoDB API リファレンス」を参照してください。

トピック

- フィルター式と条件式の構文
- 比較の実行
- 関数
- 論理評価
- 括弧
- 条件の優先順位

フィルター式と条件式の構文

以下の構文の概要で、*operand* は次のいずれかです。

- 最上位の属性名 (Id、Title、Description、ProductCategory など)
- 入れ子の属性を参照するドキュメントパス

```
condition-expression ::=
      operand comparator operand
    | operand BETWEEN operand AND operand
    | operand IN ( operand (',' operand (, ...) ))
    | function
    | condition AND condition
    | condition OR condition
    | NOT condition
    | ( condition )

comparator ::=
    =
    | <>
    | <
    | <=
    | >
    | >=

function ::=
    attribute_exists (path)
    | attribute_not_exists (path)
    | attribute_type (path, type)
    | begins_with (path, substr)
    | contains (path, operand)
    | size (path)
```

比較の実行

これらのコンパレータを使用して、オペランドを単一の値と比較します。

- *a* = *b* — *a* が *b* と等しい場合、True
- *a* <> *b* — *a* が *b* と等しくない場合、True
- *a* < *b* — *a* が *b* 未満の場合、True
- *a* <= *b* — *a* が *b* 以下である場合、True

- *a* > *b* — *a* が *b* より大きい場合、True
- *a* >= *b* — *a* が *b* 以上である場合、True

値の範囲または値の列挙リストに対してオペランドを比較するには、BETWEEN および IN キーワードを使用します。

- *a* BETWEEN *b* AND *c* - *a* が *b* 以上で、*c* 以下である場合、True。
- *a* IN (*b*, *c*, *d*) — *a* がリスト内の任意の値と等しい場合、True。例では、*b*、*c*、*d* のいずれかと等しい場合。リストには、コンマで区切って最大 100 個の値を指定できます。

ブール属性をフィルタリングするには、= または <> 演算子を使用してブール値と比較します。属性自体を条件として参照することはできません。式の属性値にブール値を指定します。例えば、次のフィルター式は、isVisible 属性が true である項目のみを返します。

```
--filter-expression "isVisible = :visible" \
--expression-attribute-values '{":visible": {"BOOL": true}}'
```

関数

以下の関数を使用して、ある属性が項目に存在するか判定したり、属性の値を評価したりします。これらの関数名では大文字と小文字が区別されます。入れ子の属性では、完全ドキュメントパスを指定する必要があります。

関数	説明
attribute_exists (*path*)	項目に、path で指定した属性が含まれる場合、true。 保存値が null 型 (NULL) である属性は引き続き存在すると見なされるため、attribute_exists はこの属性に対して true を返します。これは、項目に存在しない属性とは異なります。一部の SDK とオブジェクトマッパーは、NULL 型を保存するのではなく、アプリケーションに null 値を持つ属性を省略します。この場合、属性は存在せず、attribute_exists は false を返します。

関数	説明
	例: Product テーブルの項目に側面図があるかどうかを確認します。 • attribute_exists (#Pictures.#SideView)
attribute_not_exists (*path*)	path で指定した属性が項目に存在しない場合、true。 例: 項目に Manufacturer 属性があるかどうかを確認します。 • attribute_not_exists (Manufacturer)

関数	説明
attribute_type (*path*, *type*)	指定したパスの属性が、特定のデータ型のものである場合、true。type パラメータは次のいずれかである必要があります。 • S — 文字列 • SS — 文字列セット • N — 数値 • NS — 数値セット • B — バイナリ • BS — バイナリセット • BOOL — ブール • NULL — Null • L — リスト • M — マップ type パラメータには、式の属性値を使用する必要があります。 例: QuantityOnHand 属性がリスト型のものであるかどうかを確認します。この例では、:v_sub は文字列 L のプレースホルダーです。 • attribute_type (ProductReviews.FiveStar, :v_sub)

関数	説明
	type パラメータには、式の属性値を使用する必要があります。
begins_with (*path*, *substr*)	path で指定された属性が特定のサブ文字列から始まる場合、true。 例: 正面図 URL の最初の数文字が http:// かどうかを確認します。 • begins_with (Pictures.FrontView, :v_sub) 式の属性値 :v_sub は、http:// のプレースホルダーです。

関数	説明
`contains (path, operand)`	path で指定された属性が次の属性である場合、true。 • 特定のサブ文字列を含む String。 • 設定の中に特定の要素を含む Set。 • リスト内に特定の要素を含む List。 path で指定された属性が String である場合、operand は String である必要があります。path で指定された属性が Set の場合、operand は一連の要素タイプである必要があります。 パスとオペランドは区別する必要があります。つまり、`contains (a, a)` が エラーを返します。 例: Brand 属性にサブ文字列 Company が含まれているかどうかを確認します。 • `contains (Brand, :v_sub)` 式の属性値 :v_sub は、Company のプレースホルダーです。 例: 製品が赤で入手可能かどうかを確認します。 • `contains (Color, :v_sub)`

関数	説明
	式の属性値 :v_sub は、Red のプレースホルダーです。

関数	説明
size (*path*)	属性のサイズを表す数値を返します。以下は、size で使用できる有効なデータ型です。 属性は String 型で、size は文字列の長さを返します。 例: 文字列 Brand が 20 文字以下であるかどうかを確認します。式の属性値 :v_sub は、20 のプレースホルダーです。 • size (Brand) <= :v_sub 属性が Binary バイナリ型の場合、size は属性値のバイト数を返します。 例: ProductCatalog 項目に VideoClip という名前のバイナリ属性があるとします。この属性には使用中の製品の短いビデオが含まれます。次の式は、VideoClip が 64,000 バイトを超えるかどうかを確認します。式の属性値 :v_sub は、64000 のプレースホルダーです。 • size(VideoClip) > :v_sub 属性が Set データ型の場合、size は設定の要素数を返します。 例: 製品が複数の色で入手可能かどうかを確認します。式の属性値 :v_sub は、1 のプレースホルダーです。 • size (Color) < :v_sub

関数	説明
	属性が List 型または Map のものである場合、size は子要素の数を返します。 例: OneStar のレビューの数が特定のしきい値を超えたかどうかを確認します。式の属性値 :v_sub は、3 のプレースホルダーです。 • `size(ProductReviews.OneStar) > :v_sub`

論理評価

論理評価を実行するには、AND、OR、NOT キーワードを使用します。以下のリストでは、*a* と *b* は評価される条件を示しています。

- *a* AND *b* — *a* と *b* の両方が true である場合、True。
- *a* OR *b* — *a* または *b* のどちらか (または両方) が true の場合、True。
- NOT *a*— *a* が false の場合は True。*a* が true の場合は Falsee。

以下は、オペレーションの AND のコード例です。

```
dynamodb-local (*)> select * from exprtest where a > 3 and a < 5;
```

括弧

論理評価の優先順位を変更するには括弧を使用します。たとえば、条件 *a* と *b* が true で、条件 *c* が false であるとします。次の式は true と評価されます。

- *a* OR *b* AND *c*

しかし、条件を括弧で囲むと、それが最初に評価されます。たとえば、次の式は false と評価されます。

- (*a* OR *b*) AND *c*

> **Note**
>
> 式では括弧を入れ子にできます。最も内側の括弧が最初に評価されます。

以下は、論理評価で括弧を付けたコード例です。

```
dynamodb-local (*)> select * from exprtest where attribute_type(b, string)
or ( a = 5 and c = “coffee”);
```

条件の優先順位

DynamoDB では、条件は次の優先順位ルールを使用して左から右に評価されます。

- = <> < <= > >=
- IN
- BETWEEN
- attribute_exists attribute_not_exists begins_with contains
- 括弧
- NOT
- AND
- OR

DynamoDB 条件式 CLI の例

条件式を使用する AWS Command Line Interface (AWS CLI) の例のいくつかを次に示します。これらの例は、「ProductCatalog」で紹介されている DynamoDB で式を使用する場合の項目属性の参照 テーブルに基づいています。このテーブルのパーティションキーは Id です。ソートキーはありません。次の PutItem オペレーションは、例で参照するサンプル ProductCatalog 項目を作成します。

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json
```

--item の引数は、ファイル item.json に保存されます。(分かりやすいように、いくつかの項目属性のみが使用されます)

```
{
    "Id": {"N": "456" },
    "ProductCategory": {"S": "Sporting Goods" },
    "Price": {"N": "650" }
}
```

トピック

- 条件付き配置
- 条件付き削除
- 条件付き更新
- 条件式の例

条件付き配置

PutItem オペレーションはプライマリキーが同じ項目を上書きします (存在する場合)。これを回避するには、条件式を使用します。これにより、問題の項目が同じプライマリキーを持っていない場合にのみ書き込みが続行されます。

次の例では、attribute_not_exists() を使用して、書き込みを試みる前に、同じプライマリキーを持つ項目が既に存在するかどうかを確認しています。DynamoDB は、書き込みオペレーションの条件式をリクエストのキーで識別される項目に対して評価するため、Id 値を持つ項目がまだ存在しない場合にのみ attribute_not_exists(Id) は true になります。true の場合、書き込みは続行されます。該当するキーを持つ項目が既に存在する場合、条件は false であり、DynamoDB は書き込みを拒否するため、上書きが防止されます。

> **Note**
>
> 複合プライマリキー (パーティションキーとソートキー) を持つテーブルの場合、DynamoDB は引き続き両方のキー値で識別される単一の項目に対して条件を評価します。例えば、attribute_not_exists(pk) のように、キー属性に対して attribute_not_exists() をテストすると、そのパーティションキーとソートキーを持つ項目がまだ存在しない場合にのみ true になるため、その項目の上書きを防ぐことができます。

```
aws dynamodb put-item \
```

```
    --table-name ProductCatalog \
    --item file://item.json \
    --condition-expression "attribute_not_exists(Id)"
```

条件式が false と評価し、DynamoDB は次のエラーメッセージを返します。条件付きリクエストが失敗しました。

> **Note**
>
> `attribute_not_exists` および他の関数についての詳細は、「DynamoDB の条件式とフィルター式、演算子、関数」を参照してください。

条件付き削除

条件付き削除を実行するには、条件式とともに `DeleteItem` オペレーションを使用します。条件式は、オペレーションが成功するためには true に評価される必要があります。それ以外の場合、オペレーションは失敗します。

上で定義した項目について考えてみます。

次の条件を満たしている場合にだけ項目を削除するとします。

- `ProductCategory` は「Sporting Goods」または「Gardening Supplies」のどちらかです。
- `Price` は 500 ~ 600 です。

次の例では、項目を削除しようとしています。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"456"}}' \
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (Price between :lo
 and :hi)" \
    --expression-attribute-values file://values.json
```

`--expression-attribute-values` の引数は、ファイル `values.json` に保存されます。

```
{
    ":cat1": {"S": "Sporting Goods"},
```

```
    ":cat2": {"S": "Gardening Supplies"},
    ":lo": {"N": "500"},
    ":hi": {"N": "600"}
}
```

> **Note**
>
> 条件式では、: (コロン文字) は式の属性値 (実際の値のプレースホルダー) を示します。詳細については、「」を参照してくださいDynamoDB での式属性値の使用
> IN、AND およびその他のキーワードについての詳細は、「DynamoDB の条件式とフィルター式、演算子、関数」を参照してください。

この例では、ProductCategory 比較の評価結果は true になりますが、Price 比較は false と評価されます。これにより、条件式は false と評価され、DeleteItem オペレーションは失敗します。

条件付き更新

条件付き更新を実行するには、条件式とともに UpdateItem オペレーションを使用します。条件式は、オペレーションが成功するためには true に評価される必要があります。それ以外の場合、オペレーションは失敗します。

> **Note**
>
> UpdateItem は更新式もサポートします。項目に加える変更を指定する変更内容を指定します。詳細については、「DynamoDB での更新式の使用」を参照してください。

上で定義した項目から開始するとします。

次の例では、UpdateItem オペレーションを実行します。製品の Price を 75 減らしようとしていますが、現在の Price が 500 以下の場合、条件式が更新を防ぎます。

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --update-expression "SET Price = Price - :discount" \
    --condition-expression "Price > :limit" \
    --expression-attribute-values file://values.json
```

--expression-attribute-values の引数は、ファイル values.json に保存されます。

```
{
    ":discount": { "N": "75"},
    ":limit": {"N": "500"}
}
```

開始 Price が 650 の場合、UpdateItem オペレーションによって Price は 575 に下げられます。UpdateItem アクションを再度実行すると、Price は 500 に減らされます。3 度目に実行した場合、条件式は false と評価されて、更新は失敗します。

> **Note**
>
> 条件式では、: (コロン文字) は式の属性値 (実際の値のプレースホルダー) を示します。詳細については、「DynamoDB での式属性値の使用」を参照してください。
> 「>」およびその他の演算子についての詳細は、「DynamoDB の条件式とフィルター式、演算子、関数」を参照してください。

条件式の例

次の例で使用される関数の詳細については、「DynamoDB の条件式とフィルター式、演算子、関数」を参照してください。式で異なる属性タイプを指定する方法の詳細については、「DynamoDB で式を使用する場合の項目属性の参照」を参照してください。

項目の属性を確認

属性が存在するか (または存在しないか) を確認できます。条件式が true と評価される場合、オペレーションは成功で、それ以外の場合、オペレーションは失敗です。

次の例では、attribute_not_exists 属性がない場合にのみ製品を削除するために Price を使用します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_not_exists(Price)"
```

DynamoDB は attribute_exists 関数も提供します。次の例では、悪いレビューを受け取った場合のみ製品を削除します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_exists(ProductReviews.OneStar)"
```

属性タイプの確認

属性値のデータ型は、attribute_type 関数を使用して確認できます。条件式が true と評価される場合、オペレーションは成功で、それ以外の場合、オペレーションは失敗です。

次の例では、attribute_type を使用して、文字列セットタイプの Color 属性がある場合にのみ、製品を削除します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "attribute_type(Color, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

--expression-attribute-values の引数は、expression-attribute-values.json ファイルに格納されます。

```
{
    ":v_sub":{"S":"SS"}
}
```

文字列開始値の確認

文字列属性値が特定のサブ文字列で始まるかどうかを確認するには、begins_with 関数を使用します。条件式が true と評価される場合、オペレーションは成功で、それ以外の場合、オペレーションは失敗です。

次の例では、begins_with マップの FrontView 要素が特定の値で始まる場合にのみ、Pictures を使用して製品を削除します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "begins_with(Pictures.FrontView, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

--expression-attribute-values の引数は、expression-attribute-values.json ファイルに格納されます。

```
{
    ":v_sub":{"S":"http://"}
}
```

セット内の要素の確認

contains 関数を使用して、セット内の要素を確認したり、文字列内のサブ文字列を検索したりできます。条件式が true と評価される場合、オペレーションは成功で、それ以外の場合、オペレーションは失敗です。

次の例では、contains 文字列セットに特定の値を持つ要素がある場合にのみ、Color を使用して製品を削除します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "contains(Color, :v_sub)" \
    --expression-attribute-values file://expression-attribute-values.json
```

--expression-attribute-values の引数は、expression-attribute-values.json ファイルに格納されます。

```
{
    ":v_sub":{"S":"Red"}
}
```

属性値のサイズの確認

属性値のサイズを確認するには、size 関数を使用します。条件式が true と評価される場合、オペレーションは成功で、それ以外の場合、オペレーションは失敗です。

次の例では、size バイナリ属性のサイズが VideoClip バイトより大きい場合にのみ 64000 を使用して、製品を削除します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
```

```
    --key '{"Id": {"N": "456"}}' \
    --condition-expression "size(VideoClip) > :v_sub" \
    --expression-attribute-values file://expression-attribute-values.json
```

`--expression-attribute-values` の引数は、expression-attribute-values.json ファイルに格納されます。

```
{
    ":v_sub":{"N":"64000"}
}
```

DynamoDB での Time to Live (TTL) の使用

DynamoDB の Time to Live (TTL) は、不要になった項目を削除するためのコスト効率に優れた方法です。TTL では、項目がいつ不要になるかを示す有効期限タイムスタンプを項目ごとに定義できます。DynamoDB は、書き込みスループットを消費することなく、有効期限が切れてから数日以内に期限切れの項目を自動的に削除します。

TTL を使用するには、まずテーブルで TTL を有効にし、次に TTL の有効期限タイムスタンプを格納する特定の属性を定義します。タイムスタンプは、秒単位の UNIX エポック時間形式の数値データ型として保存する必要があります。数値型ではない TTL 属性を持つ項目は、TTL プロセスによって無視されます。項目が作成または更新されるたびに、有効期限を計算して TTL 属性に保存できます。

期限切れの有効な TTL 属性を持つ項目は、随時システムによって削除される可能性があります。削除は、通常は有効期限が切れてから数日以内に行われます。削除待ちの期限切れ項目は、TTL 属性の変更や削除を含め、引き続き更新できます。項目が削除される前に、項目の TTL 属性を将来のタイムスタンプに更新 (または TTL 属性を削除) すると、その項目は期限切れと見なされなくなり、TTL プロセスによって削除されません。期限切れの項目を更新する際には、その項目がその後削除されないように条件式を使用することをおすすめします。フィルター式を使用して、期限切れの項目を Scan 結果と Query 結果から削除します。

削除済みの項目は、通常の削除操作で削除された項目と同様に機能します。削除すると、項目はユーザーによる削除ではなくサービスによる削除として DynamoDB ストリームに入り、他の削除操作と同様にローカルセカンダリインデックスとグローバルセカンダリインデックスから削除されます。

グローバルテーブルバージョン 2019.11.21 (現行) を使用しており、TTL 機能も使用している場合、DynamoDB は TTL による削除をすべてのレプリカテーブルにレプリケートします。最初の TTL による削除の場合、TTL の有効期限切れが発生したリージョンでは、書き込みキャパシティユニット (WCU) は消費されません。ただし、レプリカテーブルにレプリケートされた TTL 削除では、プロ

ビジョンドキャパシティを使用する場合はレプリケートされた書き込みキャパシティユニットが消費され、オンデマンドのキャパシティモードを使用する場合はレプリケートされた書き込みユニットが各レプリカリージョンで消費され、該当する料金が適用されます。

TTL の詳細については、次のトピックを参照してください。

トピック

- DynamoDB での Time to Live (TTL) の有効化
- DynamoDB での Time to Live (TTL) の計算
- 期限切れのアイテムと Time to Live (TTL) の使用

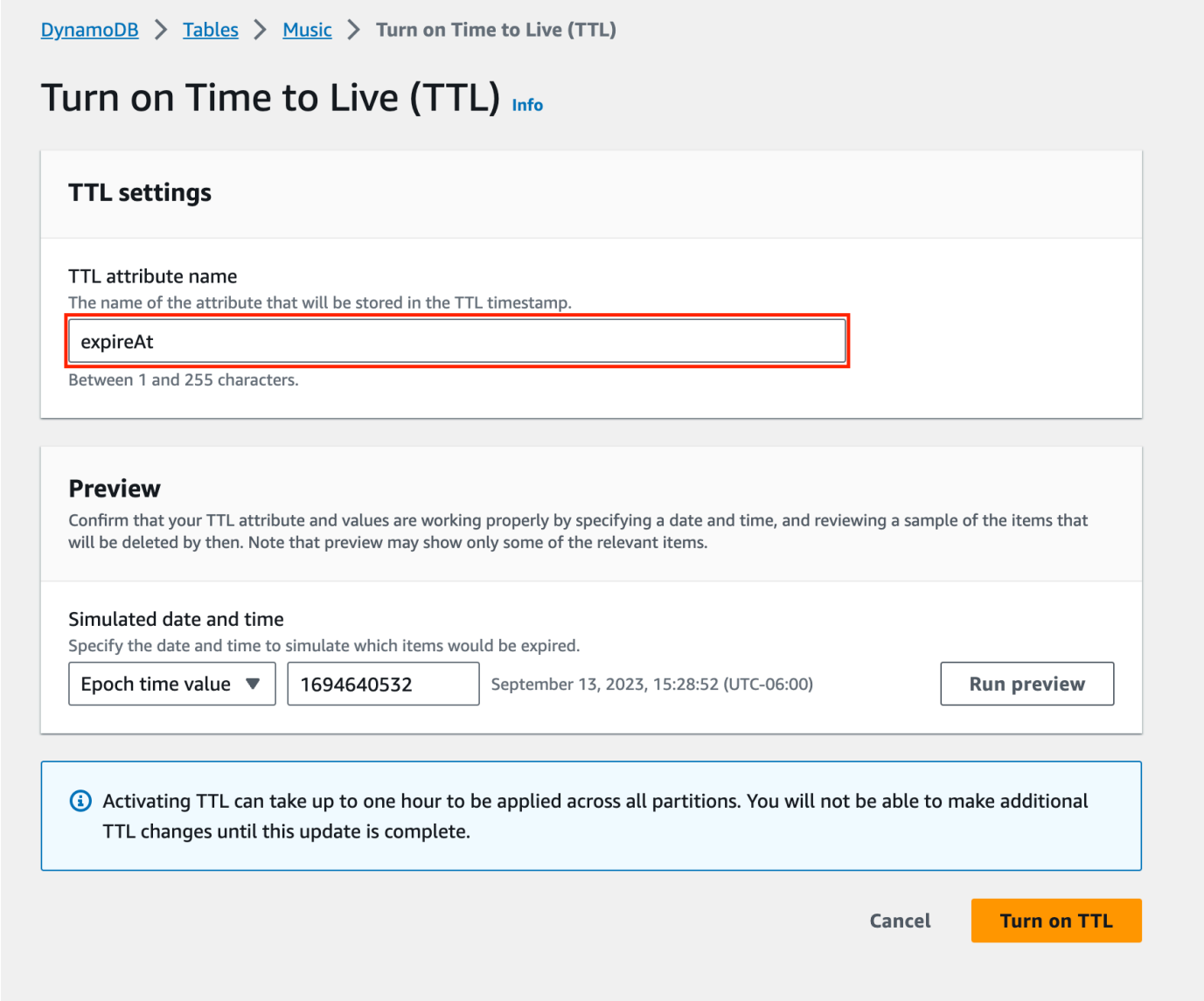
DynamoDB での Time to Live (TTL) の有効化

> **Note**
>
> TTL 機能の適切なオペレーションのデバッグと検証を支援するために、項目 TTL に指定した値は DynamoDB 診断ログにプレーンテキストとして記録されます。

TTL を有効にするには、Amazon DynamoDB コンソールまたは AWS Command Line Interface (AWS CLI) を使用するか、「Amazon DynamoDB API リファレンス」の該当する AWS SDK のいずれかを使用できます。すべてのパーティションで TTL を有効にするには約 1 時間かかります。

AWS コンソールを使用して DynamoDB TTL を有効にする

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. [テーブル] を選択し、変更するテーブルを選択します。
3. [追加の設定] タブの [Time to Live (TTL)] セクションで、[オンにする] を選択します。
4. テーブルで TTL を有効にする場合、DynamoDB では、項目が失効可能かどうかを判断するときにサービスが検索する、特定の属性名を識別する必要があります。以下に示す TTL 属性名では大文字と小文字が区別されます。また、TTL 属性名は読み取り/書き込み操作で定義された属性と一致する必要があります。不一致があると、期限切れの項目は削除されなくなります。TTL 属性の名前を変更するには、TTL を無効にした後で、新しい属性で TTL を再度有効にする必要があります。TTL は、無効化されてから約 30 分間、削除処理を続けます。TTL は復元されたテーブルで再設定する必要があります。



5. (オプション) 有効期限の日付と時刻をシミュレートし、いくつかの項目を照合することでテストを実行できます。これにより、項目のサンプルリストが提供され、指定した TTL 属性名と有効期限を含む項目があることが確認されます。

TTL を有効にすると、DynamoDB コンソールで項目を表示するときに TTL 属性に TTL のマークが付けられます。属性の上にポインターを合わせると、項目が期限切れになる日時を表示できます。

API を使用して DynamoDB TTL を有効にする

> **Note**
>
> `UpdateTimeToLive` はべき等性ではありません。TTL が既に有効になっている場合、または 1 時間以内に同じテーブルに対して `UpdateTimeToLive` を複数回呼び出した

場合、オペレーションはエラーを表示することなく成功したように処理されるのではなく、ValidationException (HTTP 400) を返します。TTL が既にアクティブなときに、別の属性名で TTL を有効にした場合も、ValidationException が返されます。呼び出しが常に成功すると想定するのではなく、呼び出しコードでこの例外を処理します。例えば、「TimeToLive が既に有効になっています」というエラーを、既に設定が指定されていることの確認として扱うことができます。

Python

コードで UpdateTimeToLive オペレーションを使用して TTL を有効にすることができます。

```
import boto3


def enable_ttl(table_name, ttl_attribute_name):
    """
    Enables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table
    :param ttl_attribute_name: The name of the TTL attribute being provided to the
 table.
    """
    try:
        dynamodb = boto3.client('dynamodb')

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={
                'Enabled': True,
                'AttributeName': ttl_attribute_name
            }
        )

        # In the returned response, check for a successful status code.
        if response['ResponseMetadata']['HTTPStatusCode'] == 200:
            print("TTL has been enabled successfully.")
        else:
```

```
            print(f"Failed to enable TTL, status code {response['ResponseMetadata']
['HTTPStatusCode']}")
    except Exception as ex:
        print("Couldn't enable TTL in table %s. Here's why: %s" % (table_name, ex))
        raise


# your values
enable_ttl('your-table-name', 'expirationDate')
```

TTL が有効になっていることを確認するには、DescribeTimeToLive オペレーションを使用してテーブルの TTL ステータスを記述します。TimeToLive ステータスは ENABLED または DISABLED です。

```
# create a DynamoDB client
dynamodb = boto3.client('dynamodb')

# set the table name
table_name = 'YourTable'

# describe TTL
response = dynamodb.describe_time_to_live(TableName=table_name)
```

JavaScript

コードで UpdateTimeToLiveCommand オペレーションを使用して TTL を有効にすることができます。

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-dynamodb";

const enableTTL = async (tableName, ttlAttribute) => {

    const client = new DynamoDBClient({});

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: true,
            AttributeName: ttlAttribute
        }
    };

    try {
```

```
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL enabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to enable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error enabling TTL: ${e}`);
        throw e;
    }
};

// call with your own values
enableTTL('ExampleTable', 'exampleTtlAttribute');
```

AWS CLI を使用して Time to Live を有効にする

1. TTLExample テーブルで TTL を有効にします。

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-
specification "Enabled=true, AttributeName=ttl"
```

2. TTLExample テーブルで TTL を無効にします。

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
    "TimeToLiveDescription": {
        "AttributeName": "ttl",
        "TimeToLiveStatus": "ENABLED"
    }
}
```

3. BASH シェルおよび AWS CLI を使用して 有効期限 (TTL) 属性が設定された項目を TTLExample テーブルに追加するには:

```
EXP=`date -d '+5 days' +%s`
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl":
 {"N": "'$EXP'"}}'
```

この例では、現在の日付から開始し、有効期限を作成するために 5 日を追加します。次に、有効期限をエポック時間形式に変換してから、最後に項目を「TTLExample」テーブルに追加します。

> **Note**
>
> 有効期限 (TTL) の有効期限切れの値を設定する 1 つの方法は、有効期限に追加する秒数を計算することです。たとえば、5 日間は 432,000 秒です。ただし、多くの場合は最初に日付で開始し、そこから作業を開始することをお勧めします。

次の例のように、現在の時間をエポック時間形式で取得するのは非常に簡単です。

- Linux ターミナル: `date +%s`
- Python: `import time; int(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

CloudFormation を使用して DynamoDB TTL を有効にします。

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  TTLExampleTable:
    Type: AWS::DynamoDB::Table
    Description: "A DynamoDB table with TTL Specification enabled"
    Properties:
      AttributeDefinitions:
        - AttributeName: "Album"
          AttributeType: "S"
        - AttributeName: "Artist"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "Album"
          KeyType: "HASH"
        - AttributeName: "Artist"
          KeyType: "RANGE"
      ProvisionedThroughput:
        ReadCapacityUnits: "5"
        WriteCapacityUnits: "5"
      TimeToLiveSpecification:
        AttributeName: "TTLExampleAttribute"
```

```
        Enabled: true
```

CloudFormation テンプレート内で TTL を使用する方法の詳細については、こちらを参照してください。

DynamoDB での Time to Live (TTL) の計算

TTL を実装する一般的な方法は、作成日または最終更新日に基づいて項目の有効期限を設定することです。これは、`createdAt` タイムスタンプと `updatedAt` タイムスタンプに時間を追加することで実現できます。例えば、新しく作成されたアイテムの TTL を `createdAt` + 90 日に設定できます。項目が更新されると、TTL は `updatedAt` + 90 日に再計算されます。

計算された有効期限は、秒単位の エポック形式である必要があります。TTL が 5 年を超えて経過すると、有効期限と削除の対象外になります。他の形式を使用すると、TTL プロセスはその項目を無視します。項目の有効期限を未来の時点に設定した場合、その期間が過ぎると項目は期限切れになります。例えば、有効期限を 1724241326 (2024 年 8 月 21 日 (月) 11:55:26 (UTC)) に設定したとします。指定した時刻以降、項目は有効期限切れになります。最小 TTL 期間はありません。有効期限は、現在の時刻から 5 分後など、将来の任意の時刻に設定できます。

トピック

- 項目を作成して Time to Live を設定する
- 項目を更新して Time to Live を再計算する

項目を作成して Time to Live を設定する

次の例は、`expireAt` を TTL 属性名として使用して、新しいアイテムを作成するときに有効期限を計算する方法を示しています。代入ステートメントは、現在の時刻を変数として取得します。この例では、有効期限は現在の時刻から 90 日後として計算されます。その後、時刻はエポック形式に変換され、TTL 属性に整数データ型として保存されます。

次のコード例は、TTL を含む項目の作成方法を示しています。

Java

SDK for Java 2.x

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
```

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

/**
 * Creates an item in a DynamoDB table with TTL attributes.
 * This class demonstrates how to add TTL expiration timestamps to DynamoDB
 items.
 */
public class CreateTTL {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
                sortKey - The name of the sort key. Also known as the range
 attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
 is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String CREATION_DATE_ATTR = "creationDate";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String SUCCESS_MESSAGE = "%s PutItem operation with TTL
 successful.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
 DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;
```

```
    /**
     * Constructs a CreateTTL instance with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public CreateTTL(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Constructs a CreateTTL with a default DynamoDB client.
     */
    public CreateTTL() {
        this.dynamoDbClient = null;
    }

    /**
     * Main method to demonstrate creating an item with TTL.
     *
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new CreateTTL().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and create an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
```

```
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final CreateTTL createTTL = new CreateTTL(ddb);
            createTTL.createItemWithTTL(tableName, primaryKey, sortKey);
            return 0;
        } catch (Exception e) {
            throw e;
        }
    }

    /**
     * Creates an item in the specified table with TTL attributes.
     *
     * @param tableName The name of the table
     * @param primaryKeyValue The value for the primary key
     * @param sortKeyValue The value for the sort key
     * @return The response from the PutItem operation
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     */
    public PutItemResponse createItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
sortKeyValue) {
        // Get current time in epoch second format
        final long createDate = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = createDate + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        final Map<String, AttributeValue> itemMap = new HashMap<>();
        itemMap.put(
            PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
```

```
        itemMap.put(SORT_KEY_ATTR,
 AttributeValue.builder().s(sortKeyValue).build());
        itemMap.put(
            CREATION_DATE_ATTR,
            AttributeValue.builder().n(String.valueOf(createDate)).build());
        itemMap.put(
            EXPIRE_AT_ATTR,
            AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final PutItemRequest request =
            PutItemRequest.builder().tableName(tableName).item(itemMap).build();

        try {
            final PutItemResponse response = dynamoDbClient.putItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「PutItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

export function createDynamoDBItem(table_name, region, partition_key, sort_key) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });
```

```
    // Get the current time in epoch second format
    const current_time = Math.floor(new Date().getTime() / 1000);

    // Calculate the expireAt time (90 days from now) in epoch second format
    const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 *
 1000) / 1000);

    // Create DynamoDB item
    const item = {
        'partitionKey': {'S': partition_key},
        'sortKey': {'S': sort_key},
        'createdAt': {'N': current_time.toString()},
        'expireAt': {'N': expire_at.toString()}
    };

    const putItemCommand = new PutItemCommand({
        TableName: table_name,
        Item: item,
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1,
        },
    });

    client.send(putItemCommand, function(err, data) {
        if (err) {
            console.log("Exception encountered when creating item %s, here's what
 happened: ", data, err);
            throw err;
        } else {
            console.log("Item created successfully: %s.", data);
            return data;
        }
    });
}

// Example usage (commented out for testing)
// createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
 'your-sort-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def create_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Creates a DynamoDB item with an attached expiry attribute.

    :param table_name: Table name for the boto3 resource to target when creating
 an item
    :param region: string representing the AWS region. Example: `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expiration time (90 days from now) in epoch second format
        expiration_time = int((datetime.now() + timedelta(days=90)).timestamp())

        item = {
            "primaryKey": primary_key,
            "sortKey": sort_key,
            "creationDate": current_time,
            "expireAt": expiration_time,
        }
        response = table.put_item(Item=item)

        print("Item created successfully.")
        return response
    except Exception as e:
        print(f"Error creating item: {e}")
        raise e
```

```
# Use your own values
create_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「PutItem」を参照してください。

項目を更新して Time to Live を再計算する

この例は、前のセクションの例の続きです。項目が更新された場合、有効期限を再計算できます。次の例では、expireAt タイムスタンプを現在の時刻から 90 日後として再計算します。

以下のコード例は、項目の TTL の更新方法を示しています。

Java

SDK for Java 2.x

テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

    public UpdateItemResponse updateItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
 sortKeyValue) {
        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
```

```
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = new HashMap<>();
        keyMap.put(PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
        keyMap.put(SORT_KEY_ATTR,
AttributeValue.builder().s(sortKeyValue).build());

        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":c",
AttributeValue.builder().n(String.valueOf(currentTime)).build());
        expressionAttributeValues.put(
            ":e",
AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final UpdateItemResponse response =
dynamoDbClient.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItem = async (tableName, partitionKey, sortKey, region = 'us-
east-1') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };

    try {
        const data = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(data.Attributes);
        console.log("Item updated successfully: %s", responseData);
        return responseData;
    } catch (err) {
        console.error("Error updating item:", err);
        throw err;
    }
}

// Example usage (commented out for testing)
// updateItem('your-table-name', 'your-partition-key-value', 'your-sort-key-
value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def update_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Update an existing DynamoDB item with a TTL.
    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        # Create the DynamoDB resource.
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expireAt time (90 days from now) in epoch second format
        expire_at = int((datetime.now() + timedelta(days=90)).timestamp())

        table.update_item(
            Key={"partitionKey": primary_key, "sortKey": sort_key},
            UpdateExpression="set updatedAt=:c, expireAt=:e",
            ExpressionAttributeValues={":c": current_time, ":e": expire_at},
        )

        print("Item updated successfully.")
```

```
    except Exception as e:
        print(f"Error updating item: {e}")


# Replace with your own values
update_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

この概要で説明した TTL の例は、最近更新された項目だけをテーブルに保持する方法を示しています。更新された項目は有効期限が延びますが、作成後に更新されなかった項目は有効期限が切れて無料で削除されるため、ストレージが削減され、テーブルが整理されます。

期限切れのアイテムと Time to Live (TTL) の使用

削除が保留になっている期限切れのアイテムは、読み取り操作と書き込み操作の対象から除外できます。これは、期限切れのデータが無効になり、使用すべきではなくなった場合に役に立ちます。フィルターが適用されていないと、バックグラウンドプロセスによって削除されるまで、読み取り/書き込み操作で引き続き表示されます。

> **Note**
>
> これらのアイテムは、削除されるまではストレージコストと読み取りコストにカウントされます。

TTL 削除は DynamoDB ストリームで確認できますが、削除が行われたリージョンでのみ確認できます。グローバルテーブルリージョンにレプリケートされた TTL 削除は、削除がレプリケートされたリージョンの DynamoDB ストリームでは確認できません。

期限切れの項目を読み取り操作から除外する

Scan や Query などの読み取り操作では、フィルター式を使用して削除待ちの期限切れアイテムを除外できます。次のコードスニペットに示すように、フィルター式は TTL 時間が現在の時刻と同等ま

たは未満の項目を除外できます。例えば、Python SDK コードには、現在の時刻を変数 (now) として取得し、それをエポック時間形式の `int` に変換する代入ステートメントが含まれています。

以下のコード例は、TTL 項目のクエリの開始方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.Map;
import java.util.Optional;

        final QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final QueryResponse response = ddb.query(request);
            System.out.println("Query successful. Found " + response.count() + "
 items that have not expired yet.");

            // Print each item
            response.items().forEach(item -> {
                System.out.println("Item: " + item);
            });
```

```
            return 0;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const queryFiltered = async (tableName, primaryKey, region = 'us-east-1')
 => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pk",
        FilterExpression: "#ea > :ea",
        ExpressionAttributeNames: {
            "#pk": "primaryKey",
            "#ea": "expireAt"
        },
        ExpressionAttributeValues: marshall({
            ":pk": primaryKey,
```

```
            ":ea": currentTime
        })
    };

    try {
        const { Items } = await client.send(new QueryCommand(params));
        Items.forEach(item => {
            console.log(unmarshall(item))
        });
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

// Example usage (commented out for testing)
// queryFiltered('your-table-name', 'your-partition-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
from datetime import datetime

import boto3


def query_dynamodb_items(table_name, partition_key):
    """

    :param table_name: Name of the DynamoDB table
    :param partition_key:
    :return:
    """
```

```
    try:
        # Initialize a DynamoDB resource
        dynamodb = boto3.resource("dynamodb", region_name="us-east-1")

        # Specify your table
        table = dynamodb.Table(table_name)

        # Get the current time in epoch format
        current_time = int(datetime.now().timestamp())

        # Perform the query operation with a filter expression to exclude expired
 items
        # response = table.query(
        #
 KeyConditionExpression=boto3.dynamodb.conditions.Key('partitionKey').eq(partition_key),
        #
 FilterExpression=boto3.dynamodb.conditions.Attr('expireAt').gt(current_time)
        # )
        response = table.query(

 KeyConditionExpression=dynamodb.conditions.Key("partitionKey").eq(partition_key),

 FilterExpression=dynamodb.conditions.Attr("expireAt").gt(current_time),
        )

        # Print the items that are not expired
        for item in response["Items"]:
            print(item)

    except Exception as e:
        print(f"Error querying items: {e}")


# Call the function with your values
query_dynamodb_items("Music", "your-partition-key-value")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

期限切れのアイテムに条件付きで書き込む

条件式を使用すると、期限切れのアイテムに対する書き込みを回避できます。以下のコードスニペットは、有効期限が現在の時間よりも大きいかどうかをチェックする条件付き更新です。true の場合、書き込みオペレーションは続行されます。

以下のコード例は、項目の TTL を条件付きで更新する方法を示しています。

Java

SDK for Java 2.x

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Map;
import java.util.Optional;

/**
 * Updates an item in a DynamoDB table with TTL attributes using a conditional
 expression.
 * This class demonstrates how to conditionally update TTL expiration timestamps.
 */
public class UpdateTTLConditional {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
```

```
                primaryKey - The name of the primary key. Also known as the hash
or partition key.
                sortKey - The name of the sort key. Also known as the range
attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String UPDATED_AT_ATTR = "updatedAt";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String UPDATE_EXPRESSION = "SET " + UPDATED_AT_ATTR +
"=:c, " + EXPIRE_AT_ATTR + "=:e";
    private static final String CONDITION_EXPRESSION = EXPIRE_AT_ATTR + " > :c";
    private static final String SUCCESS_MESSAGE = "%s UpdateItem operation with
TTL successful.";
    private static final String CONDITION_FAILED_MESSAGE = "Condition check
failed. The item may have already expired.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs an UpdateTTLConditional with a default DynamoDB client.
     */
    public UpdateTTLConditional() {
        this.dynamoDbClient = null;
    }

    /**
     * Constructs an UpdateTTLConditional with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public UpdateTTLConditional(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Main method to demonstrate conditionally updating an item with TTL.
     *
```

```
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new UpdateTTLConditional().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and conditionally update an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = Map.of(
            PRIMARY_KEY_ATTR, AttributeValue.builder().s(primaryKey).build(),
            SORT_KEY_ATTR, AttributeValue.builder().s(sortKey).build());
```

```
        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = Map.of(
            ":c",
 AttributeValue.builder().n(String.valueOf(currentTime)).build(),
            ":e",
 AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .conditionExpression(CONDITION_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final UpdateItemResponse response = ddb.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return 0;
        } catch (ConditionalCheckFailedException e) {
            System.err.println(CONDITION_FAILED_MESSAGE);
            throw e;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItemConditional = async (tableName, partitionKey, sortKey,
 region = 'us-east-1', newAttribute = 'default-value') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            artist: partitionKey,
            album: sortKey
        }),
        UpdateExpression: "SET newAttribute = :newAttribute",
        ConditionExpression: "expireAt > :expiration",
        ExpressionAttributeValues: marshall({
            ':newAttribute': newAttribute,
            ':expiration': currentTime
        }),
        ReturnValues: "ALL_NEW"
    };

    try {
        const response = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(response.Attributes);
        console.log("Item updated successfully: ", responseData);
        return responseData;
    } catch (error) {
        if (error.name === "ConditionalCheckFailedException") {
            console.log("Condition check failed: Item's 'expireAt' is expired.");
        } else {
            console.error("Error updating item: ", error);
        }
```

```
        throw error;
    }
};

// Example usage (commented out for testing)
// updateItemConditional('your-table-name', 'your-partition-key-value', 'your-
sort-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
from datetime import datetime, timedelta

import boto3
from botocore.exceptions import ClientError


def update_dynamodb_item_ttl(table_name, region, primary_key, sort_key,
 ttl_attribute):
    """
    Updates an existing record in a DynamoDB table with a new or updated TTL
 attribute.

    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :param ttl_attribute: name of the TTL attribute in the target DynamoDB table
    :return:
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Generate updated TTL in epoch second format
```

```
        updated_expiration_time = int((datetime.now() +
 timedelta(days=90)).timestamp())

        # Define the update expression for adding/updating a new attribute
        update_expression = "SET newAttribute = :val1"

        # Define the condition expression for checking if 'expireAt' is not
 expired
        condition_expression = "expireAt > :val2"

        # Define the expression attribute values
        expression_attribute_values = {":val1": ttl_attribute, ":val2":
 updated_expiration_time}

        response = table.update_item(
            Key={"primaryKey": primary_key, "sortKey": sort_key},
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeValues=expression_attribute_values,
        )

        print("Item updated successfully.")
        return response["ResponseMetadata"]["HTTPStatusCode"]  # Ideally a 200 OK
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            print("Condition check failed: Item's 'expireAt' is expired.")
        else:
            print(f"Error updating item: {e}")
    except Exception as e:
        print(f"Error updating item: {e}")


# replace with your values
update_dynamodb_item_ttl(
    "your-table-name",
    "us-east-1",
    "your-partition-key-value",
    "your-sort-key-value",
    "your-ttl-attribute-value",
)
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

DynamoDB ストリーム内の削除済みアイテムの特定

ストリームレコードにはユーザー ID フィールド `Records[<index>].userIdentity` が含まれます。TTL プロセスによって削除された項目には、次のフィールドが含まれています。

```
Records[<index>].userIdentity.type
"Service"

Records[<index>].userIdentity.principalId
"dynamodb.amazonaws.com"
```

次の JSON は 1 つのストリームレコードの関連する部分を示しています。

```
"Records": [
  {
 ...
  "userIdentity": {
  "type": "Service",
       "principalId": "dynamodb.amazonaws.com"
    }
   ...
 }
]
```

DynamoDB のテーブルに対するクエリの実行

Amazon DynamoDB の `Query` オペレーションを使用して、プライマリキーの値に基づいて項目を探すことができます。

パーティションキーの属性名、および属性の単一値を指定する必要があります。`Query` はそのパーティションキー値を持つすべての項目を返します。必要に応じて、ソートキーの属性を指定し、比較演算子を使用して、検索結果をさらに絞り込むことができます。

リクエストの構文、レスポンスパラメータ、その他の例など、`Query` の使用方法の詳細については、Amazon DynamoDB API リファレンスの「クエリ」を参照してください。

トピック

- DynamoDB でのクエリオペレーションのキー条件式
- DynamoDB でのクエリオペレーションのフィルター式
- DynamoDB でテーブルクエリ結果をページ分割する
- DynamoDB でクエリオペレーションを使用する際のその他の側面

DynamoDB でのクエリオペレーションのキー条件式

最初の文字が a-z または A-Z であり、残りの文字 (ある場合、2 番目以降の文字) が a-z、A-Z、または 0-9 である場合は、キー条件式で任意の属性値を使用できます。さらに、属性名は DynamoDB の予約語であってはいけません。(詳細リストについては、「DynamoDB の予約語」を参照してください) 属性名がこれらの要件を満たさない場合は、式の属性名をプレースホルダーとして定義する必要があります。詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

特定のパーティションキー値を持つ項目は、DynamoDB によってソートキーの値で並べ替えられた順序で近くに配置されて保存されます。Query オペレーションでは、並べ替えられた順序でDynamoDB によって項目が取得され、KeyConditionExpression や存在する任意の FilterExpression を使用して処理されます。その時初めて、Query の結果がクライアントに返されます。

Query オペレーションは常に結果セットを返します。一致する項目がない場合、結果セットは空になります。

Query の結果は常にソートキーの値によってソートされます。ソートキーのデータ型が Number である場合は、結果が番号順で返されます。その他の場合は、UTF-8 バイトの順序で結果が返されます。デフォルトの並べ替え順序は昇順です。順序を反転させるには、ScanIndexForward パラメータを false に設定します。

1 回の Query オペレーションで、最大 1 MB のデータを取得できます。この制限は、FilterExpression または ProjectionExpression が結果に適用される前に適用されます。レスポンスに LastEvaluatedKey が存在し、Null 以外の場合、結果セットをページ分割する必要があります (DynamoDB でテーブルクエリ結果をページ分割する を参照)。

キー条件式の例

検索条件を指定するには、キー条件式 (テーブルまたはインデックスから読み取る項目を決定する文字列) を使用します。

等価条件としてパーティションキーの名前と値を指定する必要があります。キー条件式では、非キー属性を使用することはできません。

オプションで、ソートキーに 2 番目の条件を指定できます (存在する場合)。ソートキーの条件では、次の比較演算子を 1 つ使用する必要があります。

- *a* = *b* — 属性 *a* が値 *b* と等しい場合、true
- *a* < *b* — *a* が *b* 未満の場合、true
- *a* <= *b* — *a* が *b* 以下である場合、true
- *a* > *b* — *a* が *b* より大きい場合、true
- *a* >= *b* — *a* が *b* 以上である場合、true
- *a* BETWEEN *b* AND *c* — *a* が *b* 以上で、*c* 以下である場合、true。

次の関数もサポートされます。

- begins_with (*a*, *substr*) — 属性の値 *a* が特定のサブ文字列から始まる場合、true。

キー条件式の使用方法を示す AWS Command Line Interface (AWS CLI) の例を次に示します。これらの式では、実際の値の代わりにプレースホルダー (:name や :sub など) を使用しています。詳細については、「DynamoDB の式の属性名 (エイリアス)」および「DynamoDB での式属性値の使用」を参照してください。

Example

Thread テーブルに対して、特定の ForumName (パーティションキー) についてのクエリを実行します。その ForumName の値を持つすべての項目はクエリによって読み込まれます。これはソートキー (Subject) が KeyConditionExpression に含まれないためです。

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :name" \
    --expression-attribute-values  '{":name":{"S":"Amazon DynamoDB"}}'
```

Example

Thread テーブルに対して、特定の ForumName (パーティションキー) についてのクエリを実行しますが、今回は指定の Subject (ソートキー) を持つ項目のみを返します。

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :name and Subject = :sub" \
    --expression-attribute-values  file://values.json
```

--expression-attribute-values の引数は、values.json のファイルに保存されます。

```
{
    ":name":{"S":"Amazon DynamoDB"},
    ":sub":{"S":"DynamoDB Thread 1"}
}
```

Example

Reply テーブルに対して、特定の Id (パーティションキー) についてのクエリを実行しますが、ReplyDateTime (ソートキー) が特定の文字で始まる項目のみを返します。

```
aws dynamodb query \
    --table-name Reply \
    --key-condition-expression "Id = :id and begins_with(ReplyDateTime, :dt)" \
    --expression-attribute-values  file://values.json
```

--expression-attribute-values の引数は、values.json のファイルに保存されます。

```
{
    ":id":{"S":"Amazon DynamoDB#DynamoDB Thread 1"},
    ":dt":{"S":"2015-09"}
}
```

DynamoDB でのクエリオペレーションのフィルター式

Query 結果の絞り込みが必要な場合は、オプションでフィルタ式を指定できます。フィルタ式によって、Query 結果の内、どの項目を返すべきか確定します。他のすべての結果は破棄されます。

フィルタ式は、Query の完了後、結果が返される前に適用されます。そのため、Query は、フィルタ式があるかどうかにかかわらず、同じ量の読み込みキャパシティーを消費します。

1 回の Query オペレーションで、最大 1 MB のデータを取得できます。この制限は、フィルタ式を評価する前に適用されます。

フィルタ式には、パーティションキーまたはソートキーの属性を含めることはできません。フィルタ式ではなく、キー条件式でこれらの属性を指定する必要があります。

フィルター式の構文は、キー条件式の構文と似ています。フィルター式は、キー条件式と同じコンパレータ、関数および論理演算子を使用できます。さらに、フィルター式では、非等号演算子 (<>)、OR 演算子、CONTAINS 演算子、IN 演算子、BEGINS_WITH 演算子、BETWEEN 演算子、EXISTS 演算子、および SIZE 演算子を使用できます。詳細については、「DynamoDB でのクエリオペレーションのキー条件式」および「フィルター式と条件式の構文」を参照してください。

Example

以下の AWS CLI の例では、Thread テーブルに対して、特定の ForumName (パーティションキー) および Subject (ソートキー) についてのクエリを実行します。見つかった項目のうち、最も一般的なディスカッションスレッド (つまり、一定数以上の Views があるスレッド) のみが返されます。

```
aws dynamodb query \
    --table-name Thread \
    --key-condition-expression "ForumName = :fn and Subject begins_with :sub" \
    --filter-expression "#v >= :num" \
    --expression-attribute-names '{"#v": "Views"}' \
    --expression-attribute-values file://values.json
```

--expression-attribute-values の引数は、values.json のファイルに保存されます。

```
{
    ":fn":{"S":"Amazon DynamoDB"},
    ":sub":{"S":"DynamoDB Thread 1"},
    ":num":{"N":"3"}
}
```

Views は DynamoDB で予約語であるため (DynamoDB の予約語 を参照)、この例では #v をプレースホルダーとして使用することにご注意ください。詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

> **Note**
>
> フィルタ式は、Query 結果セットから項目を削除します。可能であれば、大量の項目を取得してもそのほとんどを破棄する必要がある場合、 Query の使用は避けてください。

DynamoDB でテーブルクエリ結果をページ分割する

DynamoDB では、Query オペレーションの結果をページ割りします。ページ分割を行うことで Query 結果が 1 MB サイズ (またはそれ以下) のデータの「ページ」に分割されます。アプリケーションは結果の最初のページ、次に 2 ページと処理できます。

1 つの Query は、サイズの制限である1 MB 以内の結果セットだけを返します。さらに結果があるかどうかを確認して、一度に 1 ページずつ結果を取り出すには、アプリケーションで次の操作を行う必要があります。

1. 低レベルの Query 結果を確認します。
   - 結果に LastEvaluatedKey 要素が含まれており、それが Null 以外の場合、ステップ 2 に進みます。
   - 結果に LastEvaluatedKey がない場合、これ以上取得する項目はありません。
2. 同じ KeyConditionExpression を使用して Query を構築します。ただし、今回は、ステップ 1 から LastEvaluatedKey 値を取得して、新しい Query リクエストの ExclusiveStartKey パラメータとして使用します。
3. 新しい Query リクエストを実行します。
4. ステップ 1 に進んでください。

言い換えると、Query レスポンスからの LastEvaluatedKey を次の Query リクエストの ExclusiveStartKey として使用する必要があります。Query レスポンスに LastEvaluatedKey の要素がない場合、結果の最後のページを取得します。LastEvaluatedKey が空ではない場合でも、必ずしも結果セットにまだ値があることを意味するわけではありません。結果セットの最後まで到達したことを確認できるのは、LastEvaluatedKey が空になったときだけです。

空でない LastEvaluatedKey は、単に前回の Query が 1 MB のページサイズ制限または Limit 値であるページ境界で停止したことを意味するだけで、一致する項目が必ずしもまだ残っていることを意味するわけではありません。特に、FilterExpression を使用する場合、Query はフィルターを適用する前に読み取る項目に 1 MB または Limit のページ上限を適用します。これにより、ページ内に条件に一致する項目が 1 つもなくても、LastEvaluatedKey が返される可能性があります。返された LastEvaluatedKey を使用して、レスポンスにそれが含まれなくなるまで、リクエストを発行し続けます。

AWS CLI を使用してこの動作を表示できます。AWS CLI は、LastEvaluatedKey が結果に表示されなくなるまで、低レベルの Query リクエストを DynamoDB に繰り返し送信します。特定の年の映画タイトルを取得する次の AWS CLI の例を考えてみます。

```
aws dynamodb query --table-name Movies \
    --projection-expression "title" \
    --key-condition-expression "#y = :yyyy" \
    --expression-attribute-names '{"#y":"year"}' \
    --expression-attribute-values '{":yyyy":{"N":"1993"}}' \
    --page-size 5 \
    --debug
```

通常、AWS CLI はページ分割を自動的に処理します。ただし、この例では、AWS CLI `--page-size` パラメータによりページごとの項目数が制限されています。`--debug` パラメータは、リクエストとレスポンスについての低レベルの情報を表示します。

> **Note**
>
> 自動ページ分割は、AWS CLI バージョン 1 とバージョン 2 の両方のデフォルトの動作であるため、1 つのコマンドで、一致するすべての項目が返されます。ユーザー自身でページを分割する必要がある場合は、`--no-paginate` オプションを使用して最初のページのみを返すか (さらに多くの項目が残っている場合、レスポンスには `LastEvaluatedKey` が含まれます)、`--max-items` を使用して返される項目の数を制限し、次のページを取得するための `NextToken` 値を取得します。

この例を実行した場合、DynamoDB からの最初のレスポンスは次のようになります。

```
2017-07-07 11:13:15,603 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":5,"Items":[{"title":{"S":"A Bronx Tale"}},
{"title":{"S":"A Perfect World"}},{"title":{"S":"Addams Family Values"}},
{"title":{"S":"Alive"}},{"title":{"S":"Benny & Joon"}}],
"LastEvaluatedKey":{"year":{"N":"1993"},"title":{"S":"Benny & Joon"}},
"ScannedCount":5}'
```

レスポンスの `LastEvaluatedKey` は、すべての項目が取得されたわけではないことを示します。その後、AWS CLI は DynamoDB に別の `Query` リクエストを送信します。このリクエストとレスポンスのパターンが、最終レスポンスまで繰り返されます。

```
2017-07-07 11:13:16,291 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":1,"Items":[{"title":{"S":"What\'s Eating Gilbert
 Grape"}}],"ScannedCount":1}'
```

LastEvaluatedKey がない場合、これ以上取得する項目がないことを示します。

> **Note**
>
> AWS SDK は低レベルの DynamoDB レスポンス (LastEvaluatedKey の有無を含む) を処理し、ページ割りした Query 結果のさまざまな抽象化を提供します。たとえば、SDK for Java のドキュメントインターフェイスでは、java.util.Iterator サポートが利用可能なため、結果を一度に 1 つずつ確認できます。
> 各種のプログラミング言語のコード例については、「Amazon DynamoDB 利用開始ガイド」と、該当言語の「AWS SDK ドキュメント」を参照してください。

Query オペレーションの Limit パラメータを使用して結果セット内の項目数を制限することで、ページサイズを小さくすることもできます。

DynamoDB でのクエリ実行の詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

DynamoDB でクエリオペレーションを使用する際のその他の側面

このセクションでは、DynamoDB クエリオペレーションのその他の側面について説明します。これには、結果サイズの制限、スキャンされた項目と返された項目のカウント、読み取り容量の消費のモニタリング、読み取り整合性の制御が含まれます。

結果セットの項目数の制限

Query オペレーションを使用すると、読み取られる項目数を制限することができます。これを行うには、Limit パラメータに項目の最大数を設定します。

たとえば、フィルタ式を使用せず、Limit 値を 6 として、テーブルを Query するとします。Query 結果には、リクエストのキー条件式に一致するテーブルからの最初 6 つの項目が含まれます。

ここで、Query にフィルタ式を追加するとします。この場合、DynamoDB は最大 6 つの項目を読み込み、フィルタ式と一致する項目だけを返します。DynamoDB が項目の読み込みを続けた場合、さらに多くの項目がフィルタ式にマッチしても、最終的な Query 結果に含まれる項目数は 6 つ以下です。

結果での項目のカウント

Query レスポンスには、条件に一致する項目に加えて次の要素が含まれます。

- ScannedCount — フィルター式 (存在する場合) が適用される前に、キー条件式に一致した項目数。
- Count — フィルター式 (存在する場合) が適用された後に残っている項目数。

> **Note**
>
> フィルタ式を使用しない場合、ScannedCount と Count は同じ値を持ちます。

Query 結果セットのサイズが 1 MB より大きい場合、ScannedCount および Count では、合計項目数の一部のみが示されます。すべての結果を取得するためには、複数の Query オペレーションを実行する必要があります (DynamoDB でテーブルクエリ結果をページ分割する を参照してください)。

各 Query レスポンスには、特定の Query リクエストによって処理された項目の ScannedCount および Count が含まれます。すべての Query リクエストの合計を取得するには、ScannedCount および Count の実行中の集計を維持することができます。

クエリで消費されるキャパシティユニット

パーティションキーの属性名、およびその属性の単一値を指定する限り、いずれのテーブルまたはセカンダリインデックスにも、Query を実行できます。Query により、そのパーティションのキー値を持つすべての項目が返されます。必要に応じて、ソートキーの属性を指定し、比較演算子を使用して、検索結果をさらに絞り込むことができます。QueryAPI オペレーションでは、次のように読み込みキャパシティーユニットを消費します。

...を **Query** する場合	DynamoDB は ... からの読み込み容量ユニットを消費します。
テーブル	テーブルのプロビジョニングされた読み込みキャパシティー。
グローバルセカンダリインデックス	インデックスのプロビジョニングされた読み込みキャパシティー。
ローカルセカンダリインデックス	ベーステーブルのプロビジョニングされた読み込みキャパシティー。

デフォルトでは、Queryオペレーションはどのくらいの読み込みキャパシティーを消費するかについてのデータを返しません。ただし、この情報を取得するために Query リクエストで ReturnConsumedCapacity パラメータを指定できます。ReturnConsumedCapacity の有効な設定を以下に示します。

- NONE — 消費された容量データは返されません。(これがデフォルトです)
- TOTAL — レスポンスには消費された読み込み容量単位の合計値が含まれます。
- INDEXES — レスポンスは、アクセスする各テーブルとインデックスの消費される容量とともに、消費される読み込み容量単位の合計値を示します。

DynamoDB はアプリケーションに返されるデータ量ではなく、項目の数と項目のサイズに基づいて、消費される読み込みキャパシティユニットの数を計算します。このため、消費される容量ユニットの数は、(デフォルトの動作で) 属性のすべてをリクエストしても、(プロジェクション式を使用して) 一部をリクエストしても、同じになります。この数は、フィルター式を使用していてもいなくても同じです。Query は最小読み込みキャパシティユニットを消費して、最大 4 KB の項目について強力な整合性のある読み込みを 1 秒あたり 1 回、または結果整合性のある読み込みを 1 秒あたり 2 回実行します。4 KB より大きい項目を読み込む必要がある場合、DynamoDB には追加の読み込みリクエストユニットが必要です。空のテーブルや、パーティションキーの数が少ない非常に大きなテーブルでは、クエリされたデータの量を超えて追加の RCU が課金される場合があります。これにより、データが存在しない場合でも、Query リクエストを処理するためのコストがカバーされます。

クエリの読み込み整合性

Query オペレーションは、結果的に整合性のある読み込みをデフォルトで行います。つまり、Query 結果が、最近完了した PutItem または UpdateItem オペレーションによる変更を反映しない場合があります。詳細については、「DynamoDB の読み取り整合性」を参照してください。

強力な整合性のある読み込みが必要な場合は、Query リクエストで ConsistentRead パラメータを true に設定します。

DynamoDB でのテーブルのスキャン

Amazon DynamoDB の Scan オペレーションでは、テーブルまたはセカンダリインデックスのすべての項目を読み込みます。デフォルトでは、Scan オペレーションはテーブルまたはインデックスのすべての項目のデータ属性を返します。ProjectionExpression パラメータを使用し、Scan がすべての属性ではなく一部のみを返すようにできます。

Scan は常に結果セットを返します。一致する項目がない場合、結果セットは空になります。

1 回の Scan リクエストで、最大 1 MB のデータを取得できます。DynamoDB では、必要に応じてこのデータにフィルター式を適用して、結果をユーザーに返す前に絞り込むことができます。

> **Note**
>
> Scan は、テーブルまたはセカンダリインデックスが項目を内部に保存する方法によって決定される順序で項目を返します。パーティションキー値は、ソートされていない任意の順序で返されます。1 つのパーティションキー値の中で、項目はソートキー値を基準にして昇順 (ASC) で返されます。

トピック

- Scan のフィルタ式
- 結果セットの項目数の制限
- 結果のページ分割
- 結果での項目のカウント
- Scan で消費されるキャパシティユニット
- スキャンの読み込み整合性
- 並列スキャン

Scan のフィルタ式

Scan 結果の絞り込みが必要な場合は、オプションでフィルタ式を指定できます。フィルタ式によって、Scan 結果の内、どの項目を返すべきか確定します。他のすべての結果は破棄されます。

フィルタ式は、Scan の完了後、結果が返される前に適用されます。そのため、Scan は、フィルタ式があるかどうかにかかわらず、同じ量の読み込みキャパシティーを消費します。

1 回の Scan オペレーションで、最大 1 MB のデータを取得できます。この制限は、フィルタ式を評価する前に適用されます。

Scan では、パーティションキー属性やソートキー属性など、フィルター式で任意の属性を指定できます。

フィルタ式の構文は、条件式の構文と同じです。フィルタ式は、条件式と同じコンパレータ、関数および論理演算子を使用できます。論理演算子の詳細については、「DynamoDB の条件式とフィルター式、演算子、関数」を参照してください。

Example

次の AWS Command Line Interface (AWS CLI) の例では Thread テーブルをスキャンして、特定のユーザーによって最後に投稿された項目のみを返します。

```
aws dynamodb scan \
     --table-name Thread \
     --filter-expression "LastPostedBy = :name" \
     --expression-attribute-values '{":name":{"S":"User A"}}'
```

結果セットの項目数の制限

Scan オペレーションは、結果で返される項目数を制限することができます。これを行うには、フィルタ式を評価する前に、Limit パラメータに、Scan オペレーションが返す項目の最大数を設定します。

たとえば、フィルタ式を使用せず、Scan 値を Limit として、テーブルを 6 するとします。Scan 結果には、テーブルの最初の 6 つの項目が含まれます。

ここで、Scan にフィルタ式を追加するとします。この場合、DynamoDB は返される 6 つの項目にフィルター式を適用し、一致しない項目を廃棄します。最終的な Scan 結果はフィルタリングされる項目の数に応じて、6 つ以下の項目を含みます。

結果のページ分割

DynamoDB では、Scan オペレーションの結果をページ割りします。ページ分割を行うことで Scan 結果が 1 MB サイズ (またはそれ以下) のデータの「ページ」に分割されます。アプリケーションは結果の最初のページ、次に 2 ページと処理できます。

1 つの Scan は、サイズの制限である1 MB 以内の結果セットだけを返します。

さらに結果があるかどうかを確認して、一度に 1 ページずつ結果を取り出すには、アプリケーションで次の操作を行う必要があります。

1. 低レベルの Scan 結果を確認します。
   - 結果に LastEvaluatedKey 要素が含まれる場合、ステップ 2 に進みます。
   - 結果に LastEvaluatedKey がない場合、これ以上取得する項目はありません。
2. 以前のものと同じパラメータを使用して新しい Scan リクエストを作成します。ただし、今回は、ステップ 1 から LastEvaluatedKey 値を取得して、新しい Scan リクエストの ExclusiveStartKey パラメータとして使用します。

3. 新しい Scan リクエストを実行します。
4. ステップ 1 に進んでください。

言い換えると、LastEvaluatedKey レスポンスからの Scan を次の ExclusiveStartKey リクエストの Scan として使用する必要があります。LastEvaluatedKey レスポンスに Scan 要素がない場合、結果の最後のページを取得します。(結果セットの最後まで到達したことを確認できるのは、LastEvaluatedKey がないときだけです)

AWS CLI を使用してこの動作を表示できます。AWS CLI は低レベル Scan リクエストを DynamoDB に送信し、LastEvaluatedKey が結果に表示されなくなるまで送信を繰り返します。次の AWS CLI の例を見てください。この例は、Movies テーブル全体をスキャンしますが、特定のジャンルの映画のみを返します。

```
aws dynamodb scan \
    --table-name Movies \
    --projection-expression "title" \
    --filter-expression 'contains(info.genres,:gen)' \
    --expression-attribute-values '{":gen":{"S":"Sci-Fi"}}' \
    --page-size 100  \
    --debug
```

通常、AWS CLI はページ分割を自動的に処理します。ただし、この例では、AWS CLI --page-size パラメータによりページごとの項目数が制限されています。--debug パラメータは、リクエストとレスポンスについての低レベルの情報を表示します。

> **Note**
>
> ページ分割の結果は、渡した入力パラメータによっても異なります。
>
> - aws dynamodb scan --table-name Prices --max-items 1 を使用すると NextToken が返されます
> - aws dynamodb scan --table-name Prices --limit 1 を使用すると LastEvaluatedKey が返されます。
>
> また、特に --starting-token を使用するには、NextToken 値が必要であることに注意してください。

この例を実行した場合、DynamoDB からの最初のレスポンスは次のようになります。

```
2017-07-07 12:19:14,389 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":7,"Items":[{"title":{"S":"Monster on the Campus"}},{"title":{"S":"+1"}},
{"title":{"S":"100 Degrees Below Zero"}},{"title":{"S":"About Time"}},{"title":
{"S":"After Earth"}},
{"title":{"S":"Age of Dinosaurs"}},{"title":{"S":"Cloudy with a Chance of Meatballs
 2"}}],
"LastEvaluatedKey":{"year":{"N":"2013"},"title":{"S":"Curse of
 Chucky"}},"ScannedCount":100}'
```

レスポンスの LastEvaluatedKey は、すべての項目が取得されたわけではないことを示します。その後、AWS CLI は DynamoDB に別の Scan リクエストを送信します。このリクエストとレスポンスのパターンが、最終レスポンスまで繰り返されます。

```
2017-07-07 12:19:17,830 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":1,"Items":[{"title":{"S":"WarGames"}}],"ScannedCount":6}'
```

LastEvaluatedKey がない場合、これ以上取得する項目がないことを示します。

> **Note**
>
> AWS SDK は低レベルの DynamoDB レスポンス (LastEvaluatedKey の有無を含む) を処理し、ページ割りした Scan 結果のさまざまな抽象化を提供します。たとえば、SDK for Java のドキュメントインターフェイスでは、java.util.Iterator サポートが利用可能なため、結果を一度に 1 つずつ確認できます。
> 各種のプログラミング言語のコード例については、Amazon DynamoDB 利用開始ガイドと、該当言語の AWS SDK ドキュメントを参照してください。

結果での項目のカウント

Scan レスポンスには、条件に一致する項目に加えて次の要素が含まれます。

- ScannedCount — ScanFilter が適用される前に評価される項目数。ScannedCount 値が大きく、Count 結果が小さいまたはない場合は、Scan オペレーションが不十分であることを示しています。リクエストでフィルタを使用していない場合、ScannedCount は Count と同じです。
- Count — フィルター式 (存在する場合) が適用された後に残っている項目数。

> **Note**
>
> フィルタ式を使用しない場合、ScannedCount と Count は同じ値を持ちます。

Scan 結果セットのサイズが 1 MB より大きい場合、ScannedCount および Count では、合計項目数の一部のみが示されます。すべての結果を取得するためには、複数の Scan オペレーションを実行する必要があります (結果のページ分割 を参照してください)。

各 Scan レスポンスには、特定の ScannedCount によって処理された項目の Count および Scan が含まれます。すべての Scan リクエストの合計を取得するには、ScannedCount および Count の実行中の集計を維持することができます。

Scan で消費されるキャパシティユニット

任意のテーブルまたはセカンダリインデックスで Scan できます。Scan オペレーションでは、次のように読み込み容量単位を消費します。

...を **Scan** する場合	DynamoDB は ... からの読み込み容量ユニットを消費します。
テーブル	テーブルのプロビジョニングされた読み込みキャパシティー。
グローバルセカンダリインデックス	インデックスのプロビジョニングされた読み込みキャパシティー。
ローカルセカンダリインデックス	ベーステーブルのプロビジョニングされた読み込みキャパシティー。

> **Note**
>
> セカンダリインデックススキャンオペレーションのクロスアカウントアクセスは、現在、リソースベースのポリシーではサポートされていません。

デフォルトでは、Scanオペレーションはどのくらいの読み込みキャパシティーを消費するかについてのデータを返しません。ただし、この情報を取得するために Scan リクエストで

ReturnConsumedCapacity パラメータを指定できます。ReturnConsumedCapacity の有効な設定を以下に示します。

- NONE — 消費された容量データは返されません。(これがデフォルトです)
- TOTAL — レスポンスには消費された読み込み容量単位の合計値が含まれます。
- INDEXES — レスポンスは、アクセスする各テーブルとインデックスの消費される容量とともに、消費される読み込み容量単位の合計値を示します。

DynamoDB はアプリケーションに返されるデータ量ではなく、項目の数と項目のサイズに基づいて、消費される読み込みキャパシティユニットの数を計算します。このため、消費される容量ユニットの数は、(デフォルトの動作で) 属性のすべてをリクエストしても、(プロジェクション式を使用して) 一部をリクエストしても、同じになります。この数は、フィルター式を使用していてもいなくても同じです。Scan は最小読み込みキャパシティユニットを消費して、最大 4 KB の項目について強力な整合性のある読み込みを 1 秒あたり 1 回、または結果整合性のある読み込みを 1 秒あたり 2 回実行します。4 KB より大きい項目を読み込む必要がある場合、DynamoDB には追加の読み込みリクエストユニットが必要です。空のテーブルや、パーティションキーの数が少ない非常に大きなテーブルでは、スキャンされたデータ量を超える追加の RCU が課金される場合があります。これにより、データが存在しない場合でも、Scan リクエストを処理するためのコストがカバーされます。

スキャンの読み込み整合性

Scan オペレーションは、結果的に整合性のある読み込みをデフォルトで行います。つまり、Scan 結果が、最近完了した PutItem または UpdateItem オペレーションによる変更を反映しない場合があります。詳細については、「DynamoDB の読み取り整合性」を参照してください。

強力な整合性のある読み込みが必要な場合は、Scan が開始する時に ConsistentRead パラメータを true リクエストで Scan に設定できます。これにより、Scan が開始する前に完了した書き込みオペレーションがすべて Scan 応答に含められます。

ConsistentRead を true に設定し、DynamoDB Streams と同時に使用すると、テーブルのバックアップまたはレプリケーションシナリオで役立ちます。最初に、テーブル内のデータの整合性のあるコピーを取得するため、Scan を true に設定して ConsistentRead を使用します。Scan の実行中、DynamoDB Streams はテーブルで発生した追加の書き込みアクティビティをすべて記録します。Scan が完了したら、ストリームからテーブルへの書き込みアクティビティを適用できます。

> **Note**
>
> Scan を ConsistentRead に設定した true オペレーションでは、ConsistentRead をデフォルト値 (false) のままにした場合と比較して、2 倍の読み込みキャパシティーユニットが使用されます。

並列スキャン

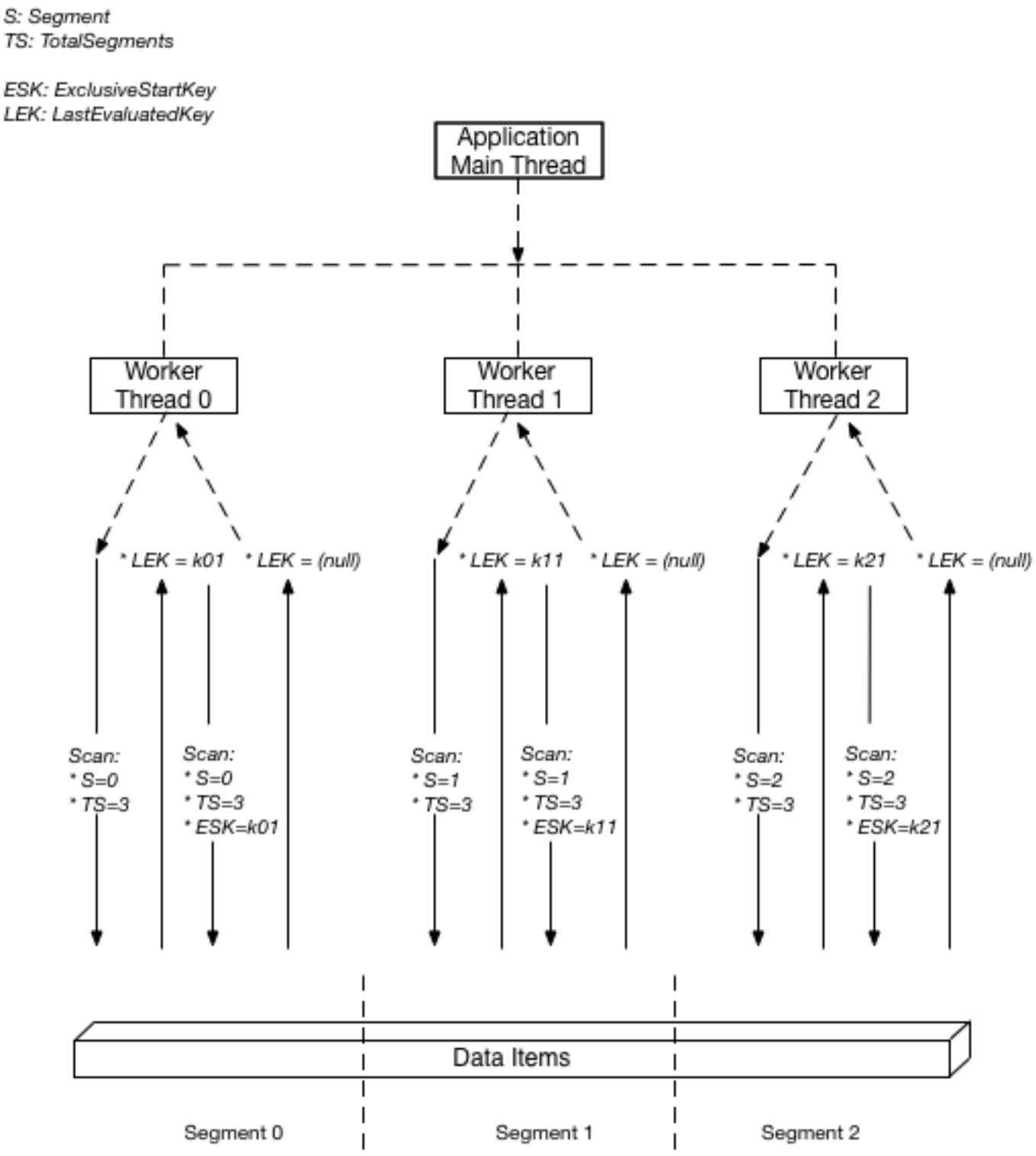
デフォルトでは、Scan オペレーションは、データを順次処理します。Amazon DynamoDB はアプリケーションに 1 MB 単位でデータを返し、アプリケーションは追加の Scan オペレーションを使用して、次の 1 MB のデータを取得できます。

スキャンするテーブルまたはインデックスが大きいほど、Scan を完了するのに時間がかかります。さらに、シーケンシャル Scan は、プロビジョンされた読み込みスループットキャパシティを常に完全に使用できるとは限りません。DynamoDB は大きなテーブルのデータを複数の物理パーティションに分散しますが、Scan オペレーションでは、一度に 1 つのパーティションしか読み込むことができません。このため、Scan のスループットは、単一のパーティションの最大スループットによって制約されます。

これらの問題に対処するために、Scan オペレーションでは、テーブルまたはセカンダリインデックスを論理的に複数のセグメントに分割し、複数のアプリケーションワーカーがセグメントを並行してスキャンします。各ワーカーは、スレッド (マルチスレッドをサポートするプログラミング言語) またはオペレーティングシステムプロセスにすることができます。並列スキャンを実行するには、各ワーカーが独自の Scan リクエストを以下のパラメータで送信します。

- Segment — 特定のワーカーによってスキャンされるセグメント。各ワーカーは、Segment に異なる値を使用する必要があります。
- TotalSegments — 並列スキャンのセグメントの合計数。この値は、アプリケーションが使用するワーカーの数と同じでなければなりません。

次の図表は、マルチスレッドアプリケーションが 3 度の並列処理で並列 Scan を実行する方法を示しています。



この図では、アプリケーションは 3 つのスレッドを生成し、各スレッドに番号を割り当てます。(セグメントはゼロベースであるため、最初の数値は常に 0 です。) 各スレッドは Scan リクエストを公開し、設定 Segment をその指定された番号に設定し、TotalSegments を 3 に設定します。各ス

レッドは、指定されたセグメントをスキャンし、一度に 1 MB のデータを取得し、アプリケーションのメインスレッドにデータを返します。

DynamoDB は、各項目のパーティションキーにハッシュ関数を適用して、項目をセグメントに割り当てます。指定された TotalSegments 値の場合、同じパーティションキーを持つすべての項目が常に同じ Segment に割り当てられます。つまり、項目 1、項目 2、項目 3 すべてが pk="account#123" を共有するテーブルでは (ソートキーは異なる)、ソートキーの値や項目コレクションのサイズに関係なく、これらの項目は同じワーカーによって処理されます。

セグメント割り当てはパーティションキーハッシュのみに基づいているため、セグメントは不均等に分散される可能性があります。一部のセグメントには項目が含まれていない場合がありますが、他のセグメントには大きな項目コレクションを持つパーティションキーが多く含まれている場合があります。その結果、セグメントの合計数を増やしても、特にパーティションキーがキースペース全体に均一に分散されていない場合、スキャンパフォーマンスが速くなることは保証されません。

Segment および TotalSegments の値は、個々の Scan リクエストに適用されるため、いつでも異なる値を使用できます。アプリケーションが最高のパフォーマンスを達成するまで、これらの値および使用するワーカーの数を試さなければならない場合があります。

> **Note**
>
> 多数のワーカーを使用した並列スキャンでは、スキャン対象のテーブルまたはインデックスに対してプロビジョンされたスループットをすべて簡単に使用できます。テーブルまたはインデックスが他のアプリケーションから大量の読み込みまたは書き込みアクティビティが発生している場合は、このようなスキャンを避けることをお勧めします。
> リクエストごとに返されるデータの量を制御するには、Limit パラメータを使用します。これにより、1 人のワーカーがプロビジョンされたスループットをすべて消費し、他のすべてのワーカーが犠牲になる状況を防ぐことができます。

PartiQL: Amazon DynamoDB 用の SQL 互換クエリ言語

Amazon DynamoDB は SQL 互換のクエリ言語である PartiQL をサポートしており、Amazon DynamoDB でデータの選択、挿入、更新、および削除を行うことができます。PartiQL を使用すると、AWS マネジメントコンソール、NoSQL Workbench、AWS Command Line Interface、および PartiQL 用の DynamoDB API を使用して、DynamoDB テーブルと簡単にやり取りを行い、アドホッククエリを実行できます。

PartiQL オペレーションは、他の DynamoDB データプレーンオペレーションと同様の可用性、レイテンシー、パフォーマンスを提供します。

次のセクションでは、PartiQL の DynamoDB 実装について説明します。

トピック

- PartiQL とは何ですか?
- Amazon DynamoDB での PartiQL
- DynamoDB 用の PartiQL の開始方法
- DynamoDB の PartiQL データ型
- DynamoDB 用の PartiQL ステートメント
- DynamoDB での PartiQL 関数の使用
- DynamoDB での PartiQL 算術演算子、比較演算子、論理演算子
- DynamoDB 用の PartiQL を使用してトランザクションを実行する
- DynamoDB 用の PartiQL を使用してバッチ操作を実行する
- DynamoDB 用 PartiQL における IAM セキュリティポリシー

PartiQL とは何ですか?

PartiQL は、構造化データ、半構造化データ、ネストされたデータを含む複数のデータストア間で、SQL 互換のクエリアクセスを提供します。PartiQL は、Amazon 内で広く使用されており、現在、DynamoDB を含む多くの AWS のサービスの一部として利用できます。

PartiQL の仕様とコアクエリ言語のチュートリアルについては、PartiQL ドキュメントを参照してください。

> **Note**
>
> - Amazon DynamoDB は、PartiQL クエリ言語のサブセットをサポートしています。
> - Amazon DynamoDB では、Amazon ion データ形式または Amazon Ion リテラルはサポートしていません。

Amazon DynamoDB での PartiQL

DynamoDB で PartiQL クエリを実行するには、次を使用します。

- DynamoDB コンソール
- NoSQL Workbench
- AWS Command Line Interface (AWS CLI)
- DynamoDB API

これらの方法を使用して DynamoDB にアクセスする方法については、「DynamoDB へのアクセス」を参照してください。

DynamoDB 用の PartiQL の開始方法

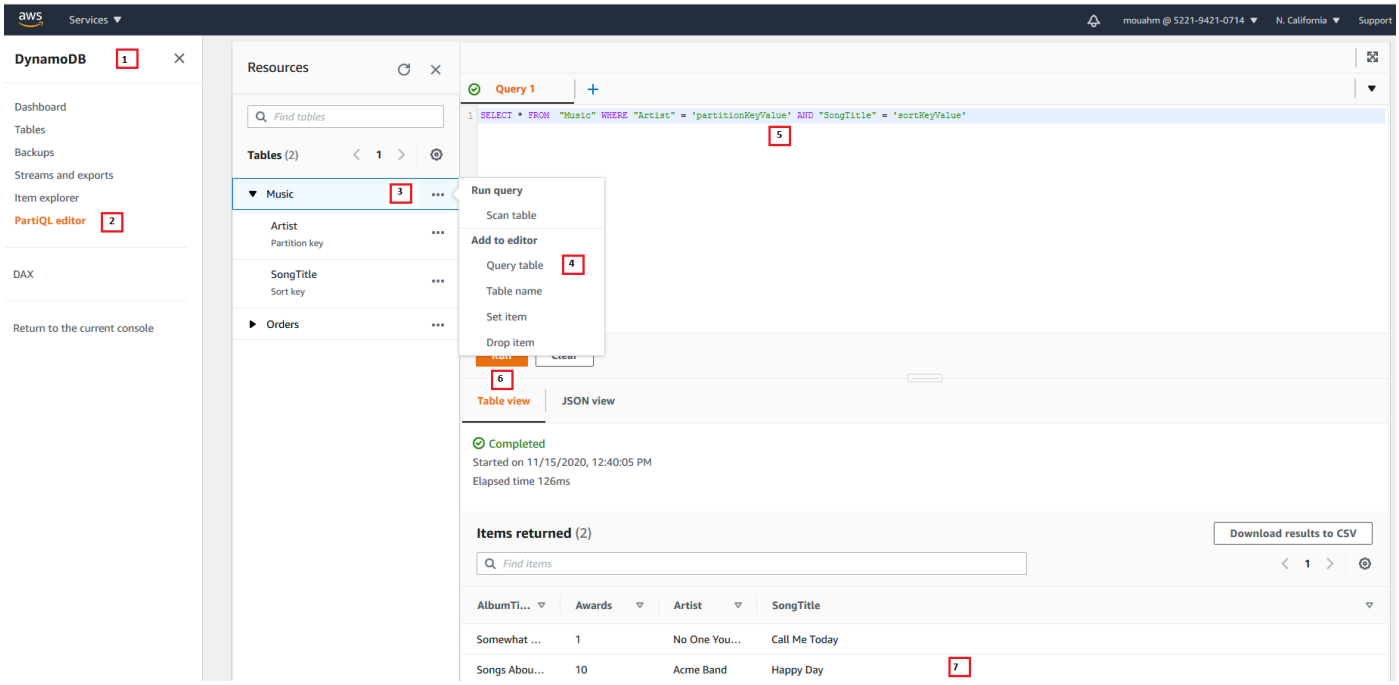
このセクションでは、Amazon DynamoDB コンソール、AWS Command Line Interface (AWS CLI)、および DynamoDB API から DynamoDB 用の PartiQL を使用する方法について説明します。

次の例では、「DynamoDB の開始方法」チュートリアルで、DynamoDB テーブルを定義していることが前提です。

DynamoDB コンソール、AWS Command Line Interface、または DynamoDB API を使用して DynamoDB にアクセスする方法については、「DynamoDB へのアクセス」を参照してください。

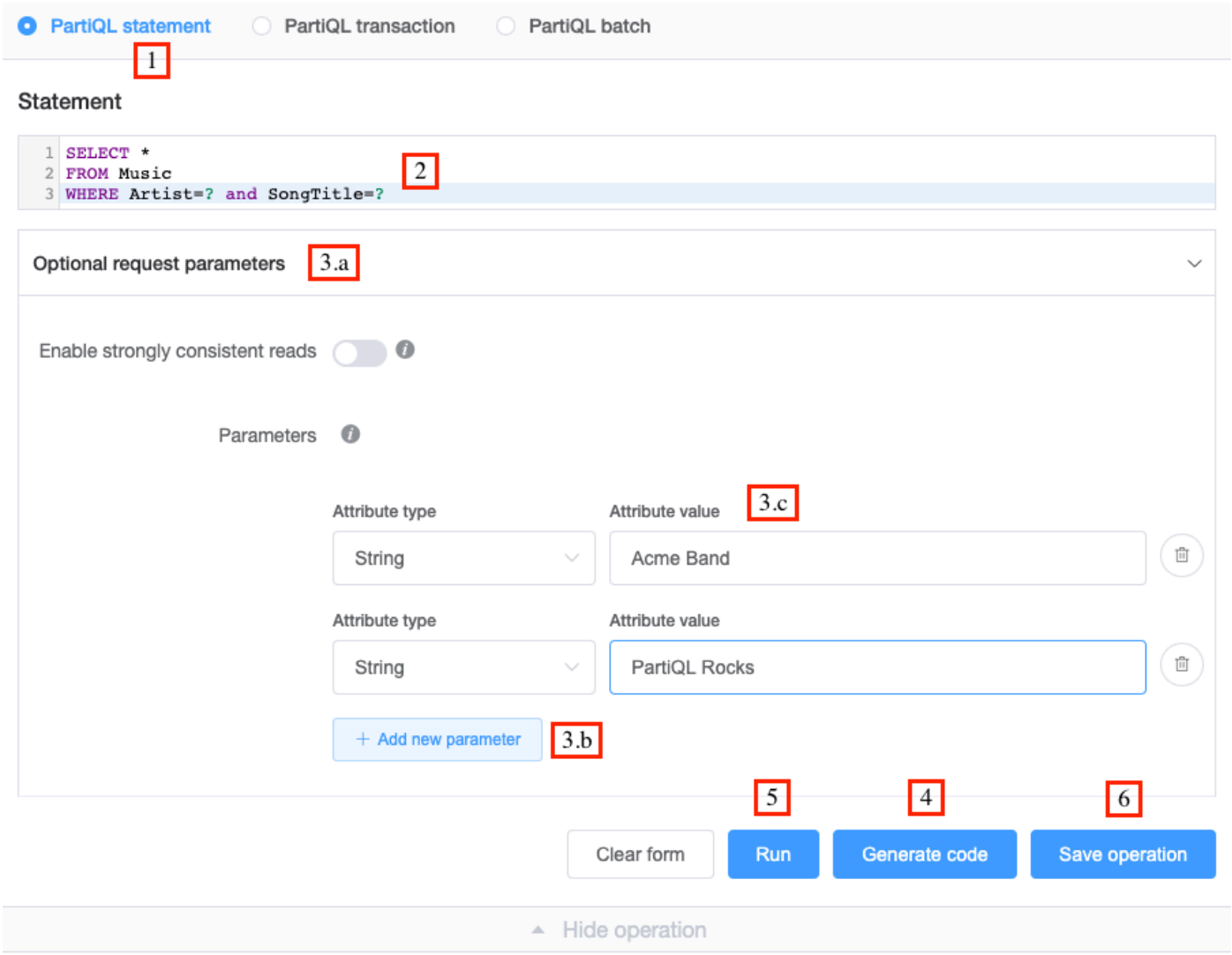
NoSQL Workbench をダウンロードして、DynamoDB 用の PartiQL ステートメントを構築するには、DynamoDB Operation Builder 用の NoSQL Workbench の右上にある [PartiQL operations] (PartiQL オペレーション) を選択します。

Console



1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソール左側のナビゲーションペインで、[PartiQL エディター] を選択します。
3. [Music] テーブルを選択します。
4. [Query table] (クエリテーブル) を選択します。このアクションで生成したクエリは、完全なテーブルスキャンを実行しません。
5. `partitionKeyValue` を文字列型の値である `Acme Band` に置換します。`sortKeyValue` を文字列型の値である `Happy Day` に置換します。
6. [実行] ボタンを選択します。
7. [Table view] (テーブルビュー) または [JSON view] (JSON ビュー) ボタンを選択すると、クエリの結果を確認できます。

NoSQL workbench



1. [PartiQL ステートメント] を選択します。
2. 次の PartiQL [SELECT statement] (SELECT ステートメント) を入力します。

```
SELECT *
FROM Music
WHERE Artist=? and SongTitle=?
```

3. `Artist` および `SongTitle` パラメータの値を指定するには

   a. [オプションのリクエストパラメータ] を選択します。

   b. [新しいパラメータの追加] を選択します。

   c. 属性タイプとして [string] (文字列型) を選択し、値に `Acme Band` を選択します。

d. ステップ b とステップ c を繰り返し、[string] (文字列型) のタイプと `PartiQL Rocks` の値を選択します。

4. コードを生成する場合は、[Generate code (コードの生成)] を選択します。

   表示されたタブから目的の言語を選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。

5. オペレーションをすぐに実行する場合は、[実行] をクリックします。

AWS CLI

1. INSERT PartiQL ステートメントを使用して、`Music` テーブルに項目を作成します。

```
aws dynamodb execute-statement --statement "INSERT INTO Music  \
         VALUE  \
         {'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}"
```

2. SELECT PartiQL ステートメントを使用して、Music テーブルから項目を取得します。

```
aws dynamodb execute-statement --statement "SELECT * FROM Music   \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

3. UPDATE PartiQL ステートメントを使用して、`Music` テーブルの項目を更新します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardsWon=1  \
                                            SET AwardDetail={'Grammys':[2020,
 2018]}  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

`Music` テーブルに、項目のリスト値を追加します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardDetail.Grammys
 =list_append(AwardDetail.Grammys,[2016])  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Music テーブルから項目のリスト値を削除します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            REMOVE AwardDetail.Grammys[2]  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Music テーブルに、項目の新しいマップメンバーを追加します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET AwardDetail.BillBoard=[2020]  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Music テーブルに、項目の新しい文字列セットの属性を追加します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET BandMembers =<<'member1',
 'member2'>>  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

Music テーブルで、項目の文字列セットの属性を更新します。

```
aws dynamodb execute-statement --statement "UPDATE Music  \
                                            SET BandMembers
 =set_add(BandMembers, <<'newmember'>>)  \
                                            WHERE Artist='Acme Band' AND
 SongTitle='PartiQL Rocks'"
```

4. DELETE PartiQL ステートメントを使用して、Music テーブルから項目を削除します。

```
aws dynamodb execute-statement --statement "DELETE  FROM Music  \
    WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'"
```

Java

```
import java.util.ArrayList;
import java.util.List;
```

```
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import software.amazon.dynamodb.AmazonDynamoDB;
import software.amazon.dynamodb.AmazonDynamoDBClientBuilder;
import software.amazon.dynamodb.model.AttributeValue;
import software.amazon.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.dynamodb.model.ExecuteStatementRequest;
import software.amazon.dynamodb.model.ExecuteStatementResult;
import software.amazon.dynamodb.model.InternalServerErrorException;
import software.amazon.dynamodb.model.ItemCollectionSizeLimitExceededException;
import software.amazon.dynamodb.model.ProvisionedThroughputExceededException;
import software.amazon.dynamodb.model.RequestLimitExceededException;
import software.amazon.dynamodb.model.ResourceNotFoundException;
import software.amazon.dynamodb.model.TransactionConflictException;

public class DynamoDBPartiQGettingStarted {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-1");

        try {
            // Create ExecuteStatementRequest
            ExecuteStatementRequest executeStatementRequest = new
 ExecuteStatementRequest();
            List<AttributeValue> parameters= getPartiQLParameters();

            //Create an item in the Music table using the INSERT PartiQL statement
            processResults(executeStatementRequest(dynamoDB, "INSERT INTO Music
 value {'Artist':?,'SongTitle':?}", parameters));

            //Retrieve an item from the Music table using the SELECT PartiQL
 statement.
            processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
 where Artist=? and SongTitle=?", parameters));

            //Update an item in the Music table using the UPDATE PartiQL statement.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music
 SET AwardsWon=1 SET AwardDetail={'Grammys':[2020, 2018]}  where Artist=? and
 SongTitle=?", parameters));

            //Add a list value for an item in the Music table.
```

```
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016])  where Artist=? and
SongTitle=?", parameters));

            //Remove a list value for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music REMOVE
AwardDetail.Grammys[2]   where Artist=? and SongTitle=?", parameters));

            //Add a new map member for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist=? and SongTitle=?", parameters));

            //Add a new string set attribute for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music
SET BandMembers =<<'member1', 'member2'>> where Artist=? and SongTitle=?",
parameters));

            //update a string set attribute for an item in the Music table.
            processResults(executeStatementRequest(dynamoDB, "UPDATE Music SET
BandMembers =set_add(BandMembers, <<'newmember'>>) where Artist=? and SongTitle=?",
parameters));

            //Retrieve an item from the Music table using the SELECT PartiQL
statement.
            processResults(executeStatementRequest(dynamoDB, "SELECT * FROM Music
where Artist=? and SongTitle=?", parameters));

            //delete an item from the Music Table
            processResults(executeStatementRequest(dynamoDB, "DELETE  FROM Music
where Artist=? and SongTitle=?", parameters));
        } catch (Exception e) {
            handleExecuteStatementErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
    }

    private static List<AttributeValue> getPartiQLParameters() {
        List<AttributeValue> parameters = new ArrayList<AttributeValue>();
        parameters.add(new AttributeValue("Acme Band"));
        parameters.add(new AttributeValue("PartiQL Rocks"));
        return parameters;
```

```
   }

   private static ExecuteStatementResult executeStatementRequest(AmazonDynamoDB
client, String statement, List<AttributeValue> parameters ) {
        ExecuteStatementRequest request = new ExecuteStatementRequest();
        request.setStatement(statement);
        request.setParameters(parameters);
        return client.executeStatement(request);
   }

   private static void processResults(ExecuteStatementResult
executeStatementResult) {
        System.out.println("ExecuteStatement successful: "+
executeStatementResult.toString());

   }

   // Handles errors during ExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
   // your application use-case.
   private static void handleExecuteStatementErrors(Exception exception) {
        try {
            throw exception;
        } catch (ConditionalCheckFailedException ccfe) {
            System.out.println("Condition check specified in the operation failed,
review and update the condition " +
                                       "check before retrying. Error: " +
ccfe.getErrorMessage());
        } catch (TransactionConflictException tce) {
            System.out.println("Operation was rejected because there is an ongoing
transaction for the item, generally " +
                                       "safe to retry with exponential back-off.
Error: " + tce.getErrorMessage());
        } catch (ItemCollectionSizeLimitExceededException icslee) {
            System.out.println("An item collection is too large, you\'re using Local
Secondary Index and exceeded " +
                                       "size limit of items per
partition key. Consider using Global Secondary Index instead. Error: " +
icslee.getErrorMessage());
        } catch (Exception e) {
            handleCommonErrors(e);
        }
   }
```

```
    private static void handleCommonErrors(Exception exception) {
        try {
            throw exception;
        } catch (InternalServerErrorException isee) {
            System.out.println("Internal Server Error, generally safe to retry with
 exponential back-off. Error: " + isee.getErrorMessage());
        } catch (RequestLimitExceededException rlee) {
            System.out.println("Throughput exceeds the current throughput limit for
 your account, increase account level throughput before " +
                                        "retrying. Error: " +
 rlee.getErrorMessage());
        } catch (ProvisionedThroughputExceededException ptee) {
            System.out.println("Request rate is too high. If you're using a custom
 retry strategy make sure to retry with exponential back-off. " +
                                        "Otherwise consider reducing frequency of
 requests or increasing provisioned capacity for your table or secondary index.
 Error: " +
                                        ptee.getErrorMessage());
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("One of the tables was not found, verify table exists
 before retrying. Error: " + rnfe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that
 the request was correctly transmitted to the DynamoDB " +
                                        "service, but for some reason, the service
 was not able to process it, and returned an error response instead. Investigate and
 " +
                                        "configure retry strategy. Error type: " +
 ase.getErrorType() + ". Error message: " + ase.getErrorMessage());
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that
 the client was unable to get a response from DynamoDB " +
                                        "service, or the client was unable to parse
 the response from the service. Investigate and configure retry strategy. "+
                                        "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure
 retry strategy. Error: " + e.getMessage());
        }
    }

}
```

パラメータ化されたステートメントの使用

PartiQL ステートメント文字列に値を直接埋め込む代わりに、疑問符 (?) プレースホルダーを使用して、Parameters フィールドに値を個別に指定できます。各 ? は、指定された順序で、対応するパラメータ値に置き換えられます。

パラメータ化されたステートメントを使用すると、ステートメント構造がデータ値から分離され、ステートメントの読み取りと再利用が容易になるため、ベストプラクティスです。また、ステートメント文字列の属性値を手動でフォーマットしたり工スケープしたりする必要もなくなります。

パラメータ化されたステートメントは ExecuteStatement、BatchExecuteStatement、および ExecuteTransaction オペレーションでサポートされています。

次の例では、パーティションキーとソートキーのパラメータ化された値を使用して、Music テーブルから項目を取得します。

AWS CLI parameterized

```
aws dynamodb execute-statement \
    --statement "SELECT * FROM \"Music\" WHERE Artist=? AND SongTitle=?" \
    --parameters '[{"S": "Acme Band"}, {"S": "PartiQL Rocks"}]'
```

Java parameterized

```
List<AttributeValue> parameters = new ArrayList<>();
parameters.add(new AttributeValue("Acme Band"));
parameters.add(new AttributeValue("PartiQL Rocks"));

ExecuteStatementRequest request = new ExecuteStatementRequest()
    .withStatement("SELECT * FROM Music WHERE Artist=? AND SongTitle=?")
    .withParameters(parameters);

ExecuteStatementResult result = dynamoDB.executeStatement(request);
```

Python parameterized

```
response = dynamodb_client.execute_statement(
    Statement="SELECT * FROM Music WHERE Artist=? AND SongTitle=?",
    Parameters=[
        {'S': 'Acme Band'},
        {'S': 'PartiQL Rocks'}
    ]
```

```
)
```

> **Note**
>
> 前の開始方法セクションの Java の例では、全体にわたってパラメータ化されたステートメントを使用しています。`getPartiQLParameters()` メソッドはパラメータリストを構築し、各ステートメントはインライン値の代わりに ? プレースホルダーを使用します。

DynamoDB の PartiQL データ型

次の表に、DynamoDB 用の PartiQL で使用できるデータ型を、一覧で表示します。

DynamoDB データ型	PartiQL 表現	コメント
`Boolean`	TRUE、FALSE	大文字と小文字は区別されません。
`Binary`	該当なし	コードでのみサポートされています。
`List`	[value1, value2,...]	List 型に保存できるデータ型に制限はなく、リスト内の要素が同じ型である必要はありません。
`Map`	{ 'name' : value }	Map 型に保存できるデータ型に制限はなく、マップ内の要素が同じ型である必要はありません。
`Null`	NULL	大文字と小文字は区別されません。
`Number`	1, 1.0, 1e0	数値は、正、負、または 0 とすることができます。数値は最大 38 桁の精度を持つことができます。

DynamoDB データ型	PartiQL 表現	コメント
`Number Set`	<<number1, number2>>	数値セット内の要素は、Number 型でなければなりません。
`String Set`	<<'string1', 'string2'>>	文字列セット内の要素は、String 型でなければなりません。
`String`	'string value'	String 型 の値を指定するには、一重引用符を使用する必要があります。

> **Note**
>
> 文字列値にリテラル一重引用符 (') を含めるには、2 つの連続した一重引用符 ('') を使用してエスケープします。例えば、文字列 `it's here` は `'it''s here'` として記述されます。バックスラッシュ (\) は、DynamoDB の PartiQL で一重引用符をエスケープしません。一重引用符を使用すると、検証エラーが発生します。

例

次のデータ型を挿入する方法を、次のステートメントで示します。`String`、`Number`、`Map`、`List`、`Number Set` および `String Set`。

```
INSERT INTO TypesTable value {'primarykey':'1',
'NumberType':1,
'MapType' : {'entryname1': 'value', 'entryname2': 4},
'ListType': [1,'stringval'],
'NumberSetType':<<1,34,32,4.5>>,
'StringSetType':<<'stringval','stringval2'>>
}
```

次のステートメントでは、`Map`、`List`、`Number Set` および `String Set` の型に、新しい要素を挿入する方法を示しています。また、`Number` 型の値を変更する方法も示しています。

```
UPDATE TypesTable
SET NumberType=NumberType + 100
SET MapType.NewMapEntry=[2020, 'stringvalue', 2.4]
SET ListType = LIST_APPEND(ListType, [4, <<'string1', 'string2'>>])
SET NumberSetType= SET_ADD(NumberSetType, <<345, 48.4>>)
SET StringSetType = SET_ADD(StringSetType, <<'stringsetvalue1', 'stringsetvalue2'>>)
WHERE primarykey='1'
```

次のステートメントでは、Map、List、Number Set および String Set の型から要素を削除する方法を示しています。また、Number 型の値を変更する方法も示しています。

```
UPDATE TypesTable
SET NumberType=NumberType - 1
REMOVE ListType[1]
REMOVE MapType.NewMapEntry
SET NumberSetType = SET_DELETE( NumberSetType, <<345>>)
SET StringSetType = SET_DELETE( StringSetType, <<'stringsetvalue1'>>)
WHERE primarykey='1'
```

詳細については、「DynamoDB のデータ型」を参照してください。

DynamoDB 用の PartiQL ステートメント

Amazon DynamoDB は、次の PartiQL ステートメントをサポートしています。

> **Note**
>
> DynamoDB はすべての PartiQL ステートメントをサポートしているわけではありません。このリファレンスでは、AWS CLI シェルまたは API で手動で実行する PartiQL ステートメントについて、ベーシックな構文と使用例を示します。

データ操作言語 (DML) は、DynamoDB テーブル内のデータを管理するために使用する、PartiQL ステートメントのセットです。DML ステートメントを使用して、テーブル内のデータを追加、変更、または削除します。

次の DML ステートメントとクエリ言語ステートメントがサポートされています。

- DynamoDB 用の PartiQL select ステートメント
- DynamoDB 用の PartiQL 更新ステートメント

- DynamoDB 用の PartiQL 挿入ステートメント
- DynamoDB 用の PartiQL 削除ステートメント

DynamoDB 用の PartiQL を使用してトランザクションを実行する および DynamoDB 用の PartiQL を使用してバッチ操作を実行する は、DynamoDB 用の PartiQL でもサポートされています。

DynamoDB 用の PartiQL select ステートメント

Amazon DynamoDB では、SELECT ステートメントを使用して、テーブルからデータを取得します。

SELECT ステートメントを使用すると、パーティションキーを持つ等価条件または IN 条件が WHERE 句で指定されていない場合、完全なテーブルスキャンになることがあります。スキャンオペレーションは、すべての項目でリクエストされた値を調べるので、大きなテーブルまたはインデックスに対してプロビジョニングされたスループットを 1 回のオペレーションで使い果たすことがあります。

PartiQL で完全なテーブルスキャンを避けたい場合は、次のようにします。

- 完全なテーブルスキャンが行われないように、WHERE 句の条件を適切に設定していることを確かめながら、SELECT ステートメントを作成します。
- DynamoDB デベロッパーガイドの 例: DynamoDB 用の PartiQL で ステートメントの選択を許可し、テーブル全体のスキャンを行うステートメントを拒否する に記載されているように、IAM ポリシーを使用して、完全なテーブルスキャンを無効にします。

詳細については、「DynamoDB デベロッパーガイド」の「データのクエリとスキャンのベストプラクティス」を参照してください。

トピック

- 構文
- パラメータ
- 例

構文

```
SELECT expression  [, ...]
FROM table[.index]
```

```
[ WHERE condition ] [ [ORDER BY key [DESC|ASC] , ...]
```

パラメータ

#

(必須) * ワイルドカードからの射影、または結果セットからの 1 つ以上の属性名かドキュメントパスの射影リスト。式には、DynamoDB での PartiQL 関数の使用 への呼び出し、または DynamoDB での PartiQL 算術演算子、比較演算子、論理演算子 によって変更されたフィールドで構成できます。

####*

(必須) クエリするテーブル名。

######

(オプション) クエリを実行するインデックスの名前です。

> **Note**
>
> インデックスにクエリを実行するときは、テーブル名とインデックス名に二重引用符を追加する必要があります。
>
> ```
> SELECT *
> FROM "TableName"."IndexName"
> ```

condition

(オプション) クエリの選択条件。

> **Important**
>
> SELECT ステートメントによって、完全なテーブルスキャンが行われないようにするには、WHERE 句の条件がパーティションキーを指定する必要があります。等価演算子または IN 演算子を使用します。
> 例えば、OrderID パーティションキーと、Address を含むその他の非キー属性がある Orders テーブルがある場合、次のステートメントは完全なテーブルスキャンを実行しません。
>
> ```
> SELECT *
> ```

```
FROM "Orders"
WHERE OrderID = 100

SELECT *
FROM "Orders"
WHERE OrderID = 100 and Address='some address'

SELECT *
FROM "Orders"
WHERE OrderID = 100 or OrderID = 200

SELECT *
FROM "Orders"
WHERE OrderID IN [100, 300, 234]
```

ただし、次の SELECT ステートメントを実行すると、完全なテーブルスキャンが行われます。

```
SELECT *
FROM "Orders"
WHERE OrderID > 1

SELECT *
FROM "Orders"
WHERE Address='some address'

SELECT *
FROM "Orders"
WHERE OrderID = 100 OR Address='some address'
```

key

(任意) 返ってきた結果の並び替えに使用するハッシュキーまたはソートキー。デフォルトの順序は昇順 (ASC) です。返ってきた結果を降順に並べる場合は、DESC を指定します。

> **Note**
>
> WHERE 句を省略すると、テーブル内のすべての項目が取得されます。

例

次のクエリは、Orders テーブルでパーティションキーと OrderID を指定し、等価演算子を使用することで、存在する場合に項目を 1 つ返します。

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID = 1
```

次のクエリは、OR 演算子を使用することで、特定のパーティションキーと OrderID、値を持つ Orders テーブルから、すべての項目を返します。

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID = 1 OR OrderID = 2
```

次のクエリは、IN 演算子を使用することで、特定のパーティションキーと OrderID、値を持つ Orders テーブルから、すべての項目を返します。返ってきた結果は、OrderID キー属性の値によって、降順に並べられます。

```
SELECT OrderID, Total
FROM "Orders"
WHERE OrderID IN [1, 2, 3] ORDER BY OrderID DESC
```

次のクエリは、Total が非キー属性である場合に、Total が 500 以上である Orders テーブルからすべての項目を返す、完全なテーブルスキャンを示します。

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total > 500
```

次のクエリは、IN 演算子と非キー属性である Total を使用して、特定の Total 範囲の順序で、Orders テーブルからすべての項目を返す、完全なテーブルスキャンを示します。

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total IN [500, 600]
```

次のクエリは、BETWEEN 演算子と非キー属性である Total を使用して、特定の Total 範囲の順序で、Orders テーブルからすべての項目を返す、完全なテーブルスキャンを示します。

```
SELECT OrderID, Total
FROM "Orders"
WHERE Total BETWEEN 500 AND 600
```

次のクエリは、WHERE 句の条件で CustomerID パーティションキーと MovieID ソートキーを指定し、SELECT 句でドキュメントのパスを使用することで、firestick デバイスで視聴を開始した最初の日付を返します。

```
SELECT Devices.FireStick.DateWatched[0]
FROM WatchList
WHERE CustomerID= 'C1' AND MovieID= 'M1'
```

次のクエリは、テーブル全体のスキャンを示しています。このスキャンでは、WHERE 句の条件でドキュメントのパスを使用し、2019 年 12 月 24 日以降に初めて firestick デバイスを使用した項目のリストを返します。

```
SELECT Devices
FROM WatchList
WHERE Devices.FireStick.DateWatched[0] >= '12/24/19'
```

DynamoDB 用の PartiQL 更新ステートメント

UPDATE ステートメントを使用して、Amazon DynamoDB テーブルの項目内にある、1 つ以上の属性の値を変更します。

> **Note**
>
> 一度に更新できる項目は 1 つだけです。1 つの DynamoDB PartiQL ステートメントを発行して、複数の項目を更新することはできません。複数項目の更新については、「DynamoDB 用の PartiQL を使用してトランザクションを実行する」または「DynamoDB 用の PartiQL を使用してバッチ操作を実行する」を参照してください。

トピック

- 構文
- パラメータ
- 戻り値

- 例

構文

```
UPDATE  table
[SET | REMOVE]  path  [=  data] […]
WHERE condition [RETURNING returnvalues]
<returnvalues>  ::= [ALL OLD | MODIFIED OLD | ALL NEW | MODIFIED NEW] *
```

パラメータ

####*

(必須) 修正されるデータを含んでいるテーブル。

##

(必須) 作成または変更される属性名、またはドキュメントパス。

data

(必須) 属性値またはオペレーションの結果。

SET で使用するためにサポートされている操作は、次の通りです。

- LIST_APPEND: List 型に値を追加します。
- SET_ADD: 数値または文字列セットに値を追加します。
- SET_DELETE: 数値または文字列セットから値を削除します。

condition

(必須) 修正される項目の選択条件。この条件は、単一のプライマリキー値を解決する必要があります。

returnvalues

(オプション) 属性が更新される前か、更新された後に、表示された項目の属性を取得したい場合に、`returnvalues` を使用します。有効な値は以下のとおりです。

- `ALL OLD *`: 更新操作の前に表示されていた項目について、すべての属性を返します。
- `MODIFIED OLD *`: 更新操作の前に表示されていた属性について、更新された属性だけを返します。

- ALL NEW *: 更新操作の後に表示される項目について、すべての属性を返します。
- MODIFIED NEW *: UpdateItem 操作の後に表示される属性について、更新された属性だけを返します。

戻り値

returnvalues パラメータが指定されない限り、このステートメントは値を返しません。

> **Note**
>
> UPDATE ステートメントの WHERE 句が、DynamoDB テーブルのどの項目も true と評価しない場合、ConditionalCheckFailedException が返ります。

例

既存の項目の属性値を更新します。属性が存在しない場合は、作成されます。

次のクエリは、数値型の属性 (AwardsWon) とマップ型の属性 (AwardDetail) を追加して、"Music" テーブルの項目を更新します。

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

RETURNING ALL OLD * を追加すると、Update 操作の前に表示されていた属性を返すことができます。

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL OLD *
```

以下が返されます。

```
{
```

```
    "Items": [
        {
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

RETURNING ALL NEW * を追加すると、Update 操作の後に表示されていた属性を返すことができます。

```
UPDATE "Music"
SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
RETURNING ALL NEW *
```

以下が返されます。

```
{
    "Items": [
        {
            "AwardDetail": {
                "M": {
                    "Grammys": {
                        "L": [
                            {
                                "N": "2020"
                            },
                            {
                                "N": "2018"
                            }
                        ]
                    }
                }
            },
            "AwardsWon": {
                "N": "1"
            }
```

```
        }
    ]
}
```

次のクエリは、"Music" テーブルの項目を、AwardDetail.Grammys リストに追加して更新します。

```
UPDATE "Music"
SET AwardDetail.Grammys =list_append(AwardDetail.Grammys,[2016])
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

次のクエリは、"Music" テーブルの項目を、AwardDetail.Grammys リストから削除して更新します。

```
UPDATE "Music"
REMOVE AwardDetail.Grammys[2]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

次のクエリは、"Music" テーブルで AwardDetail マップに BillBoard を追加して、項目を更新します。

```
UPDATE "Music"
SET AwardDetail.BillBoard=[2020]
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

次のクエリは、"Music" テーブルで、文字列セットの属性 BandMembers を追加して、項目を更新します。

```
UPDATE "Music"
SET BandMembers =<<'member1', 'member2'>>
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

次のクエリは、"Music" テーブルで文字列セットの属性 BandMembers に newbandmember を追加して、項目を更新します。

```
UPDATE "Music"
SET BandMembers =set_add(BandMembers, <<'newbandmember'>>)
WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'
```

DynamoDB 用の PartiQL 削除ステートメント

DELETE ステートメントを使用して、Amazon DynamoDB テーブルにある項目を削除します。

> **Note**
>
> 一度に削除できる項目は 1 つだけです。1 つの DynamoDB PartiQL ステートメントを発行して、複数の項目を削除することはできません。複数のアイテムの削除については、「DynamoDB 用の PartiQL を使用してトランザクションを実行する」または「DynamoDB 用の PartiQL を使用してバッチ操作を実行する」を参照してください。

トピック

- 構文
- パラメータ
- 戻り値
- 例

構文

```
DELETE FROM table
 WHERE condition [RETURNING returnvalues]
 <returnvalues>  ::= ALL OLD *
```

パラメータ

####*

(必須) 削除する項目を含む DynamoDB テーブル。

condition

(必須) 削除する項目の選択基準。この条件は、単一のプライマリキー値を解決する必要があります。

returnvalues

(オプション) 削除される前に表示された、項目の属性を取得したい場合には、returnvalues を使用します。有効な値は以下のとおりです。

- ALL OLD *: 古い項目の内容が返されます。

戻り値

`returnvalues` パラメータが指定されない限り、このステートメントは値を返しません。

> **Note**
>
> DynamoDB テーブルに、DELETE が発行された項目と同じプライマリキーを持つ項目が無い場合、削除した項目を 0 としたうえで、SUCCESS を返します。テーブルに同じプライマリキーを持つ項目があるが、DELETE ステートメントの WHERE 句の条件で false と評価された場合、`ConditionalCheckFailedException` が返されます。

例

次のクエリは、`"Music"` テーブルの項目を削除します。

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'
```

`RETURNING ALL OLD *` を追加すると、削除されたデータを返すことができます。

```
DELETE FROM "Music" WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks'
 RETURNING ALL OLD *
```

`Delete` ステートメントは、以下を返すようになりました。

```
{
    "Items": [
        {
            "Artist": {
                "S": "Acme Band"
            },
            "SongTitle": {
                "S": "PartiQL Rocks"
            }
        }
    ]
}
```

DynamoDB 用の PartiQL 挿入ステートメント

`INSERT` ステートメントを使用して、Amazon DynamoDB のテーブルに項目を追加します。

> **Note**
>
> 一度に挿入できる項目は 1 つだけです。1 つの DynamoDB PartiQL ステートメントを発行して、複数の項目を挿入することはできません。複数の項目の挿入については、「DynamoDB 用の PartiQL を使用してトランザクションを実行する」または「DynamoDB 用の PartiQL を使用してバッチ操作を実行する」を参照してください。

トピック

- 構文
- パラメータ
- 戻り値
- 例

構文

項目を 1 つ挿入します。

```
INSERT INTO table VALUE item
```

パラメータ

####*

(必須) データを挿入するテーブル。このテーブルは既存であることが必要です。

item

(必須) 有効な DynamoDB 項目は PartiQL タプル。1 つの項目のみ指定する必要があります。また、項目の各属性名は大文字と小文字が区別され、PartiQL では一重引用符 (`'...'`) で示されます。

文字列値は、PartiQL では一重引用符 (`'...'`) で示されます。

戻り値

このステートメントは値を返しません。

> **Note**
>
> DynamoDB テーブルに、挿入される項目と同じプライマリキーを持つ項目が既にある場合、`DuplicateItemException` が返されます。

例

```
INSERT INTO "Music" value {'Artist' : 'Acme Band','SongTitle' : 'PartiQL Rocks'}
```

DynamoDB での PartiQL 関数の使用

Amazon DynamoDB の PartiQL は、次の SQL 標準関数の組み込みバージョンをサポートしています。

> **Note**
>
> このリストに含まれていない SQL 関数は、DynamoDB で現在サポートされていません。

集計関数

- Amazon DynamoDB 用の PartiQL で SIZE 関数を使用する

条件関数

- DynamoDB 用の PartiQL での EXISTS 関数の使用
- DynamoDB 用の PartiQL で ATTRIBUTE_TYPE 関数を使用する
- DynamoDB 用の PartiQL で BEGINS_WITH 関数を使用する
- DynamoDB 用の PartiQL で CONTAINS 関数を使用する
- DynamoDB 用の PartiQL で MISSING 関数を使用する

DynamoDB 用の PartiQL での EXISTS 関数の使用

EXISTS を使用して、`ConditionCheck` が TransactWriteItems API で実行するのと同じ機能を実行できます。EXISTS 関数は、トランザクションでのみ使用できます。

値が指定されると、値が空でないコレクションである場合に TRUE を返します。それ以外の場合は FALSE を返します。

> **Note**
>
> この関数は、トランザクション操作でのみ使用できます。

構文

```
EXISTS ( statement )
```

引数

statement

(必須) 関数が評価する SELECT ステートメント。

> **Note**
>
> SELECT 文では、完全なプライマリキーと他の条件を、1 つ指定する必要があります。

戻り型

bool

例

```
EXISTS(
    SELECT * FROM "Music"
    WHERE "Artist" = 'Acme Band' AND "SongTitle" = 'PartiQL Rocks')
```

DynamoDB 用の PartiQL で BEGINS_WITH 関数を使用する

指定した属性が特定の部分文字列から始まる場合、TRUE を返します。

構文

```
begins_with(path, value )
```

引数

##

(必須) 使用する属性名またはドキュメントのパス。

value

(必須) 検索対象の文字列。

戻り型

bool

例

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND begins_with("Address", '7834 24th')
```

DynamoDB 用の PartiQL で MISSING 関数を使用する

指定した属性が項目に含まれていない場合、TRUE を返します。この関数では、等価演算子および不等価演算子のみを使用できます。

構文

```
 attributename IS | IS NOT  MISSING
```

引数

attributename

(必須) 検索対象の属性名。

戻り型

bool

例

```
SELECT * FROM Music WHERE "Awards" is MISSING
```

DynamoDB 用の PartiQL で ATTRIBUTE_TYPE 関数を使用する

指定したパスの属性が特定のデータ型のものである場合、TRUE を返します。

構文

```
attribute_type( attributename, type )
```

引数

attributename

(必須) 使用する属性名。

type

(必須) チェックする属性タイプ。有効な値のリストについては、DynamoDB の「attribute_type」を参照してください。

戻り型

bool

例

```
SELECT * FROM "Music" WHERE attribute_type("Artist", 'S')
```

DynamoDB 用の PartiQL で CONTAINS 関数を使用する

パスで指定した属性が次のいずれかである場合、TRUE を返します。

- 特定の部分文字列を含む文字列型。
- 特定の要素を含むセット型。

詳細については、DynamoDB の「contains 関数」を参照してください。

構文

```
contains( path, substring )
```

引数

##

(必須) 使用する属性名またはドキュメントのパス。

substring

(必須) チェックする対象の属性の部分文字列またはセットのメンバー。詳細については、DynamoDB の「contains 関数」を参照してください。

戻り型

bool

例

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND contains("Address", 'Kirkland')
```

Amazon DynamoDB 用の PartiQL で SIZE 関数を使用する

属性のサイズを表す数値をバイト単位で返します。Size で使用できる有効なデータ型は、次の通りです。詳細については、DynamoDB の「size 関数」を参照してください。

構文

```
size( path)
```

引数

##

(必須) 属性名またはドキュメントのパス。

サポートされているタイプについては、DynamoDB の「size 関数」を参照してください。

戻り型

int

例

```
SELECT * FROM "Orders" WHERE "OrderID"=1 AND size("Image") >300
```

DynamoDB での PartiQL 算術演算子、比較演算子、論理演算子

Amazon DynamoDB の PartiQL は、次の SQL 標準演算子をサポートしています。

> **Note**
> このリストに含まれていない SQL 演算子は、DynamoDB で現在サポートされていません。

算術演算子

演算子	説明
+	Add
-	- (減算)

比較演算子

演算子	説明
=	Equal to
<>	等しくない
!=	等しくない
>	超
<	未満
>=	以上
<=	以下

論理演算子

演算子	説明
AND	TRUE AND で区切られたすべての条件が TRUE の場合に
BETWEEN	TRUE オペランドが比較の範囲内にある場合に この演算子には、適用するオペランドの下限と上限が含まれます。
IN	TRUE オペランドが (最大 50 ハッシュ属性値、または 100 個の非キー属性値において) 式のリストの 1 つに等しい場合。 結果は最大 10 項目までのページで返されます。IN リストにより多くの値が含まれている場合は、レスポンスで返された NextToken を使用して後続のページを取得する必要があります。
IS	TRUE オペランドが指定されていて、PartiQL のデータ型であり、NULL または MISSING を含む場合に
NOT	指定されたブール式の値を反転する
OR	TRUE OR で区切られた条件のいずれかが TRUEの場合に

論理演算子の使用の詳細については、「比較の実行」および「論理評価」を参照してください。

DynamoDB 用の PartiQL を使用してトランザクションを実行する

このセクションでは、DynamoDB 用の PartiQL でトランザクションを使用する方法について説明します。PartiQL トランザクションは、合計 100 件のステートメント (アクション) に制限されています。

DynamoDB トランザクションの詳細については、「DynamoDB トランザクションで複雑なワークフローを管理する」を参照してください。

> **Note**
>
> トランザクション全体は、読み込みステートメントまたは 書き込みステートメントのいずれかで構成されている必要があります。1 つのトランザクションで両方を混在させることはできません。この EXISTS 関数は例外です。TransactWriteItems API オペレーションの ConditionCheck と同様に、項目の特定の属性の状態を確認するために使用できます。

トピック

- 構文
- パラメータ
- 戻り値
- 例

構文

```
[
   {
      "Statement":" statement ",
      "Parameters":[
         {
            " parametertype " : " parametervalue "
         }, ...]
   } , ...
]
```

パラメータ

#######

(必須) DynamoDB 用の PartiQL がサポートするステートメントです。

> **Note**
>
> トランザクション全体は、読み込みステートメントまたは 書き込みステートメントのいずれかで構成されている必要があります。1 つのトランザクションで両方を混在させることはできません。

parametertype

(オプション) PartiQL ステートメントを指定するときにパラメータが使用された場合の DynamoDB タイプ。

parametervalue

(オプション) PartiQL ステートメントを指定するときにパラメータが使用された場合のパラメータの値。

戻り値

このステートメントは、書き込み操作 (INSERT、UPDATE、DELETE) の値を返しません。ただし、WHERE 句で指定された条件に基づいて、読み取り操作 (SELECT) に対して異なる値を返します。

> **Note**
>
> シングルトンの INSERT、UPDATE、または DELETE 操作のいずれかでエラーが返された場合、`TransactionCanceledException` 例外によりトランザクションがキャンセルされます。キャンセル理由コードには、個々のシングルトン操作からのエラーが含まれます。

例

以下の例では、トランザクションとして複数のステートメントを実行します。

AWS CLI

1. 次の JSON コードを partiql.json というファイルに保存します

```
[
    {
```

```
        "Statement": "EXISTS(SELECT * FROM \"Music\" where Artist='No One You
 Know' and SongTitle='Call Me Today' and Awards is  MISSING)"
    },
    {
        "Statement": "INSERT INTO Music value {'Artist':?,'SongTitle':'?'}",
        "Parameters": [{\"S\": \"Acme Band\"}, {\"S\": \"Best Song\"}]
    },
    {
        "Statement": "UPDATE \"Music\" SET AwardsWon=1 SET
 AwardDetail={'Grammys':[2020, 2018]}  where Artist='Acme Band' and
 SongTitle='PartiQL Rocks'"
    }
]
```

2. コマンドプロンプトで、次のコマンドを実行します。

```
aws dynamodb execute-transaction --transact-statements  file://partiql.json
```

Java

```
public class DynamoDBPartiqlTransaction {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");

        try {
            // Create ExecuteTransactionRequest
            ExecuteTransactionRequest executeTransactionRequest =
 createExecuteTransactionRequest();
            ExecuteTransactionResult executeTransactionResult =
 dynamoDB.executeTransaction(executeTransactionRequest);
            System.out.println("ExecuteTransaction successful.");
            // Handle executeTransactionResult

        } catch (Exception e) {
            handleExecuteTransactionErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {
        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
```

```
    }

    private static ExecuteTransactionRequest createExecuteTransactionRequest() {
        ExecuteTransactionRequest request = new ExecuteTransactionRequest();

        // Create statements
        List<ParameterizedStatement> statements = getPartiQLTransactionStatements();

        request.setTransactStatements(statements);
        return request;
    }

    private static List<ParameterizedStatement> getPartiQLTransactionStatements() {
        List<ParameterizedStatement> statements = new
ArrayList<ParameterizedStatement>();

        statements.add(new ParameterizedStatement()
                                .withStatement("EXISTS(SELECT * FROM "Music" where
Artist='No One You Know' and SongTitle='Call Me Today' and Awards is  MISSING)"));

        statements.add(new ParameterizedStatement()
                                .withStatement("INSERT INTO "Music" value
{'Artist':'?','SongTitle':'?'}")
                                .withParameters(new AttributeValue("Acme Band"),new
AttributeValue("Best Song")));

        statements.add(new ParameterizedStatement()
                                .withStatement("UPDATE "Music" SET AwardsWon=1
SET AwardDetail={'Grammys':[2020, 2018]}  where Artist='Acme Band' and
SongTitle='PartiQL Rocks'"));

        return statements;
    }

    // Handles errors during ExecuteTransaction execution. Use recommendations in
error messages below to add error handling specific to
    // your application use-case.
    private static void handleExecuteTransactionErrors(Exception exception) {
        try {
            throw exception;
        } catch (TransactionCanceledException tce) {
            System.out.println("Transaction Cancelled, implies a client issue, fix
before retrying. Error: " + tce.getErrorMessage());
        } catch (TransactionInProgressException tipe) {
```

```
            System.out.println("The transaction with the given request token is
already in progress, consider changing " +
                "retry strategy for this type of error. Error: " +
tipe.getErrorMessage());
        } catch (IdempotentParameterMismatchException ipme) {
            System.out.println("Request rejected because it was retried with a
different payload but with a request token that was already used, " +
                "change request token for this payload to be accepted. Error: " +
ipme.getErrorMessage());
        } catch (Exception e) {
            handleCommonErrors(e);
        }
    }

    private static void handleCommonErrors(Exception exception) {
        try {
            throw exception;
        } catch (InternalServerErrorException isee) {
            System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
        } catch (RequestLimitExceededException rlee) {
            System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
                "retrying. Error: " + rlee.getErrorMessage());
        } catch (ProvisionedThroughputExceededException ptee) {
            System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
                "Otherwise consider reducing frequency of requests or increasing
provisioned capacity for your table or secondary index. Error: " +
                ptee.getErrorMessage());
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("One of the tables was not found, verify table exists
before retrying. Error: " + rnfe.getErrorMessage());
        } catch (AmazonServiceException ase) {
            System.out.println("An AmazonServiceException occurred, indicates that
the request was correctly transmitted to the DynamoDB " +
                "service, but for some reason, the service was not able to process
it, and returned an error response instead. Investigate and " +
                "configure retry strategy. Error type: " + ase.getErrorType() + ".
Error message: " + ase.getErrorMessage());
        } catch (AmazonClientException ace) {
            System.out.println("An AmazonClientException occurred, indicates that
the client was unable to get a response from DynamoDB " +
```

```
                "service, or the client was unable to parse the response from the
 service. Investigate and configure retry strategy. "+
                "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure
 retry strategy. Error: " + e.getMessage());
        }
    }

}
```

次の例は、DynamoDB が WHERE 句で指定された異なる条件を持つ項目を読み取るときのさまざまな戻り値を示しています。

AWS CLI

1. 次の JSON コードを partiql.json というファイルに保存します

```
[
    // Item exists and projected attribute exists
    {
        "Statement": "SELECT * FROM "Music" WHERE Artist='No One You Know' and
 SongTitle='Call Me Today'"
    },
    // Item exists but projected attributes do not exist
    {
        "Statement": "SELECT non_existent_projected_attribute FROM "Music" WHERE
 Artist='No One You Know' and SongTitle='Call Me Today'"
    },
    // Item does not exist
    {
        "Statement": "SELECT * FROM "Music" WHERE Artist='No One I Know' and
 SongTitle='Call You Today'"
    }
]
```

2. コマンドプロンプトで、次のコマンドを実行します。

```
aws dynamodb execute-transaction --transact-statements  file://partiql.json
```

3. 以下のレスポンスが返されます。

```
{
    "Responses": [
        // Item exists and projected attribute exists
        {
            "Item": {
                "Artist":{
                    "S": "No One You Know"
                },
                "SongTitle":{
                    "S": "Call Me Today"
                }
            }
        },
        // Item exists but projected attributes do not exist
        {
            "Item": {}
        },
        // Item does not exist
        {}
    ]
}
```

DynamoDB 用の PartiQL を使用してバッチ操作を実行する

このセクションでは、DynamoDB 用の PartiQL でバッチステートメントを使用する方法について説明します。

> **Note**
>
> - バッチ全体は、読み取りステートメントまたは書き込みステートメントのいずれかで構成する必要があります。1 つのバッチに両方を混在させることはできません。
> - `BatchExecuteStatement` と `BatchWriteItem` が実行できるステートメントの数は、バッチあたり 25 が上限です。
> - `BatchExecuteStatement` は、`BatchGetItem` を使用して、個別のステートメントでプライマリキーのリストを取得します。

トピック

- 構文
- パラメータ
- 例

構文

```
[
  {
    "Statement": "SELECT pk FROM ProblemSet WHERE pk = 'p#9StkWHYTxm7x2AqSXcrfu7' AND
 sk = 'info'"
  },
  {
    "Statement": "SELECT pk FROM ProblemSet WHERE pk = 'p#isC2ChceGbxHgESc4szoTE' AND
 sk = 'info'"
  }
]
```

```
[
   {
      "Statement":" statement ",
      "Parameters":[
         {
            " parametertype " : " parametervalue "
         }, ...]
   } , ...
]
```

パラメータ

#######

(必須) DynamoDB 用の PartiQL がサポートするステートメントです。

> **Note**
>
> - バッチ全体は、読み取りステートメントまたは書き込みステートメントのいずれかで構成する必要があります。1 つのバッチに両方を混在させることはできません。
> - `BatchExecuteStatement` と `BatchWriteItem` が実行できるステートメントの数は、バッチあたり 25 が上限です。

parametertype

(オプション) PartiQL ステートメントを指定するときにパラメータが使用された場合の DynamoDB タイプ。

parametervalue

(オプション) PartiQL ステートメントを指定するときにパラメータが使用された場合のパラメータの値。

例

AWS CLI

1. 次の json を partiql.json というファイルに保存します

```
[
   {
  "Statement": "INSERT INTO Music VALUE {'Artist':?,'SongTitle':?}",
   "Parameters": [{"S": "Acme Band"}, {"S": "Best Song"}]
 },
 {
  "Statement": "UPDATE Music SET AwardsWon=1, AwardDetail={'Grammys':[2020,
 2018]} WHERE Artist='Acme Band' AND SongTitle='PartiQL Rocks'"
    }
]
```

2. コマンドプロンプトで、次のコマンドを実行します。

```
aws dynamodb batch-execute-statement  --statements  file://partiql.json
```

Java

```
public class DynamoDBPartiqlBatch {

    public static void main(String[] args) {
        // Create the DynamoDB Client with the region you want
        AmazonDynamoDB dynamoDB = createDynamoDbClient("us-west-2");

        try {
            // Create BatchExecuteStatementRequest
```

```
            BatchExecuteStatementRequest batchExecuteStatementRequest =
createBatchExecuteStatementRequest();
            BatchExecuteStatementResult batchExecuteStatementResult =
dynamoDB.batchExecuteStatement(batchExecuteStatementRequest);
            System.out.println("BatchExecuteStatement successful.");
            // Handle batchExecuteStatementResult

        } catch (Exception e) {
            handleBatchExecuteStatementErrors(e);
        }
    }

    private static AmazonDynamoDB createDynamoDbClient(String region) {

        return AmazonDynamoDBClientBuilder.standard().withRegion(region).build();
    }

    private static BatchExecuteStatementRequest createBatchExecuteStatementRequest()
{
        BatchExecuteStatementRequest request = new BatchExecuteStatementRequest();

        // Create statements
        List<BatchStatementRequest> statements = getPartiQLBatchStatements();

        request.setStatements(statements);
        return request;
    }

    private static List<BatchStatementRequest> getPartiQLBatchStatements() {
        List<BatchStatementRequest> statements = new
ArrayList<BatchStatementRequest>();

        statements.add(new BatchStatementRequest()
                                .withStatement("INSERT INTO Music value
{'Artist':'Acme Band','SongTitle':'PartiQL Rocks'}"));

        statements.add(new BatchStatementRequest()
                                .withStatement("UPDATE Music set
AwardDetail.BillBoard=[2020] where Artist='Acme Band' and SongTitle='PartiQL
Rocks'"));

        return statements;
    }
```

```
   // Handles errors during BatchExecuteStatement execution. Use recommendations in
error messages below to add error handling specific to
   // your application use-case.
   private static void handleBatchExecuteStatementErrors(Exception exception) {
       try {
           throw exception;
       } catch (Exception e) {
           // There are no API specific errors to handle for BatchExecuteStatement,
common DynamoDB API errors are handled below
           handleCommonErrors(e);
       }
   }

   private static void handleCommonErrors(Exception exception) {
       try {
           throw exception;
       } catch (InternalServerErrorException isee) {
           System.out.println("Internal Server Error, generally safe to retry with
exponential back-off. Error: " + isee.getErrorMessage());
       } catch (RequestLimitExceededException rlee) {
           System.out.println("Throughput exceeds the current throughput limit for
your account, increase account level throughput before " +
               "retrying. Error: " + rlee.getErrorMessage());
       } catch (ProvisionedThroughputExceededException ptee) {
           System.out.println("Request rate is too high. If you're using a custom
retry strategy make sure to retry with exponential back-off. " +
               "Otherwise consider reducing frequency of requests or increasing
provisioned capacity for your table or secondary index. Error: " +
               ptee.getErrorMessage());
       } catch (ResourceNotFoundException rnfe) {
           System.out.println("One of the tables was not found, verify table exists
before retrying. Error: " + rnfe.getErrorMessage());
       } catch (AmazonServiceException ase) {
           System.out.println("An AmazonServiceException occurred, indicates that
the request was correctly transmitted to the DynamoDB " +
               "service, but for some reason, the service was not able to process
it, and returned an error response instead. Investigate and " +
               "configure retry strategy. Error type: " + ase.getErrorType() + ".
Error message: " + ase.getErrorMessage());
       } catch (AmazonClientException ace) {
           System.out.println("An AmazonClientException occurred, indicates that
the client was unable to get a response from DynamoDB " +
               "service, or the client was unable to parse the response from the
service. Investigate and configure retry strategy. "+
```

```
                "Error: " + ace.getMessage());
        } catch (Exception e) {
            System.out.println("An exception occurred, investigate and configure
 retry strategy. Error: " + e.getMessage());
        }
    }

}
```

DynamoDB 用 PartiQL における IAM セキュリティポリシー

以下のアクセス権限が必要です。

- DynamoDB 用の PartiQL を使用して項目を読み込むには、`dynamodb:PartiQLSelect` アクセス許可を、テーブルまたはインデックスに付与します。
- DynamoDB 用の PartiQL を使用して項目を挿入するには、`dynamodb:PartiQLInsert` アクセス許可をテーブルまたはインデックスに付与します。
- DynamoDB 用の PartiQL を使用して項目を更新するには、`dynamodb:PartiQLUpdate` アクセス許可をテーブルまたはインデックスに付与します。
- DynamoDB 用の PartiQL を使用して項目を削除するには、`dynamodb:PartiQLDelete` アクセス許可をテーブルまたはインデックスに付与します。

例: テーブルに対して、すべての DynamoDB 用の PartiQL ステートメントを許可する (選択、挿入、更新、削除)

次の IAM ポリシーは、すべての DynamoDB 用の PartiQL ステートメントをテーブルに対して実行するための、アクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
```

```
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
   ]
}
```

例: テーブルに対して、DynamoDB 用の PartiQL 選択ステートメントを許可する

次の IAM ポリシーは、特定のテーブルに対して `select` ステートメントを実行するためのアクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
   ]
}
```

例: インデックスに対して、DynamoDB 用の PartiQL select ステートメントを許可する

次の IAM ポリシーは、特定のインデックスに対して `select` ステートメントを実行するためのアクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music/index/index1"
         ]
      }
   ]
}
```

例: テーブルに対して、DynamoDB 用の PartiQL トランザクションステートメントを許可する

次の IAM ポリシーは、特定のテーブルに対してトランザクションステートメントだけを実行するためのアクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ],
         "Condition":{
            "StringEquals":{
               "dynamodb:EnclosingOperation":[
```

```
                "ExecuteTransaction"
            ]
         }
      }
   ]
}
```

例: テーブルに対して、DynamoDB 用の PartiQL 非トランザクションの読み込み/書き込みステートメントを許可し、PartiQL 読み込み/書き込みトランザクションをブロックします。

次の IAM ポリシーは、DynamoDB 用の PartiQL トランザクション読み込み/書き込みをブロックしている間に、DynamoDB 用の PartiQL 非トランザクションの読み込み/書き込みを実行するためのアクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ],
         "Condition":{
            "StringEquals":{
               "dynamodb:EnclosingOperation":[
                  "ExecuteTransaction"
               ]
            }
         }
      },
      {
         "Effect":"Allow",
```

```
         "Action":[
            "dynamodb:PartiQLInsert",
            "dynamodb:PartiQLUpdate",
            "dynamodb:PartiQLDelete",
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
         ]
      }
   ]
}
```

例: DynamoDB 用の PartiQL で ステートメントの選択を許可し、テーブル全体のスキャンを行うステートメントを拒否する

次の IAM ポリシーは、テーブル全体のスキャンを行う `select` ステートメントをブロックしている間に、特定のテーブルに対して `select` ステートメントを実行するためのアクセス許可を付与します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
         ],
         "Condition":{
            "Bool":{
               "dynamodb:FullTableScan":[
                  "true"
               ]
            }
         }
      },
```

```
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:PartiQLSelect"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/WatchList"
         ]
      }
   ]
}
```

項目の操作: Java

AWS SDK for Java ドキュメント API を使用して、テーブル内の Amazon DynamoDB 項目に対して、一般的な作成、読み込み、更新、削除 (CRUD) のオペレーションを実行できます。

> **Note**
>
> SDK for Java には、オブジェクト永続性モデルも用意されています。このモデルにより、クライアント側のクラスを DynamoDB テーブルにマッピングすることができます。この方法により、記述する必要のあるコードの量を減らすことができます。詳細については、「」を参照してくださいJava 1.x: DynamoDBMapper

このセクションには、いくつかの Java ドキュメント API 項目のアクションを実行する Java サンプルといくつかの完全な操作例が含まれています。

トピック

- 項目の置換
- 項目の取得
- バッチ書き込み: 複数の項目の置換および削除
- バッチ取得: 複数の項目の取得
- 項目の更新
- 項目の削除
- 例: AWS SDK for Java ドキュメント API を使用した CRUD オペレーション
- 例: AWS SDK for Java ドキュメント API を使用したバッチオペレーション

- 例: AWS SDK for Java ドキュメント API を使用したバイナリタイプ属性の処理

項目の置換

putItem メソッドによって、項目をテーブルに格納します。項目が存在する場合、その項目全体が置き換えられます。項目全体を置き換える代わりに固有の属性のみを更新する場合は、updateItem メソッドを使用できます。詳細については、「」を参照してください項目の更新

Java v2

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval> <awards>
 <awardsval> <Songtitle> <songtitleval>

                Where:
                    tableName - The Amazon DynamoDB table in which an item is placed
 (for example, Music3).
```

```
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example, Happy
Day).
                **Warning** This program will  place an item that you specify into a
table!
                """;

        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }

    public static void putItemInTable(DynamoDbClient ddb,
```

```
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle, AttributeValue.builder().s(songTitleVal).build());
        itemValues.put(albumTitle,
 AttributeValue.builder().s(albumTitleValue).build());
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The request
 id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

Java v1

以下のステップに従ってください。

1. DynamoDB クラスのインスタンスを作成します。
2. 操作対象のテーブルを表すために、Table クラスのインスタンスを作成します。
3. 新しい項目を表す Item クラスのインスタンスを作成します。新しい項目のプライマリキーと属性を指定する必要があります。
4. 前の手順で作成した putItem を使用して、Table オブジェクトの Item メソッドを呼び出します。

以下の Java コード例は、前述のタスクの例です。このコードでは、ProductCatalog テーブルに新しい項目を書き込みます。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

// Build a list of related items
List<Number> relatedItems = new ArrayList<Number>();
relatedItems.add(341);
relatedItems.add(472);
relatedItems.add(649);

//Build a map of product pictures
Map<String, String> pictures = new HashMap<String, String>();
pictures.put("FrontView", "http://example.com/products/123_front.jpg");
pictures.put("RearView", "http://example.com/products/123_rear.jpg");
pictures.put("SideView", "http://example.com/products/123_left_side.jpg");

//Build a map of product reviews
Map<String, List<String>> reviews = new HashMap<String, List<String>>();

List<String> fiveStarReviews = new ArrayList<String>();
fiveStarReviews.add("Excellent! Can't recommend it highly enough!  Buy it!");
fiveStarReviews.add("Do yourself a favor and buy this");
reviews.put("FiveStar", fiveStarReviews);

List<String> oneStarReviews = new ArrayList<String>();
oneStarReviews.add("Terrible product!  Do not buy this.");
reviews.put("OneStar", oneStarReviews);
```

```
// Build the item
Item item = new Item()
    .withPrimaryKey("Id", 123)
    .withString("Title", "Bicycle 123")
    .withString("Description", "123 description")
    .withString("BicycleType", "Hybrid")
    .withString("Brand", "Brand-Company C")
    .withNumber("Price", 500)
    .withStringSet("Color",  new HashSet<String>(Arrays.asList("Red", "Black")))
    .withString("ProductCategory", "Bicycle")
    .withBoolean("InStock", true)
    .withNull("QuantityOnHand")
    .withList("RelatedItems", relatedItems)
    .withMap("Pictures", pictures)
    .withMap("Reviews", reviews);

// Write the item to the table
PutItemOutcome outcome = table.putItem(item);
```

前の例では、項目にはスカラー (String、Number、Boolean、Null)、セット (String Set)、ドキュメントタイプ (List、Map) があります。

オプションパラメータの指定

必須のパラメータに加え、putItem メソッドにはオプションパラメータも指定できます。たとえば、以下の Java コード例では、オプションパラメータを使用して、項目のアップロード条件を指定します。指定した条件を満たさない場合は、AWS SDK for Java が ConditionalCheckFailedException をスローします。このコード例では、putItem メソッドに以下のオプションパラメータを指定します。

- リクエストの条件を定義する ConditionExpression。このコードでは、特定の値に等しい ISBN 属性が既存の項目にある場合にのみ同じプライマリキーを持つ既存の項目を置き換えるという条件を定義します。
- 条件で使用される ExpressionAttributeValues のマップ。この場合、必要な置き換えは 1 つだけです。条件式のプレースホルダー :val が、実行時に、チェックする実際の ISBN 値に置き換えられます。

以下の例では、これらのオプションパラメータを使用して新しい書籍項目を追加します。

Example

```
Item item = new Item()
    .withPrimaryKey("Id", 104)
    .withString("Title", "Book 104 Title")
    .withString("ISBN", "444-4444444444")
    .withNumber("Price", 20)
    .withStringSet("Authors",
        new HashSet<String>(Arrays.asList("Author1", "Author2")));

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", "444-4444444444");

PutItemOutcome outcome = table.putItem(
    item,
    "ISBN = :val", // ConditionExpression parameter
    null,          // ExpressionAttributeNames parameter - we're not using it for this
 example
    expressionAttributeValues);
```

PutItem および JSON ドキュメント

JSON ドキュメントは、DynamoDB テーブルに属性として格納できます。これを行うには、withJSON 項目の Item メソッドを使用します。このメソッドは、JSON ドキュメントを解析し、各要素を DynamoDB のネイティブデータ型にマッピングします。

特定の製品の注文を処理できるベンダーを含む、次の JSON ドキュメントを格納するとします。

Example

```
{
    "V01": {
        "Name": "Acme Books",
        "Offices": [ "Seattle" ]
    },
    "V02": {
        "Name": "New Publishers, Inc.",
        "Offices": ["London", "New York"
        ]
    },
    "V03": {
        "Name": "Better Buy Books",
        "Offices": [ "Tokyo", "Los Angeles", "Sydney"
```

```
        ]
    }
}
```

withJSON メソッドを使用して、ProductCatalog という名前の Map 属性でこれを VendorInfo テーブルに格納することができます。次の Java コード例はそれを行う方法を示しています。

```
// Convert the document into a String.  Must escape all double-quotes.
String vendorDocument = "{"
    + "    \"V01\": {"
    + "        \"Name\": \"Acme Books\","
    + "        \"Offices\": [ \"Seattle\" ]"
    + "    },"
    + "    \"V02\": {"
    + "        \"Name\": \"New Publishers, Inc.\","
    + "        \"Offices\": [ \"London\", \"New York\"" + "]" + "},"
    + "    \"V03\": {"
    + "        \"Name\": \"Better Buy Books\","
    +          "\"Offices\": [ \"Tokyo\", \"Los Angeles\", \"Sydney\""
    + "            ]"
    + "        }"
    + "    }";

Item item = new Item()
    .withPrimaryKey("Id", 210)
    .withString("Title", "Book 210 Title")
    .withString("ISBN", "210-2102102102")
    .withNumber("Price", 30)
    .withJSON("VendorInfo", vendorDocument);

PutItemOutcome outcome = table.putItem(item);
```

項目の取得

単一の項目を取り出すには、getItem オブジェクトの Table メソッドを使用します。以下のステップに従ってください。

1. DynamoDB クラスのインスタンスを作成します。
2. 操作対象のテーブルを表すために、Table クラスのインスタンスを作成します。
3. getItem インスタンスの Table メソッドを呼び出します。取り出す項目のプライマリキーを指定する必要があります。

以下の Java コード例は、前述のステップの例です。このコードでは、指定したパーティションキーを持つ項目を取得します。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Item item = table.getItem("Id", 210);
```

オプションパラメータの指定

必須のパラメータに加え、getItem メソッドにはオプションパラメータも指定できます。たとえば、以下の Java コード例では、オプションメソッドを使用して、特定の属性リストのみを取り出します。また、強い整合性のある戻り値をリクエストします。読み込み整合性の詳細については、「DynamoDB の読み取り整合性」を参照してください。

ProjectionExpression を使用すると、項目全体ではなく特定の属性または要素のみを取り出すことができます。ProjectionExpression は、ドキュメントのパスを使用して最上位の属性または入れ子になった属性を指定できます。詳細については、「」を参照してくださいDynamoDB でのプロジェクション式の使用

getItem メソッドのパラメータでは、読み取りの一貫性を指定できません。ただし、GetItemSpec を作成して、低レベル GetItem オペレーションへの入力のすべてに対するフルアクセスを提供できます。以下のコード例では、GetItemSpec を作成し、その仕様を getItem メソッドへの入力として使用します。

Example

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 206)
    .withProjectionExpression("Id, Title, RelatedItems[0], Reviews.FiveStar")
    .withConsistentRead(true);

Item item = table.getItem(spec);

System.out.println(item.toJSONPretty());
```

Item を人間が読める形式で出力するには、toJSONPretty メソッドを使用します。前のサンプルからの出力は、次のようになります。

```
{
  "RelatedItems" : [ 341 ],
  "Reviews" : {
    "FiveStar" : [ "Excellent! Can't recommend it highly enough! Buy it!", "Do yourself
 a favor and buy this" ]
  },
  "Id" : 123,
  "Title" : "20-Bicycle 123"
}
```

GetItem および JSON ドキュメント

PutItem および JSON ドキュメント セクションでは、Map という名前の VendorInfo 属性に JSON ドキュメントを格納します。getItem メソッドを使用して、ドキュメント全体を JSON 形式で取得できます。または、ドキュメントパスの表記を使用して、ドキュメントの一部の要素のみを取得することもできます。次の Java コード例は、この手法を示しています。

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210);

System.out.println("All vendor info:");
spec.withProjectionExpression("VendorInfo");
System.out.println(table.getItem(spec).toJSON());

System.out.println("A single vendor:");
spec.withProjectionExpression("VendorInfo.V03");
System.out.println(table.getItem(spec).toJSON());

System.out.println("First office location for this vendor:");
spec.withProjectionExpression("VendorInfo.V03.Offices[0]");
System.out.println(table.getItem(spec).toJSON());
```

前のサンプルからの出力は、次のようになります。

```
All vendor info:
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los
 Angeles","Sydney"]},"V02":{"Name":"New Publishers, Inc.","Offices":["London","New
 York"]},"V01":{"Name":"Acme Books","Offices":["Seattle"]}}}
A single vendor:
{"VendorInfo":{"V03":{"Name":"Better Buy Books","Offices":["Tokyo","Los
 Angeles","Sydney"]}}}
```

```
First office location for a single vendor:
{"VendorInfo":{"V03":{"Offices":["Tokyo"]}}}
```

> **Note**
>
> toJSON メソッドを使用して、任意の項目 (またはその属性) を JSON 形式文字列に変換できます。次のコード例は、最上位の属性と入れ子になった属性を複数取り出し、JSON として結果を出力します。
>
> ```
> GetItemSpec spec = new GetItemSpec()
>     .withPrimaryKey("Id", 210)
>     .withProjectionExpression("VendorInfo.V01, Title, Price");
>
> Item item = table.getItem(spec);
> System.out.println(item.toJSON());
> ```
>
> 出力は次のようになります。
>
> ```
> {"VendorInfo":{"V01":{"Name":"Acme Books","Offices":
> ["Seattle"]}},"Price":30,"Title":"Book 210 Title"}
> ```

バッチ書き込み: 複数の項目の置換および削除

バッチ書き込みは、複数の項目の書き込みと削除をバッチで行うことを意味します。batchWriteItem メソッドによって、単一の コール内にある 1 つまたは複数のテーブルから複数の項目を置換および削除できます。以下に、AWS SDK for Java ドキュメント API を使用して複数の項目を入力および削除する手順を示します。

1. DynamoDB クラスのインスタンスを作成します。
2. テーブルのすべての入力および削除オペレーションを記述する TableWriteItems クラスのインスタンスを作成します。1 つのバッチ書き込みオペレーションで複数のテーブルに書き込む場合、テーブルごとに TableWriteItems インスタンスを 1 つずつ作成する必要があります。
3. 前の手順で作成した batchWriteItem オブジェクトを指定して、TableWriteItems メソッドを呼び出します。
4. 応答を処理します。返された未処理のリクエスト項目が応答内に存在していたかどうかをチェックする必要があります。これは、プロビジョニングされたスループットクォータまたは他の何らかの一時的エラーに達する場合に、発生する可能性があります。また、DynamoDB によって、

リクエストのサイズ、およびリクエスト内で指定できるオペレーションの数が制限されます。これらの制限を超えると、DynamoDB によってリクエストが却下されます。詳細については、「Amazon DynamoDB のクォータ」を参照してください。

以下の Java コード例は、前述のステップの例です。この例では、2 つのテーブル (batchWriteItem と Forum) で Thread オペレーションを実行します。対応する TableWriteItems オブジェクトは、次のアクションを定義します。

- Forum テーブル内で項目を配置します。
- Thread テーブル内で項目を入力および削除します。

その後、コードは batchWriteItem を呼び出してオペレーションを実行します。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableWriteItems forumTableWriteItems = new TableWriteItems("Forum")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("Name", "Amazon RDS")
            .withNumber("Threads", 0));

TableWriteItems threadTableWriteItems = new TableWriteItems("Thread")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("ForumName","Amazon RDS","Subject","Amazon RDS Thread 1")
    .withHashAndRangeKeysToDelete("ForumName","Some partition key value", "Amazon S3",
 "Some sort key value");

BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
 threadTableWriteItems);

// Code for checking unprocessed items is omitted in this example
```

実例については、「例: AWS SDK for Java ドキュメント API を使用したバッチ書き込みオペレーション」を参照してください。

バッチ取得: 複数の項目の取得

batchGetItem メソッドによって、1 つまたは複数のテーブルから複数の項目を取得できます。単一の項目を取り出すために、getItem メソッドを使用できます。

以下のステップに従ってください。

1. DynamoDB クラスのインスタンスを作成します。
2. テーブルから取り出すプライマリキーのリストを記述する TableKeysAndAttributes クラスのインスタンスを作成します。1 つのバッチ取得オペレーションで複数のテーブルから読み込む場合、テーブルごとに TableKeysAndAttributes インスタンスを 1 つずつ作成する必要があります。
3. 前の手順で作成した batchGetItem オブジェクトを指定して、TableKeysAndAttributes メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。この例では、Forum テーブル内の 2 つの項目、および Thread テーブル内の 3 つの項目を取り出します。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

    TableKeysAndAttributes forumTableKeysAndAttributes = new
 TableKeysAndAttributes(forumTableName);
    forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

TableKeysAndAttributes threadTableKeysAndAttributes = new
 TableKeysAndAttributes(threadTableName);
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
    "Amazon DynamoDB","DynamoDB Thread 1",
    "Amazon DynamoDB","DynamoDB Thread 2",
    "Amazon S3","S3 Thread 1");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(
    forumTableKeysAndAttributes, threadTableKeysAndAttributes);

for (String tableName : outcome.getTableItems().keySet()) {
    System.out.println("Items in table " + tableName);
    List<Item> items = outcome.getTableItems().get(tableName);
    for (Item item : items) {
```

```
        System.out.println(item);
    }
}
```

オプションパラメータの指定

必須のパラメータに加え、batchGetItem を使用する場合はオプションパラメータも指定できます。たとえば、定義した ProjectionExpression ごとに TableKeysAndAttributes を指定できます。これにより、テーブルから取り出す属性を指定することができます。

次のサンプルコードでは、Forum から 2 つの項目を取得します。withProjectionExpression パラメータは、Threads 属性のみを取得することを指定します。

Example

```
TableKeysAndAttributes forumTableKeysAndAttributes = new
 TableKeysAndAttributes("Forum")
    .withProjectionExpression("Threads");

forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(forumTableKeysAndAttributes);
```

項目の更新

updateItem オブジェクトの Table メソッドは、既存の属性値の更新、新しい属性の追加、または既存の項目からの属性の削除を実行できます。

updateItem メソッドは、以下のように動作します。

- 項目が存在しない (指定されたプライマリキーを持つ項目がテーブル内にない) 場合、updateItem はテーブルに新しい項目を追加します
- 項目が存在する場合、updateItem は UpdateExpression パラメータで指定されたとおりに更新を実行します。

> **Note**
>
> putItem を使用して項目を「更新」することもできます。たとえば、putItem を呼び出して項目をテーブルに追加したが、指定されたプライマリキーを持つ項目がすでに存在する場

合、putItem は項目全体を置き換えます。入力で指定されていない属性が既存の項目内にある場合、putItem は項目からそれらの属性を削除します。
一般に、項目属性を変更するときは必ず updateItem を使用することをお勧めします。updateItem メソッドは、入力で指定した項目属性のみを変更し、項目内の他の属性は変更されません。

以下のステップに従ってください。

1. 操作対象のテーブルを表すために、Table クラスのインスタンスを作成します。
2. updateTable インスタンスの Table メソッドを呼び出します。変更する属性とその変更方法を記述する UpdateExpression と同時に、取り出す項目のプライマリーキーを指定する必要があります。

以下の Java コード例は、前述のタスクの例です。このコードでは、ProductCatalog テーブルの書籍項目を更新します。この例では、Authors のセットに著者を追加し、既存の ISBN 属性を削除します。また、価格を 1 引き下げます。

ExpressionAttributeValues マップは UpdateExpression で使用されます。プレースホルダー :val1 および :val2 は、実行時に、Authors と Price の実際の値に置き換えられます。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#A", "Authors");
expressionAttributeNames.put("#P", "Price");
expressionAttributeNames.put("#I", "ISBN");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1",
    new HashSet<String>(Arrays.asList("Author YY","Author ZZ")));
expressionAttributeValues.put(":val2", 1);   //Price

UpdateItemOutcome outcome =  table.updateItem(
    "Id",          // key attribute name
    101,           // key attribute value
    "add #A :val1 set #P = #P - :val2 remove #I", // UpdateExpression
```

```
    expressionAttributeNames,
    expressionAttributeValues);
```

オプションパラメータの指定

必須のパラメータに加えて、更新を実行するために満たす必要がある条件も含めて、updateItem メソッドのオプションパラメータを指定することもできます。指定した条件を満たさない場合は、AWS SDK for Java が ConditionalCheckFailedException をスローします。たとえば、以下の Java コード例では、書籍項目の価格を条件付きで 25 に更新します。現在の価格が 20 である場合にのみ価格を更新する必要があることを示す ConditionExpression を指定します。

Example

```
Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#P", "Price");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1", 25);  // update Price to 25...
expressionAttributeValues.put(":val2", 20);  //...but only if existing Price is 20

UpdateItemOutcome outcome = table.updateItem(
    new PrimaryKey("Id",101),
    "set #P = :val1", // UpdateExpression
    "#P = :val2",     // ConditionExpression
    expressionAttributeNames,
    expressionAttributeValues);
```

アトミックカウンタ

updateItem を使用してアトミックカウンターを実装できます。アトミックカウンターでは、他の書き込みリクエストを妨げることなく、既存の属性の値をインクリメントまたはデクリメントします。アトミックカウンターをインクリメントするには、UpdateExpression を使用して、set アクションで既存の Number 型の属性に数値を加算します。

次のサンプルはこのアトミックカウンターを示しており、Quantity 属性を 1 ずつインクリメントさせています。さらに、ExpressionAttributeNames での UpdateExpression パラメータの使用方法も示します。

```
Table table = dynamoDB.getTable("ProductCatalog");
```

```
Map<String,String> expressionAttributeNames = new HashMap<String,String>();
expressionAttributeNames.put("#p", "PageCount");

Map<String,Object> expressionAttributeValues = new HashMap<String,Object>();
expressionAttributeValues.put(":val", 1);

UpdateItemOutcome outcome = table.updateItem(
    "Id", 121,
    "set #p = #p + :val",
    expressionAttributeNames,
    expressionAttributeValues);
```

項目の削除

deleteItem メソッドによって、テーブルから項目を削除します。削除する項目のプライマリキーを入力する必要があります。

以下のステップに従ってください。

1. DynamoDB クライアントのインスタンスを作成します。
2. 削除する項目のキーを指定して、deleteItem メソッドを呼び出します。

次の Java サンプルは、このタスクを示しています。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

DeleteItemOutcome outcome = table.deleteItem("Id", 101);
```

オプションパラメータの指定

deleteItem のオプションパラメータを指定できます。たとえば、次の Java コード例は、ConditionExpression 内の書籍項目は書籍が絶版になった (ProductCatalog 属性が false) 場合のみ削除可能であることを示す InPublication を指定します。

Example

```
Map<String,Object> expressionAttributeValues = new HashMap<String,Object>();
```

```
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("Id",103,
    "InPublication = :val",
    null, // ExpressionAttributeNames - not used in this example
    expressionAttributeValues);
```

例: AWS SDK for Java ドキュメント API を使用した CRUD オペレーション

以下のコード例は、Amazon DynamoDB 項目に対する CRUD オペレーションの例です。この例では、項目の作成、項目の取得、さまざまな更新の実行、さらに項目の削除を行います。

> **Note**
>
> SDK for Java には、オブジェクト永続性モデルも用意されています。このモデルにより、クライアント側のクラスを DynamoDB テーブルにマッピングできます。この方法により、記述する必要のあるコードの量を減らすことができます。詳細については、「Java 1.x: DynamoDBMapper」を参照してください。

> **Note**
>
> このコード例では、アカウントの DynamoDB に対し、DynamoDB でのコード例用のテーブルの作成とデータのロード セクションの手順に従ってデータが既にロードされていることを前提としています。
> 以下の例を実行するための詳しい手順については、「Java コードの例」を参照してください。

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
```

```
import com.amazonaws.services.dynamodbv2.document.DeleteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class DocumentAPIItemCRUDExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws IOException {

        createItems();

        retrieveItem();

        // Perform various updates.
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Delete the item.
        deleteItem();

    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tableName);
        try {

            Item item = new Item().withPrimaryKey("Id", 120).withString("Title", "Book
 120 Title")
                    .withString("ISBN", "120-1111111111")
                    .withStringSet("Authors", new
 HashSet<String>(Arrays.asList("Author12", "Author22")))
```

```
                    .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
                    .withBoolean("InPublication", false).withString("ProductCategory",
"Book");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 121).withString("Title", "Book 121
Title")
                    .withString("ISBN", "121-1111111111")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author21", "Author 22")))
                    .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Create items failed.");
            System.err.println(e.getMessage());

        }
    }

    private static void retrieveItem() {
        Table table = dynamoDB.getTable(tableName);

        try {

            Item item = table.getItem("Id", 120, "Id, ISBN, Title, Authors", null);

            System.out.println("Printing item after retrieving it....");
            System.out.println(item.toJSONPretty());

        } catch (Exception e) {
            System.err.println("GetItem failed.");
            System.err.println(e.getMessage());
        }

    }

    private static void updateAddNewAttribute() {
        Table table = dynamoDB.getTable(tableName);
```

```
        try {

            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
121)
                    .withUpdateExpression("set #na = :val1").withNameMap(new
NameMap().with("#na", "NewAttribute"))
                    .withValueMap(new ValueMap().withString(":val1", "Some value"))
                    .withReturnValues(ReturnValue.ALL_NEW);

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after adding new attribute...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Failed to add new attribute in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void updateMultipleAttributes() {

        Table table = dynamoDB.getTable(tableName);

        try {

            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
120)
                    .withUpdateExpression("add #a :val1 set #na=:val2")
                    .withNameMap(new NameMap().with("#a", "Authors").with("#na",
"NewAttribute"))
                    .withValueMap(
                            new ValueMap().withStringSet(":val1", "Author YY", "Author
ZZ").withString(":val2",
                                    "someValue"))
                    .withReturnValues(ReturnValue.ALL_NEW);

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after multiple attribute update...");
            System.out.println(outcome.getItem().toJSONPretty());
```

```
        } catch (Exception e) {
            System.err.println("Failed to update multiple attributes in " + tableName);
            System.err.println(e.getMessage());

        }
    }

    private static void updateExistingAttributeConditionally() {

        Table table = dynamoDB.getTable(tableName);

        try {

            // Specify the desired price (25.00) and also the condition (price =
            // 20.00)

            UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id",
120)
                    .withReturnValues(ReturnValue.ALL_NEW).withUpdateExpression("set #p
= :val1")
                    .withConditionExpression("#p = :val2").withNameMap(new
NameMap().with("#p", "Price"))
                    .withValueMap(new ValueMap().withNumber(":val1",
25).withNumber(":val2", 20));

            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

            // Check the response.
            System.out.println("Printing item after conditional update to new
attribute...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Error updating item in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void deleteItem() {

        Table table = dynamoDB.getTable(tableName);

        try {
```

```
            DeleteItemSpec deleteItemSpec = new DeleteItemSpec().withPrimaryKey("Id",
 120)
                    .withConditionExpression("#ip = :val").withNameMap(new
 NameMap().with("#ip", "InPublication"))
                    .withValueMap(new ValueMap().withBoolean(":val",
 false)).withReturnValues(ReturnValue.ALL_OLD);

            DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);

            // Check the response.
            System.out.println("Printing item that was deleted...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Error deleting item in " + tableName);
            System.err.println(e.getMessage());
        }
    }
}
```

例: AWS SDK for Java ドキュメント API を使用したバッチオペレーション

このセクションでは、AWS SDK for Java ドキュメント API を使用した Amazon DynamoDB でのバッチ書き込みおよびバッチ取得オペレーションの例を示します。

> **Note**
>
> SDK for Java には、オブジェクト永続性モデルも用意されています。このモデルにより、クライアント側のクラスを DynamoDB テーブルにマッピングできます。この方法により、記述する必要のあるコードの量を減らすことができます。詳細については、「」を参照してくださいJava 1.x: DynamoDBMapper

トピック

- 例: AWS SDK for Java ドキュメント API を使用したバッチ書き込みオペレーション
- 例: AWS SDK for Java ドキュメント API を使用したバッチ取得オペレーション

例: AWS SDK for Java ドキュメント API を使用したバッチ書き込みオペレーション

以下の Java コード例では、batchWriteItem メソッドを使用して、以下の置換および削除のオペレーションを実行します。

- Forum テーブル内で 1 つの項目を配置します。
- Thread テーブルに対して 1 つの項目を配置および削除します。

バッチの書き込みリクエストを作成すると、1 つまたは複数のテーブルに対して多数の置換リクエストと削除リクエストを指定できます。ただし、batchWriteItem では、1 回のバッチ書き込みオペレーションで可能なバッチ書き込みリクエストのサイズ、置換および削除のオペレーションの数を制限しています。これらの制限を超えるリクエストは却下されます。プロビジョニングされたスループットがテーブルに不足しているためにこのリクエストを処理できない場合は、応答時に未処理のリクエスト項目が返されます。

以下の例では、未処理のリクエスト項目がないか、応答を確認します。未処理のリクエスト項目がある場合は、batchWriteItem リクエストをループバックして再送信します。このガイドの例に従っていれば、Forum テーブルおよび Thread テーブルは作成済みです。プログラムで、これらのテーブルを作成し、サンプルデータをアップロードすることもできます。詳細については、「」を参照してくださいAWS SDK for Java を使用してサンプルテーブルを作成してデータをアップロードする

以下の例をテストするための詳細な手順については、「Java コードの例」を参照してください。

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchWriteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableWriteItems;
import com.amazonaws.services.dynamodbv2.model.WriteRequest;
```

```
public class DocumentAPIBatchWrite {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {

        writeMultipleItemsBatchWrite();

    }

    private static void writeMultipleItemsBatchWrite() {
        try {

            // Add a new item to Forum
            TableWriteItems forumTableWriteItems = new
 TableWriteItems(forumTableName) // Forum
                    .withItemsToPut(new Item().withPrimaryKey("Name", "Amazon
 RDS").withNumber("Threads", 0));

            // Add a new item, and delete an existing item, from Thread
            // This table has a partition key and range key, so need to specify
            // both of them
            TableWriteItems threadTableWriteItems = new
 TableWriteItems(threadTableName)
                    .withItemsToPut(
                            new Item().withPrimaryKey("ForumName", "Amazon RDS",
 "Subject", "Amazon RDS Thread 1")
                                    .withString("Message", "ElastiCache Thread 1
 message")
                                    .withStringSet("Tags", new
 HashSet<String>(Arrays.asList("cache", "in-memory"))))
                    .withHashAndRangeKeysToDelete("ForumName", "Subject", "Amazon S3",
 "S3 Thread 100");

            System.out.println("Making the request.");
            BatchWriteItemOutcome outcome =
 dynamoDB.batchWriteItem(forumTableWriteItems, threadTableWriteItems);

            do {
```

```
                // Check for unprocessed keys which could happen if you exceed
                // provisioned throughput

                Map<String, List<WriteRequest>> unprocessedItems =
 outcome.getUnprocessedItems();

                if (outcome.getUnprocessedItems().size() == 0) {
                    System.out.println("No unprocessed items found");
                } else {
                    System.out.println("Retrieving the unprocessed items");
                    outcome = dynamoDB.batchWriteItemUnprocessed(unprocessedItems);
                }

            } while (outcome.getUnprocessedItems().size() > 0);

        } catch (Exception e) {
            System.err.println("Failed to retrieve items: ");
            e.printStackTrace(System.err);
        }

    }

}
```

例: AWS SDK for Java ドキュメント API を使用したバッチ取得オペレーション

以下の Java コード例では、batchGetItem メソッドを使用して、Forum テーブルおよび Thread テーブルから複数の項目を取り出します。BatchGetItemRequest は、取得する各項目のテーブル名とキーのリストを指定します。この例では、取得した項目を印刷して応答を処理します。

> **Note**
>
> このコード例では、アカウントの DynamoDB に対し、DynamoDB でのコード例用のテーブルの作成とデータのロード セクションの手順に従ってデータが既にロードされていることを前提としています。
> 以下の例を実行するための詳しい手順については、「Java コードの例」を参照してください。

Example

```
package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchGetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableKeysAndAttributes;
import com.amazonaws.services.dynamodbv2.model.KeysAndAttributes;

public class DocumentAPIBatchGet {
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {
        retrieveMultipleItemsBatchGet();
    }

    private static void retrieveMultipleItemsBatchGet() {

        try {

            TableKeysAndAttributes forumTableKeysAndAttributes = new
 TableKeysAndAttributes(forumTableName);
            // Add a partition key
            forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name", "Amazon S3",
 "Amazon DynamoDB");

            TableKeysAndAttributes threadTableKeysAndAttributes = new
 TableKeysAndAttributes(threadTableName);
            // Add a partition key and a sort key
            threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName",
 "Subject", "Amazon DynamoDB",
```

```
                    "DynamoDB Thread 1", "Amazon DynamoDB", "DynamoDB Thread 2",
 "Amazon S3", "S3 Thread 1");

            System.out.println("Making the request.");

            BatchGetItemOutcome outcome =
 dynamoDB.batchGetItem(forumTableKeysAndAttributes,
                    threadTableKeysAndAttributes);

            Map<String, KeysAndAttributes> unprocessed = null;

            do {
                for (String tableName : outcome.getTableItems().keySet()) {
                    System.out.println("Items in table " + tableName);
                    List<Item> items = outcome.getTableItems().get(tableName);
                    for (Item item : items) {
                        System.out.println(item.toJSONPretty());
                    }
                }

                // Check for unprocessed keys which could happen if you exceed
                // provisioned
                // throughput or reach the limit on response size.
                unprocessed = outcome.getUnprocessedKeys();

                if (unprocessed.isEmpty()) {
                    System.out.println("No unprocessed keys found");
                } else {
                    System.out.println("Retrieving the unprocessed keys");
                    outcome = dynamoDB.batchGetItemUnprocessed(unprocessed);
                }

            } while (!unprocessed.isEmpty());

        } catch (Exception e) {
            System.err.println("Failed to retrieve items.");
            System.err.println(e.getMessage());
        }

    }

}
```

例: AWS SDK for Java ドキュメント API を使用したバイナリタイプ属性の処理

以下の Java コード例は、バイナリタイプ属性の処理の例です。この例では、Reply テーブルに項目を追加します。この項目には、圧縮データを格納するバイナリタイプ属性 (ExtendedMessage) などがあります。また、この例では、項目を取得し、すべての属性値を印刷します。説明のため、この例では GZIPOutputStream クラスを使用して、サンプルストリームを圧縮し、圧縮したデータを ExtendedMessage 属性に割り当てます。バイナリ属性が取得されると、GZIPInputStream クラスを使用して展開されます。

> **Note**
>
> SDK for Java には、オブジェクト永続性モデルも用意されています。このモデルにより、クライアント側のクラスを DynamoDB テーブルにマッピングできます。この方法により、記述する必要のあるコードの量を減らすことができます。詳細については、「」を参照してくださいJava 1.x: DynamoDBMapper

DynamoDB でのコード例用のテーブルの作成とデータのロード セクションの手順に従っていれば、Reply テーブルは作成済みです。このテーブルは、プログラムで作成することもできます。詳細については、「」を参照してくださいAWS SDK for Java を使用してサンプルテーブルを作成してデータをアップロードする

以下の例をテストするための詳細な手順については、「Java コードの例」を参照してください。

Example

```
package com.amazonaws.codesamples.document;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimeZone;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
```

```
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;

public class DocumentAPIItemBinaryExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "Reply";
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    public static void main(String[] args) throws IOException {
        try {

            // Format the primary key values
            String threadId = "Amazon DynamoDB#DynamoDB Thread 2";

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
            String replyDateTime = dateFormatter.format(new Date());

            // Add a new reply with a binary attribute type
            createItem(threadId, replyDateTime);

            // Retrieve the reply with a binary attribute type
            retrieveItem(threadId, replyDateTime);

            // clean up by deleting the item
            deleteItem(threadId, replyDateTime);
        } catch (Exception e) {
            System.err.println("Error running the binary attribute type example: " +
 e);
            e.printStackTrace(System.err);
        }
    }

    public static void createItem(String threadId, String replyDateTime) throws
 IOException {

        Table table = dynamoDB.getTable(tableName);

        // Craft a long message
```

```
        String messageInput = "Long message to be compressed in a lengthy forum reply";

        // Compress the long message
        ByteBuffer compressedMessage = compressString(messageInput.toString());

        table.putItem(new Item().withPrimaryKey("Id",
threadId).withString("ReplyDateTime", replyDateTime)
                .withString("Message", "Long message
follows").withBinary("ExtendedMessage", compressedMessage)
                .withString("PostedBy", "User A"));
    }

    public static void retrieveItem(String threadId, String replyDateTime) throws
IOException {

        Table table = dynamoDB.getTable(tableName);

        GetItemSpec spec = new GetItemSpec().withPrimaryKey("Id", threadId,
"ReplyDateTime", replyDateTime)
                .withConsistentRead(true);

        Item item = table.getItem(spec);

        // Uncompress the reply message and print
        String uncompressed =
uncompressString(ByteBuffer.wrap(item.getBinary("ExtendedMessage")));

        System.out.println("Reply message:\n" + " Id: " + item.getString("Id") + "\n" +
" ReplyDateTime: "
                + item.getString("ReplyDateTime") + "\n" + " PostedBy: " +
item.getString("PostedBy") + "\n"
                + " Message: "
                + item.getString("Message") + "\n" + " ExtendedMessage (uncompressed):
" + uncompressed + "\n");
    }

    public static void deleteItem(String threadId, String replyDateTime) {

        Table table = dynamoDB.getTable(tableName);
        table.deleteItem("Id", threadId, "ReplyDateTime", replyDateTime);
    }

    private static ByteBuffer compressString(String input) throws IOException {
        // Compress the UTF-8 encoded String into a byte[]
```

```
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream os = new GZIPOutputStream(baos);
        os.write(input.getBytes("UTF-8"));
        os.close();
        baos.close();
        byte[] compressedBytes = baos.toByteArray();

        // The following code writes the compressed bytes to a ByteBuffer.
        // A simpler way to do this is by simply calling
        // ByteBuffer.wrap(compressedBytes);
        // However, the longer form below shows the importance of resetting the
        // position of the buffer
        // back to the beginning of the buffer if you are writing bytes directly
        // to it, since the SDK
        // will consider only the bytes after the current position when sending
        // data to DynamoDB.
        // Using the "wrap" method automatically resets the position to zero.
        ByteBuffer buffer = ByteBuffer.allocate(compressedBytes.length);
        buffer.put(compressedBytes, 0, compressedBytes.length);
        buffer.position(0); // Important: reset the position of the ByteBuffer
                            // to the beginning
        return buffer;
    }

    private static String uncompressString(ByteBuffer input) throws IOException {
        byte[] bytes = input.array();
        ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPInputStream is = new GZIPInputStream(bais);

        int chunkSize = 1024;
        byte[] buffer = new byte[chunkSize];
        int length = 0;
        while ((length = is.read(buffer, 0, chunkSize)) != -1) {
            baos.write(buffer, 0, length);
        }

        String result = new String(baos.toByteArray(), "UTF-8");

        is.close();
        baos.close();
        bais.close();

        return result;
```

```
    }
}
```

項目の操作: .NET

AWS SDK for .NET 低レベル API を使用して、テーブル内の項目に対して、一般的な作成、読み込み、更新、削除 (CRUD) のオペレーションを実行できます。以下に、.NET の低レベル API を使用してデータ CRUD オペレーションを実行するための一般的なステップを示します。

1. `AmazonDynamoDBClient` クラスのインスタンス (クライアント) を作成します。
2. 対応するリクエストオブジェクトで、オペレーション固有の必須パラメータを指定します。

   たとえば、項目をアップロードするときは `PutItemRequest` リクエストオブジェクトを使用し、既存の項目を取得するときは `GetItemRequest` リクエストオブジェクトを使用します。

   リクエストオブジェクトを使用して、必須パラメータとオプションパラメータの両方を指定できます。
3. 前述のステップで作成したリクエストオブジェクトを渡して、クライアントから提供された適切なメソッドを実行します。

   `AmazonDynamoDBClient` クライアントは、CRUD オペレーション
   に、`PutItem`、`GetItem`、`UpdateItem` および `DeleteItem` メソッドを提供します。

トピック

- 項目の置換
- 項目の取得
- 項目の更新
- アトミックカウンタ
- 項目の削除
- バッチ書き込み: 複数の項目の置換および削除
- バッチ取得: 複数の項目の取得
- 例: AWS SDK for .NET 低レベル API を使用した CRUD オペレーション
- 例: AWS SDK for .NET 低レベル API を使用したバッチオペレーション
- 例: AWS SDK for .NET 低レベル API を使用したバイナリタイプ属性の処理

項目の置換

PutItem メソッドは、項目をテーブルにアップロードします。項目が存在する場合、その項目全体が置き換えられます。

> **Note**
>
> 項目全体を置き換える代わりに固有の属性のみを更新する場合は、UpdateItem メソッドを使用できます。詳細については、「」を参照してください項目の更新

以下に、低レベル .NET SDK API を使用して項目をアップロードするステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. PutItemRequest クラスのインスタンスを作成して、必要なパラメータを指定します。

   項目を配置するには、テーブル名と項目を指定する必要があります。
3. 前のステップで作成した PutItemRequest オブジェクトを指定して、PutItem メソッドを実行します。

以下の C# サンプルは、前述のステップの例です。この例では、項目を ProductCatalog テーブルにアップロードします。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new PutItemRequest
{
   TableName = tableName,
   Item = new Dictionary<string, AttributeValue>()
      {
          { "Id", new AttributeValue { N = "201" }},
          { "Title", new AttributeValue { S = "Book 201 Title" }},
          { "ISBN", new AttributeValue { S = "11-11-11-11" }},
          { "Price", new AttributeValue { S = "20.00" }},
          {
            "Authors",
            new AttributeValue
```

```
            { SS = new List<string>{"Author1", "Author2"}   }
          }
      }
};
client.PutItem(request);
```

前の例では、Id、Title、ISBN および Authors 属性を持つブック項目をアップロードします。Id は数値型の属性であり、他のすべての属性は文字列型であることに注意してください。作成者は String セットです。

オプションパラメータの指定

次の C# 例に示すように、PutItemRequest オブジェクトを使用してオプションのパラメータを指定することもできます。この例では、次のオプションパラメータが指定されています。

- ExpressionAttributeNames、ExpressionAttributeValues および ConditionExpression は、既存の項目に特定の値を持つ ISBN 属性がある場合にのみ、項目を置き換えるように指定します。
- レスポンスで古い項目をリクエストする ReturnValues パラメータ。

Example

```
var request = new PutItemRequest
 {
   TableName = tableName,
   Item = new Dictionary<string, AttributeValue>()
               {
                   { "Id", new AttributeValue { N = "104" }},
                   { "Title", new AttributeValue { S = "Book 104  Title" }},
                   { "ISBN", new AttributeValue { S = "444-4444444444" }},
                   { "Authors",
                     new AttributeValue { SS = new List<string>{"Author3"}}}
               },
    // Optional parameters.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":isbn",new AttributeValue {S = "444-4444444444"}}
```

```
    },
    ConditionExpression = "#I = :isbn"

};
var response = client.PutItem(request);
```

詳細については、「PutItem」を参照してください。

項目の取得

GetItem メソッドは項目を取得します。

> **Note**
>
> 複数の項目を取り出すために、BatchGetItem メソッドを使用できます。詳細については、「」を参照してくださいバッチ取得: 複数の項目の取得

以下に、低レベル AWS SDK for .NET API を使用して既存の項目を取得するステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. GetItemRequest クラスのインスタンスを作成して、必要なパラメータを指定します。

   項目を吸いtpィするには、テーブル名と項目のプライマリキーを指定する必要があります。
3. 前のステップで作成した GetItemRequest オブジェクトを指定して、GetItem メソッドを実行します。

以下の C# サンプルは、前述のステップの例です。例では、ProductCatalog テーブルから項目を取得します。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
 };
 var response = client.GetItem(request);
```

```
// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item; // Attribute list in the response.
```

オプションパラメータの指定

次の C# 例に示すように、GetItemRequest オブジェクトを使用してオプションのパラメータを指定することもできます。このサンプルでは、次のオプションパラメータが指定されています。

- 取得する属性を指定する ProjectionExpression パラメータ。
- 強力な整合性のある読み込みを実行する ConsistentRead パラメータ。読み込み整合性の詳細については、「DynamoDB の読み取り整合性」を参照してください。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
 {
   TableName = tableName,
   Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
   // Optional parameters.
   ProjectionExpression = "Id, ISBN, Title, Authors",
   ConsistentRead = true
 };

 var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item;
```

詳細については、「GetItem」を参照してください。

項目の更新

UpdateItem メソッドは、既存の項目があればそれを更新します。UpdateItem オペレーションを使用して、既存の属性値を更新するか、新しい属性を追加するか、既存のコレクションから属性を削

除することができます。プライマリキーが指定されている項目がない場合は、新しい項目が追加されます。

UpdateItem オペレーションは、以下のガイドラインに従います。

- 項目が存在しない場合、UpdateItem は入力で指定されたプライマリキーを使用して、新しい項目を追加します。
- 項目が存在する場合、UpdateItem は次のように更新を適用します。
  - 既存の属性値を更新による値に置き換えます。
  - 入力で指定された属性が存在しない場合は、新しい属性を項目に追加します。
  - 入力された属性が null である場合は、属性を削除します。
  - ADD に Action を使用する場合は、既存のセット (文字列または数値セット) に値を追加するか、既存の数値属性値から数学的に加算 (正の数を使用) または減算 (負の数を使用) することができます。

> **Note**
>
> PutItem オペレーションにより、更新を実行できます。詳細については、「」を参照してください項目の置換 たとえば、PutItem をコールして項目をアップロードするときにプライマリキーが存在する場合は、PutItem オペレーションによって項目全体が置き換わります。既存の項目内に属性があり、入力でそれらの属性が指定されていない場合、それらの属性は、PutItem オペレーションによって削除されます。ただし、UpdateItem が更新するのは指定された入力属性だけです。その項目では、その他の既存の属性は変更されません。

以下に、低レベル .NET SDK API を使用して既存の項目を更新するステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. UpdateItemRequest クラスのインスタンスを作成して、必要なパラメータを指定します。

   これは、属性の追加、既存の属性の更新、属性の削除など、すべての更新を記述するリクエストオブジェクトです。既存の属性を削除するには、その属性名に null 値を指定します。
3. 前のステップで作成した UpdateItemRequest オブジェクトを指定して、UpdateItem メソッドを実行します。

以下の C# サンプルコードは、前述のステップの例です。例では、ProductCatalog テーブルのブック項目を更新します。この例では、Authors のコレクションに著者を追加し、既存の ISBN 属性を削除します。また、価格を 1 引き下げます。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#A", "Authors"},
        {"#P", "Price"},
        {"#NA", "NewAttribute"},
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":auth",new AttributeValue { SS = {"Author YY","Author ZZ"}}},
        {":p",new AttributeValue {N = "1"}},
        {":newattr",new AttributeValue {S = "someValue"}},
    },

    // This expression does the following:
    // 1) Adds two new authors to the list
    // 2) Reduces the price
    // 3) Adds a new attribute to the item
    // 4) Removes the ISBN attribute from the item
    UpdateExpression = "ADD #A :auth SET #P = #P - :p, #NA = :newattr REMOVE #I"
};
var response = client.UpdateItem(request);
```

オプションパラメータの指定

次の C# 例に示すように、UpdateItemRequest オブジェクトを使用してオプションのパラメータを指定することもできます。次のオプションパラメータが指定されています。

- ExpressionAttributeValues および ConditionExpression は、既存の料金が 20.00 である場合にのみ料金を更新できるように指定します。

- レスポンスで更新された項目をリクエストする ReturnValues パラメータ。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "202" } } },

    // Update price only if the current price is 20.00.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#P", "Price"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":newprice",new AttributeValue {N = "22"}},
        {":currprice",new AttributeValue {N = "20"}}
    },
    UpdateExpression = "SET #P = :newprice",
    ConditionExpression = "#P = :currprice",
    TableName = tableName,
    ReturnValues = "ALL_NEW" // Return all the attributes of the updated item.
};

var response = client.UpdateItem(request);
```

詳細については、「UpdateItem」を参照してください。

アトミックカウンタ

updateItem を使用してアトミックカウンターを実装できます。アトミックカウンターでは、他の書き込みリクエストを妨げることなく、既存の属性の値をインクリメントまたはデクリメントします。アトミックカウンタを更新するには、UpdateExpression パラメータにタイプに Number の属性を持つ updateItem および ADD を、Action として使用します。

次のサンプルはこのアトミックカウンターを示しており、Quantity 属性を 1 ずつインクリメントさせています。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string, AttributeValue>() { { "Id", new AttributeValue { N =
 "121" } } },
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#Q", "Quantity"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":incr",new AttributeValue {N = "1"}}
    },
    UpdateExpression = "SET #Q = #Q + :incr",
    TableName = tableName
};

var response = client.UpdateItem(request);
```

項目の削除

DeleteItem メソッドによって、テーブルから項目を削除します。

以下では、低レベル .NET SDK API を使用して項目を削除するステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. DeleteItemRequest クラスのインスタンスを作成して、必要なパラメータを指定します。

   項目を削除するには、テーブル名と項目のプライマリキーが必要です。
3. 前のステップで作成した DeleteItemRequest オブジェクトを指定して、DeleteItem メソッドを実行します。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new DeleteItemRequest
```

```
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "201" } } },
};

var response = client.DeleteItem(request);
```

オプションパラメータの指定

次の C# コード例に示すように、DeleteItemRequest オブジェクトを使用して任意のパラメータを指定することもできます。次のオプションパラメータが指定されています。

- ExpressionAttributeValues および ConditionExpression は、ブック項目が公開されていない場合にのみ削除できるよう指定します (InPublication 属性値は false です)。
- レスポンスで削除された項目をリクエストする ReturnValues パラメータ。

Example

```
var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
 "201" } } },

    // Optional parameters.
    ReturnValues = "ALL_OLD",
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":inpub",new AttributeValue {BOOL = false}}
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);
```

詳細については、「DeleteItem」を参照してください。

バッチ書き込み: 複数の項目の置換および削除

バッチ書き込みは、複数の項目の書き込みと削除をバッチで行うことを意味します。BatchWriteItem メソッドによって、単一の コール内にある 1 つまたは複数のテーブルから複数の項目を置換および削除できます。以下に、低レベル .NET SDK API を使用して複数の項目を取得するステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. BatchWriteItemRequest クラスのインスタンスを作成して、すべての入力および削除オペレーションを記述します。
3. 前のステップで作成した BatchWriteItemRequest オブジェクトを指定して、BatchWriteItem メソッドを実行します。
4. 応答を処理します。返された未処理のリクエスト項目が応答内に存在していたかどうかをチェックする必要があります。これは、プロビジョニングされたスループットクォータまたは他の何らかの一時的エラーに達する場合に、発生する可能性があります。また、DynamoDB によって、リクエストのサイズ、およびリクエスト内で指定できるオペレーションの数が制限されます。これらの制限を超えると、DynamoDB によってリクエストが却下されます。詳細については、BatchWriteItem を参照してください。

以下の C# サンプルコードは、前述のステップの例です。この例では、BatchWriteItemRequest を作成して次の書き込みオペレーションを実行します。

- Forum テーブル内で項目を配置します。
- Thread テーブルから項目を入力および削除します。

このコードは BatchWriteItem を実行してバッチ操作を実行します。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchWriteItemRequest
 {
   RequestItems = new Dictionary<string, List<WriteRequest>>
    {
      {
        table1Name, new List<WriteRequest>
```

```
        {
          new WriteRequest
          {
             PutRequest = new PutRequest
             {
                Item = new Dictionary<string,AttributeValue>
                {
                  { "Name", new AttributeValue { S = "Amazon S3 forum" } },
                  { "Threads", new AttributeValue { N = "0" }}
                }
             }
          }
        }
      } ,
      {
        table2Name, new List<WriteRequest>
        {
          new WriteRequest
          {
            PutRequest = new PutRequest
            {
               Item = new Dictionary<string,AttributeValue>
               {
                 { "ForumName", new AttributeValue { S = "Amazon S3 forum" } },
                 { "Subject", new AttributeValue { S = "My sample question" } },
                 { "Message", new AttributeValue { S = "Message Text." } },
                 { "KeywordTags", new AttributeValue { SS = new List<string> { "Amazon
S3", "Bucket" }  } }
               }
            }
          },
          new WriteRequest
          {
             DeleteRequest = new DeleteRequest
             {
                Key = new Dictionary<string,AttributeValue>()
                {
                   { "ForumName", new AttributeValue { S = "Some forum name" } },
                   { "Subject", new AttributeValue { S = "Some subject" } }
                }
             }
          }
        }
      }
```

```
    }
 };
response = client.BatchWriteItem(request);
```

実例については、「例: AWS SDK for .NET 低レベル API を使用したバッチオペレーション」を参照してください。

バッチ取得: 複数の項目の取得

BatchGetItem メソッドによって、1 つまたは複数のテーブルから複数の項目を取得できます。

> **Note**
>
> 単一の項目を取り出すために、GetItem メソッドを使用できます。

以下に、低レベル AWS SDK for .NET API を使用して複数の項目を取得するステップを示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. BatchGetItemRequest クラスのインスタンスを作成して、必要なパラメータを指定します。

   複数の項目を取得するには、テーブル名とプライマリキーバリューのリストが必要です。
3. 前のステップで作成した BatchGetItemRequest オブジェクトを指定して、BatchGetItem メソッドを実行します。
4. 応答を処理します。未処理キーがなかったかどうかを確認する必要があります。これは、プロビジョニングされたスループットクォータまたは他の何らかの一時的エラーに達する場合に発生する可能性があります。

以下の C# サンプルコードは、前述のステップの例です。この例では、Forum および Thread の 2 つのテーブルから取得します。リクエストでは、Forum テーブルに 2 つの項目、Thread テーブルに 3 つの項目を指定します。応答には、両方のテーブルの項目が含まれます。このコードは、応答を処理する方法を示しています。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchGetItemRequest
```

```
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "DynamoDB" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "Amazon S3" } }
          }
        }
      }
    },
    {
      table2Name,
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "DynamoDB" } },
            { "Subject", new AttributeValue { S = "DynamoDB Thread 1" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "DynamoDB" } },
            { "Subject", new AttributeValue { S = "DynamoDB Thread 2" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "ForumName", new AttributeValue { S = "Amazon S3" } },
            { "Subject", new AttributeValue { S = "Amazon S3 Thread 1" } }
          }
        }
      }
    }
  }
```

```
};

var response = client.BatchGetItem(request);

// Check the response.
var result = response.BatchGetItemResult;
var responses = result.Responses; // The attribute list in the response.

var table1Results = responses[table1Name];
Console.WriteLine("Items in table {0}" + table1Name);
foreach (var item1 in table1Results.Items)
{
  PrintItem(item1);
}

var table2Results = responses[table2Name];
Console.WriteLine("Items in table {1}" + table2Name);
foreach (var item2 in table2Results.Items)
{
  PrintItem(item2);
}
// Any unprocessed keys? could happen if you exceed ProvisionedThroughput or some other
 error.
Dictionary<string, KeysAndAttributes> unprocessedKeys = result.UnprocessedKeys;
foreach (KeyValuePair<string, KeysAndAttributes> pair in unprocessedKeys)
{
    Console.WriteLine(pair.Key, pair.Value);
}
```

オプションパラメータの指定

次の C# コード例に示すように、BatchGetItemRequest オブジェクトを使用して任意のパラメータを指定することもできます。例では、Forum テーブルから 2 つの項目を取得します。次のオプションパラメータが指定されています。

- 取得する属性を指定する ProjectionExpression パラメータ。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
```

```
var request = new BatchGetItemRequest
{
  RequestItems = new Dictionary<string, KeysAndAttributes>()
  {
    { table1Name,
      new KeysAndAttributes
      {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "DynamoDB" } }
          },
          new Dictionary<string, AttributeValue>()
          {
            { "Name", new AttributeValue { S = "Amazon S3" } }
          }
        }
      },
      // Optional - name of an attribute to retrieve.
      ProjectionExpression = "Title"
    }
  }
};

var response = client.BatchGetItem(request);
```

詳細については、BatchGetItem を参照してください。

例: AWS SDK for .NET 低レベル API を使用した CRUD オペレーション

以下の C# コード例は、Amazon DynamoDB 項目に対する CRUD オペレーションの例です。この例では、ProductCatalog テーブルへの項目の追加、項目の取得、さまざまな更新の実行、さらに項目の削除を行います。このテーブルを作成していない場合は、プログラムで作成することもできます。詳細については、「AWS SDK for .NET を使用してサンプルテーブルを作成してデータをアップロードする」を参照してください。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
```

```
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelItemCRUDExample
    {
        private static string tableName = "ProductCatalog";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                CreateItem();
                RetrieveItem();

                // Perform various updates.
                UpdateMultipleAttributes();
                UpdateExistingAttributeConditionally();

                // Delete item.
                DeleteItem();
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }

        private static void CreateItem()
        {
            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = new Dictionary<string, AttributeValue>()
            {
```

```
                { "Id", new AttributeValue {
                      N = "1000"
                  }},
                { "Title", new AttributeValue {
                      S = "Book 201 Title"
                  }},
                { "ISBN", new AttributeValue {
                      S = "11-11-11-11"
                  }},
                { "Authors", new AttributeValue {
                      SS = new List<string>{"Author1", "Author2" }
                  }},
                { "Price", new AttributeValue {
                      N = "20.00"
                  }},
                { "Dimensions", new AttributeValue {
                      S = "8.5x11.0x.75"
                  }},
                { "InPublication", new AttributeValue {
                      BOOL = false
                  } }
            }
            };
            client.PutItem(request);
        }

        private static void RetrieveItem()
        {
            var request = new GetItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                ProjectionExpression = "Id, ISBN, Title, Authors",
                ConsistentRead = true
            };
            var response = client.GetItem(request);

            // Check the response.
            var attributeList = response.Item; // attribute list in the response.
```

```
            Console.WriteLine("\nPrinting item after retrieving it .....");
            PrintItem(attributeList);
        }

        private static void UpdateMultipleAttributes()
        {
            var request = new UpdateItemRequest
            {
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                // Perform the following updates:
                // 1) Add two new authors to the list
                // 1) Set a new attribute
                // 2) Remove the ISBN attribute
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#A","Authors"},
                {"#NA","NewAttribute"},
                {"#I","ISBN"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
                {":auth",new AttributeValue {
                     SS = {"Author YY", "Author ZZ"}
                 }},
                {":new",new AttributeValue {
                     S = "New Value"
                 }}
            },

                UpdateExpression = "ADD #A :auth SET #NA = :new REMOVE #I",

                TableName = tableName,
                ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
            };
            var response = client.UpdateItem(request);

            // Check the response.
            var attributeList = response.Attributes; // attribute list in the response.
                                                     // print attributeList.
```

```
            Console.WriteLine("\nPrinting item after multiple attribute
update ............");
            PrintItem(attributeList);
        }

        private static void UpdateExistingAttributeConditionally()
        {
            var request = new UpdateItemRequest
            {
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#P", "Price"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
                {":newprice",new AttributeValue {
                     N = "22.00"
                 }},
                {":currprice",new AttributeValue {
                     N = "20.00"
                 }}
            },
                // This updates price only if current price is 20.00.
                UpdateExpression = "SET #P = :newprice",
                ConditionExpression = "#P = :currprice",

                TableName = tableName,
                ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
            };
            var response = client.UpdateItem(request);

            // Check the response.
            var attributeList = response.Attributes; // attribute list in the response.
            Console.WriteLine("\nPrinting item after updating price value
conditionally ............");
            PrintItem(attributeList);
        }
```

```
        private static void DeleteItem()
        {
            var request = new DeleteItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      N = "1000"
                  } }
            },

                // Return the entire item as it appeared before the update.
                ReturnValues = "ALL_OLD",
                ExpressionAttributeNames = new Dictionary<string, string>()
            {
                {"#IP", "InPublication"}
            },
                ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
            {
                {":inpub",new AttributeValue {
                     BOOL = false
                 }}
            },
                ConditionExpression = "#IP = :inpub"
            };

            var response = client.DeleteItem(request);

            // Check the response.
            var attributeList = response.Attributes; // Attribute list in the response.
                                                     // Print item.
            Console.WriteLine("\nPrinting item that was just deleted ............");
            PrintItem(attributeList);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
```

```
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
 value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }
            Console.WriteLine("*************************************************");
        }
    }
}
```

例: AWS SDK for .NET 低レベル API を使用したバッチオペレーション

トピック

- 例: AWS SDK for .NET 低レベル API を使用したバッチ書き込みオペレーション
- 例: AWS SDK for .NET 低レベル API を使用したバッチ取得オペレーション

このセクションでは、Amazon DynamoDB がサポートするバッチ操作、バッチ書き込み、およびバッチ取得の例を示します。

例: AWS SDK for .NET 低レベル API を使用したバッチ書き込みオペレーション

以下の C# コード例では、`BatchWriteItem` メソッドを使用して、以下の置換および削除のオペレーションを実行します。

- `Forum` テーブル内で 1 つの項目を配置します。
- `Thread` テーブルに対して 1 つの項目を配置および削除します。

バッチの書き込みリクエストを作成すると、1 つまたは複数のテーブルに対して多数の置換リクエストと削除リクエストを指定できます。ただし、DynamoDB `BatchWriteItem` では、1 回のバッチ書き込みオペレーションで可能なバッチ書き込みリクエストのサイズ、置換および削除のオペレーションの数を制限しています。詳細については、BatchWriteItem を参照してください。これらの制限を超えるリクエストは却下されます。プロビジョニングされたスループットがテーブルに不足しているためにこのリクエストを処理できない場合は、応答時に未処理のリクエスト項目が返されます。

以下の例では、未処理のリクエスト項目がないか、応答を確認します。未処理のリクエスト項目がある場合は、BatchWriteItem リクエストをループバックして再送信します。プログラムで、これらのサンプルテーブルを作成し、サンプルデータをアップロードすることもできます。詳細については、「」を参照してくださいAWS SDK for .NET を使用してサンプルテーブルを作成してデータをアップロードする

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchWrite
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                TestBatchWrite();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void TestBatchWrite()
        {
            var request = new BatchWriteItemRequest
            {
                ReturnConsumedCapacity = "TOTAL",
                RequestItems = new Dictionary<string, List<WriteRequest>>
```

```
            {
                {
                    table1Name, new List<WriteRequest>
                    {
                        new WriteRequest
                        {
                            PutRequest = new PutRequest
                            {
                                Item = new Dictionary<string, AttributeValue>
                                {
                                    { "Name", new AttributeValue {
                                          S = "S3 forum"
                                      } },
                                    { "Threads", new AttributeValue {
                                          N = "0"
                                      }}
                                }
                            }
                        }
                    }
                },
                {
                    table2Name, new List<WriteRequest>
                    {
                        new WriteRequest
                        {
                            PutRequest = new PutRequest
                            {
                                Item = new Dictionary<string, AttributeValue>
                                {
                                    { "ForumName", new AttributeValue {
                                          S = "S3 forum"
                                      } },
                                    { "Subject", new AttributeValue {
                                          S = "My sample question"
                                      } },
                                    { "Message", new AttributeValue {
                                          S = "Message Text."
                                      } },
                                    { "KeywordTags", new AttributeValue {
                                          SS = new List<string> { "S3", "Bucket" }
                                      } }
                                }
                            }
```

```
                        },
                        new WriteRequest
                        {
                            // For the operation to delete an item, if you provide a
primary key value
                            // that does not exist in the table, there is no error, it
is just a no-op.
                            DeleteRequest = new DeleteRequest
                            {
                                Key = new Dictionary<string, AttributeValue>()
                                {
                                    { "ForumName",  new AttributeValue {
                                          S = "Some partition key value"
                                      } },
                                    { "Subject", new AttributeValue {
                                          S = "Some sort key value"
                                      } }
                                }
                            }
                        }
                    }
                }
            }
            };

            CallBatchWriteTillCompletion(request);
        }

        private static void CallBatchWriteTillCompletion(BatchWriteItemRequest request)
        {
            BatchWriteItemResponse response;

            int callCount = 0;
            do
            {
                Console.WriteLine("Making request");
                response = client.BatchWriteItem(request);
                callCount++;

                // Check the response.

                var tableConsumedCapacities = response.ConsumedCapacity;
                var unprocessed = response.UnprocessedItems;
```

```
                Console.WriteLine("Per-table consumed capacity");
                foreach (var tableConsumedCapacity in tableConsumedCapacities)
                {
                    Console.WriteLine("{0} - {1}", tableConsumedCapacity.TableName,
 tableConsumedCapacity.CapacityUnits);
                }

                Console.WriteLine("Unprocessed");
                foreach (var unp in unprocessed)
                {
                    Console.WriteLine("{0} - {1}", unp.Key, unp.Value.Count);
                }
                Console.WriteLine();

                // For the next iteration, the request will have unprocessed items.
                request.RequestItems = unprocessed;
            } while (response.UnprocessedItems.Count > 0);

            Console.WriteLine("Total # of batch write API calls made: {0}", callCount);
        }
    }
}
```

例: AWS SDK for .NET 低レベル API を使用したバッチ取得オペレーション

次の C# コード例では、BatchGetItem メソッドを使用して、Amazon DynamoDB の Forum テーブルおよび Thread テーブルから複数の項目を取り出します。BatchGetItemRequest は、各テーブルのテーブル名とプライマリキーのリストを指定します。この例では、取得した項目を印刷して応答を処理します。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchGet
```

```
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                RetrieveMultipleItemsBatchGet();

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void RetrieveMultipleItemsBatchGet()
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
            {
                { table1Name,
                  new KeysAndAttributes
                  {
                      Keys = new List<Dictionary<string, AttributeValue> >()
                      {
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon DynamoDB"
                      } }
                          },
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                            S = "Amazon S3"
                      } }
                          }
                      }
                  }},
                {
```

```
                table2Name,
                new KeysAndAttributes
                {
                    Keys = new List<Dictionary<string, AttributeValue> >()
                    {
                        new Dictionary<string, AttributeValue>()
                        {
                            { "ForumName", new AttributeValue {
                                  S = "Amazon DynamoDB"
                              } },
                            { "Subject", new AttributeValue {
                                  S = "DynamoDB Thread 1"
                              } }
                        },
                        new Dictionary<string, AttributeValue>()
                        {
                            { "ForumName", new AttributeValue {
                                  S = "Amazon DynamoDB"
                              } },
                            { "Subject", new AttributeValue {
                                  S = "DynamoDB Thread 2"
                              } }
                        },
                        new Dictionary<string, AttributeValue>()
                        {
                            { "ForumName", new AttributeValue {
                                  S = "Amazon S3"
                              } },
                            { "Subject", new AttributeValue {
                                  S = "S3 Thread 1"
                              } }
                        }
                    }
                }
            }
        }
        };

        BatchGetItemResponse response;
        do
        {
            Console.WriteLine("Making request");
            response = client.BatchGetItem(request);
```

```
                // Check the response.
                var responses = response.Responses; // Attribute list in the response.

                foreach (var tableResponse in responses)
                {
                    var tableResults = tableResponse.Value;
                    Console.WriteLine("Items retrieved from table {0}",
tableResponse.Key);
                    foreach (var item1 in tableResults)
                    {
                        PrintItem(item1);
                    }
                }

                // Any unprocessed keys? could happen if you exceed
ProvisionedThroughput or some other error.
                Dictionary<string, KeysAndAttributes> unprocessedKeys =
response.UnprocessedKeys;
                foreach (var unprocessedTableKeys in unprocessedKeys)
                {
                    // Print table name.
                    Console.WriteLine(unprocessedTableKeys.Key);
                    // Print unprocessed primary keys.
                    foreach (var key in unprocessedTableKeys.Value.Keys)
                    {
                        PrintItem(key);
                    }
                }

                request.RequestItems = unprocessedKeys;
            } while (response.UnprocessedKeys.Count > 0);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
```

```
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
 value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }
            Console.WriteLine("*************************************************");
        }
    }
}
```

例: AWS SDK for .NET 低レベル API を使用したバイナリタイプ属性の処理

以下の C# コード例は、バイナリタイプ属性の処理の例です。この例では、Reply テーブルに項目を追加します。この項目には、圧縮データを格納するバイナリタイプ属性 (ExtendedMessage) などがあります。また、この例では、項目を取得し、すべての属性値を印刷します。説明のため、この例では GZipStream クラスを使用して、サンプルストリームを圧縮し、圧縮したデータを ExtendedMessage 属性に割り当て、属性値を出力するときに解凍します。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelItemBinaryExample
    {
        private static string tableName = "Reply";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            // Reply table primary key.
            string replyIdPartitionKey = "Amazon DynamoDB#DynamoDB Thread 1";
            string replyDateTimeSortKey = Convert.ToString(DateTime.UtcNow);
```

```
            try
            {
                CreateItem(replyIdPartitionKey, replyDateTimeSortKey);
                RetrieveItem(replyIdPartitionKey, replyDateTimeSortKey);
                // Delete item.
                DeleteItem(replyIdPartitionKey, replyDateTimeSortKey);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateItem(string partitionKey, string sortKey)
        {
            MemoryStream compressedMessage = ToGzipMemoryStream("Some long extended
message to compress.");
            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      S = partitionKey
                  }},
                { "ReplyDateTime", new AttributeValue {
                      S = sortKey
                  }},
                { "Subject", new AttributeValue {
                      S = "Binary type "
                  }},
                { "Message", new AttributeValue {
                      S = "Some message about the binary type"
                  }},
                { "ExtendedMessage", new AttributeValue {
                      B = compressedMessage
                  }}
            }
            };
            client.PutItem(request);
        }
```

```
        private static void RetrieveItem(string partitionKey, string sortKey)
        {
            var request = new GetItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      S = partitionKey
                  } },
                { "ReplyDateTime", new AttributeValue {
                      S = sortKey
                  } }
            },
                ConsistentRead = true
            };
            var response = client.GetItem(request);

            // Check the response.
            var attributeList = response.Item; // attribute list in the response.
            Console.WriteLine("\nPrinting item after retrieving it .............");

            PrintItem(attributeList);
        }

        private static void DeleteItem(string partitionKey, string sortKey)
        {
            var request = new DeleteItemRequest
            {
                TableName = tableName,
                Key = new Dictionary<string, AttributeValue>()
            {
                { "Id", new AttributeValue {
                      S = partitionKey
                  } },
                { "ReplyDateTime", new AttributeValue {
                      S = sortKey
                  } }
            }
            };
            var response = client.DeleteItem(request);
        }

        private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
```

```
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
value.SS.ToArray()) + "]") +
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
value.NS.ToArray()) + "]") +
                    (value.B == null ? "" : "B=[" + FromGzipMemoryStream(value.B) +
"]")
                    );
            }
            Console.WriteLine("************************************************");
        }

        private static MemoryStream ToGzipMemoryStream(string value)
        {
            MemoryStream output = new MemoryStream();
            using (GZipStream zipStream = new GZipStream(output,
CompressionMode.Compress, true))
            using (StreamWriter writer = new StreamWriter(zipStream))
            {
                writer.Write(value);
            }
            return output;
        }

        private static string FromGzipMemoryStream(MemoryStream stream)
        {
            using (GZipStream zipStream = new GZipStream(stream,
CompressionMode.Decompress))
            using (StreamReader reader = new StreamReader(zipStream))
            {
                return reader.ReadToEnd();
            }
        }
    }
```

```
}
```

DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上

Amazon DynamoDB は、異なるアクセスパターンを提供する 2 つのインデックスファミリーをサポートしています。

- セカンダリインデックス – ベーステーブルのプライマリキーに加えて、クエリを実行する代替キーをアプリケーションに付与します。Query と Scan で読み取ります。このページでは、セカンダリインデックスについて説明します。
- ベクトルインデックス – 項目に保存されているベクトル埋め込みに対する類似度検索を有効にします。これらはルックアップの代替キーではなく、Query や Scan ではなく SearchVectors を使用して読み取ります。詳細については、「DynamoDB でのベクトルインデックスの使用」を参照してください。

Amazon DynamoDB によって、プライマリキーの値を指定して、テーブルの項目に高速アクセスすることが可能になります。しかし多くのアプリケーションでは、プライマリキー以外の属性を使って、データに効率的にアクセスできるようにセカンダリ (または代替) キーを 1 つ以上設定することで、メリットが得られることがあります。これに対応するために、1 つのテーブルで 1 つ以上のセカンダリインデックスを作成して、それらのインデックスに対して Query または Scan リクエストを実行することができます。

セカンダリインデックス は、テーブルからの属性のサブセットと、Query オペレーションをサポートする代替キーで構成されるデータ構造です。Query をテーブルで使用する場合と同じように、Query を使用してインデックスからデータを取得できます。テーブルには、複数のセカンダリインデックスを含めることができます。これにより、アプリケーションは複数の異なるクエリパターンにアクセスできます。

> **Note**
>
> また、テーブルを Scan するのと同じように、インデックスも Scan できます。
> セカンダリインデックススキャンオペレーションのクロスアカウントアクセスは、現在、リソースベースのポリシーではサポートされていません。

すべてのセカンダリインデックスは 1 つのテーブルのみに関連付けられ、そこからデータを取得します。これはインデックスのベーステーブルと呼ばれます。インデックスを作成する場合は、インデックスの代替キー (パーティションキーおよびソートキー) を定義します。また、ベーステーブルからインデックスに射影 (コピー) したい属性を定義します。DynamoDB では、これらの属性とベーステーブルからのプライマリキー属性がインデックスにコピーされます。次に、テーブルに対してクエリまたはスキャンを実行する場合と同様に、インデックスに対してクエリまたはスキャンを実行します。

すべてのセカンダリインデックスは、DynamoDB によって自動的にメンテナンスされます。ベーステーブルの項目を追加、変更、または削除すると、そのテーブルのインデックスも更新され、この変更が反映されます。

DynamoDB は、次の 2 種類のセカンダリインデックスをサポートしています。

- グローバルセカンダリインデックス — パーティションキーおよびソートキーを持つインデックス。ベーステーブルのものとは異なる場合があります。このインデックスに関するクエリがすべてのパーティションにまたがり、ベーステーブル内のすべてのデータを対象とする可能性があるため、グローバルセカンダリインデックスは「グローバル」と見なされます。グローバルセカンダリインデックスは、ベーステーブルとは別に独自のパーティション領域に保存され、ベーステーブルとは別にスケーリングします。
- ローカルセカンダリインデックス — パーティションキーはベーステーブルと同じで、ソートキーが異なるインデックス。ローカルセカンダリインデックスは、ローカルセカンダリインデックスのすべてのパーティションの範囲が同じパーティションキーバリューを持つベーステーブルのパーティションに限定されるという意味で「ローカル」です。

グローバルセカンダリインデックスとローカルセカンダリインデックスの比較については、この動画を参照してください。

GSI と LSI のうち正しい選択を行う

トピック

- DynamoDB のグローバルセカンダリインデックスの使用
- ローカルセカンダリインデックス

使用するインデックスの種類を決定するときは、アプリケーションの要件を考慮する必要があります。次の表に、グローバルセカンダリインデックスとローカルセカンダリインデックスの主な違いを示します。

特徴	グローバルセカンダリインデックス	ローカルセカンダリインデックス
キースキーマ	グローバルセカンダリインデックスのプライマリキーはシンプル (パーティションキー) または複合 (パーティションキーとソートキー) のいずれかとすることができます。	ローカルセカンダリインデックスのプライマリキーは複合 (パーティションキーとソートキー) である必要があります。
キーの属性	インデックスパーティションキーとソートキー (存在する場合) は、文字列、数値、またはバイナリ型の任意のベーステーブル属性とすることができます。	インデックスのパーティションキーは、ベーステーブルのパーティションキーと同じ属性です。ソートキーは、文字列、数値、またはバイナリ型の任意のベーステーブル属性とすることができます。
パーティションキー値ごとのサイズ制限	グローバルセカンダリインデックスのサイズ制限はありません。	パーティションキーの値ごとに、すべてのインデックス付き項目の合計サイズが、10 GB 以下である必要があります。
オンラインインデックスオペレーション	グローバルセカンダリインデックスは、テーブルの作成と同時に作成できます。また、新しいグローバルセカンダリインデックスを既存のテーブルに追加したり、既存のグローバルセカンダリインデックスを削除したりすることもできます。詳細については、「」を参照してくださいDynamoDB のグローバルセカンダリインデックスの管理	ローカルセカンダリインデックスは、テーブルの作成と同時に作成されます。ローカルセカンダリインデックスを既存のテーブルに追加したり、既存のローカルセカンダリインデックスを削除したりすることはできません。

特徴	グローバルセカンダリインデックス	ローカルセカンダリインデックス
クエリとパーティション	グローバルセカンダリインデックスでは、すべてのパーティションでテーブル全体に対してクエリを実行できます。	ローカルセカンダリインデックスでは、クエリのパーティションキー値で指定された 1 つのパーティションに対してクエリを実行できます。
読み込み整合性	グローバルセカンダリインデックスのクエリは結果整合性をサポートします。	ローカルセカンダリインデックスのクエリを実行するとき、結果整合性または強い整合性のどちらかを選択できます。
プロビジョニングされたスループットの消費	各グローバルセカンダリインデックスには、読み込み/書き込みアクティビティに対する独自のプロビジョニングされたスループット設定があります。グローバルセカンダリインデックスのクエリまたはスキャンでは、ベーステーブルからではなく、インデックスからキャパシティーユニットを使用します。同じことが、テーブルへの書き込みによるグローバルセカンダリインデックスの更新にも当てはまります。グローバルテーブルに関連付けられたグローバルセカンダリインデックスは、書き込みキャパシティユニットを使用します。	ローカルセカンダリインデックスのクエリまたはスキャンでは、ベーステーブルから読み込みキャパシティーユニットを使用します。テーブルに書き込むと、そのローカルセカンダリインデックスも更新されます。この更新では、ベーステーブルから書き込みキャパシティユニットを使用します。グローバルテーブルに関連付けられたローカルセカンダリインデックスは、レプリケートされた書き込みキャパシティユニットを使用します。

特徴	グローバルセカンダリインデックス	ローカルセカンダリインデックス
射影される属性	グローバルセカンダリインデックスのクエリまたはスキャンでは、インデックスに射影された属性のみをリクエストできます。DynamoDB では、テーブルから属性をフェッチしません。	ローカルセカンダリインデックスをクエリまたはスキャンする場合、インデックスに射影されていない属性をリクエストできます。DynamoDB は、これらの属性をテーブルから自動的に取得します。

セカンダリインデックスを持つテーブルを複数作成する場合は、順次作成する必要があります。たとえば、最初のテーブルを作成し、そのテーブルが `ACTIVE` になるまで待ちます。次のテーブルを作成し、そのテーブルが `ACTIVE` になるまで待ちます。セカンダリインデックスを持つ複数のテーブルを同時に作成しようとすると、DynamoDB は `LimitExceededException` を返します。

各セカンダリインデックスは、関連付けられているベーステーブルと同じテーブルクラスとキャパシティモードを使用します。各セカンダリインデックスには、以下を指定する必要があります。

- 作成するインデックスのタイプ – グローバルセカンダリインデックスまたはローカルセカンダリインデックスのいずれか。
- インデックスの名前。インデックスの名前付けルールは、「Amazon DynamoDB のクォータ」に示すようにテーブルの場合と同じです。名前は関連付けられているベーステーブルに対して一意である必要がありますが、別のベーステーブルに関連付けられているインデックスでも同じ名前を使用できます。
- インデックスのキースキーマ。インデックスキースキーマの各属性は、型が `String`、`Number`、または `Binary` の最上位属性である必要があります。ドキュメントとセットを含むその他のデータ型は使用できません。キースキーマのその他の要件は、インデックスの種類によって異なります。
  - グローバルセカンダリインデックスの場合、パーティションキーはベーステーブルの任意のスカラー属性にすることができます。ソートキーはオプションです。このキーもベーステーブルの任意のスカラー属性にすることができます。
  - ローカルセカンダリインデックスの場合、パーティションキーは、ベーステーブルのパーティションキーと同じである必要があります。ソートキーは、非キーベーステーブル属性である必要があります。

- ベーステーブルからインデックスに射影する追加の属性 (ある場合)。この属性は、すべてのインデックスに自動的に射影されるテーブルのキー属性とは別の属性です。スカラー、ドキュメント、およびセットを含む任意のデータ型の属性を射影できます。
- インデックスのプロビジョニングされたスループット設定 (必要な場合) :
  - グローバルセカンダリインデックスの場合、読み込み/書き込みキャパシティーユニット設定を指定する必要があります。このプロビジョニングされたスループット設定は、ベーステーブルの設定から独立しています。
  - ローカルセカンダリインデックスの場合、読み込み/書き込みキャパシティーユニット設定を指定する必要はありません。ローカルセカンダリインデックスに対する読み込み/書き込みオペレーションは、そのベーステーブルのプロビジョニングされたスループット設定から使用します。

クエリに最大限の柔軟性を持たせるために、テーブルごとに最大 20 個のグローバルセカンダリインデックス (デフォルトのクォータ) および最大 5 個のローカルセカンダリインデックスを作成できます。

テーブルのセカンダリインデックスの詳細なリストを取得するには、`DescribeTable` オペレーションを使用します。`DescribeTable` は、テーブルのすべてのセカンダリインデックスの名前、ストレージサイズ、および項目数を返します。これらの値はリアルタイムでは更新されませんが、約 6 時間ごとに更新されます。

セカンダリインデックスのデータには、`Query` または `Scan` オペレーションを使用してアクセスできます。使用するベーステーブル名とインデックス名、結果で返される属性、および適用する条件式またはフィルタを指定する必要があります。DynamoDB は、結果を昇順で返すことも降順で返すこともできます。

テーブルを削除すると、そのテーブルに関連付けられているすべてのインデックスも削除されます。

ベストプラクティスについては、DynamoDB でセカンダリインデックスを使用するためのベストプラクティス。 を参照してください。

DynamoDB のグローバルセカンダリインデックスの使用

アプリケーションによっては、さまざまな属性をクエリ基準に使用して、いろいろな種類のクエリを実行する必要があります。このような要件に対応するために、1 つまたは複数のグローバルセカンダリインデックスを作成して、Amazon DynamoDB でこれらのインデックスに対して `Query` リクエストを発行できます。

トピック

- シナリオ: グローバルセカンダリインデックスの使用
- 属性の射影
- マルチ属性キースキーマ
- グローバルセカンダリインデックスからのデータの読み込み
- テーブルとグローバルセカンダリインデックス間のデータ同期
- グローバルセカンダリインデックスがあるテーブルクラス
- グローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項
- グローバルセカンダリインデックスのストレージに関する考慮事項
- パターンの設計
- DynamoDB のグローバルセカンダリインデックスの管理
- DynamoDB でのインデックスキー違反の検出と修正
- グローバルセカンダリインデックスの操作: Java
- グローバルセカンダリインデックスの操作: .NET
- AWS CLI を使用した DynamoDBでのグローバルセカンダリインデックスの操作

シナリオ: グローバルセカンダリインデックスの使用

たとえば、GameScores という名前のテーブルがあり、モバイルゲームアプリケーションのユーザーとスコアを記録しているとします。GameScores の各項目は、パーティションキー (UserId) およびソートキー (GameTitle) で特定されます。次の図は、テーブル内の項目の構成を示しています。一部表示されていない属性もあります。

GameScores

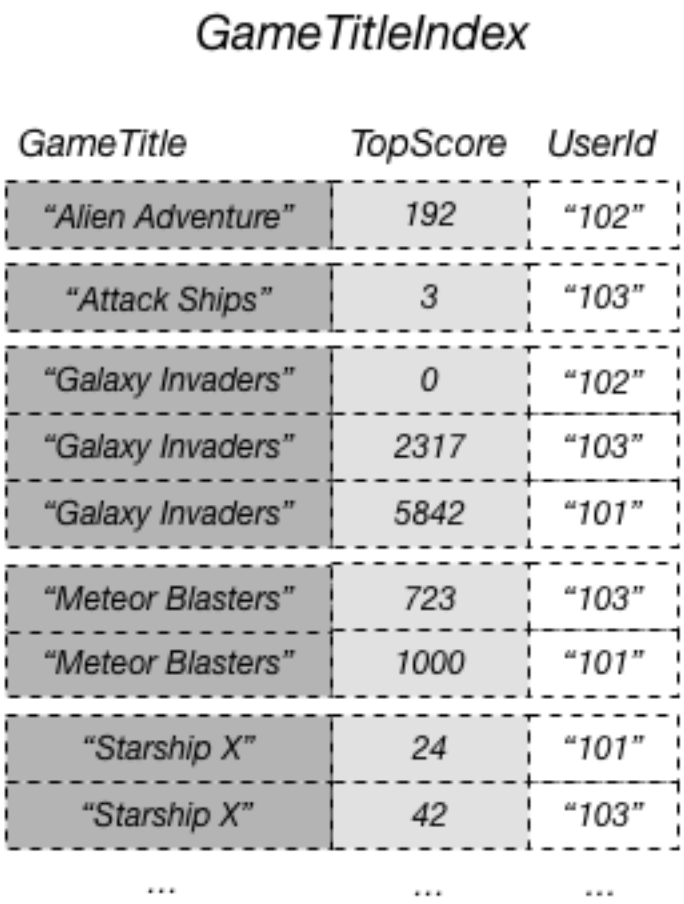
UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
“101”	“Galaxy Invaders”	5842	“2015-09-15:17:24:31”	21	72	…
“101”	“Meteor Blasters”	1000	“2015-10-22:23:18:01”	12	3	…
“101”	“Starship X”	24	“2015-08-31:13:14:21”	4	9	…
“102”	“Alien Adventure”	192	“2015-07-12:11:07:56”	32	192	…
“102”	“Galaxy Invaders”	0	“2015-09-18:07:33:42”	0	5	…
“103”	“Attack Ships”	3	“2015-10-19:01:13:24”	1	8	…
“103”	“Galaxy Invaders”	2317	“2015-09-11:06:53:00”	40	3	…
“103”	“Meteor Blasters”	723	“2015-10-19:01:13:24”	22	12	…
“103”	“Starship X”	42	“2015-07-11:06:53:00”	4	19	…
…	…	…	…	…	…	

ここで、各ゲームの最高スコアを表示する順位表アプリケーションを作成すると仮定します。キー属性 (`UserId` と `GameTitle`) を指定したクエリは非常に効率的です。ただし、アプリケーションが `GameScores` だけに基づいて `GameTitle` からデータを取り出す必要がある場合、`Scan` オペレーションを使用する必要があります。テーブルに追加される項目が増えるにつれ、すべてのデータのスキャンは低速で非効率的になります。このため、次のような質問に答えることが難しくなります。

- ゲーム Meteor Blasters で記録された最高スコアはいくつですか?
- Galaxy Invaders で最高スコアを獲得したユーザーは誰ですか?
- 勝敗の最も高い比率は何ですか?

グローバルセカンダリインデックスを作成して、非キー属性に対するクエリの速度を上げることができます。グローバルセカンダリインデックスには、ベーステーブルからの属性の一部が格納されますが、それらはテーブルのプライマリキーとは異なるプライマリキーによって構成されます。インデックスキーは、テーブルからのキー属性を持つ必要がありません。また、テーブルと同じキースキーマを使用する必要もありません。

例えば、パーティションキーとして GameTitle、ソートキーとして TopScore を使用して、GameTitleIndex という名前のグローバルセカンダリインデックスを作成できます。ベーステーブルのプライマリキー属性は必ずインデックスに射影されるので、UserId 属性も存在します。次の図は、GameTitleIndex インデックスを示しています。



これで、GameTitleIndex に対してクエリを実行し、Meteor Blasters のスコアを簡単に入手できるようになります。結果は、ソートキーの値 TopScore 別に並べられます。ScanIndexForward パラメータを false に設定した場合、結果は降順で返されます。つまり、最高スコアが最初に返されます。

すべてのグローバルセカンダリインデックスには、パーティションキーが必要で、オプションのソートキーを指定できます。インデックスキースキーマは、テーブルスキーマとは異なるものにすることができます。シンプルなプライマリキー (パーティションキー) を持つテーブルを作成し、複合プライマリキー (パーティションキーおよびソートキー) を使用してグローバルセカンダリインデックスを作成できます。またはその逆もあります。インデックスキー属性は、ベーステーブルからの任意の最上位 String、Number、または Binary 属性で構成できます。その他のスカラー型、ドキュメント型、およびセット型は許可されません。

必要な場合、他のベーステーブル属性をインデックスに射影できます。インデックスのクエリを行うと、DynamoDB ではこれらの射影された属性を効率的に取り出すことができます。ただし、グローバルセカンダリインデックス のクエリでは、ベーステーブルから属性をフェッチできません。たとえば、上記の図に示すように、GameTitleIndex にクエリを実行した場合、クエリは TopScore 以外の非キー属性にアクセスすることはできません (キー属性 GameTitle と UserId は自動的に射影されます)。

DynamoDB テーブルでは、各キー値は一意である必要があります。ただし、グローバルセカンダリインデックスのキー値は一意である必要はありません。たとえば、Comet Quest という名前のゲームが特に難しく、多くの新しいユーザーが試しても、ゼロを上回るスコアを獲得することができないとします。以下にこれを表すいくつかのデータを示します。

UserId	GameTitle	TopScore
123	Comet Quest	0
201	Comet Quest	0
301	Comet Quest	0

このデータを GameScores テーブルに追加すると、DynamoDB はそのデータを GameTitleIndex に伝達します。次に、GameTitle に Comet Quest を、TopScore に 0 を指定してインデックスに対してクエリを実行すると、次のデータが返されます。

GameTitle	*TopScore*	*UserId*
"Comet Quest"	*0*	*"123"*
"Comet Quest"	*0*	*"201"*
"Comet Quest"	*0*	*"301"*

指定したキー値を持つ項目だけがレスポンスに表示されます。その一連のデータ内では、項目は特定の順に並んでいません。

グローバルセカンダリインデックスは、キー属性が実際に存在するデータ項目のみを追跡します。たとえば、別の新しい項目を GameScores テーブルに追加し、必須のプライマリキー属性だけを指定したとします。

UserId	GameTitle
400	Comet Quest

TopScore 属性を指定していないので、DynamoDB は、この項目を GameTitleIndex に伝達しません。このため、すべての Comet Quest 項目を対象に、GameScores に対してクエリを実行した場合、次の 4 つの項目が返されます。

UserId	GameTitle	TopScore
“123”	“Comet Quest”	0
“201”	“Comet Quest”	0
“301”	“Comet Quest”	0
“400”	“Comet Quest”	

GameTitleIndex に対して同様のクエリを実行すると、4 つではなく 3 つの項目が返されます。これは、TopScore が存在しない項目はインデックスに反映されないためです。

GameTitle	*TopScore*	*UserId*
“Comet Quest”	*0*	*“123”*
“Comet Quest”	*0*	*“201”*
“Comet Quest”	*0*	*“301”*

属性の射影

射影とは、テーブルからセカンダリインデックスにコピーされる属性のセットです。テーブルのパーティションキーとソートキーは常にインデックスに射影されます。アプリケーションのクエリ要件をサポートするために、他の属性を射影できます。インデックスをクエリすると、Amazon DynamoDB は、その属性が自身のテーブル内にあるかのように、プロジェクション内の任意の属性にアクセスできます。

セカンダリインデックスを作成するときは、インデックスに射影される属性を指定する必要があります。DynamoDB には、そのために次の 3 つのオプションが用意されています。

- KEYS_ONLY – インデックス内の各項目は、テーブルパーティションキーとソートキーの値、およびインデックスキーの値のみで構成されます。KEYS_ONLY オプションを指定すると、セカンダリインデックスが最小になります。
- INCLUDE – KEYS_ONLY の属性に加えて、セカンダリインデックスにその他の非キー属性が含まれるように指定できます。
- ALL – セカンダリインデックスには、ソーステーブルのすべての属性が含まれます。すべてのテーブルデータがインデックスに複製されるため、ALL の射影にすると、セカンダリインデックスが最大になります。

前の図では、GameTitleIndex には 1 つの射影された属性 UserId のみがあります。そのため、アプリケーションはクエリで UserId および GameTitle を使用して各ゲームのトップスコアラーの TopScore を効率的に判断できますが、トップスコアラーの勝敗の最も高い比率は効率的に判断できません。そのためには、アプリケーションは基本テーブルで追加のクエリを実行して、各トップスコアラーの勝敗を取得する必要があります。このデータに対するクエリのサポートを効率化する方法として、次の図に示すように、これらの属性をベーステーブルからグローバルセカンダリインデックスに射影する方法があります。

GameTitleIndex

GameTitle	TopScore	UserId	Wins	Losses
"Alien Adventure"	192	"102"	32	192
"Attack Ships"	3	"103"	1	8
"Galaxy Invaders"	0	"102"	0	5
"Galaxy Invaders"	2317	"103"	40	3
"Galaxy Invaders"	5842	"101"	21	72
"Meteor Blasters"	723	"103"	22	12
"Meteor Blasters"	1000	"101"	12	3
"Starship X"	24	"101"	4	9
"Starship X"	42	"103"	4	19
...	...	...	...	...

非キー属性 Wins と Losses がインデックスに射影されるので、アプリケーションは、任意のゲーム、またはゲームとユーザー ID の任意の組み合わせに対して、勝敗の比率を特定できます。

グローバルセカンダリインデックスに射影する属性を選択する場合には、プロビジョニングされるスループットコストとストレージコストのトレードオフを考慮する必要があります。

- ごく一部の属性だけに最小のレイテンシーでアクセスする必要がある場合は、それらの属性だけをグローバルセカンダリインデックスに射影することを検討してください。インデックスが小さいほど少ないコストで格納でき、書き込みコストも低くなります。
- アプリケーションが非キー属性に頻繁にアクセスする場合には、それらの属性をグローバルセカンダリインデックスに射影することを検討してください。グローバルセカンダリインデックスのストレージコストの増加は、頻繁にテーブルスキャンを実行するコストの減少で相殺されます。
- ほとんどの非キー属性に頻繁にアクセスする場合は、それらの属性を (場合によっては、ベーステーブル全体を) グローバルセカンダリインデックスに射影することができます。これにより、最大限の柔軟性が得られます。ただし、ストレージコストは増加するか、倍になります。
- アプリケーションでテーブルのクエリを頻繁に行う必要がなく、テーブル内のデータに対する書き込みや更新が多数になる場合は、KEYS_ONLY を射影することを検討してください。グローバルセカンダリインデックスは、最小サイズになりますが、それでもクエリのアクティビティに必要な場合は利用可能です。

マルチ属性キースキーマ

グローバルセカンダリインデックスはマルチ属性キーをサポートしているため、複数の属性からパーティションキーとソートキーを作成できます。マルチ属性キーを使用すると、最大 4 つの属性からパーティションキーを、最大 4 つの属性からソートキーを、キースキーマごとに合計で最大 8 つの属性に作成できます。

マルチ属性キーは、属性を合成キーに手動で連結する必要がなくなるため、データモデルを簡素化します。TOURNAMENT#WINTER2024#REGION#NA-EAST のような複合文字列を作成する代わりに、ドメインモデルから自然属性を直接使用できます。DynamoDB は複合キーロジックを自動的に処理し、データ分散のために複数のパーティションキー属性をハッシュし、複数のソートキー属性にわたって階層型のソート順序を維持します。

例えば、試合をトーナメントとリージョン別に整理するゲームトーナメントシステムを考えてみましょう。マルチ属性キーを使用すると、パーティションキーを tournamentId と region の 2 つの異なる属性として定義できます。同様に、round、bracket、matchId などの複数の属性を使用してソートキーを定義し、自然な階層を作成できます。このアプローチでは、文字列の操作や解析を行わずに、データ型とコードをクリーンに保ちます。

マルチ属性キーを使用してグローバルセカンダリインデックスをクエリする場合は、等号条件を使用してすべてのパーティションキー属性を指定する必要があります。ソートキー属性の場合、キースキーマで定義されている順序で、左から右にクエリを実行できます。つまり、最初のソートキー属性のみ、最初の 2 つの属性を一緒に、またはすべての属性を一緒にクエリできますが、途中で属性をスキップすることはできません。>、<、BETWEEN、begins_with( ) などの不等条件は、クエリの最後の条件である必要があります。

マルチ属性キーは、既存のテーブルにグローバルセカンダリインデックスを作成する場合、特に効果的です。データ全体で合成キーをバックフィルすることなく、テーブルに既に存在する属性を使用できます。これにより、さまざまな属性の組み合わせを使用してデータを再編成するインデックスを作成することで、アプリケーションに新しいクエリパターンを簡単に追加できます。

マルチ属性キーの各属性は、独自のデータ型 String (S)、Number (N)、または Binary (B) を持つことができます。データ型を選択するときは、ゼロパディングを必要とせずに Number 属性を数値順にソートし、String 属性を辞書順にソートすることを検討してください。例えば、スコア属性に Number 型を使用する場合、値 5、50、500、1000 は自然な数値順にソートされます。同じ値を String 型とする場合は、先頭にゼロを埋めない限り、「1000」、「5」、「50」、「500」とソートされます。

マルチ属性キーを設計する場合は、属性を最も一般的なものから最も具体的な順に並べてください。パーティションキーの場合、常に一緒にクエリされ、適切なデータ分散を提供する属性を組み合わせます。ソートキーの場合、クエリの柔軟性を最大化するために、頻繁にクエリされる属性を階層の最初に配置します。この順序により、アクセスパターンに一致する任意の詳細度のレベルでクエリを実行できます。

実装例については、「マルチ属性キー」を参照してください。

グローバルセカンダリインデックスからのデータの読み込み

グローバルセカンダリインデックスから項目を取得するには、Query および Scan オペレーションを使用します。GetItem および BatchGetItem オペレーションは、グローバルセカンダリインデックスでは使用できません。

グローバルセカンダリインデックスのクエリ

Query オペレーションを使用して、グローバルセカンダリインデックスの 1 つ以上の項目にアクセスすることができます。クエリでは、使用するベーステーブル名とインデックス名、クエリ結果で返される属性、および適用するクエリ条件を指定する必要があります。DynamoDB は、結果を昇順で返すことも降順で返すこともできます。

順位表アプリケーションのゲームデータをリクエストする Query から返される次のデータについて考えます。

```
{
    "TableName": "GameScores",
    "IndexName": "GameTitleIndex",
    "KeyConditionExpression": "GameTitle = :v_title",
    "ExpressionAttributeValues": {
        ":v_title": {"S": "Meteor Blasters"}
    },
    "ProjectionExpression": "UserId, TopScore",
    "ScanIndexForward": false
}
```

このクエリでは次のようになっています。

- DynamoDB は、GameTitle パーティションキーを使用して GameTitleIndex にアクセスし、Meteor Blasters のインデックス項目を特定します。このキーを持つすべてのインデックス項目が、すばやく取り出せるように隣り合わせに格納されます。
- このゲーム内で、DynamoDB はインデックスを使用して、このゲームのすべてのユーザーの ID と最高スコアにアクセスします。
- ScanIndexForward パラメータが false に設定されているので、結果は降順で返されます。

グローバルセカンダリインデックスのスキャン

Scan オペレーションを使用して、グローバルセカンダリインデックスからすべてのデータを取得することができます。リクエストにはベーステーブル名とインデックス名を指定する必要があります。Scan では、DynamoDB はインデックスのすべてのデータを読み込み、それをアプリケーションに返します。また、データの一部のみを返し、残りのデータを破棄するようにリクエストすることもできます。これを行うには、FilterExpression オペレーションの Scan パラメータを使用します。詳細については、「」を参照してくださいScan のフィルタ式

テーブルとグローバルセカンダリインデックス間のデータ同期

DynamoDB では、各グローバルセカンダリインデックスをそのベーステーブルと自動的に同期させています。アプリケーションがテーブルに項目を書き込むか、削除すると、そのテーブルのすべてのグローバルセカンダリインデックスは、結果整合性モデルを使用して、非同期で更新されます。アプリケーションがインデックスに直接書き込むことはありません。ただし、DynamoDB でこれらのインデックスがどのように維持されるかを理解することは重要です。

グローバルセカンダリーインデックスは、基本テーブルから読み込み/書き込みキャパシティーモードを継承します。詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。

グローバルセカンダリインデックスを作成するときは、1 つ以上のインデックスキー属性およびそれらのデータ型を指定します。つまり、ベーステーブルに項目を書き込むとき、それらの属性のデータ型が、インデックスキースキーマのデータ型に一致する必要があります。GameTitleIndex の場合、インデックス内の GameTitle パーティションキーは、String データ型として定義されています。インデックス内の TopScore ソートキーは、Number の型です。GameScores テーブルに項目を追加し、GameTitle または TopScore に対して別のデータ型を指定する場合、データ型の不一致により DynamoDB によって ValidationException が返されます。

テーブルに項目を入力するか、削除すると、そのテーブルのグローバルセカンダリインデックスは、結果整合性のある方法で更新されます。テーブルデータへの変更は、通常は、瞬時にグローバルセカンダリインデックスに伝達されます。ただし、万が一障害が発生した場合は、長い伝達遅延が発生することがあります。このため、アプリケーションでは、グローバルセカンダリインデックスに関するクエリに対して、最新でない結果が返される状況を予期し、それに対応する必要があります。

テーブルに項目を書き込む場合には、グローバルセカンダリインデックスのソートキーに対して属性を指定する必要はありません。GameTitleIndex を例にとると、GameScores テーブルに新しい項目を書き込むために、TopScore 属性に値を指定する必要はありません。このケースでは、DynamoDB はこの特定の項目のインデックスにデータを書き込むことはありません。

多数のグローバルセカンダリインデックスがあるテーブルは、インデックス数が少ないテーブルに比べて書き込みアクティビティに多くのコストを要します。詳細については、「グローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項」を参照してください。

グローバルセカンダリインデックスがあるテーブルクラス

グローバルセカンダリインデックスは、常にベーステーブルと同じテーブルクラスを使用します。テーブルに新しいグローバルセカンダリインデックスが追加されるたびに、新しいインデックスはベーステーブルと同じテーブルクラスを使用します。テーブルのテーブルクラスが更新されると、関連するすべてのグローバルセカンダリインデックスも更新されます。

グローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項

プロビジョニングモードのテーブルでグローバルセカンダリインデックスを作成するときには、そのインデックスに対して予想されるワークロードに応じた読み込み/書き込みキャパシティーユニッ

トを指定する必要があります。グローバルセカンダリインデックスのプロビジョニングされたスループット設定は、そのベーステーブルの設定とは独立しています。グローバルセカンダリインデックスに対する Query オペレーションでは、ベーステーブルではなく、インデックスから読み込みキャパシティーユニットを使用します。テーブルで項目を入力、更新または削除すると、そのテーブルのグローバルセカンダリインデックスも更新されます。これらのインデックス更新は、ベーステーブルからではなく、インデックスから書き込みキャパシティーユニットを消費します。

例えば、グローバルセカンダリインデックスに対して Query を実行し、そのプロビジョニングされた読み込みキャパシティーを超えた場合、リクエストは調整されます。多量の書き込みアクティビティをテーブルに対して実行するときに、そのテーブルのグローバルセカンダリインデックスに十分な書き込み容量がない場合、そのテーブルに対する書き込みアクティビティは調整されます。

> **Important**
>
> スロットリングを避けるため、グローバルセカンダリインデックスのプロビジョニングされた書き込みキャパシティーは、ベーステーブルの書き込みキャパシティーと同じかそれより大きい必要があります。これは、新しい更新ではベーステーブルとグローバルセカンダリインデックスの両方に書き込むためです。

グローバルセカンダリインデックスのプロビジョニングされたスループット設定を表示するには、DescribeTable オペレーションを使用します。テーブルのすべてのグローバルセカンダリインデックスに関する詳細情報が返されます。

読み込みキャパシティユニット

グローバルセカンダリインデックスでは、結果整合性のある読み込みをサポートしており、各読み込みで、読み込みキャパシティーユニットの半分を消費します。つまり、1 回のグローバルセカンダリインデックスのクエリでは、1 読み込みキャパシティーユニットあたり最大 2 × 4 KB = 8 KB を取り出すことができます。

グローバルセカンダリインデックスのクエリの場合、DynamoDB はプロビジョニングされた読み込みアクティビティをテーブルに対するクエリと同じ方法で計算します。唯一の違いは、ベーステーブル内の項目のサイズではなくインデックスエントリのサイズに基づいて計算が行われることです。読み込みキャパシティーユニットの数は、返されたすべての項目について射影されたすべての属性のサイズの合計です。結果は、次の 4 KB 境界まで切り上げられます。DynamoDB がプロビジョニングされたスループットの利用率を計算する方法の詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。

Query オペレーションによって返される結果の最大サイズは 1 MB です。これには、返されるすべての項目にわたる、すべての属性の名前と値のサイズが含まれます。

例えば、各項目に 2,000 バイトのデータが格納されているグローバルセカンダリインデックスがあるとします。このインデックスに対して Query を実行したら、クエリの KeyConditionExpression が 8 項目に一致したとします。一致する項目の合計サイズは、2,000 バイト × 8 項目 = 16,000 バイトです。これは、最も近い 4 KB 境界に切り上げられます。グローバルセカンダリインデックスのクエリは結果整合性なので、合計コストは、0.5 × (16 KB / 4 KB)、つまり、2 読み込みキャパシティーユニットです。

書き込みキャパシティユニット

テーブルの項目が追加、更新、または削除されたときに、グローバルセカンダリインデックスがこの影響を受ける場合、グローバルセカンダリインデックスは、そのオペレーションに対してプロビジョニングされた書き込みキャパシティーユニットを使用します。書き込み用にプロビジョニングされたスループットの合計コストは、ベーステーブルに対する書き込みと、グローバルセカンダリインデックスの更新で消費された書き込みキャパシティーユニットの合計になります。テーブルへの書き込みには、グローバルセカンダリインデックスの更新は必要ないので、インデックスから使用される書き込み容量はありません。

テーブルへの書き込みに成功するように、テーブルとそのすべてのグローバルセカンダリインデックスに対するプロビジョニングされたスループット設定は、書き込みに対応できるだけの十分な書き込みキャパシティーを備えている必要があります。十分でない場合、書き込みが調整されます。

> **Important**
>
> グローバルセカンダリインデックス (GSI) を作成する際、基本テーブルへの書き込みによって生じる GSI アクティビティが GSI のプロビジョニングされた書き込み容量を超える場合、基本テーブルへの書き込み操作がスロットリングされる可能性があります。このスロットリングは、インデックス作成プロセスから本番ワークロードの中断の可能性に至るまで、すべての書き込み操作に影響します。詳細については、「Amazon DynamoDB でのスロットリングのトラブルシューティング」を参照してください。

グローバルセカンダリインデックスに項目を書き込むコストは、いくつかの要因に左右されます。

- インデックス付き属性が定義されたテーブルに新しい項目を書き込む場合、または既存の項目を更新して未定義のインデックス付き属性を定義する場合には、インデックスへの項目の挿入に 1 回の書き込みオペレーションが必要です。

- テーブルに対する更新によってインデックス付きキー属性の値が (A から B に) 変更された場合には、インデックスから既存の項目を削除し、インデックスに新しい項目を挿入するために、2 回の書き込みが必要です。
- インデックス内に既存の項目があって、テーブルに対する書き込みによってインデックス付き属性が削除された場合は、インデックスから古い項目の射影を削除するために、1 回の書き込みが必要です。
- 項目の更新の前後にインデックス内に項目が存在しない場合は、インデックスで追加の書き込みコストは発生しません。
- テーブルに対する更新によってインデックスキースキーマの射影された属性の値のみが変更され、インデックス付きキー属性の値は変更されない場合は、インデックスに射影された属性の値を更新するために、1 回の書き込みが必要です。
- テーブルの更新によって、インデックスキー属性でない属性、またはインデックスに射影されていない属性のみが変更される場合、インデックスエントリは変更されないため、そのインデックスの書き込み容量は消費されません。ベーステーブルは書き込みに対して引き続き課金されます。

これらすべての要因は、インデックス内の各項目のサイズが 1 KB 以下であるという前提で書き込みキャパシティーユニット数を算出します。インデックスエントリがそれよりも大きい場合は、書き込みキャパシティーユニットを追加する必要があります。クエリが返す必要がある属性を特定し、それらの属性だけをインデックスに射影することで、書き込みコストは最小になります。

グローバルセカンダリインデックスのストレージに関する考慮事項

アプリケーションがテーブルに項目を書き込むと、DynamoDB では、正しい属性のサブセットが、それらの属性が現れる必要があるグローバルセカンダリインデックスに自動的にコピーされます。AWS アカウントでは、テーブル内の項目のストレージと、そのベーステーブルのグローバルセカンダリインデックスにある属性のストレージに対して課金されます。

インデックス項目が使用するスペースの量は、次の量の合計になります。

- ベーステーブルのプライマリキー (パーティションキーとソートキー) のサイズのバイト数
- インデックスキー属性のサイズのバイト数
- 射影された属性 (存在する場合) のサイズのバイト数
- インデックス項目あたり 100 バイトのオーバーヘッド

グローバルセカンダリインデックスのストレージ要件の見積もりは、インデックス内の 1 項目の平均サイズの見積もり値に、ベーステーブル内のグローバルセカンダリインデックスキー属性を持つ項目の数を掛けて算出します。

特定の属性が定義されていない項目がテーブルに含まれていて、その属性がインデックスパーティションキーまたはソートキーとして定義されている場合、DynamoDB はその項目のデータをインデックスに書き込みません。

パターンの設計

設計パターンは、グローバルセカンダリインデックスを使用する際の一般的な課題に対する実証済みのソリューションを提供します。これらのパターンは、特定のユースケースに合わせてインデックスを構築する方法を示すことで、効率的でスケーラブルなアプリケーションを構築するのに役立ちます。

各パターンには、独自のアプリケーションにパターンを適用するのに役立つコード例、ベストプラクティス、実際のユースケースを含む完全な実装ガイドが含まれています。

トピック

- マルチ属性キーパターン

マルチ属性キーパターン

概要

マルチ属性キーを使用すると、それぞれ最大 4 つの属性で構成されるグローバルセカンダリインデックス (GSI) パーティションとソートキーを作成できます。これにより、クライアント側のコードが減り、最初にデータをモデル化して後で新しいアクセスパターンを追加することが容易になります。

一般的なシナリオを考えてみましょう。複数の階層属性で項目をクエリする GSI を作成するには、従来、値を連結して合成キーを作成する必要があります。例えば、ゲームアプリでは、トーナメント、リージョン、ラウンドでトーナメントマッチをクエリするために、TOURNAMENT#WINTER2024#REGION#NA-EAST などの合成 GSI パーティションキーと ROUND#SEMIFINALS#BRACKET#UPPER などの合成ソートキーを作成できます。このアプローチは機能しますが、既存のテーブルに GSI を追加する場合は、データを書き込むときに文字列を連結し、読み取り時に解析し、既存のすべての項目にわたって合成キーをバックフィルする必要があります。これにより、コードが整理整頓され、個々のキーコンポーネントで型の安全性を維持することが困難になります。

マルチ属性キーは、GSI のこの問題を解決します。GSI パーティションキーは、tournamentId や region などの複数の既存の属性を使用して定義します。DynamoDB は複合キーロジックを自動的に処理し、データ分散のためにそれらをハッシュします。ドメインモデルから自然属性を使用して項目を記述すると、GSI は自動的にインデックスを作成します。連結なし、解析なし、バックフィルなし。コードはきれいに保たれ、データは入力され、クエリはシンプルになります。このアプローチは、自然属性のグループ化を含む階層データがある場合に特に役立ちます (トーナメント → リージョン → ラウンド、または組織 → 部門 → チームなど)。

アプリケーションの例

このガイドでは、e スポーツプラットフォームのトーナメントマッチ追跡システムを構築する方法について説明します。プラットフォームは、ブラケット管理のためのトーナメントとリージョン、マッチング履歴のための選手、スケジューリングのための日付など、複数のディメンションにわたって試合を効率的にクエリする必要があります。

データモデル

このチュートリアルでは、トーナメントマッチ追跡システムは 3 つの主要なアクセスパターンをサポートし、それぞれに異なるキー構造が必要です。

アクセスパターン 1: 一意の ID で特定の試合を検索する

- 解決策: パーティションキーとして `matchId` をベーステーブルと使用する

アクセスパターン 2: 特定のトーナメントとリージョンのすべての試合をクエリし、オプションでラウンド、ブラケット、または試合でフィルタリングする

- 解決策: マルチ属性パーティションキー (`tournamentId` + `region`) とマルチ属性ソートキー (`round` + `bracket` + `matchId`) を持つグローバルセカンダリインデックス
- クエリの例: 「NA-EAST リージョン内の All WINTER2024 の試合」、または「WINTER2024/NA-EAST の UPPER ブラケット内の All SEMIFINALS の試合」

アクセスパターン 3: 選手の試合履歴をクエリし、オプションで日付範囲またはトーナメントラウンドでフィルタリングする

- 解決策: 単一パーティションキー (`player1Id`) とマルチ属性ソートキー (`matchDate` + `round`) を持つグローバルセカンダリインデックス
- クエリの例: 「プレイヤー 101 のすべての試合」または「2024 年 1 月の選手 101 の試合」

従来およびマルチ属性アプローチの主な違いは、項目構造を調べるときにはっきりします。

従来のグローバルセカンダリインデックスのアプローチ (連結キー):

```
// Manual concatenation required for GSI keys
const item = {
    matchId: 'match-001',                                          // Base table PK
    tournamentId: 'WINTER2024',
    region: 'NA-EAST',
    round: 'SEMIFINALS',
    bracket: 'UPPER',
    player1Id: '101',
    // Synthetic keys needed for GSI
    GSI_PK: `TOURNAMENT#${tournamentId}#REGION#${region}`,       // Must concatenate
    GSI_SK: `${round}#${bracket}#${matchId}`,                    // Must concatenate
    // ... other attributes
};
```

マルチ属性のグローバルセカンダリインデックスのアプローチ (ネイティブキー):

```
// Use existing attributes directly - no concatenation needed
const item = {
    matchId: 'match-001',                                          // Base table PK
    tournamentId: 'WINTER2024',
    region: 'NA-EAST',
    round: 'SEMIFINALS',
    bracket: 'UPPER',
    player1Id: '101',
    matchDate: '2024-01-18',
    // No synthetic keys needed - GSI uses existing attributes directly
    // ... other attributes
};
```

マルチ属性キーでは、自然ドメイン属性を使用して項目を 1 回書き込みます。DynamoDB は、合成連結キーを必要とせずに、複数の GSI 間で自動的にインデックスを作成します。

ベーステーブルスキーマ:

- パーティションキー: `matchId` (1 属性)

グローバルセカンダリインデックススキーマ (マルチ属性キーを持つ TournamentRegionIndex):

- パーティションキー: `tournamentId`、`region` (2 属性)
- ソートキー: `round`、`bracket`、`matchId` (3 属性)

グローバルセカンダリインデックススキーマ (マルチ属性キーを持つ PlayerMatchHistoryIndex):

- パーティションキー: `player1Id` (1 属性)
- ソートキー: `matchDate`、`round` (2 属性)

ベーステーブル: TournamentMatches

matchId (PK)	tournamentId	リージョン	round	ブラケット	player1Id	player2Id	matchDa	勝者	score
match-0(	WINTER 4	NA-EAST	FINALS	CHAMPI HIP	101	103	2024-01-0	101	3-1
match-0(	WINTER 4	NA-EAST	SEMIFIN S	UPPER	101	105	2024-01-8	101	3-2
match-0(	WINTER 4	NA-EAST	SEMIFIN S	UPPER	103	107	2024-01-8	103	3-0
match-0(	WINTER 4	NA-EAST	QUARTE NALS	UPPER	101	109	2024-01-5	101	3-1
match-0(	WINTER 4	NA-WEST	FINALS	CHAMPI HIP	102	104	2024-01-0	102	3-2
match-0(	WINTER 4	NA-WEST	SEMIFIN S	UPPER	102	106	2024-01-8	102	3-1
match-0(	SPRING 4	NA-EAST	QUARTE NALS	UPPER	101	108	2024-03-5	101	3-0
match-0(	SPRING 4	NA-EAST	QUARTE NALS	LOWER	103	110	2024-03-5	103	3-2

GSI: TournamentRegionIndex (マルチ属性キー)

tourname tId (PK)	リージョン (PK)	ラウンド (SK)	ブラケット (SK)	matchId (SK)	player1Id	player2Id	matchDa	勝者	score
WINTER 4	NA-EAST	FINALS	CHAMPI HIP	match-0(	101	103	2024-01-0	101	3-1
WINTER 4	NA-EAST	QUARTE NALS	UPPER	match-0(	101	109	2024-01-5	101	3-1
WINTER 4	NA-EAST	SEMIFIN S	UPPER	match-0(	101	105	2024-01-8	101	3-2
WINTER 4	NA-EAST	SEMIFIN S	UPPER	match-0(	103	107	2024-01-8	103	3-0
WINTER 4	NA-WEST	FINALS	CHAMPI HIP	match-0(	102	104	2024-01-0	102	3-2
WINTER 4	NA-WEST	SEMIFIN S	UPPER	match-0(	102	106	2024-01-8	102	3-1
SPRING 4	NA-EAST	QUARTE NALS	LOWER	match-0(	103	110	2024-03-5	103	3-2
SPRING 4	NA-EAST	QUARTE NALS	UPPER	match-0(	101	108	2024-03-5	101	3-0

GSI: PlayerMatchHistoryIndex (マルチ属性キー)

player1Id (PK)	matchDa (SK)	ラウンド (SK)	tourname tId	リージョン	ブラケット	matchId	player2Id	勝者	score
101	2024-01-5	QUARTE NALS	WINTER 4	NA-EAST	UPPER	match-0(	109	101	3-1

player1Id (PK)	matchDa (SK)	ラウンド (SK)	tourname tId	リージョン	ブラケット	matchId	player2Id	勝者	score
101	2024-01-8	SEMIFIN S	WINTER 4	NA-EAST	UPPER	match-0(	105	101	3-2
101	2024-01-0	FINALS	WINTER 4	NA-EAST	CHAMPI HIP	match-0(	103	101	3-1
101	2024-03-5	QUARTE NALS	SPRING 4	NA-EAST	UPPER	match-0(	108	101	3-0
102	2024-01-8	SEMIFIN S	WINTER 4	NA-WEST	UPPER	match-0(	106	102	3-1
102	2024-01-0	FINALS	WINTER 4	NA-WEST	CHAMPI HIP	match-0(	104	102	3-2
103	2024-01-8	SEMIFIN S	WINTER 4	NA-EAST	UPPER	match-0(	107	103	3-0
103	2024-03-5	QUARTE NALS	SPRING 4	NA-EAST	LOWER	match-0(	110	103	3-2

前提条件

開始する前に、以下の準備が整っていることを確認します。

アカウントとアクセス許可

- アクティブな AWS アカウント (必要に応じてここで作成)
- DynamoDB オペレーションの IAM アクセス許可:
  - `dynamodb:CreateTable`
  - `dynamodb:DeleteTable`
  - `dynamodb:DescribeTable`
  - `dynamodb:PutItem`
  - `dynamodb:Query`

- dynamodb:BatchWriteItem

> **Note**
>
> セキュリティ上の注意: 本番稼働用には、必要なアクセス許可のみを持つカスタム IAM ポリシーを作成します。このチュートリアルでは、AWS 管理ポリシー AmazonDynamoDBFullAccessV2 を使用できます。

開発環境

- マシンにインストールされている「Node.js」
- AWS次のいずれかの方法を使用して設定される 認証情報

オプション 1: AWS CLI

```
aws configure
```

オプション 2: 環境変数

```
export AWS_ACCESS_KEY_ID=your_access_key_here
export AWS_SECRET_ACCESS_KEY=your_secret_key_here
export AWS_DEFAULT_REGION=us-east-1
```

必要なパッケージのインストール

```
npm install @aws-sdk/client-dynamodb @aws-sdk/lib-dynamodb
```

実装

ステップ 1: マルチ属性キーを使用して GSI でテーブルを作成する

シンプルなベースキー構造と、マルチ属性キーを使用する GSI を持つテーブルを作成します。

コード例

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
```

```
const response = await client.send(new CreateTableCommand({
    TableName: 'TournamentMatches',

    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'matchId', KeyType: 'HASH' }              // Simple PK
    ],

    AttributeDefinitions: [
        { AttributeName: 'matchId', AttributeType: 'S' },
        { AttributeName: 'tournamentId', AttributeType: 'S' },
        { AttributeName: 'region', AttributeType: 'S' },
        { AttributeName: 'round', AttributeType: 'S' },
        { AttributeName: 'bracket', AttributeType: 'S' },
        { AttributeName: 'player1Id', AttributeType: 'S' },
        { AttributeName: 'matchDate', AttributeType: 'S' }
    ],

    // GSIs with multi-attribute keys
    GlobalSecondaryIndexes: [
        {
            IndexName: 'TournamentRegionIndex',
            KeySchema: [
                { AttributeName: 'tournamentId', KeyType: 'HASH' },    // GSI PK
 attribute 1
                { AttributeName: 'region', KeyType: 'HASH' },          // GSI PK
 attribute 2
                { AttributeName: 'round', KeyType: 'RANGE' },          // GSI SK
 attribute 1
                { AttributeName: 'bracket', KeyType: 'RANGE' },        // GSI SK
 attribute 2
                { AttributeName: 'matchId', KeyType: 'RANGE' }         // GSI SK
 attribute 3
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'PlayerMatchHistoryIndex',
            KeySchema: [
                { AttributeName: 'player1Id', KeyType: 'HASH' },       // GSI PK
                { AttributeName: 'matchDate', KeyType: 'RANGE' },      // GSI SK
 attribute 1
```

```
                { AttributeName: 'round', KeyType: 'RANGE' }           // GSI SK
 attribute 2
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],

    BillingMode: 'PAY_PER_REQUEST'
}));

console.log("Table with multi-attribute GSI keys created successfully");
```

主要な設計上の決定事項:

ベーステーブル: ベーステーブルは、直接一致検索にシンプルな matchId パーティションキーを使用し、GSI が複雑なクエリパターンを提供する間、ベーステーブル構造を分かりやすく維持します。

TournamentRegionIndex グローバルセカンダリインデックス: TournamentRegionIndex グローバルセカンダリインデックスは、tournamentId + region をマルチ属性パーティションキーとして使用し、両方の属性を組み合わせたハッシュによってデータが分散されるトーナメントリージョンの分離を作成し、特定のトーナメントリージョンのコンテキスト内で効率的なクエリを可能にします。マルチ属性ソートキー (round + bracket + matchId) は、階層の任意のレベルでクエリをサポートする階層ソートを提供し、一般的な (ラウンド) から特定の (マッチ ID) へ自然な順序で並べ替えます。

PlayerMatchHistoryIndex グローバルセカンダリインデックス: PlayerMatchHistoryIndex グローバルセカンダリインデックスは、パーティションキーとして player1Id を使用して選手ごとにデータを再編成し、特定の選手のクロストーナメントクエリを有効にします。マルチ属性ソートキー (matchDate + round) は、日付範囲または特定のトーナメントラウンドでフィルタリングできる時系列の順序を提供します。

ステップ 2: ネイティブ属性を使用してデータを挿入する

自然属性を使用してトーナメントマッチデータを追加します。GSI は、合成キーを必要とせずに、これらの属性を自動的にインデックス化します。

コード例

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
```

```
const docClient = DynamoDBDocumentClient.from(client);

// Tournament match data - no synthetic keys needed for GSIs
const matches = [
    // Winter 2024 Tournament, NA-EAST region
    {
        matchId: 'match-001',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'FINALS',
        bracket: 'CHAMPIONSHIP',
        player1Id: '101',
        player2Id: '103',
        matchDate: '2024-01-20',
        winner: '101',
        score: '3-1'
    },
    {
        matchId: 'match-002',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '101',
        player2Id: '105',
        matchDate: '2024-01-18',
        winner: '101',
        score: '3-2'
    },
    {
        matchId: 'match-003',
        tournamentId: 'WINTER2024',
        region: 'NA-EAST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '103',
        player2Id: '107',
        matchDate: '2024-01-18',
        winner: '103',
        score: '3-0'
    },
    {
        matchId: 'match-004',
        tournamentId: 'WINTER2024',
```

```
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'UPPER',
        player1Id: '101',
        player2Id: '109',
        matchDate: '2024-01-15',
        winner: '101',
        score: '3-1'
    },

    // Winter 2024 Tournament, NA-WEST region
    {
        matchId: 'match-005',
        tournamentId: 'WINTER2024',
        region: 'NA-WEST',
        round: 'FINALS',
        bracket: 'CHAMPIONSHIP',
        player1Id: '102',
        player2Id: '104',
        matchDate: '2024-01-20',
        winner: '102',
        score: '3-2'
    },
    {
        matchId: 'match-006',
        tournamentId: 'WINTER2024',
        region: 'NA-WEST',
        round: 'SEMIFINALS',
        bracket: 'UPPER',
        player1Id: '102',
        player2Id: '106',
        matchDate: '2024-01-18',
        winner: '102',
        score: '3-1'
    },

    // Spring 2024 Tournament, NA-EAST region
    {
        matchId: 'match-007',
        tournamentId: 'SPRING2024',
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'UPPER',
        player1Id: '101',
```

```
        player2Id: '108',
        matchDate: '2024-03-15',
        winner: '101',
        score: '3-0'
    },
    {
        matchId: 'match-008',
        tournamentId: 'SPRING2024',
        region: 'NA-EAST',
        round: 'QUARTERFINALS',
        bracket: 'LOWER',
        player1Id: '103',
        player2Id: '110',
        matchDate: '2024-03-15',
        winner: '103',
        score: '3-2'
    }
];

// Insert all matches
for (const match of matches) {
    await docClient.send(new PutCommand({
        TableName: 'TournamentMatches',
        Item: match
    }));

    console.log(`Added: ${match.matchId} - ${match.tournamentId}/${match.region} -
 ${match.round} ${match.bracket}`);
}

console.log(`\nInserted ${matches.length} tournament matches`);
console.log("No synthetic keys created - GSIs use native attributes automatically");
```

データ構造の説明:

自然属性の使用: 各属性は、文字列の連結や解析を必要とせずに実際のトーナメントの概念を表し、ドメインモデルへの直接マッピングを提供します。

自動グローバルセカンダリインデックスの作成: GSI、合成連結キーを必要とせずに、既存の属性 (TournamentRegionIndex `matchId`の場合は `tournamentId`、`region`、`round`、`bracket`、PlayerMatchHistoryIndex の場合は `player1Id`、`matchDate`、`round`) を使用して項目を自動的にインデックス作成します。

バックフィルが不要: マルチ属性キーを持つ新しいグローバルセカンダリインデックスを既存のテーブルに追加すると、DynamoDB は自然属性を使用して既存のすべての項目を自動的にインデックス化します。合成キーを使用して項目を更新する必要はありません。

ステップ 3: すべてのパーティションキー属性を使用して TournamentRegionIndex グローバルセカンダリインデックスをクエリする

この例では、マルチ属性パーティションキー (tournamentId + region) を持つ TournamentRegionIndex グローバルセカンダリインデックスをクエリします。すべてのパーティションキー属性は、クエリで等号条件で指定する必要があります。tournamentId 単独でクエリを実行したり、パーティションキー属性で不等価演算子を使用したりすることはできません。

コード例

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query GSI: All matches for WINTER2024 tournament in NA-EAST region
const response = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region',
    ExpressionAttributeNames: {
        '#region': 'region',  // 'region' is a reserved keyword
        '#tournament': 'tournament'
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST'
    }
}));

console.log(`Found ${response.Items.length} matches for WINTER2024/NA-EAST:\n`);
response.Items.forEach(match => {
    console.log(`  ${match.round} | ${match.bracket} | ${match.matchId}`);
    console.log(`    Players: ${match.player1Id} vs ${match.player2Id}`);
    console.log(`    Winner: ${match.winner}, Score: ${match.score}\n`);
});
```

正常な出力: 。

```
Found 4 matches for WINTER2024/NA-EAST:

  FINALS | CHAMPIONSHIP | match-001
    Players: 101 vs 103
    Winner: 101, Score: 3-1

  QUARTERFINALS | UPPER | match-004
    Players: 101 vs 109
    Winner: 101, Score: 3-1

  SEMIFINALS | UPPER | match-002
    Players: 101 vs 105
    Winner: 101, Score: 3-2

  SEMIFINALS | UPPER | match-003
    Players: 103 vs 107
    Winner: 103, Score: 3-0
```

無効なクエリ:

```
// Missing region attribute
KeyConditionExpression: 'tournamentId = :tournament'

// Using inequality on partition key attribute
KeyConditionExpression: 'tournamentId = :tournament AND #region > :region'
```

パフォーマンス: マルチ属性パーティションキーは共にハッシュされ、単一属性キーと同じ O(1) 検索パフォーマンスを提供します。

ステップ 4: グローバルセカンダリインデックスのソートキー left-to-right をクエリする

ソートキー属性は、グローバルセカンダリインデックスで定義されている順序で左から右にクエリする必要があります。この例では、TournamentRegionIndex をさまざまな階層レベルでクエリする方法を示します。フィルタリングは、round のみ、round + bracket、または 3 つのソートキー属性すべてで行います。途中で属性をスキップすることはできません。例えば、bracket のスキップ中に round や matchId でクエリを実行することはできません。

コード例

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: Filter by first sort key attribute (round)
console.log("Query 1: All SEMIFINALS matches");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS'
    }
}));
console.log(`  Found ${query1.Items.length} matches\n`);

// Query 2: Filter by first two sort key attributes (round + bracket)
console.log("Query 2: SEMIFINALS UPPER bracket matches");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND bracket = :bracket',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':bracket': 'UPPER'
    }
}));
console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Filter by all three sort key attributes (round + bracket + matchId)
console.log("Query 3: Specific match in SEMIFINALS UPPER bracket");
```

```
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND bracket = :bracket AND matchId = :matchId',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':bracket': 'UPPER',
        ':matchId': 'match-002'
    }
}));
console.log(`  Found ${query3.Items.length} matches\n`);

// Query 4: INVALID - skipping round
console.log("Query 4: Attempting to skip first sort key attribute (WILL FAIL)");
try {
    const query4 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND
 bracket = :bracket',
        ExpressionAttributeNames: {
            '#region': 'region'  // 'region' is a reserved keyword
        },
        ExpressionAttributeValues: {
            ':tournament': 'WINTER2024',
            ':region': 'NA-EAST',
            ':bracket': 'UPPER'
        }
    }));
} catch (error) {
    console.log(`  Error: ${error.message}`);
    console.log(`  Cannot skip sort key attributes - must query left-to-right\n`);
}
```

正常な出力: 。

```
Query 1: All SEMIFINALS matches
```

```
  Found 2 matches

Query 2: SEMIFINALS UPPER bracket matches
  Found 2 matches

Query 3: Specific match in SEMIFINALS UPPER bracket
  Found 1 matches

Query 4: Attempting to skip first sort key attribute (WILL FAIL)
  Error: Query key condition not supported
  Cannot skip sort key attributes - must query left-to-right
```

Left-to-right クエリルール: なにもスキップせずに左から右の順に属性をクエリする必要があります。

有効なパターン:

- 最初の属性のみ: `round = 'SEMIFINALS'`
- 最初の 2 つの属性: `round = 'SEMIFINALS' AND bracket = 'UPPER'`
- 3 つの属性すべて: `round = 'SEMIFINALS' AND bracket = 'UPPER' AND matchId = 'match-002'`

無効なパターン:

- 最初の属性をスキップする: `bracket = 'UPPER'` (ラウンドをスキップ)
- 順不同のクエリ: `matchId = 'match-002' AND round = 'SEMIFINALS'`
- ギャップを残す: `round = 'SEMIFINALS' AND matchId = 'match-002'` (ブラケットをスキップ)

> **Note**
>
> 設計のヒント: クエリの柔軟性を最大化するために、ソートキー属性を最も一般的なものから最も具体的な順に並べます。

ステップ 5: グローバルセカンダリインデックスのソートキーで不等価条件を使用する

不等価条件は、クエリの最後の条件である必要があります。この例では、ソートキー属性で比較演算子 (>=、BETWEEN) とプレフィックスマッチング (begins_with( )) を使用する方法を示します。不

等価演算子を使用すると、その後にソートキー条件を追加することはできません。不等価はキー条件式の最終条件である必要があります。

コード例

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: Round comparison (inequality on first sort key attribute)
console.log("Query 1: Matches from QUARTERFINALS onwards");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 >= :round',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'QUARTERFINALS'
    }
}));
console.log(`  Found ${query1.Items.length} matches\n`);

// Query 2: Round range with BETWEEN
console.log("Query 2: Matches between QUARTERFINALS and SEMIFINALS");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 BETWEEN :start AND :end',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':start': 'QUARTERFINALS',
        ':end': 'SEMIFINALS'
```

```
    }
}));
console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Prefix matching with begins_with (treated as inequality)
console.log("Query 3: Matches in brackets starting with 'U'");
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'TournamentRegionIndex',
    KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND round
 = :round AND begins_with(bracket, :prefix)',
    ExpressionAttributeNames: {
        '#region': 'region'  // 'region' is a reserved keyword
    },
    ExpressionAttributeValues: {
        ':tournament': 'WINTER2024',
        ':region': 'NA-EAST',
        ':round': 'SEMIFINALS',
        ':prefix': 'U'
    }
}));
console.log(`  Found ${query3.Items.length} matches\n`);

// Query 4: INVALID - condition after inequality
console.log("Query 4: Attempting condition after inequality (WILL FAIL)");
try {
    const query4 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region AND
 round > :round AND bracket = :bracket',
        ExpressionAttributeNames: {
            '#region': 'region'  // 'region' is a reserved keyword
        },
        ExpressionAttributeValues: {
            ':tournament': 'WINTER2024',
            ':region': 'NA-EAST',
            ':round': 'QUARTERFINALS',
            ':bracket': 'UPPER'
        }
    }));
} catch (error) {
    console.log(`  Error: ${error.message}`);
    console.log(`  Cannot add conditions after inequality - it must be last\n`);
```

```
}
```

不等価演算子ルール: 比較演算子 (>、>=、<、<=) を使用できます。BETWEEN は範囲クエリに、begins_with( ) はプレフィックスマッチングに使用できます。不等価は、クエリの最後の条件である必要があります。

有効なパターン:

- 等価条件の後に不等価が続きます: `round = 'SEMIFINALS' AND bracket = 'UPPER' AND matchId > 'match-001'`
- 最初の属性の不等価: `round BETWEEN 'QUARTERFINALS' AND 'SEMIFINALS'`
- 最終条件としてのプレフィックスマッチング: `round = 'SEMIFINALS' AND begins_with(bracket, 'U')`

無効なパターン:

- 不等価の後に条件を追加する: `round > 'QUARTERFINALS' AND bracket = 'UPPER'`
- 複数の不等価の使用: `round > 'QUARTERFINALS' AND bracket > 'L'`

> **Important**
>
> begins_with( ) は不等価条件として扱われるため、追加のソートキー条件をフォローすることはできません。

ステップ 6: マルチ属性ソートキーを使用して PlayerMatchHistoryIndex グローバルセカンダリインデックスをクエリする

この例では、単一のパーティションキー (player1Id) とマルチ属性ソートキー (matchDate + round) を持つ PlayerMatchHistoryIndex をクエリします。これにより、トーナメント ID を知らずに特定の選手のすべてのマッチングをクエリすることで、クロストーナメント分析が可能になります。ベーステーブルでは、トーナメントとリージョンの組み合わせごとに個別のクエリが必要になります。

コード例

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, QueryCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

// Query 1: All matches for Player 101 across all tournaments
console.log("Query 1: All matches for Player 101");
const query1 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player',
    ExpressionAttributeValues: {
        ':player': '101'
    }
}));

console.log(`  Found ${query1.Items.length} matches for Player 101:`);
query1.Items.forEach(match => {
    console.log(`    ${match.tournamentId}/${match.region} - ${match.matchDate} -
 ${match.round}`);
});
console.log();

// Query 2: Player 101 matches on specific date
console.log("Query 2: Player 101 matches on 2024-01-18");
const query2 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate = :date',
    ExpressionAttributeValues: {
        ':player': '101',
        ':date': '2024-01-18'
    }
}));

console.log(`  Found ${query2.Items.length} matches\n`);

// Query 3: Player 101 SEMIFINALS matches on specific date
console.log("Query 3: Player 101 SEMIFINALS matches on 2024-01-18");
const query3 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate = :date AND round
 = :round',
    ExpressionAttributeValues: {
```

```
        ':player': '101',
        ':date': '2024-01-18',
        ':round': 'SEMIFINALS'
    }
}));

console.log(`  Found ${query3.Items.length} matches\n`);

// Query 4: Player 101 matches in date range
console.log("Query 4: Player 101 matches in January 2024");
const query4 = await docClient.send(new QueryCommand({
    TableName: 'TournamentMatches',
    IndexName: 'PlayerMatchHistoryIndex',
    KeyConditionExpression: 'player1Id = :player AND matchDate BETWEEN :start
 AND :end',
    ExpressionAttributeValues: {
        ':player': '101',
        ':start': '2024-01-01',
        ':end': '2024-01-31'
    }
}));

console.log(`  Found ${query4.Items.length} matches\n`);
```

パターンのバリエーション

マルチ属性キーを使用した時系列データ

階層時間属性を使用して時系列クエリを最適化する

コード例

```
{
    TableName: 'IoTReadings',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'readingId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'readingId', AttributeType: 'S' },
        { AttributeName: 'deviceId', AttributeType: 'S' },
        { AttributeName: 'locationId', AttributeType: 'S' },
        { AttributeName: 'year', AttributeType: 'S' },
        { AttributeName: 'month', AttributeType: 'S' },
```

```
        { AttributeName: 'day', AttributeType: 'S' },
        { AttributeName: 'timestamp', AttributeType: 'S' }
    ],
    // GSI with multi-attribute keys for time-series queries
    GlobalSecondaryIndexes: [{
        IndexName: 'DeviceLocationTimeIndex',
        KeySchema: [
            { AttributeName: 'deviceId', KeyType: 'HASH' },
            { AttributeName: 'locationId', KeyType: 'HASH' },
            { AttributeName: 'year', KeyType: 'RANGE' },
            { AttributeName: 'month', KeyType: 'RANGE' },
            { AttributeName: 'day', KeyType: 'RANGE' },
            { AttributeName: 'timestamp', KeyType: 'RANGE' }
        ],
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query patterns enabled via GSI:
// - All readings for device in location
// - Readings for specific year
// - Readings for specific month in year
// - Readings for specific day
// - Readings in time range
```

メリット: 自然時間階層 (年 → 月 → 日 → タイムスタンプ) を使用すると、日付の解析や操作を行わずに、いつでも効率的にきめ細やかなクエリを実行できます。グローバルセカンダリインデックスは、自然時間属性を使用してすべての読み取り値を自動的にインデックス化します。

マルチ属性キーを使用した e コマース注文

複数のディメンションを持つ注文を追跡する

コード例

```
{
    TableName: 'Orders',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'orderId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
```

```
        { AttributeName: 'orderId', AttributeType: 'S' },
        { AttributeName: 'sellerId', AttributeType: 'S' },
        { AttributeName: 'region', AttributeType: 'S' },
        { AttributeName: 'orderDate', AttributeType: 'S' },
        { AttributeName: 'category', AttributeType: 'S' },
        { AttributeName: 'customerId', AttributeType: 'S' },
        { AttributeName: 'orderStatus', AttributeType: 'S' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'SellerRegionIndex',
            KeySchema: [
                { AttributeName: 'sellerId', KeyType: 'HASH' },
                { AttributeName: 'region', KeyType: 'HASH' },
                { AttributeName: 'orderDate', KeyType: 'RANGE' },
                { AttributeName: 'category', KeyType: 'RANGE' },
                { AttributeName: 'orderId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'CustomerOrdersIndex',
            KeySchema: [
                { AttributeName: 'customerId', KeyType: 'HASH' },
                { AttributeName: 'orderDate', KeyType: 'RANGE' },
                { AttributeName: 'orderStatus', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// SellerRegionIndex GSI queries:
// - Orders by seller and region
// - Orders by seller, region, and date
// - Orders by seller, region, date, and category

// CustomerOrdersIndex GSI queries:
// - Customer's orders
// - Customer's orders by date
// - Customer's orders by date and status
```

階層組織データ

モデル組織階層

コード例

```
{
    TableName: 'Employees',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'employeeId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'employeeId', AttributeType: 'S' },
        { AttributeName: 'companyId', AttributeType: 'S' },
        { AttributeName: 'divisionId', AttributeType: 'S' },
        { AttributeName: 'departmentId', AttributeType: 'S' },
        { AttributeName: 'teamId', AttributeType: 'S' },
        { AttributeName: 'skillCategory', AttributeType: 'S' },
        { AttributeName: 'skillLevel', AttributeType: 'S' },
        { AttributeName: 'yearsExperience', AttributeType: 'N' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'OrganizationIndex',
            KeySchema: [
                { AttributeName: 'companyId', KeyType: 'HASH' },
                { AttributeName: 'divisionId', KeyType: 'HASH' },
                { AttributeName: 'departmentId', KeyType: 'RANGE' },
                { AttributeName: 'teamId', KeyType: 'RANGE' },
                { AttributeName: 'employeeId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'SkillsIndex',
            KeySchema: [
                { AttributeName: 'skillCategory', KeyType: 'HASH' },
                { AttributeName: 'skillLevel', KeyType: 'RANGE' },
                { AttributeName: 'yearsExperience', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'INCLUDE', NonKeyAttributes: ['employeeId',
 'name'] }
        }
```

```
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// OrganizationIndex GSI query patterns:
// - All employees in company/division
// - Employees in specific department
// - Employees in specific team

// SkillsIndex GSI query patterns:
// - Employees by skill and experience level
```

スパースマルチ属性キー

マルチ属性キーを組み合わせてスパース GSI を作成する

コード例

```
{
    TableName: 'Products',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'productId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'productId', AttributeType: 'S' },
        { AttributeName: 'categoryId', AttributeType: 'S' },
        { AttributeName: 'subcategoryId', AttributeType: 'S' },
        { AttributeName: 'averageRating', AttributeType: 'N' },
        { AttributeName: 'reviewCount', AttributeType: 'N' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'CategoryIndex',
            KeySchema: [
                { AttributeName: 'categoryId', KeyType: 'HASH' },
                { AttributeName: 'subcategoryId', KeyType: 'HASH' },
                { AttributeName: 'productId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'ReviewedProductsIndex',
            KeySchema: [
```

```
                { AttributeName: 'categoryId', KeyType: 'HASH' },
                { AttributeName: 'averageRating', KeyType: 'RANGE' },  // Optional
 attribute
                { AttributeName: 'reviewCount', KeyType: 'RANGE' }     // Optional
 attribute
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// Only products with reviews appear in ReviewedProductsIndex GSI
// Automatic filtering without application logic
// Multi-attribute sort key enables rating and count queries
```

SaaS とマルチテナンシー

顧客分離を備えたマルチテナント SaaS プラットフォーム

コード例

```
// Table design
{
    TableName: 'SaasData',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'resourceId', KeyType: 'HASH' }
    ],
    AttributeDefinitions: [
        { AttributeName: 'resourceId', AttributeType: 'S' },
        { AttributeName: 'tenantId', AttributeType: 'S' },
        { AttributeName: 'customerId', AttributeType: 'S' },
        { AttributeName: 'resourceType', AttributeType: 'S' }
    ],
    // GSI with multi-attribute keys for tenant-customer isolation
    GlobalSecondaryIndexes: [{
        IndexName: 'TenantCustomerIndex',
        KeySchema: [
            { AttributeName: 'tenantId', KeyType: 'HASH' },
            { AttributeName: 'customerId', KeyType: 'HASH' },
            { AttributeName: 'resourceType', KeyType: 'RANGE' },
            { AttributeName: 'resourceId', KeyType: 'RANGE' }
        ],
```

```
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query GSI: All resources for tenant T001, customer C001
const resources = await docClient.send(new QueryCommand({
    TableName: 'SaasData',
    IndexName: 'TenantCustomerIndex',
    KeyConditionExpression: 'tenantId = :tenant AND customerId = :customer',
    ExpressionAttributeValues: {
        ':tenant': 'T001',
        ':customer': 'C001'
    }
}));

// Query GSI: Specific resource type for tenant/customer
const documents = await docClient.send(new QueryCommand({
    TableName: 'SaasData',
    IndexName: 'TenantCustomerIndex',
    KeyConditionExpression: 'tenantId = :tenant AND customerId = :customer AND
 resourceType = :type',
    ExpressionAttributeValues: {
        ':tenant': 'T001',
        ':customer': 'C001',
        ':type': 'document'
    }
}));
```

メリット: テナントと顧客のコンテキストと自然データ組織内の効率的なクエリ。

金融取引

GSI を使用した銀行システム追跡アカウントトランザクション

コード例

```
// Table design
{
    TableName: 'BankTransactions',
    // Base table: Simple partition key
    KeySchema: [
        { AttributeName: 'transactionId', KeyType: 'HASH' }
    ],
```

```
    AttributeDefinitions: [
        { AttributeName: 'transactionId', AttributeType: 'S' },
        { AttributeName: 'accountId', AttributeType: 'S' },
        { AttributeName: 'year', AttributeType: 'S' },
        { AttributeName: 'month', AttributeType: 'S' },
        { AttributeName: 'day', AttributeType: 'S' },
        { AttributeName: 'transactionType', AttributeType: 'S' }
    ],
    GlobalSecondaryIndexes: [
        {
            IndexName: 'AccountTimeIndex',
            KeySchema: [
                { AttributeName: 'accountId', KeyType: 'HASH' },
                { AttributeName: 'year', KeyType: 'RANGE' },
                { AttributeName: 'month', KeyType: 'RANGE' },
                { AttributeName: 'day', KeyType: 'RANGE' },
                { AttributeName: 'transactionId', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        },
        {
            IndexName: 'TransactionTypeIndex',
            KeySchema: [
                { AttributeName: 'accountId', KeyType: 'HASH' },
                { AttributeName: 'transactionType', KeyType: 'RANGE' },
                { AttributeName: 'year', KeyType: 'RANGE' },
                { AttributeName: 'month', KeyType: 'RANGE' }
            ],
            Projection: { ProjectionType: 'ALL' }
        }
    ],
    BillingMode: 'PAY_PER_REQUEST'
}

// Query AccountTimeIndex GSI: All transactions for account in 2023
const yearTransactions = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'AccountTimeIndex',
    KeyConditionExpression: 'accountId = :account AND #year = :year',
    ExpressionAttributeNames: { '#year': 'year' },
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':year': '2023'
    }
```

```
}));

// Query AccountTimeIndex GSI: Transactions in specific month
const monthTransactions = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'AccountTimeIndex',
    KeyConditionExpression: 'accountId = :account AND #year = :year AND #month
 = :month',
    ExpressionAttributeNames: { '#year': 'year', '#month': 'month' },
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':year': '2023',
        ':month': '11'
    }
}));

// Query TransactionTypeIndex GSI: Deposits in 2023
const deposits = await docClient.send(new QueryCommand({
    TableName: 'BankTransactions',
    IndexName: 'TransactionTypeIndex',
    KeyConditionExpression: 'accountId = :account AND transactionType = :type AND #year
 = :year',
    ExpressionAttributeNames: { '#year': 'year' },
    ExpressionAttributeValues: {
        ':account': 'ACC-12345',
        ':type': 'deposit',
        ':year': '2023'
    }
}));
```

完全な例

次の例は、セットアップからクリーンアップまでのマルチ属性キーを示しています。

コード例

```
import {
    DynamoDBClient,
    CreateTableCommand,
    DeleteTableCommand,
    waitUntilTableExists
} from "@aws-sdk/client-dynamodb";
import {
    DynamoDBDocumentClient,
```

```
    PutCommand,
    QueryCommand
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({ region: 'us-west-2' });
const docClient = DynamoDBDocumentClient.from(client);

async function multiAttributeKeysDemo() {
    console.log("Starting Multi-Attribute GSI Keys Demo\n");

    // Step 1: Create table with GSIs using multi-attribute keys
    console.log("1. Creating table with multi-attribute GSI keys...");
    await client.send(new CreateTableCommand({
        TableName: 'TournamentMatches',
        KeySchema: [
            { AttributeName: 'matchId', KeyType: 'HASH' }
        ],
        AttributeDefinitions: [
            { AttributeName: 'matchId', AttributeType: 'S' },
            { AttributeName: 'tournamentId', AttributeType: 'S' },
            { AttributeName: 'region', AttributeType: 'S' },
            { AttributeName: 'round', AttributeType: 'S' },
            { AttributeName: 'bracket', AttributeType: 'S' },
            { AttributeName: 'player1Id', AttributeType: 'S' },
            { AttributeName: 'matchDate', AttributeType: 'S' }
        ],
        GlobalSecondaryIndexes: [
            {
                IndexName: 'TournamentRegionIndex',
                KeySchema: [
                    { AttributeName: 'tournamentId', KeyType: 'HASH' },
                    { AttributeName: 'region', KeyType: 'HASH' },
                    { AttributeName: 'round', KeyType: 'RANGE' },
                    { AttributeName: 'bracket', KeyType: 'RANGE' },
                    { AttributeName: 'matchId', KeyType: 'RANGE' }
                ],
                Projection: { ProjectionType: 'ALL' }
            },
            {
                IndexName: 'PlayerMatchHistoryIndex',
                KeySchema: [
                    { AttributeName: 'player1Id', KeyType: 'HASH' },
                    { AttributeName: 'matchDate', KeyType: 'RANGE' },
                    { AttributeName: 'round', KeyType: 'RANGE' }
```

```
                ],
                Projection: { ProjectionType: 'ALL' }
            }
        ],
        BillingMode: 'PAY_PER_REQUEST'
    }));

    await waitUntilTableExists({ client, maxWaitTime: 120 }, { TableName:
 'TournamentMatches' });
    console.log("Table created\n");

    // Step 2: Insert tournament matches
    console.log("2. Inserting tournament matches...");
    const matches = [
        { matchId: 'match-001', tournamentId: 'WINTER2024', region: 'NA-EAST', round:
 'FINALS', bracket: 'CHAMPIONSHIP', player1Id: '101', player2Id: '103', matchDate:
 '2024-01-20', winner: '101', score: '3-1' },
        { matchId: 'match-002', tournamentId: 'WINTER2024', region: 'NA-EAST', round:
 'SEMIFINALS', bracket: 'UPPER', player1Id: '101', player2Id: '105', matchDate:
 '2024-01-18', winner: '101', score: '3-2' },
        { matchId: 'match-003', tournamentId: 'WINTER2024', region: 'NA-WEST', round:
 'FINALS', bracket: 'CHAMPIONSHIP', player1Id: '102', player2Id: '104', matchDate:
 '2024-01-20', winner: '102', score: '3-2' },
        { matchId: 'match-004', tournamentId: 'SPRING2024', region: 'NA-EAST', round:
 'QUARTERFINALS', bracket: 'UPPER', player1Id: '101', player2Id: '108', matchDate:
 '2024-03-15', winner: '101', score: '3-0' }
    ];

    for (const match of matches) {
        await docClient.send(new PutCommand({ TableName: 'TournamentMatches', Item:
 match }));
    }
    console.log(`Inserted ${matches.length} tournament matches\n`);

    // Step 3: Query GSI with multi-attribute partition key
    console.log("3. Query TournamentRegionIndex GSI: WINTER2024/NA-EAST matches");
    const gsiQuery1 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'TournamentRegionIndex',
        KeyConditionExpression: 'tournamentId = :tournament AND #region = :region',
        ExpressionAttributeNames: { '#region': 'region' },
        ExpressionAttributeValues: { ':tournament': 'WINTER2024', ':region': 'NA-
EAST' }
    }));
```

```
    console.log(`  Found ${gsiQuery1.Items.length} matches:`);
    gsiQuery1.Items.forEach(match => {
        console.log(`    ${match.round} - ${match.bracket} - ${match.winner} won`);
    });

    // Step 4: Query GSI with multi-attribute sort key
    console.log("\n4. Query PlayerMatchHistoryIndex GSI: All matches for Player 101");
    const gsiQuery2 = await docClient.send(new QueryCommand({
        TableName: 'TournamentMatches',
        IndexName: 'PlayerMatchHistoryIndex',
        KeyConditionExpression: 'player1Id = :player',
        ExpressionAttributeValues: { ':player': '101' }
    }));

    console.log(`  Found ${gsiQuery2.Items.length} matches for Player 101:`);
    gsiQuery2.Items.forEach(match => {
        console.log(`    ${match.tournamentId}/${match.region} - ${match.matchDate} -
 ${match.round}`);
    });

    console.log("\nDemo complete");
    console.log("No synthetic keys needed - GSIs use native attributes automatically");
}

async function cleanup() {
    console.log("Deleting table...");
    await client.send(new DeleteTableCommand({ TableName: 'TournamentMatches' }));
    console.log("Table deleted");
}

// Run demo
multiAttributeKeysDemo().catch(console.error);

// Uncomment to cleanup:
// cleanup().catch(console.error);
```

最小コードの雛形

コード例

```
// 1. Create table with GSI using multi-attribute keys
await client.send(new CreateTableCommand({
    TableName: 'MyTable',
```

```
    KeySchema: [
        { AttributeName: 'id', KeyType: 'HASH' }        // Simple base table PK
    ],
    AttributeDefinitions: [
        { AttributeName: 'id', AttributeType: 'S' },
        { AttributeName: 'attr1', AttributeType: 'S' },
        { AttributeName: 'attr2', AttributeType: 'S' },
        { AttributeName: 'attr3', AttributeType: 'S' },
        { AttributeName: 'attr4', AttributeType: 'S' }
    ],
    GlobalSecondaryIndexes: [{
        IndexName: 'MyGSI',
        KeySchema: [
            { AttributeName: 'attr1', KeyType: 'HASH' },    // GSI PK attribute 1
            { AttributeName: 'attr2', KeyType: 'HASH' },    // GSI PK attribute 2
            { AttributeName: 'attr3', KeyType: 'RANGE' },   // GSI SK attribute 1
            { AttributeName: 'attr4', KeyType: 'RANGE' }    // GSI SK attribute 2
        ],
        Projection: { ProjectionType: 'ALL' }
    }],
    BillingMode: 'PAY_PER_REQUEST'
}));

// 2. Insert items with native attributes (no concatenation needed for GSI)
await docClient.send(new PutCommand({
    TableName: 'MyTable',
    Item: {
        id: 'item-001',
        attr1: 'value1',
        attr2: 'value2',
        attr3: 'value3',
        attr4: 'value4',
        // ... other attributes
    }
}));

// 3. Query GSI with all partition key attributes
await docClient.send(new QueryCommand({
    TableName: 'MyTable',
    IndexName: 'MyGSI',
    KeyConditionExpression: 'attr1 = :v1 AND attr2 = :v2',
    ExpressionAttributeValues: {
        ':v1': 'value1',
        ':v2': 'value2'
```

```
    }
}));

// 4. Query GSI with sort key attributes (left-to-right)
await docClient.send(new QueryCommand({
    TableName: 'MyTable',
    IndexName: 'MyGSI',
    KeyConditionExpression: 'attr1 = :v1 AND attr2 = :v2 AND attr3 = :v3',
    ExpressionAttributeValues: {
        ':v1': 'value1',
        ':v2': 'value2',
        ':v3': 'value3'
    }
}));

// Note: If any attribute name is a DynamoDB reserved keyword, use
 ExpressionAttributeNames:
// KeyConditionExpression: 'attr1 = :v1 AND #attr2 = :v2'
// ExpressionAttributeNames: { '#attr2': 'attr2' }
```

その他のリソース

- DynamoDB のベストプラクティス
- テーブルとデータの操作
- グローバルセカンダリインデックス:
- クエリおよびスキャンオペレーション

DynamoDB のグローバルセカンダリインデックスの管理

このセクションでは、Amazon DynamoDB でグローバルセカンダリインデックスを作成、変更、削除する方法について説明します。

トピック

- グローバルセカンダリインデックスを持つテーブルの作成
- テーブルのグローバルセカンダリインデックスの説明
- グローバルセカンダリインデックスの既存テーブルへの追加
- グローバルセカンダリインデックスの削除
- 作成時のグローバルセカンダリインデックスの変更

グローバルセカンダリインデックスを持つテーブルの作成

1 つ以上のグローバルセカンダリインデックスを持つテーブルを作成するには、CreateTable オペレーションと GlobalSecondaryIndexes パラメータを使用します。最大限のクエリの柔軟性を得るために、テーブルごとに最大 20 個のグローバルセカンダリインデックス (デフォルトのクォータ) を作成できます。

インデックスパーティションキーとして機能する属性を 1 つ指定する必要があります。必要に応じて、インデックスソートキーに別の属性を指定できます。これらのキー属性のいずれも、テーブルのキー属性と同じである必要はありません。例えば、GameScores テーブル (「DynamoDB のグローバルセカンダリインデックスの使用」参照) では、TopScore も TopScoreDateTime もキー属性ではありません。パーティションキーとして TopScore、ソートキーとして TopScoreDateTime を使用して、グローバルセカンダリインデックスを作成できます。このようなインデックスを使用して、ハイスコアとゲームのプレイ時刻との間に相関があるかどうかを判定できる可能性があります。

各インデックスキー属性は、String、Number、または Binary タイプのスカラーである必要があります。(ドキュメントまたはセットは指定できません)。グローバルセカンダリインデックスには、任意のデータ型の属性を射影できます。これには、スカラー、ドキュメント、およびセットが含まれます。データ型の詳細なリストについては、「データ型」を参照してください。

プロビジョニング済みモードを使用する場合は、インデックスに ReadCapacityUnits および WriteCapacityUnits で構成される ProvisionedThroughput 設定をする必要があります。これらのプロビジョニングされたスループット設定は、テーブルの設定とは別ですが、同様の動作をします。詳細については、「」を参照してくださいグローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項

グローバルセカンダリーインデックスは、基本テーブルから読み込み/書き込みキャパシティーモードを継承します。詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。

> **Note**
>
> 新しい GSI を作成する際、選択したパーティションキーが、新しいインデックスのパーティションキー値全体でデータやトラフィックの分散が不均一だったり、狭くなっているかを確認することが重要になります。この場合、バックフィル操作と書き込み操作が同時に発生し、ベーステーブルへの書き込みをスロットリングしている可能性があります。このサービスでは、このシナリオの可能性を最小限に抑えるための対策を講じていますが、インデックスパーティションキー、選択した射影、またはインデックスプライマリキーのスパース性について、カスタマーデータのインサイトを見える形で取得していません。

新しいグローバルセカンダリインデックスで、パーティションキー値全体でデータやトラフィックの分散が狭められたり、歪められたりしている可能性がある場合は、運用上重要なテーブルに新しいインデックスを追加する前に、次の点を考慮してください。

- アプリケーションのトラフィック量が最も少ない時間帯にインデックスを追加するのが、最も安全な方法と言えます。
- ベーステーブルとインデックスで CloudWatch Contributor Insights 有効にすることを検討してください。これにより、トラフィックの分散に関する貴重なインサイトが得られます。
- `OnlineIndexPercentageProgress` CloudWatch メトリクスをモニタリングしてインデックスビルドの進行状況を追跡し、ベーステーブルの `WriteThrottleEvents` メトリクスと `ThrottledRequests` メトリクスで進行中のアプリケーショントラフィックに影響するスロットリングを監視します。インデックスのプロビジョニングされた書き込み容量を増やしても、インデックスの構築にかかる時間は短縮されません。`OnlineIndexConsumedWriteCapacity` および `OnlineIndexThrottleEvents` メトリクスは、バックフィル中に 0 と表示されます。
- 書き込みスロットリングによる運用上の影響が発生した場合は、インデックスの作成をキャンセルする準備をしてください。

テーブルのグローバルセカンダリインデックスの説明

テーブルのすべてのグローバルセカンダリインデックスのステータスを表示するには、`DescribeTable` オペレーションを使用します。レスポンスの `GlobalSecondaryIndexes` 部分は、テーブル上のすべてのインデックスと、それぞれの現在のステータス (`IndexStatus`) を示しています。

グローバルセカンダリインデックスの `IndexStatus` は、次のいずれかになります。

- `CREATING` – インデックスは現在作成されていますが、まだ使用することはできません。
- `ACTIVE` – インデックスは使用可能になり、アプリケーションはインデックスに対して `Query` オペレーションを実行できます。
- `UPDATING` – インデックスのプロビジョニングされたスループット設定の変更中です。
- `DELETING` – インデックスは現在削除されているため、使用できなくなっています。

DynamoDB がグローバルセカンダリインデックスの作成を完了すると、インデックスのステータスが `CREATING` から `ACTIVE` に変わります。

グローバルセカンダリインデックスの既存テーブルへの追加

既存のテーブルにグローバルセカンダリインデックスを追加するには、UpdateTable オペレーションと GlobalSecondaryIndexUpdates パラメータを使用します。以下の情報が必要です。

- インデックス名。名前は、テーブル内のすべてのインデックスにおいて一意である必要があります。
- インデックスのキースキーマ。インデックスパーティションキーには 1 つの属性を指定する必要があります。必要に応じて、インデックスソートキーに別の属性を指定できます。これらのキー属性のいずれも、テーブルのキー属性と同じである必要はありません。各スキーマ属性のデータ型は、String、Number、または Binary のスカラーである必要があります。
- テーブルからインデックスに射影される属性は以下の通りです。
  - KEYS_ONLY – インデックス内の各項目は、テーブルパーティションキーとソートキーの値、およびインデックスキーの値のみで構成されます。
  - INCLUDE – KEYS_ONLY の属性に加えて、セカンダリインデックスにその他の非キー属性が含まれるように指定できます。
  - ALL – インデックスには、ソーステーブルのすべての属性が含まれます。
- インデックスのプロビジョニングされたスループット設定で、ReadCapacityUnits および WriteCapacityUnits で構成されます。これは、テーブルのプロビジョニングされたスループット設定とは異なります。

UpdateTable オペレーションについて作成できるグローバルセカンダリインデックスは 1 つだけです。

インデックス作成のフェーズ

新しいグローバルセカンダリインデックスを既存のテーブルに追加すると、インデックスの構築中もテーブルが使用可能になります。ただし、新しいインデックスは、そのステータスが CREATING から ACTIVE に変わるまでクエリオペレーションに使用できません。

> **Note**
>
> グローバルセカンダリインデックスの作成では、Application Auto Scaling を使用しません。MIN Application Auto Scaling の容量を増やしても、グローバルセカンダリインデックスの作成時間は短縮されません。

その裏では、DynamoDB は 2 つのフェーズでインデックスを構築しています。

リソース割り当て

DynamoDB は、インデックスの構築に必要なコンピューティングリソースとストレージリソースを割り当てます。

リソース割り当てフェーズでは、IndexStatus 属性は CREATING、Backfilling 属性は false です。DescribeTable オペレーションを使用して、テーブルとそのセカンダリインデックスすべてのステータスを取得します。

インデックスがリソース割り当てフェーズにある間は、インデックスやその親テーブルを削除することはできません。インデックスまたはテーブルのプロビジョニングされたスループットを変更することもできません。テーブル上の他のインデックスを追加したり削除したりすることはできません。ただし、これらの他のインデックスのプロビジョニングされたスループットは変更できます。

バックフィル

テーブルの各項目について、DynamoDB は、その射影 (KEYS_ONLY、INCLUDE、または ALL) に基づいて、インデックスに書き込む属性のセットを決定します。次に、これらの属性をインデックスに書き込みます。バックフィルフェーズでは、DynamoDB はテーブルで追加、削除、または更新される項目を追跡します。これらの項目の属性も、必要に応じてインデックスで追加、削除、または更新されます。

バックフィルフェーズでは、IndexStatus 属性は CREATING に設定されていて、Backfilling 属性は true です。DescribeTable オペレーションを使用して、テーブルとそのセカンダリインデックスすべてのステータスを取得します。

インデックスのバックフィル中は、親テーブルを削除したり、テーブルに他のインデックスを追加または削除したりすることはできません。ただし、作成中のインデックスを削除したり、テーブルとそのグローバルセカンダリインデックスのプロビジョニングされたスループットを変更したりすることはできます。

> **Note**
>
> バックフィルフェーズでは、違反する項目の書き込みの一部が成功し、他の項目が拒否されることがあります。バックフィル後、新しいインデックスのキースキーマに違反する項目への書き込みはすべて拒否されます。バックフィルフェーズの終了後に Violation Detector ツールを実行して、発生した可能性のあるキー違反を検出して解決することを

お勧めします。詳細については、「」を参照してくださいDynamoDB でのインデックスキー違反の検出と修正

リソース割り当てフェーズとバックフィルフェーズの進行中、インデックスは CREATING 状態になっています。この間、DynamoDB はテーブルに対して読み込みオペレーションを実行します。グローバルセカンダリインデックスを設定するためのベーステーブルからの読み込みオペレーションに対しては課金されません。

インデックス構築が完了すると、ステータスは ACTIVE に変わります。インデックスが ACTIVE になるまで、Query や Scan はできません。

> **Note**
>
> 場合によっては、DynamoDB は、インデックスキーの違反のため、テーブルからインデックスにデータを書き込めない場合があります。これは、以下の場合に発生します。
>
> - 属性値のデータ型が、インデックスキーのスキーマデータ型のデータ型と一致しません。
> - 属性のサイズが、インデックスキー属性の最大長を超えています。
> - インデックスキー属性には、空の文字列または空のバイナリ属性値があります。
>
> インデックスキー違反があっても、グローバルセカンダリインデックスの作成はできます。ただし、インデックスが ACTIVE になると、違反しているキーはインデックスに存在しなくなります。
> DynamoDB には、これらの問題を検出して解決するためのスタンドアロンツールが用意されています。詳細については、「DynamoDB でのインデックスキー違反の検出と修正」を参照してください。

大きなテーブルへのグローバルセカンダリインデックスの追加

グローバルセカンダリインデックスの構築に必要な時間は、次のようないくつかの要因によって異なります。

- テーブルのサイズ
- インデックスに含めることができるテーブル内の項目の数
- インデックスに射影される属性の数。

- インデックス構築中のメインテーブルへの書き込みアクティビティ

非常に大きなテーブルにグローバルセカンダリインデックスを追加する場合、作成プロセスが完了するまで時間がかかることがあります。進行状況をモニタリングし、インデックスに十分な書き込み容量があるかどうかを判断するには、次の Amazon CloudWatch メトリクスを参照してください。

- `OnlineIndexPercentageProgress`

DynamoDB に関する CloudWatch メトリクスの詳細については、「DynamoDB のメトリクス」を参照してください。

> **Important**
>
> グローバルセカンダリインデックスを作成または更新する前に、非常に大きなテーブルを許可リストする必要がある場合があります。AWS サポートに連絡して、テーブルの許可リストを作成してください。

インデックスがバックフィルされている間、DynamoDB は内部システム容量を使用してテーブルから読み込みます。これは、インデックスの作成による影響を最小限に抑え、テーブルの読み込みキャパシティーが不足しないようにするためです。

グローバルセカンダリインデックスの削除

グローバルセカンダリインデックスが不要になった場合には、`UpdateTable` オペレーションを使用して削除することができます。

グローバルセカンダリインデックスは、1 回の `UpdateTable` オペレーションで 1 つだけ削除できます。

グローバルセカンダリインデックスを削除するには、`GlobalSecondaryIndexUpdates` パラメータの `Delete` アクションで `UpdateTable` オペレーションを使用し、削除するインデックスの名前を指定します。次の AWS CLI の例では、`MyGSI` という名前のインデックスを削除します。

```
aws dynamodb update-table \
    --table-name MyTable \
    --global-secondary-index-updates '[{"Delete": {"IndexName": "MyGSI"}}]'
```

グローバルセカンダリインデックスが削除されている間、親テーブルの読み込みまたは書き込みアクティビティに影響はありません。削除の進行中でも、他のインデックスでプロビジョニングされたスループットを変更できます。

> **Note**
>
> - DeleteTable アクションを使用してテーブルを削除すると、そのテーブルのすべてのグローバルセカンダリインデックスも削除されます。
> - アカウントではグローバルセカンダリインデックスの削除操作に対して課金されません。

作成時のグローバルセカンダリインデックスの変更

インデックスが作成されている間は、DescribeTable オペレーションを使用して、どのフェーズにあるかを判断します。インデックスの説明に含まれているブール属性 Backfilling は、DynamoDB がテーブルから項目を含むインデックスを現在ロードしているかどうかを示します。Backfilling が true の場合、リソース割り当てフェーズは完了していて、インデックスがバックフィルされています。

バックフィルフェーズでは、作成中のインデックスを削除できます。このフェーズでは、テーブルの他のインデックスを追加または削除することはできません。

> **Note**
>
> CreateTable オペレーションの一部として作成されたインデックスの場合、Backfilling 属性は DescribeTable 出力に現れません。詳細については、「インデックス作成のフェーズ」を参照してください。

DynamoDB でのインデックスキー違反の検出と修正

グローバルセカンダリインデックスの作成のバックフィルフェーズでは、Amazon DynamoDB はテーブルの各項目を調べて、インデックスに含める適格性があるかどうかを判断します。一部の項目は、インデックスキー違反の原因となるため、適格でない場合があります。このような場合、項目はテーブルに残りますが、インデックスにはその項目に対応するエントリがなくなります。

インデックスキー違反は次の場合に発生します。

- 属性値とインデックスキーのスキーマのデータ型が一致しない場合。たとえば、GameScores テーブルの項目の 1 つが、String 型の TopScore 値を持っていたとします。Number 型の TopScore のパーティションキーを持つグローバルセカンダリインデックスを追加した場合、テーブルの項目はインデックスキーに違反します。
- テーブルの属性値が、インデックスキー属性の最大長を超えている場合。パーティションキーの最大長は 2048 バイトで、ソートキーの最大長は 1024 バイトです。テーブル内の対応する属性値のいずれかがこれらの制限を超えると、テーブルの項目はインデックスキーに違反します。

> **Note**
>
> インデックスキーとして使用される属性に String 属性または Binary 属性値が設定されている場合、属性値の長さは 0 より大きくする必要があります。そうしないと、テーブルの項目がインデックスキーに違反します。
> 現時点では、このツールはこのインデックスキー違反にフラグを設定しません。

インデックスキー違反が発生した場合でも、バックフィルフェーズは中断することなく続行されます。ただし、違反している項目は、インデックスに含まれません。バックフィルフェーズ完了後は、新しいインデックスのキースキーマに違反する項目へのすべての書き込みが拒否されます。

インデックスキーに違反するテーブル内の属性値を識別して修正するには、Violation Detector ツールを使用します。Violation Detector を実行するには、スキャンするテーブルの名前、グローバルセカンダリインデックスパーティションキーとソートキーの名前とデータ型、インデックスキー違反が見つかった場合に実行するアクションを指定する設定ファイルを作成します。Violation Detector は、次の 2 つのモードのいずれかで実行できます。

- 検出モード – インデックスキー違反を検出します。検出モードを使用して、グローバルセカンダリインデックスでキー違反の原因となるテーブル内の項目を報告します。(必要に応じて、違反しているテーブル項目が見つかるとすぐに削除するように要求できます)。検出モードからの出力はファイルに書き込まれるので、これを使用してさらに分析できます。
- 修正モード – インデックスキー違反を修正します。修正モードでは、Violation Detector は検出モードからの出力ファイルと同じ形式の入力ファイルを読み込みます。修正モードでは、入力ファイルからレコードを読み込み、各レコードについて、テーブル内の対応する項目を削除または更新します。(項目の更新を選択した場合は、入力ファイルを編集して、これらの更新に適切な値を設定する必要があります)。

Violation Detector のダウンロードおよび実行

Violation Detector は、実行可能な Java Archive (`.jar` ファイル) で、Windows、macOS、または Linux コンピュータ上で実行されます。Violation Detector には Java 1.7 (以降) と Apache Maven が必要です。

- GitHub から Violation Detector をダウンロードします

`README.md` ファイルの指示に従って、Maven を使用して Violation Detector をダウンロードしてインストールします。

Violation Detector を起動するには、`ViolationDetector.java` を構築したディレクトリで、次のコマンドを入力します。

```
java -jar ViolationDetector.jar [options]
```

Violation Detector ツールのコマンドラインでは、以下のオプションが使用できます。

- `-h | --help` – Violation Detector の使用方法の概要とオプションを出力します。
- `-p | --configFilePath value` – Violation Detector 設定ファイルの完全修飾名。詳細については、「」を参照してくださいViolation Detector 設定ファイル
- `-t | --detect value` – テーブル内のインデックスキー違反を検出し、Violation Detector の出力ファイルに書き込みます。このパラメータの値が `keep` に設定されている場合、キー違反のある項目は変更されません。値が `delete` に設定されている場合、キー違反のある項目はテーブルから削除されます。
- `-c | --correct value` – 入力ファイルからインデックスキー違反を読み込み、テーブル内の項目に対して修正アクションを実行します。このパラメータの値が `update` に設定されている場合、キー違反のある項目は、違反していない新しい値で更新されます。値が `delete` に設定されている場合、キー違反のある項目はテーブルから削除されます。

Violation Detector 設定ファイル

実行時に、Violation Detector ツールには設定ファイルが必要です。このファイルのパラメータによって、Violation Detector がアクセスできる DynamoDB リソースと、プロビジョニングされたスループットを使用できる量が決まります。以下の表は、これらのパラメータの説明です。

パラメータ名	説明	必須?

awsCredentialsFile	AWS 認証情報を含むファイルの完全修飾名。認証情報ファイルの形式は、次のようになります。
	
	`accessKey = access_key_id_goes_here`
	`secretKey = secret_key_goes_here`
		はい

dynamoDBRegion	テーブルが存在する AWS リージョン。例: us-west-2 。	はい
tableName	スキャンされる DynamoDB テーブルの名前。	はい
gsiHashKeyName	インデックスパーティションキーの名前。	はい
gsiHashKeyType	インデックスパーティションキーのデータ型 – String、Number、または Binary: S | N | B	はい
gsiRangeKeyName	インデックスソートキーの名前。インデックスにシンプルなプライマリキー (パーティションキー) のみがある場合は、このパラメータを指定しないでください。	いいえ
gsiRangeKeyType	インデックスソートキーのデータ型 –	いいえ

パラメータ名	説明	必須?
	String、Number、または Binary: S | N | B インデックスにシンプルなプライマリキー (パーティションキー) のみがある場合は、このパラメータを指定しないでください。	
recordDetails	インデックスキー違反の詳細を出力ファイルに書き込むかどうか。設定が true の場合 (デフォルト)、違反項目に関する完全な情報が報告されます。設定が false の場合、違反の数のみが報告されます。	いいえ
recordGsiValueInViolationRecord	違反しているインデックスキーの値を出力ファイルに書き込むかどうか。設定が true の場合 (デフォルト)、キー値が報告されます。設定が false の場合、キー値は報告されません。	いいえ

パラメータ名	説明	必須?
detectionOutputPath	Violation Detector の出力ファイルのフルパス。このパラメータは、ローカルディレクトリまたはAmazon Simple Storage Service (Amazon S3) への書き込みをサポートしています。次に例を示します。 detectionOutputPath = //*local/path/filename.csv* detectionOutputPath = s3://*bucket*/*filename.csv* 出力ファイル内の情報は、CSV (Comma-Separated Values) 形式で表示されます。detectionOutputPath を設定しない場合、出力ファイル名は violation_detection.csv になり、現在の作業ディレクトリに書き込まれます。	いいえ

パラメータ名	説明	必須?
numOfSegments	Violation Detector がテーブルをスキャンするときに使用される並列スキャンセグメントの数。デフォルト値は 1 です。これは、テーブルがシーケンシャルにスキャンされることを意味します。値が 2 以上の場合、Violation Detector はテーブルをその数の論理セグメントに分割し、同じ数のスキャンスレッドにします。 numOfSegments の設定は最大 4,096 です。 大きなテーブルの場合、並列スキャンはシーケンシャルスキャンよりも一般的に高速です。さらに、テーブルが複数のパーティションにまたがるほど大きい場合、並列スキャンによって読み込みアクティビティが複数のパーティションに均等に分散されます。DynamoDB の並列スキャンの詳細については、「並列スキャン」を参照してください。	いいえ

パラメータ名	説明	必須?
numOfViolations	出力ファイルに書き込むインデックスキー違反の上限値。設定が -1 の場合 (デフォルト)、テーブル全体がスキャンされます。正の整数に設定すると、Violation Detector は設定した数の違反を検出した後に停止します。	いいえ
numOfRecords	スキャンされるテーブル内の項目数。設定が -1 の場合 (デフォルト)、テーブル全体がスキャンされます。正の整数に設定すると、Violation Detector はテーブル内の設定した数の項目をスキャンした後に停止します。	いいえ
readWriteIOPSPercent	テーブルスキャン中に使用される、プロビジョニングされた読み込みキャパシティーユニットの割合を調整します。有効な値の範囲は 1 ~ 100 です。デフォルト値 (25) は、Violation Detector がテーブルのプロビジョニングされた読み込みスループットの 25% 以下を使用することを意味します。	いいえ

パラメータ名	説明	必須?
correctionInputPath	Violation Detector 修正入力ファイルのフルパス。Violation Detector を修正モードで実行すると、このファイルの内容を使用して、グローバルセカンダリインデックスに違反するテーブル内のデータ項目を変更または削除します。 correctionInputPath ファイルの形式は、detectionOutputPath ファイルの形式と同じです。これにより、検出モードからの出力を修正モードの入力として処理できます。	いいえ

パラメータ名	説明	必須?

correctionOutputPath	Violation Detector 修正出力ファイルのフルパス。このファイルは、更新エラーがある場合にのみ作成されます。
	
	このパラメータは、ローカルディレクトリまたは Amazon S3 への書き込みをサポートします。次に例を示します。
	
	`correctionOutputPath = //local/path/filename.csv`
	
	`correctionOutputPath = s3://bucket/filename.csv`
	
	出力ファイルの情報は CSV 形式で表示されます。`correctionOutputPath` を設定しない場合、出力ファイル名は `violation_update_errors.csv` になり、現在の作業ディレクトリに書き込まれます。
		いいえ

検出

インデックスキー違反を検出するには、Violation Detector を `--detect` コマンドラインオプションで使用します。このオプションがどのように機能するかを示すには、`ProductCatalog` テーブルを検討してください。以下は、テーブルの項目の一覧です。プライマリキー (`Id`) と `Price` 属性のみを表示しています。

ID (プライマリキー)	価格
101	5
102	20
103	200
201	100
202	200
203	300
204	400
205	500

Price の値はすべて、Number 型です。ただし、DynamoDB はスキーマレスであるため、数値以外の Price の項目を追加することができます。たとえば、ProductCatalog テーブルに別の項目を追加するとします。

ID (プライマリキー)	価格
999	"Hello"

これで、テーブルには合計 9 つの項目が追加されました。

ここで、新しいグローバルセカンダリインデックスをテーブルに追加します: PriceIndex。このインデックスのプライマリキーは、パーティションキー Price で、Number 型です。インデックスが構築されると、8 つの項目が含まれますが、ProductCatalog テーブルには 9 つの項目があります。この不一致の理由は、値 "Hello" は String 型ですが、PriceIndex は Number 型のプライマリキーを持っているからです。String 値がグローバルセカンダリインデックスキーに違反するため、インデックスには存在しません。

この状況で Violation Detector を使用するには、まず次のような設定ファイルを作成します。

```
# Properties file for violation detection tool configuration.
# Parameters that are not specified will use default values.

awsCredentialsFile = /home/alice/credentials.txt
dynamoDBRegion = us-west-2
tableName = ProductCatalog
gsiHashKeyName = Price
gsiHashKeyType = N
recordDetails = true
recordGsiValueInViolationRecord = true
detectionOutputPath = ./gsi_violation_check.csv
correctionInputPath = ./gsi_violation_check.csv
numOfSegments = 1
readWriteIOPSPercent = 40
```

続いて、次の例のように、Violation Detector を実行します。

```
$  java -jar ViolationDetector.jar --configFilePath config.txt --detect keep

Violation detection started: sequential scan, Table name: ProductCatalog, GSI name:
 PriceIndex
Progress: Items scanned in total: 9,    Items scanned by this thread: 9,    Violations
 found by this thread: 1, Violations deleted by this thread: 0
Violation detection finished: Records scanned: 9, Violations found: 1, Violations
 deleted: 0, see results at: ./gsi_violation_check.csv
```

`recordDetails` 設定パラメータが `true` に設定されている場合、Violation Detector は、次の例のように、各違反の詳細を出力ファイルに書き込みます。

```
Table Hash Key,GSI Hash Key Value,GSI Hash Key Violation Type,GSI Hash Key Violation
 Description,GSI Hash Key Update Value(FOR USER),Delete Blank Attributes When Updating?
(Y/N)

999,"{""S"":""Hello""}",Type Violation,Expected: N Found: S,,
```

出力ファイルは CSV 形式です。ファイルの最初の行はヘッダーで、2 行目以降はインデックスキーに違反する項目ごとに 1 つのレコードが続きます。これらの違反レコードのフィールドは次のとおりです。

- テーブルハッシュキー – テーブル内の項目のパーティションキー値です。
- テーブル範囲キー – テーブル内の項目のソートキー値です。

- GSI ハッシュキー値 – グローバルセカンダリインデックスのパーティションキー値です。
- GSI ハッシュキー違反タイプ – `Type Violation` または `Size Violation`。
- GSI ハッシュキー違反の説明 – 違反の原因です。
- GSI ハッシュキー更新値 (ユーザー用) – 修正モードでは、ユーザーが指定した属性の新しい値。
- GSI 範囲キー値 – グローバルセカンダリインデックスのソートキー値です。
- GSI 範囲キー違反タイプ – `Type Violation` または `Size Violation`。
- GSI 範囲キー違反の説明 – 違反の原因です。
- GSI 範囲キーの更新値 (ユーザー用) – 修正モードでは、ユーザーが指定した属性の新しい値。
- 更新時に空白の属性を削除 (Y/N) – 修正モードでは、テーブル内の違反項目を削除する (Y) か、保持する (N) かを決定します。ただし、次のフィールドのいずれかが空白の場合のみ実行されます。
  - `GSI Hash Key Update Value(FOR USER)`
  - `GSI Range Key Update Value(FOR USER)`

  これらのフィールドのいずれかが空白でない場合、`Delete Blank Attribute When Updating(Y/N)` は何も実行しません。

> **Note**
>
> 出力形式は、設定ファイルとコマンドラインオプションによって異なることがあります。例えば、テーブルに単純なプライマリキー (ソートキーなし) がある場合、出力にはソートキーフィールドは表示されません。
> ファイル内の違反レコードがソート順でない可能性があります。

修正

インデックスキーの違反を修正するには、Violation Detector を `--correct` コマンドラインオプションで使用します。修正モードでは、Violation Detector は `correctionInputPath` パラメータで指定された入力ファイルを読み込みます。このファイルの形式は `detectionOutputPath` ファイルの形式と同じなので、検出からの出力を修正のための入力として使用できます。

Violation Detector には、インデックスキー違反を修正する 2 種類の方法があります。

- 違反の削除 – 属性値に違反しているテーブル項目を削除します。
- 違反の更新 – テーブル項目を更新し、違反している属性を違反しない値に置き換えます。

いずれの場合も、検出モードからの出力ファイルを修正モードの入力として使用できます。

ProductCatalog の例に進みます。違反している項目をテーブルから削除するとします。これを行うには、次のコマンドラインを使用します。

```
$  java -jar ViolationDetector.jar --configFilePath config.txt --correct delete
```

この時点で、違反している項目を削除するかどうかを確認するメッセージが表示されます。

```
Are you sure to delete all violations on the table?y/n
y
Confirmed, will delete violations on the table...
Violation correction from file started: Reading records from file: ./
gsi_violation_check.csv, will delete these records from table.
Violation correction from file finished: Violations delete: 1, Violations Update: 0
```

ProductCatalog と PriceIndex の項目の数が同じになりました。

グローバルセカンダリインデックスの操作: Java

AWS SDK for Java ドキュメント API を使用して、1 つ以上のグローバルセカンダリインデックスを持つ Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。

以下に、テーブルオペレーションの一般的な手順を示します。

1. DynamoDB クラスのインスタンスを作成します。
2. 対応するリクエストオブジェクトを作成して、オペレーションについて必要なパラメータとオプションパラメータを入力します。
3. 前のステップで作成したクライアントが提供する適切なメソッドを呼び出します。

トピック

- グローバルセカンダリインデックスを持つテーブルを作成します
- グローバルセカンダリインデックスを持つテーブルの説明
- グローバルセカンダリインデックスのクエリ
- 例: AWS SDK for Java ドキュメント API を使用したグローバルセカンダリインデックス

グローバルセカンダリインデックスを持つテーブルを作成します

グローバルセカンダリインデックスは、テーブルの作成と同時に作成できます。これを行うには、CreateTable を使用し、1 つ以上のグローバルセカンダリインデックスの仕様を指定します。次の Java コード例では、気象データに関する情報を保持するテーブルを作成しています。パーティションキーは Location で、ソートキーは Date です。グローバルセカンダリインデックス PrecipIndex は、さまざまな場所の降水データへの高速アクセスを可能にします。

DynamoDB ドキュメント API を使用して、グローバルセカンダリインデックスを持つテーブルを作成する手順を次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. リクエスト情報を指定する CreateTableRequest クラスのインスタンスを作成します。

   テーブル名、プライマリキー、およびプロビジョニングされたスループット値を指定する必要があります。グローバルセカンダリインデックスの場合は、インデックス名、プロビジョニングされたスループット設定、インデックスソートキーの属性定義、インデックスのキースキーマ、および属性射影を指定する必要があります。
3. リクエストオブジェクトをパラメータとして指定して、createTable メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。このコードは、グローバルセカンダリインデックス (PrecipIndex) を持つテーブル (WeatherData) を作成します。インデックスパーティションキーは Date で、ソートキーは Precipitation です。すべてのテーブル属性がインデックスに射影されます。ユーザーは、このインデックスに対するクエリを実行して、特定の日付の気象データを取得できます。必要に応じて、降水量によってデータを並べ替えることもできます。

Precipitation はテーブルのキー属性ではないため、必須ではありません。ただし、Precipitation のない WeatherData 項目は PrecipIndex に表示されません。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

// Attribute definitions
ArrayList<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();

attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Location")
```

```
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Date")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Precipitation")
    .withAttributeType("N"));

// Table key schema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Location")
    .withKeyType(KeyType.HASH));  //Partition key
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.RANGE));  //Sort key

// PrecipIndex
GlobalSecondaryIndex precipIndex = new GlobalSecondaryIndex()
    .withIndexName("PrecipIndex")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 10)
        .withWriteCapacityUnits((long) 1))
        .withProjection(new Projection().withProjectionType(ProjectionType.ALL));

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();

indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.HASH));  //Partition key
indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Precipitation")
    .withKeyType(KeyType.RANGE));  //Sort key

precipIndex.setKeySchema(indexKeySchema);

CreateTableRequest createTableRequest = new CreateTableRequest()
    .withTableName("WeatherData")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 5)
        .withWriteCapacityUnits((long) 1))
    .withAttributeDefinitions(attributeDefinitions)
    .withKeySchema(tableKeySchema)
    .withGlobalSecondaryIndexes(precipIndex);
```

```
Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。

グローバルセカンダリインデックスを持つテーブルの説明

テーブルでグローバルセカンダリインデックスに関する情報を取得するには、DescribeTable を使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

テーブルのグローバルセカンダリインデックス情報にアクセスする手順を次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. 操作対象のインデックスを表すために、Table クラスのインスタンスを作成します。
3. Table オブジェクトの describe メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
TableDescription tableDesc = table.describe();


Iterator<GlobalSecondaryIndexDescription> gsiIter =
 tableDesc.getGlobalSecondaryIndexes().iterator();
while (gsiIter.hasNext()) {
    GlobalSecondaryIndexDescription gsiDesc = gsiIter.next();
    System.out.println("Info for index "
         + gsiDesc.getIndexName() + ":");

    Iterator<KeySchemaElement> kseIter = gsiDesc.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
```

```
    Projection projection = gsiDesc.getProjection();
    System.out.println("\tThe projection type is: "
        + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: "
            + projection.getNonKeyAttributes());
    }
}
```

グローバルセカンダリインデックスのクエリ

テーブルに Query を実行するのとほぼ同じ方法で、グローバルセカンダリインデックスに対する Query を使用することができます。インデックス名、インデックスパーティションキーとソートキー (存在する場合) のクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは PrecipIndex で、パーティションキーが Date で、ソートキーが Precipitation です。このインデックスクエリは、降水量がゼロより大きい特定の日付のすべての気象データを返します。

AWS SDK for Java ドキュメント API を使用してグローバルセカンダリインデックスをクエリする手順を次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. 操作対象のインデックスを表すために、Table クラスのインスタンスを作成します。
3. クエリするインデックスの Index クラスのインスタンスを作成します。
4. Index オブジェクトの query メソッドを呼び出します。

属性名 Date は DynamoDB の予約語です。したがって、式の属性名を KeyConditionExpression のプレースホルダーとして使用する必要があります。

以下の Java コード例は、前述のステップの例です。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
Index index = table.getIndex("PrecipIndex");

QuerySpec spec = new QuerySpec()
```

```
    .withKeyConditionExpression("#d = :v_date and Precipitation = :v_precip")
    .withNameMap(new NameMap()
        .with("#d", "Date"))
    .withValueMap(new ValueMap()
        .withString(":v_date","2013-08-10")
        .withNumber(":v_precip",0));

ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next().toJSONPretty());
}
```

例: AWS SDK for Java ドキュメント API を使用したグローバルセカンダリインデックス

次の Java コード例は、グローバルセカンダリインデックスの操作方法を示しています。この例では、Issues というテーブルを作成しています。これは、ソフトウェア開発のためのシンプルなバグ追跡システムで使用できる可能性があります。パーティションキーは IssueId で、ソートキーは Title です。このテーブルには、次の 3 つのグローバルセカンダリインデックスがあります。

- CreateDateIndex – パーティションキーは CreateDate で、ソートキーは IssueId です。テーブルキーに加えて、属性 Description および Status はインデックスに射影されます。
- TitleIndex – パーティションキーは Title で、ソートキーは IssueId です。テーブルキー以外の属性はインデックスに射影されません。
- DueDateIndex – パーティションキーは DueDate で、ソートキーはありません。すべてのテーブル属性がインデックスに射影されます。

Issues テーブルが作成されると、プログラムはソフトウェアのバグレポートを表すデータをテーブルにロードします。次に、グローバルセカンダリインデックスを使用してデータのクエリを実行します。最後に、プログラムは Issues テーブルを削除します。

以下の例をテストするための詳細な手順については、「Java コードの例」を参照してください。

Example

```
package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIGlobalSecondaryIndexExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    public static String tableName = "Issues";

    public static void main(String[] args) throws Exception {

        createTable();
        loadData();

        queryIndex("CreateDateIndex");
        queryIndex("TitleIndex");
        queryIndex("DueDateIndex");

        deleteTable(tableName);

    }

    public static void createTable() {

        // Attribute definitions
        ArrayList<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();
```

```
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("IssueId").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Title").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("CreateDate").withAttributeType("S"));
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("DueDate").withAttributeType("S"));

        // Key schema for table
        ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.HASH)); //
Partition

                    // key
        tableKeySchema.add(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.RANGE)); // Sort

                   // key

        // Initial provisioned throughput settings for the indexes
        ProvisionedThroughput ptIndex = new
ProvisionedThroughput().withReadCapacityUnits(1L)
                .withWriteCapacityUnits(1L);

        // CreateDateIndex
        GlobalSecondaryIndex createDateIndex = new
GlobalSecondaryIndex().withIndexName("CreateDateIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("CreateDate").withKeyType(KeyType.HASH), //
Partition

                          // key
                        new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

                 // key
                .withProjection(
                        new
Projection().withProjectionType("INCLUDE").withNonKeyAttributes("Description",
"Status"));
```

```
        // TitleIndex
        GlobalSecondaryIndex titleIndex = new
GlobalSecondaryIndex().withIndexName("TitleIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.HASH), // Partition

                     // key
                        new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

                 // key
                .withProjection(new Projection().withProjectionType("KEYS_ONLY"));

        // DueDateIndex
        GlobalSecondaryIndex dueDateIndex = new
GlobalSecondaryIndex().withIndexName("DueDateIndex")
                .withProvisionedThroughput(ptIndex)
                .withKeySchema(new
KeySchemaElement().withAttributeName("DueDate").withKeyType(KeyType.HASH)) //
Partition

                       // key
                .withProjection(new Projection().withProjectionType("ALL"));

        CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
                .withProvisionedThroughput(
                        new ProvisionedThroughput().withReadCapacityUnits((long)
1).withWriteCapacityUnits((long) 1))

.withAttributeDefinitions(attributeDefinitions).withKeySchema(tableKeySchema)
                .withGlobalSecondaryIndexes(createDateIndex, titleIndex, dueDateIndex);

        System.out.println("Creating table " + tableName + "...");
        dynamoDB.createTable(createTableRequest);

        // Wait for table to become active
        System.out.println("Waiting for " + tableName + " to become ACTIVE...");
        try {
            Table table = dynamoDB.getTable(tableName);
            table.waitForActive();
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }

    public static void queryIndex(String indexName) {

        Table table = dynamoDB.getTable(tableName);


System.out.println("\n***********************************************************\n");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        QuerySpec querySpec = new QuerySpec();

        if ("CreateDateIndex".equals(indexName)) {
            System.out.println("Issues filed on 2013-11-01");
            querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
                    .withValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else if ("TitleIndex".equals(indexName)) {
            System.out.println("Compilation errors");
            querySpec.withKeyConditionExpression("Title = :v_title and
begins_with(IssueId, :v_issue)")
                    .withValueMap(
                            new ValueMap().withString(":v_title", "Compilation
error").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else if ("DueDateIndex".equals(indexName)) {
            System.out.println("Items that are due on 2013-11-30");
            querySpec.withKeyConditionExpression("DueDate = :v_date")
                    .withValueMap(new ValueMap().withString(":v_date", "2013-11-30"));
            items = index.query(querySpec);
        } else {
            System.out.println("\nNo valid index name provided");
            return;
        }

        Iterator<Item> iterator = items.iterator();
```

```
        System.out.println("Query: printing results...");

        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }

    }

    public static void deleteTable(String tableName) {

        System.out.println("Deleting table " + tableName + "...");

        Table table = dynamoDB.getTable(tableName);
        table.delete();

        // Wait for table to be deleted
        System.out.println("Waiting for " + tableName + " to be deleted...");
        try {
            table.waitForDelete();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void loadData() {

        System.out.println("Loading data into table " + tableName + "...");

        // IssueId, Title,
        // Description,
        // CreateDate, LastUpdateDate, DueDate,
        // Priority, Status

        putItem("A-101", "Compilation error", "Can't compile Project X - bad version
number. What does this mean?",
                "2013-11-01", "2013-11-02", "2013-11-10", 1, "Assigned");

        putItem("A-102", "Can't read data file", "The main data file is missing, or the
permissions are incorrect",
                "2013-11-01", "2013-11-04", "2013-11-30", 2, "In progress");

        putItem("A-103", "Test failure", "Functional test of Project X produces
errors", "2013-11-01", "2013-11-02",
                "2013-11-10", 1, "In progress");
```

```
        putItem("A-104", "Compilation error", "Variable 'messageCount' was not
 initialized.", "2013-11-15",
                "2013-11-16", "2013-11-30", 3, "Assigned");

        putItem("A-105", "Network issue", "Can't ping IP address 127.0.0.1. Please fix
 this.", "2013-11-15",
                "2013-11-16", "2013-11-19", 5, "Assigned");

    }

    public static void putItem(

            String issueId, String title, String description, String createDate, String
 lastUpdateDate, String dueDate,
            Integer priority, String status) {

        Table table = dynamoDB.getTable(tableName);

        Item item = new Item().withPrimaryKey("IssueId", issueId).withString("Title",
 title)
                .withString("Description", description).withString("CreateDate",
 createDate)
                .withString("LastUpdateDate", lastUpdateDate).withString("DueDate",
 dueDate)
                .withNumber("Priority", priority).withString("Status", status);

        table.putItem(item);
    }

}
```

グローバルセカンダリインデックスの操作: .NET

AWS SDK for .NET 低レベル API を使用して、1 つ以上のグローバルセカンダリインデックスを含む Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。これらのオペレーションは、対応する DynamoDB オペレーションにマッピングされます。詳細については、「Amazon DynamoDB API リファレンス」を参照してください。

以下に、.NET 低レベル API を使用したテーブルオペレーションの一般的な手順を示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. 対応するリクエストオブジェクトを作成して、オペレーションについて必要なパラメータとオプションパラメータを入力します。

   たとえば、テーブルを作成するには CreateTableRequest オブジェクトを作成し、テーブルやインデックスにクエリを実行するには QueryRequest オブジェクトを作成します。
3. 前述のステップで作成したクライアントから提供された適切なメソッドを実行します。

トピック

- グローバルセカンダリインデックスを持つテーブルを作成します
- グローバルセカンダリインデックスを持つテーブルの説明
- グローバルセカンダリインデックスのクエリ
- 例: AWS SDK for .NET 低レベル API を使用したグローバルセカンダリインデックス

グローバルセカンダリインデックスを持つテーブルを作成します

グローバルセカンダリインデックスは、テーブルの作成と同時に作成できます。これを行うには、CreateTable を使用し、1 つ以上のグローバルセカンダリインデックスの仕様を指定します。次の C# コード例では、気象データに関する情報を保持するテーブルを作成しています。パーティションキーは Location で、ソートキーは Date です。グローバルセカンダリインデックス PrecipIndex は、さまざまな場所の降水データへの高速アクセスを可能にします。

.NET 低レベル API を使用して、グローバルセカンダリインデックスを含むテーブルを作成する手順を次に示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエスト情報を指定する CreateTableRequest クラスのインスタンスを作成します。

   テーブル名、プライマリキー、およびプロビジョニングされたスループット値を指定する必要があります。グローバルセカンダリインデックスの場合は、インデックス名、プロビジョニングされたスループット設定、インデックスソートキーの属性定義、インデックスのキースキーマ、および属性射影を指定する必要があります。
3. リクエストオブジェクトをパラメータとして指定して、CreateTable メソッドを実行します。

以下の C# サンプルコードは、前述のステップの例です。このコードは、グローバルセカンダリインデックス (PrecipIndex) を持つテーブル (WeatherData) を作成します。インデックスパーティ

ションキーは Date で、ソートキーは Precipitation です。すべてのテーブル属性がインデックスに射影されます。ユーザーは、このインデックスに対するクエリを実行して、特定の日付の気象データを取得できます。必要に応じて、降水量によってデータを並べ替えることもできます。

Precipitation はテーブルのキー属性ではないため、必須ではありません。ただし、Precipitation のない WeatherData 項目は PrecipIndex に表示されません。

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

// Attribute definitions
var attributeDefinitions = new List<AttributeDefinition>()
{
    {new AttributeDefinition{
        AttributeName = "Location",
        AttributeType = "S"}},
    {new AttributeDefinition{
        AttributeName = "Date",
        AttributeType = "S"}},
    {new AttributeDefinition(){
        AttributeName = "Precipitation",
        AttributeType = "N"}
    }
};

// Table key schema
var tableKeySchema = new List<KeySchemaElement>()
{
    {new KeySchemaElement {
        AttributeName = "Location",
        KeyType = "HASH"}},  //Partition key
    {new KeySchemaElement {
        AttributeName = "Date",
        KeyType = "RANGE"}  //Sort key
    }
};

// PrecipIndex
var precipIndex = new GlobalSecondaryIndex
{
    IndexName = "PrecipIndex",
    ProvisionedThroughput = new ProvisionedThroughput
    {
```

```
        ReadCapacityUnits = (long)10,
        WriteCapacityUnits = (long)1
    },
    Projection = new Projection { ProjectionType = "ALL" }
};

var indexKeySchema = new List<KeySchemaElement> {
    {new KeySchemaElement { AttributeName = "Date", KeyType = "HASH"}},  //Partition
 key
    {new KeySchemaElement{AttributeName = "Precipitation",KeyType = "RANGE"}}  //Sort
 key
};

precipIndex.KeySchema = indexKeySchema;

CreateTableRequest createTableRequest = new CreateTableRequest
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)5,
        WriteCapacityUnits = (long)1
    },
    AttributeDefinitions = attributeDefinitions,
    KeySchema = tableKeySchema,
    GlobalSecondaryIndexes = { precipIndex }
};

CreateTableResponse response = client.CreateTable(createTableRequest);
Console.WriteLine(response.CreateTableResult.TableDescription.TableName);
Console.WriteLine(response.CreateTableResult.TableDescription.TableStatus);
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。

グローバルセカンダリインデックスを持つテーブルの説明

テーブルでグローバルセカンダリインデックスに関する情報を取得するには、DescribeTable を使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

.NET 低レベル API を使用してテーブルのグローバルセカンダリインデックス情報にアクセスするには次の手順に従います。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエストオブジェクトをパラメータとして指定して、describeTable メソッドを実行します。

   リクエスト情報を指定する DescribeTableRequest クラスのインスタンスを作成します。テーブル名を入力する必要があります。

3.

以下の C# サンプルコードは、前述のステップの例です。

Example

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest
 { TableName = tableName});

List<GlobalSecondaryIndexDescription> globalSecondaryIndexes =
response.DescribeTableResult.Table.GlobalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

foreach (GlobalSecondaryIndexDescription gsiDescription in globalSecondaryIndexes) {
     Console.WriteLine("Info for index " + gsiDescription.IndexName + ":");

     foreach (KeySchemaElement kse in gsiDescription.KeySchema) {
          Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
     }

      Projection projection = gsiDescription.Projection;
      Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

      if (projection.ProjectionType.ToString().Equals("INCLUDE")) {
           Console.WriteLine("\t\tThe non-key projected attributes are: "
                + projection.NonKeyAttributes);
      }
}
```

グローバルセカンダリインデックスのクエリ

テーブルに Query を実行するのとほぼ同じ方法で、グローバルセカンダリインデックスに対する Query を使用することができます。インデックス名、インデックスパーティションキーとソートキー (存在する場合) のクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは PrecipIndex で、パーティションキーが Date で、ソートキーが Precipitation です。このインデックスクエリは、降水量がゼロより大きい特定の日付のすべての気象データを返します。

.NET 低レベル API を使用して、グローバルセカンダリインデックスを含むテーブルにクエリを実行する手順を次に示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエスト情報を指定する QueryRequest クラスのインスタンスを作成します。
3. リクエストオブジェクトをパラメータとして指定して、query メソッドを実行します。

属性名 Date は DynamoDB の予約語です。したがって、式の属性名を KeyConditionExpression のプレースホルダーとして使用する必要があります。

以下の C# サンプルコードは、前述のステップの例です。

Example

```
client = new AmazonDynamoDBClient();

QueryRequest queryRequest = new QueryRequest
{
    TableName = "WeatherData",
    IndexName = "PrecipIndex",
    KeyConditionExpression = "#dt = :v_date and Precipitation > :v_precip",
    ExpressionAttributeNames = new Dictionary<String, String> {
        {"#dt", "Date"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {":v_date", new AttributeValue { S =  "2013-08-01" }},
        {":v_precip", new AttributeValue { N =  "0" }}
    },
    ScanIndexForward = true
};
```

```
var result = client.Query(queryRequest);

var items = result.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        Console.Write(attr + "---> ");
        if (attr == "Precipitation")
        {
            Console.WriteLine(currentItem[attr].N);
    }
    else
    {
        Console.WriteLine(currentItem[attr].S);
    }

         }
     Console.WriteLine();
}
```

例: AWS SDK for .NET 低レベル API を使用したグローバルセカンダリインデックス

次の C# コード例は、グローバルセカンダリインデックスの操作方法を示しています。この例では、Issues というテーブルを作成しています。これは、ソフトウェア開発のためのシンプルなバグ追跡システムで使用できる可能性があります。パーティションキーは IssueId で、ソートキーは Title です。このテーブルには、次の 3 つのグローバルセカンダリインデックスがあります。

- CreateDateIndex – パーティションキーは CreateDate で、ソートキーは IssueId です。テーブルキーに加えて、属性 Description および Status はインデックスに射影されます。
- TitleIndex – パーティションキーは Title で、ソートキーは IssueId です。テーブルキー以外の属性はインデックスに射影されません。
- DueDateIndex – パーティションキーは DueDate で、ソートキーはありません。すべてのテーブル属性がインデックスに射影されます。

Issues テーブルが作成されると、プログラムはソフトウェアのバグレポートを表すデータをテーブルにロードします。次に、グローバルセカンダリインデックスを使用してデータのクエリを実行します。最後に、プログラムは Issues テーブルを削除します。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelGlobalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        public static String tableName = "Issues";

        public static void Main(string[] args)
        {
            CreateTable();
            LoadData();

            QueryIndex("CreateDateIndex");
            QueryIndex("TitleIndex");
            QueryIndex("DueDateIndex");

            DeleteTable(tableName);

            Console.WriteLine("To continue, press enter");
            Console.Read();
        }

        private static void CreateTable()
        {
            // Attribute definitions
            var attributeDefinitions = new List<AttributeDefinition>()
        {
            {new AttributeDefinition {
                 AttributeName = "IssueId", AttributeType = "S"
             }},
            {new AttributeDefinition {
                 AttributeName = "Title", AttributeType = "S"
```

```
             }},
            {new AttributeDefinition {
                 AttributeName = "CreateDate", AttributeType = "S"
             }},
            {new AttributeDefinition {
                 AttributeName = "DueDate", AttributeType = "S"
             }}
        };

            // Key schema for table
            var tableKeySchema = new List<KeySchemaElement>() {
            {
                new KeySchemaElement {
                    AttributeName= "IssueId",
                    KeyType = "HASH" //Partition key
                }
            },
            {
                new KeySchemaElement {
                    AttributeName = "Title",
                    KeyType = "RANGE" //Sort key
                }
            }
        };

            // Initial provisioned throughput settings for the indexes
            var ptIndex = new ProvisionedThroughput
            {
                ReadCapacityUnits = 1L,
                WriteCapacityUnits = 1L
            };

            // CreateDateIndex
            var createDateIndex = new GlobalSecondaryIndex()
            {
                IndexName = "CreateDateIndex",
                ProvisionedThroughput = ptIndex,
                KeySchema = {
                new KeySchemaElement {
                    AttributeName = "CreateDate", KeyType = "HASH" //Partition key
                },
                new KeySchemaElement {
                    AttributeName = "IssueId", KeyType = "RANGE" //Sort key
                }
```

```
            },
                Projection = new Projection
                {
                    ProjectionType = "INCLUDE",
                    NonKeyAttributes = {
                    "Description", "Status"
                }
                }
            };

            // TitleIndex
            var titleIndex = new GlobalSecondaryIndex()
            {
                IndexName = "TitleIndex",
                ProvisionedThroughput = ptIndex,
                KeySchema = {
                new KeySchemaElement {
                    AttributeName = "Title", KeyType = "HASH" //Partition key
                },
                new KeySchemaElement {
                    AttributeName = "IssueId", KeyType = "RANGE" //Sort key
                }
            },
                Projection = new Projection
                {
                    ProjectionType = "KEYS_ONLY"
                }
            };

            // DueDateIndex
            var dueDateIndex = new GlobalSecondaryIndex()
            {
                IndexName = "DueDateIndex",
                ProvisionedThroughput = ptIndex,
                KeySchema = {
                new KeySchemaElement {
                    AttributeName = "DueDate",
                    KeyType = "HASH" //Partition key
                }
            },
                Projection = new Projection
                {
                    ProjectionType = "ALL"
                }
```

```
            };



            var createTableRequest = new CreateTableRequest
            {
                TableName = tableName,
                ProvisionedThroughput = new ProvisionedThroughput
                {
                    ReadCapacityUnits = (long)1,
                    WriteCapacityUnits = (long)1
                },
                AttributeDefinitions = attributeDefinitions,
                KeySchema = tableKeySchema,
                GlobalSecondaryIndexes = {
                createDateIndex, titleIndex, dueDateIndex
            }
            };

            Console.WriteLine("Creating table " + tableName + "...");
            client.CreateTable(createTableRequest);

            WaitUntilTableReady(tableName);
        }

        private static void LoadData()
        {
            Console.WriteLine("Loading data into table " + tableName + "...");

            // IssueId, Title,
            // Description,
            // CreateDate, LastUpdateDate, DueDate,
            // Priority, Status

            putItem("A-101", "Compilation error",
                "Can't compile Project X - bad version number. What does this mean?",
                "2013-11-01", "2013-11-02", "2013-11-10",
                1, "Assigned");

            putItem("A-102", "Can't read data file",
                "The main data file is missing, or the permissions are incorrect",
                "2013-11-01", "2013-11-04", "2013-11-30",
                2, "In progress");
```

```
            putItem("A-103", "Test failure",
                "Functional test of Project X produces errors",
                "2013-11-01", "2013-11-02", "2013-11-10",
                1, "In progress");

            putItem("A-104", "Compilation error",
                "Variable 'messageCount' was not initialized.",
                "2013-11-15", "2013-11-16", "2013-11-30",
                3, "Assigned");

            putItem("A-105", "Network issue",
                "Can't ping IP address 127.0.0.1. Please fix this.",
                "2013-11-15", "2013-11-16", "2013-11-19",
                5, "Assigned");
        }

        private static void putItem(
            String issueId, String title,
            String description,
            String createDate, String lastUpdateDate, String dueDate,
            Int32 priority, String status)
        {
            Dictionary<String, AttributeValue> item = new Dictionary<string,
AttributeValue>();

            item.Add("IssueId", new AttributeValue
            {
                S = issueId
            });
            item.Add("Title", new AttributeValue
            {
                S = title
            });
            item.Add("Description", new AttributeValue
            {
                S = description
            });
            item.Add("CreateDate", new AttributeValue
            {
                S = createDate
            });
            item.Add("LastUpdateDate", new AttributeValue
            {
                S = lastUpdateDate
```

```
            });
            item.Add("DueDate", new AttributeValue
            {
                S = dueDate
            });
            item.Add("Priority", new AttributeValue
            {
                N = priority.ToString()
            });
            item.Add("Status", new AttributeValue
            {
                S = status
            });

            try
            {
                client.PutItem(new PutItemRequest
                {
                    TableName = tableName,
                    Item = item
                });
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }

        private static void QueryIndex(string indexName)
        {
            Console.WriteLine
                ("\n*****************************************************\n");
            Console.WriteLine("Querying index " + indexName + "...");

            QueryRequest queryRequest = new QueryRequest
            {
                TableName = tableName,
                IndexName = indexName,
                ScanIndexForward = true
            };


            String keyConditionExpression;
```

```
            Dictionary<string, AttributeValue> expressionAttributeValues = new
Dictionary<string, AttributeValue>();

            if (indexName == "CreateDateIndex")
            {
                Console.WriteLine("Issues filed on 2013-11-01\n");

                keyConditionExpression = "CreateDate = :v_date and
begins_with(IssueId, :v_issue)";
                expressionAttributeValues.Add(":v_date", new AttributeValue
                {
                    S = "2013-11-01"
                });
                expressionAttributeValues.Add(":v_issue", new AttributeValue
                {
                    S = "A-"
                });
            }
            else if (indexName == "TitleIndex")
            {
                Console.WriteLine("Compilation errors\n");

                keyConditionExpression = "Title = :v_title and
begins_with(IssueId, :v_issue)";
                expressionAttributeValues.Add(":v_title", new AttributeValue
                {
                    S = "Compilation error"
                });
                expressionAttributeValues.Add(":v_issue", new AttributeValue
                {
                    S = "A-"
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else if (indexName == "DueDateIndex")
            {
                Console.WriteLine("Items that are due on 2013-11-30\n");

                keyConditionExpression = "DueDate = :v_date";
                expressionAttributeValues.Add(":v_date", new AttributeValue
                {
                    S = "2013-11-30"
```

```
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else
            {
                Console.WriteLine("\nNo valid index name provided");
                return;
            }

            queryRequest.KeyConditionExpression = keyConditionExpression;
            queryRequest.ExpressionAttributeValues = expressionAttributeValues;

            var result = client.Query(queryRequest);
            var items = result.Items;
            foreach (var currentItem in items)
            {
                foreach (string attr in currentItem.Keys)
                {
                    if (attr == "Priority")
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].N);
                    }
                    else
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].S);
                    }
                }
                Console.WriteLine();
            }
        }

        private static void DeleteTable(string tableName)
        {
            Console.WriteLine("Deleting table " + tableName + "...");
            client.DeleteTable(new DeleteTableRequest
            {
                TableName = tableName
            });
            WaitForTableToBeDeleted(tableName);
        }

        private static void WaitUntilTableReady(string tableName)
```

```
        {
            string status = null;
            // Let us wait until table is created. Call DescribeTable.
            do
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });

                    Console.WriteLine("Table name: {0}, status: {1}",
                              res.Table.TableName,
                              res.Table.TableStatus);
                    status = res.Table.TableStatus;
                }
                catch (ResourceNotFoundException)
                {
                    // DescribeTable is eventually consistent. So you might
                    // get resource not found. So we handle the potential exception.
                }
            } while (status != "ACTIVE");
        }

        private static void WaitForTableToBeDeleted(string tableName)
        {
            bool tablePresent = true;

            while (tablePresent)
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });

                    Console.WriteLine("Table name: {0}, status: {1}",
                              res.Table.TableName,
                              res.Table.TableStatus);
                }
```

```
                catch (ResourceNotFoundException)
                {
                    tablePresent = false;
                }
            }
        }
    }
}
```

AWS CLI を使用した DynamoDBでのグローバルセカンダリインデックスの操作

AWS CLI を使用して、1 つ以上のグローバルセカンダリインデックスを含む Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。

トピック

- グローバルセカンダリインデックスを持つテーブルを作成します
- グローバルセカンダリインデックスを既存のテーブルに追加します
- グローバルセカンダリインデックスを持つテーブルの説明
- グローバルセカンダリインデックスのクエリ

グローバルセカンダリインデックスを持つテーブルを作成します

グローバルセカンダリインデックスは、テーブルの作成と同時に作成できます。　これを行うには、create-table パラメータを使用し、1 つ以上のグローバルセカンダリインデックスの仕様を指定します。次の例では、GameTitleIndex と呼ばれるグローバルセカンダリインデックスを含む GameScores という名前のテーブルを作成します。ベーステーブルには、パーティションキー UserId とソートキー GameTitle があり、特定のゲームの個々のユーザーのベストスコアを効率的に見つけることができます。一方、GSI にはパーティションキー GameTitle とソートキー TopScore があり、特定のゲームの全体的な最高スコアをすばやく見つけることができます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=UserId,AttributeType=S \
                            AttributeName=GameTitle,AttributeType=S \
                            AttributeName=TopScore,AttributeType=N  \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
```

```
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                                {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
                \"Projection\":{
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。describe-table を使用して、テーブル作成のステータスを判断できます。

グローバルセカンダリインデックスを既存のテーブルに追加します

グローバルセカンダリインデックスは、テーブルの作成後に追加したり変更したりすることもできます。これを行うには、update-table パラメータを使用し、1 つ以上のグローバルセカンダリインデックスの仕様を指定します。次の例では、前の例と同じスキーマを使用していますが、テーブルが既に作成されており、後で GSI を追加する場合を想定しています。

```
aws dynamodb update-table \
    --table-name GameScores \
    --attribute-definitions AttributeName=TopScore,AttributeType=N  \
    --global-secondary-index-updates \
        "[
            {
                \"Create\": {
                    \"IndexName\": \"GameTitleIndex\",
                    \"KeySchema\": [{\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH
\"},
                                    {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE
\"}],
                    \"Projection\":{
```

```
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                }
            }
        }
    ]"
```

グローバルセカンダリインデックスを持つテーブルの説明

テーブルのグローバルセカンダリインデックスに関する情報を取得するには、describe-table パラメータを使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

```
aws dynamodb describe-table --table-name GameScores
```

グローバルセカンダリインデックスのクエリ

テーブルに query を実行するのとほぼ同じ方法で、グローバルセカンダリインデックスに対する query オペレーションを使用することができます。インデックス名、インデックスソートキーのクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは GameTitleIndex、インデックスソートキーは GameTitle です。

返される属性は、インデックスに射影された属性だけです。このクエリを変更して非キー属性を選択することもできますが、これには比較的コストのかかるテーブルフェッチアクティビティが必要です。テーブルのフェッチの詳細については、「属性の射影」を参照してください。

```
aws dynamodb query --table-name GameScores\
    --index-name GameTitleIndex \
    --key-condition-expression "GameTitle = :v_game" \
    --expression-attribute-values '{":v_game":{"S":"Alien Adventure"} }'
```

ローカルセカンダリインデックス

ベーステーブルのプライマリキーを使用してデータのクエリを実行する必要があるのは、一部のアプリケーションだけです。ただし、代替のソートキーが役に立つ場合があります。アプリケーションでソートキーを選択できるようにするには、Amazon DynamoDB テーブルに 1 つ以上のローカルセカンダリインデックスを作成し、これらのインデックスに対して Query または Scan のリクエストを発行できます。

トピック

- シナリオ: ローカルセカンダリインデックスの使用
- 属性の射影
- ローカルセカンダリインデックスの作成
- ローカルセカンダリインデックスからのデータの読み込み
- 項目の書き込みとローカルセカンダリインデックス
- ローカルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項
- ローカルセカンダリインデックスのストレージに関する考慮事項
- ローカルセカンダリインデックス内の項目コレクション
- ローカルセカンダリインデックスの操作: Java
- ローカルセカンダリインデックスの操作: .NET
- DynamoDB AWS CLI でのローカルセカンダリインデックスの操作

シナリオ: ローカルセカンダリインデックスの使用

例として、Thread テーブルを検討します。このテーブルは、AWS ディスカッションフォーラムのようなアプリケーションで役立ちます。次の図は、テーブル内の項目の構成を示しています。一部表示されていない属性もあります。

Thread

ForumName	Subject	LastPostDateTime	Replies	
"S3"	"aaa"	"2015-03-15:17:24:31"	12	...
"S3"	"bbb"	"2015-01-22:23:18:01"	3	...
"S3"	"ccc"	"2015-02-31:13:14:21"	4	...
"S3"	"ddd"	"2015-01-03:09:21:11"	9	...
"EC2"	"yyy"	"2015-02-12:11:07:56"	18	...
"EC2"	"zzz"	"2015-01-18:07:33:42"	0	...
"RDS"	"rrr"	"2015-01-19:01:13:24"	3	...
"RDS"	"sss"	"2015-03-11:06:53:00"	11	...
"RDS"	"ttt"	"2015-10-22:12:19:44"	5	...
...	...	...	...	

DynamoDB には、同じパーティションキー値を持つすべての項目が継続的に保存されます。この例では、特定の ForumName を指定すると、Query オペレーションはそのフォーラムのすべてのスレッドをすばやく見つけることができます。同じパーティションキー値を持つ項目のグループでは、項目がソートキーの値によって並べ替えられます。ソートキー (Subject) がクエリでも提供される

場合、DynamoDB は返される結果を絞り込むことができます。例えば、「S3」フォーラムで、文字「a」で始まる Subject を持つすべてのスレッドを返すことができます。

リクエストによっては、より複雑なデータアクセスパターンが要求される場合があります。以下に例を示します。

- ビューと返信の数が最も多いのはどのフォーラムか?
- 特定のフォーラムでメッセージ数が最も多いのはどのスレッドか?
- 特定の期間中に特定のフォーラムに投稿されたスレッド数は?

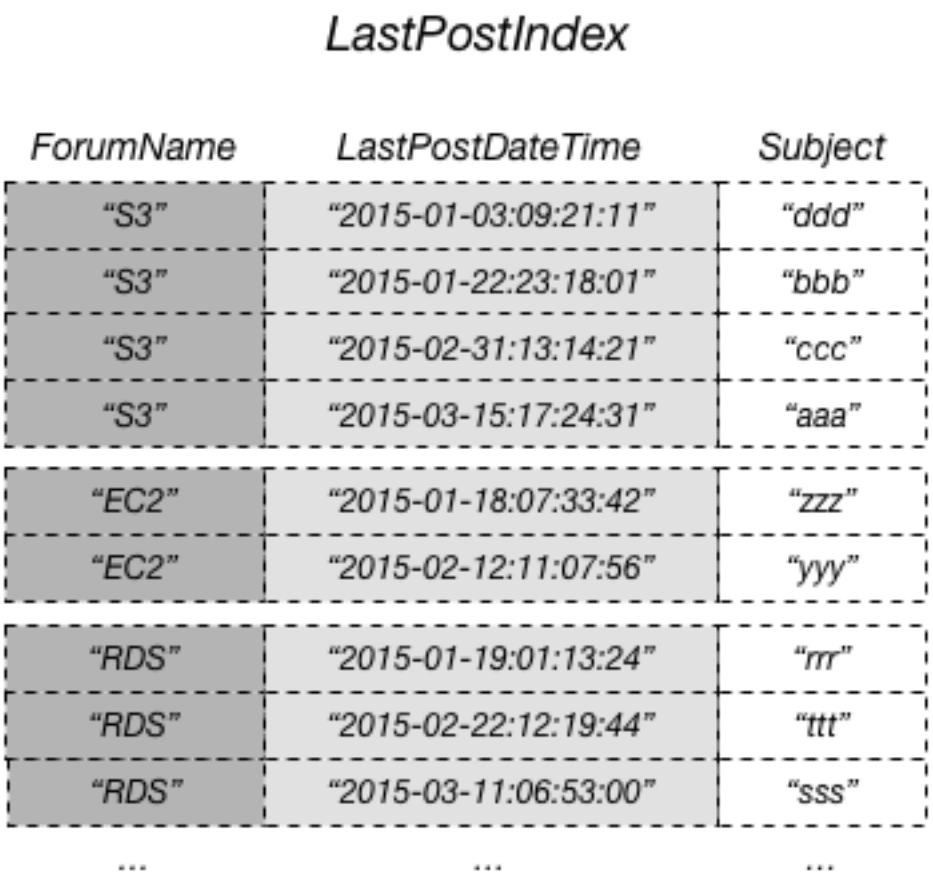
これらの質問に答えるには、Query アクションでは十分ではありません。代わりに、テーブル全体の Scan を実行する必要があります。膨大な数の項目があるテーブルでは、これにより、プロビジョニングされた読み込みスループットが大量に消費されることになり、完了するまでに長時間かかります。

ただし、Replies や LastPostDateTime などの非キー属性に 1 つ以上のローカルセカンダリインデックスを指定することができます。

ローカルセカンダリインデックスは特定のパーティションキー値の代替ソートキーを維持します。またローカルセカンダリインデックスには、ベーステーブルの一部またはすべての属性のコピーが含まれます。テーブルを作成する際に、ローカルセカンダリインデックスに射影する属性を指定します。ローカルセカンダリインデックスのデータは、ベーステーブルと同じパーティションキーと、異なるソートキーで構成されます。これにより、この異なるディメンションにわたってデータ項目に効率的にアクセスできます。クエリまたはスキャンの柔軟性を向上させるために、テーブルごとに最大 5 つのローカルセカンダリインデックスを作成できます。

たとえば、あるアプリケーションで、特定のフォーラムに過去 3 か月以内に投稿されたすべてのスレッドを検索する必要があるとします。ローカルセカンダリインデックスがない場合、アプリケーションは Scan テーブル全体の Thread を実行して、指定された時間枠から外れた投稿を破棄する必要があります。ローカルセカンダリインデックスがあれば、Query オペレーションでは LastPostDateTime をソートキーとして使用し、データをすばやく見つけることができます。

次の図は、LastPostIndex という名前のローカルセカンダリインデックスを示しています。パーティションキーは Thread テーブルのパーティションキーと同じですが、ソートキーは LastPostDateTime です。



ローカルセカンダリインデックスは、すべて次の条件を満たす必要があります。

- パーティションキーはそのベーステーブルのパーティションキーと同じである。
- ソートキーは完全に 1 つのスカラー属性で構成されている。
- ベーステーブルのソートキーがインデックスに射影され、非キー属性として機能する。

この例で、パーティションキーは ForumName、ローカルセカンダリインデックスのソートキーは LastPostDateTime です。さらに、ベーステーブルのソートキーの値 (この例では Subject) がインデックスに射影されますが、その値はインデックスキーの一部ではありません。アプリケーションが ForumName と LastPostDateTime に基づくリストを必要とする場合は、Query に対して LastPostIndex リクエストを実行できます。クエリ結果は LastPostDateTime によってソートされ、昇順または降順で返すことができます。このクエリは、特定の時間枠内に LastPostDateTime がある項目だけを返すなどのキー条件を適用することもできます。

すべてのローカルセカンダリインデックスには、そのベーステーブルからのパーティションキーとソートキーが自動的に格納されます。必要に応じて、非キー属性をインデックスに射影できます。インデックスのクエリを行うと、DynamoDB ではこれらの射影された属性を効率的に取り出すこと

ができます。ローカルセカンダリインデックスにクエリを実行すると、インデックスに射影されていない属性もクエリで取得できます。DynamoDB では、これらの属性をベーステーブルから自動的にフェッチしますが、レイテンシーが大きくなり、プロビジョニングされたスループットコストが高くなります。

任意のローカルセカンダリインデックスについて、異なるパーティションキーの値ごとに最大 10 GB のデータを保存できます。この数字には、ベーステーブル内のすべての項目に加えて、同じパーティションキーの値を持つインデックス内のすべての項目も含まれています。詳細については、「ローカルセカンダリインデックス内の項目コレクション」を参照してください。

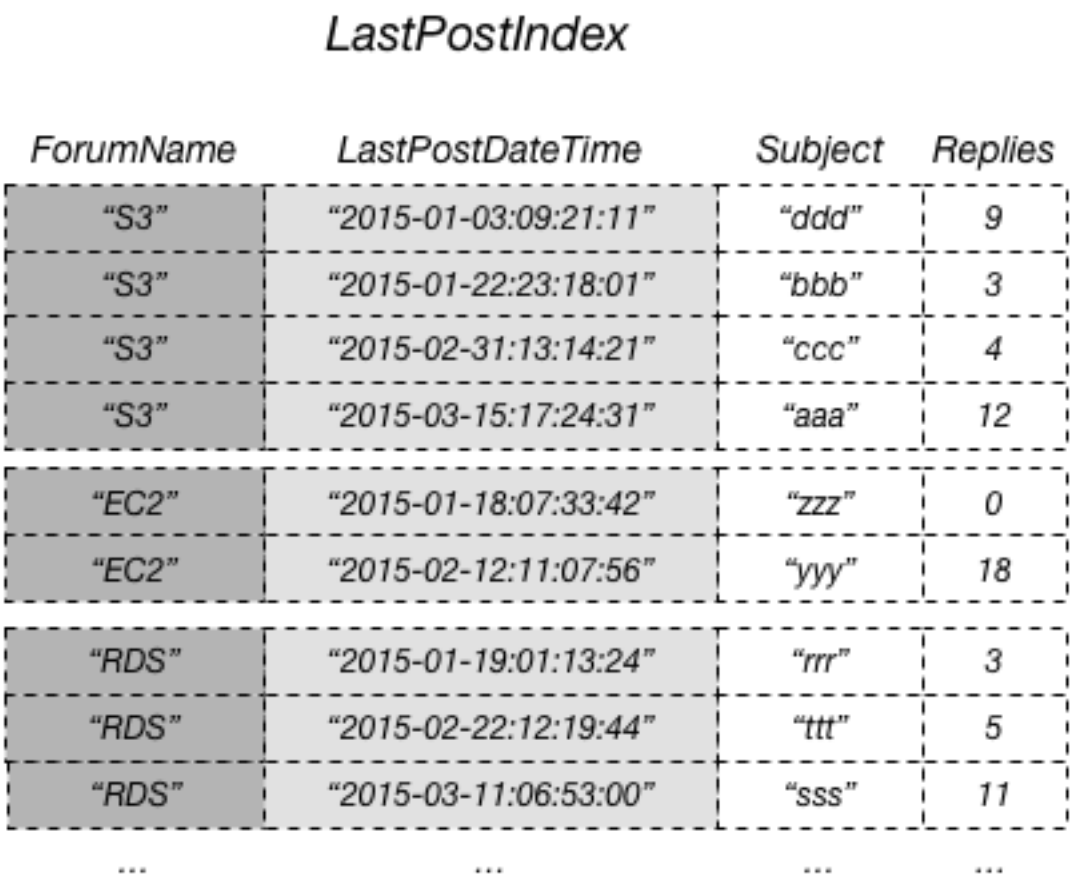
属性の射影

`LastPostIndex` では、アプリケーションはクエリの基準として `ForumName` と `LastPostDateTime` を使用できます。ただし、追加の属性を取得する場合、DynamoDB は `Thread` テーブルに対して追加の読み取りオペレーションを実行する必要があります。このような追加の読み込みをフェッチと言い、それによってクエリにプロビジョニングするスループットの合計量が増加することがあります。

ウェブページに「S3」内のすべてのスレッドと各スレッドの返信のリストを表示し、最新の返信の最後の返信日時で並べ替えるとします。このリストに入力するには、次の属性が必要になります。

- `Subject`
- `Replies`
- `LastPostDateTime`

このデータのクエリを最も効率的に行い、フェッチオペレーションを回避するには、次の図に示すように、ローカルセカンダリインデックスにテーブルからの `Replies` 属性を射影します。



射影とは、テーブルからセカンダリインデックスにコピーされる属性のセットです。テーブルのパーティションキーとソートキーは常にインデックスに射影されます。アプリケーションのクエリ要件をサポートするために、他の属性を射影できます。インデックスをクエリすると、Amazon DynamoDB は、その属性が自身のテーブル内にあるかのように、プロジェクション内の任意の属性にアクセスできます。

セカンダリインデックスを作成するときは、インデックスに射影される属性を指定する必要があります。DynamoDB には、そのために次の 3 つのオプションが用意されています。

- KEYS_ONLY – インデックス内の各項目は、テーブルパーティションキーとソートキーの値、およびインデックスキーの値のみで構成されます。KEYS_ONLY オプションを指定すると、セカンダリインデックスが最小になります。
- INCLUDE – KEYS_ONLY の属性に加えて、セカンダリインデックスにその他の非キー属性が含まれるように指定できます。

- ALL – セカンダリインデックスには、ソーステーブルのすべての属性が含まれます。すべてのテーブルデータがインデックスに複製されるため、ALL の射影にすると、セカンダリインデックスが最大になります。

前の図では、非キー属性の Replies が LastPostIndex に射影されています。アプリケーションでは LastPostIndex テーブル全体ではなく Thread に対するクエリを実行し、ウェブページに Subject、Replies、LastPostDateTime を表示することができます。他の非キー属性がリクエストされた場合、DynamoDB はそれらの属性を Thread テーブルからフェッチする必要があります。

アプリケーションの観点から見ると、ベーステーブルから追加の属性をフェッチする処理は、自動的で透過的なので、アプリケーションロジックを書き直す必要はありません。ただし、このようなフェッチによって、ローカルセカンダリインデックスを使用することで得られるパフォーマンスの利点が大幅に小さくなる可能性があります。

ローカルセカンダリインデックスに射影する属性を選択する場合には、プロビジョニングされるスループットコストとストレージコストのトレードオフを考慮する必要があります。

- ごく一部の属性だけに最小のレイテンシーでアクセスする必要がある場合は、それらの属性だけをローカルセカンダリインデックスに射影することを検討してください。インデックスが小さいほど少ないコストで格納でき、書き込みコストも低くなります。まれにしかフェッチされない属性がある場合には、それらの属性を格納するコストよりも、プロビジョニングされるスループットのコストのほうが長期的に大きくなる可能性があります。
- アプリケーションが非キー属性に頻繁にアクセスする場合には、それらの属性をローカルセカンダリインデックスに射影することを検討してください。ローカルセカンダリインデックスのストレージコストの増加は、頻繁にテーブルスキャンを実行するコストの減少で相殺されます。
- ほとんどの非キー属性に頻繁にアクセスする場合は、それらの属性を (場合によっては、ベーステーブル全体を) ローカルセカンダリインデックスに射影することができます。それによってフェッチが不要になるため、柔軟性が最大になり、プロビジョニングされるスループットが最小限になります。ただし、ストレージコストが増加し、すべての属性を射影する場合には 2 倍にまで増大します。
- アプリケーションでテーブルのクエリを頻繁に行う必要がなく、テーブル内のデータに対する書き込みや更新が多数になる場合は、KEYS_ONLY を射影することを検討してください。ローカルセカンダリインデックスは、最小サイズになりますが、それでもクエリのアクティビティに必要な場合は利用可能です。

ローカルセカンダリインデックスの作成

テーブルで 1 つ以上のローカルセカンダリインデックスを作成するには、CreateTable オペレーションの LocalSecondaryIndexes パラメータを使用します。テーブルのローカルセカンダリインデックスは、テーブルが作成された時点で作成されます。テーブルを削除すると、そのテーブルにあるローカルセカンダリインデックスも削除されます。

ローカルセカンダリインデックスのソートキーにとなる 1 つの非キー属性を指定する必要があります。選択する属性はスカラー String、Number、または Binary である必要があります。その他のスカラー型、ドキュメント型、およびセット型は許可されません。データ型の詳細なリストについては、「データ型」を参照してください。

> **Important**
>
> ローカルセカンダリインデックスがあるテーブルには、パーティションキーの値ごとに 10 GB のサイズ制限があります。ローカルセカンダリインデックスがあるテーブルには、1 つのパーティションキー値の合計サイズが 10 GB を超えない限り、任意の数の項目を格納できます。詳細については、「」を参照してください項目コレクションのサイズ制限

ローカルセカンダリインデックスには、任意のデータ型の属性を射影できます。これには、スカラー、ドキュメント、およびセットが含まれます。データ型の詳細なリストについては、「データ型」を参照してください。

ローカルセカンダリインデックスからのデータの読み込み

Query および Scan オペレーションを使用して、ローカルセカンダリインデックスから項目を取得できます。GetItem および BatchGetItem オペレーションは、ローカルセカンダリインデックスでは使用できません。

ローカルセカンダリインデックスのクエリ

DynamoDB テーブルでは、各項目のパーティションキー値とソートキー値の組み合わせは、一意である必要があります。ただし、ローカルセカンダリインデックスでは、ソートキーの値は、特定のパーティションキーの値に対して一意である必要がありません。ローカルセカンダリインデックスに同じソートキーを持つ複数の項目がある場合、Query オペレーションは、同じパーティションキーの値を持つすべての項目を返します。レスポンスでは、一致する項目は特定の順序で返されません。

結果整合性の読み込みまたは完全整合性の読み込みを使用して、ローカルセカンダリインデックスのクエリを行うことができます。必要な整合性のタイプを指定するには、ConsistentRead オペ

レーションの Query パラメータを使用します。ローカルセカンダリインデックスからの完全整合性の読み込みでは、常に最新の更新された値が返されます。クエリがベーステーブルからさらに属性をフェッチする必要がある場合、それらの属性はインデックスについて整合性があることになります。

Example

特定のフォーラムのディスカッションスレッドにあるデータをリクエストする Query から返される次のデータを考えてみます。

```
{
    "TableName": "Thread",
    "IndexName": "LastPostIndex",
    "ConsistentRead": false,
    "ProjectionExpression": "Subject, LastPostDateTime, Replies, Tags",
    "KeyConditionExpression":
        "ForumName = :v_forum and LastPostDateTime between :v_start and :v_end",
    "ExpressionAttributeValues": {
        ":v_start": {"S": "2015-08-31T00:00:00.000Z"},
        ":v_end": {"S": "2015-11-31T00:00:00.000Z"},
        ":v_forum": {"S": "EC2"}
    }
}
```

このクエリでは次のようになっています。

- DynamoDB は ForumName パーティションキーを使用して LastPostIndex にアクセスし、「EC2」のインデックス項目を特定します。このキーを持つすべてのインデックス項目が、すばやく取り出せるように隣り合わせに格納されます。
- このフォーラム内で、DynamoDB はインデックスを使用して、指定された LastPostDateTime 条件に一致するキーを検索します。
- Replies 属性はインデックスに射影されているため、DynamoDB は追加でプロビジョニングされたスループットを使用することなく、この属性を取得できます。
- Tags 属性はインデックスに射影されていないため、DynamoDB は Thread テーブルにアクセスしてこの属性をフェッチする必要があります。
- 結果が、LastPostDateTime によってソートされ、返されます。インデックスのエントリはまずパーティションキーの値によって、次にソートキーの値によって並べ替えられ、保存される順序で Query から返されます（ScanIndexForward パラメータを使用すると、結果が降順で返されます）。

ローカルセカンダリインデックスには Tags 属性が射影されていないため、DynamoDB は、読み取りキャパシティユニットを追加で使用して、ベーステーブルからこの属性をフェッチする必要があります。このクエリを頻繁に実行する必要がある場合は、Tags 属性を LastPostIndex に射影して、ベーステーブルからフェッチされないようにする必要があります。ただし、Tags 属性をまれにしか使用しない場合は、Tags 属性をインデックスに射影するためにストレージを追加することが有効でない可能性があります。

ローカルセカンダリインデックスのスキャン

Scan を使用して、ローカルセカンダリインデックスからすべてのデータを取得できます。リクエストにはベーステーブル名とインデックス名を指定する必要があります。Scan では、DynamoDB はインデックスのすべてのデータを読み込み、それをアプリケーションに返します。また、データの一部のみを返し、残りのデータを破棄するようにリクエストすることもできます。これを行うには、FilterExpression API の Scan パラメータを使用します。詳細については、「Scan のフィルタ式」を参照してください。

項目の書き込みとローカルセカンダリインデックス

DynamoDB によって、すべてのローカルセカンダリインデックスがそれぞれのベーステーブルと自動的に同期されます。アプリケーションがインデックスに直接書き込むことはありません。ただし、DynamoDB でこれらのインデックスがどのように維持されるかを理解することは重要です。

ローカルセカンダリインデックスを作成する場合は、インデックスのソートキーになる属性を指定します。その属性のデータ型も指定します。つまり、ベーステーブルに項目を書き込むとき、その項目にインデックスキー属性が定義されている場合は、その型がインデックスキースキーマのデータ型に一致する必要があります。LastPostIndex の場合、インデックス内の LastPostDateTime ソートキーは、String データ型として定義されています。Thread テーブルに項目を追加し、LastPostDateTime に対して別のデータ型を指定する場合 (Number など)、データ型の不一致により DynamoDB によって ValidationException が返されます。

ベーステーブル内の項目とローカルセカンダリインデックス内の項目を 1 対 1 の関係にする必要はありません。この動作は、多くのアプリケーションにとってメリットになります。

多数のローカルセカンダリインデックスがあるテーブルは、インデックス数が少ないテーブルに比べて書き込みアクティビティに多くのコストを要します。詳細については、「ローカルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項」を参照してください。

> **Important**
>
> ローカルセカンダリインデックスがあるテーブルには、パーティションキーの値ごとに 10 GB のサイズ制限があります。ローカルセカンダリインデックスがあるテーブルには、1 つのパーティションキー値の合計サイズが 10 GB を超えない限り、任意の数の項目を格納できます。詳細については、「項目コレクションのサイズ制限」を参照してください。

ローカルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項

DynamoDB にテーブルを作成する場合には、テーブルで予想されるワークロードに応じた読み込み/書き込みキャパシティーユニットをプロビジョニングします。このワークロードには、テーブルのローカルセカンダリインデックスの読み込み/書き込みアクティビティが含まれます。

プロビジョニングされたスループット容量の現在の料金を確認するには、「Amazon DynamoDB 料金」を参照してください。

読み込みキャパシティユニット

ローカルセカンダリインデックスに対してクエリを実行する場合、使用される読み込みキャパシティーユニットの数は、データのアクセス方法によって異なります。

テーブルに対するクエリと同様に、インデックスクエリでは `ConsistentRead` の値に応じて、結果整合性のある読み込みまたは強力な整合性のある読み込みを実行できます。1 回の強力な整合性のある読み込みでは、1 つの読み込みキャパシティーユニットが消費され、結果整合性のある読み込みではその半分が消費されます。したがって結果整合性のある読み込みを選択することで、読み込みキャパシティーユニットのコストを削減できます。

インデックスキーと射影された属性だけをリクエストするインデックスクエリの場合、DynamoDB はテーブルに対するクエリの場合と同じ方法で、プロビジョニングされた読み込みアクティビティを計算します。唯一の違いは、ベーステーブル内の項目のサイズではなくインデックスエントリのサイズに基づいて計算が行われることです。読み込みキャパシティーユニットの数は、返されたすべての項目について射影されたすべての属性のサイズの合計です。結果は、次の 4 KB 境界まで切り上げられます。DynamoDB がプロビジョニングされたスループットの利用率を計算する方法の詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。

ローカルセカンダリインデックスに射影されていない属性を読み込むインデックスクエリの場合、DynamoDB は射影された属性をインデックスから読み込むのに加えて、それらの属性をベーステーブルからフェッチする必要があります。これらのフェッチは、`Select` オペレーションの

ProjectionExpression または Query パラメータに、射影されていない属性を含めた場合に実行されます。フェッチによってクエリ応答のレイテンシーが増加し、プロビジョニングされるスループットのコストも増加します。前述のローカルセカンダリインデックスからの読み込みに加えて、フェッチされるベーステーブル項目それぞれについて、読み込みキャパシティーユニットの料金がかかります。この料金は、リクエストした属性だけでなく、項目全体をテーブルから読み込むために発生するものです。

Query オペレーションによって返される結果の最大サイズは 1 MB です。これには、返されるすべての項目にわたる、すべての属性の名前と値のサイズが含まれます。ただし、ローカルセカンダリインデックスに対するクエリによって、DynamoDB がベーステーブルから項目の属性をフェッチする場合には、結果で示されるデータの最大サイズが小さくなる可能性があります。この場合、結果のサイズは次の合計になります。

- インデックス内で一致する項目のサイズ (次の 4 KB に切り上げ)
- ベーステーブル内で一致する各項目のサイズ (項目ごとに次の 4 KB に切り上げ)

この式を使用すると、クエリオペレーションによって返される結果の最大サイズは依然として 1 MB になります。

たとえば、各項目のサイズが 300 バイトであるテーブルがあるとします。そのテーブルにはローカルセカンダリインデックスがありますが、各項目のうち 200 バイトだけがインデックスに射影されます。このインデックスに対して Query を行うときに各項目についてテーブルのフェッチが必要で、クエリによって 4 つの項目が返されるとします。DynamoDB では、次のように合計されます。

- インデックス内で一致する項目のサイズは 200 バイト × 4 項目 = 800 バイトになり、それが 4 KB に切り上げられます。
- ベーステーブル内で一致する項目のサイズ: (300 バイト、4 KB に切り上げ) × 4 項目 = 16 KB。

それによって、結果データの合計サイズは 20 KB になります。

書き込みキャパシティユニット

テーブル内の項目が追加、更新、または削除された場合にローカルセカンダリインデックスを更新すると、テーブルにプロビジョニングされた書き込みキャパシティーユニットが消費されます。書き込み用にプロビジョニングされたスループットの合計コストは、テーブルに対する書き込みと、ローカルセカンダリインデックスの更新で消費された書き込みキャパシティーユニットの合計になります。

ローカルセカンダリインデックスに項目を書き込むコストは、いくつかの要因に左右されます。

- インデックス付き属性が定義されたテーブルに新しい項目を書き込む場合、または既存の項目を更新して未定義のインデックス付き属性を定義する場合には、インデックスへの項目の挿入に 1 回の書き込みオペレーションが必要です。
- テーブルに対する更新によってインデックス付きキー属性の値が (A から B に) 変更された場合には、インデックスから既存の項目を削除し、インデックスに新しい項目を挿入するために、2 回の書き込みが必要です。
- インデックス内に既存の項目があって、テーブルに対する書き込みによってインデックス付き属性が削除された場合は、インデックスから古い項目の射影を削除するために、1 回の書き込みが必要です。
- 項目の更新の前後にインデックス内に項目が存在しない場合は、インデックスで追加の書き込みコストは発生しません。

これらすべての要因は、インデックス内の各項目のサイズが 1 KB 以下であるという前提で書き込みキャパシティーユニット数を算出します。インデックスエントリがそれよりも大きい場合は、書き込みキャパシティーユニットを追加する必要があります。クエリが返す必要がある属性を特定し、それらの属性だけをインデックスに射影することで、書き込みコストは最小になります。

ローカルセカンダリインデックスのストレージに関する考慮事項

アプリケーションがテーブルに項目を書き込むと、DynamoDB では正しい属性のサブセットが、それらの属性が表示されるローカルセカンダリインデックスに自動的にコピーされます。AWS アカウントでは、テーブル内の項目のストレージと、そのベーステーブルのローカルセカンダリインデックスにある属性のストレージに対して課金されます。

インデックス項目が使用するスペースの量は、次の量の合計になります。

- ベーステーブルのプライマリキー (パーティションキーとソートキー) のサイズのバイト数
- インデックスキー属性のサイズのバイト数
- 射影された属性 (存在する場合) のサイズのバイト数
- インデックス項目あたり 100 バイトのオーバーヘッド

ローカルセカンダリインデックスに必要なストレージは、インデックス内の 1 項目の平均サイズの見積もり値に、インデックス内の項目数を掛けて見積もることができます。

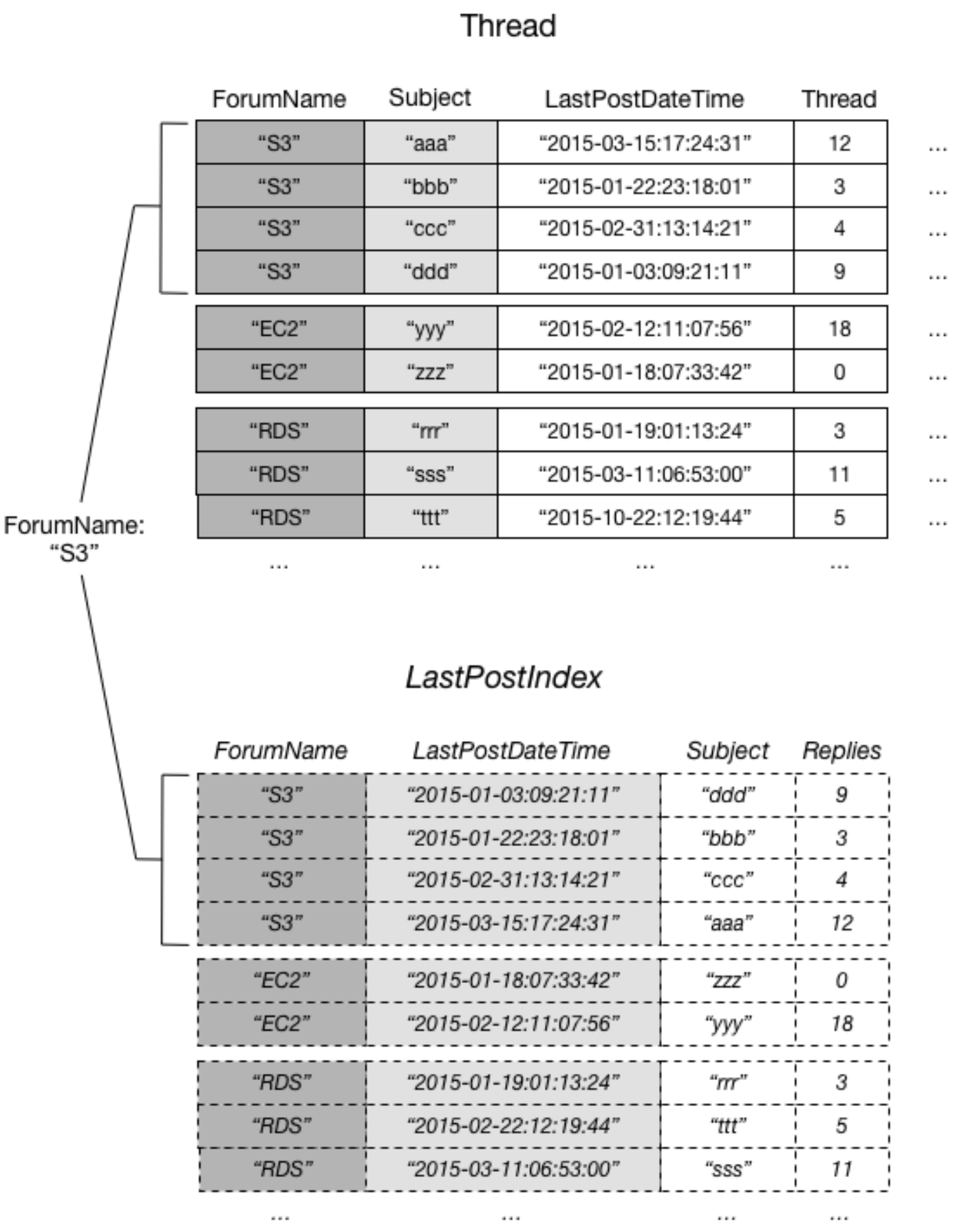
特定の属性が定義されていない項目がテーブルに含まれており、その属性がインデックスソートキーとして定義されている場合、DynamoDB はその項目に関連するデータをインデックスに書き込みません。

ローカルセカンダリインデックス内の項目コレクション

> **Note**
>
> このセクションは、ローカルセカンダリインデックスがあるテーブルだけに関係します。

DynamoDB において、項目コレクションとは、テーブルおよびそのローカルセカンダリインデックス全体で同じパーティションキーの値を持つ項目のグループです。このセクションで使用する例では、Thread テーブルのパーティションキーは ForumName で、LastPostIndex のパーティションキーも ForumName です。同じ ForumName を持つテーブルとインデックス項目は、すべて同じ項目コレクションの一部です。例えば、Thread テーブルと LastPostIndex ローカルセカンダリインデックスには、フォーラム EC2 用の項目コレクションと、フォーラム RDS 用の別の項目コレクションがあります。

次の図は、フォーラム S3 の項目コレクションを示しています。


Thread
ForumName
Subject
LastPostDateTime
Thread
"S3"
"aaa"
"2015-03-15:17:24:31"
12
"S3"
"bbb"
"2015-01-22:23:18:01"
3
"S3"
"ccc"
"2015-02-31:13:14:21"
4
"S3"
"ddd"
"2015-01-03:09:21:11"
9
"EC2"
"yyy"
"2015-02-12:11:07:56"
18
"EC2"
"zzz"
"2015-01-18:07:33:42"
0
"RDS"
"rrr"
"2015-01-19:01:13:24"
3
"RDS"
"sss"
"2015-03-11:06:53:00"
11
"RDS"
"ttt"
"2015-10-22:12:19:44"
5
ForumName:
"S3"
LastPostIndex
ForumName
LastPostDateTime
Subject
Replies
"S3"
"2015-01-03:09:21:11"
"ddd"
9
"S3"
"2015-01-22:23:18:01"
"bbb"
3
"S3"
"2015-02-31:13:14:21"
"ccc"
4
"S3"
"2015-03-15:17:24:31"
"aaa"
12
"EC2"
"2015-01-18:07:33:42"
"zzz"
0
"EC2"
"2015-02-12:11:07:56"
"yyy"
18
"RDS"
"2015-01-19:01:13:24"
"rrr"
3
"RDS"
"2015-02-22:12:19:44"
"ttt"
5
"RDS"
"2015-03-11:06:53:00"
"sss"
11

この図では、項目コレクションは、Thread および LastPostIndex 内のすべての項目で構成されていて、ForumName パーティションキーの値は「S3」です。テーブルにその他のローカルセカンダリインデックスがあった場合には、ForumName が「S3」であるそれらのインデックス内の項目も、項目コレクションの一部になります。

DynamoDB では次のオペレーションのいずれかを使用して、項目コレクションに関する情報を返すことができます。

- BatchWriteItem
- DeleteItem
- PutItem
- UpdateItem
- TransactWriteItems

これらのオペレーションでは、それぞれ ReturnItemCollectionMetrics パラメータがサポートされています。このパラメータを SIZE に設定すると、インデックス内の各項目コレクションのサイズに関する情報が表示されます。

Example

以下に、UpdateItem が Thread に設定されている、ReturnItemCollectionMetrics テーブルでの SIZE オペレーションの出力の例を示します。更新された項目には ForumName 値「EC2」があったため、出力にはその項目コレクションに関する情報が含まれています。

```
{
    ItemCollectionMetrics: {
        ItemCollectionKey: {
            ForumName: "EC2"
        },
        SizeEstimateRangeGB: [0.0, 1.0]
    }
}
```

SizeEstimateRangeGB オブジェクトは、この項目コレクションのサイズが 0 GB と 1 GB の間であることを示します。DynamoDB では、このサイズ見積りが定期的に更新されるため、項目を変更した時には次回の数字が異なる場合があります。

項目コレクションのサイズ制限

1 つ以上のローカルセカンダリインデックスを含む項目コレクションのサイズは、すべて最大 10 GB です。これは、ローカルセカンダリインデックスのないテーブルの項目コレクションには適用されません。また、グローバルセカンダリインデックスの項目コレクションにも適用されません。1 つ以上のローカルセカンダリインデックスを含むテーブルのみに影響します。

項目コレクションが 10 GB の制限を超えると、DynamoDB は `ItemCollectionSizeLimitExceededException` を返す可能性があります。この場合、その項目コレクションに項目を追加したり、項目サイズを大きくしたりすることができなくなる場合があります。(項目コレクションのサイズを小さくする読み込み/書き込みオペレーションは、引き続き実行できます)。その他の項目コレクションには項目を追加することができます。

項目コレクションのサイズを小さくするには、次のいずれかを実行します。

- 問題になっているパーティションキーの値を持つ不要な項目を削除します。ベーステーブルからこれらの項目を削除すると、DynamoDB では同じパーティションキーの値を持つインデックスエントリも削除されます。
- 属性を削除するか属性のサイズを小さくすることで、項目を更新します。これらの属性がローカルセカンダリインデックスに射影されている場合には、DynamoDB では対応するインデックスエントリのサイズも小さくなります。
- 同じパーティションキーおよびソートキーを使用して新しいテーブルを作成し、古いテーブルから新しいテーブルに項目を移動します。これは、頻繁にアクセスされない履歴データがテーブルに含まれている場合に適した方法です。また、この履歴データを Amazon Simple Storage Service (Amazon S3) にアーカイブすることを検討することもできます。

項目コレクションの合計サイズが 10 GB 未満になると、再び同じパーティションキーの値を使用して項目を追加できます。

項目コレクションのサイズを監視するようにアプリケーションを設定することをお勧めします。1 つの方法としては、`ReturnItemCollectionMetrics`、`SIZE`、`BatchWriteItem`、または `DeleteItem` を使用する場合に、`PutItem` パラメータを `UpdateItem` に設定するというものがあります。アプリケーションで出力内の `ReturnItemCollectionMetrics` オブジェクトを調査し、項目コレクションがユーザー定義の制限 (たとえば 8 GB) を超えた場合にエラーメッセージを記録するようにします。制限を 10 GB より低く設定すれば早期警告システムになり、項目コレクションが上限に達する前に余裕をもって何らかの対処をとることができます。

項目コレクションおよびパーティション

1 つ以上のローカルセカンダリインデックスを含むテーブルでは、各項目コレクションは 1 つのパーティションに保存されます。このような項目コレクションの合計サイズは、そのパーティションのキャパシティー (10 GB) に制限されます。データモデルにサイズに制限のない項目コレクションが含まれている場合や、一部の項目コレクションが将来に 10 GB を超えると合理的に予期されるアプリケーションでは、代わりにグローバルセカンダリインデックスの使用を検討します。

アプリケーションを設計する場合は、テーブルデータが異なるパーティションキー値に均一に分配されるようにする必要があります。ローカルセカンダリインデックス を持つテーブルについては、単一のパーティション内の単一の項目コレクションに読み込み/書き込みアクティビティの「ホットスポット」が作られないように、アプリケーションを設計します。

ローカルセカンダリインデックスの操作: Java

AWS SDK for Java ドキュメント API を使用して、1 つ以上のローカルセカンダリインデックスを持つ Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。

以下に、AWS SDK for Java ドキュメント API を使用したテーブルオペレーションの一般的な手順を示します。

1. DynamoDB クラスのインスタンスを作成します。
2. 対応するリクエストオブジェクトを作成して、オペレーションについて必要なパラメータとオプションパラメータを入力します。
3. 前のステップで作成したクライアントが提供する適切なメソッドを呼び出します。

トピック

- ローカルセカンダリインデックスを持つテーブルを作成する
- ローカルセカンダリインデックスを持つテーブルの説明
- ローカルセカンダリインデックスのクエリ
- 例: Java ドキュメント API を使用したローカルセカンダリインデックス

ローカルセカンダリインデックスを持つテーブルを作成する

ローカルセカンダリインデックスは、テーブルの作成と同時に作成する必要があります。これを行うには、`createTable` メソッドを使用し、1 つ以上のローカルセカンダリインデックスの仕様を指定

します。次の Java コード例では、ミュージックコレクション内の曲に関する情報を保持するためのテーブルを作成しています。パーティションキーは Artist で、ソートキーは SongTitle です。セカンダリインデックス AlbumTitleIndex は、アルバムタイトルによるクエリを容易にします。

DynamoDB ドキュメント API を使用して、ローカルセカンダリインデックスを持つテーブルを作成する手順を次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. リクエスト情報を指定する CreateTableRequest クラスのインスタンスを作成します。

   テーブル名、プライマリキー、およびプロビジョニングされたスループット値を指定する必要があります。ローカルセカンダリインデックスの場合は、インデックス名、インデックスソートキーの名前とデータ型、インデックスのキースキーマ、および属性射影を指定する必要があります。
3. リクエストオブジェクトをパラメータとして指定して、createTable メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。このコードは、AlbumTitle 属性に関するセカンダリインデックスを持つテーブル (Music) を作成します。テーブルパーティションキーとソートキー、およびインデックスソートキーのみが、インデックスに射影される属性です。

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

CreateTableRequest createTableRequest = new
 CreateTableRequest().withTableName(tableName);

//ProvisionedThroughput
createTableRequest.setProvisionedThroughput(new
 ProvisionedThroughput().withReadCapacityUnits((long)5).withWriteCapacityUnits((long)5));

//AttributeDefinitions
ArrayList<AttributeDefinition> attributeDefinitions= new
 ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("Artist").withAttributeType("S"));
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("SongTitle").withAttributeType("S"));
```

```
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("AlbumTitle").withAttributeType("S"));

createTableRequest.setAttributeDefinitions(attributeDefinitions);

//KeySchema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
 KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH));  //
Partition key
tableKeySchema.add(new
 KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE));  //Sort
 key

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
 KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH));  //
Partition key
indexKeySchema.add(new
 KeySchemaElement().withAttributeName("AlbumTitle").withKeyType(KeyType.RANGE));  //
Sort key

Projection projection = new Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("Genre");
nonKeyAttributes.add("Year");
projection.setNonKeyAttributes(nonKeyAttributes);

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()

 .withIndexName("AlbumTitleIndex").withKeySchema(indexKeySchema).withProjection(projection);

ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
 ArrayList<LocalSecondaryIndex>();
localSecondaryIndexes.add(localSecondaryIndex);
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。

ローカルセカンダリインデックスを持つテーブルの説明

テーブルのローカルセカンダリインデックスに関する情報を取得するには、describeTable メソッドを使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

AWS SDK for Java ドキュメント API を使用して、テーブルのローカルセカンダリインデックス情報にアクセスするためのステップを次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. Table クラスのインスタンスを作成します。テーブル名を入力する必要があります。
3. Table オブジェクトの describeTable メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);

TableDescription tableDescription = table.describe();

List<LocalSecondaryIndexDescription> localSecondaryIndexes
    = tableDescription.getLocalSecondaryIndexes();

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

Iterator<LocalSecondaryIndexDescription> lsiIter = localSecondaryIndexes.iterator();
while (lsiIter.hasNext()) {

    LocalSecondaryIndexDescription lsiDescription = lsiIter.next();
    System.out.println("Info for index " + lsiDescription.getIndexName() + ":");
    Iterator<KeySchemaElement> kseIter = lsiDescription.getKeySchema().iterator();
    while (kseIter.hasNext()) {
```

```
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
    Projection projection = lsiDescription.getProjection();
    System.out.println("\tThe projection type is: " + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: " +
 projection.getNonKeyAttributes());
    }
}
```

ローカルセカンダリインデックスのクエリ

テーブルに Query を実行するのとほぼ同じ方法で、ローカルセカンダリインデックスに対する Query オペレーションを使用することができます。インデックス名、インデックスソートキーのクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは AlbumTitleIndex、インデックスソートキーは AlbumTitle です。

返される属性は、インデックスに射影された属性だけです。このクエリを変更して非キー属性を選択することもできますが、これには比較的コストのかかるテーブルフェッチアクティビティが必要です。テーブルのフェッチの詳細については、「属性の射影」を参照してください。

AWS SDK for Java ドキュメント API を使用してローカルセカンダリインデックスをクエリする手順を次に示します。

1. DynamoDB クラスのインスタンスを作成します。
2. Table クラスのインスタンスを作成します。テーブル名を入力する必要があります。
3. Index クラスのインスタンスを作成します。インデックス名を入力する必要があります。
4. Index クラスの query メソッドを呼び出します。

以下の Java コード例は、前述のステップの例です。

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);
Index index = table.getIndex("AlbumTitleIndex");
```

```
QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
    .withValueMap(new ValueMap()
        .withString(":v_artist", "Acme Band")
        .withString(":v_title", "Songs About Life"));

ItemCollection<QueryOutcome> items = index.query(spec);

Iterator<Item> itemsIter = items.iterator();

while (itemsIter.hasNext()) {
    Item item = itemsIter.next();
    System.out.println(item.toJSONPretty());
}
```

ローカルセカンダリインデックスの整合性のある読み取り

結果整合性のある読み込みのみをサポートするグローバルセカンダリインデックスとは異なり、ローカルセカンダリインデックスは結果整合性のある読み込みと強力な整合性のある読み込みの両方をサポートします。ローカルセカンダリインデックスからの完全整合性の読み込みでは、常に最新の更新された値が返されます。クエリがベーステーブルからさらに属性をフェッチする必要がある場合、それらのフェッチされた属性もインデックスについて整合性があることになります。

デフォルトでは、Query は結果整合性のある読み込みを使用します。強力な整合性のある読み込みをリクエストするには、QuerySpec で ConsistentRead を true に設定します。次の例では、強力な整合性のある読み込みを使用して AlbumTitleIndex をクエリを実行します。

Example

```
QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
    .withValueMap(new ValueMap()
        .withString(":v_artist", "Acme Band")
        .withString(":v_title", "Songs About Life"))
    .withConsistentRead(true);
```

> **Note**
>
> 強力な整合性のある読み込みでは、返されるデータ 4 KB ごとに 1 つの読み込みキャパシティーユニット (切り上げ) が消費されますが、結果整合性のある読み込みではその半分が消費されます。例えば、9 KB のデータを返す強力な整合性のある読み込みでは、3 つの読み込

みキャパシティーユニット (9 KB / 4 KB = 2.25、3 に切り上げ) が消費されますが、結果整合性のある読み込みを使用した同じクエリでは、1.5 の読み込みキャパシティーユニットが消費されます。アプリケーションがわずかに古くなる可能性のあるデータの読み取りを許容できる場合は、結果整合性のある読み込みを使用して読み込みキャパシティーの使用量を減らします。詳細については、「読み込みキャパシティユニット」を参照してください。

例: Java ドキュメント API を使用したローカルセカンダリインデックス

次の Java コード例は、Amazon DynamoDB でローカルセカンダリインデックスを操作する方法を示しています。この例では、パーティションキーを CustomerId として 、ソートキーを OrderId として CustomerOrders という名前のテーブルを作成しています。このテーブルには、次の 2 つのローカルセカンダリインデックスがあります。

- OrderCreationDateIndex – ソートキーは OrderCreationDate です。次の属性がインデックスに射影されます。
  - ProductCategory
  - ProductName
  - OrderStatus
  - ShipmentTrackingId
- IsOpenIndex – ソートキーは IsOpen です。すべてのテーブル属性がインデックスに射影されます。

CustomerOrders テーブルが作成されると、プログラムは顧客の注文を表すデータをテーブルにロードします。次に、ローカルセカンダリインデックスを使用してデータにクエリを実行します。最後に、プログラムは CustomerOrders テーブルを削除します。

以下の例をテストするための詳細な手順については、「Java コードの例」を参照してください。

Example

```
package com.example.dynamodb;

import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;
```

```
import java.util.HashMap;
import java.util.Map;

public class DocumentAPILocalSecondaryIndexExample {

    static DynamoDbClient client = DynamoDbClient.create();
    public static String tableName = "CustomerOrders";

    public static void main(String[] args) {
        createTable();
        loadData();
        query(null);
        query("IsOpenIndex");
        query("OrderCreationDateIndex");
        deleteTable(tableName);
    }

    public static void createTable() {
        CreateTableRequest request = CreateTableRequest.builder()
            .tableName(tableName)
            .provisionedThroughput(ProvisionedThroughput.builder()
                .readCapacityUnits(1L)
                .writeCapacityUnits(1L)
                .build())
            .attributeDefinitions(

 AttributeDefinition.builder().attributeName("CustomerId").attributeType(ScalarAttributeType.S)

 AttributeDefinition.builder().attributeName("OrderId").attributeType(ScalarAttributeType.N).bu

 AttributeDefinition.builder().attributeName("OrderCreationDate").attributeType(ScalarAttribute

 AttributeDefinition.builder().attributeName("IsOpen").attributeType(ScalarAttributeType.N).bui
            .keySchema(

 KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

 KeySchemaElement.builder().attributeName("OrderId").keyType(KeyType.RANGE).build())
            .localSecondaryIndexes(
                LocalSecondaryIndex.builder()
                    .indexName("OrderCreationDateIndex")
                    .keySchema(
```

```
 KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

 KeySchemaElement.builder().attributeName("OrderCreationDate").keyType(KeyType.RANGE).build())
                    .projection(Projection.builder()
                        .projectionType(ProjectionType.INCLUDE)
                        .nonKeyAttributes("ProductCategory", "ProductName")
                        .build())
                    .build(),
                LocalSecondaryIndex.builder()
                    .indexName("IsOpenIndex")
                    .keySchema(

 KeySchemaElement.builder().attributeName("CustomerId").keyType(KeyType.HASH).build(),

 KeySchemaElement.builder().attributeName("IsOpen").keyType(KeyType.RANGE).build())
                    .projection(Projection.builder()
                        .projectionType(ProjectionType.ALL)
                        .build())
                    .build())
            .build();

        System.out.println("Creating table " + tableName + "...");
        client.createTable(request);

        try (DynamoDbWaiter waiter = client.waiter()) {
            WaiterResponse<DescribeTableResponse> response =
 waiter.waitUntilTableExists(r -> r.tableName(tableName));
            response.matched().response().ifPresent(System.out::println);
        }
    }

    public static void query(String indexName) {

 System.out.println("\n***********************************************************\n");
        System.out.println("Querying table " + tableName + "...");

        if ("IsOpenIndex".equals(indexName)) {
            System.out.println("\nUsing index: '" + indexName + "': Bob's orders that
 are open.");
            System.out.println("Only a user-specified list of attributes are returned
\n");

            Map<String, AttributeValue> values = new HashMap<>();
```

```
            values.put(":v_custid",
AttributeValue.builder().s("bob@example.com").build());
            values.put(":v_isopen", AttributeValue.builder().n("1").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
                .indexName(indexName)
                .keyConditionExpression("CustomerId = :v_custid and IsOpen
= :v_isopen")
                .expressionAttributeValues(values)
                .projectionExpression("OrderCreationDate, ProductCategory, ProductName,
OrderStatus")
                .build();

            System.out.println("Query: printing results...");
            client.query(request).items().forEach(System.out::println);

        } else if ("OrderCreationDateIndex".equals(indexName)) {
            System.out.println("\nUsing index: '" + indexName + "': Bob's orders that
were placed after 01/31/2015.");
            System.out.println("Only the projected attributes are returned\n");

            Map<String, AttributeValue> values = new HashMap<>();
            values.put(":v_custid",
AttributeValue.builder().s("bob@example.com").build());
            values.put(":v_orddate", AttributeValue.builder().n("20150131").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
                .indexName(indexName)
                .keyConditionExpression("CustomerId = :v_custid and OrderCreationDate
>= :v_orddate")
                .expressionAttributeValues(values)
                .select(Select.ALL_PROJECTED_ATTRIBUTES)
                .build();

            System.out.println("Query: printing results...");
            client.query(request).items().forEach(System.out::println);

        } else {
            System.out.println("\nNo index: All of Bob's orders, by OrderId:\n");

            Map<String, AttributeValue> values = new HashMap<>();
```

```
            values.put(":v_custid",
 AttributeValue.builder().s("bob@example.com").build());

            QueryRequest request = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression("CustomerId = :v_custid")
                .expressionAttributeValues(values)
                .build();

            System.out.println("Query: printing results...");
            client.query(request).items().forEach(System.out::println);
        }
    }

    public static void deleteTable(String tableName) {
        System.out.println("Deleting table " + tableName + "...");
        client.deleteTable(DeleteTableRequest.builder().tableName(tableName).build());

        try (DynamoDbWaiter waiter = client.waiter()) {
            waiter.waitUntilTableNotExists(r -> r.tableName(tableName));
        }
    }

    public static void loadData() {
        System.out.println("Loading data into table " + tableName + "...");

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
            "OrderId", AttributeValue.builder().n("1").build(),
            "IsOpen", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150101").build(),
            "ProductCategory", AttributeValue.builder().s("Book").build(),
            "ProductName", AttributeValue.builder().s("The Great Outdoors").build(),
            "OrderStatus", AttributeValue.builder().s("PACKING ITEMS").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
            "OrderId", AttributeValue.builder().n("2").build(),
            "IsOpen", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150221").build(),
            "ProductCategory", AttributeValue.builder().s("Bike").build(),
            "ProductName", AttributeValue.builder().s("Super Mountain").build(),
            "OrderStatus", AttributeValue.builder().s("ORDER RECEIVED").build()));
```

```
        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("alice@example.com").build(),
            "OrderId", AttributeValue.builder().n("3").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150304").build(),
            "ProductCategory", AttributeValue.builder().s("Music").build(),
            "ProductName", AttributeValue.builder().s("A Quiet Interlude").build(),
            "OrderStatus", AttributeValue.builder().s("IN TRANSIT").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("176493").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150111").build(),
            "ProductCategory", AttributeValue.builder().s("Movie").build(),
            "ProductName", AttributeValue.builder().s("Calm Before The Storm").build(),
            "OrderStatus", AttributeValue.builder().s("SHIPPING DELAY").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("859323").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("2").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150124").build(),
            "ProductCategory", AttributeValue.builder().s("Music").build(),
            "ProductName", AttributeValue.builder().s("E-Z Listening").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("756943").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("3").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150221").build(),
            "ProductCategory", AttributeValue.builder().s("Music").build(),
            "ProductName", AttributeValue.builder().s("Symphony 9").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("645193").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("4").build(),
            "IsOpen", AttributeValue.builder().n("1").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150222").build(),
            "ProductCategory", AttributeValue.builder().s("Hardware").build(),
            "ProductName", AttributeValue.builder().s("Extra Heavy Hammer").build(),
            "OrderStatus", AttributeValue.builder().s("PACKING ITEMS").build()));
```

```
        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("5").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150309").build(),
            "ProductCategory", AttributeValue.builder().s("Book").build(),
            "ProductName", AttributeValue.builder().s("How To Cook").build(),
            "OrderStatus", AttributeValue.builder().s("IN TRANSIT").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("440185").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("6").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150318").build(),
            "ProductCategory", AttributeValue.builder().s("Luggage").build(),
            "ProductName", AttributeValue.builder().s("Really Big Suitcase").build(),
            "OrderStatus", AttributeValue.builder().s("DELIVERED").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("893927").build()));

        putItem(Map.of(
            "CustomerId", AttributeValue.builder().s("bob@example.com").build(),
            "OrderId", AttributeValue.builder().n("7").build(),
            "OrderCreationDate", AttributeValue.builder().n("20150324").build(),
            "ProductCategory", AttributeValue.builder().s("Golf").build(),
            "ProductName", AttributeValue.builder().s("PGA Pro II").build(),
            "OrderStatus", AttributeValue.builder().s("OUT FOR DELIVERY").build(),
            "ShipmentTrackingId", AttributeValue.builder().s("383283").build()));
    }

    private static void putItem(Map<String, AttributeValue> item) {

 client.putItem(PutItemRequest.builder().tableName(tableName).item(item).build());
    }
}
```

ローカルセカンダリインデックスの操作: .NET

トピック

- ローカルセカンダリインデックスを持つテーブルを作成する
- ローカルセカンダリインデックスを持つテーブルの説明
- ローカルセカンダリインデックスのクエリ

- 例: AWS SDK for .NET 低レベル API を使用したローカルセカンダリインデックス

AWS SDK for .NET 低レベル API を使用して、1 つ以上のローカルセカンダリインデックスを含む Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。これらのオペレーションは、対応する低レベル DynamoDB API アクションにマッピングされます。詳細については、「.NET コード例」を参照してください。

以下に、.NET 低レベル API を使用したテーブルオペレーションの一般的な手順を示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. 対応するリクエストオブジェクトを作成して、オペレーションについて必要なパラメータとオプションパラメータを入力します。

   例えば、テーブルを作成するには CreateTableRequest オブジェクトを作成し、テーブルやインデックスにクエリを実行するには QueryRequest オブジェクトを作成します。
3. 前述のステップで作成したクライアントから提供された適切なメソッドを実行します。

ローカルセカンダリインデックスを持つテーブルを作成する

ローカルセカンダリインデックスは、テーブルの作成と同時に作成する必要があります。これを行うには、CreateTable を使用し、1 つ以上のローカルセカンダリインデックスの仕様を指定します。次の C# コード例では、ミュージックコレクション内の曲に関する情報を保持するためのテーブルを作成しています。パーティションキーは Artist で、ソートキーは SongTitle です。セカンダリインデックス AlbumTitleIndex は、アルバムタイトルによるクエリを容易にします。

.NET 低レベル API を使用して、ローカルセカンダリインデックスを含むテーブルを作成する手順を次に示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエスト情報を指定する CreateTableRequest クラスのインスタンスを作成します。

   テーブル名、プライマリキー、およびプロビジョニングされたスループット値を指定する必要があります。ローカルセカンダリインデックスの場合は、インデックス名、インデックスソートキーの名前とデータ型、インデックスのキースキーマ、および属性射影を指定する必要があります。
3. リクエストオブジェクトをパラメータとして指定して、CreateTable メソッドを実行します。

以下の C# サンプルコードは、前述のステップの例です。このコードは、AlbumTitle 属性に関するセカンダリインデックスを持つテーブル (Music) を作成します。テーブルパーティションキーとソートキー、およびインデックスソートキーのみが、インデックスに射影される属性です。

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

CreateTableRequest createTableRequest = new CreateTableRequest()
{
    TableName = tableName
};

//ProvisionedThroughput
createTableRequest.ProvisionedThroughput = new ProvisionedThroughput()
{
    ReadCapacityUnits = (long)5,
    WriteCapacityUnits = (long)5
};

//AttributeDefinitions
List<AttributeDefinition> attributeDefinitions = new List<AttributeDefinition>();

attributeDefinitions.Add(new AttributeDefinition()
{
    AttributeName = "Artist",
    AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
 {
     AttributeName = "SongTitle",
     AttributeType = "S"
 });

attributeDefinitions.Add(new AttributeDefinition()
 {
     AttributeName = "AlbumTitle",
     AttributeType = "S"
 });

createTableRequest.AttributeDefinitions = attributeDefinitions;

//KeySchema
```

```
List<KeySchemaElement> tableKeySchema = new List<KeySchemaElement>();

tableKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
 "HASH" });  //Partition key
tableKeySchema.Add(new KeySchemaElement() { AttributeName = "SongTitle", KeyType =
 "RANGE" });  //Sort key

createTableRequest.KeySchema = tableKeySchema;

List<KeySchemaElement> indexKeySchema = new List<KeySchemaElement>();
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType =
 "HASH" });  //Partition key
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "AlbumTitle", KeyType =
 "RANGE" });  //Sort key

Projection projection = new Projection() { ProjectionType = "INCLUDE" };

List<string> nonKeyAttributes = new List<string>();
nonKeyAttributes.Add("Genre");
nonKeyAttributes.Add("Year");
projection.NonKeyAttributes = nonKeyAttributes;

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
{
    IndexName = "AlbumTitleIndex",
    KeySchema = indexKeySchema,
    Projection = projection
};

List<LocalSecondaryIndex> localSecondaryIndexes = new List<LocalSecondaryIndex>();
localSecondaryIndexes.Add(localSecondaryIndex);
createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

CreateTableResponse result = client.CreateTable(createTableRequest);
Console.WriteLine(result.CreateTableResult.TableDescription.TableName);
Console.WriteLine(result.CreateTableResult.TableDescription.TableStatus);
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。

ローカルセカンダリインデックスを持つテーブルの説明

テーブルのローカルセカンダリインデックスに関する情報を取得するには、DescribeTable API を使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

.NET 低レベル API を使用してテーブルのローカルセカンダリインデックス情報にアクセスするには次の手順に従います。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエスト情報を指定する DescribeTableRequest クラスのインスタンスを作成します。テーブル名を入力する必要があります。
3. リクエストオブジェクトをパラメータとして指定して、describeTable メソッドを実行します。
4.

以下の C# サンプルコードは、前述のステップの例です。

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest()
 { TableName = tableName });
List<LocalSecondaryIndexDescription> localSecondaryIndexes =
    response.DescribeTableResult.Table.LocalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.
foreach (LocalSecondaryIndexDescription lsiDescription in localSecondaryIndexes)
{
    Console.WriteLine("Info for index " + lsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in lsiDescription.KeySchema)
    {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = lsiDescription.Projection;
```

```
    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE"))
    {
        Console.WriteLine("\t\tThe non-key projected attributes are:");

        foreach (String s in projection.NonKeyAttributes)
        {
            Console.WriteLine("\t\t" + s);
        }

    }
}
```

ローカルセカンダリインデックスのクエリ

テーブルに Query を実行するのとほぼ同じ方法で、ローカルセカンダリインデックスに対する Query を使用することができます。インデックス名、インデックスソートキーのクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは AlbumTitleIndex、インデックスソートキーは AlbumTitle です。

返される属性は、インデックスに射影された属性だけです。このクエリを変更して非キー属性を選択することもできますが、これには比較的コストのかかるテーブルフェッチアクティビティが必要です。テーブルのフェッチの詳細については、「属性の射影」を参照してください。

.NET 低レベル API を使用して、ローカルセカンダリインデックスを含むテーブルにクエリを実行する手順を次に示します。

1. AmazonDynamoDBClient クラスのインスタンスを作成します。
2. リクエスト情報を指定する QueryRequest クラスのインスタンスを作成します。
3. リクエストオブジェクトをパラメータとして指定して、query メソッドを実行します。

以下の C# サンプルコードは、前述のステップの例です。

Example

```
QueryRequest queryRequest = new QueryRequest
{
```

```
    TableName = "Music",
    IndexName = "AlbumTitleIndex",
    Select = "ALL_ATTRIBUTES",
    ScanIndexForward = true,
    KeyConditionExpression = "Artist = :v_artist and AlbumTitle = :v_title",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":v_artist",new AttributeValue {S = "Acme Band"}},
        {":v_title",new AttributeValue {S = "Songs About Life"}}
    },
};

QueryResponse response = client.Query(queryRequest);

foreach (var attribs in response.Items)
{
    foreach (var attrib in attribs)
    {
        Console.WriteLine(attrib.Key + " ---> " + attrib.Value.S);
    }
    Console.WriteLine();
}
```

例: AWS SDK for .NET 低レベル API を使用したローカルセカンダリインデックス

次の C# コード例は、Amazon DynamoDB でローカルセカンダリインデックスを操作する方法を示しています。この例では、パーティションキーを CustomerId として 、ソートキーを OrderId として CustomerOrders という名前のテーブルを作成しています。このテーブルには、次の 2 つのローカルセカンダリインデックスがあります。

- OrderCreationDateIndex – ソートキーは OrderCreationDate です。次の属性がインデックスに射影されます。
  - ProductCategory
  - ProductName
  - OrderStatus
  - ShipmentTrackingId
- IsOpenIndex – ソートキーは IsOpen です。すべてのテーブル属性がインデックスに射影されます。

CustomerOrders テーブルが作成されると、プログラムは顧客の注文を表すデータをテーブルにロードします。次に、ローカルセカンダリインデックスを使用してデータにクエリを実行します。最後に、プログラムは CustomerOrders テーブルを削除します。

以下の例をテストするための詳細な手順については、「.NET コード例」を参照してください。

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelLocalSecondaryIndexExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "CustomerOrders";

        static void Main(string[] args)
        {
            try
            {
                CreateTable();
                LoadData();

                Query(null);
                Query("IsOpenIndex");
                Query("OrderCreationDateIndex");

                DeleteTable(tableName);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
```

```
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateTable()
        {
            var createTableRequest =
                new CreateTableRequest()
                {
                    TableName = tableName,
                    ProvisionedThroughput =
                    new ProvisionedThroughput()
                    {
                        ReadCapacityUnits = (long)1,
                        WriteCapacityUnits = (long)1
                    }
                };

            var attributeDefinitions = new List<AttributeDefinition>()
        {
            // Attribute definitions for table primary key
            { new AttributeDefinition() {
                  AttributeName = "CustomerId", AttributeType = "S"
              } },
            { new AttributeDefinition() {
                  AttributeName = "OrderId", AttributeType = "N"
              } },
            // Attribute definitions for index primary key
            { new AttributeDefinition() {
                  AttributeName = "OrderCreationDate", AttributeType = "N"
              } },
            { new AttributeDefinition() {
                  AttributeName = "IsOpen", AttributeType = "N"
              }}
        };

            createTableRequest.AttributeDefinitions = attributeDefinitions;

            // Key schema for table
            var tableKeySchema = new List<KeySchemaElement>()
        {
            { new KeySchemaElement() {
                  AttributeName = "CustomerId", KeyType = "HASH"
              } },                                                    //Partition key
            { new KeySchemaElement() {
```

```
                AttributeName = "OrderId", KeyType = "RANGE"
              } }                                                    //Sort key
        };

            createTableRequest.KeySchema = tableKeySchema;

            var localSecondaryIndexes = new List<LocalSecondaryIndex>();

            // OrderCreationDateIndex
            LocalSecondaryIndex orderCreationDateIndex = new LocalSecondaryIndex()
            {
                IndexName = "OrderCreationDateIndex"
            };

            // Key schema for OrderCreationDateIndex
            var indexKeySchema = new List<KeySchemaElement>()
        {
            { new KeySchemaElement() {
                  AttributeName = "CustomerId", KeyType = "HASH"
              } },                                                       //Partition key
            { new KeySchemaElement() {
                  AttributeName = "OrderCreationDate", KeyType = "RANGE"
              } }                                                                 //Sort key
        };

            orderCreationDateIndex.KeySchema = indexKeySchema;

            // Projection (with list of projected attributes) for
            // OrderCreationDateIndex
            var projection = new Projection()
            {
                ProjectionType = "INCLUDE"
            };

            var nonKeyAttributes = new List<string>()
        {
            "ProductCategory",
            "ProductName"
        };
            projection.NonKeyAttributes = nonKeyAttributes;

            orderCreationDateIndex.Projection = projection;

            localSecondaryIndexes.Add(orderCreationDateIndex);
```

```
            // IsOpenIndex
            LocalSecondaryIndex isOpenIndex
                = new LocalSecondaryIndex()
                {
                    IndexName = "IsOpenIndex"
                };

            // Key schema for IsOpenIndex
            indexKeySchema = new List<KeySchemaElement>()
        {
            { new KeySchemaElement() {
                  AttributeName = "CustomerId", KeyType = "HASH"
              }},                                                    //Partition key
            { new KeySchemaElement() {
                  AttributeName = "IsOpen", KeyType = "RANGE"
              }}                                                  //Sort key
        };

            // Projection (all attributes) for IsOpenIndex
            projection = new Projection()
            {
                ProjectionType = "ALL"
            };

            isOpenIndex.KeySchema = indexKeySchema;
            isOpenIndex.Projection = projection;

            localSecondaryIndexes.Add(isOpenIndex);

            // Add index definitions to CreateTable request
            createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

            Console.WriteLine("Creating table " + tableName + "...");
            client.CreateTable(createTableRequest);
            WaitUntilTableReady(tableName);
        }

        public static void Query(string indexName)
        {

Console.WriteLine("\n*********************************************************\n");
            Console.WriteLine("Querying table " + tableName + "...");
```

```
            QueryRequest queryRequest = new QueryRequest()
            {
                TableName = tableName,
                ConsistentRead = true,
                ScanIndexForward = true,
                ReturnConsumedCapacity = "TOTAL"
            };


            String keyConditionExpression = "CustomerId = :v_customerId";
            Dictionary<string, AttributeValue> expressionAttributeValues = new
Dictionary<string, AttributeValue> {
            {":v_customerId", new AttributeValue {
                 S = "bob@example.com"
             }}
       };


            if (indexName == "IsOpenIndex")
            {
                Console.WriteLine("\nUsing index: '" + indexName
                          + "': Bob's orders that are open.");
                Console.WriteLine("Only a user-specified list of attributes are
returned\n");
                queryRequest.IndexName = indexName;

                keyConditionExpression += " and IsOpen = :v_isOpen";
                expressionAttributeValues.Add(":v_isOpen", new AttributeValue
                {
                    N = "1"
                });

                // ProjectionExpression
                queryRequest.ProjectionExpression = "OrderCreationDate,
ProductCategory, ProductName, OrderStatus";
            }
            else if (indexName == "OrderCreationDateIndex")
            {
                Console.WriteLine("\nUsing index: '" + indexName
                          + "': Bob's orders that were placed after 01/31/2013.");
                Console.WriteLine("Only the projected attributes are returned\n");
                queryRequest.IndexName = indexName;

                keyConditionExpression += " and OrderCreationDate > :v_Date";
```

```
                expressionAttributeValues.Add(":v_Date", new AttributeValue
                {
                    N = "20130131"
                });

                // Select
                queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
            }
            else
            {
                Console.WriteLine("\nNo index: All of Bob's orders, by OrderId:\n");
            }
            queryRequest.KeyConditionExpression = keyConditionExpression;
            queryRequest.ExpressionAttributeValues = expressionAttributeValues;

            var result = client.Query(queryRequest);
            var items = result.Items;
            foreach (var currentItem in items)
            {
                foreach (string attr in currentItem.Keys)
                {
                    if (attr == "OrderId" || attr == "IsOpen"
                        || attr == "OrderCreationDate")
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].N);
                    }
                    else
                    {
                        Console.WriteLine(attr + "---> " + currentItem[attr].S);
                    }
                }
                Console.WriteLine();
            }
            Console.WriteLine("\nConsumed capacity: " +
result.ConsumedCapacity.CapacityUnits + "\n");
        }

        private static void DeleteTable(string tableName)
        {
            Console.WriteLine("Deleting table " + tableName + "...");
            client.DeleteTable(new DeleteTableRequest()
            {
                TableName = tableName
            });
```

```
            WaitForTableToBeDeleted(tableName);
        }

        public static void LoadData()
        {
            Console.WriteLine("Loading data into table " + tableName + "...");

            Dictionary<string, AttributeValue> item = new Dictionary<string,
AttributeValue>();

            item["CustomerId"] = new AttributeValue
            {
                S = "alice@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "1"
            };
            item["IsOpen"] = new AttributeValue
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130101"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Book"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "The Great Outdoors"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "PACKING ITEMS"
            };
            /* no ShipmentTrackingId attribute */
            PutItemRequest putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
```

```
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "alice@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "2"
            };
            item["IsOpen"] = new AttributeValue
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130221"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Bike"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Super Mountain"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "ORDER RECEIVED"
            };
            /* no ShipmentTrackingId attribute */
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
```

```
            S = "alice@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "3"
        };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue
        {
            N = "20130304"
        };
        item["ProductCategory"] = new AttributeValue
        {
            S = "Music"
        };
        item["ProductName"] = new AttributeValue
        {
            S = "A Quiet Interlude"
        };
        item["OrderStatus"] = new AttributeValue
        {
            S = "IN TRANSIT"
        };
        item["ShipmentTrackingId"] = new AttributeValue
        {
            S = "176493"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
        {
            S = "bob@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "1"
        };
```

```
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130111"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Movie"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Calm Before The Storm"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "SHIPPING DELAY"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "859323"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "2"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130124"
            };
            item["ProductCategory"] = new AttributeValue
```

```
            {
                S = "Music"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "E-Z Listening"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "DELIVERED"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "756943"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "3"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130221"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Music"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Symphony 9"
```

```
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "DELIVERED"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "645193"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);

            item = new Dictionary<string, AttributeValue>();
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "4"
            };
            item["IsOpen"] = new AttributeValue
            {
                N = "1"
            };
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130222"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Hardware"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "Extra Heavy Hammer"
            };
            item["OrderStatus"] = new AttributeValue
            {
```

```
            S = "PACKING ITEMS"
        };
        /* no ShipmentTrackingId attribute */
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
        {
            S = "bob@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "5"
        };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue
        {
            N = "20130309"
        };
        item["ProductCategory"] = new AttributeValue
        {
            S = "Book"
        };
        item["ProductName"] = new AttributeValue
        {
            S = "How To Cook"
        };
        item["OrderStatus"] = new AttributeValue
        {
            S = "IN TRANSIT"
        };
        item["ShipmentTrackingId"] = new AttributeValue
        {
            S = "440185"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
```

```
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
        item["CustomerId"] = new AttributeValue
        {
            S = "bob@example.com"
        };
        item["OrderId"] = new AttributeValue
        {
            N = "6"
        };
        /* no IsOpen attribute */
        item["OrderCreationDate"] = new AttributeValue
        {
            N = "20130318"
        };
        item["ProductCategory"] = new AttributeValue
        {
            S = "Luggage"
        };
        item["ProductName"] = new AttributeValue
        {
            S = "Really Big Suitcase"
        };
        item["OrderStatus"] = new AttributeValue
        {
            S = "DELIVERED"
        };
        item["ShipmentTrackingId"] = new AttributeValue
        {
            S = "893927"
        };
        putItemRequest = new PutItemRequest
        {
            TableName = tableName,
            Item = item,
            ReturnItemCollectionMetrics = "SIZE"
        };
        client.PutItem(putItemRequest);

        item = new Dictionary<string, AttributeValue>();
```

```
            item["CustomerId"] = new AttributeValue
            {
                S = "bob@example.com"
            };
            item["OrderId"] = new AttributeValue
            {
                N = "7"
            };
            /* no IsOpen attribute */
            item["OrderCreationDate"] = new AttributeValue
            {
                N = "20130324"
            };
            item["ProductCategory"] = new AttributeValue
            {
                S = "Golf"
            };
            item["ProductName"] = new AttributeValue
            {
                S = "PGA Pro II"
            };
            item["OrderStatus"] = new AttributeValue
            {
                S = "OUT FOR DELIVERY"
            };
            item["ShipmentTrackingId"] = new AttributeValue
            {
                S = "383283"
            };
            putItemRequest = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
                ReturnItemCollectionMetrics = "SIZE"
            };
            client.PutItem(putItemRequest);
        }

        private static void WaitUntilTableReady(string tableName)
        {
            string status = null;
            // Let us wait until table is created. Call DescribeTable.
            do
            {
```

```
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

                Console.WriteLine("Table name: {0}, status: {1}",
                          res.Table.TableName,
                          res.Table.TableStatus);
                status = res.Table.TableStatus;
            }
            catch (ResourceNotFoundException)
            {
                // DescribeTable is eventually consistent. So you might
                // get resource not found. So we handle the potential exception.
            }
        } while (status != "ACTIVE");
    }

    private static void WaitForTableToBeDeleted(string tableName)
    {
        bool tablePresent = true;

        while (tablePresent)
        {
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

                Console.WriteLine("Table name: {0}, status: {1}",
                          res.Table.TableName,
                          res.Table.TableStatus);
            }
            catch (ResourceNotFoundException)
            {
                tablePresent = false;
            }
        }
```

```
        }
    }
}
```

DynamoDB AWS CLI でのローカルセカンダリインデックスの操作

AWS CLI を使用して、1 つ以上のローカルセカンダリインデックスを含む Amazon DynamoDB テーブルを作成し、テーブルのインデックスを記述し、インデックスを使用してクエリを実行できます。

トピック

- ローカルセカンダリインデックスを持つテーブルを作成する
- ローカルセカンダリインデックスを持つテーブルの説明
- ローカルセカンダリインデックスのクエリ

ローカルセカンダリインデックスを持つテーブルを作成する

ローカルセカンダリインデックスは、テーブルの作成と同時に作成する必要があります。これを行うには、create-table パラメータを使用し、1 つ以上のローカルセカンダリインデックスの仕様を指定します。次の例では、ミュージックコレクション内の曲に関する情報を保持するためのテーブル (Music) を作成しています。パーティションキーは Artist で、ソートキーは SongTitle です。AlbumTitle 属性に関するセカンダリインデックス AlbumTitleIndex は、アルバムタイトルによるクエリを容易にします。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions AttributeName=Artist,AttributeType=S
 AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S  \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[{\"IndexName\": \"AlbumTitleIndex\",
        \"KeySchema\":[{\"AttributeName\":\"Artist\",\"KeyType\":\"HASH\"},
                      {\"AttributeName\":\"AlbumTitle\",\"KeyType\":\"RANGE\"}],
        \"Projection\":{\"ProjectionType\":\"INCLUDE\",  \"NonKeyAttributes\":[\"Genre
\", \"Year\"]}}]"
```

DynamoDB がテーブルを作成し、テーブルのステータスを ACTIVE に設定するまで待機する必要があります。その後、テーブルへのデータ項目の入力を開始できます。describe-table を使用して、テーブル作成のステータスを判断できます。

ローカルセカンダリインデックスを持つテーブルの説明

テーブルのローカルセカンダリインデックスに関する情報を取得するには、describe-table パラメータを使用します。インデックスごとに、名前、キースキーマ、および射影された属性にアクセスできます。

```
aws dynamodb describe-table --table-name Music
```

ローカルセカンダリインデックスのクエリ

テーブルに query を実行するのとほぼ同じ方法で、ローカルセカンダリインデックスに対する query オペレーションを使用することができます。インデックス名、インデックスソートキーのクエリ条件、および返す属性を指定する必要があります。この例では、インデックスは AlbumTitleIndex、インデックスソートキーは AlbumTitle です。

返される属性は、インデックスに射影された属性だけです。このクエリを変更して非キー属性を選択することもできますが、これには比較的コストのかかるテーブルフェッチアクティビティが必要です。テーブルのフェッチの詳細については、「属性の射影」を参照してください。

```
aws dynamodb query \
    --table-name Music \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v_artist and AlbumTitle = :v_title" \
    --expression-attribute-values  '{":v_artist":{"S":"Acme Band"},":v_title":
{"S":"Songs About Life"} }'
```

DynamoDB でのベクトルインデックスの使用

ベクトルインデックスは Amazon DynamoDB のインデックスの一種で、テーブルの項目に保存されているベクトル埋め込みの類似度検索を有効にします。Query および Scan オペレーションを使用した完全一致クエリや範囲クエリをサポートするグローバルセカンダリインデックスやローカルセカンダリインデックスとは異なり、ベクトルインデックスは近似最近傍 (ANN) 検索を使用して、指定したクエリベクトルに最も近いベクトルを持つ項目を検索します。これらの類似度検索は、類似度スコアでランク付けされた最も類似した項目を返す SearchVectors API を呼び出して実行します。

ベクトルインデックスを使用すると、ベクトル埋め込みを運用データと一緒に DynamoDB に保存し、別のベクトルデータベースを必要とせずに類似度検索を実行できます。これにより、DynamoDB と外部ベクトルストア間の複雑なデータレプリケーションパイプラインが不要になります。

ベクトルインデックスは、`VectorIndexes` パラメータ (`CreateTable` の場合) と `VectorIndexUpdates` パラメータ (`UpdateTable` の場合) を使用して、既に使用しているのと同じ `CreateTable` および `UpdateTable` API を通じて管理されます。

トピック

- ベクトルインデックスのユースケース
- ベクトルインデックスとセカンダリインデックスの比較
- 距離関数
- 距離関数による結果のランク付け
- SearchSchema
- 計画
- ベクトルインデックスを他の DynamoDB 機能と併用する
- ベクトルインデックスの作成と検索
- 要件と制限
- セキュリティとアクセスコントロール
- チュートリアル: 初めてのベクトル検索
- テーブルとベクトルインデックス間のデータ同期
- ベクトルインデックスのベストプラクティス
- ベクトルインデックスのストレージに関する考慮事項
- ベクトルインデックスキャパシティのモニタリング
- ベクトルインデックスのトラブルシューティング
- グローバルテーブルでのベクトルインデックスの使用

ベクトルインデックスのユースケース

ベクトルインデックスは、ベクトル表現に基づいて類似した項目を検索するさまざまなユースケースをサポートします。

- セマンティック検索 – キーワードの一致ではなく、クエリの意味を理解する検索エンジンを構築します。機械学習モデルによって生成されたテキスト埋め込みを保存し、意味的に類似したコンテンツを見つけます。
- 検索拡張生成 (RAG) – 大規模言語モデル (LLM) を関連するナレッジベースに接続します。ドキュメントの埋め込みを DynamoDB に保存し、LLM プロンプトに最も関連性の高いコンテキストを取得します。
- レコメンデーションシステム – 特徴や行動のベクトル表現に基づいて、類似した製品、コンテンツ、またはユーザーを検索します。
- AI エージェントメモリ – 会話の埋め込みを保存してセッション間のコンテキストを維持し、AI エージェントのパフォーマンスを向上させます。
- 異常検出および不正検出 – 新しいイベントを既知の通常の動作の埋め込みと比較して、異常なトランザクションや不正行為などの外れ値にフラグを付けます。

ベクトルインデックスとセカンダリインデックスの比較

次の表は、ベクトルインデックスとグローバルセカンダリインデックスおよびローカルセカンダリインデックスを比較したものです。

機能	ベクトルインデックス	グローバルセカンダリインデックス	ローカルセカンダリインデックス
クエリタイプ	類似度検索	完全一致と範囲	完全一致と範囲
API の読み取り	`SearchVectors`	`Query`, `Scan`	`Query`, `Scan`
スキーマ	ベクトル属性とオプションの SearchSchema (パーティションキー、インラインフィルター)	パーティションキー (複数可) とオプションのソートキー (複数可)	同じパーティションキー、異なるソートキー
テーブルあたりの最大数	5	20	5

機能	ベクトルインデックス	グローバルセカンダリインデックス	ローカルセカンダリインデックス
キャパシティモード	オンデマンドのみ	オンデマンドまたはプロビジョニング済み	オンデマンドまたはプロビジョニング済み

距離関数

ベクトルインデックスを作成するときは、距離関数を選択します。距離関数は、DynamoDB がベクトル間の類似性を測定する方法を決定します。選択した距離関数は、ランキングの品質と検索の精度に影響します。Amazon DynamoDB は 3 つの距離関数をサポートしています。

距離関数	スコア解釈	ベストマッチ
`COSINE`	スコアが低いほど類似度が高いことを示します。2 つのベクトル間のコサイン距離 (1 からコサイン類似度を引いた値) を測定します。値の範囲は 0 (同一方向) から 2 (反対方向) です。	最小スコア
`DOT_PRODUCT`	スコアが高いほど類似度が高いことを示します。2 つのベクトル間のドット積を測定します。	最高スコア
`EUCLIDEAN`	スコアが低いほど類似度が高いことを示します。2 つのベクトル間の直線距離を測定します。	最小スコア

距離関数が同じクエリベクトルをどのようにランク付けするかを比較するには、「the section called “距離関数による結果のランク付け”」を参照してください。

次のガイダンスは、ワークロードに適した距離関数を選択するのに役立ちます。

COSINE

方向を比較し、大きさを無視します。テキスト埋め込みモデルを用いたセマンティック類似性には COSINE を使用します。これらのモデルは意味を方向によってエンコードし、ベクトルの長さは異なる場合があります。例としては、Amazon Titan Text Embeddings や Cohere Embed などがあります。

COSINE は、以下のユースケースに最適です。

- 製品の説明またはドキュメントに対するセマンティック検索
- 検索拡張生成 (RAG)
- よくある質問のマッチング

COSINE は、使用する関数が不明な場合の安全なデフォルトです。

DOT_PRODUCT

DOT_PRODUCT は、方向と大きさ (ベクトルの長さ) の両方に影響を受けます。埋め込みモデルのドキュメントで類似度指標としてドット積を推奨している場合、またはベクトルの長さをランキングに影響させたい場合は、DOT_PRODUCT を選択します。

DOT_PRODUCT は、以下のユースケースに最適です。

- 人気スコアまたは信頼スコアを使用して埋め込みをスケールし、ランキングに影響を与えるレコメンデーションシステム
- ドキュメントにおいて、類似度指標としてドット積が特に推奨されているモデル
- ベクトル長が意味のあるシグナルを伝達する、大きさの影響を受けるランキング

埋め込みを単位長さに正規化することをお勧めします。正規化すると、DOT_PRODUCT は COSINE と同じ方法で結果をランク付けします。大きさがランキングに影響する場合にのみ正規化をスキップします。

ユースケースの例: 各製品の埋め込みを、人気度スコアでスケールする製品レコメンデーションシステム。人気のある製品では、ベクトルが長くなり、検索結果のランクが高くなります。

EUCLIDEAN

2 つのベクトル間の直線距離を測定します。EUCLIDEAN は大きさに影響を受けます。埋め込みスペースの絶対位置が重要な場合に使用します。

EUCLIDEAN は、以下のユースケースに最適です。

- 空間距離が重要な画像または音声の埋め込み
- ほぼ重複しているデータの検出
- クラスタリングと異常検出

ユースケースの例: 画像の埋め込みからほぼ重複する画像を見つける。

> **埋め込みモデルに合った距離関数を選択する**
>
> 使用する関数が不明な場合は、埋め込みモデルのドキュメントを確認してください。代表的なデータセットを用いて、その選択を検証します。インデックスの作成後に距離関数を変更することはできません。詳細については、「the section called “距離関数を埋め込みに一致させる”」を参照してください。

距離関数による結果のランク付け

インデックスの作成時に選択する距離関数によって、Score 値と結果のソート順序の両方が決まります。同じクエリでも、異なる距離関数を使用すると、同じ項目のランク付けが異なる場合があります。次の例では、4 つの保存されたベクトルに対してクエリベクトル `[1, 0, 0, 0]` を使用します。

保存されたベクトル	**COSINE** (値が低いほど類似度が高い)	**EUCLIDEAN** (値が低いほど類似度が高い)	**DOT_PRODUCT** (値が高いほど類似度が高い)
`[1, 0, 0, 0]`	0.0	0.0	1.0
`[10, 0, 0, 0]`	0.0	9.0	10.0
`[0.7071, 0.7071, 0, 0]`	0.29	0.77	0.71
`[-1, 0, 0, 0]`	2.0	2.0	-1.0

次の 2 つの動作に注意してください。

- COSINE は大きさを無視します。同じ方向を指すため、[1, 0, 0, 0] と [10, 0, 0, 0] を同じようにスコアリングします (両方とも 0.0)。EUCLIDEAN は、ベクトルの大きさに応じて増加する絶対距離を測定するため、同じクエリに対して [10, 0, 0, 0] を最下位にランク付けします。
- DOT_PRODUCT のスコアは負の値になる場合があります。反対方向を指すベクトル ([-1, 0, 0, 0]) のスコアは -1.0 です。結果をソートしたり、しきい値を適用したりするときに、スコアが常に負でないと想定しないでください。

SearchSchema

ベクトルインデックスを作成するときに、オプションでベクトルインデックスのパーティションキーとインラインフィルター属性を指定する SearchSchema を定義できます。

HASH (ベクトルインデックスのパーティションキー)

ベクトルインデックスパーティションキーは、選択した属性に対して同じ値を共有するベクトルに各検索のスコープを限定します。同じパーティションキー値を持つ項目は一緒に保存されるため、検索はインデックス全体をスキャンするのではなく、そのサブセットのみを読み取ります。大規模な場合、ベクトルインデックスパーティションキーを使用すると、検索はベクトルスペースのサブセットのみを調べるため、検索レイテンシーが短縮されます。Category や Country など、低〜中程度のカーディナリティの属性を使用します。最大 1 つのベクトルインデックスのパーティションキーを指定できます。

SearchSchema でベクトルインデックスのパーティションキーを定義する場合は、SearchVectors を呼び出すときに SearchConditionExpression でその値を指定する必要があります。

> **パーティションキーを使用して検索スループットをスケールする**
>
> 大規模なインデックスまたは検索ボリュームが多いことが予想される場合は、ベクトルインデックスのパーティションキーを定義します。各 SearchVectors 呼び出しは単一のパーティションキー値にスコープされているため、多くのパーティションキー値にデータを分散することで、1 秒あたりにより多くの検索オペレーションを実行し、各検索が調べるデータの量を減らすことができます。「the section called “クエリパターンに合ったパーティションキーを選択する”」を参照してください。

INLINE_FILTER

インラインフィルター属性は、DynamoDB が各ベクトルの隣のベクトルインデックスに値を保存する、通常の非ベクトル属性です。値はベクトルと共に保存されるため、DynamoDB はインデックスを検索する際にフィルターに一致しないベクトルをスキップします。これにより、最初にインデックス全体を検索してから一致しない結果を破棄する必要がなくなります。

例えば、Language 属性をインラインフィルターとして定義すると、Language でフィルタリングする検索では、Language の値が一致するベクトルのみが考慮されます。

インラインフィルターは、SearchConditionExpression において等価演算子 (=) をサポートしています。比較、範囲、および集合メンバー演算子 (<>、<、<=、>、>=、IN) はまだ利用できません。ベクトルインデックスのパーティションキーとは異なり、インラインフィルターは検索時にオプションです。

SearchSchema でパーティションキーを定義せずにベクトルインデックスを作成できます。この場合、SearchVectors 呼び出しごとにインデックス全体を検索します。これは SearchConditionExpression が不要なため、よりシンプルです。ただし、各検索ではインデックス全体を調べるため、各検索の作業量、つまりレイテンシーとコストは、インデックスが大きくなるにつれて増加します。また、検索スループットを複数のパーティションキー値に分散させることもできません。ワークロードで高スループットが必要な場合、またはインデックスに多数のベクトルが含まれている場合は、パーティションキーを定義してパーティション間でデータを分散し、個別にスケーリングします。「the section called “クエリパターンに合ったパーティションキーを選択する”」を参照してください。

計画

グローバルセカンダリインデックスと同様に、ベクトルインデックスは、ベーステーブルのどの属性をインデックスにコピーするかを制御する射影をサポートします。ベクトルインデックスを作成するときに射影を指定します。

- KEYS_ONLY – ベーステーブルのプライマリキー属性、ベクトル属性、および SearchSchema で定義されたインラインフィルター属性のみがインデックスに射影されます。
- INCLUDE – KEYS_ONLY 属性に加えて、射影する追加の非キー属性を指定します。ベクトルインデックスの作成後に、含まれる属性のセットを変更することはできません。別の属性セットを射影するには、インデックスを削除し、必要な射影を指定して再作成します。
- ALL – ベーステーブルのすべての属性がインデックスに射影されます。

射影によって SearchVectors が返せる属性が制限される

ベクトルインデックスに射影されていない属性は、SearchVectors レスポンスで返されません。検索結果に特定の属性が必要な場合は、それらを射影に含めるか、ALL を使用します。

ベクトルインデックスを他の DynamoDB 機能と併用する

DynamoDB Streams

ベクトルインデックスを持つテーブルで DynamoDB Streams を有効にするには、テーブルの作成時に StreamSpecification パラメータを使用するか、UpdateTable を使用します。ストリームは、ベーステーブルに対する項目レベルの変更をキャプチャし、ベクトルインデックスとは独立して動作します。

グローバルテーブル

ベクトルインデックスをグローバルテーブルに追加できます。また、UpdateTable を使用してレプリカを追加することで、ベクトルインデックスを持つテーブルをグローバルテーブルに変換することもできます。ディメンション、距離関数、SearchSchema、射影を含むベクトルインデックス定義は、新しいレプリカリージョンごとに自動的にレプリケートされます。レプリカリージョンではベクトルインデックスを個別に作成しません。

レプリカリージョンで書き込んだ項目は、他のリージョンにレプリケートされ、そこにインデックスが作成されます。レプリケーションが完了すると、各リージョンの SearchVectors は同一のベクトルセットを検索します。ベクトル検索では近似最近傍 (ANN) が使用されるため、同一のデータであっても、異なるリージョンで同じクエリに対して個別に検索を行うと、結果や順序がわずかに異なる場合があります。リージョン間のテーブルレプリケーションは、マルチリージョン強力整合性 (MRSC) グローバルテーブルの強力な整合性を含む、グローバルテーブルの整合性保証に従います。ただし、ベクトルインデックス作成は、書き込んだリージョンを含むすべてのリージョンで、テーブルへの書き込みが適用された後、常に非同期的に行われます。MRSC 保証は、ベクトルインデックスではなくテーブルを対象としています。その結果、書き込みが確認されてから、そのベクトルが SearchVectors を通じて検索可能になるまでに遅延が発生します。この遅延は、ローカルおよびリージョン間の両方に適用されます。

> **オンデマンドキャパシティが必要**
>
> ベクトルインデックスはオンデマンドキャパシティモードのみを使用するため、オンデマンドキャパシティモードを使用するテーブルが必要です。2 つのキャパシティモードを混在させることはできません。グローバルテーブルはオンデマンドキャパシティモードをサポートしています。オンデマンドキャパシティを既に使用しているテーブルで、ベクトルインデックスとレプリカを作成します。

ポイントインタイムリカバリ (PITR) とバックアップ

ポイントインタイムリカバリまたはオンデマンドバックアップからテーブルを復元すると、DynamoDB はベーステーブルのデータとベクトルインデックス定義を復元します。グローバルセカンダリインデックスと同様に、DynamoDB はベクトルインデックスをバイト単位でコピーするのではなく、復元されたベーステーブルのデータから再構築するため、検索の準備が整う前にインデックスのバックフィルが行われます。`SearchVectors` を実行する前に、復元されたインデックスの `IndexStatus` が `ACTIVE` になり、`Backfilling` が `true` でなくなるまで待機します。

有効期限 (TTL)

ベクトルインデックスを持つテーブルで DynamoDB TTL を使用できます。TTL がベーステーブルから期限切れの項目を削除すると、DynamoDB は手動での削除と同様に、ベクトルインデックスから対応するエントリを削除します。したがって、期限切れの項目は、削除がインデックスに伝達されると `SearchVectors` の結果に表示されなくなります。

テーブルデータのインポートとエクスポート

ベクトルインデックスを持つテーブルを Amazon S3 にエクスポートできます。エクスポートには、格納されているベクトル属性を含むベーステーブルの項目が含まれます。Amazon S3 から新しいテーブルにデータをインポートするときは、`CreateTable` の場合と同様に、インポートリクエストでベクトルインデックスを定義します。DynamoDB は、インポートされた項目が書き込まれるとインデックスを作成し、インポートの完了後にベクトルインデックスが利用可能になります。

DAX

DynamoDB Accelerator (DAX) は、`SearchVectors` オペレーションをサポートしていません。アプリケーションが他の読み取りオペレーションに DAX を使用している場合でも、DynamoDB に直接 `SearchVectors` リクエストを送信します。ベーステーブル読み取りの DAX キャッシュ

は、ベクトルインデックスの存在の影響を受けません。ベクトル検索結果にキャッシュレイヤーが必要な場合は、Amazon ElastiCache for Valkey などの外部キャッシュに SearchVectors レスポンスをキャッシュできます。そのキャッシュへの入力と無効化はお客様の責任です。

ベクトルインデックスの作成と検索

このセクションでは、SearchVectors API を使用してベクトルインデックスを作成および管理し、ベクトルデータを含む項目の書き込み、類似度検索を実行する方法について説明します。

トピック

- 開始する前に
- SearchVectors エンドポイント
- ベクトル埋め込みの生成
- ベクトルインデックスの作成
- ベクトルデータを含む項目の書き込み
- SearchVectors を使用した検索
- ベクトルインデックスの削除

開始する前に

ベクトルインデックスを使用する前に、以下を確認してください。

- テーブルはオンデマンドキャパシティモード (PAY_PER_REQUEST) を使用しています。ベクトルインデックスはオンデマンドキャパシティモードのみを使用し、同じくオンデマンドキャパシティモードを使用するテーブルが必要です。そのため、2 つのキャパシティモードを混在させることはできません。
- AWS Identity and Access Management (IAM) ID には、ベクトルインデックスを作成するための dynamodb:CreateTable または dynamodb:UpdateTable アクセス許可があります。
- IAM ID には、検索を実行するためのベクトルインデックスリソースに対する dynamodb:SearchVectors アクセス許可があります。リソース ARN 形式は arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*index-name* です。

SearchVectors エンドポイント

SearchVectors リクエストは専用ベクトル検索エンドポイントを使用します。これは、ベクトルインデックスの作成と管理 (CreateTable、UpdateTable、DescribeTable など) に使用する標準の DynamoDB エンドポイントとは異なります。AWS SDK と AWS CLI は、SearchVectors リクエストを正しいエンドポイントに自動的にルーティングします。アプリケーションコードでエンドポイントを設定または上書きする必要はありません。

AWS SDK を使用せずに DynamoDB API を直接呼び出すカスタム HTTP クライアントを構築する場合は、*account-id* と *region* を適切に置き換えて、次のいずれかのベクトル検索エンドポイントを使用します。

- *account-id*.search-ddb.*region*.amazonaws.com – アカウントベースのエンドポイント。
- search-dynamodb.*region*.api.aws – IPv4 と IPv6 の両方と互換性のあるデュアルスタックエンドポイント。

ベクトル埋め込みの生成

DynamoDB はベクトル埋め込みを保存して検索しますが、生成は行いません。Amazon Bedrock Titan Text Embeddings や Cohere Embed モデルなどの埋め込みモデル、またはお客様が運用する埋め込みモデルを使用して埋め込みを生成します。次に、生成されたベクトルを DynamoDB 項目に保存し、クエリベクトルを SearchVectors に渡します。

> ⚠ **クエリベクトルと保存済みベクトルは、同じモデルを使用する必要があります**
>
> SearchVectors に渡すクエリベクトルと項目に保存されるベクトルは、同じ埋め込みモデルによって生成され、ベクトルインデックスと同じ数のディメンションを持つ必要があります。モデルを混在させたり、インデックス作成時とは異なるディメンション数でクエリを実行すると、意味のない結果が生じるか、検証エラーが発生します。

一般的なフローは次のとおりです。

1. ソースコンテンツ (製品説明など) を埋め込みモデルに送信し、ベクトルを受け取ります。
2. そのベクトルを、ベクトルインデックス (VectorAttribute) で命名された属性に、数値 (N) のリスト (L) として DynamoDB 項目に保存します。
3. クエリ時に、同じモデルを使用して検索テキストからベクトルを生成し、SearchVector として渡します。

> **インデックスを作成する前に埋め込みモデルを選択する**
>
> ベクトルインデックスを作成する前に埋め込みモデルを選択してください。モデルによってディメンション数が決まります。一般的な埋め込みモデルでは、384、768、1024、1536、または 3072 ディメンションが生成されます。DynamoDB は最大 4,096 ディメンションをサポートします。「the section called “要件と制限”」を参照してください。

選択する距離関数は、モデルが埋め込みを生成する方法と相互作用します。COSINE は方向を比較し、大きさを無視するため、正規化されているかどうかにかかわらず埋め込みで機能します。DOT_PRODUCT は大きさの影響を受けます。埋め込みが単位長に正規化されていない場合、より大きなベクトルは方向に関係なくより高いスコアを得ることになります。DOT_PRODUCT を使用し、かつ方向ベースの類似性が必要な場合は、埋め込みを保存する前に単位長に正規化します。「the section called “距離関数による結果のランク付け”」を参照してください。

ベクトルインデックスの作成

ベクトルインデックスは、新しいテーブルを作成するときに作成することも、既存のテーブルに追加することもできます。

ベクトルインデックスを持つテーブルの作成

VectorIndexes パラメータを指定して CreateTable API を使用し、ベクトルインデックスを持つテーブルを作成します。次の AWS CLI 例では、ProductEmbeddingIndex という名前のベクトルインデックスを持つ Products テーブルを作成します。

```
aws dynamodb create-table \
    --table-name Products \
    --attribute-definitions AttributeName=ProductId,AttributeType=S \
                            AttributeName=Category,AttributeType=S \
                            AttributeName=Brand,AttributeType=S \
    --key-schema AttributeName=ProductId,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --vector-indexes \
        "[
            {
                \"IndexName\": \"ProductEmbeddingIndex\",
                \"VectorAttribute\": {\"AttributeName\": \"Embedding\"},
                \"SearchSchema\": [{\"AttributeName\":\"Category\",
\"SearchSchemaElementType\":\"HASH\"},
```

```
                                    {\"AttributeName\":\"Brand\",
\"SearchSchemaElementType\":\"INLINE_FILTER\"}],
                \"Projection\": {\"ProjectionType\": \"ALL\"},
                \"Dimensions\": 1536,
                \"DistanceFunction\": \"COSINE\"
            }
        ]"
```

この例では、以下のようになっています：

- VectorAttribute は、ベクトルデータを含む属性として Embedding を指定します。
- SearchSchema は、ベクトルインデックスのパーティションキー (HASH) として Category を定義します。これにより、スケーリングのためにカテゴリ別にインデックスがパーティション化されます。また、Brand を INLINE_FILTER として定義します。これにより、ストレージレイヤーでブランド別に検索結果をフィルタリングできます。Category と Brand の両方が SearchSchema で参照されるため、グローバルセカンダリインデックスに対してキー属性を宣言するのと同じ方法で、AttributeDefinitions でも宣言する必要があります。
- Dimensions は 1536 に設定され、一般的な埋め込みモデルの出力と一致します。
- DistanceFunction は COSINE に設定され、スコアが低いほど類似度が高いことを示します。

ベクトルインデックスの既存テーブルへの追加

VectorIndexUpdates パラメータを指定して UpdateTable API を使用して、既存のテーブルにベクトルインデックスを追加します。この例では、同じ Products テーブルに ProductEmbeddingIndexV2 という名前の 2 つ目の独立したインデックスを追加します。

```
aws dynamodb update-table \
    --table-name Products \
    --vector-index-updates \
        "[
            {
                \"Create\": {
                    \"IndexName\": \"ProductEmbeddingIndexV2\",
                    \"VectorAttribute\": {\"AttributeName\": \"Embedding\"},
                    \"Projection\": {\"ProjectionType\": \"ALL\"},
                    \"Dimensions\": 1536,
                    \"DistanceFunction\": \"EUCLIDEAN\"
                }
            }
```

```
        ]"
```

既存のテーブルにベクトルインデックスを追加すると、DynamoDB は DescribeTable レスポンスの 2 つのフィールド (IndexStatus 値と個別の Backfilling ブール値) を通じてインデックスの進行状況をレポートします。

1. IndexStatus が CREATING で、Backfilling は存在しないか false です。DynamoDB はインデックスインフラストラクチャをプロビジョニングしています。
2. IndexStatus は CREATING で、Backfilling は true です。DynamoDB はベーステーブルの既存のデータを使用してインデックスに入力しています。ベーステーブルへの新しい書き込みも、このフェーズ中にインデックスにレプリケートされます。SearchVectors は ValidationException を返します。このフェーズでは、ベーステーブルの項目数が非常に少ない場合でも、かなりの時間がかかる場合があります。
3. IndexStatus は ACTIVE であり、Backfilling は報告されなくなっています。インデックスは、すぐに検索できます。

インデックスのバックフィル中は検索できません

SearchVectors は、ベクトルインデックスのバックフィル中に ValidationException を返します。DescribeTable を使用して IndexStatus と Backfilling フラグの両方を確認し、IndexStatus が ACTIVE になり、Backfilling が true でなくなるまで待ってから検索します。BACKFILLING インデックスステータス値はありません。バックフィル中のインデックスは、Backfilling が true に設定された IndexStatus CREATING をレポートします。

新しく準備されたインデックスはすぐには検索できません

SearchVectors リクエストは、DescribeTable を提供するエンドポイントとは別の、専用の検索エンドポイントによって処理されます。DescribeTable が初めて IndexStatus を ACTIVE とレポートした後も、検索エンドポイントがインデックスの提供を開始するまでにさらに時間がかかる場合があります。その間、SearchVectors は ValidationException を返します。通常は The table does not have the specified index というメッセージが表示されます。

インデックス作成後の最初の検索で ValidationException が発生した場合は、失敗としてではなく再試行可能として扱います。ACTIVE を検出した直後に検索するコードは、ある環境では動作し、別の環境では失敗する可能性があります。この間隔は短く、変動します。どちらのステータスフィールドにも依存しない準備状況チェックを行う場合は、再試行ループで実際の SearchVectors リクエストを発行します。最初に成功したレスポンスを、インデックスの準備が完了したことを示すシグナルとして扱います。

一度に作成または削除できるベクトルインデックスは、テーブルごとに 1 つだけです。この制限は、同じテーブルのグローバルセカンダリインデックスの作成にも共有されます。別のインデックスオペレーションの進行中にインデックスを作成または削除する 2 番目の UpdateTable リクエストは、`LimitExceededException: Subscriber limit exceeded: Only 1 online index can be created or deleted simultaneously per table` で失敗します。

これは 1 つのリクエストにも適用されます。VectorIndexUpdates パラメータに 2 つの Create アクションを含む UpdateTable 呼び出しは、同じエラーで失敗します。ベクトルインデックスを 1 つずつ作成し、それぞれが ACTIVE に達するのを待ってから次を開始します。各インデックスは次が開始する前にバックフィルを完了する必要があるため、1 つのテーブルに複数のベクトルインデックスを構築するには、1 つを構築するよりもかなり時間がかかります。

CreateTable リクエストでは、テーブルあたりの上限である 5 つまで、複数のベクトルインデックスを一度に定義できます。この制限を超えると、`ValidationException: One or more parameter values were invalid: VectorIndex count exceeds the per-table limit of 5` でエラーが発生します。

ベクトルデータを含む項目の書き込み

標準の DynamoDB 書き込み API (PutItem、UpdateItem、BatchWriteItem、TransactWriteItems) を使用して、ベクトルデータを含む項目を書き込みます。ベクトル埋め込みを数値のリスト (N 要素を含む L 型) として保存します。

ベクトルには多くの値が含まれているため、項目を item.json などのファイルに保存し、そのファイルを AWS CLI に渡します。

```
{
    "ProductId": { "S": "prod-123" },
    "Category": { "S": "Electronics" },
    "Title": { "S": "Wireless Headphones" },
    "Embedding": {
```

```
        "L": [
            { "N": "0.1234" },
            { "N": "-0.5678" },
            { "N": "0.9012" },
            ...
        ]
    }
}
```

```
aws dynamodb put-item \
    --table-name Products \
    --item file://item.json
```

> **ベクトルの長さはインデックスのディメンションと一致する必要があります**
>
> ここに示す Embedding ベクトルは省略されています。item.json では、ProductEmbeddingIndex で設定した Dimensions と一致させるには、1,536 個の値が含まれている必要があります。ディメンション数が正しくないベクトルの書き込みは拒否されます。

DynamoDB は、ベクトルインデックスを持つテーブルに項目を書き込むときにベクトルデータを検証します。次の表は、検証動作の説明です。

条件	動作
ベクトル属性のディメンション数が正しくありません	書き込みは拒否されます。
ベクトルインデックスのパーティションキー属性がありません	ベーステーブルへの書き込みは成功しますが、その項目はベクトルインデックスにレプリケートされません。
ベクトルインデックスのパーティションキー属性の型がインデックススキーマと一致しません	書き込みは拒否されます。
インラインフィルター属性がありません	書き込みは成功し、項目はベクトルインデックスにレプリケートされます。

条件	動作
ベクトル値は 32 ビット浮動小数点 (f32) よりも高い精度を持ちます	書き込みが成功します。値はベーステーブルにそのまま保存されますが、ベクトルインデックスにレプリケートする際には精度が低下します。
ベクトル属性が項目から削除される	ベクトルインデックスの対応するエントリが削除されます。

⚠ パーティションキーがないと、サイレントなインデックス解除が発生します。

ベクトルインデックスが SearchSchema でパーティションキーを定義し、その属性なしで項目を書き込む (または UpdateItem で削除する) 場合、書き込みはベーステーブルで成功しますが、項目はベクトルインデックスからサイレントに除外されます。ベーステーブルの項目とそのベクトル埋め込みがまだ存在する場合でも、SearchVectors の結果には表示されません。検索対象とするすべての項目に、ベクトルインデックスのパーティションキー属性が含まれていることを確認します。

⚠ 古い埋め込みでは誤った結果が生成されます

DynamoDB は、埋め込みを自動的に再計算することはありません。埋め込みを生成したソースコンテンツを変更した場合 (製品の説明を編集した場合など)、保存されたベクトルは自動的には更新されません。埋め込みモデルを使用して埋め込みを再生成し、新しいベクトルを項目に書き戻す必要があります。そうしないと、ベクトルインデックスは古いベクトルに基づいて結果を返し続けるため、誤った一致がサイレントに発生する可能性があります。

SearchVectors を使用した検索

SearchVectors API を使用して、クエリベクトルに最も近い項目をベクトルインデックス内で検索します。結果は関連性に基づいてソートされ、最も類似した項目が最初に表示されます。検索結果は結果整合性に基づいています。ベクトルの書き込みまたは更新から検索結果に表示されるまでに、わずかな遅延が発生する場合があります。詳細については、「the section called “継続的な書き込み同期”」を参照してください。

基本的な検索

次の例では、ProductEmbeddingIndex インデックス内で最も類似している 10 個の項目を検索します。このインデックスには、SearchSchema でベクトルインデックスのパーティションキー (Category) が定義されているため、SearchConditionExpression にはベクトルインデックスのパーティションキー値を含める必要があります。

クエリベクトルを query-vector.json などのファイルに、数値のプレーン JSON 配列として保存します。

```
[
    { "N": "0.1234" },
    { "N": "-0.5678" },
    { "N": "0.9012" },
    ...
]
```

```
aws dynamodb search-vectors \
    --table-name Products \
    --index-name ProductEmbeddingIndex \
    --search-vector file://query-vector.json \
    --top-k 10 \
    --search-condition-expression "Category = :cat" \
    --expression-attribute-values "{\":cat\": {\"S\": \"Electronics\"}}"
```

レスポンスには SearchResults 配列が含まれます。各要素には、一致する Item と、項目がクエリベクトルとどの程度類似しているかを示す Score が含まれます。

```
{
    "SearchResults": [
        {
            "Item": {
                "ProductId": { "S": "prod-456" },
                "Category": { "S": "Electronics" },
                "Title": { "S": "Bluetooth Speaker" }
            },
            "Score": 0.0023
        },
        {
            "Item": {
                "ProductId": { "S": "prod-789" },
                "Category": { "S": "Electronics" },
```

```
                "Title": { "S": "Noise Cancelling Earbuds" }
            },
            "Score": 0.0145
        }
    ]
}
```

> **ベクトル属性はデフォルトで結果から除外されます**
>
> デフォルトでは、SearchVectors からの結果にはベクトル属性 (埋め込み) は含まれません。ベクトルデータは大きく、通常はレスポンスでは必要ありません。結果には、他の射影された属性と Score 値が含まれます。ベクトル属性を含めるには、ProjectionExpression でリクエストします。詳細については、「the section called "ProjectionExpression の使用"」を参照してください。

> **SearchVector はプレーンリストであり、DynamoDB L 型ではありません**
>
> SearchVector リクエストパラメータは、数値オブジェクトのプレーン JSON 配列 ([{"N": "0.1234"}, ...]) です。項目属性にベクトルを保存する場合のように、DynamoDB L 型でラップしないでください。L ラッパーは、項目属性でベクトルデータを書き込みまたは読み取る場合にのみ使用されます。

Score の意味は、インデックスの作成時に選択した距離関数によって異なります。COSINE と EUCLIDEAN の場合、スコアが低いほど類似度が高いことを示します。DOT_PRODUCT の場合、スコアが高いほど類似度が高いことを示します。

SearchConditionExpression を使用したフィルタリング

SearchConditionExpression を使用して、SearchSchema で定義されたベクトルインデックスのパーティションキーとインラインフィルター属性に基づいて検索結果をフィルタリングします。この式は、他の DynamoDB 式パラメータと同じ構文を使用します。

次の例では、Brand インラインフィルターを使用して Electronics カテゴリ (ベクトルインデックスのパーティションキー) 内の項目を検索します。

```
aws dynamodb search-vectors \
    --table-name Products \
    --index-name ProductEmbeddingIndex \
```

```
    --search-vector file://query-vector.json \
    --top-k 10 \
    --search-condition-expression "Category = :cat AND Brand = :brand" \
    --expression-attribute-values "{\":cat\": {\"S\": \"Electronics\"}, \":brand\":
{\"S\": \"Acme\"}}"
```

ベクトルインデックスに SearchSchema でパーティションキーが定義されている場合は、SearchConditionExpression に含める必要があります。インラインフィルター属性はオプションです。

SearchConditionExpression において、等価演算子 (=) は、ベクトルインデックスのパーティションキーとインラインフィルター属性の両方でサポートされています。比較、範囲、および集合メンバー演算子 (<>、<、<=、>、>=、IN) はまだ利用できません。

このフィルタリングは、類似度検索の範囲をデータのサブセットに絞り込むための手法であり、マルチテナントおよび検索拡張生成 (RAG) アプリケーションの一般的な要件です。例えば、クエリに似ているドキュメントを特定のテナント内でのみ検索するには、テナント属性を SearchSchema のベクトルインデックスのパーティションキー (HASH) として定義し、検索のたびにその値を渡します。これにより、結果をそのテナントのみに限定できます。また、関連するデータのみが検索されるため、パフォーマンスも向上します。ルーティングされたパーティション内に適用するドキュメントタイプやステータスなど、追加の等価制約にはインラインフィルター属性を使用します。

> ⚠ パーティションキーの範囲設定はセキュリティ境界ではない
>
> パーティションキーを使用して検索の範囲を単一のテナントに絞り込むのは、アクセスコントロールメカニズムではなく、データの局所性とパフォーマンス最適化のための手法です。インデックスに対する dynamodb:SearchVectors アクセス許可を持つプリンシパルは、任意のパーティションキー値を検索できます。dynamodb:LeadingKeys などのきめ細かなアクセスコントロール (FGAC) 条件キーは SearchVectors には適用されないため、IAM ポリシーレベルで個々のパーティションキー値へのアクセスを制限することはできません。ワークロードがデータレイヤーで厳密なテナント分離を必要とする場合は、テナントごとに個別の IAM 許可を持つ個別のテーブルまたはインデックスを使用します。

ProjectionExpression の使用

ProjectionExpression を使用して、検索結果内の特定の属性のみを返します。これにより、すべての射影された属性を必要としない場合にレスポンスサイズを削減できます。ProductEmbeddingIndex は SearchSchema でベクトルインデックスのパーティションキー

(Category) を定義するため、この例でも SearchConditionExpression にベクトルインデックスのパーティションキー値が含まれています。

```
aws dynamodb search-vectors \
    --table-name Products \
    --index-name ProductEmbeddingIndex \
    --search-vector file://query-vector.json \
    --top-k 5 \
    --search-condition-expression "Category = :cat" \
    --expression-attribute-values "{\":cat\": {\"S\": \"Electronics\"}}" \
    --projection-expression "ProductId, Title"
```

> **射影された属性のみ返すことができます**
>
> ベクトルインデックスに射影された属性のみを返すことができます。インデックス射影にない属性は、SearchVectors によって返されません。

ベクトルインデックスの削除

VectorIndexUpdates パラメータを指定して UpdateTable API を使用し、ベクトルインデックスを削除します。

```
aws dynamodb update-table \
    --table-name Products \
    --vector-index-updates \
        "[
            {\"Delete\": {\"IndexName\": \"ProductEmbeddingIndex\"}}
        ]"
```

ベクトルインデックスを削除すると、DynamoDB はインデックスとそのすべてのデータを削除します。このオペレーションは、ベーステーブルまたはその項目には影響しません。

要件と制限

ベクトルインデックスを使用する場合は、次の要件と制限に注意してください。

- ベクトルインデックスはオンデマンドキャパシティモードのみを使用するため、オンデマンドキャパシティモードを使用するテーブルが必要です。2 つのキャパシティモードを混在させることはできません。

- ベクトル埋め込みは、32 ビット浮動小数点 (f32) の精度でインデックスに保存されます。より精度の高い値も受け入れられますが、インデックスにレプリケートされる際には精度が低下します。
- SearchVectors API では、きめ細かなアクセスコントロール (FGAC) はサポートされていません。
- SearchVectors レスポンスは 16 MB に制限されています。ページ分割はサポートされません。各結果には射影された属性が含まれているため、大きな項目に対して ProjectionType: ALL を指定し、かつ TopK の値を大きく設定すると、この制限に達する可能性があります。項目が大きい場合は、より狭い射影を使用するか、TopK の値を小さくしてください。
- ベクトルインデックスは、Query または Scan オペレーションをサポートしていません。SearchVectors API を使用してベクトルインデックスから読み取ります。
- ベクトルインデックスには、PartiQL を介してアクセスすることはできません。SearchVectors API を使用して類似度検索を実行します。
- ベクトルインデックスは、すべての商用 AWS リージョン、AWS GovCloud (US) リージョン、中国リージョンで使用できます。

最大ディメンション、TopK 範囲、インデックスあたりのインラインフィルター、テーブルあたりのインデックス、ベーステーブルサイズのしきい値などの数値制限については、「Amazon DynamoDB のクォータ」の「the section called “ベクトルインデックス”」を参照してください。

セキュリティとアクセスコントロール

IAM ポリシーは、ベクトルインデックスオペレーションへのアクセスを制御します。次のアクセス許可が適用されます。

- ベクトルインデックスの作成と削除 – テーブルリソースに対する dynamodb:CreateTable または dynamodb:UpdateTable アクセス許可が必要です。ベクトルインデックス管理に追加のアクセス許可は必要ありません。
- ベクトルインデックスの検索 – インデックスリソースに対する dynamodb:SearchVectors アクセス許可が必要です。リソース ARN 形式は arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*index-name* です。
- ベクトルを含む項目の書き込み – 標準の書き込みオペレーション (dynamodb:PutItem、dynamodb:UpdateItem) と同じアクセス許可を使用します。ベクトルデータ自体に追加のアクセス許可は必要ありません。

> **⚠ FGAC 条件キーは SearchVectors には適用されません**
>
> SearchVectors API で Amazon DynamoDB のきめ細かなアクセスコントロール (FGAC) を使用することはできません。FGAC を適用する dynamodb: IAM 条件コンテキストキー (dynamodb:LeadingKeys、dynamodb:Attributes、dynamodb:Select など) は、SearchVectors には影響しません。つまり、これらを使用して、検索できる項目または属性を制限することはできません。代わりに、インデックスリソース ARN に対して dynamodb:SearchVectors アクションを付与して、インデックスレベルでアクセスを制御します。

これらの条件キーは SearchVectors リクエストコンテキストに存在しないため、いずれかの条件を参照するポリシーステートメントは SearchVectors リクエストと一致せず、DynamoDB はアクセスを許可せずに拒否します。FGAC 条件を持つステートメントに dynamodb:SearchVectors を追加しないでください。インデックスリソース ARN によってスコープされ、dynamodb: FGAC 条件がアタッチされていない独自のステートメントで dynamodb:SearchVectors を付与します。

既存のポリシーが dynamodb:LeadingKeys、dynamodb:Attributes、または dynamodb:Select を使用して Amazon DynamoDB アクセスをスコープしている場合、ベクトル検索を追加すると、既存のステートメントを拡張するのではなく、別のステートメントを追加することになります。

DynamoDB は、ベーステーブルと同じ暗号化を使用して、保管中のベクトルデータを暗号化します。ベクトルインデックスは、AWS 所有キー、AWS マネージドキー、または AWS KMS のカスタマーマネージドキーのいずれであっても、テーブルの暗号化設定を継承します。ベクトルインデックスに対して暗号化を個別に設定することはありません。

検索専用アクセスの最小特権ポリシーを含む IAM ポリシーの例については、「ベクトルインデックスを検索するためのアクセスを許可する IAM ポリシー」を参照してください。

チュートリアル: 初めてのベクトル検索

買い物客が正確なキーワードを照合するのではなく、自分の言葉で必要なものを説明できる製品カタログを想像してみてください。各製品説明のベクトル埋め込みを DynamoDB に保存し、ベクトルインデックスをクエリして最も近い一致を見つけることができます。

このチュートリアルでは、ベクトルインデックスを持つテーブルを作成し、Amazon Bedrock Titan Text Embeddings V2 を使用して 1024 次元の埋め込みを生成し、50 個の製品をロードして、類似度の高い上位 5 件を返す類似度検索を実行します。すべてのコマンドは、ターミナルに直接貼り付け

ることができます。DynamoDB での埋め込みの仕組みに関する背景情報については、「the section called “ベクトル埋め込みの生成”」を参照してください。

> **スケールと再現率について**
>
> 本番環境のベクトルインデックスは、数百万から数十億個のベクトルを保持します。ベクトルインデックスは近似最近傍検索を使用し、そのアプローチの再現率特性は、はるかに大きな規模でのみ観測可能になります。ここでは、50 項目のカタログをメカニズムのデモンストレーションとして扱います。サイズ設定と調整のガイダンスについては、「the section called “ベストプラクティス”」を参照してください。

前提条件

作業を開始する前に、次の項目があることを確認します。

- AWS CLI バージョン 2.36.16 以降 2026 年 8 月 4 日にリリースされたサービスモデルの更新で、ベクトルインデックスのサポートが AWS CLI および AWS SDK に追加されました。以前のバージョンでは、`--vector-indexes` パラメータまたは `search-vectors` コマンドは認識されません。`aws --version` でバージョンを確認し、必要に応じてアップグレードします。AWS CLI の代わりに AWS SDK を使用する場合は、`botocore` 1.43.64 以降、または使用する言語の SDK の同等のリリースが必要です。
- DynamoDB の `CreateTable`、`DescribeTable`、`PutItem`、`BatchWriteItem`、`Scan`、`SearchVectors`、`UpdateT` および `DeleteTable` アクションと、Amazon Bedrock の `InvokeModel` アクションを実行するアクセス許可を持つ認証情報。`dynamodb:SearchVectors` は新しいアクションであるため、DynamoDB への読み取りアクセスを許可する既存のポリシーには含まれません。
- お客様のアカウントおよびリージョンにおいて、Amazon Bedrock の Titan Text Embeddings V2 モデルへのアクセスが有効になっています。Amazon Bedrock のモデルへのアクセスはアカウントおよびリージョンごとに付与されるため、モデルを呼び出す前に有効にする必要があります。
- モデルの出力を DynamoDB 形式に整形するために `jq` をインストールしました。
- Amazon Bedrock Titan Text Embeddings V2 が利用可能なリージョン。ベクトルインデックスはすべての商用 AWS リージョンで利用できるため、リージョンの選択における制約となるのは Amazon Bedrock モデルのアベイラビリティです。Amazon Bedrock のモデルの可用性については、「Amazon Bedrock ユーザーガイド」の、「AWS リージョン別のモデルサポート」を参照してください。

料金

このチュートリアルでは、Amazon Bedrock のモデル呼び出しと DynamoDB ストレージの料金が発生します。短い単一文入力に対して、それぞれが Amazon Bedrock 推論リクエストとして請求される 51 の埋め込み呼び出しを行います。DynamoDB ストレージ料金は、テーブルとベクトルインデックスが存在する限り適用されます。

使用しているリージョンを確認する

このチュートリアルのすべてのコマンドは、同じリージョンで実行する必要があります。`AWS_REGION` は `AWS_DEFAULT_REGION` よりも優先され、どちらも AWS CLI 設定ファイルの `region` 設定よりも優先されるため、起動前に AWS CLI が実際に使用するリージョンを確認します。シェルで意図しない `AWS_REGION` 値が設定されていると、意図したリージョン以外の場所にテーブルが作成され、Amazon Bedrock モデルが有効になっていない可能性があります。疑問をすべて解消するには、各コマンドで `--region` *`region`* を明示的に渡します。

`SearchVectors` は別のエンドポイントを使用します

`SearchVectors` は、標準の DynamoDB エンドポイントではなく、専用の検索エンドポイントに解決されます。商用リージョンでは、リクエストは `search-dynamodb.`*`region`*`.amazonaws.com` に送信され、このチュートリアルの他のすべてのオペレーションは `dynamodb.`*`region`*`.amazonaws.com` に送信されます。FIPS とデュアルスタックのバリアントも、同じパターンに従います。これには 2 つの影響があります。

- ネットワークが VPC エンドポイント、プロキシ、または出力許可リストを介してアウトバウンドトラフィックを制限する場合は、検索ホスト名も許可する必要があります。そうでない場合、`CreateTable` と書き込みオペレーションは成功し、`SearchVectors` のみが失敗します。通常は、原因が特定できない接続エラーが発生します。
- `--endpoint-url` を使用して、これらのコマンドの DynamoDB エンドポイントを上書きしないでください。単一のオーバーライドでは両方のホスト名を処理することはできず、検索ルーティングが中断されます。

1. ベクトルインデックスを持つテーブルを作成します。これにより、DescriptionVector 属性に DescriptionIndex という名前のベクトルインデックスを持つ Products テーブルが作成されます。このインデックスは、Titan Text Embeddings V2 の出力と整合させるため、1024 次元の COSINE 距離関数を使用しています。ベクトルインデックスのパーティションキーが定義されていないため、検索に SearchConditionExpression は必要ありません。

```
aws dynamodb create-table \
    --table-name Products \
    --attribute-definitions AttributeName=ProductId,AttributeType=S \
    --key-schema AttributeName=ProductId,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --vector-indexes \
        "[
            {
                \"IndexName\": \"DescriptionIndex\",
                \"VectorAttribute\": {\"AttributeName\": \"DescriptionVector\"},
                \"Projection\": {\"ProjectionType\": \"ALL\"},
                \"Dimensions\": 1024,
                \"DistanceFunction\": \"COSINE\"
            }
        ]"
```

   この例では、検索結果ですべての属性が使用できるように、ProjectionType に ALL を使用しています。代わりに INCLUDE を使用する場合は、射影された非キー属性の上限が共有されることに注意してください。ベクトル属性は 1 つの属性としてカウントされ、各 INLINE_FILTER 検索スキーマ要素は 1 つの属性としてカウントされます。HASH 検索スキーマ要素は制限にカウントされません。

2. インデックスがアクティブになるまで待ちます。IndexStatus が ACTIVE になるまでこれを実行します。

```
aws dynamodb describe-table \
    --table-name Products \
    --query 'Table.VectorIndexes[0].[IndexName,IndexStatus,Backfilling]'
```

   このチュートリアルのように CreateTable の一部として作成されたインデックスの場合、Backfilling は報告されず、コマンドはそのインデックスに対して null を返します。この場合、シグナルとして IndexStatus 単独で使用してください。

UpdateTable を使用して既存のテーブルに追加するベクトルインデックスについて、Backfilling が報告されます。この場合、IndexStatus が ACTIVE になり、Backfilling が true でなくなるまで待ってから検索してください。Backfilling はインデックスが CREATING の状態のときにのみ報告され、インデックスが ACTIVE になると報告されなくなることに注意してください。

> ⚠ テーブルだけでなくインデックスを待機する
>
> aws dynamodb wait table-exists を検索の実行条件として使用しないでください。そのウエーターは Table.TableStatus に基づいてマッチングを行い、ベクトルインデックスがまだ CREATING であっても ACTIVE になります。ベクトルインデックスの準備状況のウェーターがないため、次に示すように DescribeTable をポーリングする必要があります。まだ ACTIVE でないインデックスを検索すると、ValidationException で失敗します。バックフィル中に検索すると、ValidationException も返されます。部分的な結果は返されません。同じ理由で、すべてのベクトルインデックスの作成が完了するまでテーブルを削除することはできません。DeleteTable は、「インデックスの作成、更新、削除中にテーブルを削除することはできません」というメッセージとともに ResourceInUseException を返します。

IndexStatus が ACTIVE になると、検索エンドポイントがインデックスの提供を開始するまでにさらに時間がかかる場合があります。その間は ValidationException で SearchVectors を再試行してください。

3. 製品カタログを作成します。次の 50 個の製品を products.tsv という名前のタブ区切りファイルに保存します。各行には、製品 ID、名前、および 1 文の説明が含まれます。カタログには、5 つの関連製品からなる 10 個のグループが含まれており、最終段階での検索結果を簡単に解釈できます。

```
p01 Insulated Travel Mug A vacuum insulated stainless steel mug that keeps hot
 drinks warm for up to twelve hours.
p02 Stovetop Espresso Maker A compact aluminum pot that brews strong espresso style
 coffee directly on a gas or electric burner.
p03 Manual Burr Coffee Grinder A hand cranked grinder with adjustable ceramic burrs
 for consistent coffee grounds.
p04 Pour Over Coffee Dripper A ceramic cone that sits on a mug and brews a single
 cup of filter coffee.
```

```
p05 Electric Milk Frother A handheld battery powered whisk that creates dense foam
 for lattes and cappuccinos.
p06 Lightweight Running Shoe A breathable mesh road shoe with cushioned foam
 midsole for daily training runs.
p07 Trail Running Shoe An aggressive lugged outsole shoe built for grip on loose
 gravel and muddy trails.
p08 Moisture Wicking Running Socks Ankle height socks knitted from synthetic yarn
 that pulls sweat away from the skin.
p09 Reflective Running Vest A lightweight vest with high visibility strips for
 running safely after dark.
p10 Hydration Waist Belt An elastic belt that holds two small water flasks and a
 phone during long runs.
p11 Ergonomic Mesh Office Chair An adjustable desk chair with breathable mesh back
 and lumbar support for long work sessions.
p12 Sit Stand Desk Converter A height adjustable platform that raises a monitor and
 keyboard for standing work.
p13 Monitor Arm Mount A clamp mounted articulating arm that lifts a display off the
 desk surface.
p14 Under Desk Footrest An angled cushioned platform that supports the feet and
 improves seated posture.
p15 Wireless Split Keyboard A two piece keyboard that separates for a natural
 shoulder width typing position.
p16 Noise Cancelling Headphones Over ear wireless headphones that actively silence
 engine noise on long flights.
p17 Wireless Earbuds Compact in ear buds with a charging case and multi hour
 battery for commuting.
p18 Portable Bluetooth Speaker A water resistant rechargeable speaker sized to fit
 in a backpack side pocket.
p19 Studio Monitor Headphones Wired closed back headphones with flat frequency
 response for audio mixing.
p20 Wired Lapel Microphone A small clip on microphone for recording clear speech
 during interviews.
p21 Four Season Backpacking Tent A double wall tent with an aluminum pole set rated
 for wind and heavy rain.
p22 Down Sleeping Bag A mummy shaped bag filled with compressible down insulation
 for cold weather camping.
p23 Inflatable Sleeping Pad A lightweight pad that inflates in a few breaths and
 packs down to bottle size.
p24 Canister Camping Stove A screw on burner that boils water quickly using a
 compact fuel canister.
p25 Rechargeable Camp Lantern A collapsible lantern with adjustable brightness and
 a built in battery.
p26 Cast Iron Skillet A preseasoned heavy pan that holds heat evenly for searing
 and oven baking.
```

```
p27 Nonstick Frying Pan A coated aluminum pan that releases eggs and fish without
 added oil.
p28 Stainless Steel Stock Pot A tall wide pot for boiling pasta and simmering large
 batches of soup.
p29 Enameled Dutch Oven A heavy lidded pot that moves from stovetop to oven for
 slow braising.
p30 Bamboo Cutting Board A large reversible board with a juice groove around the
 edge.
p31 Padded Laptop Backpack A water resistant pack with a suspended sleeve that
 protects a fifteen inch laptop.
p32 Slim Laptop Sleeve A close fitting neoprene case that shields a notebook inside
 a larger bag.
p33 Rolling Carry On Suitcase A hard shell four wheel case sized to fit most
 overhead cabin bins.
p34 Packing Cube Set Zippered fabric cubes that compress clothing and organize a
 suitcase.
p35 Leather Messenger Bag A single strap shoulder bag with a padded compartment and
 interior pockets.
p36 Daily Facial Moisturizer A light lotion with humectants that hydrates skin
 without leaving residue.
p37 Mineral Sunscreen Lotion A broad spectrum zinc based sunscreen formulated for
 sensitive facial skin.
p38 Gentle Foaming Cleanser A low pH face wash that removes oil and sunscreen
 without stripping the skin.
p39 Vitamin C Serum A brightening serum applied before moisturizer to even skin
 tone over time.
p40 Overnight Repair Cream A rich night cream with ceramides that restores the skin
 barrier while sleeping.
p41 Stainless Steel Dog Bowl A weighted nonslip bowl that resists tipping during
 enthusiastic feeding.
p42 Padded Dog Harness An adjustable chest harness that distributes pull away from
 the neck on walks.
p43 Retractable Dog Leash A spring loaded leash that extends and locks at several
 walking lengths.
p44 Interactive Cat Puzzle Feeder A slow feed tray that makes a cat work for dry
 food and eat more slowly.
p45 Self Cleaning Litter Box An enclosed box with a raking mechanism that sifts
 waste after each use.
p46 Bypass Pruning Shears Sharp hardened blades that make clean cuts on green stems
 and small branches.
p47 Long Handled Garden Spade A steel bladed spade with a wooden shaft for turning
 soil and digging beds.
p48 Adjustable Hose Spray Nozzle A metal nozzle that shifts from a fine mist to a
 strong jet stream.
```

```
p49 Raised Garden Bed Kit Interlocking cedar panels that assemble into an elevated
 planting box.
p50 Drip Irrigation Starter Kit Tubing and emitters that deliver water slowly to
 the base of each plant.
```

3 つのフィールド間の区切り文字はリテラルのタブ文字である必要があります。ブラウザからカタログをコピーする場合は、続行する前にタブが保持されていることを確認してください。

4. 1 つの製品の埋め込みを生成します。最初に、埋め込み呼び出しを 1 回実行して、Amazon Bedrock モデルへのアクセスが機能することを確認します。inputText フィールドには埋め込むテキストが含まれ、dimensions には出力サイズ (有効な値は 256、512、または 1024) を設定します。normalize は単位長ベクトルを生成しますが、これはコサイン類似度検索に推奨されます。

```
mkdir -p emb

aws bedrock-runtime invoke-model \
    --model-id amazon.titan-embed-text-v2:0 \
    --body '{"inputText":"A vacuum insulated stainless steel mug that keeps hot
 drinks warm for up to twelve hours.","dimensions":1024,"normalize":true}' \
    --cli-binary-format raw-in-base64-out \
    --content-type application/json \
    --accept application/json \
    emb/p01.json
```

--cli-binary-format raw-in-base64-out フラグは必須です。AWS CLI v2 はバイナリパラメータの base64 エンコーディングをデフォルトとしているため、このフラグがないと raw JSON 本文が正しく送信されません。レスポンスは emb/p01.json に書き込まれ、1024 個の浮動小数点数の embedding 配列が含まれます。ディメンション数を確認します。

```
jq '.embedding | length' emb/p01.json
```

この出力は 1024 です。

5. 最初の製品を書き込みます。埋め込みを DynamoDB 項目形式に変換して書き込みます。保存されたベクトル属性は、DynamoDB L (リスト) 型を使用して、各数値を N 型でラップします。

```
jq '{"ProductId":{"S":"p01"},"Title":{"S":"Insulated Travel
 Mug"},"DescriptionVector":{"L":[.embedding[]|{"N":(.|tostring)}]}}' emb/p01.json >
 item-p01.json
```

```
aws dynamodb put-item --table-name Products --item file://item-p01.json
```

> **ベクトルサイズと項目の制限**
>
> 1024 次元ベクトルは、DynamoDB の項目サイズ制限である 400 KB を十分に下回ります。埋め込みがベーステーブルで占めるスペースを左右する要因は 2 つあります。ディメンション数と、各値が保持する有効桁数です。DynamoDB は、固定幅ではなく有効桁数に比例して数値を保存します。埋め込みモデルは通常、多数の有効桁を持つ値を返すため、保存された埋め込みは、ベクトルインデックスの 32 ビット浮動小数点精度で保持されている同じベクトルよりもかなり大きくなる可能性があります。アイテムのサイズが設計上重要な場合は、ディメンション数から推定するのではなく、代表的なアイテムを実際に測定してください。

6. 残りの製品を埋め込みます。このループは、残りの各製品について埋め込みを生成します。既存のファイルはスキップされるため、呼び出しが失敗した場合でも安全に再実行できます。

```
while IFS=$'\t' read -r id title description; do
    [ -s "emb/$id.json" ] && continue
    body=$(jq -n --arg t "$description" '{inputText:
$t,dimensions:1024,normalize:true}')
    aws bedrock-runtime invoke-model \
        --model-id amazon.titan-embed-text-v2:0 \
        --body "$body" \
        --cli-binary-format raw-in-base64-out \
        --content-type application/json \
        --accept application/json \
        "emb/$id.json" >/dev/null || echo "FAILED $id"
done < products.tsv

ls emb/*.json | wc -l
```

続行する前に、カウントが 50 になっている必要があります。いずれかの呼び出しが FAILED を出力した場合は、ループを再度実行します。欠落しているファイルのみが再試行されます。

> **InvokeModel レートクォータ**
>
> Amazon Bedrock は、InvokeModel に対してリクエストレートクォータを適用します。このループを変更して呼び出しを並行して発行する場合は、一部の呼び出しでスロットリング例外が発生することを想定し、すべての呼び出しが成功したと仮定するの

ではなく、常に最終的なファイル数を確認してください。部分的に埋め込まれたカタログはエラーなしでロードされ、欠落している製品をサイレントに省略した検索結果を生成します。

7. 残りの製品をロードします。BatchWriteItem が受け入れる最大値である 25 項目ずつのリクエストペイロードを作成します。

```
batch=0
count=0
echo -n '{"Products":[' > batch-0.json
while IFS=$'\t' read -r id title description; do
    if [ "$count" -eq 25 ]; then
        echo ']}' >> "batch-$batch.json"
        batch=$((batch+1)); count=0
        echo -n '{"Products":[' > "batch-$batch.json"
    fi
    [ "$count" -gt 0 ] && echo -n ',' >> "batch-$batch.json"
    jq -c --arg id "$id" --arg title "$title" \
        '{PutRequest:{Item:{ProductId:{S:$id},Title:{S:$title},
          DescriptionVector:{L:[.embedding[]|{"N":(.|tostring)}]}}}}' \
        "emb/$id.json" >> "batch-$batch.json"
    count=$((count+1))
done < <(tail -n +2 products.tsv)
echo ']}' >> "batch-$batch.json"
```

パイプされた while ループが一部のシェルのサブシェルで実行され、batch および count の値が破棄されて不正な形式のバッチファイルが生成されるため、ループはパイプではなくリダイレクトから読み取ります。

各バッチを送信します。BatchWriteItem は部分的に成功する場合があるため、UnprocessedItems で返されたものを再送信します。

```
for f in batch-*.json; do
    cp "$f" pending.json
    for attempt in 1 2 3 4 5; do
        aws dynamodb batch-write-item \
            --request-items file://pending.json \
            --output json > resp.json
        left=$(jq '(.UnprocessedItems.Products // []) | length' resp.json)
        echo "$f attempt $attempt: unprocessed=$left"
        [ "$left" -eq 0 ] && break
```

```
        jq '.UnprocessedItems' resp.json > pending.json
        sleep 2
    done
done
```

50 項目すべてが存在することを確認します。

```
aws dynamodb scan --table-name Products --select COUNT --query 'Count'
```

> **ItemCount の更新が遅延する**
>
> DescribeTable がベクトルインデックスについて報告する ItemCount および IndexSizeBytes の値は、約 6 時間ごとに更新されるため、すべての項目が書き込まれた場合でも、ロード直後は 0 と読み取られることがあります。次に示すように、--select COUNT を指定して Scan を使用して、ロードを検証します。ItemCount がゼロであっても、失敗したロードとして処理しないでください。

8. クエリの埋め込みを生成して検索します。保存された項目に使用したのと同じモデルとディメンション数で検索フレーズを埋め込みます。

```
aws bedrock-runtime invoke-model \
    --model-id amazon.titan-embed-text-v2:0 \
    --body '{"inputText":"How can I make my desk more comfortable to work
 at","dimensions":1024,"normalize":true}' \
    --cli-binary-format raw-in-base64-out \
    --content-type application/json \
    --accept application/json \
    embedding-query.json
```

> **SearchVector 形式が保存されたベクトル形式と異なる**
>
> ベクトルを項目属性に保存するときは、L (リスト) 型でラップします (例: {"L": [{"N":"0.123"},...]})。クエリベクトルを SearchVectors に渡すときは、L ラッパーなしで N 値のプレーン配列を使用します (例: [{"N":"0.123"},...])。この理由により、次の jq 変換は、保存された項目に使用した変換とは異なります。詳細については、「the section called “基本的な検索”」を参照してください。

```
jq '[.embedding[]|{"N":(.|tostring)}]' embedding-query.json > query-vector.json

aws dynamodb search-vectors \
    --table-name Products \
    --index-name DescriptionIndex \
    --search-vector file://query-vector.json \
    --top-k 5 \
    --projection-expression "ProductId, Title" \
    --return-consumed-capacity TOTAL
```

クエリベクトルと保存されたベクトルは、同じ埋め込みモデルから取得され、同じ数のディメンションを持つ必要があります。異なるモデルまたはディメンション数を使用すると、意味のない結果または検証エラーが発生します。

9. 結果を読み取ります。DynamoDB は、類似度でソートされた結果を返します。最も類似度の高い項目が先頭に表示されます。各結果には、射影された Item 属性と Score が含まれます。

```
{
    "SearchResults": [
        {
            "Item": {
                "ProductId": { "S": "p11" },
                "Title": { "S": "Ergonomic Mesh Office Chair" }
            },
            "Score": 0.6130197048187256
        },
        {
            "Item": {
                "ProductId": { "S": "p12" },
                "Title": { "S": "Sit Stand Desk Converter" }
            },
            "Score": 0.781868577003479
        },
        {
            "Item": {
                "ProductId": { "S": "p14" },
                "Title": { "S": "Under Desk Footrest" }
            },
            "Score": 0.816369354724884
        },
        {
```

```
            "Item": {
                "ProductId": { "S": "p13" },
                "Title": { "S": "Monitor Arm Mount" }
            },
            "Score": 0.8283305168151855
        },
        {
            "Item": {
                "ProductId": { "S": "p15" },
                "Title": { "S": "Wireless Split Keyboard" }
            },
            "Score": 0.8469693064689636
        }
    ],
    "ConsumedCapacity": {
        "VectorSearchRequestBytes": 31449.0
    }
}
```

スコアは埋め込みモデルと正確な入力テキストに依存するため、値はわずかに異なります。重要なのは、どの項目がどの順序で選択されたかです。クエリには、椅子、モニター、キーボードという単語が含まれていませんでしたが、検索では、カタログ内の他の 45 項目よりも先に、デスクおよびオフィス関連製品 5 件がすべて返されました。コーヒー、キャンプ、ペットのグループからは何も表示されません。

他のクエリを試して、異なるグループでも同様の動作になるか確認します。"Something to brew fresh coffee at home" という条件で同様の手順を繰り返します。上位の結果はエスプレッソメーカー、プアオーバードリッパー、ミルクフローサーです。"Keeping my dog safe on walks" で同様に行うと、リードとハーネスが最初に来て、その後に関連性があまり高くない項目が続きます。カタログには、ほぼ一致する製品が 2 つしか含まれていないためです。最後のケースは注目に値します。カタログに条件に合う項目がそれほど多く含まれていない場合でも、検索は常にリクエストした数だけの項目を返します。一致の質を判断するには、結果の数ではなく Score 値を使用します。

スコアの読み方は、インデックスが使用する距離関数によって異なります。

- COSINE と EUCLIDEAN はスコアが最も小さい項目を返すため、低いほど似ています。コサインスコアは、同一方向の場合の 0 から、反対方向の場合の 2 までの範囲をとります。
- DOT_PRODUCT はスコアが最も高い項目を返すため、高いほど似ています。

このインデックスは COSINE を使用するため、最初の結果のスコアが最も低くなります。保存された説明と同一のテキストで検索すると、最初にその項目が返され、スコアはゼロまたはゼロに近い値になります。

クリーンアップ

このチュートリアルで作成したリソースは、継続的な料金の発生を回避するために削除してください。ベクトルインデックスのストレージは、検索を実行するかどうかにかかわらず、インデックスが存在する限り課金されます。

ベクトルインデックスを削除し、Products テーブルとその項目を保持するには、UpdateTable を使用します。項目はテーブルに残り、インデックスのみが削除されます。

```
aws dynamodb update-table \
    --table-name Products \
    --vector-index-updates \
        "[
            {\"Delete\": {\"IndexName\": \"DescriptionIndex\"}}
        ]"
```

テーブルとそのベクトルインデックスを一緒に削除するには、テーブルを削除します。

```
aws dynamodb delete-table --table-name Products
```

テーブルがなくなったことを確認します。次のコマンドは、削除が完了すると ResourceNotFoundException を返します。

```
aws dynamodb describe-table --table-name Products
```

保持するテーブルからベクトルインデックスを削除する方法の詳細については、「the section called “ベクトルインデックスの削除”」を参照してください。

次のステップ

この基本的なベクトル検索が完了したら、これらの関連トピックを確認してください。

- ベクトルインデックスのパーティションキーとインラインフィルターを使用した検索結果のフィルタリング – 「the section called “基本的な検索”」を参照してください。

- 距離関数、パーティションキー、ディメンションを選択するためのベストプラクティス – 「the section called “ベストプラクティス”」を参照してください。
- ベクトルインデックスキャパシティとメトリクスのモニタリング – 「the section called “キャパシティのモニタリング”」を参照してください。

テーブルとベクトルインデックス間のデータ同期

DynamoDB は、ベクトルインデックスをベーステーブルと自動的に同期させます。同期の仕組みを理解すると、検索結果が完了するタイミングと、項目が結果に表示されない理由を予測するのに役立ちます。

既存のテーブルにインデックスをバックフィルする

既に項目を含むテーブルにベクトルインデックスを追加すると、DynamoDB は既存のデータを使用してインデックスをバックフィルします。DynamoDB がインデックスのインフラストラクチャをプロビジョニングしている間、インデックスは最初に `IndexStatus` `CREATING` に移行します。DynamoDB が既存のベーステーブルデータからインデックスを入力する間は `CREATING` のままになり、そのフェーズ中は `true` として `Backfilling` がレポートされます。ベーステーブルへの新しい書き込みはバックフィル中にインデックスにレプリケートされますが、バックフィルが完了するまで `SearchVectors` は `ValidationException` を返します。インデックスの準備が完了すると、`IndexStatus` は `ACTIVE` になり、`Backfilling` フィールドはレポートされなくなります。`DescribeTable` を使用して、`IndexStatus` が `ACTIVE` になり、`Backfilling` が `true` でなくなるまで待ってから検索します。

バックフィルの所要時間を左右するのは、ベーステーブルの項目数ではなく、インデックスの構築です。項目数が非常に少ないテーブルでも、バックフィルが完了するまでにかなりの時間がかかることがあります。小さなテーブルがすぐに準備完了になると想定するのではなく、`DescribeTable` をポーリングしてください。

「the section called “ベクトルインデックスの既存テーブルへの追加”」を参照してください。

継続的な書き込み同期

インデックスがアクティブになると、DynamoDB はベーステーブルへの各書き込みをベクトルインデックスにレプリケートします。次の動作は、書き込みがインデックスに到達するかどうか、およびその到達方法に影響します。

- 有効なベクトル属性が含まれている場合にのみ、項目がインデックスにレプリケートされます。インデックスがパーティションキーを定義している場合、項目にはそのパーティションキー属性も含

まれている必要があります。含まれていない場合、書き込みはベーステーブルで成功しますが、項目はインデックスにレプリケートされません。

- ベクトル値は書き込まれたとおりにベーステーブルに保存されますが、インデックスには 32 ビット浮動小数点 (f32) の精度で保存されます。精度の高い値は、インデックスにレプリケートされるとその精度を失います。
- 項目からベクトル属性を削除するか、項目を削除すると、対応するエントリがインデックスから削除されます。

書き込み検証ルールの完全なセットについては、「the section called “ベクトルデータを含む項目の書き込み”」を参照してください。

リージョン間の同期

ベクトルインデックスを持つテーブルがグローバルテーブルの場合、DynamoDB はいずれかのレプリカリージョンへの書き込みを他のリージョンにレプリケートし、そこにインデックスを作成します。DynamoDB は、データをベクトルインデックスに非同期的にレプリケートします。あるリージョンで書き込んだベクトルは、テーブルがマルチリージョンの強力な整合性 (MRSC) を使用している場合でも、別のリージョンの `SearchVectors` 結果にすぐには表示されない場合があります。「the section called “グローバルテーブル”」を参照してください。

ベクトルインデックスのベストプラクティス

以下の推奨事項は、正確でパフォーマンスが高く、費用対効果の高いベクトルインデックスの設計に役立ちます。

最初に埋め込みモデルとディメンションを選択する

使用する埋め込みモデルによってベクトルのディメンション数が決まり、インデックスの作成時に `Dimensions` 設定します。作成後にディメンション数を変更することはできません。インデックスを作成する前に埋め込みモデルを決定し、同じモデルを使用して保存されるベクトルとクエリベクトルの両方を生成します。ディメンションが少ないほど検索、書き込み、ストレージのコストを削減できますが、ディメンションが多いモデルのほうが、意味的な詳細をキャプチャできます。関連性の要件を満たす最小限のディメンション数を選択します。「the section called “ベクトル埋め込みの生成”」を参照してください。

距離関数を埋め込みに一致させる

埋め込みモデルが類似度をどのように表現するかに合った距離関数を選択します。`COSINE` は方向を比較し、大きさを無視するため、ほとんどのテキスト埋め込みモデルに適しています。`EUCLIDEAN`

は絶対距離を測定し、大きさの影響を受けます。DOT_PRODUCT も大きさの影響を受けます。これを使用する場合は、埋め込みを単位長に正規化して、スコアがベクトルの長さではなく方向を反映するようにします。インデックスの作成後に距離関数を変更することはできないため、まず代表的なデータセットを使って選択を検証してください。「the section called “距離関数による結果のランク付け”」を参照してください。

クエリパターンに合ったパーティションキーを選択する

パーティションキーは、各 SearchVectors 呼び出しを単一のパーティションキー値に属するベクトルインデックスの範囲に制限します。その呼び出しは、インデックス全体を検索しません。検索するデータを減らすと、コストを削減し、レイテンシーと再現率を向上させ、パーティションキー値全体でスループットを水平方向にスケーリングできます。

検索のたびに、SearchConditionExpression にパーティションキーの値を指定する必要があります。各検索の範囲は、単一のパーティションキー値のみです。アプリケーションがサポートするクエリパターンに一致するパーティションキーを選択します。

例えば、ロケーションベースのデータを米国の州別に保存する場合、パーティションキーの値は約 50 個になります。各州は、再現率を高めるために意味のある数のベクトルを保持します。50 個のパーティションによって、水平方向のスループットを最大で約 50 倍にスケールできます。これは、各検索が単一の州をターゲットとする場合に機能します。

どちらの方向でも極端なカーディナリティは避けてください。

- 高すぎる (一意の項目 ID など) – 各パーティションに項目が 1 つしか含まれず、比較対象となるネイバーが存在しないため、再現率が低くなります。
- 低すぎる (ブール値など) – ほとんどの項目が 1 つのパーティションに配置されるため、スループットのスケーリングが制限され、レイテンシーやコスト面での利点が減ります。

パーティション内でさらにフィルタリングするには、インラインフィルター属性を使用します。

スループットの例。768 次元の埋め込みモデル (Cohere Embed v3 など) と 1 KB の非ベクトル項目データを使用する場合を想定すると、項目の合計サイズは約 4 KB (768 次元 × 4 バイト + 1 KB) となります。この項目サイズでは、パーティションキーごとの制限は次のようになります。

- 検索: 1 GBps ÷ 4 KB ≈ パーティションキー値ごとに 1 秒あたり 250,000 個のベクトルが検査されます。パーティション内のベクトルの数が増えると、各検索はより多くのデータを調べ、この制限に達するまでの時間が早まります。

- 書き込み: 10 MBps ÷ 4 KB ≈ パーティションキー値あたり 1 秒あたり 2,500 個のベクトル書き込み

増加は比例的ではなく対数的です。パーティション内のベクトルが桁違いに増えても、各検索が調べるデータの増加はわずかです。

より多くのパーティションキー値にデータを分散すると、これらの制限が倍増します。例えば、50 個のパーティションキー値を使用すると、検索と書き込みの合計スループットが最大 50 倍になります。ワークロードがこれらのパーティションキーごとの制限を超える場合は、AWS サポートにお問い合わせください。

埋め込みをソースコンテンツと同期させる

DynamoDB は、埋め込みを自動的に再計算することはありません。埋め込みが表すソースコンテンツを変更するたびに、同じ埋め込みモデルを使用してベクトルを再生成し、項目に書き戻します。そうしないと、インデックスは古いベクトルに基づいて結果を返し続けます。DynamoDB Streams でコンテンツの変更をキャプチャし、ダウンストリームプロセスで影響を受ける埋め込みを再生成および書き換えることを検討してください。

必要な属性のみを射影する

`SearchVectors` は、ベクトルインデックスに射影されていない属性を返すことはできません。より多くの属性を射影すると、インデックスストレージと書き込みコストが増加します。アプリケーションが検索結果から直接読み取る属性を射影し、それ以外の属性は必要に応じてベーステーブルへのフォローアップ `GetItem` または `BatchGetItem` で取得するようにします。

複数のインデックスを使用して埋め込みモデルを比較する

1 つのテーブルに最大 5 つのベクトルインデックスを作成できます。個別のインデックスを使用して、さまざまな埋め込みモデルまたはモデルバージョンを並べて評価します。各モデルの埋め込みを異なるベクトル属性に保存し、それぞれにベクトルインデックスを作成します。これにより、本番稼働用インデックスを移行することなく、同一の基盤データを用いてモデル間の検索品質を比較できます。

例えば、あるモデルバージョンから別のモデルバージョンにアップグレードする場合は、新しいモデルのディメンションと距離関数を使用した 2 つ目のインデックスを作成します。新しいモデルからの埋め込みでバックフィルし、両方のインデックスに対してテストクエリを実行し、関連性を比較します。問題がなければ、アプリケーションを新しいインデックスに移行し、古いインデックスを削除します。

ベクトルインデックスのストレージに関する考慮事項

ベクトルインデックスは、ベーステーブルとは別にストレージを消費します。ベクトルインデックスが使用するストレージは、次の要因によって決まります。

- ベクトルデータ – インデックス付きの各項目は、ベクトルを 32 ビット浮動小数点 (f32) 値として保存します。項目のベクトル部分のストレージは、インデックスのディメンション数に応じてスケールされます。例えば、1,536 次元のベクトルは、384 次元ベクトルの約 4 倍のベクトルストレージを使用します。
- 射影された属性 – どの属性を射影するかを選択することで、インデックスにコピーされるベーステーブル属性が決まります。KEYS_ONLY は最小限のデータを保存し、INCLUDE はキー属性と指定した非キー属性を保存し、ALL はすべての属性を保存します。射影を大きくすると、インデックスストレージの使用量が増加します。
- インデックス付き項目の数 – 有効なベクトル属性 (インデックスがパーティションキーを定義する場合はそのパーティションキー属性) を含む項目のみがインデックスにレプリケートされます。ベクトル属性がない項目は、ベクトルインデックスストレージを消費しません。

ベクトルインデックスストレージを制御するには、関連性のニーズを満たす最小数のディメンションを選択し、アプリケーションが検索結果から直接読み取る属性のみを射影します。現在のストレージ料金については、AWS ウェブサイトの「Amazon DynamoDB 料金」を参照してください。

ベクトルインデックスキャパシティのモニタリング

ベクトルインデックスオペレーションのキャパシティ消費量をモニタリングするには、SearchVectors リクエストで ReturnConsumedCapacity パラメータを INDEXES または TOTAL に設定し、書き込み API リクエストで INDEXES に設定します。

ベクトルインデックスオペレーションは、ベーステーブルで使用される読み取りキャパシティユニットと書き込みキャパシティユニットとは別に、2 つの単位で計測されます。

- ベクトル検索 (VS) – SearchVectors オペレーションを計測する単位。VS の消費量は VectorSearchRequestBytes として報告され、検索対象および検索結果として返されるベクトルデータのサイズに合わせてスケーリングされます。
- ベクトル書き込み (VWR) – ベクトルインデックスにレプリケートされた書き込みを計測する単位。VWR の消費量は VectorWriteRequestBytes として報告され、インデックスにレプリケートされたデータのサイズに合わせてスケーリングされます。

次の例は、SearchVectors リクエストによって返される ConsumedCapacity を示しています。

```
{
    "ConsumedCapacity": {
        "VectorSearchRequestBytes": 41714.0
    }
}
```

書き込みオペレーション
(PutItem、UpdateItem、DeleteItem、BatchWriteItem、TransactWriteItems) の場合、レスポンスの ConsumedCapacity に、インデックス名でキー指定された VectorIndexes マップが含まれます。各エントリは、各ベクトルインデックスへの変更をレプリケートするときに消費されるキャパシティとして、VectorWriteRequestBytes を報告します。

```
{
    "ConsumedCapacity": {
        "TableName": "Products",
        "CapacityUnits": 5.0,
        "Table": {
            "CapacityUnits": 5.0
        },
        "VectorIndexes": {
            "ProductEmbeddingIndex": {
                "VectorWriteRequestBytes": 4125.0
            }
        }
    }
}
```

ベクトルインデックスキャパシティは処理されたバイト単位で計測され、ベーステーブルの読み取りおよび書き込みキャパシティとは別に報告されます。これらのフィールドを使用して、ベクトルインデックスのコストを左右する要因を理解します。

- 検索コスト (VectorSearchRequestBytes) は、主に検索対象となるベクトルのサイズに応じてスケールされます。これは、インデックス内のディメンション数と返されるデータの量に応じて増加します。検索を単一のパーティションキー値に制限すると、検索対象となるデータ量が減ります。結果にベクトル属性を含めて返すと、レスポンスにベクトルデータ全体が含まれるため、コストがさらに増加します。検索コストは、インデックス内の項目数に比例して増加しません。ディメンションの数、TopK 値、および射影が同じままである場合、項目を追加すると、検索が調べるデータは対数的に増加します。これは、近似最近傍検索がすべてのベクトルを読み取るのではな

く、インデックスの小さなサブセットを走査するために発生します。インデックス内の項目数を 2 倍にすると、各検索が調べるデータに少量の一定量が追加されます。

- 書き込みコスト (VectorWriteRequestBytes) は、ベクトルインデックス付き属性を変更する項目の書き込み、更新、または削除を行うたびに発生し、インデックスにレプリケートされるデータのサイズに合わせてスケーリングされます。インデックス付き属性を変更しない書き込みでは、ベクトル書き込みキャパシティは発生しません。

ディメンション数が多い埋め込みでは、各ベクトルがより多くのデータを保持するため、検索コストと書き込みコストの両方が増加します。現在の料金については、AWS ウェブサイトの「Amazon DynamoDB 料金」を参照してください。

DynamoDB は、TableName および VectorIndexName をディメンションとして、ベクトルインデックスキャパシティを VectorSearchRequestBytes および VectorWriteRequestBytes メトリクスとして CloudWatch に発行します。これらのメトリクスを使用して、ベクトルインデックスの経時的な使用状況をグラフ化してアラームを設定します。メトリクスの定義については、「the section called “VectorSearchRequestBytes”」および「the section called “VectorWriteRequestBytes”」を参照してください。

リクエストごとの計測最小値

DynamoDB は、リクエストごとに最小 1 KB のベクトルインデックス容量を計測し、その最小値を超えるバイト数に対して請求します。これはリクエストタイプすべてに適用されます。1 KB 未満のベクトルデータを調べる SearchVectors リクエストは、1 KB で計測されます。1 KB 未満をテーブルのベクトルインデックスにレプリケートする書き込みリクエストも 1 KB で計測されます。

最小値は、インデックスごとではなく、リクエストごとに適用されます。テーブルのいくつかのベクトルインデックスでベクトルを更新する書き込みでは、ConsumedCapacity がインデックスごとに VectorWriteRequestBytes をレポートする場合でも、インデックスごとに 1 KB の最小値が個別に適用されることはありません。

その結果、低いディメンションのベクトルが比例的に低く計測されるわけではありません。ディメンション数が少ないベクトルは、数バイトの 32 ビット浮動小数点データのみを保持しますが、最低 1 KB で計測されます。

最小値を超えると、VectorSearchRequestBytes は、指定したクエリベクトルのサイズではなく、検索がインデックス内で調べるベクトルデータを反映します。その結果、VectorSearchRequestBytes はクエリベクトル単体よりも大きくなります。

前述のように、検索コストは対数的に増加しますが、その関係は請求額を見積もるための計算式ではありません。VectorSearchRequestBytes には、検索が返すデータも含まれます。そのデータは、TopK と、プロジェクションする属性に応じてスケールされます。このメトリクスも、リクエストごとに 1 KB の最小値が下限として設定されます。TopK が大きく、完全なプロジェクションを使用する場合、返されるデータ量が検索で調べるデータ量を上回ることがあります。

ディメンション数からベクトルインデックスのコストを見積もらないでください。独自のワークロードから返される VectorSearchRequestBytes および VectorWriteRequestBytes の値、または対応する CloudWatch メトリクスを使用してください。ワークロードのサイズを設定する前に、代表的なデータセットに対して検証してください。

ベクトルインデックスのトラブルシューティング

次の表に、ベクトルインデックスを使用する際の一般的な問題とその解決方法を示します。

症状	原因	解決策
`ValidationException: SearchConditionExpression must be provided when SearchSchema has a HASH key`	インデックスは SearchSchema でベクトルインデックスのパーティションキー (HASH) を定義していますが、検索でその値が指定されませんでした。	`SearchConditionExpression` にベクトルインデックスのパーティションキーの等価条件 (例: `Category = :cat`) を追加し、それに対応する `ExpressionAttributeValues` エントリも指定します。
`ValidationException: One element in SearchSchema is not defined in attribute definitions`	SearchSchema 属性 (ベクトルインデックスのパーティションキーまたはインラインフィルター) が、テーブルの `AttributeDefinitions` で宣言されていません。	グローバルセカンダリインデックスに対してキー属性が宣言されるのと同じ方法で、すべての SearchSchema 属性をそのタイプと共に `AttributeDefinitions` に追加します。
`ValidationException: Invalid comparato`	= 以外の演算子が使用されました。現在、ベクトルインデックスのパーティション	= のみを使用するように条件を書き換えます。比較、範囲、および集合メンバーシッ

症状	原因	解決策
r used in SearchCon ditionExpression	キーとインラインフィルター属性では、等価演算子のみがサポートされています。	プ演算子はまだ利用できません。
ValidationException: Provided TopK value is out of valid range	TopK は 1～100 の範囲外でした。	TopK を 1〜100 の値に設定します。
ValidationExceptio n: Cannot search backfilling vector index: <indexNam e> -または-Validatio nException: The table does not have the specified index: <indexName>	ベクトルインデックスはまだ検索の準備ができていません。IndexStatus は CREATING です。インデックスがプロビジョニングされている間、または既存のデータをバックフィルしている間に発生します。	DescribeTable を使用して、IndexStatus が ACTIVE になり、Backfilli ng が true でなくなるまで待ってから、検索を再試行します。インデックスの作成中は ValidationExceptio n を再試行可能として扱います。
インデックスが存在する場合は ValidationExceptio n: The table does not have the specified index: <indexName> で、DescribeTable が IndexStatus ACTIVE をレポートする	専用検索エンドポイントはインデックスの処理をまだ開始していません。DescribeT able と SearchVectors は異なるエンドポイントによって提供されており、インデックスが ACTIVE になった後、検索エンドポイントが短時間遅延することがあります。	検索を再試行します。インデックスが ACTIVE になった直後は、Validatio nException を再試行可能として扱います。エラーが解決しない場合は、インデックス名とネットワークが検索エンドポイントのホスト名を許可していることを確認します。
ベクトルを含む項目を書き込む際、書き込みが拒否されます	ベクトルのインデックスのディメンション数が正しくありません。	ベクトルの長さがインデックスの Dimensions と一致することを確認します。「the section called “ベクトルデータを含む項目の書き込み”」を参照してください。

症状	原因	解決策
「パラメータのタイプが無効です」で書き込みが拒否されました	ベクトル属性のデータ型が正しくありません。DynamoDB は、32 ビット浮動小数点数のリストを想定しています。一般的な原因には、文字列 (S)、数値スカラー (N)、数値セット (NS)、または数値以外の要素を含むリストの送信などがあります。エラーメッセージは、単一のメンバーの型が間違っている場合に、問題のある要素のインデックスを特定します。	ベクトル属性を数値 (N) のリスト (L) として送信します。リスト内のすべての要素が有効な数値であり、リストの長さがインデックスの `Dimensions` と一致することを確認します。
ベクトルインデックスのパーティションキー属性を削除または省略した後、項目が `SearchVectors` によって返されなくなりました	項目からベクトルインデックスのパーティションキー属性を削除したり (`UpdateItem` で `REMOVE` を使用)、`PutItem` で省略したりしても エラーは発生しません。ただし、ベーステーブル項目とそのベクトル埋め込みがまだ存在していても、項目はサイレントにインデックス解除され、検索結果に表示されなくなります。	検索対象とするすべての項目が、ベクトルインデックスのパーティションキー属性を保持していることを確認します。項目が予期せず検索結果に表示されない場合は、そのパーティションキー属性がベーステーブル項目に依然として存在することを確認します。
ベクトルインデックスのパーティションキーに対して、「空の文字列値を含めることはできません」または型の不一致で書き込みが拒否されました	ベクトルインデックスのパーティションキー属性が空の文字列に設定されたか、間違ったデータ型で書き込まれました。DynamoDB は、インデックスキー属性を空にできず、宣言された型と一致する必要があるため、これらの書き込みを拒否します。	ベクトルインデックスのパーティションキー属性に、空でない有効な値と正しいデータ型を使用します。インデックス定義を確認して、期待される型を確認します。

症状	原因	解決策
検索結果に属性が不足しています	ベクトルインデックスはこの属性を射影しません。	インデックスに射影された属性のみを取得できます。追加の属性を返すには、より広範な射影でインデックスを再作成するか、ProjectionType を ALL に設定します。

既存のベクトルインデックスにパーティションキーを追加する

ベクトルインデックスの SearchSchema は作成時に固定されます。インデックスの作成後にパーティションキー (HASH) を追加、削除、または変更することはできません。インラインフィルターの属性を変更することもできません。スキーマを変更するには、新しいインデックスを作成して移行します。

パーティションキーを追加したり、他の SearchSchema 属性を変更したりするには、次の手順を実行します。

1. 必要なパーティションキーを使用して新しいベクトルインデックスを作成します。VectorIndexUpdates パラメータの Create アクションで UpdateTable API を使用します。新しいインデックスに別の名前を付けます。手順については、「the section called “ベクトルインデックスの既存テーブルへの追加”」を参照してください。
2. 新しいインデックスがバックフィルを完了するまで待ちます。DescribeTable を呼び出し、IndexStatus が ACTIVE で、Backfilling が false であることを確認します。
3. 新しいインデックスに SearchVectors リクエストを送信するように、アプリケーションを更新します。パーティションキーの値を SearchConditionExpression に含めます。詳細については、「the section called “SearchConditionExpression を使用したフィルタリング”」を参照してください。
4. 正しい結果を確認したら、古いインデックスを削除してストレージ料金の発生を停止します。VectorIndexUpdates パラメータの Delete アクションで UpdateTable API を使用します。手順については、「the section called “ベクトルインデックスの削除”」を参照してください。

> **インデックスオペレーションはベーステーブルのデータには影響しません**
>
> ベクトルインデックスを作成または削除しても、ベーステーブルやその項目には影響しません。DynamoDB は、バックフィル中にベーステーブルの項目からベクトルデータを再取得します。

グローバルテーブルでのベクトルインデックスの使用

グローバルテーブルでベクトルインデックスを使用して、複数の AWS リージョンで類似度検索を実行できます。ベクトルインデックスを持つテーブルにレプリカを追加すると、DynamoDB はベクトルインデックスの定義を新しいレプリカリージョンに自動的にレプリケートします。リージョンごとに個別にベクトルインデックスを作成することはありません。任意のレプリカリージョンで書き込んだ項目は、他のリージョンにレプリケートされ、そこにインデックスが付けられるため、`SearchVectors` はローカルリージョンから結果を返します。

ベクトルインデックスは、すべてのグローバルテーブル設定で使用できます。

- マルチリージョンの結果整合性 (MREC) – グローバルテーブルのデフォルトの整合性モード。詳細については、「the section called “マルチリージョンの結果整合性 (MREC)”」を参照してください。
- マルチリージョンの強力な整合性 (MRSC) – リージョン間で強力な整合性のある読み込みを提供する整合性モード。詳細については、「the section called “マルチリージョンの強力な整合性 (MRSC)”」を参照してください。
- マルチアカウントグローバルテーブル – AWS アカウント間でレプリケートするグローバルテーブル。詳細については、「the section called “マルチアカウントグローバルテーブル”」を参照してください。

MRSC は同一アカウント内の設定のみをサポートしているため、マルチアカウントグローバルテーブルでは常に MREC が使用されます。これはグローバルテーブルの制約であり、ベクトルインデックスの制限ではありません。

次の例では、`us-west-2` リージョンのレプリカを、ベクトルインデックスが既にある `Products` という名前のテーブルに追加します。このテーブルでは、ベクトルインデックスに必要なオンデマンドキャパシティモードを使用する必要があります。

```
aws dynamodb update-table \
```

```
    --table-name Products \
    --region us-east-1 \
    --replica-updates '[{"Create": {"RegionName": "us-west-2"}}]'
```

レプリカの作成中、ソーステーブルのステータスは UPDATING で、レプリカの ReplicaStatus は CREATING です。DescribeTable を使用して、レプリカが ACTIVE になり、ベクトルインデックスがレプリケートされたことを確認します。レプリカリージョンは、ソースリージョンと同じベクトルインデックス名、ディメンション、距離関数、射影を報告します。

```
aws dynamodb describe-table \
    --table-name Products \
    --region us-west-2
```

レプリカがアクティブになったら、ソースリージョンと同じベクトルセットを使用して、レプリカリージョンに対して SearchVectors を実行できます。レプリカリージョンで検索する前に、そのリージョンで DescribeTable を使用し、ベクトルインデックスのバックフィルが完了したことを確認します。IndexStatus は ACTIVE で、Backfilling は true ではありません。レプリカのベクトルインデックスは個別にバックフィルされるため、レプリカ自体がアクティブになってからしばらくの間はバックフィルが続く場合があります。ベクトル検索では近似最近傍 (ANN) が使用されるため、同じデータであっても、同じクエリに対してリージョン間でランク付けがわずかに異なる場合があります。

```
aws dynamodb search-vectors \
    --table-name Products \
    --region us-west-2 \
    --index-name ProductEmbeddingIndex \
    --search-vector '[{"N": "0.1234"}, {"N": "-0.5678"}, {"N": "0.9012"}, ...]' \
    --top-k 10
```

⚠ レプリケーション遅延がクロスリージョンの検索結果に影響する

ベクトルインデックスは、マルチリージョンの強力な整合性 (MRSC) グローバルテーブルであっても、リージョン間で非同期に伝播します。あるリージョンで書き込んだベクトルが、別のリージョンの SearchVectors 結果にすぐには反映されない場合があります。最新の書き込みを読み取るには、書き込みが行われたのと同じリージョンに検索を送信します。

DynamoDB トランザクションで複雑なワークフローを管理する

Amazon DynamoDB Transactions は、テーブル内およびテーブル間の複数の項目を調整したり、変更しないといった、デベロッパーのエクスペリエンスを簡素化します。トランザクションによって DynamoDB に不可分性、一貫性、分離性、耐久性 (ACID) が実現されるため、アプリケーション内でのデータの精度を維持することができます。

DynamoDB トランザクション読み込み/書き込み API を使用し、1 つのオースオアナッシングオペレーションとして複数の項目の追加、更新、または削除が必要となる複雑なビジネスワークフローを管理できます。例えば、ビデオゲームデベロッパーであれば、ゲーム内での項目交換やゲーム内購入の際に、プレイヤーのプロファイル更新の正確性を確保できます。

トランザクション書き込み API を使用して、複数の `Put`、`Update`、`Delete`、`ConditionCheck` の各アクションをグループ化できます。その後、アクションを単一の `TransactWriteItems` オペレーションとして送信できます。このオペレーションはユニットとして成功または失敗します。同じことが複数の `Get` アクションにも当てはまります。この場合、1 つの `TransactGetItems` オペレーションとしてグループ化し、送信できます。

DynamoDB テーブルのトランザクションを有効にするために、追加コストはかかりません。料金の支払いは、トランザクションの一部である読み込みまたは書き込みに対してのみ行われます。DynamoDB は、トランザクション内の各項目の基になっている 2 つの読み込みまたは書き込みを実行します。1 つはトランザクションの準備用で、もう 1 つはトランザクションのコミット用です。これらの基になっている 2 つの読み込み/書き込みオペレーションは、Amazon CloudWatch メトリクスに表示されます。

DynamoDB トランザクションを開始するには、最新の AWS SDK または AWS Command Line Interface (AWS CLI) をダウンロードします。その後、「DynamoDB トランザクションの例」に従います。

以下のセクションでは、トランザクション API の詳細についての概要とそれらを DynamoDB で使用する方法を示します。

トピック

- Amazon DynamoDB Transactions: 仕組み
- DynamoDB トランザクションでの IAM の使用
- DynamoDB トランザクションの例

Amazon DynamoDB Transactions: 仕組み

Amazon DynamoDB Transactions を使用すれば、複数のアクションをまとめてグループ化し、1 つのオールオアナッシングの `TransactWriteItems` または `TransactGetItems` オペレーションとして送信できます。以下のセクションでは、API オペレーション、容量管理、ベストプラクティス、DynamoDB でのトランザクション操作の使用に関する他の詳細について説明します。

トピック

- TransactWriteItems API
- TransactGetItems API
- DynamoDB トランザクションの分離レベル
- DynamoDB でのトランザクション競合の処理
- DynamoDB アクセラレーター (DAX) でのトランザクション API の使用
- トランザクションの容量管理
- トランザクションのベストプラクティス
- グローバルテーブルでのトランザクション API の使用
- DynamoDB トランザクションと AWSLabs トランザクションクライアントライブラリ

TransactWriteItems API

`TransactWriteItems` は、最大 100 の書き込みアクションを 1 つのオールオアナッシングオペレーションにグループ化する、同期的でべき等な書き込みオペレーションです。これらのアクションは、同じ AWS アカウントおよび同じリージョン内の 1 つ以上の DynamoDB テーブルにある最大 100 個の異なる項目をターゲットにすることができます。トランザクション内のアイテムの合計サイズは 4 MB を超えることはできません。すべて成功するかどれも成功しないかのどちらとなるように、アトミックに実行されます。

> **Note**
>
> - `TransactWriteItems` オペレーションは、含まれるすべてのアクションを正常に完了する必要があり、そうでない場合は変更がまったく行われないという点で `BatchWriteItem` オペレーションとは異なります。`BatchWriteItem` オペレーションでは、バッチ内の一部のアクションのみ成功し、他のアクションは成功しないことがあり得ます。

- インデックスを使用してトランザクションを実行することはできません。

同じトランザクション内の複数のオペレーションが同じ項目をターゲットとすることはできません。たとえば、同じトランザクション内で同じ項目に対して ConditionCheck を実行し、Update アクションも実行することはできません。

以下のタイプのアクションをトランザクションに追加できます。

- Put — PutItem オペレーションを開始し、条件付きで、または条件をまったく指定せずに、新しい項目を作成するか、古い項目を新しい項目に置き換えます。
- Update — UpdateItem オペレーションを開始し、既存の項目の属性を編集するか、まだ存在しない場合は新しい項目をテーブルに追加します。条件付きまたは条件なしで既存の項目で属性を追加、削除、更新するには、このアクションを使用します。
- Delete — DeleteItem オペレーションを開始し、プライマリキーにより識別される 1 つの項目をテーブルで削除します。
- ConditionCheck — 項目が存在することを確認するか、項目の特定の属性の条件を確認します。

DynamoDB でトランザクションが完了すると、変更のグローバルセカンダリインデックス (GSIs)、ストリーム、バックアップへの伝播が開始されます。この伝播は以下のとおり、徐々に行われます。同じトランザクションからのストリームレコードは異なるタイミングで表示されるため、他のトランザクションからのレコードとインターリーブする可能性があります。ストリームコンシューマーは、トランザクションの原子性や順番が保証されていると想定すべきではありません。

トランザクションで変更された項目のアトミックスナップショットを確保するには、TransactGetItems オペレーションを使用して、関連するすべての項目をまとめて読み取ります。このオペレーションを使用すると、データの一貫したビューが得られ、完了済みのトランザクションからのすべての変更が表示されるか、まったく表示されません。

伝播は即時に実行されるわけでないため、伝播中にテーブルがバックアップから復元された場合や (RestoreTableFromBackup)、特定時点のエクスポートが行われた場合 (ExportTableToPointInTime)、含まれるのが最近のトランザクション中に行われた変更の一部のみとなる場合もあります。

べき等性

TransactWriteItems 呼び出しを行ってリクエストがべき等であることを確認する場合、オプションでクライアントトークンを含めることができます。トランザクションを冪等にすると、接続

のタイムアウトや他の接続問題に伴って同じオペレーションが複数回送信された場合に、アプリケーションエラーを防ぐことができます。

元の TransactWriteItems 呼び出しが成功した場合、同じクライアントトークンを持つ以降の TransactWriteItems 呼び出しは変更なしで正常に返ります。ReturnConsumedCapacity パラメータが設定されている場合、最初の TransactWriteItems 呼び出しは変更時に消費された書き込みキャパシティーユニットの数を返します。同じクライアントトークンを持つ以降の TransactWriteItems 呼び出しは、項目の読み取り時に消費された読み取りキャパシティユニットの数を返します。

冪等性について重要な点

- クライアントトークンは、それを使用するリクエストが完了してから 10 分間有効です。10 分後、同じクライアントトークンを使用するリクエストは新しいリクエストとして扱われます。10 分が経過してから、同じリクエストに同じクライアントトークンを再利用しないでください。
- 10 分間のべき等性期間内に同じクライアントトークンを使用してリクエストを繰り返すとき、他の一部のリクエストパラメータを変更した場合、DynamoDB は IdempotentParameterMismatch 例外を返します。

書き込みのエラー処理

以下の条件下では、書き込みトランザクションが成功しません。

- いずれかの条件式の条件が満たされていない場合。
- 同じ TransactWriteItems オペレーション内の複数のアクションが同じ項目をターゲットとしているために、トランザクション検証エラーが発生した場合。
- TransactWriteItems リクエストが、TransactWriteItems リクエスト内の 1 つ以上の項目に対する継続中の TransactWriteItems オペレーションと競合する場合。この場合、リクエストは TransactionCanceledException で失敗します。
- トランザクションを完了するプロビジョンドキャパシティーが足りない場合。
- 項目サイズが大きくなりすぎる (400 KB 超)、ローカルセカンダリインデックス (LSI) が大きくなりすぎる、またはトランザクションにより変更が加えられたために同様の検証エラーが発生した場合。
- 無効なデータ形式などのユーザーエラーがある場合。

TransactWriteItems オペレーションとの競合がどのように処理されるかについて詳しくは、「DynamoDB でのトランザクション競合の処理」を参照してください。

TransactGetItems API

TransactGetItems は、最大 100 個の Get アクションをまとめてグループ化する同期読み取りオペレーションです。これらのアクションは、同じ AWS アカウントおよびリージョン内の 1 つ以上の DynamoDB テーブルにある最大 100 個の異なる項目をターゲットにすることができます。トランザクション内の項目の合計サイズは 4 MB を超えることはできません。

Get アクションは、すべて成功するかすべて失敗するかのどちらとなるように、アトミックに実行されます。

- Get — GetItem オペレーションを開始し、指定されたプライマリキーを持つ項目の属性のセットを取得します。一致する項目が見つからない場合、Get はデータを返しません。

読み込みのエラー処理

以下の条件下では、読み取りトランザクションが成功しません。

- TransactGetItems リクエストが、TransactWriteItems リクエスト内の 1 つ以上の項目に対する継続中の TransactGetItems オペレーションと競合する場合。この場合、リクエストは TransactionCanceledException で失敗します。
- トランザクションを完了するプロビジョンドキャパシティーが足りない場合。
- 無効なデータ形式などのユーザーエラーがある場合。

TransactGetItems オペレーションとの競合がどのように処理されるかについて詳しくは、「DynamoDB でのトランザクション競合の処理」を参照してください。

DynamoDB トランザクションの分離レベル

トランザクションオペレーション (TransactWriteItems または TransactGetItems) と他のオペレーションの分離レベルは、次のとおりです。

SERIALIZABLE

直列化可能分離レベルでは、複数の同時オペレーションの結果は、前のオペレーションが完了するまでオペレーションが開始されない場合と同じになります。

以下のタイプのオペレーション間には、直列化可能分離があります。

- トランザクションオペレーションと標準書き込みオペレーション (PutItem、UpdateItem、または DeleteItem) の間。

- トランザクションオペレーションと標準読み取りオペレーション (GetItem) の間。
- TransactWriteItems オペレーションと TransactGetItems オペレーションの間。

トランザクションオペレーション間と BatchWriteItem オペレーション内の個々の標準書き込み間には直列化可能分離がありますが、トランザクションとユニットとしての BatchWriteItem オペレーションの間には直列化可能分離はありません。

同様に、トランザクションオペレーションと GetItems オペレーションの個別の BatchGetItem 間の分離レベルは直列化可能です。ただし、トランザクションとユニットとしての BatchGetItem オペレーション間の分離レベルはコミット済み読み取りです。

単一の GetItem リクエストは、TransactWriteItems リクエストの前または後に行う 2 つの方法のいずれかで、TransactWriteItems リクエストに関してシリアル化することができます。同時実行 TransactWriteItems リクエストのキーに対する複数の GetItem リクエストは、任意の順序で実行できるため、結果は読み込みがコミットされます。

たとえば、項目 A と項目 B の GetItem リクエストが、項目 A と項目 B の両方を変更する TransactWriteItems リクエストと同時に実行される場合、次の 4 つの可能性があります。

- 両方の GetItem リクエストは、TransactWriteItems リクエストの前に実行されます。
- 両方の GetItem リクエストは、TransactWriteItems リクエストの後に実行されます。
- 項目 A の GetItem リクエストは、TransactWriteItems リクエストの前に実行されます。項目 B の場合、GetItem は TransactWriteItems の後に実行されます。
- 項目 B の GetItem リクエストは、TransactWriteItems リクエストの前に実行されます。項目 A の場合、GetItem は TransactWriteItems の後に実行されます。

複数の GetItem リクエストにシリアル化可能な分離レベルが望ましい場合は、TransactGetItems を使用してください。

処理中に同じトランザクション書き込みリクエストの一部であった複数の項目に対して非トランザクション読み取りが行われた場合、一部の項目の新しい状態と他の項目の古い状態を読み取ることができる可能性があります。トランザクション書き込みリクエストに含まれていたすべての項目の新しい状態を読み取ることができるのは、トランザクションが完了したことを示すトランザクション書き込みの応答が成功した場合のみです。

トランザクションが正常に完了し、応答が受信されると、DynamoDB の結果整合性モデルにより、結果整合性のある読み込みオペレーションが短期間、古い状態を返す可能性があります。トラン

ザクションの直後に最新のデータを確実に読み取るには、ConsistentRead を true に設定して、強力な整合性のある読み込みを使用する必要があります。

コミット済み読み取り

コミット済み読み取り分離により、読み取りオペレーションは常に項目のコミット済み値を返します。つまり、読み取りによって、最終的に成功しなかったトランザクション書き込みの状態を表すビューが項目に表示されることはありません。コミット済み読み取り分離では、読み取りオペレーションの直後に項目の変更が防止されません。

分離レベルは、トランザクションオペレーションと、複数の標準読み取り (BatchGetItem、Query、または Scan) が関係する読み取りオペレーションの間ではコミット済み読み取りです。トランザクション書き込みにより BatchGetItem、Query、または Scan オペレーションの途中で項目が更新された場合、その後の読み取りオペレーションの部分では、新しくコミットされた値 (ConsistentRead) を使用) か、場合によってはそれ以前のコミット済み値 (結果整合性のある読み込み) を返します。

オペレーションの概要

以下の表は、トランザクションオペレーション (TransactWriteItems または TransactGetItems) と他のオペレーションの間の分離レベルをまとめたものです。

オペレーション	分離レベル
DeleteItem	直列化可能
PutItem	直列化可能
UpdateItem	直列化可能
GetItem	直列化可能
BatchGetItem	コミット済み読み取り*
BatchWriteItem	直列化不可*
Query	コミット済み読み取り
Scan	コミット済み読み取り
他のトランザクションオペレーション	直列化可能

アスタリスク (*) が付いたレベルは、ユニットとしてオペレーションに適用されます。ただし、これらのオペレーション内の個々のアクションの分離レベルは直列化可能です。

DynamoDB でのトランザクション競合の処理

トランザクション競合は、トランザクション内の項目に対する項目レベルの同時リクエスト中に発生する場合があります。トランザクション競合は、次のシナリオで発生する場合があります。

- 項目に対する PutItem、UpdateItem、または DeleteItem リクエストが、同じ項目を含む継続中の TransactWriteItems リクエストと競合する。
- TransactWriteItems リクエスト内の項目が、継続中の別の TransactWriteItems リクエストの一部である。
- TransactGetItems リクエスト内の項目が、継続中の TransactWriteItems、BatchWriteItem、PutItem、UpdateItem、または DeleteItem リクエストの一部である。

> **Note**
>
> - PutItem、UpdateItem、または DeleteItem リクエストが拒否された場合、リクエストは TransactionConflictException で失敗します。
> - TransactWriteItems または TransactGetItems 内の項目レベルのリクエストが拒否された場合、リクエストは TransactionCanceledException で失敗します。そのリクエストが失敗した場合、AWS SDK はリクエストを再試行しません。
>
>   例外には、TransactItems リクエストパラメータの項目リストどおりに順序付けられた CancellationReasons のリストが含まれます。例外のエラーメッセージには、キャンセル理由コードの文字列表現も含まれます。
> - 継続中の TransactWriteItems オペレーションまたは TransactGetItems オペレーションが同時 GetItem リクエストと競合している場合、両方のオペレーションが成功する可能性があります。

TransactionConflict CloudWatch メトリクスは、項目レベルのリクエストが失敗するたびに増分されます。

DynamoDB アクセラレーター (DAX) でのトランザクション API の使用

`TransactWriteItems` と `TransactGetItems` が、どちらも DynamoDB と同じ分離レベルで DynamoDB アクセラレーター (DAX) でサポートされています。

`TransactWriteItems` は DAX を介して書き込みます。DAX は DynamoDB に `TransactWriteItems` コールを渡し、応答を返します。書き込み後にキャッシュにデータを追加するために、DAX は、`TransactWriteItems` オペレーション内の各項目に対してバックグラウンドで `TransactGetItems` をコールします。これにより、追加の読み込み容量単位が消費されます。(詳しくは、トランザクションの容量管理 を参照してください)。この機能により、アプリケーションロジックをシンプルに保ち、トランザクション処理と非トランザクション処理の両方に DAX を使用できます。

`TransactGetItems` コールは、項目がローカルにキャッシュされることなく DAX を通過します。これは、DAX の強い整合性のある読み込み API と同じです。

トランザクションの容量管理

DynamoDB テーブルのトランザクションを有効にするために、追加コストはかかりません。料金の支払いは、トランザクションの一部である読み込みまたは書き込みに対してのみ行われます。DynamoDB は、トランザクション内の各項目の基になっている 2 つの読み込みまたは書き込みを実行します。1 つはトランザクションの準備用で、もう 1 つはトランザクションのコミット用です。基になっている 2 つの読み込み/書き込みオペレーションは、Amazon CloudWatch メトリクスに表示されます。

この容量は、トランザクションが成功しない場合でも消費されます。例えば、`ConditionCheck` の条件または他のオペレーションの条件が false (キャンセル理由は `ConditionalCheckFailed`) と評価されるためにキャンセルされた `TransactWriteItems` または `TransactGetItems` 呼び出しは、トランザクションが成功した場合と同様に、試行した項目の基盤となる読み取りまたは書き込み容量を引き続き消費します。

容量をテーブルにプロビジョニングするとき、トランザクション API により要求される追加の読み取りと書き込みを計画してください。たとえば、アプリケーションが 1 秒あたり 1 件のトランザクションを実行し、各トランザクションは 500 バイトの項目を 3 個テーブルに書き込むとします。各項目には、2 つの書き込みキャパシティーユニット (WCU) が必要です。1 つはトランザクションの準備用で、もう 1 つはトランザクションのコミット用です。したがって、テーブルには WCU を 6 個プロビジョニングする必要があります。

前の例で DynamoDB アクセラレーター (DAX) を使用していた場合、TransactWriteItems のコールで項目ごとに 2 つの読み込み容量単位 (RCU) も使用します。したがって、テーブルには追加の RCU を 6 個プロビジョニングする必要があります。

同様に、アプリケーションが 1 秒あたり 1 件の読み込みトランザクションを実行し、各トランザクションは 500 バイトの項目を 3 個テーブルで読み取る場合、読み込み容量単位 (RCU) を 6 個テーブルにプロビジョンする必要があります。各項目を読み取るには、2 つの RCU が必要です。1 つはトランザクションの準備用で、もう 1 つはトランザクションのコミット用です。

さらに、SDK のデフォルトの動作は、TransactionInProgressException 例外が発生した場合にトランザクションを再試行します。これらの再試行で消費される追加の読み込みキャパシティーユニット (RCU) を計画してください。同じことは、ClientRequestToken を使用して独自のコードでトランザクションを再試行しようとする場合に当てはまります。

トランザクションのベストプラクティス

DynamoDB トランザクションの使用時に推奨される以下の手法を検討してください。

- テーブルで自動スケーリングを有効にするか、トランザクションの項目ごとに 2 つの読み取りまたは書き込みオペレーションを実行するのに十分なスループット容量をプロビジョニングしたことを確認します。
- AWS によって提供された SDK を使用していない場合、TransactWriteItems 呼び出しを行うときに ClientRequestToken 属性を含め、リクエストがべき等となるようにします。
- 必要でない場合は、トランザクションにオペレーションをまとめてグループ化しないでください。たとえば、10 個のオペレーションを持つ 1 つのトランザクションを、アプリケーションの正確性を低下させずに複数のトランザクションに分割できる場合、トランザクションを分割することをお勧めします。トランザクションをシンプルにするとスループットが向上し、成功する可能性が高まります。
- 同じ項目を同時に更新する複数のトランザクションによって、トランザクションをキャンセルする競合が発生する可能性があります。そのような競合を最小限に抑えるため、データモデリングには以下の DynamoDB のベストプラクティスをお勧めします。
- 属性のセットが 1 つのトランザクションの一部として複数の項目間で頻繁に更新される場合、属性を 1 つの項目にグループ化し、トランザクションのスコープを減らすことを検討してください。
- データを大量に取り込むためにトランザクションを使用しないでください。一括書き込みには、BatchWriteItem の使用をお勧めします。

グローバルテーブルでのトランザクション API の使用

トランザクションオペレーションは、書き込み API が最初に呼び出された AWS リージョン内でのみ、不可分性、一貫性、分離性、および耐久性 (ACID) を保証します。グローバルテーブルのリージョン間では、トランザクションはサポートされていません。たとえば、米国東部 (オハイオ) リージョンと米国西部 (オレゴン) リージョンにレプリカを含むグローバルテーブルがあり、米国東部 (バージニア北部) リージョンで `TransactWriteItems` オペレーションを実行するとします。変更がレプリケートされると、米国西部 (オレゴン) リージョンで部分的に完了したトランザクションを確認できます。変更は、ソースリージョンでコミットされた後でのみ、他のリージョンにレプリケートされます。

DynamoDB トランザクションと AWSLabs トランザクションクライアントライブラリ

DynamoDB トランザクションには、AWSLabs トランザクションクライアントライブラリを置き換えるより費用対効率、堅牢性、パフォーマンスに優れた手段が用意されています。ネイティブのサーバー側トランザクション API を使用するようにアプリケーションを更新することをお勧めします。

DynamoDB トランザクションでの IAM の使用

AWS Identity and Access Management (IAM) を使用すると、トランザクションオペレーションが Amazon DynamoDB で実行可能なアクションを制限できます。DynamoDB での IAM ポリシーの詳細な使用については、DynamoDB のアイデンティティベースのポリシー を参照してください。

`Put`、`Update`、`Delete`、および `Get` アクションの権限は、基になる `PutItem`、`UpdateItem`、`DeleteItem`、および `GetItem` オペレーションに使用される権限により決定されます。`ConditionCheck` アクションの場合、IAM ポリシーで `dynamodb:ConditionCheckItem` 許可を使用できます。

以下に、DynamoDB トランザクションの設定に使用できる IAM ポリシーの例を示します。

例 1: トランザクションオペレーションを許可する

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ]
        }
    ]
}
```

例 2: トランザクションオペレーションのみを許可する

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "dynamodb:EnclosingOperation": [
                        "TransactWriteItems",
                        "TransactGetItems"
                    ]
                }
            }
```

```
        }
    ]
}
```

例 3: トランザクションを使用しない読み込み/書き込みを許可し、トランザクションを使用する読み込み/書き込みをブロックする

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "dynamodb:EnclosingOperation": [
                        "TransactWriteItems",
                        "TransactGetItems"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
             "Action": [
                 "dynamodb:PutItem",
                 "dynamodb:DeleteItem",
                 "dynamodb:GetItem",
                 "dynamodb:UpdateItem"
             ],
```

```
            "Resource": [
                "arn:aws:dynamodb:*:*:table/table04"
            ]
        }
    ]
}
```

例 4: ConditionCheck の失敗時に情報が返されないようにする

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/table01",
            "Condition": {
                "StringEqualsIfExists": {
                    "dynamodb:ReturnValues": "NONE"
                }
            }
        }
    ]
}
```

DynamoDB トランザクションの例

Amazon DynamoDB Transactions が役に立つ状況の例として、このサンプルの Java アプリケーションをオンラインマーケットプレイスで検討してください。

アプリケーションには、バックエンドに 3 つの DynamoDB テーブルがあります。

- Customers — このテーブルには、マーケットプレイスの顧客に関する詳細が保存されます。プライマリキーは CustomerId 一意の識別子です。
- ProductCatalog — このテーブルには、マーケットプレイスで販売されている製品の価格や在庫状況などの詳細が保存されます。プライマリキーは ProductId 一意の識別子です。
- Orders — このテーブルには、マーケットプレイスからの注文に関する詳細が保存されます。プライマリキーは OrderId 一意の識別子です。

注文を作成する

次のコードスニペットは、DynamoDB トランザクションを使用して、注文の作成と処理に必要な複数のステップを調整する方法を示しています。単一のオールオアナッシングオペレーションを使用すると、トランザクションのいずれかの部分が失敗しても、トランザクション内のアクションは実行されず、変更も行われません。

この例では、customerId が 09e8e9c8-ec48 である顧客からの注文を設定します。次に、次の単純な注文処理ワークフローを使用して、単一のトランザクションとして実行します。

1. 顧客 ID が有効であることを確認します。
2. 製品が IN_STOCK であることを確認し、製品のステータスを SOLD に更新します。
3. 注文がまだ存在していないことを確認し、注文を作成します。

顧客を検証する

まず、customerId が 09e8e9c8-ec48 に等しい顧客が顧客テーブルに存在することを確認するアクションを定義します。

```
final String CUSTOMER_TABLE_NAME = "Customers";
final String CUSTOMER_PARTITION_KEY = "CustomerId";
final String customerId = "09e8e9c8-ec48";
final HashMap<String, AttributeValue> customerItemKey = new HashMap<>();
customerItemKey.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));

ConditionCheck checkCustomerValid = new ConditionCheck()
    .withTableName(CUSTOMER_TABLE_NAME)
    .withKey(customerItemKey)
    .withConditionExpression("attribute_exists(" + CUSTOMER_PARTITION_KEY + ")");
```

製品のステータスを更新する

次に、製品ステータスが現在 IN_STOCK に設定されている条件が true の場合に、製品ステータスを SOLD に更新するアクションを定義します。項目の製品ステータス属性が IN_STOCK に等しくなかった場合、ReturnValuesOnConditionCheckFailure パラメータを設定すると項目が返されます。

```
final String PRODUCT_TABLE_NAME = "ProductCatalog";
final String PRODUCT_PARTITION_KEY = "ProductId";
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
expressionAttributeValues.put(":new_status", new AttributeValue("SOLD"));
expressionAttributeValues.put(":expected_status", new AttributeValue("IN_STOCK"));

Update markItemSold = new Update()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey)
    .withUpdateExpression("SET ProductStatus = :new_status")
    .withExpressionAttributeValues(expressionAttributeValues)
    .withConditionExpression("ProductStatus = :expected_status")

 .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD);
```

注文を作成する

最後に、OrderId の注文がまだ存在しない場合に限って、注文を作成します。

```
final String ORDER_PARTITION_KEY = "OrderId";
final String ORDER_TABLE_NAME = "Orders";

HashMap<String, AttributeValue> orderItem = new HashMap<>();
orderItem.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));
orderItem.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));
orderItem.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));
orderItem.put("OrderStatus", new AttributeValue("CONFIRMED"));
orderItem.put("OrderTotal", new AttributeValue("100"));

Put createOrder = new Put()
    .withTableName(ORDER_TABLE_NAME)
    .withItem(orderItem)
```

```
.withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD)
   .withConditionExpression("attribute_not_exists(" + ORDER_PARTITION_KEY + ")");
```

トランザクションを実行する

次の例は、単一のオールオアナッシングオペレーションとして以前に定義されたアクションを実行する方法を示しています。

```
   Collection<TransactWriteItem> actions = Arrays.asList(
       new TransactWriteItem().withConditionCheck(checkCustomerValid),
       new TransactWriteItem().withUpdate(markItemSold),
       new TransactWriteItem().withPut(createOrder));

   TransactWriteItemsRequest placeOrderTransaction = new TransactWriteItemsRequest()
       .withTransactItems(actions)
       .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

   // Run the transaction and process the result.
   try {
       client.transactWriteItems(placeOrderTransaction);
       System.out.println("Transaction Successful");

   } catch (ResourceNotFoundException rnf) {
       System.err.println("One of the table involved in the transaction is not found"
+ rnf.getMessage());
   } catch (InternalServerErrorException ise) {
       System.err.println("Internal Server Error" + ise.getMessage());
   } catch (TransactionCanceledException tce) {
       System.out.println("Transaction Canceled " + tce.getMessage());
   }
```

注文詳細の読み込み

次の例は、完了した注文を Orders テーブルと ProductCatalog テーブルでのトランザクションから読み込む方法を示しています。

```
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

HashMap<String, AttributeValue> orderKey = new HashMap<>();
orderKey.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));
```

```
Get readProductSold = new Get()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey);
Get readCreatedOrder = new Get()
    .withTableName(ORDER_TABLE_NAME)
    .withKey(orderKey);

Collection<TransactGetItem> getActions = Arrays.asList(
    new TransactGetItem().withGet(readProductSold),
    new TransactGetItem().withGet(readCreatedOrder));

TransactGetItemsRequest readCompletedOrder = new TransactGetItemsRequest()
    .withTransactItems(getActions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Run the transaction and process the result.
try {
    TransactGetItemsResult result = client.transactGetItems(readCompletedOrder);
    System.out.println(result.getResponses());
} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found" +
 rnf.getMessage());
} catch (InternalServerErrorException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.err.println("Transaction Canceled" + tce.getMessage());
}
```

Amazon DynamoDB の変更データキャプチャ

多くのアプリケーションでは、DynamoDB テーブルに保存された項目の変更を、変更の発生時にキャプチャすることで利点を利用できます。以下に示しているのは、いくつかのユースケースの例です。

- 人気のモバイルアプリケーションは、1 秒あたり数千件の更新速度で、DynamoDB テーブルのデータを変更します。別のアプリケーションは、これらの更新に関するデータをキャプチャして保存し、モバイルアプリの使用状況メトリクスをほぼリアルタイムで提供します。
- 金融アプリケーションは、DynamoDB テーブル内の株式市場データを変更します。並行して実行されるさまざまなアプリケーションは、これらの変化をリアルタイムで追跡し、リスクのある価値を計算し、株価の動きに基づいてポートフォリオを自動的にリバランスします。

- 輸送車両や産業機器のセンサーは、DynamoDB テーブルにデータを送信します。さまざまなアプリケーションがパフォーマンスをモニタリングし、問題が検出されたときにメッセージングアラートを送信し、機械学習アルゴリズムを適用して潜在的な欠陥を予測し、データを圧縮して Amazon Simple Storage Service (Amazon S3) にアーカイブします。
- アプリケーションは、友人の 1 人が新しい画像をアップロードするとすぐに、グループ内のすべての友人のモバイルデバイスに通知を自動送信します。
- 新しいお客様がデータを DynamoDB テーブルに追加します。このイベントにより、新しいお客様にようこそメールを送信する別のアプリケーションが起動されます。

DynamoDB は、項目レベルの変更データキャプチャレコードのストリーミングをほぼリアルタイムでサポートします。これらのストリーミングを使用し、内容に基づいてアクションを実行するアプリケーションを構築できます。

> **Note**
>
> DynamoDB Streams へのタグの追加、DynamoDB Streams での属性ベースのアクセス制御 (ABAC) の使用はサポートされていません。

次の動画では、データキャプチャ変更の概念を紹介します。

変更データキャプチャストリーム

トピック

- 変更データキャプチャのストリーミングオプション
- Kinesis Data Streams を使用して DynamoDB への変更をキャプチャする。
- DynamoDB Streams の変更データキャプチャ

変更データキャプチャのストリーミングオプション

DynamoDB には、変更データキャプチャ用の 2 つのストリーミングモデルがあります。DynamoDB 用 Kinesis Data Streams と DynamoDB Streams です。

アプリケーションに適したソリューションを選択しやすくするために、次の表に、各ストリーミングモデルの特徴をまとめてあります。

プロパティ	DynamoDB 用 Kinesis Data Streams	DynamoDB Streams
データ保持期間	1 年まで。	24 時間。
Kinesis Client Library (KCL) サポート	KCL バージョン 1.X、2.X および 3.X サポート。	KCL バージョン 1.X および 2.X サポート。
コンシューマー数	シャードごとに最大 5 つの同時コンシューマー、またはファンアウトが強化されたシャードごとに最大 20 の同時コンシューマー。	シャードごとに最大 2 つの同時コンシューマー。
スループットクォータ	無制限。	DynamoDB テーブルと AWS リージョンによるスループットクォータの対象となります。
レコードの配信モデル	GetRecords を使用して HTTP 経由でモデルをプルし、ファンアウトを強化すると、Kinesis Data Streams は SubscribeToShard を使用して HTTP/2 経由でレコードをプッシュします。	GetRecords を使用した HTTP 経由のプルモデル。
レコードの順序	各ストリーミングレコードのタイムスタンプ属性を使用して、DynamoDB テーブルで変更が発生した実際の順序を特定できます。	DynamoDB テーブルで変更された各項目について、ストリーミングレコードは項目に対する実際の変更と同じ順序で出現します。
重複レコード	重複レコードがストリーミングに表示される場合があります。	重複レコードがストリーミングに表示されません。

プロパティ	DynamoDB 用 Kinesis Data Streams	DynamoDB Streams
ストリーミング処理オプション	AWS Lambda、Amazon Managed Service for Apache Flink。Kinesis Data Firehose、または AWS Glue ストリーミング ETL を使用してストリームレコードを処理します。	AWS Lambda、DynamoDB Streams Kinesis adapter を使用してストリームレコードを処理します。
耐久性	アベイラビリティーゾーンにより、中断することなく自動的にフェイルオーバーできます。	アベイラビリティーゾーンにより、中断することなく自動的にフェイルオーバーできます。

同じ DynamoDB テーブルで両方のストリーミングモデルを有効にすることができます。

次の動画では、これら 2 つのオプションの違いを詳しく説明しています。

DynamoDB ストリームと Kinesis データストリームの比較

Kinesis Data Streams を使用して DynamoDB への変更をキャプチャする。

Amazon Kinesis Data Streams を使用して Amazon DynamoDB への変更をキャプチャできます。

Kinesis Data Streams は、DynamoDB テーブルの項目レベルの変更をキャプチャーし、それらを Kinesis Data Streams にレプリケートします。アプリケーションは このストリームにアクセスして、項目レベルの変更をほぼリアルタイムで表示できます。1 時間あたりテラバイトのデータを継続的に取り込み保存できます。より長いデータ保持時間を利用し、強化されたファンアウト機能により、2 つ以上のダウンストリームアプリケーションに同時にアクセスできます。その他のメリットには、追加の監査とセキュリティの透明性が含まれます。

Kinesis Data Streams を使用すると、Amazon Data Firehose および Amazon Managed Service for Apache Flink にもアクセスできます。これらのサービスは、リアルタイムでのダッシュボードの強化、アラートの生成、動的な料金設定、広告の実装、高度なデータ分析および機械学習アルゴリズムを実装するアプリケーションの構築に役立ちます。

> **Note**
>
> DynamoDB で Kinesis データストリームを使用すると、データストリームに対する Kinesis Data Streams 料金とソーステーブルに対する DynamoDB 料金の両方が適用されます。

コンソール、AWS CLI、または Java SDK を使用して DynamoDB テーブルで Kinesis ストリーミングを有効にするには、「Amazon DynamoDB 用 Kinesis Data Streams の開始方法」を参照してください。

トピック

- Kinesis Data Streams の DynamoDB との連携について
- Amazon DynamoDB 用 Kinesis Data Streams の開始方法
- DynamoDB Streams および Kinesis Data Streams でのシャードとメトリクスの使用
- Amazon Kinesis Data Streams および Amazon DynamoDB の IAM ポリシーを使用する

Kinesis Data Streams の DynamoDB との連携について

DynamoDB テーブルで Kinesis データストリームが有効になっている場合、そのテーブルは、テーブルのデータに対するすべての変更をキャプチャしたデータレコードを送信します。このデータレコードは以下を含みます。

- 項目が最後に作成、更新、または削除された特定の時刻
- その項目のプライマリキー
- 変更前のレコードのスナップショット
- 変更後のレコードのスナップショット

これらのデータレコードは、ほぼリアルタイムに取り込まれ、公開されます。これらが Kinesis データストリームに書き込まれた後は、他のレコードと同じように読み込むことができます。Kinesis Client Library の使用、AWS Lambda の使用、Kinesis Data Streams API の呼び出し、およびその他接続サービスの利用が可能です。詳細については、Amazon Kinesis Data Streams デベロッパーガイドのAmazon Kinesis Data Streams からのデータの読み込みを参照してください。

これらのデータへの変更は、非同期的にキャプチャーされます。Kinesis は、ストリーミング元のテーブルに対するパフォーマンスに影響を与えません。Kinesis データストリームに保存されてい

るストリームレコードも、保存時に暗号化されます。詳細については、「Amazon Kinesis Data Streams のデータ保護」を参照してください。

Kinesis データストリームレコードは、項目の変更が発生した順序とは異なる順序で表示される場合があります。同じ項目の通知がストリームに複数回表示される場合もあります。ApproximateCreationDateTime 属性をチェックして、項目の変更が発生した順序を特定したり、重複するレコードを特定したりできます。

Kinesis データストリームを DynamoDB テーブルのストリーミング先として有効にすると、ApproximateCreationDateTime 値の精度をミリ秒またはマイクロ秒単位で設定できます。デフォルトでは、ApproximateCreationDateTime は変更の時刻をミリ秒単位で示します。さらに、アクティブなストリーミング先でこの値を変更できます。このような変更後、Kinesis に書き込まれるストリームレコードは、希望する精度の ApproximateCreationDateTime 値を持つようになります。

DynamoDB に書き込むバイナリ値は、base64 エンコード形式でエンコードする必要があります。ただし、データレコードを Kinesis データストリームに書き込む場合、これらのエンコードされたバイナリ値は、再度 base64 エンコーディングでエンコードされます。これらのレコードを Kinesis データストリームから読み取る場合、未加工のバイナリ値を取得するには、アプリケーションでこれらの値を 2 回デコードする必要があります。

DynamoDB では、Kinesis Data Streams の使用に対して変更データキャプチャ単位で課金されます。単一の項目あたり 1 KB の変更が、1 つの変更データキャプチャ単位としてカウントされます。各項目で変更したキロバイト数は、書き込み操作のキャパシティーユニット消費量と同じロジックを使用して、ストリームに書き込まれた項目の「前の」イメージと「後の」イメージの大きい方で計算されます。DynamoDB オンデマンドモードの動作と同様に、変更データキャプチャ単位のキャパシティースループットをプロビジョニングする必要はありません。

DynamoDB テーブルの Kinesis データストリームを有効にする

AWS マネジメントコンソール、AWS SDK、または AWS Command Line Interface (AWS CLI) を使用して、既存の DynamoDB テーブルから Kinesis へのストリーミングを有効または無効にすることができます。

- テーブルと同じ AWS アカウントと AWS リージョンでのみ、DynamoDB から Kinesis Data Streams にデータをストリーミングできます。
- DynamoDB テーブルからのデータを 1 つの Kinesis データストリームにだけストリーミングできます。

DynamoDB テーブルの Kinesis Data Streams 送信先に変更を加える

デフォルトでは、すべての Kinesis データストリームレコードには `ApproximateCreationDateTime` 属性が含まれます。この属性は、各レコードが作成されたおおよその時刻のタイムスタンプをミリ秒単位で表します。この値の精度は、https://console.aws.amazon.com/kinesis、SDK、または AWS CLI を使用して変更できます。

Amazon DynamoDB 用 Kinesis Data Streams の開始方法

このセクションでは、Amazon DynamoDB コンソール、AWS Command Line Interface (AWS CLI)、および API を活用して Amazon DynamoDB 用の Amazon Kinesis Data Streams テーブルを使用する方法について説明します。

アクティブな Amazon Kinesis Data Streams の作成

これらの例はすべて、DynamoDB の使用開始チュートリアルの一部として作成された `Music` DynamoDB テーブルを使用しています。

コンシューマーを構築し、Kinesis データストリームを他の AWS のサービスに接続する詳細方法については、「Amazon Kinesis Data Streams デベロッパーガイド」の「Kinesis Data Streams からのデータの読み込み」を参照してください。

> **Note**
>
> KDS シャードを初めて使用するときは、使用パターンに合わせてシャードをスケールアップまたはスケールダウンするように設定することをお勧めします。使用パターンに関するデータをさらに蓄積したら、それに合わせてストリーム内のシャードを調整できます。

Console

1. AWS マネジメントコンソール にサインインし、Kinesis コンソール (https://console.aws.amazon.com/kinesis/) を開きます。
2. [Create data stream (データストリーミングの作成)] を選択し、指示に従って `samplestream` というストリーミングを作成します。
3. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
4. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
5. [Music] テーブルを選択します。

6. [エクスポートとストリーム] タブを選択します。
7. (オプション) [Amazon Kinesis データストリームの詳細] で、レコードのタイムスタンプの精度をマイクロ秒 (デフォルト) からミリ秒に変更できます。
8. ドロップダウンリストから samplestream を選択します。
9. [オンにする] ボタンを選択します。

AWS CLI

1. create-stream コマンドを使用して、`samplestream` という名前の Kinesis データストリームを作成します。

```
aws kinesis create-stream --stream-name samplestream --shard-count 3
```

Kinesis データストリームのシャード数を設定する前に「Kinesis Data Streams のシャード管理に関する考慮事項」を参照してください。

2. Kinesis ストリームがアクティブで、使用できる状態になっていることを確認するには、describe-stream コマンドを使用します。

```
aws kinesis describe-stream --stream-name samplestream
```

3. DynamoDB `enable-kinesis-streaming-destination` コマンドを使用して、DynamoDB テーブルで Kinesis ストリーミングを有効にします。`stream-arn` の値を、前のステップの `describe-stream` によって返された値で置き換えます。オプションで、各レコードに返されるタイムスタンプ値の精度をより細かく (マイクロ秒) したストリーミングを有効にします。

マイクロ秒のタイムスタンプ精度でのストリーミングを有効にします。

```
aws dynamodb enable-kinesis-streaming-destination \
  --table-name Music \
  --stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
  --enable-kinesis-streaming-configuration
 ApproximateCreationDateTimePrecision=MICROSECOND
```

または、デフォルトのタイムスタンプ精度 (ミリ秒) でのストリーミングを有効にします。

```
aws dynamodb enable-kinesis-streaming-destination \
  --table-name Music \
```

```
  --stream-arn arn:aws:kinesis:us-west-2:12345678901:stream/samplestream
```

4. DynamoDB describe-kinesis-streaming-destination コマンドを使用して、テーブルで Kinesis ストリーミングがアクティブかどうかを確認します。

```
aws dynamodb describe-kinesis-streaming-destination --table-name Music
```

5. 「DynamoDB デベロッパーガイド」の説明通りに、put-item コマンドを使用して DynamoDB テーブルにデータを書き込みます。

```
aws dynamodb put-item \
    --table-name Music  \
    --item \
        '{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me
 Today"}, "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}'

aws dynamodb put-item \
    --table-name Music \
    --item \
        '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"},
 "AlbumTitle": {"S": "Songs About Life"}, "Awards": {"N": "10"} }'
```

6. Kinesis get-records CLI コマンドを使用して、Kinesis ストリームコンテンツを取得します。次に、以下のコードスニペットを使用して、ストリーミングコンテンツを逆シリアル化します。

```
/**
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as
 an example.
 */
public void processRecord(Record kinesisRecord) throws IOException {
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);

    /**
     * Say for example our record contains a String attribute named "stringName"
 and we want to fetch the value
```

```
     * of this attribute from the new item image. The following code fetches
 this value.
     */
    JsonNode attributeNode = newItemImage.get("stringName");
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"
 type attribute
    String attributeValue = attributeValueNode.textValue();
    System.out.println(attributeValue);
}

private Instant fetchTimestamp(JsonNode dynamoDBRecord) {
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");
    JsonNode timestampPrecisionJson =
 dynamoDBRecord.get("ApproximateCreationDateTimePrecision");
    if (timestampPrecisionJson != null &&
 timestampPrecisionJson.equals("MICROSECOND")) {
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);
    }
    return Instant.ofEpochMilli(timestampJson.longValue());
}
```

Java

1. 「Kinesis Data Streams デベロッパーガイド」の指示に従って、Java を使用し `samplestream` という名前の Kinesis データストリームを作成します。

   Kinesis Data Streams のシャード数を設定する前に Kinesis Data Streams のシャード管理に関する考慮事項 を参照してください。

2. 次のコードスニペットを使用して、DynamoDB テーブルで Kinesis ストリーミングを有効にします。オプションで、各レコードに返されるタイムスタンプ値の精度をより細かく (マイクロ秒) したストリーミングを有効にします。

   マイクロ秒のタイムスタンプ精度でのストリーミングを有効にします。

```
EnableKinesisStreamingConfiguration enableKdsConfig =
 EnableKinesisStreamingConfiguration.builder()

 .approximateCreationDateTimePrecision(ApproximateCreationDateTimePrecision.MICROSECOND)
  .build();
```

```
EnableKinesisStreamingDestinationRequest enableKdsRequest =
 EnableKinesisStreamingDestinationRequest.builder()
  .tableName(tableName)
  .streamArn(kdsArn)
  .enableKinesisStreamingConfiguration(enableKdsConfig)
  .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
 ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

または、デフォルトのタイムスタンプ精度 (ミリ秒) でのストリーミングを有効にします。

```
EnableKinesisStreamingDestinationRequest enableKdsRequest =
 EnableKinesisStreamingDestinationRequest.builder()
  .tableName(tableName)
  .streamArn(kdsArn)
  .build();

EnableKinesisStreamingDestinationResponse enableKdsResponse =
 ddbClient.enableKinesisStreamingDestination(enableKdsRequest);
```

3. 「Kinesis Data Streams デベロッパーガイド」の指示に従って、作成したデータストリームから読み込みます。
4. 次のコードスニペットを使用して、ストリームコンテンツを逆シリアル化します。

```
/**
 * Takes as input a Record fetched from Kinesis and does arbitrary processing as
 an example.
 */
public void processRecord(Record kinesisRecord) throws IOException {
    ByteBuffer kdsRecordByteBuffer = kinesisRecord.getData();
    JsonNode rootNode = OBJECT_MAPPER.readTree(kdsRecordByteBuffer.array());
    JsonNode dynamoDBRecord = rootNode.get("dynamodb");
    JsonNode oldItemImage = dynamoDBRecord.get("OldImage");
    JsonNode newItemImage = dynamoDBRecord.get("NewImage");
    Instant recordTimestamp = fetchTimestamp(dynamoDBRecord);

    /**
     * Say for example our record contains a String attribute named "stringName"
 and we wanted to fetch the value
     * of this attribute from the new item image, the below code would fetch
 this.
```

```
     */
    JsonNode attributeNode = newItemImage.get("stringName");
    JsonNode attributeValueNode = attributeNode.get("S"); // Using DynamoDB "S"
 type attribute
    String attributeValue = attributeValueNode.textValue();
    System.out.println(attributeValue);
}

private Instant fetchTimestamp(JsonNode dynamoDBRecord) {
    JsonNode timestampJson = dynamoDBRecord.get("ApproximateCreationDateTime");
    JsonNode timestampPrecisionJson =
 dynamoDBRecord.get("ApproximateCreationDateTimePrecision");
    if (timestampPrecisionJson != null &&
 timestampPrecisionJson.equals("MICROSECOND")) {
        return Instant.EPOCH.plus(timestampJson.longValue(), ChronoUnit.MICROS);
    }
    return Instant.ofEpochMilli(timestampJson.longValue());
}
```

アクティブな Amazon Kinesis データストリームに変更を加える

このセクションでは、コンソール、AWS CLI、API を使用して DynamoDB 用 Kinesis Data Streams のセットアップを変更する方法について説明します。

AWS マネジメントコンソール

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. テーブルに移動します。
3. [エクスポートおよびストリーム] タブを選択します。

AWS CLI

1. `describe-kinesis-streaming-destination` を呼び出して、ストリームが `ACTIVE` であることを確認します。
2. 次の例のように、`UpdateKinesisStreamingDestination` を呼び出します。

```
aws dynamodb update-kinesis-streaming-destination --table-name
 enable_test_table --stream-arn arn:aws:kinesis:us-east-1:12345678901:stream/
enable_test_stream --update-kinesis-streaming-configuration
 ApproximateCreationDateTimePrecision=MICROSECOND
```

3. describe-kinesis-streaming-destination を呼び出して、ストリームが UPDATING であることを確認します。
4. ストリーミングステータスが再び ACTIVE になるまで describe-kinesis-streaming-destination を定期的に呼び出します。タイムスタンプの精度の更新が有効になるまで、最大 5 分かかることがあります。このステータスが更新されると、更新が完了したことを表し、新しい精度値が今後のレコードに適用されます。
5. putItem を使用してテーブルに書き込みます。
6. Kinesis get-records コマンドを使用して、Kinesis ストリームコンテンツを取得します。
7. 書き込みの ApproximateCreationDateTime の精度が希望どおりであることを確認します。

Java API

1. UpdateKinesisStreamingDestination リクエストと UpdateKinesisStreamingDestination レスポンスを構成するコードスニペットを提供します。
2. DescribeKinesisStreamingDestination リクエストと DescribeKinesisStreamingDestination response を構成するコードスニペットを提供します。
3. ストリーミングのステータスが、更新が完了し、将来のレコードに新しい精度値が適用されることを示す ACTIVE に戻るまで、describe-kinesis-streaming-destination を定期的に呼び出します。
4. テーブルへの書き込みを実行します。
5. ストリームから読み取り、ストリームコンテンツを逆シリアル化します。
6. 書き込みの ApproximateCreationDateTime の精度が希望どおりであることを確認します。

> **Note**
>
> Kinesis データストリームへのストリーミングの無効化は、非同期オペレーションです。送信先を無効にした後 (DisableKinesisStreamingDestination を呼び出すか、コンソールから無効にする)、DynamoDB は変更が完全に伝播するまで、処理中のレコードをデータストリームに短時間送信し続けることができます。ストリーミングを停止するときは、この短い重複を考慮し、describe-kinesis-streaming-destination を使用して送信先のステータスが DISABLED に変更されたことを確認します。

DynamoDB Streams および Kinesis Data Streams でのシャードとメトリクスの使用

Kinesis Data Streams のシャード管理に関する考慮事項

Kinesis データストリームは、シャードでスループットをカウントします。Amazon Kinesis Data Streams では、データストリームにオンデマンドモードとプロビジョンドモードのどちらかを選択できます。

DynamoDB の書き込みワークロードが大きく変動しやすく予測不可能な場合は、Kinesis データストリームにオンデマンドモードを使用することをお勧めします。オンデマンドモードでは、Kinesis Data Streams が必要なスループットを提供するためにシャードを自動的に管理するため、キャパシティプランニングは必要ありません。

予測可能なワークロードの場合は、Kinesis データストリームにプロビジョニングモードを使用できます。プロビジョンドモードでは、DynamoDB からの変更データ取得レコードを格納するためのデータストリームのシャード数を指定する必要があります。Kinesis データストリームが DynamoDB テーブルをサポートするために必要なシャードの数を決定するには、次の入力値が必要です。

- DynamoDB テーブルのレコードの平均サイズ (バイト単位、`average_record_size_in_bytes`)。
- DynamoDB テーブルで実行される 1 秒あたりの書き込み操作の最大数。これには、アプリケーションで実行される作成、削除、更新操作のほか、Time to Live で生成された削除操作 (`write_throughput`) などの自動生成された操作も含まれます。
- テーブルに対して実行する作成操作や削除操作と比較した、更新操作と上書き操作の割合 (`percentage_of_updates`)。更新操作と上書き操作では、変更された項目の古いイメージと新しいイメージの両方がストリームにレプリケートされることに留意してください。これにより、DynamoDB 項目のサイズが 2 倍になります。

Kinesis データストリームに必要なシャードの数 (`number_of_shards`) を計算するには、入力値を以下の式にあてはめます。

```
number_of_shards = ceiling( max( ((write_throughput * (4+percentage_of_updates) *
 average_record_size_in_bytes) / 1024 / 1024), (write_throughput/1000), 1) )
```

例えば、書き込み操作の最大スループットが 1,040 回/秒で (`write_throughput`)、平均レコードサイズが 800 バイト (`average_record_size_in_bytes`) であるとします。 この書き込み操作の 25% が更新操作 (`percentage_of_updates`) である場合、DynamoDB ストリーミングスループットに対応するために 2 つのシャード (`number_of_shards`) が必要になります。

```
ceiling( max( ((1040 * (4+25/100) * 800)/ 1024 / 1024), (1040/1000), 1) ).
```

式を使用して Kinesis データストリームのプロビジョニングモードで必要なシャード数を計算する前に、次の点を考慮してください。

- この式では、DynamoDB 変更データレコードを格納するのに必要なシャードの数を見積もることができます。これは、追加の Kinesis データストリームのコンシューマーをサポートするために必要なシャードの数など、Kinesis データストリームで必要なシャードの総数を表すものではありません。
- ピークスループットを処理するようにデータストリームを設定しないと、プロビジョンドモードで読み取りおよび書き込みのスループットの例外が発生する可能性があります。この場合、データトラフィックに対応するようにデータストリームを手動でスケーリングする必要があります。
- この式では、変更ログデータレコードを Kinesis Data Stream にストリーミングする前に DynamoDB によって生成される追加の肥大化を考慮に入れています。

Kinesis データストリームのキャパシティモードの詳細については、「データストリームのキャパシティモードの選択」を参照してください。さまざまなキャパシティモード間の料金の違いについて詳しくは、「Amazon Kinesis Data Streams の料金」を参照してください。

Kinesis Data Streams を使用した変更データキャプチャのモニタリング

DynamoDB には、Kinesis への変更データキャプチャのレプリケーションをモニタリングするのに役立つ複数の Amazon CloudWatch メトリクスが用意されています。CloudWatch メトリクスの詳細な一覧については、「DynamoDB のメトリクスとディメンション」を参照してください。

ストリームに十分な容量があるかどうかを判断する場合は、ストリームの有効化時と実稼働時の両方で次の項目をモニタリングすることをお勧めします。

- `ThrottledPutRecordCount`: Kinesis データストリームのキャパシティが不足しているために、Kinesis データストリームによってスロットリングされたレコードの数。例外的な使用量のピーク時にスロットリングが発生する可能性がありますが、`ThrottledPutRecordCount` は可能な限り低く保つ必要があります。DynamoDB は、スロットリングされたレコードを Kinesis データストリームに再送信しますが、これによりレプリケーションのレイテンシーが高くなる可能性があります。

  過剰で定期的なスロットリングが発生した場合は、テーブルで観測された書き込みスループットに比例して Kinesis ストリーミングシャードの数を増やす必要があります。Kinesis Data Streams

のサイズ決定の詳細については、「Kinesis Data Streams の初期サイズの決定」を参照してください。

- `AgeOfOldestUnreplicatedRecord`: Kinesis Data Streams にまだレプリケートされていない最も古い項目レベルの変更からの経過時間が DynamoDB テーブルに表示されました。通常のオペレーションでは、`AgeOfOldestUnreplicatedRecord` はミリ秒単位で順序を指定しなければなりません。この数字は、カスタマー管理設定上の選択が原因で失敗したレプリケーションの試行回数に基づいて増加します。

  `AgeOfOldestUnreplicatedRecord` メトリクスが 168 時間を超えると、DynamoDB テーブルから Kinesis データストリームへの項目レベルの変更のレプリケーションは自動的に無効になります。

  カスタマー管理設定でレプリケーション試行の失敗の原因になる例として、Kinesis データストリームキャパシティーのプロビジョニングが不足していたために過剰なスロットリングにつながった場合や、Kinesis データストリームのアクセスポリシーを手動で更新したために DynamoDB がデータストリームにデータを追加できなくなった場合が挙げられます。このメトリクスを可能な限り低く保つには、Kinesis データストリームの容量が適切にプロビジョニングされていること、および DynamoDB のアクセス許可が変更されていないことを確認する必要がある場合があります。

- `FailedToReplicateRecordCount`: DynamoDB が Kinesis データストリームにレプリケートできなかったレコードの数。DynamoDB は、Kinesis データストリームの最大レコードサイズより大きい変更データレコードをレプリケートできません。デフォルトの最大レコードサイズは 1 MB で、最大 10 MB まで増やすことができます。詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の「クォータと制限」を参照してください。

  変更データレコードは、記述する項目よりも大きくなる場合があります。更新オペレーションでは、項目の古いイメージと新しいイメージの両方がレプリケートされ、レコードサイズが約 2 倍になります。ブール値と空の属性値は、DynamoDB に 1 バイトで格納されますが、Kinesis Data Streams レプリケーションで標準 JSON を使用してシリアル化すると、最大 5 バイトまで拡張されます。例えば、34 KB を超える特定の項目は、デフォルトの最大レコードサイズである 1 MB を超える変更データレコードに拡張される可能性があります。DynamoDB は、ストリームの最大レコードサイズを超える変更データレコードをスキップし、後続のレコードを自動的にレプリケートします。

前述のメトリクスのいずれかが特定のしきい値を超えた場合に通知するために、Amazon Simple Notification Service (Amazon SNS) メッセージを送信する Amazon CloudWatch アラームを作成できます。

Amazon Kinesis Data Streams および Amazon DynamoDB の IAM ポリシーを使用する

Amazon DynamoDB 用 Amazon Kinesis Data Streams を初めて有効にすると、DynamoDB は AWS Identity and Access Management (IAM) サービスにリンクされたロールを自動的に作成します。このロール `AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` を使用すると、DynamoDB はユーザーに代わって Kinesis Data Streams への項目レベルの変更のレプリケーションを管理できます。このサービスにリンクされたロールは削除しないでください。

サービスリンクロールの詳細については、「IAM ユーザーガイド」の「サービスリンクロールの使用」を参照してください。

> **Note**
>
> DynamoDB は、IAM ポリシーのタグベースの条件をサポートしていません。

Amazon DynamoDB 用 Amazon Kinesis Data Streams を有効にするには、テーブルに対する以下の許可が必要です。

- `dynamodb:EnableKinesisStreamingDestination`
- `kinesis:ListStreams`
- `kinesis:PutRecords`
- `kinesis:DescribeStream`

特定の DynamoDB テーブルに対する Amazon DynamoDB 用 Amazon Kinesis Data Streams を記述するには、テーブルに対する以下の許可が必要です。

- `dynamodb:DescribeKinesisStreamingDestination`
- `kinesis:DescribeStreamSummary`
- `kinesis:DescribeStream`

Amazon DynamoDB 用 Amazon Kinesis Data Streams を無効にするには、テーブルに対する以下の許可が必要です。

- `dynamodb:DisableKinesisStreamingDestination`

Amazon DynamoDB 用 Amazon Kinesis Data Streams を更新にするには、テーブルに対する以下の許可が必要です。

- `dynamodb:UpdateKinesisStreamingDestination`

以下の例は、IAM ポリシーを使用して Amazon DynamoDB 用 Amazon Kinesis Data Streams の許可を付与する方法を示しています。

例: Amazon DynamoDB 用 Amazon Kinesis Data Streams を有効にする

以下の IAM ポリシーは、`Music` テーブルに対して Amazon DynamoDB 用 Amazon Kinesis Data Streams を有効にする権限を付与します。`Music` テーブルに対して DynamoDB 用 Kinesis Data Streams を無効化、更新、記述する権限は付与されません。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/kinesisreplication.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBKinesisDataStreamsReplication",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName":
 "kinesisreplication.dynamodb.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:EnableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
}
```

例: Amazon DynamoDB 用 Amazon Kinesis Data Streams を更新する

以下の IAM ポリシーは、`Music` テーブルに対して Amazon DynamoDB 用 Amazon Kinesis Data Streams を更新する権限を付与します。`Music` テーブルに対して DynamoDB 用 Kinesis Data Streams を有効化、無効化、記述する権限は付与されません。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
}
```

例: Amazon DynamoDB 用 Amazon Kinesis Data Streams を無効にする

以下の IAM ポリシーは、`Music` テーブルに対して Amazon DynamoDB 用 Amazon Kinesis Data Streams を無効にする権限を付与します。`Music` テーブルに対して Amazon DynamoDB 用 Amazon Kinesis Data Streams を有効化、更新、記述する権限は付与されません。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DisableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        }
    ]
```

```
}
```

例: リソースに基づいて、選択的に Amazon DynamoDB 用 Amazon Kinesis Data Streams に許可を適用する

以下の IAM ポリシーは、`Music` テーブルの Amazon DynamoDB 用 Amazon Kinesis Data Streams を有効にするか、記述する許可を付与しますが、`Orders` テーブルの Amazon DynamoDB 用 Amazon Kinesis Data Streams を無効にする許可は拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:EnableKinesisStreamingDestination",
                "dynamodb:DescribeKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Music"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:DisableKinesisStreamingDestination"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Orders"
        }
    ]
}
```

DynamoDB 用 Kinesis Data Streams に対するサービスにリンクされたロールの使用

Amazon DynamoDB 用 Amazon Kinesis Data Streams は、AWS Identity and Access Management (IAM) のサービスにリンクされたロールを使用します。サービスにリンクされたロールは、一意のタイプの IAM ロールで、DynamoDB 用 Kinesis Data Streams に直接リンクされます。サービスにリンクされたロールは、DynamoDB 用 Kinesis Data Streams によって事前定義されており、お客様の代

わりにサービスから他の AWS のサービスを呼び出す必要のあるアクセス許可がすべて含まれています。

サービスにリンクされたロールを使用することで、必要なアクセス権限を手動で追加する必要がなくなるため、DynamoDB 用 Kinesis Data Streams の設定が簡単になります。DynamoDB 用 Kinesis Data Streams は、サービスにリンクされたロールの許可を定義します。特に定義されている場合を除き、DynamoDB 用 Kinesis Data Streams のみがそのロールを引き受けることができます。定義される許可は、信頼ポリシーと許可ポリシーに含まれており、その許可ポリシーを他の IAM エンティティにアタッチすることはできません。

サービスにリンクされたロールをサポートする他のサービスについては、「IAM と連携する AWS のサービス」を参照して、「サービスにリンクされたロール」列が「はい」になっているサービスを確認してください。サービスにリンクされたロールに関するドキュメントをサービスで表示するには、[はい] リンクを選択します。

DynamoDB 用 Kinesis Data Streams に対するサービスにリンクされたロールの許可

DynamoDB 用 Kinesis Data Streams で
は、AWSServiceRoleForDynamoDBKinesisDataStreamsReplication という名前の、サービスにリンクされたロールを使用します。サービスにリンクされたロールの目的は、Amazon DynamoDB がお客様に代わって Kinesis Data Streams に対する項目レベルの変更のレプリケーションを管理できるように許可することです。

`AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` サービスリンクロールは、ロールの引き受けについて以下のサービスを信頼します。

- `kinesisreplication.dynamodb.amazonaws.com`

ロールのアクセス許可ポリシーは、指定したリソースに対して以下のアクションを完了することを DynamoDB 用 Kinesis Data Streams に許可します。

- アクション: `Kinesis stream` での `Put records and describe`
- アクション: `AWS KMS` で `Generate data keys`、ユーザーが生成した AWS KMS キーを使用して暗号化された Kinesis ストリームにデータを格納します。

ポリシードキュメントの正確な内容については、
「DynamoDBKinesisReplicationServiceRolePolicy」を参照してください。

サービスリンクロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) に許可するにはアクセス許可を設定する必要があります。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの許可」を参照してください。

DynamoDB 用 Kinesis Data Streams のサービスにリンクされたロールの作成

サービスにリンクされたロールを手動で作成する必要はありません。AWS マネジメントコンソール、AWS CLI または AWS API で DynamoDB 用 Kinesis Data Streams を有効にすると、DynamoDB 用 Kinesis Data Streams がサービスにリンクされたロールを作成します。

このサービスにリンクされたロールを削除した後で再度作成する必要が生じた場合は、同じ方法でアカウントにロールを再作成できます。DynamoDB 用 Kinesis Data Streams を有効にすると、DynamoDB 用 Kinesis Data Streams がサービスにリンクされたロールを再作成します。

DynamoDB 用 Kinesis Data Streams のサービスにリンクされたロールの編集

DynamoDB 用 Kinesis Data Streams では、`AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` サービスにリンクされたロールを編集することはできません。サービスにリンクされたロールを作成すると、多くのエンティティによってロールが参照される可能性があるため、ロール名を変更することはできません。ただし、IAM を使用してロールの説明を編集することはできます。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの編集」を参照してください。

DynamoDB 用 Kinesis Data Streams のサービスにリンクされたロールの削除

サービスにリンクされたロールは、IAM コンソール、AWS CLI 、または AWS API を使用して手動で削除することもできます。そのためにはまず、サービスにリンクされたロールのリソースをクリーンアップする必要があります。その後で、手動で削除できます。

> **Note**
>
> リソースを削除する際に、DynamoDB 用 Kinesis Data Streams サービスでロールが使用されている場合、削除は失敗することがあります。失敗した場合は、数分待ってから再度オペレーションを実行してください。

IAM を使用してサービスリンクロールを手動で削除するには

IAM コンソール、AWS CLI、または AWS API を使用して、`AWSServiceRoleForDynamoDBKinesisDataStreamsReplication` サービスリンクロール

を削除します。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの削除」を参照してください。

DynamoDB Streams の変更データキャプチャ

DynamoDB Streams は、DynamoDB テーブル内の項目レベルの変更に関するシーケンスを時間順にキャプチャし、その情報を最大 24 時間ログに保存します。アプリケーションは、このログにアクセスし、データ項目の変更前および変更後の内容をほぼリアルタイムで参照できます。

保管時の暗号化では、DynamoDB Streams のデータが暗号化されます。詳細については、「保管時の DynamoDB 暗号化」を参照してください。

DynamoDB Streams は、DynamoDB テーブル内の項目に加えられた変更に関する情報の順序付けされた情報です。テーブルでストリーミングを有効にすると、DynamoDB はテーブル内のデータ項目に加えられた各変更に関する情報をキャプチャします。

アプリケーションがテーブル内の項目を作成、更新、または削除するたびに、DynamoDB Streams は変更された項目のプライマリキー属性を付けてストリーミングレコードを書き込みます。ストリーミングレコードには、DynamoDB テーブル内の単一の項目に加えられたデータ変更についての情報が含まれています。ストリームレコードが追加情報 (変更された項目の前後のイメージ) をキャプチャするようにストリームを設定できます。

DynamoDB Streams を使用すれば、以下のことを確認できます。

- 各ストリームレコードは、ストリームに 1 回だけ出現します。
- DynamoDB テーブルで変更された各項目について、ストリーミングレコードは項目に対する実際の変更と同じ順序で出現します。

DynamoDB Streams は、ストリーミングレコードをほぼリアルタイムで書き込むため、これらのストリーミングを使用し、内容に基づいてアクションを実行するアプリケーションを構築できます。

トピック

- DynamoDB Streams のエンドポイント
- ストリームの有効化
- ストリームの読み込みと処理
- DynamoDB Streams と有効期限 (TTL)
- DynamoDB Streams のタグ付け
- DynamoDB Streams Kinesis Adapter を使用したストリームレコードの処理

- DynamoDB Streams 低レベル API: Java の例
- DynamoDB Streams と AWS Lambda のトリガー
- DynamoDB Streams と Apache Flink

DynamoDB Streams のエンドポイント

AWS では、DynamoDB と DynamoDB Streams 用に個別のエンドポイントを維持しています。データベースのテーブルとインデックスを使用するには、アプリケーションが DynamoDB エンドポイントにアクセスする必要があります。DynamoDB Streams レコードを読み込んで処理するには、アプリケーションが同じリージョンの DynamoDB Streams エンドポイントにアクセスする必要があります。

DynamoDB Streams には 2 つのエンドポイントセットが用意されています。具体的には次の 2 つです。

- IPv4 専用エンドポイント: `streams.dynamodb.<region>.amazonaws.com` 命名規則を持つエンドポイント。
- デュアルスタックエンドポイント: IPv4 と IPv6 の両方と互換性があり、`streams-dynamodb.<region>.api.aws` 命名規則に従う新しいエンドポイント。

> **Note**
>
> DynamoDB および DynamoDB Streams のリージョンとエンドポイントの完全なリストについては、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。

AWS SDK は、DynamoDB と DynamoDB Streams 用に個別のクライアントを提供します。要件によっては、アプリケーションは、DynamoDB エンドポイント、DynamoDB Streams エンドポイント、または両方に同時にアクセスできます。両方のエンドポイントに接続するには、アプリケーションで 2 つのクライアントをインスタンス化する必要があります。1 つは DynamoDB 用、もう 1 つは DynamoDB Streams 用です。

ストリームの有効化

新しいテーブルでは、AWS CLI または AWS SDK 経由でそのテーブルの作成時にストリームを有効にできます。また、既存のテーブルでストリーミングを有効または無効にすることや、ストリーミン

グの設定を変更することができます。DynamoDB Streams は非同期的に動作するため、ストリーミングを有効にしてもテーブルのパフォーマンスに影響はありません。

DynamoDB Streams を管理する最も簡単な方法は、AWS マネジメントコンソール を使用することです。

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. DynamoDB コンソールのダッシュボードで、[Tables (テーブル)] を選択して既存テーブルを選びます。
3. [エクスポートとストリーム] タブを選択します。
4. [DynamoDB ストリームの詳細] セクションで、[オンにする] を選択します。
5. [DynamoDB ストリームをオンにする] ウィンドウで、テーブルのデータが変更されるたびにストリーミングに書き込まれる情報を選択します。

   - [キー属性のみ] - 変更された項目のキー属性のみ。
   - [New image] (新規イメージ) — 変更後に表示される項目全体。
   - [Old image] (古いイメージ) — 変更前に表示されていた項目全体。
   - [New and old images] (新規イメージおよび古いイメージ) — 項目の新しいイメージと古いイメージの両方。

   すべての設定が正しいことを確認したら、[ストリームをオンにする] を選択します。
6. (オプション) 既存のストリーミングを無効にするには、[DynamoDB ストリームの詳細] で [オフにする] を選択します。

`CreateTable` または `UpdateTable` API オペレーションを使用して、ストリームを有効にするか、変更することもできます。ストリームの設定内容は、`StreamSpecification` パラメータにより決まります。

- `StreamEnabled` — テーブルでストリーミングが有効 (`true`) か無効 (`false`) かを指定します。
- `StreamViewType` — テーブル内のデータが変更されるたびにストリーミングに書き込まれる情報を指定します。
  - `KEYS_ONLY` — 変更された項目のキー属性のみ。
  - `NEW_IMAGE` — 変更後に表示される項目全体。
  - `OLD_IMAGE` — 変更前に表示されていた項目全体。

- NEW_AND_OLD_IMAGES — 項目の新しいイメージと古いイメージの両方。

ストリームはいつでも有効または無効にできます。ただし、既にストリームがあるテーブルでストリームを有効にしようとした場合、ValidationException を受け取ります。また、ストリームのないテーブルでストリームを無効にしようとすると、ValidationException が発生します。

StreamEnabled を true に設定すると、一意のストリーミング記述子が割り当てられた新しいストリーミングが DynamoDB で作成されます。テーブルでストリームを無効にして再度有効にすると、新しいストリームは異なるストリーム記述子で作成されます。

各ストリームは、Amazon リソースネーム (ARN) により一意に識別されます。次に、TestTable という名前の DynamoDB テーブルにあるストリーミングのサンプル ARN を示します。

```
arn:aws:dynamodb:us-west-2:111122223333:table/TestTable/stream/2015-05-11T21:21:33.291
```

テーブルの最新のストリーミング記述子を調べるには、DynamoDB DescribeTable リクエストを発行し、レスポンスで LatestStreamArn 要素を探します。

> **Note**
>
> ストリームのセットアップ後は StreamViewType を編集できません。セットアップ後にストリームを変更する必要がある場合は、現在のストリームを無効にして新しいストリームを作成する必要があります。

ストリームの読み込みと処理

ストリームを読み取って処理するには、アプリケーションから DynamoDB Streams エンドポイントに接続して API リクエストを発行する必要があります。

ストリームは、ストリームレコードで構成されています。各ストリーミングレコードは、ストリーミングが属する DynamoDB テーブル内の 1 件のデータ変更を表しています。各ストリームレコードには、レコードがストリームに発行された順序を反映したシーケンス番号が割り当てられます。

ストリームレコードは、グループ (つまり、シャード) に整理されます。各シャードは、複数のストリームレコードのコンテナとして機能し、これらのレコードへのアクセスと反復処理に必要な情報が含まれています。シャード内のストリームレコードは 24 時間後に自動的に削除されます。

シャードは エフェメラルであり、必要に応じて自動的に作成および削除されます。また、任意のシャードは複数の新しいシャードに分割できます。これもまた自動的に行われます (親シャードが 1 つの子シャードのみを持つ場合もあります)。アプリケーションが複数のシャードからレコードを並列処理できるように、シャードは親テーブルで高レベルな書き込みアクティビティに応じて分割される場合があります。

ストリームを無効にすると、開かれているシャードは閉じられます。ストリーミング内のデータは 24 時間読み込み可能な状態になります。

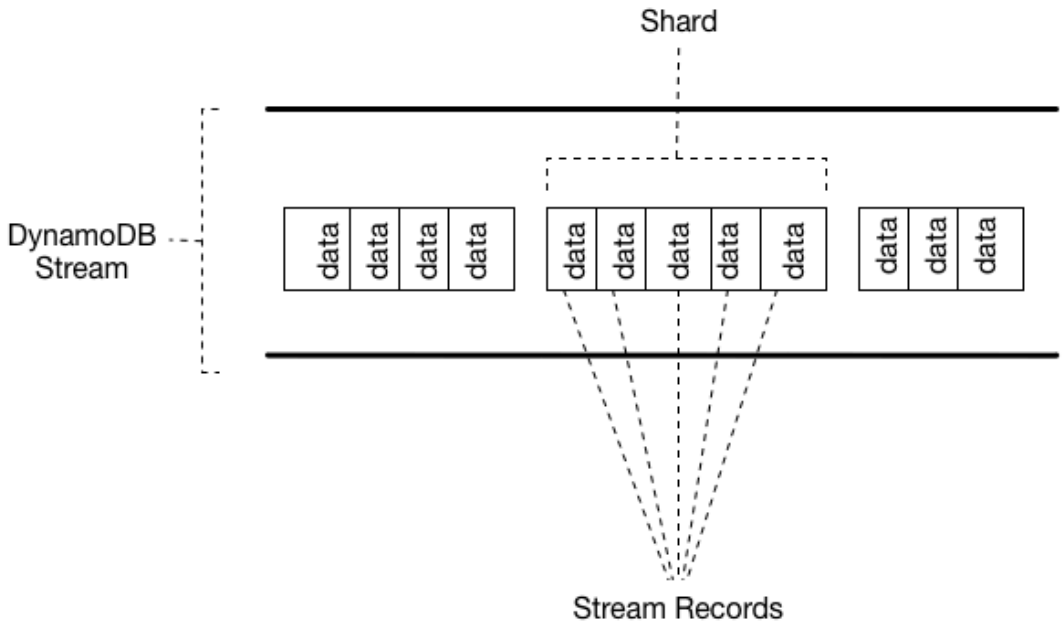
シャードには系列 (親と子) があるため、アプリケーションは子シャードを処理する前に、必ず親シャードを処理する必要があります。これにより、ストリームレコードも正しい順序で処理されるようになります。(DynamoDB Streams Kinesis Adapter を使用している場合、これは自動的に処理されます。アプリケーションは、シャードとストリーミングレコードを正しい順序で処理します。アプリケーションの実行中に分割されたシャードに加えて、新しいシャードまたは有効期限切れのシャードは自動的に処理されます。詳細については、「DynamoDB Streams Kinesis Adapter を使用したストリームレコードの処理」を参照してください。)

各オープンシャードは厳密に 1 つのテーブルに対応します (DynamoDB におけるパーティションとデータ分散)。特定のパーティションはストリームレコードを単一の専用シャードに書き込み、他のパーティションはそのシャードに書き込みません。DynamoDB は、より多くのデータまたはスループットを処理するためにテーブルにパーティションを追加する際、一致するシャードを追加するため、ストリームはテーブルとともにスケールします。

AWS Lambda を使用してイベントソースマッピングを介してストリームを消費すると、Lambda は単一の関数インスタンスで各オープンシャードを処理し、そのシャードのレコードをシーケンス番号順に読み取ります。また、Lambda はシャードの系譜に従い、子シャードの前に親シャードを処理します。デフォルトでは、1 つの関数インスタンスが 1 つのシャードを処理します。`ParallelizationFactor` (最大 10) を増やすことで、各項目の変更順序を維持しつつ、Lambda が複数の同時インスタンスを使用して 1 つのシャードを処理するように設定できます。

DynamoDB Streams は、パーティション全体ではなく、個々の項目レベルで、つまり同じプライマリキー (パーティションキー、またはパーティションキーとソートキー) に対するすべての変更にわたって順序付けを保証します。パーティションキーを共有する項目コレクションは複数のパーティションにまたがる可能性があるため、DynamoDB は項目コレクション全体ではなく、各項目の変更順序を保持します。

次の図は、ストリーム、ストリーム内のシャード、シャード内のストリームレコードの関係を示しています。



> **Note**
>
> 項目内のデータを何も変更しない `PutItem` または `UpdateItem` オペレーションを実行した場合、そのオペレーションのストリーミングレコードは DynamoDB Streams によって書き込まれません。

ストリームにアクセスしてその中のストリームレコードを処理するには、以下の操作を実行する必要があります。

- アクセスするストリームの一意の ARN を調べます。
- 目的のストリームレコードがストリーム内のどのシャードに含まれているかを調べます。
- シャードにアクセスし、目的のストリームレコードを取得します。

> **Note**
>
> 最大でも 2 つを超えるプロセスが、同時に同じストリームシャードから読み込みを行うことはできません。シャードごとに 2 つを超えるリーダーがあると、スロットリングが発生する場合があります。

DynamoDB Streams API は、アプリケーションプログラム用の以下のアクションを提供します。

- ListStreams — 現在のアカウントおよびエンドポイントのストリーミング記述子のリストを返します。必要に応じて、特定のテーブル名のストリーム記述子だけをリクエストできます。
- DescribeStream — ストリームに関する情報 (例: ストリームの最新ステータス、Amazon リソースネーム (ARN)、シャードの構成、対応する DynamoDB テーブル) を返します。オプションで ShardFilter フィールドを使用して、親シャードに関連付けられた既存の子シャードを取得できます。
- GetShardIterator — シャード内の場所を表すシャードイテレーターを返します。イテレータがストリーム内の最も古いポイント、最も新しいポイント、特定のポイントへのアクセスを提供することをリクエストできます。
- GetRecords — 特定のシャード内からストリーミングレコードを返します。GetShardIterator リクエストから返されたシャードイテレーターを指定する必要があります。

リクエストやレスポンスの例など、これらの API オペレーションの詳細な説明については、「Amazon DynamoDB Streams API リファレンス」を参照してください。

シャード検出

2 つの強力な方法で、DynamoDB ストリーム内の新しいシャードを検出します。Amazon DynamoDB Streams ユーザーには、新しいシャードを追跡して識別するための 2 つの効果的な方法があります。

ストリームトポロジ全体のポーリング

DescribeStream API を使用して、ストリームを定期的にポーリングします。これにより、作成された新しいシャードを含む、ストリーム内のすべてのシャードが返されます。時間経過に伴う結果を比較することで、新しく追加されたシャードを検出できます。

子シャードの検出

DescribeStream API を ShardFilter パラメータとともに使用して、シャードのサブセットを見つけます。リクエストで親シャードを指定することで、DynamoDB Streams は直下の子シャードを返します。このアプローチは、ストリーム全体をスキャンせずにシャード系統のみを追跡する必要がある場合に役立ちます。

DynamoDB Streams からデータを使用するアプリケーションは、この ShardFilter パラメータを使用して閉じたシャードの読み取りから子シャードに効率的に移行でき、DescribeStream API への繰り返しの呼び出しを回避して、すべての閉じたシャードと開いているシャードの

シャードマップを取得してトラバースできます。これにより、親シャードが閉じられた後に子シャードをすばやく検出できるため、ストリーム処理アプリケーションの応答性とコスト効率が向上します。

どちらの方法でも、DynamoDB Streams の進化する構造を常に把握できるため、重要なデータの更新やシャードの変更を見逃すことはありません。

DynamoDB Streams のデータ保持期限

DynamoDB Streams 内のすべてのデータは、24 時間保持されます。特定のテーブルの直近 24 時間のアクティビティを取得して分析できます。ただし、24 時間を超えたデータはすぐにトリミング (削除) される可能性があります。

テーブルのストリームを無効にした場合、ストリーム内のデータは 24 時間読み込み可能な状態になります。この時間が経過すると、データは期限切れになり、ストリームレコードは自動的に削除されます。既存のストリームを手動で削除するためのメカニズムはありません。保持期限 (24 時間) が切れ、すべてのストリームレコードが削除されるまで待つ必要があります。

DynamoDB Streams と有効期限 (TTL)

テーブルに対して Amazon DynamoDB Streams を有効にし、期限切れの項目のストリーミングレコードを処理することで、有効期限 (TTL) によって削除された項目をバックアップ (または処理) できます。詳細については、「ストリームの読み込みと処理」を参照してください。

ストリームレコードにはユーザー ID フィールド `Records[<index>].userIdentity` が含まれます。

有効期限切れの後に有効期限 (TTL) プロセスによって削除された項目には、次のフィールドが含まれています。

- `Records[<index>].userIdentity.type`

  `"Service"`
- `Records[<index>].userIdentity.principalId`

  `"dynamodb.amazonaws.com"`

> **Note**
>
> TTL をグローバルテーブルで使用すると、TTL が実行されたリージョンに `userIdentity` フィールドが設定されます。削除が複製されても、このフィールドは他のリージョンには設定されません。

次の JSON は 1 つのストリームレコードの関連する部分を示しています。

```
"Records": [
    {
        ...

        "userIdentity": {
            "type": "Service",
            "principalId": "dynamodb.amazonaws.com"
        }

        ...

    }
]
```

DynamoDB Streams と Lambda を使用して TTL 削除済みアイテムをアーカイブする

DynamoDB 有効期限 (TTL)、DynamoDB Streams、および AWS Lambda を組み合わせると、データのアーカイブを簡素化し、DynamoDB のストレージコストを削減し、コードの複雑さを軽減するのに役立ちます。ストリームコンシューマーとして Lambda を使用すると、Kinesis Client Library (KCL) などの他のコンシューマーと比較してコストが削減されるなど、多くの利点があります。Lambda を使用してイベントを消費する場合、DynamoDB ストリームの `GetRecords` API 呼び出しでは課金が発生しません。Lambda はストリームイベント内の JSON パターンを識別してイベントフィルタリングを提供できます。イベントパターンのコンテンツフィルタリングでは、最大 5 つの異なるフィルターを定義して、処理のために Lambda に送信されるイベントを制御できます。これにより、Lambda 関数の呼び出しを減らしてコードを簡素化し、全体的なコストを削減できます。

DynamoDB Streams には、`Create`、`Modify` および`Remove` アクションなどのすべてのデータ変更が含まれています。これは、アーカイブ Lambda 関数の不要な呼び出しを引き起こす可能性があります。例えば、1 時間あたり 200 万件のデータ変更がストリームに流れ込むテーブルがあり、そ

のうち 5% 未満が TTL プロセスによって期限切れになり、アーカイブする必要があるアイテム削除であるとします。Lambda イベントソースフィルターを使用すると、Lambda 関数は 1 時間あたり 100,000 回しか呼び出されません。イベントフィルタリングを使用した結果、イベントフィルタリングを行わなければ 200 万回の呼び出しが発生するところを、必要な呼び出しに対してのみ課金されることになります。

イベントフィルタリングは、Lambda イベントソースマッピングに適用されます。これは、選択されたイベント (DynamoDB ストリーム) から読み取り、Lambda 関数を呼び出すリソースです。次の図は、ストリームとイベントフィルターを使用して Lambda 関数によって有効期限 (TTL) 削除済みアイテムがどのように消費されるかを示しています。



DynamoDB の有効期限 (TTL) イベントフィルターパターン

イベントソースマッピングのフィルター条件に次の JSON を追加することで、TTL 削除済みアイテムに対してのみ Lambda 関数の呼び出しを許可します。

```
{
    "Filters": [
        {
            "Pattern": { "userIdentity": { "type": ["Service"], "principalId":
 ["dynamodb.amazonaws.com"] } }
        }
    ]
}
```

AWS Lambda イベントソースマッピングを作成します。

次のコードスニペットを使用して、テーブルの DynamoDB ストリームに接続できる、フィルター処理されたイベントソースマッピングを作成します。各コードブロックには、イベントフィルターパターンが含まれます。

AWS CLI

```
aws lambda create-event-source-mapping \
--event-source-arn 'arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000' \
```

```
--batch-size 10 \
--enabled \
--function-name test_func \
--starting-position LATEST \
--filter-criteria '{"Filters": [{"Pattern": "{\"userIdentity\":{\"type\":[\"Service
\"],\"principalId\":[\"dynamodb.amazonaws.com\"]}}"}]}'
```

Java

```
LambdaClient client = LambdaClient.builder()
        .region(Region.EU_WEST_1)
        .build();

Filter userIdentity = Filter.builder()
        .pattern("{\"userIdentity\":{\"type\":[\"Service\"],\"principalId\":
[\"dynamodb.amazonaws.com\"]}}")
        .build();

FilterCriteria filterCriteria = FilterCriteria.builder()
        .filters(userIdentity)
        .build();

CreateEventSourceMappingRequest mappingRequest =
 CreateEventSourceMappingRequest.builder()
        .eventSourceArn("arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000")
        .batchSize(10)
        .enabled(Boolean.TRUE)
        .functionName("test_func")
        .startingPosition("LATEST")
        .filterCriteria(filterCriteria)
        .build();

try{
    CreateEventSourceMappingResponse eventSourceMappingResponse =
 client.createEventSourceMapping(mappingRequest);
    System.out.println("The mapping ARN is
 "+eventSourceMappingResponse.eventSourceArn());

}catch (ServiceException e){
    System.out.println(e.getMessage());
}
```

Node

```
const client = new LambdaClient({ region: "eu-west-1" });

const input = {
    EventSourceArn: "arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000",
    BatchSize: 10,
    Enabled: true,
    FunctionName: "test_func",
    StartingPosition: "LATEST",
    FilterCriteria: { "Filters": [{ "Pattern": "{\"userIdentity\":{\"type\":
[\"Service\"],\"principalId\":[\"dynamodb.amazonaws.com\"]}}" }] }
}

const command = new CreateEventSourceMappingCommand(input);

try {
    const results = await client.send(command);
    console.log(results);
} catch (err) {
    console.error(err);
}
```

Python

```
session = boto3.session.Session(region_name = 'eu-west-1')
client = session.client('lambda')

try:
    response = client.create_event_source_mapping(
        EventSourceArn='arn:aws:dynamodb:eu-west-1:012345678910:table/test/
stream/2021-12-10T00:00:00.000',
        BatchSize=10,
        Enabled=True,
        FunctionName='test_func',
        StartingPosition='LATEST',
        FilterCriteria={
            'Filters': [
                {
                    'Pattern': "{\"userIdentity\":{\"type\":[\"Service\"],
\"principalId\":[\"dynamodb.amazonaws.com\"]}}"
                },
```

```
            ]
        }
    )
    print(response)
except Exception as e:
    print(e)
```

JSON

```
{
  "userIdentity": {
     "type": ["Service"],
     "principalId": ["dynamodb.amazonaws.com"]
   }
}
```

DynamoDB Streams のタグ付け

> ⚠ 作成時のタグ付けはサポートされていません
>
> DynamoDB Streams は現在、作成時のタグ付けをサポートしていません。既存のストリームにのみタグを追加できます。

Amazon DynamoDB コンソール、AWS Command Line Interface (AWS CLI)、または AWS SDK を使用して、既存の DynamoDB Streams にタグを追加、一覧表示、または管理できます。次に、これらのユーザー定義タグをアクティブ化し、コスト配分の追跡のため、AWS Billing and Cost Management コンソールに表示できます。詳細については、「DynamoDB タグを使用して、コスト配分レポートを作成する」を参照してください。

トピック

- 既存のストリームのタグ付け (AWS マネジメントコンソール)
- 既存のストリームのタグ付け (AWS CLI)

既存のストリームのタグ付け (AWS マネジメントコンソール)

DynamoDB コンソールを使用して、既存のストリームのタグを追加、編集、または削除できます。

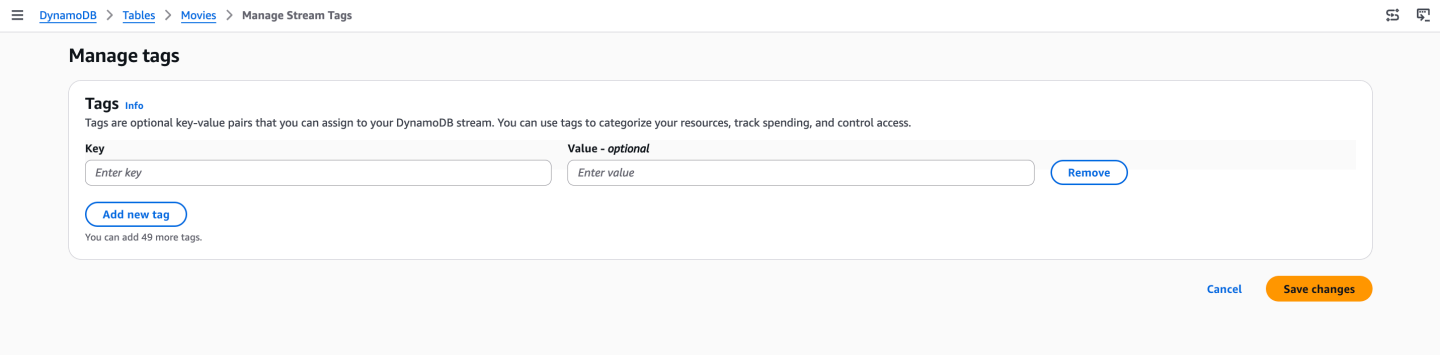
既存のストリーム (コンソール) にタグ付けするには

https://console.aws.amazon.com/dynamodb/ で DynamoDB コンソールを開きます。

1. ナビゲーションペインで [テーブル] を選択します。
2. ストリームにタグを付けるテーブルを選択します。
3. [エクスポートとストリーム] タブを選択します。
4. [DynamoDB ストリームの詳細] セクションで、[タグの管理] を選択します。
5. 必要なタグを追加し、[保存] を選択します。

   タグ構造の詳細については、「DynamoDB でのタグ付けの制限」を参照してください。

次のスクリーンショットは、DynamoDB ストリームの [タグを管理] ページを示しています。ここでは、キーと値のタグペアを追加し、[変更を保存] を選択します。



既存のストリームのタグ付け (AWS CLI)

次の例は、AWS CLI を使用して既存のストリームにタグを付け、ストリームのタグを一覧表示する方法を示しています。

既存のストリーム (AWS CLI) にタグ付けするには

- 次の例では、`Movies` テーブルの既存のストリームに、値が `blueTeam` の `Owner` タグを追加します。

```
aws dynamodb tag-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:111122223333:table/Movies/
stream/2024-01-01T00:00:00.000 \
    --tags Key=Owner,Value=blueTeam
```

ストリームのすべてのタグを一覧表示するには (AWS CLI)

- 次の例では、Movies テーブルのストリームに関連付けられたすべてのタグを一覧表示します。

```
aws dynamodb list-tags-of-resource \
    --resource-arn arn:aws:dynamodb:us-east-1:111122223333:table/Movies/
stream/2024-01-01T00:00:00.000
```

DynamoDB Streams Kinesis Adapter を使用したストリームレコードの処理

Amazon DynamoDB からのストリーミングを使用するには、Amazon Kinesis Adapter を使用することをお勧めします。DynamoDB Streams API は、意図的に Kinesis Data Streams と似せています。どちらのサービスでも、データストリーミングは、ストリーミングレコードのコンテナであるシャードで構成されています。両方のサービスの API には ListStreams、DescribeStream、GetShards、および GetShardIterator オペレーションが含まれています。(これらの DynamoDB Streams アクションは、Kinesis Data Streams の対応するアクションと似ていますが、100% 同一ではありません)。

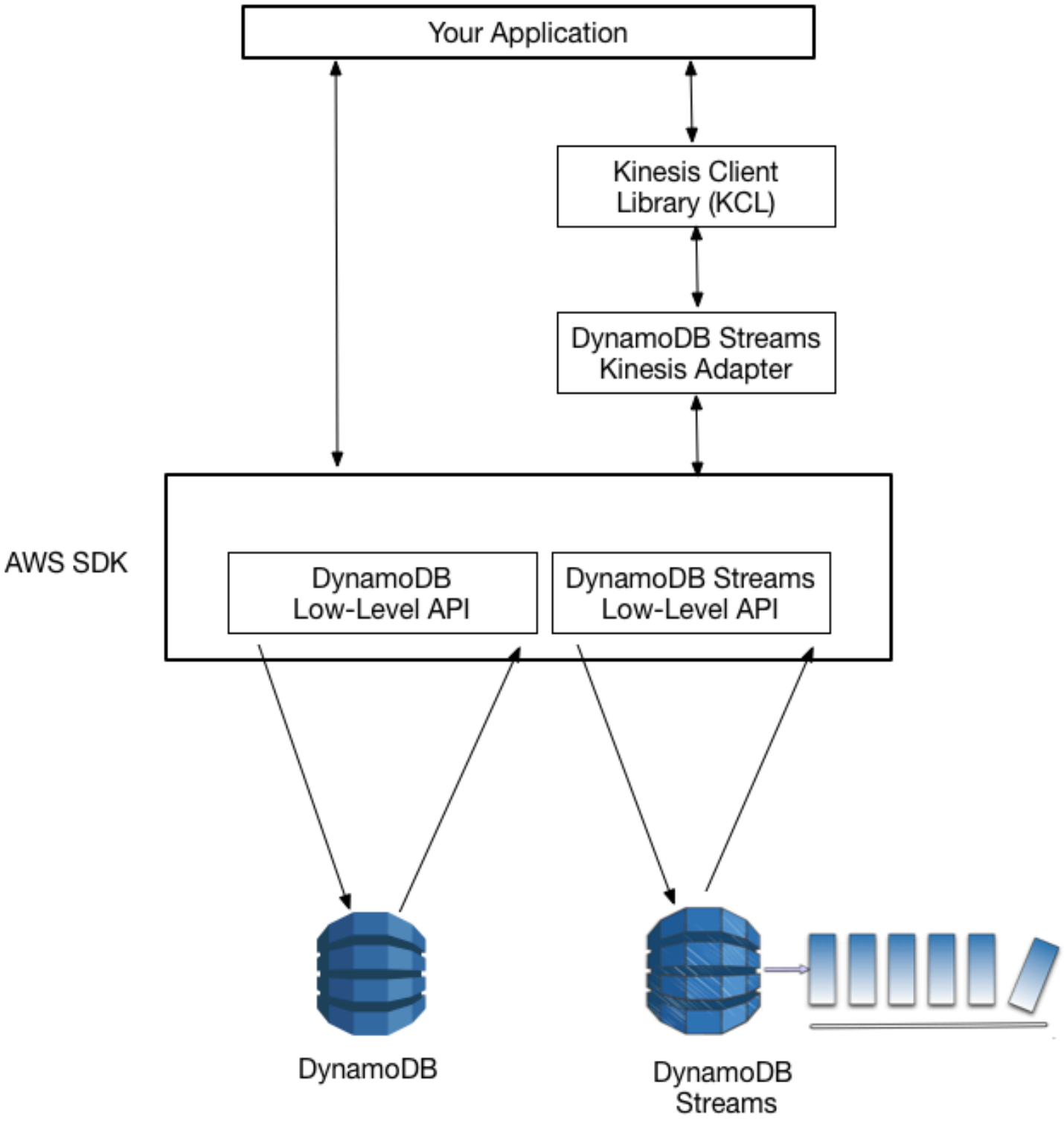
DynamoDB Streams ユーザーは、KCL 内のデザインパターンを使用して DynamoDB Streams のシャードとストリーミングレコードを処理できます。これを行うには、DynamoDB Streams Kinesis Adapter を使用します。Kinesis Adapter は Kinesis Data Streams インターフェイスを実装しているため、KCL を使用して DynamoDB Streams からのレコードを消費および処理することができます。DynamoDB Streams Kinesis アダプターのセットアップとインストールの方法については、「GitHub リポジトリ」を参照してください。

Kinesis Client Library (KCL) を使用して、Kinesis Data Streams のアプリケーションを書き込むことができます。KCL は、低レベルの Kinesis Data Streams API の上で役に立つ抽象化を提供することによりコーディングを簡素化します。KCL の詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の「Kinesis Client Library を使用したコンシューマーの開発」を参照してください。

DynamoDB では、AWS SDK for Java v2.x で KCL バージョン 3.5.x 以降を使用することをお勧めします。現在の DynamoDB Streams Kinesis Adapter バージョン 1.x と AWS SDK for Java v1.x は、AWS SDK とツールのメンテナンスポリシーに従って、移行期間中、意図したとおりライフサイクルを通じて引き続き完全にサポートされます。

> **Note**
>
> Amazon Kinesis Client Library (KCL) バージョン 1.x および 2.x は古くなっています。KCL 1.x は 2026 年 1 月 30 日にサポートが終了します。2026 年 1 月 30 日より前に、バージョン 1.x を使用して KCL アプリケーションを最新の KCL バージョンに移行することを強くお勧めします。最新の KCL バージョンを確認するには、GitHub の「Amazon Kinesis Client Library」ページを参照してください。最新の KCL バージョンの詳細については、「Kinesis Client Library」を参照してください。KCL 1.x から KCL 3.5.x 以降への移行については、「KCL 1.x から KCL 3.5.x 以降への移行」を参照してください。

次の図表は、これらのライブラリがどのように連携するかを示しています。



DynamoDB Streams Kinesis Adapter を使用すれば、DynamoDB Streams エンドポイントにシームレスに誘導された API コールを使用して、KCL インターフェイスに対して開発を開始できます。

アプリケーションは起動時に KCL をコールしてワーカーをインスタンス化します。ワーカーに、アプリケーションの構成情報 (ストリーミング記述子や AWS 認証情報、および提供するレコードプロセッサクラスの名前など) を提供する必要があります。レコードプロセッサでコードを実行すると、ワーカーは次のタスクを実行します。

- ストリームに接続する
- ストリーミング内のシャードを列挙します。
- ストリーム内の閉じた親シャードの子シャードをチェックして列挙します
- シャードと他のワーカー (存在する場合) の関連付けを調整する
- レコードプロセッサで管理する各シャードのレコードプロセッサをインスタンス化する
- ストリーミングからレコードを取得します。
- 高スループット時の GetRecords API 呼び出しレートをスケーリングします (キャッチアップモードが設定されている場合)
- 対応するレコードプロセッサにレコードを送信する
- 処理されたレコードのチェックポイントを作成する
- ワーカーのインスタンス数が変化したときに、シャードとワーカーの関連付けを調整する
- シャードが分割されたときに、シャードとワーカーの関連付けを調整します。

KCL アダプターは、一時的なスループットの増加を処理するための自動呼び出しレート調整機能であるキャッチアップモードをサポートしています。ストリーム処理の遅延が設定可能なしきい値 (デフォルトは 1 分) を超えると、キャッチアップモードでは GetRecords API の呼び出し頻度が設定可能な値 (デフォルトは 3 倍) でスケーリングされ、レコードの取得速度が上がり、遅延が減少すると通常の状態に戻ります。これは、デフォルトのポーリングレートを使用しているコンシューマーが DynamoDB 書き込みアクティビティに圧迫される可能性がある高スループット期間中に役立ちます。キャッチアップモードは、`catchupEnabled` 設定パラメータ (デフォルトは false) を使用して有効にできます。

> **Note**
>
> こちらに記載されている KCL 概念の説明については、「Amazon Kinesis Data Streams デベロッパーガイド」の「Kinesis Client Library を使用したコンシューマーの開発」を参照してください。
> AWS Lambda でのストリームの使用の詳細については、「DynamoDB Streams と AWS Lambda のトリガー」を参照してください。

KCL 1.x から KCL 3.5.x 以降への移行

概要

このガイドでは、コンシューマーアプリケーションを KCL 1.x から KCL 3.5.x 以降に移行する手順について説明します。KCL 1.x と KCL 3.5.x 以降のアーキテクチャの違いにより、移行では互換性を確保するために複数のコンポーネントを更新する必要があります。

KCL 1.x は、KCL 3.5.x 以降とは異なるクラスとインターフェイスを使用します。まずレコードプロセッサ、レコードプロセッサファクトリ、ワーカークラスを KCL 3.5.x 以降と互換性のある形式に移行し、KCL 1.x から KCL 3.5.x 以降への移行手順に従う必要があります。

> **Note**
>
> KCL 3.5.x 以降は、GitHub ウェブサイトの Amazon DynamoDB Streams Kinesis Adapter バージョン 2.4.x 以降でサポートされています。

移行手順

トピック

- ステップ 1: レコードプロセッサを移行する
- ステップ 2: レコードプロセッサファクトリーを移行する
- ステップ 3: ワーカーを移行する
- ステップ 4: KCL 3.5.x 以降の設定の概要と推奨事項
- ステップ 5: KCL 2.x から KCL 3.5.x 以降に移行する

ステップ 1: レコードプロセッサを移行する

以下の例は、KCL 1.x DynamoDB Streams Kinesis Adapter に実装されたレコードプロセッサを示しています。

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

```
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;

public class StreamsRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        for (Record record : processRecordsInput.getRecords()) {
            String data = new String(record.getData().array(),
 Charset.forName("UTF-8"));
            System.out.println(data);
            if (record instanceof RecordAdapter) {
                // record processing and checkpointing logic
            }
        }
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
                shutdownInput.getCheckpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
        try {
            checkpointer.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
```

```
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
}
```

RecordProcessor クラスを移行するには

1. インターフェイスを com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor および com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificat から com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsSha に変更します。以下に例を示します。

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;

import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProce
```

2. initialize メソッド processRecords とメソッドの import ステートメントを更新します。

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;

// import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
```

3. shutdownRequested メソッドを以下の新しいメソッドに置き換えます。leaseLost、shardEnded、および shutdownRequested。

```
//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
```

```
//        // This is moved to shardEnded(...) and
 shutdownRequested(ShutdownReauestedInput)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
```

以下に示しているのは、レコードプロセッサのクラスの更新されたバージョンです。

```
package com.amazonaws.codesamples;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import
 software.amazon.dynamodb.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor;
import
 software.amazon.dynamodb.streamsadapter.adapter.DynamoDBStreamsKinesisClientRecord;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor
import
 com.amazonaws.services.dynamodbv2.streamsadapter.adapter.DynamoDBStreamsClientRecord;
import software.amazon.awssdk.services.dynamodb.model.Record;

public class StreamsRecordProcessor implements DynamoDBStreamsShardRecordProcessor {

    @Override
    public void initialize(InitializationInput initializationInput) {

    }

    @Override
    public void processRecords(DynamoDBStreamsProcessRecordsInput processRecordsInput)
 {
        for (DynamoDBStreamsKinesisClientRecord record: processRecordsInput.records())
            Record ddbRecord = record.getRecord();
            // processing and checkpointing logic for the ddbRecord
        }
    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
```

```
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

> **Note**
>
> DynamoDB Streams Kinesis Adapter は SDKv2 レコードモデルを使用するようになりました。SDKv2 では、複雑な `AttributeValue` オブジェクト (BS、NS、M、L、SS) が null を返すことはありません。`hasBs()`、`hasNs()`、`hasM()`、`hasL()`、`hasSs()` メソッドを使用して、これらの値が存在するかどうかを確認します。

ステップ 2: レコードプロセッサファクトリーを移行する

レコードプロセッサファクトリーは、リースが取得された際にレコードプロセッサの作成を担当します。以下に示しているのは、KCL 1.x ファクトリーの例です。

```
package com.amazonaws.codesamples;

import software.amazon.dynamodb.AmazonDynamoDB;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
```

```
public class StreamsRecordProcessorFactory implements IRecordProcessorFactory {

    @Override
    public IRecordProcessor createProcessor() {
        return new StreamsRecordProcessor(dynamoDBClient, tableName);
    }
}
```

RecordProcessorFactory を移行するには

- 実装されているインターフェイスを com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFa から software.amazon.kinesis.processor.ShardRecordProcessorFactory に変更します。以下に例を示します。

```
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

// public class TestRecordProcessorFactory implements IRecordProcessorFactory {
public class StreamsRecordProcessorFactory implements ShardRecordProcessorFactory {

Change the return signature for createProcessor.

// public IRecordProcessor createProcessor() {
public ShardRecordProcessor shardRecordProcessor() {
```

以下は、3.5.x 以降のレコードプロセッサファクトリーの例です。

```
package com.amazonaws.codesamples;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class StreamsRecordProcessorFactory implements ShardRecordProcessorFactory {
```

```
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new StreamsRecordProcessor();
    }
}
```

ステップ 3: ワーカーを移行する

バージョン 3.5.x 以降の KCL では、新しいクラス Scheduler によって Worker クラスが置き換えられます。KCL 1.x のワーカーの例を次に示します。

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = StreamsWorkerFactory.createDynamoDbStreamsWorker(
        recordProcessorFactory,
        workerConfig,
        adapterClient,
        amazonDynamoDB,
        amazonCloudWatchClient);
```

ワーカーを移行するには

1. import クラスの Worker ステートメントを Scheduler クラスと ConfigsBuilder クラスのインポートステートメントに変更します。

   ```
   // import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
   import software.amazon.kinesis.coordinator.Scheduler;
   import software.amazon.kinesis.common.ConfigsBuilder;
   ```

2. StreamTracker をインポートし、StreamsWorkerFactory のインポートを StreamsSchedulerFactory に変更します。

   ```
   import software.amazon.kinesis.processor.StreamTracker;
   // import software.amazon.dynamodb.streamsadapter.StreamsWorkerFactory;
   import software.amazon.dynamodb.streamsadapter.StreamsSchedulerFactory;
   ```

3. アプリケーションを起動するポジションを選択します。TRIM_HORIZON、LATEST のいずれかになります。

   ```
   import software.amazon.kinesis.common.InitialPositionInStream;
   import software.amazon.kinesis.common.InitialPositionInStreamExtended;
   ```

4. StreamTracker インスタンスを作成します。

```
StreamTracker streamTracker = StreamsSchedulerFactory.createSingleStreamTracker(
        streamArn,

 InitialPositionInStreamExtended.newInitialPosition(InitialPositionInStream.TRIM_HORIZON)
);
```

5. AmazonDynamoDBStreamsAdapterClient オブジェクトを作成します。

```
import software.amazon.dynamodb.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;

...

AwsCredentialsProvider credentialsProvider = DefaultCredentialsProvider.create();

AmazonDynamoDBStreamsAdapterClient adapterClient = new
 AmazonDynamoDBStreamsAdapterClient(
        credentialsProvider, awsRegion);
```

6. ConfigsBuilder オブジェクトを作成します。

```
import software.amazon.kinesis.common.ConfigsBuilder;

...
ConfigsBuilder configsBuilder = new ConfigsBuilder(
                streamTracker,
                applicationName,
                adapterClient,
                dynamoDbAsyncClient,
                cloudWatchAsyncClient,
                UUID.randomUUID().toString(),
                new StreamsRecordProcessorFactory());
```

7. 次の例に示すように、ConfigsBuilder を使用して Scheduler を作成します。

```
import java.util.UUID;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
```

```
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;

import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;

...


DynamoDbAsyncClient dynamoClient =
 DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();


DynamoDBStreamsPollingConfig pollingConfig = new
 DynamoDBStreamsPollingConfig(adapterClient);
pollingConfig.idleTimeBetweenReadsInMillis(idleTimeBetweenReadsInMillis);

// Use ConfigsBuilder to configure settings
RetrievalConfig retrievalConfig = configsBuilder.retrievalConfig();
retrievalConfig.retrievalSpecificConfig(pollingConfig);

CoordinatorConfig coordinatorConfig = configsBuilder.coordinatorConfig();
coordinatorConfig.clientVersionConfig(CoordinatorConfig.ClientVersionConfig.CLIENT_VERSION_

Scheduler scheduler = StreamsSchedulerFactory.createScheduler(
                configsBuilder.checkpointConfig(),
                coordinatorConfig,
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                retrievalConfig,
                adapterClient
        );
```

Note

KCL 3.5.x 以降の移行では、次の 3 つのフェーズを使用します。

- フェーズ 1 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1): Pure KCL 1.x 互換モード。移行固有のメタデータはリーステーブルに書き込まれません。以前のコードを再デプロイすることで、KCL v1 への安全なロールバックを行います。このフェーズを使用して安定性を検証します。
- フェーズ 2 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X): 移行を開始します。リーステーブルに WORKER_METRIC_STATS と Migration3.0 エントリを書き込みます。すべてのワーカーの準備ができたら、KCL は完全な 3.x の負荷分散に自動的に移行します。フェーズ 1 へのロールバックがサポートされています (KCL 移行ツールを使用)。KCL v1 へのロールバックはできなくなります。
- フェーズ 3 (CLIENT_VERSION_CONFIG_3X): KCL 3.x の全機能。お客様が明示的に設定するか、設定が削除されるとデフォルトとして使用されます。終了状態、ロールバックなし。

この設定は、KCL v2 と v3 ではなく、KCL v3 と KCL v1 の DynamoDB Streams Kinesis Adapter 間の互換性を維持します。

Important

CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1 (フェーズ 1) から移行を開始する必要があります。フェーズ 1 は KCL v1 と下位互換性があり、移行固有のエントリをリーステーブルに書き込まないため、以前のコードを再デプロイするだけで、以前の KCL バージョンへの安全なロールバックが可能になります。フェーズ 1 で徹底的なベイクテストを行った後、フェーズ 2 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) に進み、完全な移行を開始できます。フェーズ 1 をスキップしてフェーズ 2 を直接開始すると、リース以外のエントリがすぐにリーステーブルに書き込まれるため、DynamoDB を手動でクリーンアップしない限り、KCL v1 へのロールバックが永続的に不可能になります。

ステップ 4: KCL 3.5.x 以降の設定の概要と推奨事項

KCL 3.5.x 以降に関連する KCL 1.x 後に導入された設定の詳細については、「KCL 設定」と「KCL 移行クライアント設定」を参照してください。

> **Important**
>
> KCL 3.5.x 以降のバージョンでは、checkpointConfig、coordinatorConfig、leaseManagementConfig、metricsConfig、および retrievalConfig オブジェクトを直接作成する代わりに、スケジューラの初期化の問題を回避するために、ConfigsBuilder を使用して設定を行うことをお勧めします。ConfigsBuilder を使用すると、KCL アプリケーションをより柔軟で保守性の高い方法で構成できます。

KCL 3.5.x 以降でデフォルト値を更新する設定

billingMode

KCL バージョン 1.x では、billingMode のデフォルト値は PROVISIONED に設定されます。ただし、KCL バージョン 3.5.x 以降では、デフォルトの billingMode は PAY_PER_REQUEST (オンデマンドモード) です。使用量に基づいて容量を自動的に調整するには、リーステーブルのオンデマンドキャパシティモードを使用することをお勧めします。リーステーブルにプロビジョンドキャパシティを使用するガイダンスについては、「プロビジョンドキャパシティモードのリーステーブルのベストプラクティス」を参照してください。

idleTimeBetweenReadsInMillis

KCL バージョン 1.x では、idleTimeBetweenReadsInMillis のデフォルト値は 1,000 (または 1 秒) に設定されています。KCL バージョン 3.5.x 以降は idleTimeBetweenReadsInMillis のデフォルト値を 1,500 (または 1.5 秒) に設定しますが、Amazon DynamoDB Streams Kinesis Adapter はデフォルト値を 1,000 (または 1 秒) に上書きします。

KCL 3.5.x 以降の新しい設定

leaseAssignmentIntervalMillis

この設定では、新しく検出されたシャードが処理を開始するまでの時間間隔を定義し、1.5 × leaseAssignmentIntervalMillis として計算されます。この設定が明示的に設定されていない場合、時間間隔はデフォルトで 1.5 × failoverTimeMillis に設定されます。新しいシャードの処理には、リーステーブルをスキャンし、リーステーブルのグローバルセカンダリインデックス (GSI) をクエリする必要があります。leaseAssignmentIntervalMillis を小さくすると、これらのスキャンおよびクエリオペレーションの頻度が増加し、DynamoDB のコスト

が高くなります。新しいシャードの処理の遅延を最小限に抑えるために、この値を 2,000 (または 2 秒) に設定することをお勧めします。

`shardConsumerDispatchPollIntervalMillis`

この設定では、シャードコンシューマーによる連続するポーリング間の間隔を定義して、状態遷移をトリガーします。KCL バージョン 1.x では、この動作は `idleTimeInMillis` パラメータによって制御され、設定可能な設定として公開されませんでした。KCL バージョン 3.5.x 以降では、KCL バージョン 1.x のセットアップで `idleTimeInMillis` で使用される値と一致するようにこの設定を行うことをお勧めします。

ステップ 5: KCL 2.x から KCL 3.5.x 以降に移行する

最新の Kinesis Client Library (KCL) バージョンとのスムーズな移行と互換性を確保するには、KCL 2.x から KCL 3.5.x 以降へのアップグレードに関する移行ガイドの手順のステップ 5～8 に従います。

一般的な KCL 3.5.x 以降のトラブルシューティングの問題については、「KCL コンシューマーアプリケーションのトラブルシューティング」を参照してください。

以前のバージョンにロールバックする

このトピックでは、KCL 3.5.x 以降のコンシューマーアプリケーションを KCL 1.x にロールバックする方法について説明します。ロールバックプロセスは、アプリケーションが現在どの移行フェーズにあるかによって異なります。

フェーズ 1 から KCL 1.x にロールバックする

アプリケーションがフェーズ 1 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) にある場合は、以前のコードを再デプロイして KCL 1.x にロールバックできます。フェーズ 1 は KCL 1.x と下位互換性があり、リーステーブルに移行固有のエントリを作成しません。移行ツールは必要ありません。フェーズ 1 からロールバックするには、KCL 1.x バージョンのコードをすべてのワーカーに再デプロイします。

> **Important**
>
> フェーズ 2 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X`) はロールバックに関する重大な変更です。アプリケーションがフェーズ 2 に入ると、KCL 1.x と下位互換性のないリース以外のエントリ (`WORKER_METRIC_STATS` および `Migration3.0`) がリーステーブルに書き込まれます。これにより、KCL 1.x への直接的なロールバックが恒久的に防止されま

す。フェーズ 2 に進む前に安定性を検証するために、フェーズ 1 でアプリケーションを長期間ベイクすることを強くお勧めします。

フェーズ 2 からフェーズ 1 にロールバックする

アプリケーションがフェーズ 2 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) にある場合は、GitHub ウェブサイトの KCL 移行ツールを使用してフェーズ 1 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1) にロールバックする必要があります。これは 2 ステップのプロセスです。

1. GitHub ウェブサイトの KCL 移行ツールを実行します。
2. フェーズ 1 の設定でコードを再デプロイします (オプション)。

> **Important**
>
> 2 段階ロールバックすること (フェーズ 2 からフェーズ 1 へ、さらに KCL 1.x へロールバックすること) はできません。KCL 移行ツールは、フェーズ 2 からフェーズ 1 へのロールバックのみを処理します。このツールは、リーステーブルからリース以外のエントリを削除しません。これらのエントリは KCL 1.x と下位互換性がないため、フェーズ 2 から KCL 1.x に直接 2 段階ロールバックすることはできません。

ステップ 1: KCL 移行ツールを実行する

フェーズ 2 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) からフェーズ 1 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1) にロールバックする必要がある場合は、KCL 移行ツールを実行します。このツールは、次のタスクを実行します。

- これにより、DynamoDB のリーステーブルのグローバルセカンダリインデックス (LeaseOwnerToLeaseKeyIndex) が削除されます。このインデックスは KCL 3.5.x 以降によって作成されますが、フェーズ 1 にロールバックするときには必要ありません。
- これにより、すべてのワーカーが KCL 1.x と互換性のあるモードで実行され、以前の KCL バージョンで使用されていた負荷分散アルゴリズムの使用が開始されます。KCL 3.5.x 以降の新しい負荷分散アルゴリズムに問題がある場合、この問題はすぐに軽減されます。

> **Important**
>
> リーステーブル内のコーディネーター状態エントリ (`Migration3.0`) は、移行、ロールバック、ロールフォワードプロセス中に削除してはなりません。

> **Note**
>
> コンシューマーアプリケーションのすべてのワーカーが、一度に同じ負荷分散アルゴリズムを使用する必要があります。KCL 移行ツールを使用すると、KCL 3.5.x 以降のコンシューマーアプリケーション内のすべてのワーカーが KCL 1.x 互換モードに切り替えられ、フェーズ 1 へのローリングデプロイによるロールバック中も、すべてのワーカーが同じ負荷分散アルゴリズムで動作するようにします。

KCL 移行ツールは、KCL GitHub リポジトリのスクリプトディレクトリでダウンロードできます。任意のワーカーまたはリーステーブルへの書き込みと更新に必要なアクセス許可を持つホストからスクリプトを実行します。KCL コンシューマーアプリケーションに適切な IAM アクセス許可が設定されていることを確認します。このスクリプトは、KCL アプリケーションごとに 1 回だけ実行する必要があります。次のコマンドを使用して、KCL 移行ツールを実行します。

```
python3 ./KclMigrationTool.py --region region --mode rollback [--
application_name applicationName] [--lease_table_name leaseTableName]
```

パラメータ

`--region`

*#####*をお客様の AWS リージョンに置き換えてください。

`--application_name`

このパラメータは、リーステーブルにデフォルト名を使用している場合に必要です。リーステーブルにカスタム名を指定している場合は、このパラメータを省略できます。*applicationName* を実際の KCL アプリケーションの名前に置き換えます。カスタム名が指定されていない場合、ツールはこの名前を使用してデフォルトのテーブル名を取得します。

`--lease_table_name`

このパラメータは、KCL 設定でリーステーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している場合は、このパラメータを省略できます。*`leaseTableName`* をリーステーブルに指定したカスタムテーブル名に置き換えます。

ステップ 2: フェーズ 1 の設定でコードを再デプロイする (オプション)

フェーズ 2 からフェーズ 1 へのロールバックのために KCL 移行ツールを実行すると、次のいずれかのメッセージが表示されます。

メッセージ 1

「ロールバックが完了しました。アプリケーションはフェーズ 2 (2x 互換) の機能を実行していました。フェーズ 1 設定で KCL 3.5.x アプリケーションをデプロイして、フェーズ 1 にロールバックしてください。」

必要なアクション: ワーカーが KCL 1.x 互換モードで実行されていました (フェーズ 2 から完全な 3.x の負荷分散に自動的に移行されていません)。フェーズ 1 の設定 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) を使用して KCL 3.5.x 以降のアプリケーションをワーカーに再デプロイします。

メッセージ 2

「ロールバックが完了しました。KCL アプリケーションがフェーズ 2 (3x) の機能を実行していましたが、フェーズ 2 (2x 互換) モードにロールバックされました。短期間で緩和が確認できない場合は、フェーズ 1 の設定で KCL 3.5.x アプリケーションをデプロイして、フェーズ 1 にロールバックしてください。」

必要なアクション: ワーカーは完全な KCL 3.x の負荷分散に自動的に移行していましたが、KCL 移行ツールによって KCL 1.x 互換モードに切り替えられました。問題が解決した場合は、再デプロイする必要はありません。問題が解決しない場合は、フェーズ 1 の設定 (`CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X_PHASE1`) を使用して KCL 3.5.x 以降のアプリケーションをワーカーに再デプロイします。

ロールバック後にフェーズ 2 の KCL 3.5.x 以降にロールフォワードする

このトピックでは、ロールバック後にコンシューマーアプリケーションを KCL 3.5.x 以降にロールフォワードする方法について説明します。ロールフォワードが必要な場合は、2 ステップのプロセスを完了する必要があります。

1. GitHub ウェブサイトの KCL 移行ツールを実行します。
2. フェーズ 2 のコードをデプロイします。

ステップ 1: KCL 移行ツールを実行する

次のコマンドを使用して KCL 移行ツールを実行し、KCL 3.5.x 以降にロールフォワードします。

```
python3 ./KclMigrationTool.py --region region --mode rollforward [--
application_name applicationName] [--lease_table_name leaseTableName]
```

パラメータ

`--region`

*#####*をお客様の AWS リージョンに置き換えてください。

`--application_name`

このパラメータは、リーステーブルにデフォルト名を使用している場合に必要です。リーステーブルにカスタム名を指定している場合は、このパラメータを省略できます。*applicationName* を実際の KCL アプリケーションの名前に置き換えます。カスタム名が指定されていない場合、ツールはこの名前を使用してデフォルトのテーブル名を取得します。

`--lease_table_name`

このパラメータは、KCL 設定でリーステーブルのカスタム名を設定している場合に必要です。デフォルトのテーブル名を使用している場合は、このパラメータを省略できます。*leaseTableName* をリーステーブルに指定したカスタムテーブル名に置き換えます。

移行ツールをロールフォワードモードで実行すると、KCL は KCL 3.5.x 以降に必要な次の DynamoDB リソースを作成します。

- リーステーブルのグローバルセカンダリインデックス

> **Note**
>
> KCL 3.5.x 以降では、移行中に個別のワーカーメトリクスやコーディネーター状態テーブルは作成されません。すべてのメタデータはリーステーブルに保存されます。

ステップ 2: フェーズ 2 の設定でコードをデプロイする

ロールフォワードの KCL 移行ツールを実行したら、KCL 3.5.x 以降のフェーズ 2 の設定 (CLIENT_VERSION_CONFIG_COMPATIBLE_WITH_2X) を使用してコードをワーカーにデプロイします。移行を完了するには、「ステップ 8: 移行を完了する」を参照してください。

KCL 3.5.x 以降の単一テーブル形式への移行

KCL 3.5 以降では、単一テーブル形式を使用して、すべての DynamoDB メタデータを単一のリーステーブルに統合できます。デフォルトでは、KCL 3.x はアプリケーションごとにリーステーブル、ワーカーメトリクステーブル、コーディネーター状態テーブルの 3 つの DynamoDB テーブルを作成します。単一テーブル形式では、これら 3 つのテーブルが 1 つに減るため、DynamoDB のアカウントレベルのテーブル制限を回避できます。

単一テーブル形式の仕組み

単一テーブル形式では、KCL はワーカーメトリクスとコーディネーター状態エントリをリースエントリとともにリーステーブルに保存します。各項目には、異なるレコードタイプを区別する entityType 属性が含まれています。

ワーカーメトリクスとコーディネーター状態項目は、リーステーブルと同じプライマリキー構造を使用しますが、個別の entityType 値が含まれます。この属性により、KCL はテーブルスキャン中に各項目の目的を識別できます。

KCL の各コンポーネントは、entityType 属性に基づいてビジネスロジックに必要なエントリをフィルタリングして抽出します。例えば、リース割り当てマネージャー (LAM) はリースとワーカーのメトリクスエントリをフィルタリングしてリース割り当てを実行します。

単一テーブル形式を設定する

単一テーブル形式を有効にする方法は、現在の KCL バージョンによって異なります。

- KCL 2.x を使用している場合: 更新された移行ガイドに従って KCL 3.5 にアップグレードします。2.x から 3.5 への新しい移行には、デフォルトで単一テーブル形式が使用されます。
- KCL 3.0 ~ 3.4 を使用している場合: 単一テーブル形式に移行するには、2 フェーズのデプロイを実行する必要があります。以下の設定と移行の手順を参照してください。

既存の KCL 3.x のお客様の場合は、CoordinatorConfig で migrateAllEntitiesToLeaseTable 設定オプションを設定します。このオプションは、KCL がすべてのメタデータエンティティタイプをリーステーブルに保存するかどうかを制御します。

migrateAllEntitiesToLeaseTable の設定値

値	デフォルト	効果
false	はい	KCL は、ワーカーメトリクスとコーディネーター状態に対して個別のテーブルを使用します。アプリケーションコードは単一テーブル形式をサポートしていますが、アクティブ化していません。
true	いいえ	KCL は、ワーカーメトリクスとコーディネーター状態データのリーステーブルへの書き込みを開始します。TableMigrationStatus が DEPLOYED に達し、リグレッションがないことをベイク期間で確認した後、フェーズ 2 のデプロイでこの値を設定します。

KCL 3.x から単一テーブル形式に移行するには、2 フェーズのデプロイが必要です。

1. フェーズ 1: migrateAllEntitiesToLeaseTable を false (デフォルト) に設定して、更新された KCL 3.5 コードをデプロイします。これにより、単一テーブル形式をサポートする新しいコードがインストールされますが、移行はアクティブ化されません。
2. フェーズ 2: すべてのワーカーが新しいコードを実行し、TableMigrationStatus が DEPLOYED に達し、リグレッションがないことを確認したら、migrateAllEntitiesToLeaseTable を true に設定して再度デプロイし、移行を開始します。

移行ステータス

TableMigrationStateMachine は、複数テーブル形式から単一テーブル形式への移行を管理します。KCL は、コーディネーター状態テーブルの TableMigration3.5 という個別のコーディネー

ター状態エントリで、現在の移行状態を追跡します。状態、遷移、説明の完全なリストについては、GitHub ウェブサイトの KCL 単一テーブルの移行状態を参照してください。

単一テーブル形式の移行状態

状態	説明	移行条件
INIT	初期状態。すべてのワーカーは、ワーカーメトリクス統計の最小限のサポートコードを出力しています。ワーカーは引き続きワーカーメトリクスをレガシーテーブルに出力し、ワーカーメトリクスとコーディネーター状態をレガシーテーブルとリーステーブルの両方から読み取ります。機能的には、INIT と DEPLOYED に違いはありません。	すべてのワーカーは、ベイク時間の間、最小限のサポートコードを着実に出力します。アプリケーションはフェーズ 2 のデプロイに移行する準備ができています。
DEPLOYED	すべてのワーカーに新しいコードがデプロイされました (フェーズ 1 が完了)。アプリケーションは単一テーブル形式をサポートしていますが、アクティブ化されていません。	migrateAllEntitiesToLeaseTable を true に設定して、フェーズ 2 のデプロイを開始します。
PENDING	すべてのワーカーが新しいコードを実行します。KCL は、ワーカーメトリクスとコーディネーター状態テーブルからリーステーブルにデータを移行します。	移行が完了すると、24 時間のデフォルトのベイク時間が経過します。
COMPLETE	移行が完了しました。KCL は、すべての読み取りと書き込みにリーステーブルだけを使用します。以前のワーカーメトリクスとコーディネーター状態テーブルは、使用されなくなりました。	終了状態。これ以上の移行は行われません。

各状態、移行条件、および完全なステートマシンの動作の詳細については、GitHub ウェブサイトの KCL 単一テーブル移行ステートマシンについてのページを参照してください。

> **Note**
>
> PENDING 状態と COMPLETE 状態の間のデフォルトのベイク時間は 24 時間ですが、最大 1 週間まで設定できます。この期間中、KCL はすべての読み取りと書き込みにリーステーブルを使用しますが、古いテーブルは削除しません。移行が成功したことを確認したら、古いワーカーメトリクスとコーディネーター状態テーブルを手動で削除する必要があります。KCL はこれらのテーブルを自動的に削除しません。

移行メトリクス

KCL を使用すると、CloudWatch メトリクスを使用して単一テーブルの移行の進行状況と正常性をモニタリングできます。これらのメトリクスを使用して、ワーカーが新しいコードを採用していることを確認し、移行の状態を追跡します。移行中に DynamoDB の読み取り、書き込み、または削除の障害を検出することもできます。次の表は、メトリクスを出力する KCL オペレーション (メトリクスディメンション) と、各メトリクスが出力されるタイミングによってメトリクスをグループ化しています。

選出されたリーダーワーカーは、移行が進行中かどうかにかかわらず、次のメトリクスを継続的に出力します。

リーダーが常に発行するメトリクス

Operation	メトリクス	単位	説明
`TableMigration`	`StatusOrdinal`	なし	現在の DynamoDB 移行ステータスの序数: `0` = UNKNOWN、`1` = INIT、`2` = DEPLOYED、`3` = PENDING、`4` = COMPLETE。
`WorkerMetrics`	`FleetMinSupportCode`	なし	フリート内のすべてのリース所有ワーカーの最小限のサポートコード。

選択されたリーダーワーカーは、移行の進行中にのみ、次のメトリクスを出力します。

移行中にリーダーが発行するメトリクス

Operation	メトリクス	単位	説明
TableMigration	Phase1Worker	カウント	単一の DynamoDB テーブルでの運用をサポートしているが、単一テーブルの使用にまだ移行していないワーカーの数。
TableMigration	Phase2Worker	カウント	単一テーブルへの書き込みに移行したワーカーの数。これらのワーカーは、テーブルの移行が完了するまで、複数のテーブルから読み取りを行う可能性があります。
TableMigration	PrePhase1Worker	カウント	単一の DynamoDB テーブルでの運用をサポートしていない 3.5 より前のバージョンのワーカーの数。
TableMigration	WriteFault	カウント	DynamoDB への書き込み失敗時は 1、成功時は 0。
TableMigration	DeleteFault	カウント	DynamoDB での削除の失敗時は 1、成功時は 0。
TableMigration	CompletionFault	カウント	COMPLETE ステータスのトランザクション書き込みが失敗した場合は 1、成功した場合は 0。
TableMigrationAsyncMove	Success	カウント	CoordinatorState エントリのトランザクション移動が成功した場合は 1、失敗した場合は 0。
TableMigrationAsyncMove	Time	ミリ秒	非同期移動オペレーションの所要時間。
TableMigrationAsyncMove	BatchCount	カウント	正常に移動されたバッチの数。

すべてのワーカーは、移行の進行中に次のメトリクスを出力します。

移行中にすべてのワーカーによって出力されるメトリクス

Operation	メトリクス	単位	説明
TableMigration	ReadFault	カウント	DynamoDB から TableMigrationState の読み取りに失敗した場合は 1、成功した場合は 0。
TableMigration	Time	ミリ秒	ステートマシンの実行期間。
TableMigrationInitialize	Success	カウント	正常に初期化された場合は 1、失敗した場合は 0。
TableMigrationInitialize	Time	ミリ秒	初期化オペレーションの所要時間。

これらのメトリクスを使用して、移行をいつ進めるかを決定します。StatusOrdinal が一貫して 2 (DEPLOYED) で、PrePhase1Worker が 0 の場合、すべてのワーカーが単一テーブル形式をサポートしていることになり、テーブル移行デプロイのフェーズ 2 に移行できます。StatusOrdinal が 4 (COMPLETE) に達したら、レガシーテーブルを安全に削除できます。

ロールバックに関する考慮事項

ロールバックのサポートは、現在の TableMigrationStatus によって異なります。

- フェーズ 1 中 (TableMigrationStatus が DEPLOYED または未設定): 以前のバージョンに安全にロールバックできます。新しいコードは下位互換性モードで実行され、リーステーブルのリース以外のエントリにデータが書き込まれていません。
- フェーズ 2 中 (TableMigrationStatus が DEPLOYED または PENDING): フェーズ 1 にロールバックできます。ワーカーは、ワーカーメトリクスとコーディネーター状態のレガシーテーブル (複数テーブル形式) の使用に戻ります。移行は元に戻されます。
- COMPLETE 状態後: ロールバックはサポートされていません。KCL は、すべてのエンティティに対してリーステーブルのみを使用します。コードがフェーズ 1 にロールバックされても、ワーカーはすべてのエンティティに対してリーステーブルを使用し続け、設定は無視されます。

> **Warning**
>
> 移行が COMPLETE 状態になると、アプリケーションは単一テーブルモードでのみ動作します。`migrateAllEntitiesToLeaseTable` 設定は無視され、KCL は個別のテーブルの使用には戻りません。COMPLETE に移行する前にベイク時間が十分であることを確認してください。COMPLETE に達した後は、コードをロールバックしても複数テーブル設定には戻らないためです。

ベストプラクティス

単一テーブル形式を採用する場合は、次のベストプラクティスに従ってください。

- フェーズ 1 のデプロイ後、コーディネーター状態エントリ `TableMigration3.5` が `DEPLOYED` ステータスに達していることを確認します。フェーズ 2 に進む前に、リグレッションがないことを確認してください。
- `TableMigration3.5` コーディネーター状態エントリの `TableMigrationStatus` をモニタリングして、DEPLOYED、PENDING、COMPLETE の各ステータスの進行状況を追跡します。移行が COMPLETE に達するまで、ステータスは (リーステーブルではなく) コーディネーター状態テーブルに個別のエントリとして保存されます。
- COMPLETE に移行する前にベイク時間が十分であることを確認してください。COMPLETE に達した後は、コードをロールバックしても複数テーブル設定には戻らないためです。アプリケーションは単一テーブルモードでのみ機能します。
- 移行が COMPLETE に達したら、古いワーカーメトリクスとコーディネーター状態テーブルを手動で削除します。KCL はこれらのテーブルを自動的に削除せず、使用を停止するだけです。
- `CoordinatorConfig.coordinatorStateTableConfig` または `LeaseManagementConfig.workerUtilizationAwareAssignmentConfig.workerMetricsTable` を設定している場合は、移行の完了後にこれらの設定を削除できます。これらの設定は KCL 3.5 以降では廃止されています。

チュートリアル: DynamoDB Streams Kinesis Adapter

このセクションは、Amazon Kinesis Client Library と Amazon DynamoDB Streams Kinesis Adapter を使用する Java アプリケーションのチュートリアルです。アプリケーションには、データレプリケーションの例が表示されます。データレプリケーションでは、1 つのテーブルからの書き込みアクティビティが 2 番目のテーブルに適用され、両方のテーブルの内容が同期されます。ソースコードについては、「完成したプログラム: DynamoDB Streams Kinesis Adapter」を参照してください。

このプログラムでは、次のような処理を実行します。

1. KCL-Demo-src と KCL-Demo-dst という 2 つの DynamoDB テーブルを作成します。これらの各テーブルでは、ストリームが有効になっています。
2. 項目を追加、更新、削除することで、ソーステーブルで更新アクティビティを生成します。これにより、データがテーブルのストリームに書き込まれます。
3. ストリーミングからレコードを読み込んで、DynamoDB リクエストとして再構築し、ターゲットテーブルにリクエストを適用します。
4. ソーステーブルとターゲットテーブルをスキャンし、内容が同じであることを確認します。
5. テーブルを削除してクリーンアップします。

これらのステップについては次のセクションで説明します。完成したアプリケーションは、チュートリアルの最後に示します。

トピック

- ステップ 1: DynamoDB テーブルを作成する
- ステップ 2: ソーステーブルに更新アクティビティを生成する
- ステップ 3: ストリームを処理する
- ステップ 4: 両方のテーブルの内容が同じであることを確認する
- ステップ 5: クリーンアップ
- 完成したプログラム: DynamoDB Streams Kinesis Adapter

ステップ 1: DynamoDB テーブルを作成する

最初のステップでは、2 つの DynamoDB テーブル (送信元テーブルと送信先テーブル) を作成します。ソーステーブルのストリームにある StreamViewType は NEW_IMAGE です。これは、このテーブルで項目が変更されると必ず、イメージの "後の" 項目がストリームに書き込まれることを意味します。このようにして、ストリームはテーブル内のすべての書き込みアクティビティを記録します。

次の例は、両方のテーブルを作成するためのコードを示しています。

```
java.util.List<AttributeDefinition> attributeDefinitions = new
 ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
 AttributeDefinition().withAttributeName("Id").withAttributeType("N"));
```

```
java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new
 KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition

  // key

ProvisionedThroughput provisionedThroughput = new
 ProvisionedThroughput().withReadCapacityUnits(2L)
    .withWriteCapacityUnits(2L);

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
CreateTableRequest createTableRequest = new
 CreateTableRequest().withTableName(tableName)
    .withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

 .withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification);
```

ステップ 2: ソーステーブルに更新アクティビティを生成する

次のステップでは、ソーステーブルにいくつかの書き込みアクティビティを生成します。このアクティビティの実行中、ソーステーブルのストリームもほぼリアルタイムで更新されます。

アプリケーションは、データを書き込むための PutItem、UpdateItem、および DeleteItem API オペレーションを呼び出すメソッドを持つヘルパークラスを定義します。次の例は、これらのメソッドの使用方法を示しています。

```
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
```

ステップ 3: ストリームを処理する

ここでは、プログラムがストリームの処理を開始します。DynamoDB Streams Kinesis Adapter は、低レベルの DynamoDB Streams コールを行わなくてもコードが KCL を十分に使用できるよう

に、KCL と DynamoDB Streams エンドポイントの間の透過的なレイヤーとして機能します。このプログラムでは次のタスクを実行しています。

- KCL インターフェイス定義に従ったメソッド（`StreamsRecordProcessor`、`initialize`、`processRecords`）を使用して、レコードプロセッサクラス `shutdown` を定義します。`processRecords` メソッドには、ソーステーブルのストリームからの読み込みとターゲットテーブルへの書き込みに必要なロジックが含まれています。
- レコードプロセッサクラスのクラスファクトリを定義します（`StreamsRecordProcessorFactory`）。これは、KCL を使用する Java プログラムに必要です。
- クラスファクトリに関連付けられた新しい KCL `Worker` をインスタンス化します。
- レコード処理が完了すると、`Worker` をシャットダウンします。

必要に応じて、Streams KCL Adapter 設定でキャッチアップモードを有効にして、ストリーム処理の遅延が 1 分 (デフォルト) を超えたときに GetRecords API 呼び出しレートを 3 倍 (デフォルト) に自動的にスケーリングし、ストリームコンシューマーがテーブル内の高スループットのスパイクを処理できるようにします。

KCL インターフェイス定義の詳細については、「Amazon Kinesis Data Streams デベロッパーガイド」の「Kinesis Client Library を使用したコンシューマーの開発」を参照してください。

次の例は、`StreamsRecordProcessor` におけるメインループを示しています。`case` ステートメントは、ストリームレコードに出現する `OperationType` に基づいて、実行するアクションを決定します。

```
for (Record record : records) {
    String data = new String(record.getData().array(), Charset.forName("UTF-8"));
    System.out.println(data);
    if (record instanceof RecordAdapter) {
                software.amazon.dynamodb.model.Record streamRecord = ((RecordAdapter)
 record)
                    .getInternalObject();

                switch (streamRecord.getEventName()) {
                    case "INSERT":
                    case "MODIFY":
                        StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,
                            streamRecord.getDynamodb().getNewImage());
```

```
                        break;
                    case "REMOVE":
                        StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,
                            streamRecord.getDynamodb().getKeys().get("Id").getN());
                }
    }
    checkpointCounter += 1;
    if (checkpointCounter % 10 == 0) {
        try {
            checkpointer.checkpoint();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

ステップ 4: 両方のテーブルの内容が同じであることを確認する

この時点で、ソーステーブルとターゲットテーブルの内容が同期されています。アプリケーションは、両方のテーブルに対して Scan リクエストを発行し、内容が実際に同じであることを確認します。

DemoHelper クラスには、低レベルの ScanTable API を呼び出す Scan メソッドが含まれています。次の例は、その使用方法を示しています。

```
if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
    .equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient, destTable).getItems()))
 {
    System.out.println("Scan result is equal.");
}
else {
    System.out.println("Tables are different!");
}
```

ステップ 5: クリーンアップ

デモは完了したため、アプリケーションによりソーステーブルとターゲットテーブルが削除されます。次のコード例を参照してください。テーブルが削除されても、そのストリームは最大 24 時間使用可能です。その後、自動的に削除されます。

```
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
```

完成したプログラム: DynamoDB Streams Kinesis Adapter

次に、チュートリアル: DynamoDB Streams Kinesis Adapter で説明したタスクを実行する、完成した Java プログラムを次に示します。実行すると、次のような出力が表示されます。

```
Creating table KCL-Demo-src
Creating table KCL-Demo-dest
Table is active.
Creating worker for stream: arn:aws:dynamodb:us-west-2:111122223333:table/KCL-Demo-src/
stream/2015-05-19T22:48:56.601
Starting worker...
Scan result is equal.
Done.
```

> **Important**
>
> このプログラムを実行するには、クライアントアプリケーションがポリシーを使用して DynamoDB および Amazon CloudWatch にアクセスできることを確認します。詳細については、「DynamoDB のアイデンティティベースのポリシー」を参照してください。

ソースコードは、4 つの `.java` ファイルから構成されています。このプログラムを構築するには、Amazon Kinesis Client Library (KCL) 3.x と AWS SDK for Java v2 を含む次の依存関係を推移的な依存関係として追加します。

Maven

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>dynamodb-streams-kinesis-adapter</artifactId>
    <version>2.1.0</version>
</dependency>
```

Gradle

```
implementation 'com.amazonaws:dynamodb-streams-kinesis-adapter:2.1.0'
```

ソースファイルは次のとおりです。

- StreamsAdapterDemo.java
- StreamsRecordProcessor.java
- StreamsRecordProcessorFactory.java
- StreamsAdapterDemoHelper.java

StreamsAdapterDemo.java

```
package com.amazonaws.codesamples;

import
 com.amazonaws.services.dynamodbv2.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import com.amazonaws.services.dynamodbv2.streamsadapter.StreamsSchedulerFactory;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.polling.DynamoDBStreamsPollingConfig;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.InitialPositionInStream;
import software.amazon.kinesis.common.InitialPositionInStreamExtended;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.processor.StreamTracker;
import software.amazon.kinesis.retrieval.RetrievalConfig;

public class StreamsAdapterDemo {

    private static DynamoDbAsyncClient dynamoDbAsyncClient;
    private static CloudWatchAsyncClient cloudWatchAsyncClient;
    private static AmazonDynamoDBStreamsAdapterClient
 amazonDynamoDbStreamsAdapterClient;

    private static String tablePrefix = "KCL-Demo";
    private static String streamArn;
```

```
    private static Region region = Region.US_EAST_1;
    private static AwsCredentialsProvider credentialsProvider =
DefaultCredentialsProvider.create();

    public static void main( String[] args ) throws Exception {
        System.out.println("Starting demo...");
        dynamoDbAsyncClient = DynamoDbAsyncClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build();
        cloudWatchAsyncClient = CloudWatchAsyncClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build();
        amazonDynamoDbStreamsAdapterClient = new
AmazonDynamoDBStreamsAdapterClient(credentialsProvider, region);

        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";

        setUpTables();

        StreamTracker streamTracker =
StreamsSchedulerFactory.createSingleStreamTracker(streamArn,

InitialPositionInStreamExtended.newInitialPosition(InitialPositionInStream.TRIM_HORIZON));

        ShardRecordProcessorFactory shardRecordProcessorFactory =
                new StreamsAdapterDemoProcessorFactory(dynamoDbAsyncClient, destTable);

        ConfigsBuilder configsBuilder = new ConfigsBuilder(
                streamTracker,
                "streams-adapter-demo",
                amazonDynamoDbStreamsAdapterClient,
                dynamoDbAsyncClient,
                cloudWatchAsyncClient,
                "streams-demo-worker",
                shardRecordProcessorFactory
        );

        DynamoDBStreamsPollingConfig pollingConfig = new
DynamoDBStreamsPollingConfig(amazonDynamoDbStreamsAdapterClient);
        RetrievalConfig retrievalConfig = configsBuilder.retrievalConfig();
        retrievalConfig.retrievalSpecificConfig(pollingConfig);
```

```
        System.out.println("Creating scheduler for stream " + streamArn);
        Scheduler scheduler = StreamsSchedulerFactory.createScheduler(
                configsBuilder.checkpointConfig(),
                configsBuilder.coordinatorConfig(),
                configsBuilder.leaseManagementConfig(),
                configsBuilder.lifecycleConfig(),
                configsBuilder.metricsConfig(),
                configsBuilder.processorConfig(),
                retrievalConfig,
                amazonDynamoDbStreamsAdapterClient
        );

        System.out.println("Starting scheduler...");
        Thread t = new Thread(scheduler);
        t.start();

        Thread.sleep(250000);

        System.out.println("Stopping scheduler...");
        scheduler.shutdown();
        t.join();

        if (StreamsAdapterDemoHelper.scanTable(dynamoDbAsyncClient, srcTable).items()
                .equals(StreamsAdapterDemoHelper.scanTable(dynamoDbAsyncClient,
destTable).items())) {
            System.out.println("Scan result is equal.");
        } else {
            System.out.println("Tables are different!");
        }

        System.out.println("Done.");
        cleanupAndExit(0);
    }

    private static void setUpTables() {
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";
        streamArn = StreamsAdapterDemoHelper.createTable(dynamoDbAsyncClient,
srcTable);
        StreamsAdapterDemoHelper.createTable(dynamoDbAsyncClient, destTable);

        awaitTableCreation(srcTable);
```

```
        performOps(srcTable);
    }

    private static void awaitTableCreation(String tableName) {
        Integer retries = 0;
        Boolean created = false;
        while (!created && retries < 100) {
            DescribeTableResponse result =
StreamsAdapterDemoHelper.describeTable(dynamoDbAsyncClient, tableName);
            created = result.table().tableStatusAsString().equals("ACTIVE");
            if (created) {
                System.out.println("Table is active.");
                return;
            } else {
                retries++;
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // do nothing
                }
            }
        }
        System.out.println("Timeout after table creation. Exiting...");
        cleanupAndExit(1);
    }

    private static void performOps(String tableName) {
        StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName, "101",
"test1");
        StreamsAdapterDemoHelper.updateItem(dynamoDbAsyncClient, tableName, "101",
"test2");
        StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName, "101");
        StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName, "102",
"demo3");
        StreamsAdapterDemoHelper.updateItem(dynamoDbAsyncClient, tableName, "102",
"demo4");
        StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName, "102");
    }

    private static void cleanupAndExit(Integer returnValue) {
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";

dynamoDbAsyncClient.deleteTable(DeleteTableRequest.builder().tableName(srcTable).build());
```

```
 dynamoDbAsyncClient.deleteTable(DeleteTableRequest.builder().tableName(destTable).build());
        System.exit(returnValue);
    }
}
```

StreamsRecordProcessor.java

```
package com.amazonaws.codesamples;

import
 com.amazonaws.services.dynamodbv2.streamsadapter.adapter.DynamoDBStreamsClientRecord;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.model.DynamoDBStreamsProcessRecordsInput;
import
 com.amazonaws.services.dynamodbv2.streamsadapter.processor.DynamoDBStreamsShardRecordProcessor
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.Record;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;

public class StreamsRecordProcessor implements DynamoDBStreamsShardRecordProcessor {

    private Integer checkpointCounter;

    private final DynamoDbAsyncClient dynamoDbAsyncClient;
    private final String tableName;

    public StreamsRecordProcessor(DynamoDbAsyncClient dynamoDbAsyncClient, String
 tableName) {
        this.dynamoDbAsyncClient = dynamoDbAsyncClient;
        this.tableName = tableName;
    }

    @Override
    public void initialize(InitializationInput initializationInput) {
```

```
        this.checkpointCounter = 0;
    }

    @Override
    public void processRecords(DynamoDBStreamsProcessRecordsInput
dynamoDBStreamsProcessRecordsInput) {
        for (DynamoDBStreamsClientRecord record:
dynamoDBStreamsProcessRecordsInput.records()) {
            String data = new String(record.data().array(), StandardCharsets.UTF_8);
            System.out.println(data);
            Record streamRecord = record.getRecord();

            switch (streamRecord.eventName()) {
                case INSERT:
                case MODIFY:
                    StreamsAdapterDemoHelper.putItem(dynamoDbAsyncClient, tableName,
                            streamRecord.dynamodb().newImage());
                case REMOVE:
                    StreamsAdapterDemoHelper.deleteItem(dynamoDbAsyncClient, tableName,
                            streamRecord.dynamodb().keys().get("Id").n());
            }
            checkpointCounter += 1;
            if (checkpointCounter % 10 == 0) {
                try {
                    dynamoDBStreamsProcessRecordsInput.checkpointer().checkpoint();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {
        System.out.println("Lease Lost");
    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            e.printStackTrace();
        }
```

```
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            e.printStackTrace();
        }
    }
}
```

StreamsRecordProcessorFactory.java

```
package com.amazonaws.codesamples;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class StreamsAdapterDemoProcessorFactory implements ShardRecordProcessorFactory
 {
    private final String tableName;
    private final DynamoDbAsyncClient dynamoDbAsyncClient;

    public StreamsAdapterDemoProcessorFactory(DynamoDbAsyncClient asyncClient, String
 tableName) {
        this.tableName = tableName;
        this.dynamoDbAsyncClient = asyncClient;
    }

    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new StreamsRecordProcessor(dynamoDbAsyncClient, tableName);
    }
}
```

StreamsAdapterDemoHelper.java

```
package com.amazonaws.codesamples;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
```

```
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.ResourceInUseException;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import software.amazon.awssdk.services.dynamodb.model.StreamSpecification;
import software.amazon.awssdk.services.dynamodb.model.StreamViewType;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class StreamsAdapterDemoHelper {

    /**
     * @return StreamArn
     */
    public static String createTable(DynamoDbAsyncClient client, String tableName) {
        List<AttributeDefinition> attributeDefinitions = new ArrayList<>();
        attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName("Id")
                .attributeType("N")
                .build());

        List<KeySchemaElement> keySchema = new ArrayList<>();
        keySchema.add(KeySchemaElement.builder()
                .attributeName("Id")
                .keyType(KeyType.HASH) // Partition key
                .build());

        StreamSpecification streamSpecification = StreamSpecification.builder()
                .streamEnabled(true)
                .streamViewType(StreamViewType.NEW_IMAGE)
```

```
                .build();

        CreateTableRequest createTableRequest = CreateTableRequest.builder()
                .tableName(tableName)
                .attributeDefinitions(attributeDefinitions)
                .keySchema(keySchema)
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .streamSpecification(streamSpecification)
                .build();

        try {
            System.out.println("Creating table " + tableName);
            CreateTableResponse result = client.createTable(createTableRequest).join();
            return result.tableDescription().latestStreamArn();
        } catch (Exception e) {
            if (e.getCause() instanceof ResourceInUseException) {
                System.out.println("Table already exists.");
                return describeTable(client, tableName).table().latestStreamArn();
            }
            throw e;
        }
    }

    public static DescribeTableResponse describeTable(DynamoDbAsyncClient client,
String tableName) {
        return client.describeTable(DescribeTableRequest.builder()
                        .tableName(tableName)
                        .build())
                .join();
    }

    public static ScanResponse scanTable(DynamoDbAsyncClient dynamoDbClient, String
tableName) {
        return dynamoDbClient.scan(ScanRequest.builder()
                        .tableName(tableName)
                        .build())
                .join();
    }

    public static void putItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
String id, String val) {
        Map<String, AttributeValue> item = new HashMap<>();
        item.put("Id", AttributeValue.builder().n(id).build());
        item.put("attribute-1", AttributeValue.builder().s(val).build());
```

```
        putItem(dynamoDbClient, tableName, item);
    }

    public static void putItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
                               Map<String, AttributeValue> items) {
        PutItemRequest putItemRequest = PutItemRequest.builder()
                .tableName(tableName)
                .item(items)
                .build();
        dynamoDbClient.putItem(putItemRequest).join();
    }

    public static void updateItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
String id, String val) {
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("Id", AttributeValue.builder().n(id).build());

        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#attr2", "attribute-2");

        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":val", AttributeValue.builder().s(val).build());

        UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #attr2 = :val")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

        dynamoDbClient.updateItem(updateItemRequest).join();
    }

    public static void deleteItem(DynamoDbAsyncClient dynamoDbClient, String tableName,
String id) {
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("Id", AttributeValue.builder().n(id).build());

        DeleteItemRequest deleteItemRequest = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .build();
```

```
        dynamoDbClient.deleteItem(deleteItemRequest).join();
    }
}
```

DynamoDB Streams 低レベル API: Java の例

> **Note**
>
> このページのコードはすべてを網羅していないため、Amazon DynamoDB Streams を使用するすべてのシナリオに対応しているわけではありません。DynamoDB からストリーミングレコードを使用する場合は、DynamoDB Streams Kinesis Adapter を使用したストリームレコードの処理 で説明されているとおり、Kinesis Client Library (KCL) を使用し、Amazon Kinesis Adapter を介して行うことを推奨します。

このセクションには、動作中の DynamoDB Streams を示す Java プログラムが含まれています。このプログラムでは、次のような処理を実行します。

1. ストリーミングが有効になった DynamoDB テーブルを作成します。
2. このテーブルのストリーム設定を記述します。
3. テーブル内のデータを変更します。
4. ストリーム内のシャードを記述します。
5. シャードからストリームレコードを読み込みます。
6. 子シャードを取得し、レコードの読み取りを続行します。
7. クリーンアップします。

プログラムを実行すると、以下のような出力が表示されます。

```
Testing Streams Demo
Creating an Amazon DynamoDB table TestTableForStreams with a simple primary key: Id
Waiting for TestTableForStreams to be created...
Current stream ARN for TestTableForStreams: arn:aws:dynamodb:us-
east-2:123456789012:table/TestTableForStreams/stream/2018-03-20T16:49:55.208
Stream enabled: true
Update view type: NEW_AND_OLD_IMAGES

Performing write activities on TestTableForStreams
Processing item 1 of 100
```

```
Processing item 2 of 100
Processing item 3 of 100
...
Processing item 100 of 100
Shard: {ShardId: shardId-1234567890-...,SequenceNumberRange: {StartingSequenceNumber:
 100002572486797508907,},}
    Shard iterator: EjYFEkX2a26eVTWe...
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, NewImage={Message=AttributeValue(S=New Item!),
 Id=AttributeValue(S=4)}, SequenceNumber=200000158404754583390 9, SizeBytes=22,
 StreamViewType=NEW_AND_OLD_IMAGES)
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, NewImage={Message=AttributeValue(S=This is an updated
 item), Id=AttributeValue(S=4)}, OldImage={Message=AttributeValue(S=New Item!),
 Id=AttributeValue(S=4)}, SequenceNumber=210000360486976789270 1, SizeBytes=55,
 StreamViewType=NEW_AND_OLD_IMAGES)
        StreamRecord(ApproximateCreationDateTime=2025-04-09T13:11:58Z,
 Keys={Id=AttributeValue(S=4)}, OldImage={Message=AttributeValue(S=This is an updated
 item), Id=AttributeValue(S=4)}, SequenceNumber=220000109977111289843 4, SizeBytes=36,
 StreamViewType=NEW_AND_OLD_IMAGES)
...
Deleting the table...
Table StreamsDemoTable deleted.
Demo complete
```

Example例

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
```

```
import software.amazon.awssdk.services.dynamodb.model.DescribeStreamRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeStreamResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetRecordsRequest;
import software.amazon.awssdk.services.dynamodb.model.GetRecordsResponse;
import software.amazon.awssdk.services.dynamodb.model.GetShardIteratorRequest;
import software.amazon.awssdk.services.dynamodb.model.GetShardIteratorResponse;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.Record;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.Shard;
import software.amazon.awssdk.services.dynamodb.model.ShardFilter;
import software.amazon.awssdk.services.dynamodb.model.ShardFilterType;
import software.amazon.awssdk.services.dynamodb.model.ShardIteratorType;
import software.amazon.awssdk.services.dynamodb.model.StreamSpecification;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.streams.DynamoDbStreamsClient;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

public class StreamsLowLevelDemo {


    public static void main(String[] args) {
        final String usage = "Testing Streams Demo";
        try {
            System.out.println(usage);

            String tableName = "StreamsDemoTable";
            String key = "Id";
            System.out.println("Creating an Amazon DynamoDB table " + tableName + "
 with a simple primary key: " + key);
            Region region = Region.US_WEST_2;
            DynamoDbClient ddb = DynamoDbClient.builder()
                    .region(region)
                    .build();

            DynamoDbStreamsClient ddbStreams = DynamoDbStreamsClient.builder()
                    .region(region)
                    .build();
```

```
            DescribeTableRequest describeTableRequest = DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build();
            TableDescription tableDescription = null;
            try{
                tableDescription = ddb.describeTable(describeTableRequest).table();
            }catch (Exception e){
                System.out.println("Table " + tableName + " does not exist.");
                tableDescription = createTable(ddb, tableName, key);
            }

            // Print the stream settings for the table
            String streamArn = tableDescription.latestStreamArn();

            StreamSpecification streamSpec = tableDescription.streamSpecification();
            System.out.println("Current stream ARN for " + tableDescription.tableName()
+ ": " +
                   streamArn);
            System.out.println("Stream enabled: " + streamSpec.streamEnabled());
            System.out.println("Update view type: " + streamSpec.streamViewType());
            System.out.println();
            // Generate write activity in the table
            System.out.println("Performing write activities on " + tableName);
            int maxItemCount = 100;
            for (Integer i = 1; i <= maxItemCount; i++) {
                System.out.println("Processing item " + i + " of " + maxItemCount);
                // Write a new item
                putItemInTable(key, i, tableName, ddb);
                // Update the item
                updateItemInTable(key, i, tableName, ddb);
                // Delete the item
                deleteDynamoDBItem(key, i, tableName, ddb);
            }

            // Process Stream
            processStream(streamArn, maxItemCount, ddb, ddbStreams, tableName);

            // Delete the table
            System.out.println("Deleting the table...");
            DeleteTableRequest deleteTableRequest = DeleteTableRequest.builder()
                    .tableName(tableName)
                    .build();
            ddb.deleteTable(deleteTableRequest);
            System.out.println("Table " + tableName + " deleted.");
```

```
            System.out.println("Demo complete");
            ddb.close();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    private static void processStream(String streamArn, int maxItemCount,
DynamoDbClient ddb, DynamoDbStreamsClient ddbStreams, String tableName) {
        // Get all the shard IDs from the stream. Note that DescribeStream returns
        // the shard IDs one page at a time.
        String lastEvaluatedShardId = null;
        do {
            DescribeStreamRequest describeStreamRequest =
DescribeStreamRequest.builder()
                    .streamArn(streamArn)
                    .exclusiveStartShardId(lastEvaluatedShardId).build();
            DescribeStreamResponse describeStreamResponse =
ddbStreams.describeStream(describeStreamRequest);

            List<Shard> shards = describeStreamResponse.streamDescription().shards();

            // Process each shard on this page

            fetchShardsAndReadRecords(streamArn, maxItemCount, ddbStreams, shards);

            // If LastEvaluatedShardId is set, then there is
            // at least one more page of shard IDs to retrieve
            lastEvaluatedShardId =
describeStreamResponse.streamDescription().lastEvaluatedShardId();

        } while (lastEvaluatedShardId != null);

    }

    private static void fetchShardsAndReadRecords(String streamArn, int maxItemCount,
DynamoDbStreamsClient ddbStreams, List<Shard> shards) {
        for (Shard shard : shards) {
            String shardId = shard.shardId();
            System.out.println("Shard: " + shard);

            // Get an iterator for the current shard
            GetShardIteratorRequest shardIteratorRequest =
GetShardIteratorRequest.builder()
```

```
                    .streamArn(streamArn).shardId(shardId)
                    .shardIteratorType(ShardIteratorType.TRIM_HORIZON).build();

            GetShardIteratorResponse getShardIteratorResult =
ddbStreams.getShardIterator(shardIteratorRequest);

            String currentShardIter = getShardIteratorResult.shardIterator();

            // Shard iterator is not null until the Shard is sealed (marked as
READ_ONLY).
            // To prevent running the loop until the Shard is sealed, we process only
the
            // items that were written into DynamoDB and then exit.
            int processedRecordCount = 0;
            while (currentShardIter != null && processedRecordCount < maxItemCount) {
                // Use the shard iterator to read the stream records
                GetRecordsRequest getRecordsRequest = GetRecordsRequest.builder()
                        .shardIterator(currentShardIter).build();
                GetRecordsResponse getRecordsResult =
ddbStreams.getRecords(getRecordsRequest);
                List<Record> records = getRecordsResult.records();
                for (Record record : records) {
                    System.out.println("        " + record.dynamodb());
                }
                processedRecordCount += records.size();
                currentShardIter = getRecordsResult.nextShardIterator();
            }
            if (currentShardIter == null){
                System.out.println("Shard has been fully processed. Shard iterator is
null.");
                System.out.println("Fetch the child shard to continue processing
instead of bulk fetching all shards");
                DescribeStreamRequest describeStreamRequestForChildShards =
DescribeStreamRequest.builder()
                        .streamArn(streamArn)
                        .shardFilter(ShardFilter.builder()
                                .type(ShardFilterType.CHILD_SHARDS)
                                .shardId(shardId).build())
                        .build();
                DescribeStreamResponse describeStreamResponseChildShards =
ddbStreams.describeStream(describeStreamRequestForChildShards);
                fetchShardsAndReadRecords(streamArn, maxItemCount, ddbStreams,
describeStreamResponseChildShards.streamDescription().shards());
            }
```

```
        }
    }

    private static void putItemInTable(String key, Integer i, String tableName,
DynamoDbClient ddb) {
        Map<String, AttributeValue> item = new HashMap<>();
        item.put(key, AttributeValue.builder()
                .s(i.toString())
                .build());
        item.put("Message", AttributeValue.builder()
                .s("New Item!")
                .build());
        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(item)
                .build();
        ddb.putItem(request);
    }

    private static void updateItemInTable(String key, Integer i, String tableName,
DynamoDbClient ddb) {

        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put(key, AttributeValue.builder()
                .s(i.toString())
                .build());


        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put("Message", AttributeValueUpdate.builder()
                .value(AttributeValue.builder().s("This is an updated item").build())
                .action(AttributeAction.PUT)
                .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(itemKey)
                .attributeUpdates(updatedValues)
                .build();
        ddb.updateItem(request);
    }

    public static void deleteDynamoDBItem(String key, Integer i, String tableName,
DynamoDbClient ddb) {
```

```
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(i.toString())
                .build());

        DeleteItemRequest deleteReq = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(keyToGet)
                .build();
        ddb.deleteItem(deleteReq);
    }

    public static TableDescription createTable(DynamoDbClient ddb, String tableName,
String key) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        StreamSpecification streamSpecification = StreamSpecification.builder()
                .streamEnabled(true)
                .streamViewType("NEW_AND_OLD_IMAGES")
                .build();
        CreateTableRequest request = CreateTableRequest.builder()
                .attributeDefinitions(AttributeDefinition.builder()
                        .attributeName(key)
                        .attributeType(ScalarAttributeType.S)
                        .build())
                .keySchema(KeySchemaElement.builder()
                        .attributeName(key)
                        .keyType(KeyType.HASH)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
                .tableName(tableName)
                .streamSpecification(streamSpecification)
                .build();

        TableDescription newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                    .tableName(tableName)
                    .build();

            System.out.println("Waiting for " + tableName + " to be created...");

            // Wait until the Amazon DynamoDB table is created.
```

```
            WaiterResponse<DescribeTableResponse> waiterResponse =
 dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            newTable = response.tableDescription();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return null;
    }



}
```

DynamoDB Streams と AWS Lambda のトリガー

Amazon DynamoDB は AWS Lambda と統合されているため、トリガー (DynamoDB Streams 内のイベントに自動的に応答するコードの一部) を作成できます。トリガーを使用すると、DynamoDB テーブル内のデータ変更に対応するアプリケーションを構築できます。

トピック

- チュートリアル #1: AWS CLI を使用した Amazon DynamoDB と AWS Lambda での、フィルターを使ったすべてのイベントの処理
- チュートリアル #2: DynamoDB と Lambda での、フィルターを使用したいくつかのイベントの処理。
- Lambda での DynamoDB ストリームの使用に関するベストプラクティス

テーブルで DynamoDB Streams を有効にした場合、書き込む AWS Lambda 関数にストリーミングの Amazon リソースネーム (ARN) を関連付けることができます。その後、その DynamoDB テーブルに対するすべてのミューテーションアクションをストリーム上の項目としてキャプチャできます。例えば、テーブル内の項目が変更されたときに、そのテーブルのストリームに新しいレコードがすぐに表示されるようにトリガーを設定できます。

> **Note**
>
> 3 つ以上の Lambda 関数を 1 つの DynamoDB ストリームにサブスクライブすると、読み込みスロットリングが発生する可能性があります。

AWS Lambda サービスは、1 秒に 4 回、ストリームをポーリングして新しいレコードを探します。新しいストリームレコードが利用可能になると、Lambda 関数が同期的に呼び出されます。同じ DynamoDB ストリームに最大 2 つの Lambda 関数をサブスクライブできます。同じ DynamoDB ストリームに 3 つ以上の Lambda 関数をサブスクライブすると、読み込みスロットリングが発生する可能性があります。

Lambda 関数は、通知の送信やワークフローの開始など、指定した他の多くのアクションを実行できます。各ストリームレコードを Amazon S3 File Gateway (Amazon S3) などの永続的ストレージにコピーするだけで、書き込みアクティビティの永続的な監査追跡をテーブルに作成する Lambda 関数を記述できます。または、`GameScores` テーブルに書き込みを行うモバイルゲームアプリがあるとします。`TopScore` テーブルの `GameScores` 属性が更新されるたびに、対応するストリームレコードがテーブルのストリームに書き込まれます。その後、このイベントによって Lambda 関数をトリガーし、ソーシャルメディアネットワークにおめでとうメッセージを投稿できます この関数は、`GameScores` の更新でないストリームレコードや、`TopScore` 属性を変更しないストリームレコードを無視するように記述できます。

関数がエラーを返した場合、処理が正常に完了するか、データの有効期限が切れるまで Lambda はバッチを再試行します。Lambda では、より小さなバッチで再試行したり、再試行回数を制限したり、古すぎるレコードを破棄したりするよう設定できます。

パフォーマンスのベストプラクティスとして、Lambda 関数は存続期間を短くする必要があります。また、不必要な処理の遅延が発生するのを防ぐため、複雑なロジックは実行しないでください。特に高速ストリームの場合は、同期的に長時間実行する Lambda よりも、非同期的な後処理ステップ関数ワークフローをトリガーすることをお勧めします。

DynamoDB ストリームでリソースベースのポリシーを設定し、Lambda 関数へのクロスアカウント読み取りアクセスを許可することで、異なる AWS アカウント間で Lambda トリガーを使用できます。クロスアカウントアクセスを許可するようにストリームを設定する方法については、DynamoDB 開発者ガイドの「クロスアカウントの AWS Lambda 関数を使用したアクセスの共有」を参照してください。

AWS Lambda の詳細については、「AWS Lambda デベロッパーガイド」を参照してください。

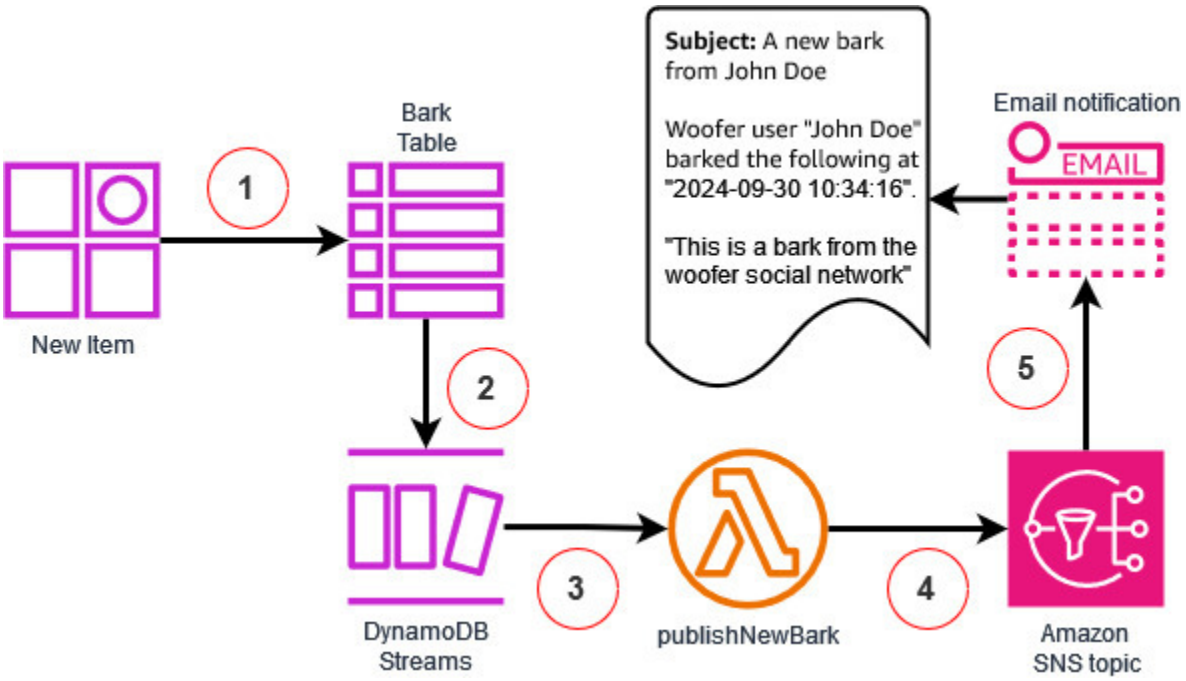
チュートリアル #1: AWS CLI を使用した Amazon DynamoDB と AWS Lambda での、フィルターを使ったすべてのイベントの処理

このチュートリアルでは、AWS Lambda トリガーを作成して、DynamoDB テーブルからのストリーミングを処理します。

トピック

- ステップ 1: ストリーミングが有効になった DynamoDB テーブルを作成する
- ステップ 2: Lambda 実行ロールを作成する
- ステップ 3: Amazon SNS トピックを作成する
- ステップ 4: Lambda 関数を作成してテストする
- ステップ 5: トリガーを作成してテストする

このチュートリアルのシナリオは、シンプルなソーシャルネットワークである Woofer です。Woofer ユーザーは、他の Woofer ユーザーに送信される bark (短いテキストメッセージ) を使用して通信します。次の図は、このアプリケーションのコンポーネントとワークフローを示しています。



1. ユーザーは DynamoDB テーブル (`BarkTable`) に項目を書き込みます。テーブルの各項目は bark を表します。
2. 新しいストリームレコードが書き込まれ、新しい項目が `BarkTable` に追加されたことを反映します。

3. 新しいストリームレコードは AWS Lambda 関数 (publishNewBark) をトリガーします。
4. ストリーミングレコードに、新しい項目が BarkTable に追加されたことが示された場合、Lambda 関数はストリーミングレコードからデータを読み込み、Amazon Simple Notification Service (Amazon SNS) のトピックにメッセージを発行します。
5. メッセージは Amazon SNS トピックの受信者によって受信されます (このチュートリアルでは、唯一の受信者は E メールアドレスです)。

開始する前に

このチュートリアルでは AWS Command Line Interface AWS CLI を使用します。まだ行っていない場合は、「AWS Command Line Interface ユーザーガイド」の手順に従って、AWS CLI をインストールおよび設定します。

ステップ 1: ストリーミングが有効になった DynamoDB テーブルを作成する

このステップでは、Woofer ユーザーからのすべての bark を保存する DynamoDB テーブル (BarkTable) を作成します。プライマリキーは、Username (パーティションキー) と Timestamp (ソートキー) で構成されます。これらの属性は両方とも文字列型になります。

BarkTable ではストリームが有効になっています。このチュートリアルの後半では、AWS Lambda 関数をストリームと関連付けてトリガーを作成します。

1. 次のコマンドを入力して、テーブルを作成します。

```
aws dynamodb create-table \
    --table-name BarkTable \
    --attribute-definitions AttributeName=Username,AttributeType=S
 AttributeName=Timestamp,AttributeType=S \
    --key-schema AttributeName=Username,KeyType=HASH
 AttributeName=Timestamp,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

2. 出力で、LatestStreamArn を探します。

```
...
"LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/
stream/timestamp
...
```

region と *accountID* をメモしておきます。これらは、このチュートリアルの他のステップで必要になります。

ステップ 2: Lambda 実行ロールを作成する

このステップでは、AWS Identity and Access Management (IAM) ロール (`WooferLambdaRole`) を作成し、それにアクセス権限を割り当てます。このロールは、ステップ 4: Lambda 関数を作成してテストする で作成する Lambda 関数で使用されます。

また、ロールのポリシーを作成します。このポリシーには、Lambda 関数がランタイム時に必要とするすべてのアクセス許可が含まれます。

1. 次の内容で、`trust-relationship.json` というファイルを作成します。

   JSON

   ```
   {
      "Version":"2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
   ```

2. `WooferLambdaRole` を作成するため、以下のコマンドを入力します。

   ```
   aws iam create-role --role-name WooferLambdaRole \
       --path "/service-role/" \
       --assume-role-policy-document file://trust-relationship.json
   ```

3. 次の内容で、`role-policy.json` というファイルを作成します。(*region* および *accountID* をお客様の AWS リージョンとアカウント ID に置き換えます。)

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:us-east-1:111122223333:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/
BarkTable/stream/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

ポリシーには 4 つのステートメントがあり、これにより WooferLambdaRole は以下を実行できます。

- Lambda 関数 (publishNewBark) を実行します。このチュートリアルの後半で、この関数を作成します。
- Amazon CloudWatch Logs のアクセス。Lambda 関数はランタイム時に診断を CloudWatch Logs に書き込みます。
- BarkTable の DynamoDB Streams からデータを読み込みます。
- Amazon SNS にメッセージを公開します。

4. 次のコマンドを入力して、WooferLambdaRole にポリシーをアタッチします。

```
aws iam put-role-policy --role-name WooferLambdaRole \
    --policy-name WooferLambdaRolePolicy \
    --policy-document file://role-policy.json
```

ステップ 3: Amazon SNS トピックを作成する

このステップでは、Amazon SNSトピック (wooferTopic) を作成し、そのトピックに E メールアドレスをサブスクライブします。Lambda 関数はこのトピックを使用して、Woofer ユーザーからの新しい bark を公開します。

1. 次のコマンドを入力して、新しい Amazon SNS トピックを作成します。

```
aws sns create-topic --name wooferTopic
```

2. 次のコマンドを入力して、wooferTopic に E メールアドレスをサブスクライブします (*region* および *accountID* は AWS リージョンとアカウント ID に置き換え、*example@example.com* は有効な E メールアドレスと置き換えます)。

```
aws sns subscribe \
    --topic-arn arn:aws:sns:region:accountID:wooferTopic \
    --protocol email \
    --notification-endpoint example@example.com
```

3. Amazon SNS は E メールアドレスに確認メッセージを送信します。そのメッセージの [サブスクリプションを確認] リンクを選択して、サブスクリプションプロセスを完了します。

ステップ 4: Lambda 関数を作成してテストする

このステップでは、AWS Lambda 関数 (publishNewBark) を作成して BarkTable からのストリームレコードを処理します。

publishNewBark 関数は、BarkTable の新しい項目に対応するストリームイベントのみを処理します。この関数は、そのようなイベントからデータを読み取ってから、Amazon SNS をコールしてデータを公開します。

1. 次の内容で、publishNewBark.js というファイルを作成します。*region* および *accountID* をお客様の AWS リージョンとアカウント ID に置き換えます。

```
'use strict';
var AWS = require("aws-sdk");
var sns = new AWS.SNS();

exports.handler = (event, context, callback) => {

    event.Records.forEach((record) => {
        console.log('Stream record: ', JSON.stringify(record, null, 2));

        if (record.eventName == 'INSERT') {
            var who = JSON.stringify(record.dynamodb.NewImage.Username.S);
            var when = JSON.stringify(record.dynamodb.NewImage.Timestamp.S);
            var what = JSON.stringify(record.dynamodb.NewImage.Message.S);
            var params = {
                Subject: 'A new bark from ' + who,
                Message: 'Woofer user ' + who + ' barked the following at ' + when
 + ':\n\n ' + what,
                TopicArn: 'arn:aws:sns:region:accountID:wooferTopic'
            };
            sns.publish(params, function(err, data) {
                if (err) {
                    console.error("Unable to send message. Error JSON:",
 JSON.stringify(err, null, 2));
                } else {
                    console.log("Results from sending message: ",
 JSON.stringify(data, null, 2));
                }
            });
        }
    });
    callback(null, `Successfully processed ${event.Records.length} records.`);
```

```
};
```

2. publishNewBark.js を含める zip ファイルを作成します。zip コマンドラインユーティリティがある場合は、次のコマンドを入力してこれを行うことができます。

```
zip publishNewBark.zip publishNewBark.js
```

3. Lambda 関数を作成する場合は、ステップ 2: Lambda 実行ロールを作成する で作成した WooferLambdaRole の Amazon リソースネーム (ARN) を指定します。この ARN を取得するには、次のコマンドを入力します。

```
aws iam get-role --role-name WooferLambdaRole
```

出力で、WooferLambdaRole の ARN を探します。

```
...
"Arn": "arn:aws:iam::region:role/service-role/WooferLambdaRole"
...
```

次のコマンドを入力して、Lambda 関数を作成します。*roleARN* を WooferLambdaRole の ARN に置き換えます。

```
aws lambda create-function \
    --region region \
    --function-name publishNewBark \
    --zip-file fileb://publishNewBark.zip \
    --role roleARN \
    --handler publishNewBark.handler \
    --timeout 5 \
    --runtime nodejs16.x
```

4. ここで、publishNewBark をテストして、これが動作することを確認します。これを行うには、DynamoDB Streams の実際のレコードに似た情報を入力します。

次の内容で、payload.json というファイルを作成します。*region* および *accountID* をお客様の AWS リージョンとアカウント ID に置き換えます。

```
{
    "Records": [
```

```
        {
            "eventID": "7de3041dd709b024af6f29e4fa13d34c",
            "eventName": "INSERT",
            "eventVersion": "1.1",
            "eventSource": "aws:dynamodb",
            "awsRegion": "region",
            "dynamodb": {
                "ApproximateCreationDateTime": 1479499740,
                "Keys": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "NewImage": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"
                    },
                    "Message": {
                        "S": "This is a bark from the Woofer social network"
                    },
                    "Username": {
                        "S": "John Doe"
                    }
                },
                "SequenceNumber": "13021600000000001596893679",
                "SizeBytes": 112,
                "StreamViewType": "NEW_IMAGE"
            },
            "eventSourceARN": "arn:aws:dynamodb:region:account ID:table/BarkTable/
stream/2016-11-16T20:42:48.104"
        }
    ]
}
```

次のコマンドを入力して、publishNewBark 関数をテストします。

```
aws lambda invoke --function-name publishNewBark --payload file://payload.json --
cli-binary-format raw-in-base64-out output.txt
```

テストが成功すると、次の出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

さらに、output.txt ファイルには次のテキストが含まれます。

```
"Successfully processed 1 records."
```

また、数分以内に新しい E メールメッセージが届きます。

> **Note**
>
> AWS Lambda は、Amazon CloudWatch Logs に診断情報を書き込みます。Lambda 関数でエラーが発生した場合、この診断情報をトラブルシューティングに使用できます。
>
> 1. CloudWatch コンソール (https://console.aws.amazon.com/cloudwatch/) を開きます。
> 2. ナビゲーションペインで [ログ] を選択します。
> 3. 次のロググループを選択: /aws/lambda/publishNewBark
> 4. 最新のログストリーミングを選択して、関数からの出力 (およびエラー) を表示します。

ステップ 5: トリガーを作成してテストする

ステップ 4: Lambda 関数を作成してテストする で、Lambda 関数をテストして、正しく実行されたことを確認しました。このステップでは、Lambda 関数 (publishNewBark) をイベントソース (BarkTable ストリーミング) に関連付けることでトリガーを作成します。

1. トリガーを作成する場合、BarkTable ストリーム用の ARN を指定する必要があります。この ARN を取得するには、次のコマンドを入力します。

   ```
   aws dynamodb describe-table --table-name BarkTable
   ```

   出力で、LatestStreamArn を探します。

   ```
   ...
   ```

```
 "LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/
stream/timestamp
...
```

2. 次のコマンドを入力してトリガーを作成します *streamARN* を実際のストリーム ARN に置き換えます。

```
aws lambda create-event-source-mapping \
    --region region \
    --function-name publishNewBark \
    --event-source streamARN  \
    --batch-size 1 \
    --starting-position TRIM_HORIZON
```

3. トリガーをテストします。次のコマンドを入力して、BarkTable に項目を追加します。

```
aws dynamodb put-item \
    --table-name BarkTable \
    --item Username={S="Jane
 Doe"},Timestamp={S="2016-11-18:14:32:17"},Message={S="Testing...1...2...3"}
```

数分以内に新しい E メールメッセージが届きます。

4. DynamoDB コンソールを開き、さらにいくつかの項目を BarkTable に追加します。Username および Timestamp 属性の値を指定する必要があります (必須ではないものの、Message の値を指定する必要があります)。BarkTable に追加した各項目について、新しい E メールメッセージが届きます。

Lambda 関数は、BarkTable に追加した新しい項目のみを処理します。テーブル内の項目を更新または削除すると、この関数は何も行いません。

> **Note**
>
> AWS Lambda は、Amazon CloudWatch Logs に診断情報を書き込みます。Lambda 関数でエラーが発生した場合、この診断情報をトラブルシューティングに使用できます。
>
> 1. CloudWatch コンソール (https://console.aws.amazon.com/cloudwatch/) を開きます。
> 2. ナビゲーションペインで [ログ] を選択します。
> 3. 次のロググループを選択: /aws/lambda/publishNewBark

4. 最新のログストリーミングを選択して、関数からの出力 (およびエラー) を表示します。

チュートリアル #2: DynamoDB と Lambda での、フィルターを使用したいくつかのイベントの処理。

このチュートリアルでは、AWS Lambda トリガーを作成して、DynamoDB テーブルからのストリームの一部のイベントのみを処理します。

トピック

- すべてをまとめる - CloudFormation
- すべてをまとめる - CDK

Lambda イベントフィルタリングでは、フィルター式を使用して、処理のために Lambda が関数に送信するイベントを制御できます。DynamoDB ストリームごとに最大 5 つの異なるフィルターを設定できます。バッチ処理ウィンドウを使用している場合、Lambda は新しいイベントそれぞれにフィルター条件を適用して、現在のバッチに含めるかどうかを確認します。

フィルターは `FilterCriteria` と呼ばれる構造を介して適用されます。`FilterCriteria` の 3 つの主な属性は `metadata properties`、`data properties`、および `filter patterns` です。

DynamoDB Streams イベントの構造の例を次に示します。

```
{
  "eventID": "c9fbe7d0261a5163fcb6940593e41797",
  "eventName": "INSERT",
  "eventVersion": "1.1",
  "eventSource": "aws:dynamodb",
  "awsRegion": "us-east-2",
  "dynamodb": {
    "ApproximateCreationDateTime": 1664559083.0,
    "Keys": {
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" }
    },
    "NewImage": {
      "quantity": { "N": "50" },
      "company_id": { "S": "1000" },
      "fabric": { "S": "Florida Chocolates" },
      "price": { "N": "15" },
```

```
      "stores": { "N": "5" },
      "product_id": { "S": "1000" },
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" },
      "state": { "S": "FL" },
      "type": { "S": "" }
    },
    "SequenceNumber": "700000000000888747038",
    "SizeBytes": 174,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
  },
  "eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-
StreamsSampleDDBTable-LUOI6UXQY7J1/stream/2022-09-30T17:05:53.209"
}
```

`metadata properties` は、イベントオブジェクトのフィールドです。DynamoDB Streams の場合、`metadata properties` は `dynamodb` や `eventName` のようなフィールドです。

`data properties` は、イベント本文のフィールドです。`data properties` をフィルタリングするには、それらを適切なキー内の `FilterCriteria` に含めるようにしてください。DynamoDB イベントソースのデータキーは `NewImage` または `OldImage` です。

最後に、フィルタールールは、特定のプロパティに適用するフィルター式を定義します。次に例を示します。

比較演算子	例	ルール構文 (一部)
Null	製品タイプは NULL	`{ "product_type": { "S": null } }`
空	製品名は空白	`{ "product_name": { "S": [ ""] } }`
Equals	州はフロリダに等しい	`{ "state": { "S": ["FL"] } }`
And	製品州はフロリダ、製品カテゴリはチョコレート	`{ "state": { "S": ["FL"] } , "category": { "S": [ "CHOCOLATE"] } }`

比較演算子	例	ルール構文 (一部)
または	製品州はフロリダまたはカリフォルニア	`{ "state": { "S": ["FL","CA"] } }`
Not	製品州はフロリダ州ではない	`{"state": {"S": [{"anything-but": ["FL"]}]}}`
存在する	地産品は存在する	`{"homemade": {"S": [{"exists": true}]}}`
存在しない	地産品は存在しない	`{"homemade": {"S": [{"exists": false}]}}`
Begins with	PK は COMPANY から始まる	`{"PK": {"S": [{"prefix": "COMPANY"}]}}`

Lambda 関数には、最大 5 つのイベントフィルタリングパターンを指定できます。これら 5 つのイベントのそれぞれが論理 OR として評価されることに注意してください。そのため、`Filter_One` および `Filter_Two` という名前の 2 つのフィルターを設定すると、Lambda 関数は `Filter_One` OR `Filter_Two` を実行します。

> **Note**
>
> Lambda イベントのフィルタリングページには、数値をフィルタリングして比較するオプションがいくつかありますが、DynamoDB のフィルタイベントの場合、DynamoDB の数値は文字列として保存されるため、適用されません。例えば `"quantity": { "N": "50" }` の場合は、`"N"` プロパティのおかげでそれが数字だとわかります。

すべてをまとめる - CloudFormation

実際のイベントフィルタリング機能を見ていただくために、CloudFormation テンプレートのサンプルを以下に示します。このテンプレートは、Amazon DynamoDB Streams を有効にしたパーティションキー PK とソートキー SK を含むシンプルな DynamoDB テーブルを生成します。これにより、Amazon Cloudwatch へのログの書き込みと Amazon DynamoDB Streams から

のイベントの読み取りを許可する Lambda 関数とシンプルな Lambda 実行ロールが作成されます。また、DynamoDB Streams と Lambda 関数間にイベントソースマッピングも追加されるため、Amazon DynamoDB Streams にイベントが発生するたびに関数を実行できます。

```
AWSTemplateFormatVersion: "2010-09-09"

Description: Sample application that presents AWS Lambda event source filtering
with Amazon DynamoDB Streams.

Resources:
  StreamsSampleDDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: "PK"
          AttributeType: "S"
        - AttributeName: "SK"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "PK"
          KeyType: "HASH"
        - AttributeName: "SK"
          KeyType: "RANGE"
      StreamSpecification:
        StreamViewType: "NEW_AND_OLD_IMAGES"
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5

  LambdaExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17",
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - sts:AssumeRole
      Path: "/"
      Policies:
```

```
        - PolicyName: root
          PolicyDocument:
            Version: "2012-10-17",
            Statement:
              - Effect: Allow
                Action:
                  - logs:CreateLogGroup
                  - logs:CreateLogStream
                  - logs:PutLogEvents
                Resource: arn:aws:logs:*:*:*
              - Effect: Allow
                Action:
                  - dynamodb:DescribeStream
                  - dynamodb:GetRecords
                  - dynamodb:GetShardIterator
                  - dynamodb:ListStreams
                Resource: !GetAtt StreamsSampleDDBTable.StreamArn

  EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST

  ProcessEventLambda:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      Timeout: 300
      Handler: index.handler
      Role: !GetAtt LambdaExecutionRole.Arn
      Code:
        ZipFile: |
          import logging

          LOGGER = logging.getLogger()
          LOGGER.setLevel(logging.INFO)

          def handler(event, context):
            LOGGER.info('Received Event: %s', event)
            for rec in event['Records']:
```

```
            LOGGER.info('Record: %s', rec)

Outputs:
  StreamsSampleDDBTable:
    Description: DynamoDB Table ARN created for this example
    Value: !GetAtt StreamsSampleDDBTable.Arn
  StreamARN:
    Description: DynamoDB Table ARN created for this example
    Value: !GetAtt StreamsSampleDDBTable.StreamArn
```

この CloudFormation テンプレートをデプロイすると、次の Amazon DynamoDB 項目を挿入できます。

```
{
 "PK": "COMPANY#1000",
 "SK": "PRODUCT#CHOCOLATE#DARK",
 "company_id": "1000",
 "type": "",
 "state": "FL",
 "stores": 5,
 "price": 15,
 "quantity": 50,
 "fabric": "Florida Chocolates"
}
```

この CloudFormation テンプレートにインラインで組み込まれているシンプルな Lambda 関数により、Lambda 関数の Amazon CloudWatch ロググループのイベントは次のように表示されます。

```
{
  "eventID": "c9fbe7d0261a5163fcb6940593e41797",
  "eventName": "INSERT",
  "eventVersion": "1.1",
  "eventSource": "aws:dynamodb",
  "awsRegion": "us-east-2",
  "dynamodb": {
    "ApproximateCreationDateTime": 1664559083.0,
    "Keys": {
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" }
    },
    "NewImage": {
      "quantity": { "N": "50" },
```

```
      "company_id": { "S": "1000" },
      "fabric": { "S": "Florida Chocolates" },
      "price": { "N": "15" },
      "stores": { "N": "5" },
      "product_id": { "S": "1000" },
      "SK": { "S": "PRODUCT#CHOCOLATE#DARK#1000" },
      "PK": { "S": "COMPANY#1000" },
      "state": { "S": "FL" },
      "type": { "S": "" }
    },
    "SequenceNumber": "700000000000888747038",
    "SizeBytes": 174,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
  },
  "eventSourceARN": "arn:aws:dynamodb:us-east-2:111122223333:table/chocolate-table-
StreamsSampleDDBTable-LUOI6UXQY7J1/stream/2022-09-30T17:05:53.209"
}
```

フィルター例

- 特定の州に一致する商品のみ

この例では、CloudFormation テンプレートを変更して、フロリダで生産されたすべての製品 (略称「FL」) と一致するフィルターを含めます。

```
EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      FilterCriteria:
        Filters:
          - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST
```

スタックを再デプロイしたら、テーブルに次の DynamoDB 項目を追加できます。この例での製品はカリフォルニア産なので、Lambda 関数ログには表示されないことに注意してください。

```
{
 "PK": "COMPANY#1000",
```

```
 "SK": "PRODUCT#CHOCOLATE#DARK#1000",
 "company_id": "1000",
 "fabric": "Florida Chocolates",
 "price": 15,
 "product_id": "1000",
 "quantity": 50,
 "state": "CA",
 "stores": 5,
 "type": ""
}
```

- PK と SK のうち、ある値で始まるアイテムのみ

この例では、CloudFormation テンプレートを変更して次の条件を含めます。

```
EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      FilterCriteria:
        Filters:
          - Pattern: '{"dynamodb": {"Keys": {"PK": { "S": [{ "prefix":
 "COMPANY" }] },"SK": { "S": [{ "prefix": "PRODUCT" }] }}}}'
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST
```

AND 条件では、条件がパターン内になければならないことに注意してください。ここで、キー PK と SK は同じ式内にあり、カンマで区切られます。

PK と SK がある値から始まるか、特定の州産かのどちらかです。

この例では、CloudFormation テンプレートを変更して次の条件を含めます。

```
  EventSourceDDBTableStream:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      BatchSize: 1
      Enabled: True
      FilterCriteria:
        Filters:
```

```
          - Pattern: '{"dynamodb": {"Keys": {"PK": { "S": [{ "prefix":
"COMPANY" }] },"SK": { "S": [{ "prefix": "PRODUCT" }] }}}}'
          - Pattern: '{ "dynamodb": { "NewImage": { "state": { "S": ["FL"] } } } }'
      EventSourceArn: !GetAtt StreamsSampleDDBTable.StreamArn
      FunctionName: !GetAtt ProcessEventLambda.Arn
      StartingPosition: LATEST
```

フィルターセクションに新しいパターンを導入することで、OR 条件が追加されます。

すべてをまとめる - CDK

次のサンプル CDK プロジェクト形成テンプレートでは、イベントフィルタリング機能について説明します。この CDK プロジェクトに取り組む前に、準備スクリプトの実行を含む前提条件をインストールする必要があります。

CDK プロジェクトを作成する

まず、空のディレクトリで `cdk init` を呼び出して、新しい AWS CDK プロジェクトを作成します。

```
mkdir ddb_filters
cd ddb_filters
cdk init app --language python
```

この `cdk init` コマンドは、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名に含まれるハイフンはすべてアンダースコアに変換されます。それ以外の場合、名前は Python 識別子の形式に従う必要があります。例えば、数字で始めたり、スペースを含めたりはしないでください。

新しいプロジェクトで作業するには、その仮想環境を有効にします。これにより、プロジェクトの依存関係をグローバルにインストールするのではなく、プロジェクトフォルダにローカルにインストールできます。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

> **Note**
>
> これは、仮想環境をアクティブにする Mac/Linux コマンドとして認識されるかもしれません。Python テンプレートには、同じコマンドを Windows で使用できるようにするバッチファイル、`source.bat` が含まれています。従来の Windows コマンド、`.venv\Scripts`

\activate.bat も機能します。AWS CDK Toolkit v1.70.0 以前を使用して AWS CDK プロジェクトを初期化した場合、仮想環境は .venv ではなく .env ディレクトリにあります。

基本インフラストラクチャ

任意のテキストエディタでファイル ./ddb_filters/ddb_filters_stack.py を開きます。このファイルは、AWS CDK プロジェクトを作成したときに自動生成されました。

次に、_create_ddb_table および _set_ddb_trigger_function 関数を追加します。これらの関数は、プロビジョニングモードのオンデマンドモードでパーティションキー PK とソートキー SK を含む DynamoDB テーブルを作成します。Amazon DynamoDB Streams をデフォルトで有効にして、新しいイメージと古いイメージを表示できます。

Lambda 関数はファイル app.py の下のフォルダ lambda に保存されます。このファイルは後で作成されます。これには環境変数 APP_TABLE_NAME が含まれます。この変数は、このスタックによって作成される Amazon DynamoDB テーブルの名前になります。同じ関数で、コードは Lambda 関数にストリーム読み取り権限を付与します。最後に、Lambda 関数のイベントソースとして DynamoDB Streams にサブスクライブします。

__init__ メソッド内のファイルの最後で、それぞれの構成を呼び出してスタック内で初期化します。追加のコンポーネントやサービスを必要とする大規模なプロジェクトでは、これらの構成を基本スタックの外部で定義するのが最適な場合があります。

```
import os
import json

import aws_cdk as cdk
from aws_cdk import (
    Stack,
    aws_lambda as _lambda,
    aws_dynamodb as dynamodb,
)
from constructs import Construct


class DdbFiltersStack(Stack):

    def _create_ddb_table(self):
        dynamodb_table = dynamodb.Table(
            self,
```

```
            "AppTable",
            partition_key=dynamodb.Attribute(
                name="PK", type=dynamodb.AttributeType.STRING
            ),
            sort_key=dynamodb.Attribute(
                name="SK", type=dynamodb.AttributeType.STRING),
            billing_mode=dynamodb.BillingMode.PAY_PER_REQUEST,
            stream=dynamodb.StreamViewType.NEW_AND_OLD_IMAGES,
            removal_policy=cdk.RemovalPolicy.DESTROY,
        )

        cdk.CfnOutput(self, "AppTableName", value=dynamodb_table.table_name)
        return dynamodb_table

    def _set_ddb_trigger_function(self, ddb_table):
        events_lambda = _lambda.Function(
            self,
            "LambdaHandler",
            runtime=_lambda.Runtime.PYTHON_3_9,
            code=_lambda.Code.from_asset("lambda"),
            handler="app.handler",
            environment={
                "APP_TABLE_NAME": ddb_table.table_name,
            },
        )

        ddb_table.grant_stream_read(events_lambda)

        event_subscription = _lambda.CfnEventSourceMapping(
            scope=self,
            id="companyInsertsOnlyEventSourceMapping",
            function_name=events_lambda.function_name,
            event_source_arn=ddb_table.table_stream_arn,
            maximum_batching_window_in_seconds=1,
            starting_position="LATEST",
            batch_size=1,
        )

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        ddb_table = self._create_ddb_table()
        self._set_ddb_trigger_function(ddb_table)
```

次に、Amazon CloudWatch にログを出力する、シンプルな Lambda 関数を作成します。それには、lambda という新しいフォルダを作成します。

```
mkdir lambda
touch app.py
```

任意のテキストエディタを使用して、次の内容を app.py ファイルに追加します。

```
import logging

LOGGER = logging.getLogger()
LOGGER.setLevel(logging.INFO)


def handler(event, context):
    LOGGER.info('Received Event: %s', event)
    for rec in event['Records']:
        LOGGER.info('Record: %s', rec)
```

/ddb_filters/ フォルダにいることを確認し、次のコマンドを入力してサンプルアプリケーションを作成します。

```
cdk deploy
```

ある時点で、ソリューションをデプロイするかどうか確認する画面が表示されます。Y を入力して変更を確定します。

```
##################################################################################################
# + # ${LambdaHandler/ServiceRole} # arn:${AWS::Partition}:iam::aws:policy/service-
role/AWSLambdaBasicExecutionRole #
##################################################################################################

Do you wish to deploy these changes (y/n)? y

...

#  Deployment time: 67.73s

Outputs:
DdbFiltersStack.AppTableName = DdbFiltersStack-AppTable815C50BC-1M1W7209V5YPP
Stack ARN:
```

```
arn:aws:cloudformation:us-east-2:111122223333:stack/
DdbFiltersStack/66873140-40f3-11ed-8e93-0a74f296a8f6
```

変更がデプロイされたら、AWS コンソールを開いてテーブルに項目を 1 つ追加します。

```
{
 "PK": "COMPANY#1000",
 "SK": "PRODUCT#CHOCOLATE#DARK",
 "company_id": "1000",
 "type": "",
 "state": "FL",
 "stores": 5,
 "price": 15,
 "quantity": 50,
 "fabric": "Florida Chocolates"
}
```

これで CloudWatch ログには、このエントリの情報がすべて含まれているはずです。

フィルター例

- 特定の州に一致する商品のみ

ファイル ddb_filters/ddb_filters/ddb_filters_stack.py を開き、「FL」に等しいすべての製品と一致するフィルターを含めるように変更します。これは 45 行目の event_subscription のすぐ下で修正できます。

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}
                )
            },
        ]
    },
)
```

- PK と SK のうち、ある値で始まる項目のみ

Python スクリプトを次の条件を含むように変更します。

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {
                        {
                            "dynamodb": {
                                "Keys": {
                                    "PK": {"S": [{"prefix": "COMPANY"}]},
                                    "SK": {"S": [{"prefix": "PRODUCT"}]},
                                }
                            }
                        }
                    }
                )
            },
        ]
    },
```

- PK と SK がある値から始まるか、特定の州産かのどちらかです。

Python スクリプトを次の条件を含むように変更します。

```
event_subscription.add_property_override(
    property_path="FilterCriteria",
    value={
        "Filters": [
            {
                "Pattern": json.dumps(
                    {
                        {
                            "dynamodb": {
                                "Keys": {
                                    "PK": {"S": [{"prefix": "COMPANY"}]},
                                    "SK": {"S": [{"prefix": "PRODUCT"}]},
                                }
                            }
                        }
```

```
                    }
                )
            },
            {
                "Pattern": json.dumps(
                    {"dynamodb": {"NewImage": {"state": {"S": ["FL"]}}}}
                )
            },
        ]
    },
)
```

Filters 配列に要素を追加することで OR 条件が追加されることに注意してください。

クリーンアップ

作業ディレクトリのベースにあるフィルタースタックを見つけて、`cdk destroy` を実行します。リソースの削除を確認するメッセージが表示されます。

```
cdk destroy
Are you sure you want to delete: DdbFiltersStack (y/n)? y
```

Lambda での DynamoDB ストリームの使用に関するベストプラクティス

AWS Lambda 関数は、他の関数から分離された実行環境であるコンテナ内で実行されます。この関数を初めて実行すると、AWS Lambda は新しいコンテナを作成し、関数のコードを実行し始めます。

Lambda 関数には、コールごとに 1 回実行されるハンドラがあります。ハンドラには、関数用の主要なビジネスロジックが含まれます。たとえば、ステップ 4: Lambda 関数を作成してテストする に示す Lambda 関数には、DynamoDB Streams のレコードを処理できるハンドラがあります。

コンテナの作成後、AWS Lambda が初めてハンドラーを実行する前に、1 回だけ実行される初期化コードを提供することもできます。ステップ 4: Lambda 関数を作成してテストする に示す Lambda 関数には、SDK for JavaScript in Node をインポートし、Amazon SNS 用のクライアントを作成する初期化コードがあります。これらのオブジェクトはハンドラの外部で 1 回のみ定義します。

関数の実行後、AWS Lambda は関数のそれ以降の呼び出しに対してコンテナを再利用する場合があります。この場合、関数ハンドラは、初期化コードで定義したリソースを再利用できる可能性があります (AWS Lambda がコンテナを保持する期間や、コンテナを再利用するかどうかを制御することはできません)。

AWS Lambda を使用した DynamoDB トリガーの場合は、次のことをお勧めします。

- AWS のサービスのクライアントは、ハンドラではなく初期化コードでインスタンス化する必要があります。これにより、AWS Lambda コンテナは、コンテナの有効期間中は既存の接続を再利用することができます。
- 通常、お客様が明示的に接続を管理したり、接続プールを実装したりする必要はありません。これは AWS Lambda によって自動的に管理されます。

DynamoDB ストリームの Lambda コンシューマーは、少なくとも 1 回の配信を行います。確実に 1 回のみの配信は保証されず、同じレコードを複数回処理することがあります (例えば、関数呼び出しが再試行された場合など)。これは、各ストリームレコードがストリームに確実に 1 回だけ表示される DynamoDB Streams の保証とは異なります。この保証は、コンシューマーがレコードを処理する回数ではなく、ストリームに保存されているレコードに関するものです。重複処理で予期しない問題が発生しないように、Lambda 関数コードは必ずべき等性にしてください。

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数を操作するためのベストプラクティス」を参照してください。

DynamoDB Streams と Apache Flink

Apache Flink で Amazon DynamoDB Streams レコードを使用できます。Amazon Managed Service for Apache Flink を使用すると、Apache Flink を使用してストリーミングデータをリアルタイムで変換および分析できます。Apache Flink は、リアルタイムデータを処理するためのオープンソースのストリーム処理フレームワークです。Apache Flink 用 Amazon DynamoDB Streams コネクタは、Apache Flink ワークロードの構築と管理を簡素化し、アプリケーションを他の AWS のサービスと統合できるようにします。

Amazon Managed Service for Apache Flink は、ログ分析、クリックストリーム分析、モノのインターネット (IoT)、アドテック、ゲームなどのためのエンドツーエンドのストリーム処理アプリケーションを迅速に構築するのに役立ちます。最も一般的な 4 つのユースケースは、ストリーミングの抽出変換ロード (ETL)、イベント駆動型アプリケーション、応答性の高いリアルタイム分析、データストリームのインタラクティブなクエリです。Amazon DynamoDB Streams から Apache Flink への書き込みの詳細については、「Amazon DynamoDB Connector」を参照してください。

DynamoDB Accelerator (DAX) とインメモリアクセラレーション

Amazon DynamoDB はスケールとパフォーマンスのために設計されています。ほとんどの場合、DynamoDB の応答時間は 1 桁台のミリ秒単位で測定できます。ただし、マイクロ秒の応答時間を必要とする一部のユースケースがあります。これらのユースケースでは DynamoDB Accelerator (DAX) が整合性データへのアクセス時に素早い対応を実現します。

DAX は、DynamoDB と互換性のあるキャッシュサービスで、条件の厳しいアプリケーションで高速なインメモリパフォーマンスを可能にします。DAX は、次の 3 つのコアシナリオに対応します。

1. インメモリキャッシュとしての DAX は、1 桁台のミリ秒単位からマイクロ秒単位まで、結果整合性のある読み込みワークロードの応答時間を短縮します。
2. DAX は、DynamoDB と API の互換性のあるマネージドサービスを提供することにより、オペレーションとアプリケーションの複雑さを軽減します。したがって、既存のアプリケーションで使用するために必要なのは最小限の機能変更だけです。
3. 読み込みの多いワークロードや急激に増大するワークロードにおいて、DAX はスループットを強化することや、読み込み容量ユニットを必要以上にプロビジョニングしないようにすることで運用コストの節約を可能にします。個々のキーで繰り返し読み込みが必要なアプリケーションにおいては特にメリットがあります。

DAX はサーバー側の暗号化をサポートします。保存時の暗号化では、ディスク上で DAX によって保持されるデータが暗号化されます。DAX は、プライマリノードからリードレプリカへの変更の伝播の一部として、データをディスクに書き込みます。詳細については、「」を参照してください保管時の DAX 暗号化

DAX は転送中の暗号化もサポートしており、アプリケーションとクラスター間のすべてのリクエストとレスポンスがトランスポートレベルセキュリティ (TLS) によって暗号化され、クラスターへの接続はクラスター x509 証明書の検証によって認証されます。詳細については、「」を参照してください転送時の DAX 暗号化

トピック

- DAX のユースケース
- DAX の使用に関する注意事項
- DAX: 仕組み

- DAX クラスターコンポーネント
- DAX クラスターの作成
- DAX および DynamoDB の整合性モデル
- DynamoDB Accelerator (DAX) クライアントで開発する
- DAX クラスターの管理
- DynamoDB Accelerator のモニタリング
- DAX T3/T2 バーストインスタンス
- DAX のアクセスコントロール
- 保管時の DAX 暗号化
- 転送時の DAX 暗号化
- DAX のサービスにリンクされた IAM ロールの使用
- AWS アカウント間での DAX へのアクセス
- DAX クラスターサイジングガイド

DAX のユースケース

DAX は、マイクロ秒のレイテンシーで DynamoDB テーブルからの結果整合性データへのアクセスを提供します。マルチ AZ DAX クラスターは、1 秒間に数百万件のリクエストを処理できます。

DAX は、次のタイプのアプリケーションに最適です。

- 可能な限り迅速な読み込み応答時間を必要とするアプリケーション。たとえば、リアルタイム入札、ソーシャルゲーム、トレーディングアプリケーションなどです。DAX はこれらのユースケースに対して、高速なインメモリ読み取りパフォーマンスを提供します。
- 少数の項目を他のものよりも頻繁に読み込むアプリケーション。たとえば、人気商品の 1 日限りのセールを行う e コマースシステムを考えてみます。セール中は、その商品 (および DynamoDB 内のそのデータ) に対する需要が他の商品全般に比べて急増します。「ホット」キーおよび不均一なトラフィックディストリビューションの影響を緩和するために、1 日限りのセールが終了するまで、読み込みアクティビティを DAX キャッシュにオフロードできます。
- 読み込み負荷が高いが、コストも重要なアプリケーション。DynamoDB を使用する場合、アプリケーションが要求する読み込み数を 1 秒ごとにプロビジョニングします。読み込みアクティビティが上昇すれば、テーブルにプロビジョニングされた読み込みスループットも増加し (追加コストがかかり) ます。または、アプリケーションからのアクティビティを DAX クラスターにオフ

ロードして、そうしない場合に購入する必要がある読み込みキャパシティユニットの数を削減できます。

- 大規模データセットに対して繰り返し読み込みが必要なアプリケーション。このようなアプリケーションは、データベースリソースを他のアプリケーションから引き離してしまう可能性があります。たとえば、リージョンの天気データの長期分析では、DynamoDB テーブルのすべての読み込みキャパシティが一時的に消費される可能性があります。この状況は、同じデータにアクセスする必要がある他のアプリケーションに悪影響を及ぼします。DAX を使用することで、代わりにキャッシュデータに対して気象分析を実行できます。

DAX は、次のタイプのアプリケーションには最適ではありません。

- 強整合性のある読み込みを必要とする (または結果整合性のある読み込みを許容できない) アプリケーション。
- 読み込みでマイクロ秒の応答時間を要求しない、または基礎となるテーブルから繰り返される読み込みアクティビティをオフロードする必要がないアプリケーション。
- 書き込み負荷の高いアプリケーション。書き込み量が多い場合、クラスター内の DAX ノード間のレプリケーションが増加します。これにより、リソースの消費量が増加し、可用性の問題が発生するリスクがあります。
- 読み込みの繰り返しが少ないアプリケーション。DAX は、キャッシュヒットレートが 90% を超える場合に最適に動作します。キャッシュヒットレートが低いとキャッシュミスが増加し、DAX クラスター全体でより多くのリソースが消費されます。

DAX の使用に関する注意事項

- DAX を使用できる AWS リージョンのリストについては、「Amazon DynamoDB の料金表」を参照してください。
- DAX は、Go、Java、Node.js、Python、および .NET で記述されたアプリケーションをサポートします (これらのプログラミング言語用に AWS が提供するクライアントを使用します)。
- DAX は EC2-VPC プラットフォームでのみ使用できます。
- DAX クラスターサービスロールポリシーは、DynamoDB テーブルに関するメタデータを維持するために `dynamodb:DescribeTable` アクションを許可する必要があります。
- DAX クラスターは、格納する項目の属性名に関するメタデータを保持します。そのメタデータは (項目の有効期限が切れた後、またはキャッシュから削除された後でも) 無期限に保持されます。大量の属性名を使用するアプリケーションは、時間の経過につれて DAX クラスターのメモリ消耗を

引き起こすことがあります。この制限は最上位の属性名のみに適用され、ネスト属性名には適用されません。問題が発生しやすい最上位の属性名の例としては、タイムスタンプ、UUID や ID があります。

この制限は属性名のみに適用され、その値には適用されません。以下のような項目に問題はありません。

```
{
    "Id": 123,
    "Title": "Bicycle 123",
    "CreationDate": "2017-10-24T01:02:03+00:00"
}
```

ただし、以下のような項目は、十分な数があり、それぞれ別のタイムスタンプがある場合は、問題になります。

```
{
    "Id": 123,
    "Title": "Bicycle 123",
    "2017-10-24T01:02:03+00:00": "created"
}
```

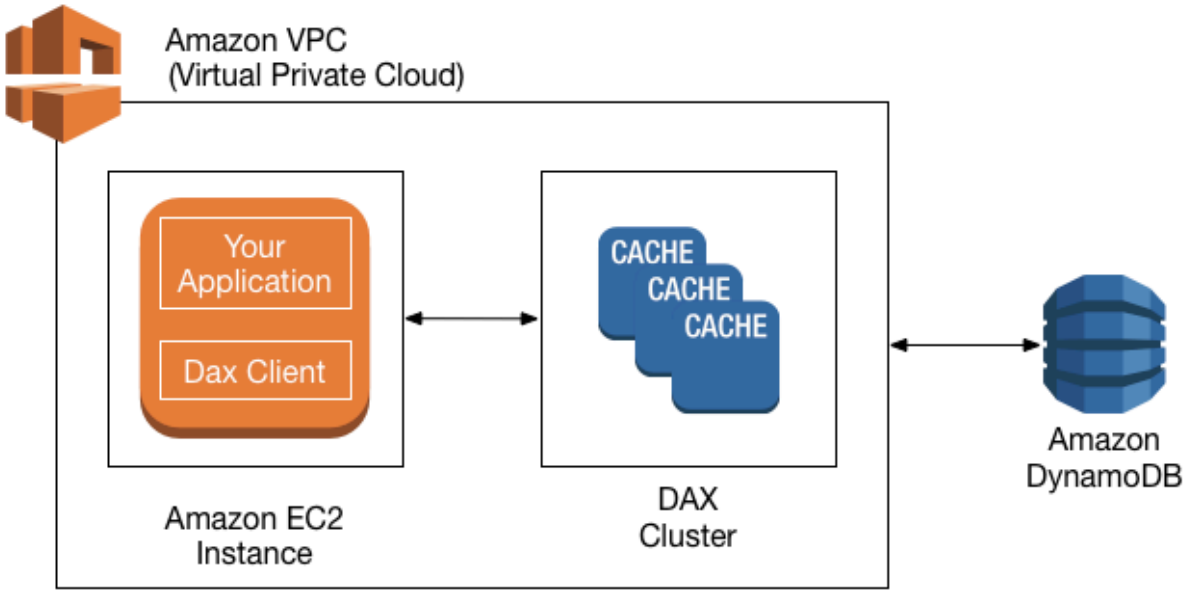
DAX: 仕組み

Amazon DynamoDB Accelerator (DAX) は、Amazon Virtual Private Cloud (Amazon VPC) 環境内で実行するように設計されています。Amazon VPC サービスは、従来のデータセンターに非常によく似た仮想ネットワークを定義します。VPC を使用すると、IP アドレス範囲、サブネット、ルーティングテーブル、ネットワークゲートウェイ、セキュリティ設定を適切に制御できます。仮想ネットワーク内で DAX クラスターを起動し、Amazon VPC のセキュリティグループを使用して、クラスターへのアクセスを制御できます。

> **Note**
>
> 2013 年 12 月 4 日以降に AWS アカウントを作成した場合は、各 AWS リージョンにデフォルトで VPC が用意されています。VPC はすぐに使用できます。追加のステップは必要ありません。

詳細については、「Amazon VPC ユーザーガイド」の「デフォルト VPC とデフォルトサブネット」を参照してください。

次の図は、DAX の高レベルの概要を示しています。



DAX クラスターを作成するには、AWS マネジメントコンソール を使用します。特に指定しない限り、DAX クラスターはデフォルト VPC 内で実行されます。アプリケーションを実行するには、Amazon VPC 内に Amazon EC2 インスタンスを起動します。次に、アプリケーションを (DAX クライアントとともに) EC2 インスタンスにデプロイします。

実行時に、DAX クライアントはアプリケーションのすべての DynamoDB API リクエストを DAX クラスターに送信します。DAX がこれらの API リクエストのいずれかを直接処理できる場合は、処理します。それ以外の場合は、リクエストは DynamoDB に渡されます。

最終的に、DAX クラスターはアプリケーションに結果を返します。

トピック

- DAX でのリクエスト処理方法
- 項目キャッシュ
- クエリキャッシュ

DAX でのリクエスト処理方法

DAX クラスターは 1 つ以上のノードで構成されています。各ノードはそれぞれ、自身の DAX キャッシングソフトウェアのインスタンスを実行します。ノードのうち 1 つがクラスターのプライマリノードとして機能します。他のノード (存在する場合) はリードレプリカとして動作します。詳細については、「ノード」を参照してください。

アプリケーションは、DAX クラスターのエンドポイントを指定することで DAX にアクセスできます。DAX クライアントソフトウェアはクラスターエンドポイントと連携して、インテリジェントなロードバランシングとルーティングを実行します。

読み込み操作

DAX は次の API コールに対応できます。

- `GetItem`
- `BatchGetItem`
- `Query`
- `Scan`

リクエストが結果的に整合性のある読み込み (デフォルトの動作) を指定する場合は、その項目を DAX から読み込もうとします。

- DAX に項目がある場合 (キャッシュヒット)、DAX は DynamoDB にアクセスせずにアプリケーションに項目を返します。
- DAX に項目がない場合 (キャッシュミス)、DAX はリクエストを DynamoDB に渡します。DynamoDB からの応答を受信すると、DAX は結果をアプリケーションに返します。ただし、プライマリノードのキャッシュにも結果を書き込みます。

> **Note**
>
> クラスターにリードレプリカがある場合、DAX は自動的にレプリカをプライマリノードと同期した状態に保ちます。詳細については、「」を参照してくださいクラスター

リクエストが強力な整合性のある読み込みを指定する場合、DAX はリクエストを DynamoDB に渡します。DynamoDB からの結果は DAX にキャッシュされません。代わりに、それらは単にアプリケーションに返されます。

書き込み操作

次の DAX API オペレーションは「書き込みスルー」と見なされます。

- `BatchWriteItem`
- `UpdateItem`
- `DeleteItem`
- `PutItem`

これらのオペレーションでは、データはまず DynamoDB テーブルに書き込まれ、その後 DAX クラスターに書き込まれます。オペレーションは、データがテーブルと DAX の両方に正常に書き込まれた場合にのみ成功します。

その他のオペレーション

DAX では、テーブルを管理する DynamoDB のオペレーション (`CreateTable` や `UpdateTable` など) を認識しません。アプリケーションがこのようなオペレーションを行う必要がある場合は、DAX を使用するのではなく、直接 DynamoDB にアクセスする必要があります。

DAX および DynamoDB の整合性の詳細については、「DAX および DynamoDB の整合性モデル」を参照してください。

DAX でのトランザクションの動作の詳細については、「DynamoDB アクセラレーター (DAX) でのトランザクション API の使用」を参照してください。

リクエストレート制限

DAX に送信された要求の数がノードの容量を超える場合、DAX は ThrottlingException を返して追加の要求を受け入れるレートを制限します。DAX は CPU 使用率を継続的に評価し、正常なクラスター状態を維持しながら処理できるリクエスト量を判断します。

DAX が Amazon CloudWatch に公開する ThrottledRequestCount メトリクスをモニタリングできます。これらの例外が定期的に発生する場合は、クラスターをスケールアップすることを検討してください。

項目キャッシュ

DAX は項目キャッシュを維持して、GetItem および BatchGetItem オペレーションからの結果を保存します。キャッシュ内の項目は DynamoDB からの結果整合性データを表し、プライマリキー値別に保存されています。

アプリケーションが GetItem または BatchGetItem リクエストを送信すると、DAX は指定されたキー値を使用して項目キャッシュから直接項目の読み込みを試みます。項目が見つかった場合 (キャッシュヒット)、DAX はすぐにアプリケーションに項目を返します。項目が見つからない (キャッシュミス) 場合、DAX は DynamoDB にリクエストを送信します。DynamoDB は、結果整合性のある読み取りを使用してリクエストを処理し、項目を DAX に返します。DAX は、それらを項目キャッシュに格納し、アプリケーションに返します。

項目キャッシュには有効期限 (TTL) があります。デフォルトでは 5 分です。DAX は項目キャッシュに書き込むすべての項目にタイムスタンプを割り当てます。TTL 設定より長期間キャッシュに残り続けている項目は、期限切れになります。期限切れの項目について GetItem リクエストを発行すると、キャッシュミスとみなされ、DAX は GetItem リクエストを DynamoDB に送信します。

> **Note**
>
> 項目キャッシュの TTL 設定は、新しい DAX クラスターを作成する際に指定できます。詳細については、「」を参照してくださいDAX クラスターの管理

DAX は項目キャッシュの LRU (least recently used) リストも保持します。LRU リストは、項目が最初にキャッシュに書き込まれた時とその項目が最後にキャッシュから読み込まれた時を記録して保存します。項目キャッシュがいっぱいになると、DAX は古い項目を (期限切れでない場合であっても) 削除し、新しい項目のためのスペースを空けます。LRU アルゴリズムは項目キャッシュでは常に有効であり、ユーザーが設定することはできません。

項目キャッシュ TTL 設定としてゼロを指定した場合、項目キャッシュ内の項目は、LRU 削除または「書き込みスルー」オペレーションによってのみ更新されます。

DAX での項目キャッシュの整合性の詳細については、「DAX 項目キャッシュの動作」を参照してください。

クエリキャッシュ

DAX は、クエリキャッシュを維持して、Query および Scan オペレーションからの結果を保存します。このキャッシュの項目は、DynamoDB テーブルに対するクエリおよびスキャンの結果セットを表します。これらの結果セットはパラメータ値別に保存されています。

アプリケーションが Query または Scan リクエストを送信すると、DAX は指定されたパラメータ値を使用してクエリキャッシュから一致する結果セットの読み込みを試みます。結果セットが見つかった場合 (キャッシュヒット)、DAX はすぐにアプリケーションに結果セットを返します。結果セットが見つからない場合 (キャッシュミス)、DAX は DynamoDB にリクエストを送信します。DynamoDB は、結果整合性のある読み取りを使用してリクエストを処理し、結果セットを DAX に返します。DAX は結果セットをクエリキャッシュに保存し、アプリケーションに返します。

> **Note**
>
> クエリキャッシュの TTL 設定は、新しい DAX クラスターを作成する際に指定できます。詳細については、「」を参照してくださいDAX クラスターの管理

DAX は、クエリキャッシュの LRU リストも保持します。リストは、結果セットが最初にキャッシュに書き込まれた時とその結果が最後にキャッシュから読み込まれた時を記録して保存します。クエリキャッシュがいっぱいになると、DAX は古い結果セットを (期限切れでなくても) 削除して新しい結果セットのためのスペースを解放します。LRU アルゴリズムはクエリキャッシュでは常に有効であり、ユーザーが設定することはできません。

クエリキャッシュ TTL 設定としてゼロを指定した場合、クエリ応答はキャッシュされません。

DAX でのクエリキャッシュの整合性の詳細については、「DAX クエリキャッシュの動作」を参照してください。

DAX クラスターコンポーネント

Amazon DynamoDB Accelerator (DAX) クラスターは、AWS のインフラストラクチャのコンポーネントで構成されます。このセクションでは、これらのコンポーネントと、それらがどのように連携するかについて説明します。

トピック

- ノード

- クラスター
- リージョンとアベイラビリティーゾーン
- パラメータグループ
- セキュリティグループ
- クラスター ARN
- クラスターエンドポイント
- ノードエンドポイント
- サブネットグループ
- Events
- メンテナンスウィンドウ

ノード

ノードとは、DAX クラスターにおける最小の構成要素です。各ノードは DAX ソフトウェアのインスタンス 1 つを実行し、キャッシュされたデータのレプリカを 1 つ保持します。

DAX クラスターは 2 通りの方法のいずれかでスケールできます。

- クラスターにさらにノードを追加する。これは、クラスター全体の読み込みスループットを増加します。
- より大きなノードタイプを使用する。ノードタイプが大きくなると、容量が大きくなり、スループットを増やすことができます。(新しいノードタイプで新しいクラスターを作成する必要があります。)

クラスター内の各ノードは同じノードタイプであり、同じ DAX キャッシュソフトウェアを実行します。使用可能なノードタイプのリストについては、「Amazon DynamoDB の料金表」を参照してください。

クラスター

クラスターは、DAX がユニットとして管理する 1 つまたは複数のノードの論理グループです。クラスター内のノードの 1 つがプライマリノードとして指定され、他のノード (ある場合) はリードレプリカです。

プライマリノードは以下を担当します。

- キャッシュ済みデータに対するアプリケーションリクエストの対応。
- DynamoDB への書き込みオペレーションの処理。
- クラスターの削除ポリシーに従った、キャッシュからのデータの削除。

プライマリノードにキャッシュされたデータが変更されると、DAX は、レプリケーションログを使用して、その変更をすべてのリードレプリカノードに伝播します。すべてのリードレプリカから確認を受け取ると、DynamoDB はプライマリノードからレプリケーションログを削除します。

DAX クラスターは、クラスターごとに最大 11 個のノード (プライマリノードと最大 10 個のリードレプリカ) をサポートできます。

リードレプリカは以下を担当します。

- キャッシュ済みデータに対するアプリケーションリクエストの対応。
- クラスターの削除ポリシーに従った、キャッシュからのデータの削除。

ただし、プライマリノードとは異なり、リードレプリカは DynamoDB には書き込みません。

リードレプリカにはさらに 2 つの目的があります。

- スケーラビリティ。DAX への同時アクセスを必要とするクライアントが多数ある場合は、レプリカを追加して読み込みをスケーリングできます。DAX は、クラスター内のすべてのノードに均等にロードを分散します。(スループットを増やすもう 1 つの方法は、より大規模なキャッシュノードタイプを使用することです。)
- 高可用性。プライマリノード障害時に、DAX は自動的にリードレプリカにフェイルオーバーし、新しいプライマリとして指定します。レプリカノードに障害が発生しても、障害が発生したノードが回復するまで、DAX クラスター内の他のノードがリクエストを処理することができます。耐障害性を最大にするため、リードレプリカを異なるアベイラビリティーゾーンにデプロイする必要があります。この設定により、アベイラビリティーゾーン全体が使用できない場合でも DAX クラスターが機能し続けることができます。

> **Important**
>
> 本稼働環境での使用においては、少なくとも 3 つのノードをそれぞれ異なるアベイラビリティーゾーンに置いて DAX を使用することを強くお勧めします。DAX クラスターが耐障害性を持つためには 3 つのノードが必要です。

DAX クラスターは、開発またはテストワークロードでは 1 つまたは 2 つのノードでデプロイできます。1 つのノードまたは 2 つのノードクラスターでは耐障害性がないため、本稼働での使用では 3 つ未満のノードはお勧めしません。1 つのノードまたは 2 つのノードクラスターでソフトウェアまたはハードウェアの障害が発生した場合、クラスターが使用できなくなったり、キャッシュ済みデータが失われることがあります。

Important

DAX クラスターは、最大 500 の DynamoDB テーブルをサポートします。500 テーブルを超えると、クラスターの可用性とパフォーマンスが低下する可能性があります。

リージョンとアベイラビリティーゾーン

ある AWS リージョンにある DAX クラスターは、同じリージョンにある DynamoDB テーブルとのみやり取りできます。したがって、適切なリージョンで DAX クラスターを起動することを確認してください。他のリージョンに DynamoDB テーブルがある場合は、そのリージョンにも DAX クラスターを起動する必要があります。

各リージョンは、他のリージョンと完全に分離されるように設計されています。各リージョンには複数のアベイラビリティーゾーンがあります。別のアベイラビリティーゾーンでノードを起動して、最大限の耐障害性を実現できます。

Important

単一アベイラビリティーゾーンにクラスターのすべてのノードを置かないでください。この設定では、アベイラビリティーゾーンの障害時に DAX クラスターが利用できなくなります。
本稼働環境での使用においては、少なくとも 3 つのノードをそれぞれ異なるアベイラビリティーゾーンに置いて DAX を使用することを強くお勧めします。DAX クラスターが耐障害性を持つためには 3 つのノードが必要です。
DAX クラスターは、開発またはテストワークロードでは 1 つまたは 2 つのノードでデプロイできます。1 つまたは 2 つのノードクラスターでは耐障害性がないため、本稼働での使用では 3 つ未満のノードはお勧めしません。1 つまたは 2 つのノードクラスターでソフトウェアまたはハードウェアの障害が発生した場合、クラスターが使用できなくなったり、キャッシュ済みデータが失われることがあります。

パラメータグループ

パラメータグループは、DAX クラスターのランタイム設定を管理するために使用されます。DAX には、パフォーマンスを最適化するために使用できるいくつかのパラメータ (キャッシュ済みデータの TTL ポリシーを定義するなど) があります。パラメータグループはクラスターに適用可能なパラメータの名前付きセットです。これにより、そのクラスター内のすべてのノードがまったく同じ方法で設定されていることを確認できます。

セキュリティグループ

DAX クラスターは、Amazon Virtual Private Cloud (Amazon VPC) 環境で実行されます。この環境は、AWS アカウント専用の仮想ネットワークであり、他の VPC からは独立しています。セキュリティグループは、VPC の仮想ファイアウォールとして機能し、インバウンドトラフィックとアウトバウンドネットワークトラフィックをコントロールできます。

VPC でクラスターを起動する際に、着信ネットワークトラフィックを許可する進入ルールをセキュリティグループに追加します。進入ルールは、クラスターのプロトコル (TCP) とポート番号 (8111) を指定します。このルールをセキュリティグループに追加すると、VPC 内で実行されるアプリケーションが DAX クラスターにアクセスできます。

クラスター ARN

各 DAX クラスターには Amazon リソースネーム (ARN) が割り当てられます。ARN 形式は次のとおりです。

```
arn:aws:dax:region:accountID:cache/clusterName
```

IAM ポリシー内でクラスター ARN を使用して、DAX API オペレーションのアクセス許可を定義します。詳細については、「DAX のアクセスコントロール」を参照してください。

クラスターエンドポイント

すべての DAX クラスターは、アプリケーションで使用できるクラスターエンドポイントを提供します。エンドポイントを使用してクラスターにアクセスすることで、アプリケーションでクラスターの個別のノードのホスト名とポート番号を把握する必要がなくなります。アプリケーションは、リードレプリカを追加または削除した場合でも、自動的にクラスターのすべてのノードを「把握」します。

転送中に暗号化を使用するように設定されていない、us-east-1 リージョンのクラスターエンドポイントの例を以下に示します。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

転送中に暗号化を使用するように設定されている、同じリージョン内のクラスターエンドポイントの例を以下に示します。

```
daxs://my-encrypted-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

ノードエンドポイント

DAX クラスターのノードそれぞれに独自のホスト名とポート番号があります。ノードエンドポイントの例を以下に示します。

```
myDAXcluster-a.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com:8111
```

アプリケーションは、エンドポイントを使用してノードに直接アクセスできます。しかし、DAX クラスターを単一のユニットとして扱い、代わりにクラスターのエンドポイントを使用してアクセスすることをお勧めします。クラスターエンドポイントを使用することで、アプリケーションでノードのリストを保持し、クラスターでノードが追加または削除されたときにリストを最新の状態に保つ必要がなくなります。

サブネットグループ

DAX クラスターノードへのアクセスは、Amazon VPC 環境内の Amazon EC2 インスタンスで実行されるアプリケーションのみに制限されます。サブネットグループを使用して、特定のサブネットで実行されている Amazon EC2 インスタンスからのクラスターアクセスを許可できます。サブネットグループは、Amazon VPC 環境で実行しているクラスターに対して指定できるサブネット (通常はプライベート) の集合です。

DAX クラスターを作成するときに、サブネットグループを指定する必要があります。DAX はそのキャッシュサブネットグループを使用して、そのサブネット内でノードに関連付けるサブネットおよび IP アドレスを選択します。

Events

DAX は、ノードの追加の失敗、ノードの追加の成功、セキュリティグループの変更など、重要なイベントをクラスター内に記録します。主要イベントをモニタリングすることで、クラスターの現在の状態を知り、イベントに基づいて是正措置を取ることができます。AWS マネジメントコンソール、または DAX 管理 API の `DescribeEvents` アクションを使用して、これらのイベントにアクセスできます。

特定の Amazon Simple Notification Service (Amazon SNS) トピックに通知を送信するようにリクエストすることもできます。そうすれば、DAX クラスターでイベントが発生したことがすぐにわかります。

メンテナンスウィンドウ

システムの変更を適用するため、すべてのクラスターには週ごとのメンテナンスウィンドウがあります。変更が順次適用されるにしたがって、既存のノードが置き換えられ、変更が適用された新しいノードがクラスターに追加されます。この間、アプリケーションでは、一時的なエラーやスロットリングが発生する可能性があります。そのため、使用量が最も少ない時間にメンテナンスウィンドウを設定し、必要に応じてこのスケジュールを定期的に調整することをお勧めします。お客様がリクエストしたメンテナンス作業が発生する時間範囲を最大 24 時間で指定できます。

キャッシュクラスターの作成または変更時に、必要なメンテナンス期間を指定しない場合、DAX では、ランダムに選択された曜日に対して 60 分のメンテナンスウィンドウが割り当てられます。この 60 分間のメンテナンスウィンドウは、各 AWS リージョン の 8 時間の時間ブロックからランダムに選択されます。次の表に、デフォルトでメンテナンス時間が割り当てられる各リージョンの時間ブロックを示します。

リージョンコード	リージョン名	メンテナンスウィンドウ
ap-northeast-1	Asia Pacific (Tokyo) Region	13:00 ~ 21:00 UTC
ap-southeast-1	Asia Pacific (Singapore) Region	14:00 ~ 22:00 UTC
ap-southeast-2	Asia Pacific (Sydney) Region	12:00 ~ 20:00 UTC
ap-south-1	Asia Pacific (Mumbai) Region	17:30 ~ 1:30 UTC
cn-northwest-1	中国 (寧夏) リージョン	23:00 ~ 07:00 UTC
cn-north-1	中国 (北京) リージョン	14:00 ~ 22:00 UTC
eu-central-1	Europe (Frankfurt) Region	23:00 ~ 07:00 UTC
eu-north-1	欧州 (ストックホルム) リージョン	01:00 ~ 09:00 UTC
eu-south-2	欧州 (スペイン) リージョン	21:00 ~ 05:00 UTC

リージョンコード	リージョン名	メンテナンスウィンドウ
eu-west-1	欧州 (アイルランド) リージョン	22:00 ~ 06:00 UTC
eu-west-2	Europe (London) Region	23:00 ~ 07:00 UTC
eu-west-3	欧州 (パリ) リージョン	23:00 ~ 07:00 UTC
sa-east-1	South America (São Paulo) Region	01:00 ~ 09:00 UTC
us-east-1	米国東部 (バージニア北部) リージョン	03:00 ~ 11:00 UTC
us-east-2	米国東部 (オハイオ) リージョン	23:00 ~ 07:00 UTC
us-west-1	米国西部 (北カリフォルニア) リージョン	06:00 ~ 14:00 UTC
us-west-2	米国西部 (オレゴン) リージョン	06:00 ~ 14:00 UTC

DAX クラスターの作成

このセクションでは、デフォルトの Amazon Virtual Private Cloud (Amazon VPC) 環境で Amazon DynamoDB Accelerator (DAX) を最初にセットアップし使用する手順を順を追って説明します。最初の DAX クラスターは、AWS Command Line Interface (AWS CLI) または AWS マネジメントコンソール のいずれかを使用して作成できます。

DAX クラスターを作成すると、同じ VPC で実行されている Amazon EC2 インスタンスからアクセスできるようになります。その後、アプリケーションプログラムで DAX クラスターを使用できるようになります。詳細については、「DynamoDB Accelerator (DAX) クライアントで開発する」を参照してください。

トピック

- DynamoDB にアクセスする DAX 用の IAM サービスロールを作成します

- DAX と IPv6
- AWS CLI を使用した DAX クラスターの作成
- AWS マネジメントコンソール を使用した DAX クラスターの作成

DynamoDB にアクセスする DAX 用の IAM サービスロールを作成します

DAX クラスターがユーザーに代わって DynamoDB テーブルにアクセスできるようにサービスロールを作成する必要があります。サービスロールは、ユーザーに代わって AWS のサービスを承認する AWS Identity and Access Management (IAM) ロールです。サービスロールは、ユーザーがテーブルそのものにアクセスしているかのように、DAX に DynamoDB テーブルへのアクセスを許可します。DAX クラスターを作成する前にサービスロールを作成する必要があります。

コンソールを使用している場合は、クラスターを作成するワークフローで DAX サービスロールの有無が確認されます。何も見つからない場合は、コンソールは新しいサービスロールを作成します。詳細については、「」を参照してくださいthe section called “ステップ 2: DAX クラスターを作成”

AWS CLI を使用している場合は、すでに作成してある DAX サービスロールを指定するか、事前に新しいサービスロールを作成する必要があります。詳細については、「」を参照してくださいステップ 1: AWS CLI を使用して、DAX から DynamoDB にアクセスするための IAM サービスロールを作成する

サービスロールの作成に必要なアクセス許可

AWS マネージド `AdministratorAccess` ポリシーには、DAX クラスターおよびサービスロールの作成に必要なすべてのアクセス許可が含まれています。ユーザーに `AdministratorAccess` がアタッチされている場合、それ以上のアクションは不要です。

アタッチされていない場合は、IAM ポリシーに次のアクセス許可を追加して、ユーザーがサービスロールを作成できるようにする必要があります。

- `iam:CreateRole`
- `iam:CreatePolicy`
- `iam:AttachRolePolicy`
- `iam:PassRole`

アクションの実行を試みるユーザーにこれらのアクセス許可をアタッチする必要があります。

> **Note**
>
> iam:CreateRole、iam:CreatePolicy、iam:AttachRolePolicy、および iam:PassRole の各アクセス許可は、DynamoDB の AWS マネージドポリシーには含まれていません。これは意図的なものです。これらのアクセス許可が含まれていると、特権のエスカレーションが可能になり、ユーザーがこれらのアクセス権限を使用して新しい管理者ポリシーを作成し、それを既存のロールにアタッチできるようになるからです。そのため、DAX クラスターの管理者は、これらのアクセス権限を自分のポリシーに明示的に追加する必要があります。

トラブルシューティング

ユーザーポリシーに iam:CreateRole、iam:CreatePolicy、iam:AttachPolicy の各アクセス許可が含まれていないと、エラーメッセージが表示されます。次の表に、これらのメッセージを示し、問題を解決する方法を説明します。

次のエラーメッセージが表示された場合は ...。	次の作業を行います。
User: arn:aws:iam:: *accountID* :user/*userName* is not authorized to perform: iam:CreateRole on resource: arn:aws:iam:: *accountID* :role/service-role/ *roleName*	iam:CreateRole をユーザーポリシーに追加します。
User: arn:aws:iam:: *accountID* :user/*userName* is not authorized to perform: iam:CreatePolicy on resource: policy *policyName*	iam:CreatePolicy をユーザーポリシーに追加します。
User: arn:aws:iam:: *accountID* :user/*userName* is not authorized to perform: iam:AttachRolePolicy on resource: role *daxServiceRole*	iam:AttachRolePolicy をユーザーポリシーに追加します。

DAX クラスターの管理に必要な IAM ポリシーの詳細については、「DAX のアクセスコントロール」を参照してください。

DAX と IPv6

DynamoDB DAX で IPv6 アドレス指定がサポートされるようになりました。これにより、IPv4 のみ、IPv6 のみ、またはデュアルスタックのネットワークモードで動作するクラスターを作成できます。進化するインフラストラクチャ要件を満たすネットワーク機能を強化するのに役立ちます。

ネットワークの種類:

クラスターの作成には、次のいずれかの種類のネットワークが使用できます。

- IPv4 のみ
- IPv6 のみ
- デュアルスタック (IPv4 と IPv6 の両方をサポート)

の主な特徴

IPv6 のサポートを使用して、次のことを実行できます。

- ネットワーク設定のオプション:
  - IPv4 のみ、および `dual_stack subnets` の デュアルスタッククラスター。
  - IPv6 専用サブネットの IPv6 専用クラスター。
- サブネットグループの管理:
  - IPv4 のみ、IPv6 のみ、またはデュアルスタックをサポートする新しいサブネットグループを作成する。
  - 追加の VPC サブネットを使用して既存のサブネットグループを変更する
  - IPv6 専用サブネットを、IPv6 用に設定されたサブネットグループに追加する
  - IPv4 およびデュアルスタック用に設定されたサブネットグループに IPv4 またはデュアルスタックサブネットを追加する
- クライアント設定:
  - データプレーンの呼び出しを行うとき、次を使用してデュアルスタッククラスターの優先 IP プロトコルを設定できます。
    - Python SDK の `ip_discovery` パラメータ
    - 他の SDK の `ipDiscovery` パラメータ
  - デフォルト: プロトコル設定が指定されていない場合、IPv4

DAX クラスターに IPv6 を実装する前に、次の点を考慮する必要があります。

- クラスター作成後にネットワークタイプを変更することはできません。
- デュアルスタッククラスターの場合、クライアント設定の `ip_discovery/ipDiscovery` パラメータによって、使用する IP プロトコル (IPv4 または IPv6) が決まります。
- 異なるアプリケーションは、設定に基づいて異なる IP プロトコルを使用して同じデュアルスタッククラスターに接続できます。

Exampleクライアント設定の例

```
DynamoDbAsyncClient client = ClusterDaxAsyncClient.builder()
        .overrideConfiguration(Configuration.builder()
            .url(endpoint)             // DAX cluster endpoint
            .ipDiscovery(ipDiscovery)       // IP discovery type (IPv4 or IPv6)
            .build())
        .build();
```

Important

リソースベースの IAM ポリシーを使用して DAX を使用する IPv6 専用の環境にある DynamoDB テーブルの IP アドレスを制限しているとき、IPv4 アドレス空間 (`0.0.0.0/0`) をブロックする場合は、DAX クラスターの IAM ロールに例外を作成する必要があります。他のアクセスパスの IP ベースの制限を維持しながら、DAX クラスターの IAM ロールへのアクセスを特に許可するポリシーに、`ArnNotEquals` 条件を追加します。この例外がない限り、DAX は DynamoDB テーブルにアクセスできません。
例：

Example

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "dynamodb:PutItem",
```

```
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection",
      "Condition": {
        "ArnNotEquals": {
          "aws:PrincipalArn": "arn:aws:iam::123456789012:role/
DAXServiceRoleForDynamoDBAccess"
        },
        "IpAddress": {
          "aws:SourceIp": "0.0.0.0/0"
        }
      }
    }
  ]
}
```

AWS CLI を使用した DAX クラスターの作成

このセクションでは、AWS Command Line Interface (AWS CLI) を使用して Amazon DynamoDB Accelerator (DAX) クラスターを作成する方法を説明します。まだ AWS CLI をインストールして設定していない場合は、インストールして設定する必要があります。これを行うには、AWS Command Line Interface ユーザーガイドの手順を参照してください。

- のインストールAWS CLI
- の設定AWS CLI

> **Important**
>
> AWS CLI を使用して DAX クラスターを管理するには、バージョン 1.11.110 以上をインストールするか、アップグレードしてください。

AWS CLI のすべての例で、`us-west-2` リージョンと架空のアカウント ID を使用しています。

トピック

- ステップ 1: AWS CLI を使用して、DAX から DynamoDB にアクセスするための IAM サービスロールを作成する
- ステップ 2: サブネットグループの作成

- ステップ 3: AWS CLI を使用して DAX クラスターを作成
- ステップ 4: AWS CLI を使用してセキュリティグループのインバウンドルールを設定

ステップ 1: AWS CLI を使用して、DAX から DynamoDB にアクセスするための IAM サービスロールを作成する

Amazon DynamoDB Accelerator (DAX) クラスターを作成する前に、クラスター用のサービスロールを作成する必要があります。サービスロールは、ユーザーに代わって AWS のサービスを承認する AWS Identity and Access Management (IAM) ロールです。サービスロールは、ユーザーがテーブルそのものにアクセスしているかのように、DAX に DynamoDB テーブルへのアクセスを許可します。

このステップでは、IAM ポリシーを作成し、次にそのポリシーを IAM ロールにアタッチします。こうすることで、DAX クラスターにロールを割り当て、ユーザーの代わりに DynamoDB オペレーションを実行させることができます。

DAX の IAM サービスロールを作成するには

1. 次の内容で、`service-trust-relationship.json` というファイルを作成します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
       {
            "Effect": "Allow",
            "Principal": {
                "Service": "dax.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

2. サービスロールを作成します。

```
aws iam create-role \
    --role-name DAXServiceRoleForDynamoDBAccess \
    --assume-role-policy-document file://service-trust-relationship.json
```

3. 次の内容で、service-role-policy.json というファイルを作成します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:111122223333:*"
            ]
        }
    ]
}
```

   accountID をご自分の AWS アカウント ID に置き換えます。AWS アカウント ID を見つけるには、コンソールの右上隅で、ログイン ID を選択します。ドロップダウンメニューに AWS アカウント ID が表示されます

   この例の Amazon リソースネーム (ARN) では、*accountID* は 12 桁の数字である必要があります。ハイフンなどの区切り文字を使用しないでください。

4. サービスロールの IAM ポリシーを作成します。

```
aws iam create-policy \
    --policy-name DAXServicePolicyForDynamoDBAccess \
    --policy-document file://service-role-policy.json
```

出力では、以下の例のように、作成したポリシーの ARN に注意してください。

```
arn:aws:iam::123456789012:policy/DAXServicePolicyForDynamoDBAccess
```

5. ポリシーをサービスロールにアタッチします。以下のコードにある *arn* を、前のステップで書き留めた実際のロール ARN に置き換えます。

```
aws iam attach-role-policy \
    --role-name DAXServiceRoleForDynamoDBAccess \
    --policy-arn arn
```

次に、デフォルト VPC のサブネットグループを指定します。サブネットグループは、VPC 内の 1 つ以上のサブネットのコレクションです。「」を参照してくださいステップ 2: サブネットグループの作成

ステップ 2: サブネットグループの作成

この手順に従って、AWS Command Line Interface (AWS CLI) を使用して Amazon DynamoDB Accelerator (DAX) クラスターのサブネットグループを作成します。

> **Note**
>
> デフォルト VPC のサブネットグループを作成済みである場合は、このステップを省略できます。

DAX は、Amazon Virtual Private Cloud (Amazon VPC) 環境内で実行するように設計されています。2013 年 12 月 4 日以降に AWS アカウントを作成した場合は、各 AWS リージョンにデフォルトで VPC が用意されています。詳細については、「Amazon VPC ユーザーガイド」の「デフォルト VPC とデフォルトサブネット」を参照してください。

サブネットグループを作成するには

1. デフォルト VPC の ID を確認するには、次のコマンドを入力します。

```
aws ec2 describe-vpcs
```

出力では、以下の例のように、デフォルト VPC の識別子に注意してください。

vpc-12345678

2. デフォルト VPC に関連付けられているサブネットの ID を確認します。*vpcID* を実際の VPC ID に置き換えます (vpc-12345678 など)。

```
aws ec2 describe-subnets \
    --filters "Name=vpc-id,Values=vpcID" \
    --query "Subnets[*].SubnetId"
```

出力で、サブネット ID を記録します (subnet-11111111 など)。

3. サブネットグループを作成します。--subnet-ids パラメータに少なくとも 1 つのサブネット ID を指定してください。

```
aws dax create-subnet-group \
    --subnet-group-name my-subnet-group \
    --subnet-ids subnet-11111111 subnet-22222222 subnet-33333333 subnet-44444444
```

クラスターを作成するには、「ステップ 3: AWS CLI を使用して DAX クラスターを作成」を参照してください。

ステップ 3: AWS CLI を使用して DAX クラスターを作成

次の手順に従って、AWS Command Line Interface (AWS CLI) を使用して、デフォルト Amazon VPC に Amazon DynamoDB Accelerator (DAX) クラスターを作成します。

DAX クラスターを作成するには

1. サービスロールの Amazon リソースネーム (ARN) を取得します。

```
aws iam get-role \
    --role-name DAXServiceRoleForDynamoDBAccess \
    --query "Role.Arn" --output text
```

出力では、以下の例のように、サービスロール ARN に注意してください。

```
arn:aws:iam::123456789012:role/DAXServiceRoleForDynamoDBAccess
```

2. DAX クラスターを作成します。*roleARN* を、前のステップで書き留めた ARN に置き換えます。

```
aws dax create-cluster \
    --cluster-name mydaxcluster \
    --node-type dax.r4.large \
    --replication-factor 3 \
    --iam-role-arn roleARN \
    --subnet-group my-subnet-group \
    --sse-specification Enabled=true \
    --region us-west-2
```

クラスター内のすべてのノードは、タイプ `dax.r4.large` (`--node-type`) です。3つのノード (`--replication-factor`) があります (1 つのプライマリノードと 2 つのレプリカ)。

クラスターステータスを表示するには、次のコマンドを入力します。

```
aws dax describe-clusters
```

ステータスは出力に表示されます (`"Status": "creating"` など)。

> **Note**
>
> クラスターを作成するには数分かかります。クラスターが使用可能になると、ステータスは `available` に変わります。その間に、「ステップ 4: AWS CLI を使用してセキュリティグループのインバウンドルールを設定」に進んでその指示に従います。

ステップ 4: AWS CLI を使用してセキュリティグループのインバウンドルールを設定

Amazon DynamoDB Accelerator (DAX) クラスター内のノードは、Amazon VPC のデフォルトのセキュリティグループを使用します。デフォルトのセキュリティグループでは、暗号化されていないクラスターには TCP ポート 8111、暗号化されたクラスターには 9111 のインバウンドトラフィックを許可する必要があります。これで Amazon VPC の Amazon EC2 インスタンスは DAX クラスターにアクセスできます。

> **Note**
>
> 異なるセキュリティグループ (`default` 以外) で DAX クラスターを起動した場合、グループに対してこの手順を代わりに実行する必要があります。

セキュリティグループのインバウンドルールを設定するには

1. デフォルトセキュリティグループの ID を確認するには、次のコマンドを入力します。*vpcID* を実際の VPC ID (「ステップ 2: サブネットグループの作成」を参照) に置き換えます。

```
aws ec2 describe-security-groups \
    --filters Name=vpc-id,Values=vpcID Name=group-name,Values=default \
    --query "SecurityGroups[*].{GroupName:GroupName,GroupId:GroupId}"
```

   出力で、セキュリティグループ ID を記録します (sg-01234567 など)。

2. 次のように入力します。*sgID* を実際のセキュリティグループ ID に置き換えます。暗号化されていないクラスターの場合はポート 8111 を、暗号化されたクラスターの場合は 9111 を使用します。

```
aws ec2 authorize-security-group-ingress \
    --group-id sgID --protocol tcp --port 8111
```

AWS マネジメントコンソール を使用した DAX クラスターの作成

このセクションでは、AWS マネジメントコンソールを使用して Amazon DynamoDB Accelerator (DAX) クラスターを作成する方法を説明します。

トピック

- ステップ 1: AWS マネジメントコンソール を使用してサブネットグループを作成
- ステップ 2: AWS マネジメントコンソール を使用して DAX クラスターを作成
- ステップ 3: AWS マネジメントコンソール を使用してセキュリティグループのインバウンドルールを設定

ステップ 1: AWS マネジメントコンソール を使用してサブネットグループを作成

この手順に従って、AWS マネジメントコンソール を使用して Amazon DynamoDB Accelerator (DAX) クラスターのサブネットグループを作成します。

> **Note**
>
> デフォルト VPC のサブネットグループを作成済みである場合は、このステップを省略できます。

DAX は、Amazon Virtual Private Cloud (Amazon VPC) 環境内で実行するように設計されています。2013 年 12 月 4 日以降に AWS アカウントを作成した場合は、各 AWS リージョンにデフォルトで VPC が用意されています。詳細については、「Amazon VPC ユーザーガイド」の「デフォルト VPC とデフォルトサブネット」を参照してください。

DAX クラスターの作成プロセスの一部として、サブネットグループを指定する必要があります。サブネットグループは、VPC 内の 1 つ以上のサブネットのコレクションです。DAX クラスターを作成する際、ノードがサブネットグループ内のサブネットにデプロイされます。

> **Note**
>
> この DAX クラスターを持つ VPC には、他のリソースや、他のサービスの VPC エンドポイント (ElastiCache の VPC エンドポイントを除く) さえも含まれる場合があり、DAX クラスターオペレーションでエラーが発生する可能性があります。

サブネットグループを作成するには

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインの、[DAX] で、[サブネットグループ]を選択します。
3. [サブネットグループの作成] を選択します。
4. [サブネットグループの作成] ウィンドウで以下を行います。
   a. [Name] (名前) - サブネットグループの短い名前を入力します。
   b. [Description] (説明) - サブネットグループの説明を入力します。
   c. VPC ID - Amazon の VPC 環境の ID を選択します。
   d. [Subnets] (サブネット) - リストから 1 つ以上のサブネットを選択します。

      > **Note**
      >
      > サブネットは、複数のアベイラビリティーゾーンに分散されます。マルチノード DAX クラスター (プライマリノードと 1 つ以上のリードレプリカ) の作成を計画す

る場合は、複数のサブネット ID を選択することをお勧めします。その後、DAX はクラスターノードを複数のアベイラビリティーゾーンにデプロイできます。アベイラビリティーゾーンが使用できなくなると、DAX は自動的に残っているアベイラビリティーゾーンにフェイルオーバーします。DAX クラスターは中断することなく機能し続けます。

設定が正しいことを確認したら、[サブネットグループの作成] を選択します。

クラスターを作成するには、「ステップ 2: AWS マネジメントコンソール を使用して DAX クラスターを作成」を参照してください。

ステップ 2: AWS マネジメントコンソール を使用して DAX クラスターを作成

次の手順に従って、デフォルト Amazon VPC に Amazon DynamoDB Accelerator (DAX) クラスターを作成します。

DAX クラスターを作成するには

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインの、[DAX] で、[クラスター]を選択します。
3. [Create cluster (クラスターの作成)] を選択します。
4. [クラスターの作成] ウィンドウで、以下の操作を行います。
   a. [クラスター名] - DAX クラスターの短縮名を入力します。
   b. [Cluster description] (クラスターの説明) - クラスターの説明を入力します。
   c. ノードタイプ - クラスター内のすべてのノードのノードタイプを選択します。
   d. [Cluster size] (クラスターサイズ) - クラスター内のノードの数を選択します。クラスターは 1 つのプライマリノードと最大 9 つのリードレプリカで構成されます。

> **Note**
>
> シングルノードクラスターを作成する場合は、[1] を選択します。クラスターは 1 つのプライマリノードで構成されます。
> マルチノードクラスターを作成する場合は、[3] (プライマリノード 1 つとリードレプリカ 2 つ) ~ [10] (プライマリノード 1 つとリードレプリカ 9 つ) の数字を選択します。

> **Important**
>
> 本稼働環境での使用においては、少なくとも 3 つのノードをそれぞれ異なるアベイラビリティーゾーンに置いて DAX を使用することを強くお勧めします。DAX クラスターが耐障害性を持つためには 3 つのノードが必要です。
>
> DAX クラスターは、開発またはテストワークロードでは 1 つまたは 2 つのノードでデプロイできます。1 つまたは 2 つのノードクラスターでは耐障害性がないため、本稼働での使用では 3 つ未満のノードはお勧めしません。1 つまたは 2 つのノードクラスターでソフトウェアまたはハードウェアの障害が発生した場合、クラスターが使用できなくなったり、キャッシュ済みデータが失われることがあります。

e. [次へ] を選択します。

f. サブネットグループ - [既存のものを選択] を選択し、ステップ 1: AWS マネジメントコンソール を使用してサブネットグループを作成 で作成したサブネットグループを選択します。

g. アクセスコントロール - デフォルトのセキュリティグループを選択します。

h. アベイラビリティーゾーン (AZ) - [自動] を選択します。

i. [次へ] を選択します。

j. [IAM service role for DynamoDB access] (DynamoDB アクセス用の IAM サービスロール) - [Create new] (新規作成) を選択し、次の情報を入力します。

- IAM ロール名 - IAM ロールの名前 (`DAXServiceRole` など) を入力します。コンソールは新しい IAM ロールを作成し、DAX クラスターは実行時にこのロールを割り当てます。
- [ポリシーの作成] の横にあるボックスを選択します。
- [IAM role policy] (IAM ロールポリシー) - [Read/Write] (読み込み/書き込み) を選択します。これにより、DAX クラスターが DynamoDB で読み込み/書き込み操作を実行できるようになります。
- 新しい IAM ポリシー名 — IAM ロール名を入力すると、このフィールドが入力されます。IAM ポリシーの名前も取得できます (`DAXServicePolicy` など)。コンソールは新しい IAM ポリシーを作成し、ポリシーを IAM ロールにアタッチします。
- DynamoDB テーブルへのアクセス — [すべてのテーブル] を選択します。

k. 暗号化 - [保管時の暗号化を有効にする] および [転送時の暗号化を有効にする] を選択します。詳細については、保管時の DAX 暗号化および転送時の DAX 暗号化を参照してください。

DAX が Amazon EC2 にアクセスするための別のサービスロールも必要です。このサービスロールは DAX によって自動的に作成されます。詳細については、「DAX のサービスにリンクされたロールの使用」を参照してください。

5. すべての設定が正しいことを確認したら、[次へ] を選択します。
6. パラメータグループ — [既存のものを選択] を選択します。
7. メンテナンスウィンドウ — ソフトウェアのアップグレードが適用されるタイミングに指定がない場合は、[指定なし] を選択するか、[時間枠を指定] を選択し、[平日]、[時間 (UTC)]、[開始時間 (時間)] のオプションを指定して、メンテナンスウィンドウのスケジュールを設定します。
8. タグ: — [新しいタグを追加] を選択して、タグ付けするキーと値のペアを入力します。
9. [次へ] を選択します。

[確認と作成] 画面では、すべての設定を確認できます。クラスターを作成する準備ができたら、[クラスターの作成] を選択します。

[クラスター] 画面で、DAX クラスターが [作成中] というステータスで一覧表示されます。

> **Note**
>
> クラスターを作成するには数分かかります。クラスターが使用可能になると、ステータスは Available に変わります。
> その間に、「ステップ 3: AWS マネジメントコンソール を使用してセキュリティグループのインバウンドルールを設定」に進んでその指示に従います。

ステップ 3: AWS マネジメントコンソール を使用してセキュリティグループのインバウンドルールを設定

Amazon DynamoDB Accelerator (DAX) クラスターは通信に TCP ポート 8111 (暗号化されていないクラスター) または 9111 (暗号化されたクラスター) を使用するため、このポートのインバウンドトラフィックを認可する必要があります。これで Amazon VPC の Amazon EC2 インスタンスは DAX クラスターにアクセスできます。

> **Note**
>
> 異なるセキュリティグループ (`default` 以外) で DAX クラスターを起動した場合、グループに対してこの手順を代わりに実行する必要があります。

セキュリティグループのインバウンドルールを設定するには

1. Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. ナビゲーションペインで [Security Groups] を選択します。
3. [デフォルト] セキュリティグループを選択します。[アクション] メニューで、[インバウンドルールの編集] を選択します。
4. [ルールの追加] を選択し、次の情報を入力します。
   - [ポート範囲] - 8111 (クラスターが暗号化されていない場合) または 9111 (クラスターが暗号化されている場合)を入力します。
   - ソース — [カスタム] のままにして、右側の検索フィールドを選択します。ドロップダウンメニューが表示されます。デフォルトのセキュリティグループの識別子を選択します。
5. [Save rules] (ルールの保存) を選択して、変更を保存します。
6. コンソールの名前を更新するには、[名前] プロパティに移動して、表示される [編集] オプションを選択します。

DAX および DynamoDB の整合性モデル

Amazon DynamoDB Accelerator (DAX) は、DynamoDB テーブルにキャッシュを追加するプロセスを簡素化するために設計された書き込みスルーキャッシュサービスです。DAX は DynamoDB とは別に動作するため、アプリケーションが意図どおりに動作することを確認するには DAX と DynamoDB の両方の整合性モデルを理解することが重要です。

多くのユースケースでは、アプリケーションでの DAX の使用方法が、DAX クラスター内のデータの整合性、および DAX と DynamoDB 間のデータの整合性に影響します。

トピック

- DAX クラスターノード間の整合性
- DAX 項目キャッシュの動作
- DAX クエリキャッシュの動作

- 強力な整合性のあるトランザクション読み込み
- ネガティブキャッシング
- 書き込みの方法

DAX クラスターノード間の整合性

アプリケーションの高可用性を実現するには、DAX クラスターのプロビジョニングには少なくとも 3 つのノードを使用してください。次にこれらのノードをリージョン内の複数のアベイラビリティーゾーンに配置します。

クラスターの実行中、データがクラスター内のすべてのノードにレプリケートされます (複数のノードをプロビジョニング済みの場合)。DAX を使用して `UpdateItem` を正常に実行するアプリケーションの場合を考えてみましょう。このアクションにより、プライマリノードの項目キャッシュが新しい値で変更されます。この値は、クラスター内の他のすべてのノードにレプリケートされます。このレプリケーションの結果、整合性が保たれます。完了までにかかる時間は通常は 1 秒未満です。

このシナリオでは、2 つのクライアントが同じ DAX クラスターから同じキーを読み取った時に、それぞれのクライアントがアクセスしたノードによって、異なる値を受け取る可能性があります。更新がクラスター内のすべてのノードで完全にレプリケートされると、ノードはすべて整合性がとれます。(この動作は、DynamoDB の結果整合性特性と似ています。)

DAX を使用するアプリケーションを構築する場合、そのアプリケーションは結果整合性データを許容するように設計されている必要があります。

DAX 項目キャッシュの動作

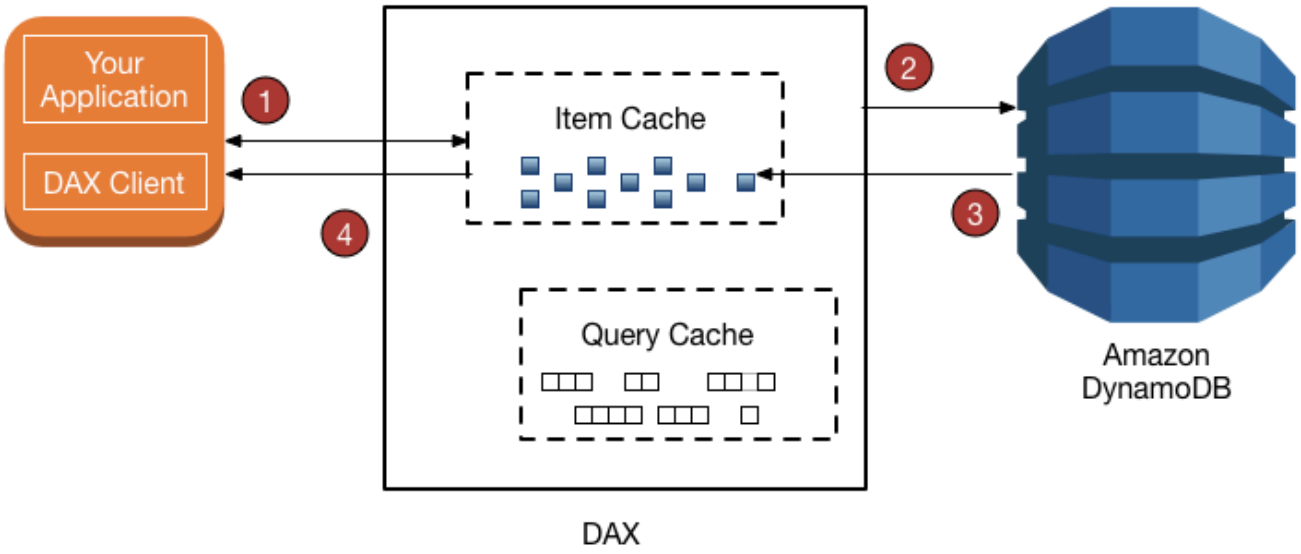
各 DAX クラスターには、2 つの異なるキャッシュ (項目キャッシュとクエリキャッシュ) があります。詳細については、「DAX: 仕組み」を参照してください。

このセクションでは、DAX 項目キャッシュに対する読み込みおよび書き込みの整合性の実装について取り扱います。

読み込みの整合性

DynamoDB では、`GetItem` オペレーションはデフォルトで結果整合性のある読み込みを行います。DynamoDB クライアントで `UpdateItem` を使用するとします。すぐ後に同じ項目を読み取ろうとすると、更新前のデータが表示される場合があります。これは、DynamoDB ストレージロケーション全体への伝播が遅れるためです。整合性は通常、数秒以内に達成します。そのため、読み込みを再試行すると、更新された項目が表示されます。

DAX クライアントで GetItem を使用する場合、オペレーション (この場合は結果整合性のある読み込み) は次のように進行します。



1. DAX クライアントが GetItem リクエストを発行します。DAX は、項目キャッシュからリクエストされた項目を読み取ろうとします。項目がキャッシュ内にある場合 (キャッシュヒット)、DAX によりアプリケーションに返されます。
2. 項目がない場合 (キャッシュミス)、DAX は DynamoDB に対して結果整合性 GetItem オペレーションを実行します。
3. DynamoDB からリクエストされた項目が返され、DAX で項目キャッシュに保存されます。
4. DAX はその項目をアプリケーションに返します。
5. (非表示) DAX クラスターに複数のノードがある場合、項目がクラスター内の他のすべてのノードにレプリケートされます。

項目は、キャッシュの有効期限 (TTL) 設定および最も長い間使用されていない (LRU) アルゴリズムによりますが、DAX 項目キャッシュに残ります。詳細については、「DAX: 仕組み」を参照してください。

ただし、この期間は、DAX は項目を DynamoDB から再ロードしません。他のユーザーが DynamoDB クライアントを使用して、DAX 全体をバイパスして項目を更新すると、DAX クライアントを使用した GetItem リクエストで、DynamoDB クライアントを使用した GetItem リクエスト

とは異なる結果が呼び出されます。このシナリオでは、DAX と DynamoDB は、DAX 項目の TTL が期限切れになるまで、同じキーに対して矛盾する値を保持します。

アプリケーションが をバイパスして基礎となる テーブルのデータを変更する場合は、そのアプリケーションはデータの不整合が発生する可能性を予期し許容する必要があります。

> **Note**
>
> GetItem に加えて、DAX クライアントは BatchGetItem リクエストもサポートします。BatchGetItem は基本的には 1 つ以上の GetItem リクエストをまとめるラッパーです。そのため、DAX ではそれぞれを個別の GetItem オペレーションとして扱います。

書き込みの整合性

DAX は書き込みスルーキャッシュであり、DAX の項目キャッシュが基礎となる DynamoDB テーブルに対して整合性を保つためのプロセスを簡易化します。

DAX クライアントは、DynamoDB と同じ書き込み API オペレーションをサポートします (PutItem、UpdateItem、DeleteItem、BatchWriteItem、および TransactWriteItems)。DAX クライアントでこれらのオペレーションを使用すると、項目は DAX と DynamoDB の両方で変更されます。DAX は、その TTL 値に関係なく項目キャッシュ内の項目を更新します。

たとえば、DAX クライアントから GetItem リクエストを発行して、ProductCatalog テーブルから項目を読み取るとします。(パーティションキーは Id で、ソートキーはありません。) Id が 101 である項目を取得します。その項目の QuantityOnHand 値は 42 です。DAX は特定の TTL で項目を項目キャッシュに保存します。この例では、TTL が 10 分とします。3 分後、別のアプリケーションが DAX クライアント使用して、QuantityOnHand 値が 41 になるように同じ項目を更新します。その後はその項目が更新されないとすると、その後 10 分間の同じ項目に対する後続の読み込みは、QuantityOnHand に対してキャッシュされた値を返します (41)。

DAX の書き込み処理方法

DAX は、高パフォーマンスの読み込みを必要とするアプリケーションを対象としています。書き込みスルーキャッシュとして、DAX は書き込みを DynamoDB に同期的に渡し、その結果生じた更新をクラスター内のすべてのノードで項目キャッシュに自動的に非同期にレプリケートします。キャッシュ無効化ロジックを管理する必要はありません。DAX がお客様の代わりにそれを行います。

DAX では、PutItem、UpdateItem、DeleteItem、BatchWriteItem、および TransactWriteItems の各書き込みオペレーションがサポートされます。

これらの PutItem、UpdateItem、DeleteItem、または BatchWriteItem リクエストを DAX に送信すると、次のことが発生します。

- DAX が DynamoDB にリクエストを送信します。
- DynamoDB が DAX に応答し、書き込みが成功したことが確認されます。
- DAX がキャッシュ項目に項目を書き込みます。
- DAX がリクエスタに成功を返します。

これらの TransactWriteItems リクエストを DAX に送信すると、次のことが発生します。

- DAX が DynamoDB にリクエストを送信します。
- DynamoDB が DAX に応答し、トランザクションが完了したことが確認されます。
- DAX がリクエスタに成功を返します。
- バックグラウンドで、DAX は項目キャッシュに項目を格納するために、TransactWriteItems リクエスト内の各項目に対して TransactGetItems リクエストを行います。TransactGetItems は、直列化可能分離を確実にするために使用されます。

スロットルなどの理由で DynamoDB への書き込みが失敗した場合、項目は DAX にキャッシュされません。失敗の例外がリクエスタに返されます。これにより、データが先に DynamoDB に正常に書き込まれない限り、DAX には書き込まれません。

> **Note**
>
> DAX へのすべての書き込みは、項目キャッシュの状態を変更します。ただし、項目キャッシュへの書き込みはクエリキャッシュに影響しません。(DAX の項目キャッシュとクエリキャッシュは異なる目的で使用されます。オペレーションは互いに独立しています。)

DAX クエリキャッシュの動作

DAX は、Query および Scan リクエストからの結果をクエリキャッシュにキャッシュします。ただし、これらの結果は項目キャッシュにまったく影響しません。アプリケーションが Query または Scan 要求を発行すると、結果セットは項目キャッシュではなく、クエリキャッシュに保存されま

す。Scan オペレーションを実行して項目キャッシュを「ウォームアップ」することはできません。項目キャッシュとクエリキャッシュは別々の存在です。

クエリ-更新-クエリの整合性

項目キャッシュ、または基礎となる DynamoDB テーブルに対する更新によって、クエリキャッシュに保存されている結果が無効化されたり変更されたりすることはありません。

説明のため、以下のシナリオを想定してください。アプリケーションは、パーティションキーとして DocumentRevisions を使用し、ソートキーとして DocId を使用する RevisionNumber テーブルを使用しています。

1. 5 より大きいか等しい Query を持つすべての項目に対して、クライアントが DocId 101 に RevisionNumber を発行します。結果を DAX クエリキャッシュに保存し、ユーザーに結果セットを返します。
2. クライアントは、PutItem 値が DocId の 101 RevisionNumber に対して 20 リクエストを発行します。
3. クライアントはステップ 1 の説明と同じ Query を発行します (DocId 101 および RevisionNumber >= 5)。

このシナリオでは、ステップ 3 で発行された Query のキャッシュ結果セットは、ステップ 1 でキャッシュされた結果セットと同じです。これは、DAX では Query または Scan の結果セットを個別の項目に対する更新に基づいて無効化しないためです。ステップ 2 の PutItem オペレーションは、Query の TTL の有効期限が切れたときにのみ DAX クエリキャッシュに反映されます。

アプリケーションでは、クエリキャッシュの TTL 値、およびクエリキャッシュと項目キャッシュ間の結果の不一致をアプリケーションで許容できる期間を考慮に入れる必要があります。

強力な整合性のあるトランザクション読み込み

強い整合性のある GetItem、BatchGetItem、Query、または Scan リクエストを実行するには、ConsistentRead パラメータを true に設定します。DAX は強力な整合性のある読み込みリクエストを DynamoDB に渡します。DynamoDB から返答を受け取ると、DAX はクライアントに結果を返しますが、結果をキャッシュしません。DAX は DynamoDB と緊密には連携していないため、単独では強い整合性のある読み込みを提供できません。このため、DAX からの後続の読み込みは、結果整合性のある読み込みである必要があります。そして、後続の強い整合性のある読み込みはすべて DynamoDB に渡す必要があります。

DAX は、TransactGetItems リクエストを一貫性のある読み込みと同じ方法で処理します。DAX はすべての TransactGetItems リクエストを DynamoDB に渡します。DynamoDB から返答を受け取ると、DAX はクライアントに結果を返しますが、結果をキャッシュしません。

ネガティブキャッシング

DAX は、項目キャッシュとクエリキャッシュの両方でネガティブキャッシュエントリをサポートします。ネガティブキャッシュエントリは、DAX が基礎となる DynamoDB テーブルでリクエストされた項目を見つけられなかったときに発生します。DAX は、エラーを生成する代わりに空の結果をキャッシュし、その結果をユーザーに返します。

たとえば、アプリケーションが GetItem リクエストを DAX クラスターに送信し、DAX 項目キャッシュの中に一致する項目がなかったとします。この場合、DAX は基礎となる DynamoDB テーブルから対応する項目を読み込むことになります。項目が DynamoDB に存在しない場合、DAX は空の項目を項目キャッシュに保存し、その空の項目をアプリケーションに返します。次に、アプリケーションが別の GetItem リクエストを同じ項目に送信するとします。DAX は、項目キャッシュ内に空の項目を見つけ、それをすぐにアプリケーションに返します。DynamoDB は一切参照されません。

ネガティブキャッシュエントリは、項目 TTL の有効期限が切れるか、LRU が呼び出されるか、PutItem、UpdateItem、または DeleteItem を使用して項目が変更されるまで、DAX 項目キャッシュに残ります。

DAX クエリキャッシュも同様の方法でネガティブキャッシュ結果を処理します。アプリケーションが Query または Scan を実行し、DAX クエリキャッシュにキャッシュされた結果がない場合、DAX は DynamoDB にリクエストを送信します。結果セットに一致する項目がない場合、DAX は空の項目セットをクエリキャッシュに保存し、その空の結果セットをアプリケーションに返します。それ以降の Query または Scan リクエストは、その結果セットの TTL の有効期限が切れるまで、同じ (空の) 結果セットを呼び出します。

書き込みの方法

DAX の書き込みスルー動作は、多くのアプリケーションパターンに適しています。ただし、書き込みスルーモデルが適切ではないアプリケーションパターンもいくつかあります。

レイテンシーの影響を受けやすいアプリケーションの場合、DAX を介した書き込みでは余分なネットワークホップが発生します。そのため、DAX への書き込みは、DynamoDB への直接の書き込みよりも少し遅くなります。書き込みレイテンシーが重要なアプリケーションの場合は、代わりに DynamoDB に直接書き込むことで、レイテンシーを軽減できます。詳細については、「書き込み迂回」を参照してください。

書き込み負荷の高いアプリケーション (バルクデータのロードを実行するものなど) では、すべてのデータを DAX 経由で書き込むのは望ましくない場合があります。アプリケーションで読み込むのはそのデータのほんのわずかな割合であるためです。大量のデータを DAX 経由で書き込む場合、読み取られる新しい項目用にキャッシュ内に空きを作るため、LRU アルゴリズムが実行される必要があります。そのため、読み込みキャッシュとしての DAX の効果が低下します。

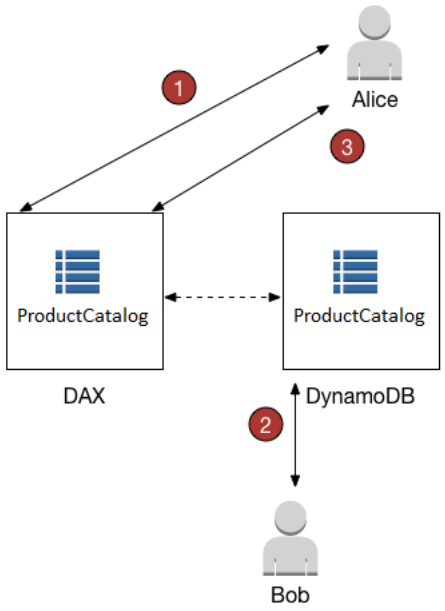
DAX に項目を書き込む場合、項目キャッシュの状態が新しい項目に対応するように変更されます。(たとえば、DAX は新しい項目用の空きを作るために、古いデータを項目キャッシュから削除する必要がある場合があります。) 新しい項目は、キャッシュの LRU アルゴリズムおよびキャッシュの TTL 設定によりますが、項目キャッシュに残ります。項目が項目キャッシュに保持される限り、DAX はその項目を DynamoDB から再ロードしません。

書き込みスルー

DAX 項目キャッシュには書き込みスルーポリシーが導入されています。詳細については、「DAX の書き込み処理方法」を参照してください。

項目を書き込むと、DAX はキャッシュされた項目が DynamoDB に存在する項目と同期されていることを確認します。これは、項目を書き込み直後に再ロードする必要があるアプリケーションにとって有用です。ただし、他のアプリケーションが DynamoDB テーブルに直接書き込んだ場合、DAX 項目キャッシュ内のその項目は、DynamoDB と同期しなくなります。

例として、`ProductCatalog` テーブルを使用している 2 人のユーザー (Alice と Bob) がいるとします。Alice は DAX を使用してテーブルにアクセスしますが、Bob は DAX をバイパスして、DynamoDB 内のテーブルに直接アクセスします。



1. Alice が `ProductCatalog` テーブル内の項目を更新します。DAX が DynamoDB にリクエストを転送し、更新が正常終了します。DAX は、項目を項目キャッシュに書き込み、Alice に成功応答を返します。この時点から、項目が最終的にキャッシュから削除されるまで、この項目を DAX から読み込んだユーザーには Alice が更新した項目が表示されます。
2. しばらくして、Bob は Alice が書き込んだのと同じ `ProductCatalog` 項目を更新します。ただし、ボブは項目を DynamoDB で直接更新します。DAX では、DynamoDB 経由の更新に応じて項目キャッシュが自動的に更新されません。そのため、DAX のユーザーからは、Bob の更新が見えません。
3. Alice が DAX からその項目を再度読み込みます。項目は項目キャッシュにあります。したがって、DAX は DynamoDB テーブルにアクセスせずに Alice に項目を返します。

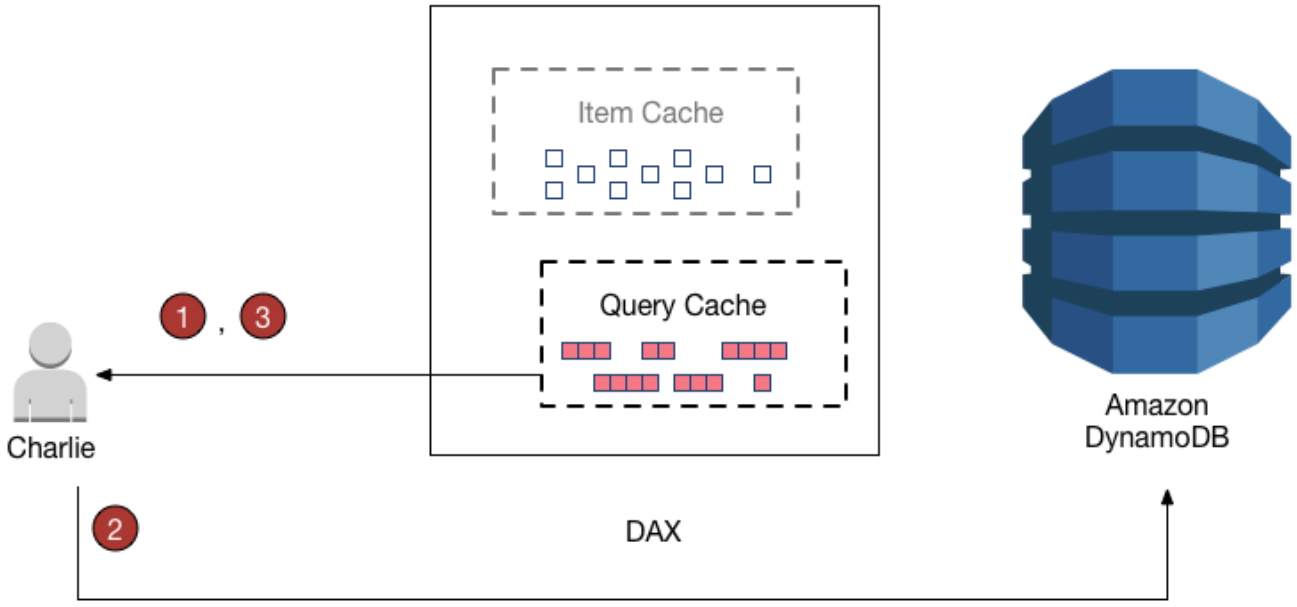
このシナリオでは、Alice と Bob では、同じ `ProductCatalog` 項目で表示が異なります。DAX がその項目を項目キャッシュから削除するか、別のユーザーが同じ項目を DAX で再度更新するまで、この状態になります。

書き込み迂回

アプリケーションで大量のデータ (バルクデータのロードなど) を書き込む必要がある場合、DAX をバイパスしてデータを直接 DynamoDB に書き込む方が妥当な場合があります。このような書き込み迂回の方法により、書き込みレイテンシーを軽減します。ただし、項目キャッシュは DynamoDB のデータと同期しません。

書き込み迂回戦略を使用する場合は、DAX で項目キャッシュに書き込まれるのは、アプリケーションが DAX クライアントを使用してデータを読み取るときであることに注意してください。これは一部のケースでは利点になります。最も頻繁に読み取られるデータのみが (最も頻繁に書き込まれるデータとは反対に) キャッシュされるためです。

たとえば、あるユーザー (Charlie) が、DAX を使用して別のテーブルである `GameScores` テーブルを使用するとしましょう。`GameScores` のパーティションキーは `UserId` であるため、Charlie のスコアは同じ `UserId` を持ちます。



1. Charlie は自分のスコアをすべて取得するため、`Query` を DAX に送信します。このクエリが以前に発行されていないとすると、DAX は処理のためにクエリを DynamoDB に転送します。結果を DAX クエリキャッシュに保存し、Charlie に結果セットを返します。この結果セットは、削除されるまでクエリキャッシュで使用できます。
2. ここで、Charlie が Meteor Blasters ゲームをプレイし、ハイスコアを達成したとします。Charlie は `UpdateItem` リクエストを DynamoDB に送信して、`GameScores` テーブルの項目を変更します。
3. 最後に、Charlie は先の `Query` を再実行して、自分のデータをすべて `GameScores` から取得することにします。この結果には、Meteor Blasters のハイスコアは表示されません。これは、クエリの結果が項目キャッシュではなくクエリキャッシュから取得されるためです。2 つのキャッシュは互いに独立しており、一方のキャッシュでの変更は他方のキャッシュに影響しません。

DAX では、クエリキャッシュの結果セットを DynamoDB の最新のデータに更新しません。クエリキャッシュ内の各結果セットは、`Query` または `Scan` オペレーションが実行された時点のものです。したがって、Charlie の `Query` の結果は、彼の `PutItem` オペレーションを反映していません。DAX が結果セットをクエリキャッシュから削除するまで、この状態が続きます。

DynamoDB Accelerator (DAX) クライアントで開発する

DAX をアプリケーションから使用するには、プログラミング言語に DAX クライアントを使用します。DAX クライアントは、既存の Amazon DynamoDB アプリケーションの中断を最小限に抑えるように設計されています。必要な作業は簡単なコード変更だけです。

> **Note**
>
> さまざまなプログラミング言語用の DAX クライアントは以下のサイトで入所できます。
>
> - http://dax-sdk.s3-website-us-west-2.amazonaws.com

このセクションでは、デフォルト Amazon VPC で Amazon EC2 インスタンスを起動し、インスタンスに接続してサンプルアプリケーションを実行する方法を示します。また、DAX クラスターを使用できるように既存のアプリケーションを変更する方法についての情報を提供します。

トピック

- チュートリアル: DynamoDB Accelerator (DAX) を使用してサンプルアプリケーションを実行する
- 既存のアプリケーションを DAX を使用するように変更する

チュートリアル: DynamoDB Accelerator (DAX) を使用してサンプルアプリケーションを実行する

このチュートリアルでは、デフォルトの Virtual Private Cloud (VPC) で Amazon EC2 インスタンスを起動し、インスタンスに接続して、Amazon DynamoDB Accelerator (DAX) を使用するサンプルアプリケーションを実行する方法を示します。

> **Note**
>
> このチュートリアルを完了するには、デフォルト VPC で実行している DAX クラスターが必要です。まだ DAX クラスターを作成していない場合は、「DAX クラスターの作成」の手順を参照してください。

トピック

- ステップ 1: Amazon EC2 インスタンスを起動する
- ステップ 2: ユーザーとポリシーを作成する
- ステップ 3: Amazon EC2 インスタンスの設定
- ステップ 4: サンプルアプリケーションの実行

ステップ 1: Amazon EC2 インスタンスを起動する

Amazon DynamoDB Accelerator (DAX) クラスターが使用可能な場合、デフォルトの Amazon Virtual Private Cloud (Amazon VPC) で Amazon EC2 インスタンスを起動できます。その後、そのインスタンスで DAX クライアントソフトウェアをインストールし、実行できます。

EC2 インスタンスを起動するには

1. AWS マネジメントコンソール にサインインし、Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. [Launch Instance] (インスタンスを起動) を選択して、以下を実行します。

   ステップ 1: Amazon マシンイメージ (AMI) の選択

   1. AMI のリストで Amazon Linux AMI を見つけ、[Select] (選択) を選択します。

   ステップ 2: インスタンスタイプを選択する

   1. インスタンスタイプのリストで、t2.micro を選択します。
   2. [Next: Configure Instance Details] (次のステップ: インスタンスの詳細の設定) を選択します。

   ステップ 3: インスタンスの詳細を設定する

   1. [Network] (ネットワーク) で、デフォルトの VPC を選択します。

2. [Next: Add Storage] (次の手順: ストレージの追加) をクリックします。

ステップ 4: ストレージを追加する

1. [Next: Add Tags] (次の手順: タグの追加) を選択してこのステップをスキップします。

ステップ 5: タグの追加

1. [Next: Configure Security Group] (次のステップ: セキュリティグループの設定) を選択してこのステップをスキップします。

ステップ 6: セキュリティグループを設定する

1. [Select an existing security group] (既存のセキュリティグループの選択) を選択します。
2. セキュリティグループのリストで、[default (デフォルト)] を選択します。これは VPC のデフォルトのセキュリティグループです。
3. [Next: Review and Launch] (次のステップ: 確認と起動) を選択します。

ステップ 7: インスタンス起動の確認

1. [Launch] (起動する) を選択します。

3. [Select an existing key pair or create a new key pair] (既存のキーペアを選択するか、新しいキーペアを作成する) ウィンドウで、次のいずれかを実行します。

   - Amazon EC2 キーペアがない場合は、[Create a new key pair] (新しいキーペアの作成) を選択して画面に表示される指示に従います。プライベートキーファイル (`.pem` ファイル) をダウンロードするかどうか質問されます。このファイルは、後で Amazon EC2 インスタンスにログインするときに必要になります。
   - 既存の Amazon EC2 キーペアがすでにある場合は、[Select a key pair] (キーペアの選択) を選択して、リストからキーペアを選択します。Amazon EC2 インスタンスにログインするには、すでにプライベートキーファイル (`.pem` ファイル) が利用可能になっている必要があります。

4. キーペアを設定したら、[Launch Instances] (インスタンスの起動) を選択します。
5. コンソールナビゲーションペインで [EC2 Dashboard] (EC2 ダッシュボード) を選択し、起動したインスタンスを選択します。下のペインの [Description] (説明) タブで、インスタン

スの [Public DNS] (パブリック DNS) を見つけます。たとえば、`ec2-11-22-33-44.us-west-2.compute.amazonaws.com` です。このパブリック DNS 名をメモします。[ステップ 3: Amazon EC2 インスタンスの設定] で必要になります。

> **Note**
>
> Amazon EC2 インスタンスが使用可能になるまでに数分かかります。その間に、「ステップ 2: ユーザーとポリシーを作成する」に進んでその指示に従います。

ステップ 2: ユーザーとポリシーを作成する

このステップでは、AWS Identity and Access Management を使用して Amazon DynamoDB Accelerator (DAX) クラスターおよび DynamoDB へのアクセス許可を付与するポリシーを持つユーザーを作成します。その後、DAX クラスターを使用するアプリケーションを実行できるようになります。

アクセス許可を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- AWS IAM アイデンティティセンター のユーザーとグループ:

  アクセス許可セットを作成します。「AWS IAM アイデンティティセンター ユーザーガイド」の「アクセス許可セットを作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については IAM ユーザーガイド の サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する を参照してください。
- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については IAM ユーザーガイド の IAM ユーザーのロールの作成 を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス許可の追加」の指示に従います。

JSON ポリシーエディタでポリシーを作成するには

1. AWS マネジメントコンソール にサインインして、IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. 左側のナビゲーションペインで、[ポリシー] を選択します。

   初めて [ポリシー] を選択する場合には、[管理ポリシーにようこそ] ページが表示されます。今すぐ始める を選択します。
3. ページの上部で、[ポリシーを作成] を選択します。
4. ポリシーエディタ セクションで、JSON オプションを選択します。
5. JSON ポリシードキュメントを入力するか貼り付けます。IAM ポリシー言語の詳細については、「IAM JSON ポリシーリファレンス」を参照してください。
6. ポリシーの検証中に生成されたセキュリティ警告、エラー、または一般警告をすべて解決してから、[次へ] を選択します。

   > **Note**
   >
   > いつでも [ビジュアル] と [JSON] エディタオプションを切り替えることができます。ただし、[ビジュアル] エディタで [次へ] に変更または選択した場合、IAM はポリシーを再構成してビジュアルエディタに合わせて最適化することがあります。詳細については、「IAM ユーザーガイド」の「ポリシーの再構成」を参照してください。
7. (オプション) AWS マネジメントコンソール でポリシーを作成または編集するときに、CloudFormation テンプレートで使用できる JSON または YAML ポリシーテンプレートを生成できます。

   これを行うには、ポリシーエディタで [アクション] を選択し、次に [CloudFormation テンプレートを生成] を選択します。CloudFormation の詳細については、「AWS CloudFormation ユーザーガイド」の「AWS Identity and Access Management リソースタイプリファレンス」を参照してください。
8. ポリシーにアクセス権限を追加し終えたら、[次へ] を選択します。
9. [確認と作成] ページで、作成するポリシーの [ポリシー名] と [説明] (オプション) を入力します。このポリシーで定義されているアクセス許可 を確認して、ポリシーによって付与されたアクセス許可を確認します。

10. (オプション) タグをキーと 値のペアとしてアタッチして、メタデータをポリシーに追加します。IAM でのタグの使用の詳細については、「IAM ユーザーガイド」の「AWS Identity and Access Management リソースのタグ」を参照してください。
11. [ポリシーを作成] をクリックして、新しいポリシーを保存します。

[ポリシードキュメント] — 以下のドキュメントをコピーして貼り付け、JSON を作成します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        },
        {
            "Action": [
                "dynamodb:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        }
    ]
}
```

ステップ 3: Amazon EC2 インスタンスの設定

Amazon EC2 インスタンスが使用可能になったら、インスタンスにログインして使用準備を整えます。

> **Note**
>
> 次のステップは、Linux を実行するコンピュータから Amazon EC2 インスタンスに接続することを想定しています。その他の接続方法については、「Amazon EC2 ユーザーガイド」の「Linux インスタンスへの接続」を参照してください。

EC2 インスタンスを設定するには

1. Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. 次の例のように、ssh コマンドを使用して Amazon EC2 インスタンスにログインします。

   ```
   ssh -i my-keypair.pem ec2-user@public-dns-name
   ```

   プライベートキーファイル (.pem ファイル) とインスタンスのパブリック DNS 名を指定する必要があります。(「ステップ 1: Amazon EC2 インスタンスを起動する」を参照してください。)

   ログイン ID は ec2-user です。パスワードは不要です。
3. EC2 インスタンスにログインしたら、次に示すように AWS 認証情報を設定します。AWS アクセスキー ID とシークレットキー (「ステップ 2: ユーザーとポリシーを作成する」で取得したもの) を入力し、デフォルトリージョン名に現在のリージョンを設定します。(次の例では、デフォルトリージョン名は us-west-2 です)。

   ```
   aws configure

   AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
   AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   Default region name [None]: us-west-2
   Default output format [None]:
   ```

Amazon EC2 インスタンスを起動して設定した後、利用可能なサンプルアプリケーションの 1 つを使用して DAX の機能をテストできます。詳細については、「ステップ 4: サンプルアプリケーションの実行」を参照してください。

ステップ 4: サンプルアプリケーションの実行

Amazon DynamoDB Accelerator (DAX) の機能をテストしやすくするため、Amazon EC2 インスタンスで、使用可能なサンプルアプリケーションの 1 つを実行できます。

トピック

- Node.js および DAX
- DAX SDK for Go
- Java および DAX
- .NET および DAX
- Python および DAX

Node.js および DAX

Node.js のデフォルトのクライアント構成

DAX JavaScript SDK クライアントを構成する際には、さまざまなパラメータをカスタマイズすることでパフォーマンス、接続処理、エラー耐性を最適化できます。次の表は、タイムアウト値、再試行メカニズム、認証情報管理、ヘルスモニタリングオプションなど、クライアントが DAX クラスターとやり取りする方法をコントロールするデフォルト構成設定をまとめたものです。詳細については、「DynamoDBClient Operations」を参照してください。

DAX JS SDK クライアントのデフォルト

パラメータ	Type	説明
`region` オプション	`string`	DAX クライアントに使用する AWS リージョン (例 - 'us-east-1')。環境変数を介して指定されていない場合は必須のパラメータとなります。
`endpoint` 必須	`string`	SDK が接続するクラスターのエンドポイント。 例: 暗号化されていない場合 – dax-cluster-name.region.amazonaws.com 暗号化されている場合 – daxs://my-cluster.l6fzcv.da

パラメータ	Type	説明
		x-clusters.us-east-1.amazonaws.com
`requestTimeout` デフォルトは 6000 ミリ秒	`number`	クライアントが DAX からのレスポンスを待機する最大時間を定義します。
`writeRetries` デフォルトは 1	`number`	書き込みリクエスト失敗時の再試行回数。
`readRetries` デフォルトは 1	`number`	読み取りリクエスト失敗時の再試行回数。
`maxRetries` デフォルトは 1	`number`	リクエスト失敗時の最大再試行回数。 readRetries/writeRetries が設定されている場合、readRetries と writeRetries の構成セットが maxRetries よりも優先されます。
`connectTimeout` デフォルトは 10000 ミリ秒	`number`	いずれかのクラスターノードへの接続を確立するためのタイムアウト (ミリ秒単位)。
`maxConcurrentConnections` デフォルトは 100	`number`	クライアントインスタンスが DAX クラスター内のノードごとに作成できる同時接続の最大数を制限します。

パラメータ	Type	説明
`maxRetryDelay` デフォルトは 7000 ミリ秒	`number`	`waitForRecoveryBeforeRetrying` フラグを true に設定してリカバリが必要であることを DAX サーバーが示すと、クライアントは再試行前に一時停止します。これらのリカバリ期間中は、`maxRetryDelay` パラメータによってある再試行から次の再試行までの最大待機時間が決定されます。このリカバリ固有の構成は、DAX サーバーがリカバリモードになっている場合にのみ適用されます。他のすべてのシナリオでは、再試行回数に基づく指数関数的な遅延 (`writeRetries` 、`readRetries` 、`maxRetries` のいずれかのパラメータによって制御)、または例外タイプに応じた即時再試行のいずれかのパターンの再試行動作に従います。

パラメータ	Type	説明
`credentials` オプション	AwsCredentialIdentity | AwsCredentialIdentityProvider	リクエストの認証に使用する AWS 認証情報。AwsCredentialIdentity または AwsCredentialIdentityProvider として指定できます。指定しなかった場合、AWS SDK ではデフォルトの認証情報プロバイダーチェーンを自動的に使用します。例: `{ accessKeyId: 'AKIA...', secretAccessKey: '...', sessionToken: '...' }` * @default ではデフォルトの AWS 認証情報プロバイダーチェーンを使用します。
`healthCheckInterval` デフォルトは 5000 ミリ秒	`number`	クラスターヘルスチェックの間隔 (ミリ秒単位)。間隔を短くするほどより頻繁にチェックされます。
`healthCheckTimeout` デフォルトは 1000 ミリ秒	`number`	ヘルスチェックが完了するまでのタイムアウト (ミリ秒単位)。
`skipHostnameVerification` デフォルトは false	`boolean`	TLS 接続のホスト名検証をスキップします。暗号化されていないクラスターには影響しません。デフォルトではホスト名検証が実行されます。これを True に設定すると検証がスキップされます。オフにすると接続先のクラスターを認証できなくなるため注意してください。

パラメータ	Type	説明
unhealthyConsecutiveErrorCount デフォルトは 5	number	エラーが何回連続で発生したらヘルスチェック間隔内でノードの異常を通知するかを設定します。
clusterUpdateInterval デフォルトは 4000 ミリ秒	number	メンバーシップ変更についてクラスターメンバーをポーリングする間隔を返します。
clusterUpdateThreshold デフォルトは 125	number	メンバーシップ変更についてクラスターがポーリングされない下限しきい値を返します。
credentailProvider 省略可 | デフォルトは null	AwsCredentialIdentityProvider	DAX へのリクエストを認証するために使用される AWS 認証情報のユーザー定義プロバイダー。

DaxDocument のページ分割構成

名前	型	詳細
client	DaxDocument	DaxDocument タイプのインスタンス。
pageSize	数値	ページごとの項目数を指定します。
startingToken オプションです。	すべて	前のレスポンスの LastEvaluatedKey を後続のリクエストに使用できます。

ページ分割の使用については、「the section called “TryDax.js”」を参照してください。

DAX Node.js SDK V3 への移行

この移行ガイドは、既存の DAX Node.js アプリケーションの移行に役立ちます。新しい SDK には Node.js 18 以降が必要で、DynamoDB Accelerator コードの作成方法にいくつかの重要な変更が導入されています。このガイドでは、構文の変更、新しいインポート方法、更新された非同期プログラミングパターンなど、主な違いについて説明します。

V2 Node.js DAX の使用方法

```
const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var client = new AWS.DynamoDB.DocumentClient();

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  client = new AWS.DynamoDB.DocumentClient({ service: dax });
}

// Make Get Call using Dax
var params = {
    TableName: 'TryDaxTable',
    pk: 1,
    sk: 1
}
client.get(params, function (err, data) {
    if (err) {
        console.error(
            "Unable to read item. Error JSON:",
            JSON.stringify(err, null, 2)
          );
    } else {
        console.log(data);
    }
```

```
});
```

V3 Node.js DAX の使用方法

DAX Node.js V3 ノードバージョン 18 以降を使用することをお勧めします。ノード 18 に移行するには以下を使用します。

```
import { DaxDocument } from '@amazon-dax-sdk/lib-dax';
import { DynamoDBDocument } from '@aws-sdk/lib-dynamodb';
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';

let client: DynamoDBDocument | DaxDocument = DynamoDBDocument.from(
  new DynamoDBClient({ region: 'us-west-2' })
);

if (process.argv.length > 2) {
  client = new DaxDocument({
    endpoints: [process.argv[2]],
    region: 'us-west-2',
  });
}

const params = {
  TableName: 'TryDaxTable',
  Key: { pk: 1, sk: 1 },
};

try {
  const results = await client.get(params);
  console.log(results);
} catch (err) {
  console.error(err);
}
```

DAX SDK for Node.js v3.x は AWS SDK for Node.js v3.x と互換性があります。DAX SDK for Node.js v3.x では集約クライアントの使用をサポートしています。DAX では必要最低限のクライアントの作成をサポートしていないことに注意してください。サポート対象外の機能の詳細については、「the section called “AWS SDK V3 と同等のものがない機能”」を参照してください。

このステップに従って、Amazon EC2 インスタンスで Node.js サンプルアプリケーションを実行します。

DAX の Node.js サンプルを実行するには

1. 以下のように、Amazon EC2 インスタンスで Node.js を設定します。

   a. ノードバージョンマネージャー (nvm) をインストールします。

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
   ```

   b. nvm を使用して Node.js をインストールします。

   ```
   nvm install 18
   ```

   c. nvm を使用して Node 18 を使用する

   ```
   nvm use 18
   ```

   d. Node.js が正しくインストールおよび実行されているかテストします。

   ```
   node -e "console.log('Running Node.js ' + process.version)"
   ```

   これにより次のメッセージが表示されるはずです。

   ```
   Running Node.js v18.x.x
   ```

2. ノードパッケージマネージャー (npm) を使用して、DaxDocument Node.js クライアントをインストールします。

   ```
   npm install @amazon-dax-sdk/lib-dax
   ```

TryDax サンプルコード

DynamoDB Accelerator (DAX) のパフォーマンス上の利点を評価するには、以下の手順に従って、標準の DynamoDB クラスターと DAX クラスター間の読み取り操作時間を比較するサンプルテストを実行します。

1. ワークスペースを設定し、`lib-dax` を依存関係としてインストールしたら、the section called “TryDax.js” をプロジェクトにコピーします。
2. DAX クラスターに対してプログラムを実行します。DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- AWS CLI の使用 — 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

次の例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

3. 次に、クラスターエンドポイントをコマンドラインパラメータとして指定してプログラムを実行します。

```
node TryDax.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

次のような出力が表示されます:

```
Attempting to create table; please wait...
Successfully created table. Table status: ACTIVE
Writing data to the table...
Writing 20 items for partition key:  1
Writing 20 items for partition key:  2
Writing 20 items for partition key:  3
...
Running GetItem Test
        Total time: 153555.10 µs - Avg time: 383.89 µs
        Total time: 44679.96 µs - Avg time: 111.70 µs
        Total time: 36885.86 µs - Avg time: 92.21 µs
        Total time: 32467.25 µs - Avg time: 81.17 µs
        Total time: 32202.60 µs - Avg time: 80.51 µs
Running Query Test
        Total time: 14869.25 µs - Avg time: 2973.85 µs
```

```
        Total time: 3036.31 µs - Avg time: 607.26 µs
        Total time: 2468.92 µs - Avg time: 493.78 µs
        Total time: 2062.53 µs - Avg time: 412.51 µs
        Total time: 2178.22 µs - Avg time: 435.64 µs
Running Scan Test
        Total time: 2395.88 µs - Avg time: 479.18 µs
        Total time: 2207.16 µs - Avg time: 441.43 µs
        Total time: 2443.14 µs - Avg time: 488.63 µs
        Total time: 2038.24 µs - Avg time: 407.65 µs
        Total time: 1972.17 µs - Avg time: 394.43 µs
Running Pagination Test
Scan Pagination
[
  { pk: 1, sk: 1, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 2, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 3, someData: 'XXXXXXXXXX' }
]
[
  { pk: 1, sk: 4, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 5, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 6, someData: 'XXXXXXXXXX' }
]
...
Query Pagination
[
  { pk: 1, sk: 1, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 2, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 3, someData: 'XXXXXXXXXX' }
]
[
  { pk: 1, sk: 4, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 5, someData: 'XXXXXXXXXX' },
  { pk: 1, sk: 6, someData: 'XXXXXXXXXX' }
]
...
Attempting to delete table; please wait...
Successfully deleted table.
```

タイミング情報を書き留めます。`GetItem`、`Query`、`Scan` の各テストに必要なマイクロ秒数。

4. このケースでは、DAX クラスターに対してプログラムを実行しました。次に、プログラムを再度実行します。今回は DynamoDB に対して実行します。

5. ここでプログラムを再度実行しますが、今度はクラスターエンドポイント URL をコマンドラインパラメータとして指定しません。

```
node TryDax.js
```

出力を見て、タイミング情報を書き留めます。GetItem、Query および Scan の経過時間は、DynamoDB と比較して DAX の方が大幅に低いはずです。

AWS SDK V3 と同等のものがない機能

- 必要最低限のクライアント – Dax Node.js V3 では必要最低限のクライアントをサポートしていません。

```
const dynamoDBClient = new DynamoDBClient({ region: 'us-west-2' });
const regularParams = {
    TableName: 'TryDaxTable',
    Key: {
        pk: 1,
        sk: 1
    }
};
// The DynamoDB client supports the send operation.
const dynamoResult = await dynamoDBClient.send(new GetCommand(regularParams));

// However, the DaxDocument client does not support the send operation.
const daxClient = new DaxDocument({
    endpoints: ['your-dax-endpoint'],
    region: 'us-west-2',
});

const params = {
    TableName: 'TryDaxTable',
    Key: {
        pk: 1,
        sk: 1
    }
};

// This will throw an error - send operation is not supported for DAX. Please refer
 to documentation.
const result = await daxClient.send(new GetCommand(params));
```

```
console.log(result);
```

- ミドルウェアスタック – Dax Node.js V3 ではミドルウェア関数の使用をサポートしていません。

```
const dynamoDBClient = new DynamoDBClient({ region: 'us-west-2' });
// The DynamoDB client supports the middlewareStack.
dynamoDBClient.middlewareStack.add(
  (next, context) =>> async (args) => {
    console.log("Before operation:", args);
    const result = await next(args);
    console.log("After operation:", result);
    return result;
  },
  {
    step: "initialize", // or "build", "finalizeRequest", "deserialize"
    name: "loggingMiddleware",
  }
);

// However, the DaxDocument client does not support the middlewareStack.
const daxClient = new DaxDocument({
    endpoints: ['your-dax-endpoint'],
    region: 'us-west-2',
});

// This will throw an error - custom middleware and middlewareStacks are not
 supported for DAX. Please refer to documentation.
daxClient.middlewareStack.add(
  (next, context) => async (args) => {
    console.log("Before operation:", args);
    const result = await next(args);
    console.log("After operation:", result);
    return result;
  },
  {
    step: "initialize", // or "build", "finalizeRequest", "deserialize"
    name: "loggingMiddleware",
  }
);
```

TryDax.js

```
import { DynamoDB, waitUntilTableExists, waitUntilTableNotExists } from "@aws-sdk/
client-dynamodb";
import { DaxDocument, daxPaginateScan, daxPaginateQuery } from "@amazon-dax-sdk/lib-
dax";
import { DynamoDBDocument, paginateQuery, paginateScan } from "@aws-sdk/lib-dynamodb";

const region = "us-east-1";
const tableName = "TryDaxTable";

// Determine the client (DynamoDB or DAX)
let client = DynamoDBDocument.from(new DynamoDB({ region }));
if (process.argv.length > 2) {
  client = new DaxDocument({ region, endpoint: process.argv[2] });
}

// Function to create table
async function createTable() {
  const dynamodb = new DynamoDB({ region });
  const params = {
    TableName: tableName,
    KeySchema: [
      { AttributeName: "pk", KeyType: "HASH" },
      { AttributeName: "sk", KeyType: "RANGE" },
    ],
    AttributeDefinitions: [
      { AttributeName: "pk", AttributeType: "N" },
      { AttributeName: "sk", AttributeType: "N" },
    ],
    ProvisionedThroughput: { ReadCapacityUnits: 25, WriteCapacityUnits: 25 },
  };

  try {
    console.log("Attempting to create table; please wait...");
    await dynamodb.createTable(params);
    await waitUntilTableExists({ client: dynamodb, maxWaitTime: 30 }, { TableName:
 tableName });
    console.log("Successfully created table. Table status: ACTIVE");
  } catch (err) {
    console.error("Error in table creation:", err);
  }
}
```

```
// Function to insert data
async function writeData() {
  console.log("Writing data to the table...");
  const someData = "X".repeat(10);
  for (let ipk = 1; ipk <= 20; ipk++) {
    console.log("Writing 20 items for partition key: ", ipk)
    for (let isk = 1; isk <= 20; isk++) {
      try {
        await client.put({ TableName: tableName, Item: { pk: ipk, sk: isk,
 someData } });
      } catch (err) {
        console.error("Error inserting data:", err);
      }
    }
  }
}

// Function to test GetItem
async function getItemTest() {
  console.log("Running GetItem Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let ipk = 1; ipk <= 20; ipk++) {
      for (let isk = 1; isk <= 20; isk++) {
        promises.push(client.get({ TableName: tableName, Key: { pk: ipk, sk: isk } }));
      }
    }
    await Promise.all(promises);
    const endTime = performance.now();
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
    const avgTime = (iterTime / 400).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Query
async function queryTest() {
  console.log("Running Query Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let pk = 1; pk <= 5; pk++) {
```

```
      const params = {
        TableName: tableName,
        KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
        ExpressionAttributeValues: { ":pkval": pk, ":skval1": 1, ":skval2": 2 },
      };
      promises.push(client.query(params));
    }
    await Promise.all(promises);
    const endTime = performance.now();
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
    const avgTime = (iterTime / 5).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Scan
async function scanTest() {
  console.log("Running Scan Test");
  for (let i = 0; i < 5; i++) {
    const startTime = performance.now();
    const promises = [];
    for (let pk = 1; pk <= 5; pk++) {
      const params = {
        TableName: tableName,
        FilterExpression: "pk = :pkval and sk between :skval1 and :skval2",
        ExpressionAttributeValues: { ":pkval": pk, ":skval1": 1, ":skval2": 2 },
      };
      promises.push(client.scan(params));
    }
    await Promise.all(promises);
    const endTime = performance.now();
    const iterTime = (endTime - startTime) * 1000;
    const totalTime = iterTime.toFixed(2).padStart(3, ' ');
    const avgTime = (iterTime / 5).toFixed(2).padStart(3, ' ');
    console.log(`\tTotal time: ${totalTime} \u00B5s - Avg time: ${avgTime} \u00B5s`);
  }
}

// Function to test Pagination
async function paginationTest() {
  console.log("Running Pagination Test");
  console.log("Scan Pagination");
  const scanParams = { TableName: tableName };
```

```
  const paginator = process.argv.length > 2 ? daxPaginateScan : paginateScan;
  for await (const page of paginator({ client, pageSize: 3 }, scanParams)) {
    console.log(page.Items);
  }

  console.log("Query Pagination");
  const queryParams = {
    TableName: tableName,
    KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
    ExpressionAttributeValues: { ":pkval": 1, ":skval1": 1, ":skval2": 10 },
  };
  const queryPaginator = process.argv.length > 2 ? daxPaginateQuery : paginateQuery;
  for await (const page of queryPaginator({ client, pageSize: 3 }, queryParams)) {
    console.log(page.Items);
  }
}

// Function to delete the table
async function deleteTable() {
  const dynamodb = new DynamoDB({ region });
  console.log("Attempting to delete table; please wait...")
  try {
    await dynamodb.deleteTable({ TableName: tableName });
    await waitUntilTableNotExists({ client: dynamodb, maxWaitTime: 30 }, { TableName:
 tableName });
    console.log("Successfully deleted table.");
  } catch (err) {
    console.error("Error deleting table:", err);
  }
}

// Execute functions sequentially
(async function () {
  await createTable();
  await writeData();
  await getItemTest();
  await queryTest();
  await scanTest();
  await paginationTest();
  await deleteTable();
})();
```

DAX SDK for Go

この手順に従って、Amazon EC2 インスタンスで Go 対応 Amazon DynamoDB Accelerator (DAX) SDK for Go サンプルアプリケーションを実行します。

DAX の SDK for Go サンプルを実行するには

1. Amazon EC2 インスタンスで SDK for Go をセットアップします。

   a. Go プログラミング言語をインストール (Golang) します。

   ```
   sudo yum install -y golang
   ```

   b. Golang が正しくインストールおよび実行されているかテストします。

   ```
   go version
   ```

   次のようなメッセージが表示されます。

   ```
   go version go1.23.4 linux/amd64
   ```

2. サンプル Golang アプリケーションをインストールします。

   ```
   go get github.com/aws-samples/sample-aws-dax-go-v2
   ```

3. 以下の Golang プログラムを実行します。最初のプログラムは、TryDaxGoTable という DynamoDB テーブルを作成します。2 番目のプログラムは、テーブルにデータを書き込みます。

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
   service dynamodb -command create-table
   ```

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
   service dynamodb -command put-item
   ```

4. 以下の Golang プログラムを実行します。

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
   service dynamodb -command get-item
   ```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command query
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command scan
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

5. 前のステップで、DynamoDB エンドポイントに対してプログラムを実行しました。ここでプログラムを再度実行しますが、今度は GetItem、Query、Scan オペレーションが DAX クラスターによって処理されます。

   DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

   - DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - AWS CLI の使用 — 次のコマンドを入力します。

     ```
     aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
     ```

     次の例のように、クラスターエンドポイントが出力に表示されます。

     ```
     {
         "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
         "Port": 8111,
         "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
     }
     ```

   ここでプログラムを再度実行しますが、今度はクラスターエンドポイントをコマンドラインパラメータとして指定します。

   ```
   go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
    -service dax -command get-item -endpoint my-cluster.l6fzcv.dax-clusters.us-
   east-1.amazonaws.com:8111
   ```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command query -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-query -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go
 -service dax -command paginated-batch-get -endpoint my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com:8111
```

出力の残りの部分を見て、タイミング情報を書き留めます。GetItem、Query および Scan の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

6. 次の Golang プログラムを実行して TryDaxGoTable を削除します。

```
go run ~/go/pkg/mod/github.com/aws-samples/sample-aws-dax-go-v2@v1.0.0/try_dax.go -
service dynamodb -command delete-table
```

AWS SDK for Go V2 と同等のものがない機能

ミドルウェアスタック – DAX Go V2 では APIoptions を通じたミドルウェアスタックの使用をサポートしていません。詳細については、「ミドルウェアを使用した AWS SDK for Go v2 クライアントリクエストのカスタマイズ」を参照してください。

例:

```
// Custom middleware implementation
type customSerializeMiddleware struct{}
```

```
// ID returns the identifier for the middleware
func (m *customSerializeMiddleware) ID() string {
    return "CustomMiddleware"
}
// HandleSerialize implements the serialize middleware handler
func (m *customSerializeMiddleware) HandleSerialize(
    ctx context.Context,
    in middleware.SerializeInput,
    next middleware.SerializeHandler,
) (
    out middleware.SerializeOutput,
    metadata middleware.Metadata,
    err error,
) {
    // Add your custom logic here before the request is serialized
    fmt.Printf("Executing custom middleware for request: %v\n", in)
    // Call the next handler in the middleware chain
    return next.HandleSerialize(ctx, in)
}

func executeGetItem(ctx context.Context) error {
    client, err := initItemClient(ctx)
    if err != nil {
        os.Stderr.WriteString(fmt.Sprintf("failed to initialize client: %v\n", err))
        return err
    }

    st := time.Now()
    for c := 0; c < iterations; c++ {
        for i := 0; i < pkMax; i++ {
            for j := 0; j < skMax; j++ {
                // Create key using attributevalue.Marshal for type safety
                pk, err := attributevalue.Marshal(fmt.Sprintf("%s_%d", keyPrefix, i))
                if err != nil {
                    return fmt.Errorf("error marshaling pk: %v", err)
                }
                sk, err := attributevalue.Marshal(fmt.Sprintf("%d", j))
                if err != nil {
                    return fmt.Errorf("error marshaling sk: %v", err)
                }
                key := map[string]types.AttributeValue{
                    "pk": pk,
                    "sk": sk,
                }
```

```
                in := &dynamodb.GetItemInput{
                    TableName: aws.String(table),
                    Key:       key,
                }

                // Custom middleware option
                customMiddleware := func(o *dynamodb.Options) {
                    o.APIOptions = append(o.APIOptions, func(stack *middleware.Stack)
 error {
                        // Add custom middleware to the stack
                        return stack.Serialize.Add(&customSerializeMiddleware{},
 middleware.After)
                    })
                }

                // Apply options to the GetItem call
                out, err := client.GetItem(ctx, in, customMiddleware)
                if err != nil {
                    return err
                }
                writeVerbose(out)
            }
        }
    }
    d := time.Since(st)
    os.Stdout.WriteString(fmt.Sprintf("Total Time: %v, Avg Time: %v\n", d, d/
iterations))
    return nil
}
```

出力:

```
failed to execute command: custom middleware through APIOptions is not supported in DAX
 client
exit status 1
```

Go のデフォルトのクライアント構成

このガイドでは、DAX クライアントのパフォーマンス、接続管理、ログ記録の動作を微調整できる構成オプションについて説明します。デフォルト設定とそれをカスタマイズする方法を理解することで、Go アプリケーションと DAX とのやり取りを最適化できます。

このセクションの内容

- DAX Go SDK クライアントのデフォルト
- クライアント作成

DAX Go SDK クライアントのデフォルト

パラメータ	Type	説明
`Region` 必須	`string`	DAX クライアントに使用する AWS リージョン (例 - 'us-east-1')。環境を介して指定されていない場合は必須のパラメータとなります。
`HostPorts` 必須	`[] string`	SDK が接続する DAX クラスターエンドポイントのリスト。 例えば、次のようになります。 暗号化されていない場合 – dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com 暗号化されている場合 – daxs://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
`MaxPendingConnectionsPerHost` デフォルトは 10	`number`	連続接続試行の数 (複数の接続が同時に確立されている最中である場合があります)。
`ClusterUpdateThreshold`	`time.Duration`	あるクラスター更新から次のクラスター更新までの最小経過時間。

パラメータ	Type	説明
デフォルトは 125 * time.Millisecond		
`ClusterUpdateInterval` デフォルトは 4 * time.Second	`time.Duration`	クライアントが DAX クラスター情報を自動的に更新する間隔。
`IdleConnectionsReapDelay` デフォルトは 30 * time.Second	`time.Duration`	クライアントが DAX クライアントのアイドル接続を閉じる間隔。
`ClientHealthCheckInterval` デフォルトは 5 * time.Second	`time.Duration`	クライアントが DAX クラスターエンドポイントでヘルスチェックを実行する間隔。
`Credentials` デフォルト	`aws.CredentialsProvider`	DAX サービスへのリクエストを認証するために DAX クライアントが使用する AWS 認証情報。「認証情報と認証情報プロバイダー」を参照してください。
`DialContext` デフォルト	`func`	DAX クラスターへの接続を確立するために DAX クライアントが使用するカスタム関数。

パラメータ	Type	説明
`SkipHostnameVerification` デフォルトは false	ブール	TLS 接続のホスト名検証をスキップします。この設定は暗号化されたクラスターにのみ影響します。True に設定すると、ホスト名検証が無効になります。検証を無効にすると、接続先のクラスターの ID を認証できないため、セキュリティリスクが生じます。デフォルトではホスト名検証が有効になっています。
`RouteManagerEnabled` デフォルトは false	`bool`	このフラグは、ネットワークエラーに直面するルートを削除するために使用されます。
`RequestTimeout` デフォルトは 60 * time.Second	`time.Duration`	クライアントが DAX からのレスポンスを待機する最大時間を定義します。 優先順位: コンテキストタイムアウト (設定されている場合) > `RequestTimmeout` (設定されている場合) > デフォルトの 60 秒の `RequestTimeout` 。
`WriteRetries` デフォルトは 2	`number`	書き込みリクエスト失敗時の再試行回数。
`ReadRetries` デフォルトは 2	`number`	読み取りリクエスト失敗時の再試行回数。

パラメータ	Type	説明
`RetryDelay` デフォルトは 0	`time.Duration`	リクエストが失敗したときの再試行のスロットリングされていないエラーの遅延 (秒単位)。
`Logger` オプション	`logging.Logger`	ロガーは、特定の分類でエントリをログ記録するためのインターフェイスです。
`LogLevel` デフォルトは utils.LogOff	`number`	このログレベルは DAX に対してのみ定義されます。github.com/aws/aws-dax-go-v2/tree/main/dax/utils を使用してインポートできます。 `const ( LogOff LogLevelType = 0 LogDebug LogLevelType = 1 LogDebugWithRequestRetries LogLevelType = 2 )`

> **Note**
>
> `time.Duration` の場合、デフォルトの単位はナノ秒です。どのパラメータにも単位を指定しない場合はナノ秒とみなされます。`daxCfg.ClusterUpdateInterval = 10` は 10 ナノ秒 (`daxCfg.ClusterUpdateInterval = 10 * time.Millisecond` は 10 ミリ秒) を意味します。

クライアント作成

DAX クラスターを作成するには:

- DAX 構成を作成してから、DAX 構成を使用して DAX クライアントを作成します。これを使用すると、必要に応じて DAX 構成を上書きできます。

```
import (
"github.com/aws/aws-dax-go-v2/dax/utils"
"github.com/aws/aws-dax-go-v2/dax"
)

// Non - Encrypted : 'dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com'.
// Encrypted : daxs://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com'.

config := dax.DefaultConfig()
config.HostPorts = []string{endpoint}
config.Region = region
config.LogLevel = utils.LogDebug
daxClient, err := dax.New(config)
```

DAX Go SDK V2 への移行

この移行ガイドは、既存の DAX Go アプリケーションの移行に役立ちます。

V1 DAX Go SDK の使用方法

```
package main

import (
    "fmt"
    "os"

    "github.com/aws/aws-dax-go/dax"
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/dynamodb"
)

func main() {
    region := "us-west-2"
    endpoint := "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
```

```
    // Create session
    sess, err := session.NewSession(&aws.Config{
        Region: aws.String(region),
    })
    if err != nil {
        fmt.Printf("Failed to create session: %v\n", err)
        os.Exit(1)
    }

    // Configure DAX client
    cfg := dax.DefaultConfig()
    cfg.HostPorts = []string{endpoint}
    cfg.Region = region

    // Create DAX client
    daxClient, err := dax.New(cfg)
    if err != nil {
        fmt.Printf("Failed to create DAX client: %v\n", err)
        os.Exit(1)
    }
    defer daxClient.Close() // Don't forget to close the client

    // Create GetItem input
    input := &dynamodb.GetItemInput{
        TableName: aws.String("TryDaxTable"),
        Key: map[string]*dynamodb.AttributeValue{
            "pk": {
                N: aws.String("1"),
            },
            "sk": {
                N: aws.String("1"),
            },
        },
    }

    // Make the GetItem call
    result, err := daxClient.GetItem(input)
    if err != nil {
        fmt.Printf("Failed to get item: %v\n", err)
        os.Exit(1)
    }

    // Print the result
```

```
    fmt.Printf("GetItem succeeded: %+v\n", result)
}
```

V2 DAX Go SDK の使用方法

```
package main

import (
    "context"
    "fmt"
    "os"

    "github.com/aws/aws-dax-go-v2/dax"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb"
    "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
    "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
    "github.com/aws/aws-sdk-go-v2/aws"
)

func main() {
    ctx := context.Background()
    region := "us-west-2"
    endpoint := "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"

    // Create DAX config
    config := dax.DefaultConfig()
    // Specify Endpoint and Region
    config.HostPorts = []string{endpoint}
    config.Region = region
    // Enabling logging
    config.LogLevel = utils.LogDebug
    // Create DAX client
    daxClient, err := dax.New(config)
    if err != nil {
        fmt.Printf("Failed to create DAX client: %v\n", err)
        os.Exit(1)
    }
    defer daxClient.Close() // Don't forget to close the client

    // Create key using attributevalue.Marshal for type safety
    pk, err := attributevalue.Marshal(fmt.Sprintf("%s_%d", keyPrefix, i))
    if err != nil {
```

```
        return fmt.Errorf("error marshaling pk: %v", err)
    }
    sk, err := attributevalue.Marshal(fmt.Sprintf("%d", j))
    if err != nil {
        return fmt.Errorf("error marshaling sk: %v", err)
    }

    // Create GetItem input
    input := &dynamodb.GetItemInput{
        TableName: aws.String("TryDaxTable"),
        Key: map[string]types.AttributeValue{
             "pk": pk,
             "sk": sk,
        },
    }

    // Make the GetItem call
    result, err := daxClient.GetItem(ctx, input)
    if err != nil {
        fmt.Printf("Failed to get item: %v\n", err)
        os.Exit(1)
    }

    // Print the result
    fmt.Printf("GetItem succeeded: %+v\n", result)
}
```

API の使用方法の詳細については、AWS のサンプルを参照してください。

Java および DAX

DAX SDK for Java 2.x は AWS SDK for Java 2.x と互換性があります。Java 8 以降をベースに構築されており、非ブロック I/O がサポートされています。AWS SDK for Java 1.x での DAX の使用については、「DAX を AWS SDK for Java 1.x で使用する」を参照してください。

クライアントを Maven 依存関係として使用する

これらのステップに従って、アプリケーションで Java 用 DAX SDK のクライアントを依存関係として使用します。

1. Apache Maven をダウンロードし、インストールします。詳細については、「Apache Maven のダウンロード」および「Apache Maven のインストール」を参照してください。

2. Maven 依存関係クライアントをアプリケーションのプロジェクトオブジェクトモデル (POM) ファイルに追加します。この例では、*x.x.x* を クライアントの実際のバージョン番号に置き換えます。

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>software.amazon.dax</groupId>
        <artifactId>amazon-dax-client</artifactId>
        <version>x.x.x</version>
    </dependency>
</dependencies>
```

TryDax サンプルコード

ワークスペースを設定し、DAX SDK を依存関係として追加したら、TryDax.java をプロジェクトに追加します。

このコマンドを使用してコードを実行します。

```
java -cp classpath TryDax
```

次のような出力が表示されます。

```
Creating a DynamoDB client

Attempting to create table; please wait...
Successfully created table.  Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
...

Running GetItem and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits

GetItem test - partition key 1-100 and sort keys 1-10
  Total time: 4390.240 ms - Avg time: 4.390 ms
  Total time: 3097.089 ms - Avg time: 3.097 ms
```

```
  Total time: 3273.463 ms - Avg time: 3.273 ms
  Total time: 3353.739 ms - Avg time: 3.354 ms
  Total time: 3533.314 ms - Avg time: 3.533 ms
Query test - partition key 1-100 and sort keys between 2 and 9
  Total time: 475.868 ms - Avg time: 4.759 ms
  Total time: 423.333 ms - Avg time: 4.233 ms
  Total time: 460.271 ms - Avg time: 4.603 ms
  Total time: 397.859 ms - Avg time: 3.979 ms
  Total time: 466.644 ms - Avg time: 4.666 ms

Attempting to delete table; please wait...
Successfully deleted table.
```

タイミング情報を書き留めます。これは、GetItem および Query テストに必要なミリ秒の数字です。このケースでは、DynamoDB エンドポイントに対してプログラムを実行しました。次に、プログラムを再度実行します。今回は DAX クラスターに対して実行します。

DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- DynamoDB コンソールで DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

  ```
  dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
  ```

- AWS CLI を使用して、次のコマンドを入力します。

  ```
  aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
  ```

  次の例のように、クラスターエンドポイントのアドレス、ポート、URL が出力に表示されます。

  ```
  {
      "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
      "Port": 8111,
      "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
  }
  ```

ここでプログラムを再度実行しますが、今度はクラスターエンドポイント URL をコマンドラインパラメータとして指定します。

```
java -cp classpath TryDax dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

出力を見て、タイミング情報を書き留めます。GetItem および Query の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

SDK のメトリクス

DAX SDK for Java 2.x を使用すると、アプリケーションのサービスクライアントに関するメトリクスを収集し、Amazon CloudWatch で出力を分析できます。詳細については、「メトリクスを有効にする」を参照してください。

> **Note**
>
> DAX SDK for Java は ApiCallSuccessful および ApiCallDuration メトリクスのみを収集します。

TryDax.java

```
import java.util.Map;

import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.dax.ClusterDaxAsyncClient;
import software.amazon.dax.Configuration;

public class TryDax {
    public static void main(String[] args) throws Exception {
        DynamoDbAsyncClient ddbClient = DynamoDbAsyncClient.builder()
                .build();

        DynamoDbAsyncClient daxClient = null;
        if (args.length >= 1) {
```

```
            daxClient = ClusterDaxAsyncClient.builder()
                    .overrideConfiguration(Configuration.builder()
                            .url(args[0]) // e.g. dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com
                            .build())
                    .build();
        }

        String tableName = "TryDaxTable";

        System.out.println("Creating table...");
        createTable(tableName, ddbClient);

        System.out.println("Populating table...");
        writeData(tableName, ddbClient, 100, 10);

        DynamoDbAsyncClient testClient = null;
        if (daxClient != null) {
            testClient = daxClient;
        } else {
            testClient = ddbClient;
        }

        System.out.println("Running GetItem and Query tests...");
        System.out.println("First iteration of each test will result in cache misses");
        System.out.println("Next iterations are cache hits\n");

        // GetItem
        getItemTest(tableName, testClient, 100, 10, 5);

        // Query
        queryTest(tableName, testClient, 100, 2, 9, 5);

        System.out.println("Deleting table...");
        deleteTable(tableName, ddbClient);
    }

    private static void createTable(String tableName, DynamoDbAsyncClient client) {
        try {
            System.out.println("Attempting to create table; please wait...");

            client.createTable(CreateTableRequest.builder()
                    .tableName(tableName)
                    .keySchema(KeySchemaElement.builder()
```

```
                        .keyType(KeyType.HASH)
                        .attributeName("pk")
                        .build(), KeySchemaElement.builder()
                        .keyType(KeyType.RANGE)
                        .attributeName("sk")
                        .build())
                .attributeDefinitions(AttributeDefinition.builder()
                        .attributeName("pk")
                        .attributeType(ScalarAttributeType.N)
                        .build(), AttributeDefinition.builder()
                        .attributeName("sk")
                        .attributeType(ScalarAttributeType.N)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build()).get();
        client.waiter().waitUntilTableExists(DescribeTableRequest.builder()
                .tableName(tableName)
                .build()).get();
        System.out.println("Successfully created table.");

    } catch (Exception e) {
        System.err.println("Unable to create table: ");
        e.printStackTrace();
    }
}

private static void deleteTable(String tableName, DynamoDbAsyncClient client) {
    try {
        System.out.println("\nAttempting to delete table; please wait...");
        client.deleteTable(DeleteTableRequest.builder()
                .tableName(tableName)
                .build()).get();
        client.waiter().waitUntilTableNotExists(DescribeTableRequest.builder()
                .tableName(tableName)
                .build()).get();
        System.out.println("Successfully deleted table.");

    } catch (Exception e) {
        System.err.println("Unable to delete table: ");
        e.printStackTrace();
    }
}
```

```
   private static void writeData(String tableName, DynamoDbAsyncClient client, int
pkmax, int skmax) {
       System.out.println("Writing data to the table...");

       int stringSize = 1000;
       StringBuilder sb = new StringBuilder(stringSize);
       for (int i = 0; i < stringSize; i++) {
           sb.append('X');
       }
       String someData = sb.toString();

       try {
           for (int ipk = 1; ipk <= pkmax; ipk++) {
               System.out.println(("Writing " + skmax + " items for partition key: " +
ipk));
               for (int isk = 1; isk <= skmax; isk++) {
                   client.putItem(PutItemRequest.builder()
                           .tableName(tableName)
                           .item(Map.of("pk", attr(ipk), "sk", attr(isk), "someData",
attr(someData)))
                           .build()).get();
               }
           }
       } catch (Exception e) {
           System.err.println("Unable to write item:");
           e.printStackTrace();
       }
   }

   private static AttributeValue attr(int n) {
       return AttributeValue.builder().n(String.valueOf(n)).build();
   }

   private static AttributeValue attr(String s) {
       return AttributeValue.builder().s(s).build();
   }

   private static void getItemTest(String tableName, DynamoDbAsyncClient client, int
pk, int sk, int iterations) {
       long startTime, endTime;
       System.out.println("GetItem test - partition key 1-" + pk + " and sort keys 1-"
+ sk);

       for (int i = 0; i < iterations; i++) {
```

```
            startTime = System.nanoTime();
            try {
                for (int ipk = 1; ipk <= pk; ipk++) {
                    for (int isk = 1; isk <= sk; isk++) {
                        client.getItem(GetItemRequest.builder()
                                .tableName(tableName)
                                .key(Map.of("pk", attr(ipk), "sk", attr(isk)))
                                .build()).get();
                    }
                }
            } catch (Exception e) {
                System.err.println("Unable to get item:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk * sk);
        }
    }

    private static void queryTest(String tableName, DynamoDbAsyncClient client, int pk,
int sk1, int sk2, int iterations) {
        long startTime, endTime;
        System.out.println("Query test - partition key 1-" + pk + " and sort keys
between " + sk1 + " and " + sk2);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            for (int ipk = 1; ipk <= pk; ipk++) {
                try {
                    // Pagination API for Query.
                    client.queryPaginator(QueryRequest.builder()
                            .tableName(tableName)
                            .keyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
                            .expressionAttributeValues(Map.of(":pkval", attr(ipk),
":skval1", attr(sk1), ":skval2", attr(sk2)))
                            .build()).items().subscribe((item) -> {
                    }).get();
                } catch (Exception e) {
                    System.err.println("Unable to query table:");
                    e.printStackTrace();
                }
            }
            endTime = System.nanoTime();
```

```
            printTime(startTime, endTime, pk);
        }
    }

    private static void printTime(long startTime, long endTime, int iterations) {
        System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
 (1000000.0));
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *
 1000000.0));
    }
}
```

.NET および DAX

次のステップに従って、Amazon EC2 インスタンスで .NET サンプルを実行します。

> **Note**
>
> このチュートリアルでは .NET 9 SDK を使用します。ここでは、デフォルト Amazon VPC のプログラムを実行して、Amazon DynamoDB Accelerator (DAX) クラスターにアクセスする方法を示しています。AWS SDK v4 for .NET で動作します。V4 の変更と移行に関する詳細については、「AWS SDK for .NET のバージョン 4 への移行」を参照してください。必要に応じて、AWS Toolkit for Visual Studio を使用して .NET アプリケーションを記述し、VPC 内にデプロイできます。
> 詳細については、「AWS Elastic Beanstalk デベロッパーガイド」の「AWS Toolkit for Visual Studio を使用した .NET での Elastic Beanstalk アプリケーションの作成とデプロイ」を参照してください。

DAX の .NET サンプルを実行するには

1. Microsoft ダウンロードページ にアクセスし、最新の .NET 9 SDK for Linux をダウンロードします。ダウンロードされたファイルは、`dotnet-sdk-N.N.N-linux-x64.tar.gz` です。
2. SDK ファイルを展開します。

   ```
   mkdir dotnet
   tar zxvf dotnet-sdk-N.N.N-linux-x64.tar.gz -C dotnet
   ```

   N.N.N を .NET SDK の実際のバージョン番号 (例: `9.0.305`) で置き換えます。

3. インストールを確認します。

```
alias dotnet=$HOME/dotnet/dotnet
dotnet --version
```

これにより、.NET SDK のバージョン番号が表示されます。

> **Note**
>
> バージョン番号ではなく、以下のエラーが表示される場合があります。
> error: libunwind.so.8: 共有オブジェクトファイルを開けません: そのようなファイルまたはディレクトリは存在しません
> このエラーを解決するには、libunwind パッケージをインストールします。
>
> ```
> sudo yum install -y libunwind
> ```
>
> この操作を行うと、dotnet --version コマンドがエラーなしで実行されます。

4. 新しい .NET プロジェクトを作成します。

```
dotnet new console -o myApp
```

この 1 回限りの設定には数分かかります。設定が完了したら、サンプルプロジェクトを実行します。

```
dotnet run --project myApp
```

Hello World! というメッセージが表示されます。

5. myApp/myApp.csproj ファイルにはプロジェクトに関するメタデータが含まれています。アプリケーションで DAX クライアントを使用するには、このファイルを次のように変更します。

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net9.0</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="AWSSDK.DAX.Client" Version="*" />
```

```
    </ItemGroup>
</Project>
```

6. サンプルプログラムソースコード (.zip ファイル) をダウンロードします。

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

ダウンロードが完了したら、ソースファイルを解凍します。

```
unzip TryDax.zip
```

7. 次に、サンプルプログラム「dotNet」を一度に 1 つずつ実行します。プログラムごとにそのコンテンツを myApp/Program.cs 内にコピーし、MyApp プロジェクトを実行します。

以下の .NET プログラムを実行します。最初のプログラムは、TryDaxTable という DynamoDB テーブルを作成します。2 番目のプログラムは、テーブルにデータを書き込みます。

```
cp TryDax/dotNet/01-CreateTable.cs myApp/Program.cs
dotnet run --project myApp

cp TryDax/dotNet/02-Write-Data.cs myApp/Program.cs
dotnet run --project myApp
```

8. 次に、いくつかのプログラムを実行し、GetItem、Query、および Scan の各オペレーションを DAX クラスターに対して実行します。DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- AWS CLI の使用 — 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

次の例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

次に、以下のプログラムを実行し、クラスターエンドポイントをコマンドラインパラメータとして指定します (サンプルのエンドポイントを実際の DAX クラスターエンドポイントに置き換えてください)。

```
cp TryDax/dotNet/03-GetItem-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com

cp TryDax/dotNet/04-Query-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com

cp TryDax/dotNet/05-Scan-Test.cs myApp/Program.cs
dotnet run --project myApp dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

9. 次の .NET プログラムを実行し、TryDaxTable を削除します。

```
cp TryDax/dotNet/06-DeleteTable.cs myApp/Program.cs
dotnet run --project myApp
```

これらのプログラムの詳細については、以下のセクションを参照してください。

- 01-CreateTable.cs
- 02-Write-Data.cs
- 03-GetItem-Test.cs
- 04-Query-Test.cs
- 05-Scan-Test.cs

- 06-DeleteTable.cs

01-CreateTable.cs

01-CreateTable.cs プログラムはテーブル (TryDaxTable) を作成します。このセクションの残りの .NET プログラムは、このテーブルに依存します。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new CreateTableRequest()
            {
                TableName = tableName,
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement{ AttributeName = "pk",KeyType = "HASH"},
                    new KeySchemaElement{ AttributeName = "sk",KeyType = "RANGE"}
                },
                AttributeDefinitions = new List<AttributeDefinition>() {
                    new AttributeDefinition{ AttributeName = "pk",AttributeType = "N"},
                    new AttributeDefinition{ AttributeName = "sk",AttributeType  = "N"}
                },
                ProvisionedThroughput = new ProvisionedThroughput()
                {
                    ReadCapacityUnits = 10,
                    WriteCapacityUnits = 10
                }
            };
```

```
            var response = await client.CreateTableAsync(request);

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

02-Write-Data.cs

02-Write-Data.cs プログラムはテストデータを TryDaxTable に書き込みます。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            string someData = new string('X', 1000);
            var pkmax = 10;
            var skmax = 10;

            for (var ipk = 1; ipk <= pkmax; ipk++)
            {
                Console.WriteLine($"Writing {skmax} items for partition key: {ipk}");
                for (var isk = 1; isk <= skmax; isk++)
                {
                    var request = new PutItemRequest()
                    {
                        TableName = tableName,
                        Item = new Dictionary<string, AttributeValue>()
                       {
```

```
                            { "pk", new AttributeValue{N = ipk.ToString() } },
                            { "sk", new AttributeValue{N = isk.ToString() } },
                            { "someData", new AttributeValue{S = someData } }
                        }
                    };

                    var response = await client.PutItemAsync(request);
                }
            }

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

03-GetItem-Test.cs

`03-GetItem-Test.cs` プログラムは、`GetItem` で `TryDaxTable` オペレーションを実行します。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");

            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);
```

```
            var tableName = "TryDaxTable";

            var pk = 1;
            var sk = 10;
            var iterations = 5;

            var startTime = System.DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                for (var ipk = 1; ipk <= pk; ipk++)
                {
                    for (var isk = 1; isk <= sk; isk++)
                    {
                        var request = new GetItemRequest()
                        {
                            TableName = tableName,
                            Key = new Dictionary<string, AttributeValue>() {
                            {"pk", new AttributeValue {N = ipk.ToString()} },
                            {"sk", new AttributeValue {N = isk.ToString() } }
                        }
                        };
                        var response = await client.GetItemAsync(request);
                        Console.WriteLine($"GetItem succeeded for pk: {ipk},sk:
 {isk}");
                    }
                }
            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

04-Query-Test.cs

04-Query-Test.cs プログラムは、Query で TryDaxTable オペレーションを実行します。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");


            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var pk = 5;
            var sk1 = 2;
            var sk2 = 9;
            var iterations = 5;

            var startTime = DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                var request = new QueryRequest()
                {
                    TableName = tableName,
                    KeyConditionExpression = "pk = :pkval and sk between :skval1
 and :skval2",
                    ExpressionAttributeValues = new Dictionary<string,
 AttributeValue>() {
                            {":pkval", new AttributeValue {N = pk.ToString()} },
```

```
                            {":skval1", new AttributeValue {N = sk1.ToString()} },
                            {":skval2", new AttributeValue {N = sk2.ToString()} }
                    }
                };
                var response = await client.QueryAsync(request);
                Console.WriteLine($"{i}: Query succeeded");

            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

05-Scan-Test.cs

05-Scan-Test.cs プログラムは、Scan で TryDaxTable オペレーションを実行します。

```
using System;
using System.Threading.Tasks;
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;

namespace ClientTest
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            string endpointUri = args[0];
            Console.WriteLine($"Using DAX client - endpointUri={endpointUri}");

            var clientConfig = new DaxClientConfig(endpointUri)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()
```

```
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var iterations = 5;

            var startTime = DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                var request = new ScanRequest()
                {
                    TableName = tableName
                };
                var response = await client.ScanAsync(request);
                Console.WriteLine($"{i}: Scan succeeded");
            }

            var endTime = DateTime.Now;
            TimeSpan timeSpan = endTime - startTime;
            Console.WriteLine($"Total time: {timeSpan.TotalMilliseconds}
 milliseconds");

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

06-DeleteTable.cs

06-DeleteTable.cs プログラムは TryDaxTable を削除します。テストが完了してから、このプログラムを実行します。

```
using System;
using System.Threading.Tasks;
using Amazon.DynamoDBv2.Model;
using Amazon.DynamoDBv2;

namespace ClientTest
```

```
{
    class Program
    {
        public static async Task Main(string[] args)
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new DeleteTableRequest()
            {
                TableName = tableName
            };

            var response = await client.DeleteTableAsync(request);

            Console.WriteLine("Hit <enter> to continue...");
            Console.ReadLine();
        }
    }
}
```

Python および DAX

この手順に従って、Amazon EC2 インスタンスで Python サンプルアプリケーションを実行します。

DAX の Python サンプルを実行するには

1. pip ユーティリティを使用して DAX Python クライアントをインストールします。

   ```
   pip install amazon-dax-client
   ```

2. サンプルプログラムソースコード (.zip ファイル) をダウンロードします。

   ```
   wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
   TryDax.zip
   ```

   ダウンロードが完了したら、ソースファイルを解凍します。

   ```
   unzip TryDax.zip
   ```

3. 以下の Python プログラムを実行します。最初のプログラムは、TryDaxTable という Amazon DynamoDB テーブルを作成します。2 番目のプログラムは、テーブルにデータを書き込みます。

```
python 01-create-table.py
python 02-write-data.py
```

4. 以下の Python プログラムを実行します。

```
python 03-getitem-test.py
python 04-query-test.py
python 05-scan-test.py
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

5. 前のステップで、DynamoDB エンドポイントに対してプログラムを実行しました。ここでプログラムを再度実行しますが、今度は GetItem、Query、Scan オペレーションが DAX クラスターによって処理されます。

DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- AWS CLI の使用 — 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

この例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

プログラムを再度実行しますが、今度はクラスターエンドポイントをコマンドラインパラメータとして指定します。

```
python 03-getitem-test.py dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
python 04-query-test.py dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
python 05-scan-test.py dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

出力の残りの部分を見て、タイミング情報を書き留めます。GetItem、Query および Scan の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

6. 次の Python プログラムを実行して TryDaxTable を削除します。

```
python 06-delete-table.py
```

これらのプログラムの詳細については、以下のセクションを参照してください。

- 01-create-table.py
- 02-write-data.py
- 03-getitem-test.py
- 04-query-test.py
- 05-scan-test.py
- 06-delete-table.py

01-create-table.py

01-create-table.py プログラムはテーブル (TryDaxTable) を作成します。このセクションの残りの Python プログラムは、このテーブルに依存します。

```
import boto3


def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.
```

```
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "BillingMode": "PAY_PER_REQUEST",
    }
    table = dyn_resource.create_table(**params)
    print(f"Creating {table_name}...")
    table.wait_until_exists()
    return table


if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

02-write-data.py

02-write-data.py プログラムはテストデータを TryDaxTable に書き込みます。

```
import boto3


def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
```

```
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
                Item={
                    "partition_key": partition_key,
                    "sort_key": sort_key,
                    "some_data": some_data,
                }
            )
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")


if __name__ == "__main__":
    write_key_count = 10
    write_item_size = 1000
    print(
        f"Writing {write_key_count*write_key_count} items to the table. "
        f"Each item is {write_item_size} characters."
    )
    write_data_to_dax_table(write_key_count, write_item_size)
```

03-getitem-test.py

03-getitem-test.py プログラムは、GetItem で TryDaxTable オペレーションを実行します。この例は、リージョン eu-west-1 の場合に使用できます。

```
import argparse
import sys
import time
import amazondax
import boto3


def get_item_test(key_count, iterations, dyn_resource=None):
    """
```

```
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource('dynamodb')

    table = dyn_resource.Table('TryDaxTable')
    start = time.perf_counter()
    for _ in range(iterations):
        for partition_key in range(1, key_count + 1):
            for sort_key in range(1, key_count + 1):
                table.get_item(Key={
                    'partition_key': partition_key,
                    'sort_key': sort_key
                })
                print('.', end='')
                sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url,
 region_name='eu-west-1') as dax:
            test_start, test_end = get_item_test(
```

```
                test_key_count, test_iterations, dyn_resource=dax)
    else:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the Boto3 client.")
        test_start, test_end = get_item_test(
            test_key_count, test_iterations)
    print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
          f"{(test_end - test_start)/ test_iterations}.")
```

04-query-test.py

04-query-test.py プログラムは、Query で TryDaxTable オペレーションを実行します。

```
import argparse
import time
import sys
import amazondax
import boto3
from boto3.dynamodb.conditions import Key


def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this value.
    :param sort_keys: The range of sort key values for the query. The query returns
                      items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)
```

```
    start = time.perf_counter()
    for _ in range(iterations):
        table.query(KeyConditionExpression=key_condition_expression)
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations, dyn_resource=dax
            )
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations
        )

    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

05-scan-test.py

05-scan-test.py プログラムは、Scan で TryDaxTable オペレーションを実行します。

```
import argparse
import time
import sys
import amazondax
import boto3


def scan_test(iterations, dyn_resource=None):
    """
    Scans the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        table.scan()
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.",
    )
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
```

```
        print(f"Scanning the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = scan_test(test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

06-delete-table.py

`06-delete-table.py` プログラムは `TryDaxTable` を削除します。Amazon DynamoDB Accelerator (DAX) 機能のテストが完了してから、このプログラムを実行します。

```
import boto3


def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()


if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

既存のアプリケーションを DAX を使用するように変更する

Amazon DynamoDB を使用している Java アプリケーションが既にある場合、DynamoDB Accelerator (DAX) クラスターにアクセスできるように変更できます。DAX Java クライアントは、AWS SDK for Java 2.x に含まれる DynamoDB 低レベルクライアントとよく似ているため、アプリケーション全体を書き換える必要はありません。詳細については、「DynamoDB での項目の操作」を参照してください。

> **Note**
>
> この例では AWS SDK for Java 2.x を使用します。レガシー SDK for Java 1.x バージョンの場合は、「既存の SDK for Java 1.x アプリケーションを DAX を使用するように変更する」を参照してください。

プログラムを変更するには、DynamoDB クライアントを DAX クライアントに置き換えます。

```
Region region = Region.US_EAST_1;

// Create an asynchronous DynamoDB client
DynamoDbAsyncClient client = DynamoDbAsyncClient.builder()
                .region(region)
                .build();

// Create an asynchronous DAX client
DynamoDbAsyncClient client = ClusterDaxAsyncClient.builder()
                .overrideConfiguration(Configuration.builder()
                    .url(<cluster url>) // for example, "dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com"
                    .region(region)
                    .addMetricPublisher(cloudWatchMetricsPub) // optionally enable SDK
 metric collection
                    .build())
                .build();
```

また、AWS SDK for Java 2.x の一部である高レベルのライブラリも使用できます。DynamoDB クライアントを DAX クライアントに置き換えてください。

```
Region region = Region.US_EAST_1;
DynamoDbAsyncClient dax = ClusterDaxAsyncClient.builder()
```

```
        .overrideConfiguration(Configuration.builder()
            .url(<cluster url>) // for example, "dax://my-cluster.l6fzcv.dax-
clusters.us-east-1.amazonaws.com"
            .region(region)
            .build())
        .build();

DynamoDbEnhancedAsyncClient enhancedClient = DynamoDbEnhancedAsyncClient.builder()
        .dynamoDbClient(dax)
        .build();
```

詳細については、「DynamoDB テーブル内の項目のマッピング」を参照してください。

DAX クラスターの管理

このセクションでは、Amazon DynamoDB Accelerator (DAX) クラスターの一般的な管理タスクの一部を紹介します。

トピック

- DAX クラスターを管理するための IAM アクセス許可
- DAX クラスターのスケーリング
- DAX クラスターの設定のカスタマイズ
- TTL 設定の構成
- DAX のタグ付けサポート
- AWS CloudTrail の統合
- クラスターを削除する

DAX クラスターを管理するための IAM アクセス許可

AWS マネジメントコンソール または AWS Command Line Interface (AWS CLI)を使用して DAX クラスターを管理する場合は、ユーザーが実行できるアクションの範囲を絞り込むことを強くお勧めします。そうすることで、最小権限の原則に従って、リスクを緩和できます。

次の説明は、DAX 管理 API のアクセスコントロールに焦点を当てます。詳細については、「Amazon DynamoDB API リファレンス」の「Amazon DynamoDB アクセラレーター」を参照してください。

> **Note**
>
> AWS Identity and Access Management (IAM) アクセス許可を管理する方法の詳細については、以下を参照してください。
>
> - IAM と DAX クラスターの作成: DAX クラスターの作成。
> - IAM と DAX データプレーンオペレーション: DAX のアクセスコントロール。

DAX 管理 API では、特定のリソースへの API アクションを範囲に含めることはできません。`Resource` 要素は `"*"` に設定する必要があります。これは `GetItem`、`Query`、`Scan` などの DAX データプレーン API オペレーションとは異なります。データプレーンオペレーションは DAX のクライアントを通じて表示され、これらのオペレーションは特定のリソースを範囲にできます。

例として、次の IAM ポリシードキュメントについて考えます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]
        }
    ]
}
```

このポリシーの目的は、特定のクラスター `DAXCluster01` のみの DAX 管理 API コールを許可することであるとします。

ユーザーが次の AWS CLI コマンドを発行するとします。

```
aws dax describe-clusters
```

基盤となる DescribeClusters API コールが特定のクラスターを範囲にできないため、このコマンドは Not Authorized (権限がありません) という例外で失敗します。ポリシーが構文としては有効であっても、このコマンドは Resource 要素が "*" に設定される必要があるため、失敗します。ただし、ユーザーが DAX データプレーンコール (GetItem や Query など) を DAXCluster01 に送信するプログラムを実行する場合は、それらのコールは成功します。これは、DAX データプレーン API が特定のリソース (この場合は DAXCluster01) を範囲にできるためです。

DAX 管理 API と DAX データプレーン API の両方を包括する単一の包括 IAM ポリシーを作成する場合は、ポリシードキュメントに 2 つの異なるステートメントを含めることをお勧めします。これらのステートメントの 1 つが DAX データプレーン API に対応し、もう 1 つのステートメントが DAX 管理 API に対応するようにします。

次のポリシー例で、この方法を示します。DAXDataAPIs ステートメントは DAXCluster01 リソースを範囲とするが、DAXManagementAPIs のリソースは "*" にする必要がある点に注意してください。各ステートメントに示されているアクションは、説明のみを目的としています。アプリの必要に応じて、これらをカスタマイズできます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXDataAPIs",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]},
            {
            "Sid": "DAXManagementAPIs",
            "Action": [
```

```
                "dax:CreateParameterGroup",
                "dax:CreateSubnetGroup",
                "dax:DecreaseReplicationFactor",
                "dax:DeleteCluster",
                "dax:DeleteParameterGroup",
                "dax:DeleteSubnetGroup",
                "dax:DescribeClusters",
                "dax:DescribeDefaultParameters",
                "dax:DescribeEvents",
                "dax:DescribeParameterGroups",
                "dax:DescribeParameters",
                "dax:DescribeSubnetGroups",
                "dax:IncreaseReplicationFactor",
                "dax:ListTags",
                "dax:RebootNode",
                "dax:TagResource",
                "dax:UntagResource",
                "dax:UpdateCluster",
                "dax:UpdateParameterGroup",
                "dax:UpdateSubnetGroup"
                ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        }
    ]
}
```

DAX クラスターのスケーリング

DAX クラスターのスケーリングには 2 つのオプションを使用できます。第 1 のオプションは水平スケーリングです。クラスターにリードレプリカを追加します。2 番目のオプションは、垂直スケーリング で、さまざまなノードタイプを選択します。アプリケーションに適切なクラスターサイズとノードタイプの選択方法の詳細については、「DAX クラスターサイジングガイド」を参照してください。

水平スケーリング

水平スケーリングでは、クラスターにリードレプリカを追加することで、読み込みオペレーションのスループットを向上させることができます。1 つの DAX クラスターで最大 10 個のリードレプリカをサポートし、レプリカはクラスターの実行中に追加または削除できます。

新しいノードを追加する際は、ピアノードからキャッシュデータを同期する必要があります。そのため、追加時間はキャッシュサイズとアプリケーションのワークロードによって異なります。ベストプラクティスとして、予想されるトラフィックのピークに合わせてクラスターを事前にスケールすることをお勧めします。適切なサイズ設定のガイドラインとモニタリングの推奨事項については、「DAX クラスターサイジングガイド」を参照してください。

次の AWS CLI の例で、ノードの数を増減する方法を説明します。`--new-replication-factor` 引数は、クラスター内のノードの合計数を指定します。ノードの 1 つはプライマリノードであり、他のノードはリードレプリカです。

```
aws dax increase-replication-factor \
    --cluster-name MyNewCluster  \
    --new-replication-factor 5
```

```
aws dax decrease-replication-factor \
    --cluster-name MyNewCluster  \
    --new-replication-factor 3
```

> **Note**
>
> レプリケーション係数を変更すると、クラスターの状態が `modifying` に変わります。変更が完了すると、ステータスが `available` に変わります。

垂直スケーリング

大規模なデータセットがある場合、より大きいノードタイプを使用することでアプリケーションに有益な場合があります。ノードが大きいほど、クラスターがより多くのデータをメモリに保存できるため、キャッシュミスが軽減され、アプリケーション全体のパフォーマンスが向上します。(DAX クラスターのすべてのノードは同じタイプでなければならないことに注意してください。)

DAX クラスターの書き込みオペレーションやキャッシュミスの割合が高い場合は、より大きなノードタイプを使用するとアプリケーションにも利点があります。書き込みオペレーションとキャッシュ

ミスは、クラスターのプライマリノード上のリソースを消費します。したがって、より大きなノードタイプを使用すると、プライマリノードのパフォーマンスが向上し、これらのタイプのオペレーションのスループットが向上します。

実行中の DAX クラスターのノードタイプを変更することはできません。代わりに、目的のノードタイプの新しいクラスターを作成する必要があります。サポートされているノードタイプについては、「ノード」を参照してください。

新しい DAX クラスターを作成するには、AWS マネジメントコンソール、CloudFormation、AWS CLI、または AWS SDK を使用します。(AWS CLI の場合、`--node-type` パラメータを使用してノードタイプを指定します。)

DAX クラスターの設定のカスタマイズ

DAX クラスターを作成する場合、次のデフォルト設定が使用されます。

- 5 分の有効期限 (TTL) で有効化されたキャッシュの自動削除
- アベイラビリティーゾーンの指定なし
- メンテナンスウィンドウの指定なし
- 通知の無効化

新しいクラスターの場合は、作成時に設定をカスタマイズできます。AWS マネジメントコンソールでこれを行うには、[デフォルト設定の使用] をオフにして以下の設定を変更します。

- [Network and Security] (ネットワークとセキュリティ) - 現在の AWS リージョン内の異なるアベイラビリティーゾーンで個別の DAX クラスターノードを実行できます。[No Preference] (指定なし) を選択すると、ノードが自動的にアベイラビリティーゾーン内に分散されます。
- [Parameter Group] (パラメータグループ) - クラスター内の各ノードに適用される指定のパラメータのセット。パラメータグループを使用してキャッシュ TTL の動作を指定できます。パラメータグループ (デフォルトのパラメータグループ `default.dax.1.0` を除く) 内の任意のパラメータの値をいつでも変更できます。
- [Maintenance Window] (メンテナンスウィンドウ) - ソフトウェアのアップグレードやパッチをクラスター内のノードに適用する間隔 (週)。メンテナンスウィンドウの開始日、開始時刻、および所要時間を選択できます。[No Preference] (指定なし) を選択すると、メンテナンスウィンドウはリージョンごとに 8 時間の時間ブロックからランダムに選択されます。詳細については、「メンテナンスウィンドウ」を参照してください。

> **Note**
>
> [Parameter Group] (パラメータグループ) と [Maintenance Window] (メンテナンスウィンドウ) も、実行中のクラスター上でいつでも変更できます。

メンテナンスイベントが発生すると、DAX から Amazon Simple Notification Service (Amazon SNS) を使用して通知が送信されます。通知を設定するには、[SNS 通知のトピック] セレクタからオプションを選択します。新しい Amazon SNS トピックを作成することも、既存のトピックを使用することもできます。

Amazon SNS トピックの設定とサブスクリプションの詳細については、「Amazon Simple Notification Service 開発者ガイド」の「Amazon SNS の開始方法」を参照してください。

TTL 設定の構成

DAX は、DynamoDB から読み込んだデータを 2 つのキャッシュに保持します。

- 項目キャッシュ - `GetItem` または `BatchGetItem` を使用して取得した項目用。
- クエリキャッシュ - `Query` または `Scan` を使用して取得した結果セット用。

詳細については、「項目キャッシュ」および「クエリキャッシュ」を参照してください。

これらのキャッシュそれぞれのデフォルトの TTL は 5 分です。異なる TTL 設定を使用する場合は、カスタムパラメータグループを使用した DAX クラスターを起動できます。コンソールでこれを行うには、ナビゲーションペインの [DAX | Parameter groups (DAX | パラメータグループ)] を選択します。

上記のタスクを AWS CLI を使用して実行することもできます。次の例では、カスタムパラメータグループを使用して新しい DAX クラスターを起動する方法を示しています。この例では、項目キャッシュ TTL は 10 分に設定され、クエリキャッシュ TTL は 3 分に設定されます。

1. 新しいパラメータグループを作成します。

```
aws dax create-parameter-group \
    --parameter-group-name custom-ttl
```

2. 項目キャッシュ TTL を 10 分 (600000 ミリ秒) に設定します。

```
aws dax update-parameter-group \
```

```
    --parameter-group-name custom-ttl \
    --parameter-name-values "ParameterName=record-ttl-millis,ParameterValue=600000"
```

3. クエリキャッシュ TTL を 3 分 (180000 ミリ秒) に設定します。

```
aws dax update-parameter-group \
    --parameter-group-name custom-ttl \
    --parameter-name-values "ParameterName=query-ttl-millis,ParameterValue=180000"
```

4. パラメータが正しく設定されたことを確認します。

```
aws dax describe-parameters --parameter-group-name custom-ttl \
    --query "Parameters[*].[ParameterName,Description,ParameterValue]"
```

このパラメータグループを使用して、新しい DAX クラスターを起動できます。

```
aws dax create-cluster \
    --cluster-name MyNewCluster \
    --node-type dax.r3.large \
    --replication-factor 3 \
    --iam-role-arn arn:aws:iam::123456789012:role/DAXServiceRole \
    --parameter-group custom-ttl
```

> **Note**
>
> 実行中の DAX クラスターにアタッチされたパラメータグループを変更できます。パラメータの変更は、変更後に書き込まれた項目にのみ適用されます。既存のキャッシュされた項目は、最初に書き込まれたときの TTL と設定を保持します。

DAX のタグ付けサポート

DynamoDB を始めとする多くの AWS サービスは、タグ付け (ユーザー定義の名前でリソースにラベル付けする機能) をサポートします。DAX クラスターにタグ付けを割り当てて、同じタグ付けを持つすべての AWS リソースをすばやく識別したり、タグ付け別に AWS の請求書を分類したりできます。

詳細については、「」を参照してくださいDynamoDB でのリソースへのタグとラベルの追加

AWS マネジメントコンソール の使用

DAX クラスタータグを管理するには

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインの、[DAX] で、[クラスター]を選択します。
3. 作業するクラスターを選択します。
4. [タグ] タブを選択します。ここでタグを追加、リスト、編集、または削除できます。

   設定が希望どおりになったら、[変更の適用] を選択します。

AWS CLI の使用

AWS CLI を使用して DAX クラスタータグを管理している場合は、最初にクラスターの Amazon リソースネーム (ARN) を確認する必要があります。次の例は、`MyDAXCluster` という名前のクラスターの ARN を確認する方法を示しています。

```
aws dax describe-clusters \
    --cluster-name MyDAXCluster \
    --query "Clusters[*].ClusterArn"
```

出力で、ARN は次の例のようになります。`arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster`

次の例は、クラスターにタグ付けする方法を示しています。

```
aws dax tag-resource  \
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
    --tags="Key=ClusterUsage,Value=prod"
```

クラスターのすべてのタグを一覧表示します。

```
aws dax list-tags \
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster
```

タグを削除するには、キーを指定します。

```
aws dax untag-resource  \
```

```
    --resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
    --tag-keys ClusterUsage
```

AWS CloudTrail の統合

DAX は AWS CloudTrail と統合されているため、DAX クラスターのアクティビティを監査できます。CloudTrail ログを使用して、クラスターレベルで行われたすべての変更を表示できます。ノード、サブネットグループ、パラメータグループなどのクラスターコンポーネントへの変更も確認できます。詳細については、「」を参照してくださいAWS CloudTrail を使用して DynamoDB オペレーションをログに記録する

クラスターを削除する

使用していないリソースに課金されないように、不要になった DAX クラスターを削除する必要があります。

コンソールまたは AWS CLI を使用して、DAX クラスターを削除できます。以下はその例です。

```
aws dax delete-cluster --cluster-name mydaxcluster
```

DynamoDB Accelerator のモニタリング

モニタリングは、Amazon DynamoDB Accelerator (DAX) および AWS ソリューションの信頼性、可用性、パフォーマンスを維持する上で重要な部分です。マルチポイント障害が発生した場合は、その障害をより簡単にデバッグできるように、AWS ソリューションのすべての部分からモニタリングデータを収集する必要があります。

ただし、DAX のモニタリングを開始する前に、以下の質問に対する回答を反映したモニタリング計画を作成する必要があります。

- どのような目的でモニタリングしますか?
- どのリソースをモニタリングしますか?
- どのくらいの頻度でこれらのリソースをモニタリングしますか?
- どのモニタリングツールを利用しますか?
- 誰がモニタリングタスクを実行しますか?
- 問題が発生したときに誰が通知を受け取りますか?

トピック

- DynamoDB Accelerator のモニタリングツール
- Amazon CloudWatch によるモニタリング
- AWS CloudTrail を使用した DAX オペレーションの口グ記録

DynamoDB Accelerator のモニタリングツール

AWS では、Amazon DynamoDB Accelerator (DAX) のモニタリングに使用できるツールを提供しています。これらの中には、自動モニタリングを設定できるものもあれば、手動操作を必要とするものもあります。モニタリングタスクをできるだけ自動化することをお勧めします。

トピック

- 自動モニタリングツール
- 手動モニタリングツール

自動モニタリングツール

以下の自動化されたモニタリングツールを使用して DAX を監視し、問題が発生したときにレポートできます。

- Amazon CloudWatch のアラーム – 単一のメトリクスを指定した期間モニタリングし、特定のしきい値に対する複数の期間にわたるメトリクスの値に基づいて、1 つ以上のアクションを実行します。アクションは、Amazon Simple Notification Service (Amazon SNS) のトピックまたは Amazon EC2 Auto Scaling のポリシーに送信される通知です。CloudWatch アラームは、特定の状態にあるという理由だけでアクションを呼び出すことはありません。状態が変更され、指定された期間維持されている必要があります。詳細については、「Amazon CloudWatch で DynamoDB のメトリクスをモニタリングする」を参照してください。
- Amazon CloudWatch Logs – AWS CloudTrail またはその他のソースのログファイルのモニタリング、保存、アクセスを行います。詳細については、「Amazon CloudWatch ユーザーガイド」の「ログファイルのモニタリング」を参照してください。
- Amazon CloudWatch Events - イベントに一致したものを 1 つ以上のターゲットの関数またはストリームに渡して、変更、状態の情報の収集、是正措置を行います。詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon CloudWatch Events とは」を参照してください。
- AWS CloudTrail のログのモニタリング – アカウント間でログファイルを共有し、CloudTrail のログファイルを CloudWatch Logs に送信してリアルタイムでモニタリングします。また、ログを処

理するアプリケーションを Java で作成し、CloudTrail からの提供後にログファイルが変更されていないことを確認します。詳細については、「AWS CloudTrail ユーザーガイド」の「CloudTrail ログファイルの使用」を参照してください。

手動モニタリングツール

DAX のモニタリングでもう 1 つ重要な点は、CloudWatch のアラームの対象外の項目を手動でモニタリングすることです。DAX、CloudWatch、Trusted Advisor などの AWS マネジメントコンソールダッシュボードには、AWS 環境の状態が一目でわかるビューが表示されます。DAX のログファイルを確認することもお勧めします。

- DAX ダッシュボードでは以下について確認できます。
  - サービス状態
- CloudWatch のホームページでは以下について確認できます。
  - 現在のアラームとステータス
  - アラームとリソースのグラフ
  - サービスのヘルスステータス

  また、CloudWatch を使用して、次のことが可能です。
  - 重視するサービスをモニタリングするためのカスタマイズしたダッシュボードを作成する。
  - メトリクスデータをグラフ化して、問題をトラブルシューティングして、傾向を確認する。
  - AWS リソースのすべてのメトリクスを検索して参照する。
  - 問題があることを通知するアラームを作成および編集する。

Amazon CloudWatch によるモニタリング

Amazon CloudWatch を使用して DynamoDB Accelerator (DAX) をモニタリングできます。CloudWatch は、DAX から raw データを収集し、読み込み可能なほぼリアルタイムのメトリクスに加工します。これらの統計は 2 週間記録されます。その後は履歴情報にアクセスして、ウェブアプリケーションやサービスの動作をより的確に把握できます。デフォルトでは、DAX メトリクスデータは CloudWatch に自動的に送信されます。詳細については、Amazon CloudWatch ユーザーガイドの「Amazon CloudWatch とは」を参照してください。

トピック

- DAX メトリクスの使用方法は?

- DAX メトリクスとディメンションの表示
- DAX をモニタリングする CloudWatch アラームを作成します
- 本番モニタリング

DAX メトリクスの使用方法は?

DAX によってレポートされるメトリクスが提供する情報は、さまざまな方法で分析できます。以下のリストは、メトリクスの一般的な利用方法をいくつか示しています。ここで紹介するのは開始するための提案事項です。すべてを網羅しているわけではありません。

目的	関連するメトリクス
システムエラーが発生したかどうかを判断する	`FaultRequestCount` をモニタリングして HTTP 500 (サーバーエラー) コードを発生させたリクエストがあるかどうかを判断します。これは DAX の内部サービスエラー、または基盤となるテーブルの SystemErrors メトリクス の HTTP 500 の可能性を示します。
ユーザーエラーが発生したかどうかを判断する	`ErrorRequestCount` をモニタリングして HTTP 400 (クライアントエラー) コードを発生させたリクエストがあるかどうかを判断します。エラーカウントが増えている場合、よく調べて正しいクライアントリクエストを送信していることを確認してください。
キャッシュミスが発生したかどうかを判断する	`ItemCacheMisses` をモニタリングして、項目がキャッシュに見つからなかった回数を判別します。また、`QueryCacheMisses` と `ScanCacheMisses` をモニタリングして、クエリやスキャンの結果がキャッシュで見つからなかった回数を判別します。
キャッシュヒットレートをモニタリングする	CloudWatch Metric Math を使用し、キャッシュヒットレートメトリクスを数式を使用して定義します。 たとえば、項目キャッシュの場合、m1/SUM([m1, m2])*100 という式を使用できます。m1 はクラスターの `ItemCacheHits` メトリクス、m2 は `ItemCacheMisses` メトリクスを意味します。クエリキャッシュやスキャンキャッシュでは、対応する

目的	関連するメトリクス
	クエリやスキャンのキャッシュメトリクスを使用して、次のサンプルパターンに従います。

DAX メトリクスとディメンションの表示

Amazon DynamoDB を操作するとき、メトリクスとディメンションが Amazon CloudWatch に送信されます。以下の手順を使用して、DynamoDB Accelerator (DAX) のメトリクスを表示できます。

メトリクスを表示するには (コンソール)

メトリクスはまずサービスの名前空間ごとにグループ化され、次に各名前空間内のさまざまなディメンションの組み合わせごとにグループ化されます。

1. CloudWatch コンソール (https://console.aws.amazon.com/cloudwatch/) を開きます。
2. ナビゲーションペインで [メトリクス] を選択します。
3. [DAX] 名前空間を選択します。

メトリクス () を表示するにはAWS CLI

- コマンドプロンプトで、次のコマンドを使用します。

```
aws cloudwatch list-metrics --namespace "AWS/DAX"
```

DAX メトリクスとディメンション

以下のセクションでは、DAX が CloudWatch に送信するメトリクスとディメンションについて説明します。

DAX メトリクス

次のメトリクスは DAX から入手できます。DAX は、値がゼロ以外のメトリクスのみを CloudWatch に送信します。

> **Note**
>
> CloudWatch は 1 分間隔で以下の DAX メトリクスを集計します。

- CPUUtilization
- CacheMemoryUtilization
- NetworkBytesIn
- NetworkBytesOut
- BaselineNetworkBytesInUtilization
- BaselineNetworkBytesOutUtilization
- NetworkPacketsIn
- NetworkPacketsOut
- GetItemRequestCount
- BatchGetItemRequestCount
- BatchWriteItemRequestCount
- DeleteItemRequestCount
- PutItemRequestCount
- UpdateItemRequestCount
- TransactWriteItemsCount
- TransactGetItemsCount
- ItemCacheHits
- ItemCacheMisses
- QueryCacheHits
- QueryCacheMisses
- ScanCacheHits
- ScanCacheMisses
- TotalRequestCount
- ErrorRequestCount
- FaultRequestCount
- FailedRequestCount
- QueryRequestCount
- ScanRequestCount
- ClientConnections
- EstimatedDbSize

- EvictedSize
- CPUCreditUsage
- CPUCreditBalance
- CPUSurplusCreditBalance
- CPUSurplusCreditsCharged

Average や Sum など、すべての統計が必ずしも常にすべてのメトリクスに適用可能であるとは限りません。ただし、これらの値はすべて DAX コンソール経由で利用できます。または CloudWatch コンソール、AWS CLI、AWS SDK を使用してすべてのメトリクスを利用できます。次の表は、各メトリクスに適用可能な有効な統計のリストを示します。

メトリクス	説明
CPUUtilization	ノードまたはクラスタの CPU 使用率の割合。 単位: Percent 有効な統計: • Minimum • Maximum • Average
CacheMemoryUtilization	ノードまたはクラスター上のアイテムキャッシュおよびクエリキャッシュによって使用されている使用可能なキャッシュメモリの割合。メモリ使用率が 100% に達する前に、キャッシュされたデータが削除され始めます (EvictedSize メトリクスを参照)。いずれかのノードで CacheMemoryUtilization が 100% に達すると書き込み要求が抑制されるので、より大きなノードタイプのクラスターへの切り替えを検討する必要があります。 単位: Percent 有効な統計:

メトリクス	説明
	• Minimum • Maximum • Average
NetworkBytesIn	ノードまたはクラスターによってすべてのネットワークインターフェイスで受信されたバイトの数。 単位: Bytes 有効な統計: • Minimum • Maximum • Average
NetworkBytesOut	ノートまたはクラスターによってすべてのネットワークインターフェイスで送信されたバイトの数。このメトリクスは、送信トラフィックのボリュームを単一のノードまたはクラスターでのバイト数として識別します。 単位: Bytes 有効な統計: • Minimum • Maximum • Average

メトリクス	説明
BaselineNetworkBytesInUtilization	受信トラフィックの特定の時点で消費されたベースラインネットワーク帯域幅の割合。参考までに、50% は、受信トラフィックに使用可能なネットワーク帯域幅の半分が使用されていることを意味します。 単位: Percent 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
BaselineNetworkBytesOutUtilization	送信トラフィックの特定の時点で消費されたベースラインネットワーク帯域幅の割合。参考までに、50% は、送信トラフィックに使用可能なネットワーク帯域幅の半分が使用されていることを意味します。 単位: Percent 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
NetworkPacketsIn	ノードまたはクラスターによってすべてのネットワークインターフェイスで受信されたパケットの数。 単位: Count 有効な統計: • Minimum • Maximum • Average
NetworkPacketsOut	ノートまたはクラスターによってすべてのネットワークインターフェイスで送信されたパケットの数。このメトリクスは、送信トラフィックのボリュームを単一のノードまたはクラスターでのパケット数として識別します。 単位: Count 有効な統計: • Minimum • Maximum • Average
GetItemRequestCount	ノードまたはクラスターで処理された GetItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
BatchGetItemRequestCount	ノードまたはクラスターで処理された BatchGetItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
BatchWriteItemRequestCount	ノードまたはクラスターで処理された BatchWriteItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
DeleteItemRequestCount	ノードまたはクラスターで処理された DeleteItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
PutItemRequestCount	ノードまたはクラスターで処理された PutItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
UpdateItemRequestCount	ノードまたはクラスターで処理された UpdateItem リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
TransactWriteItemsCount	ノードまたはクラスターで処理された TransactWriteItems リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
TransactGetItemsCount	ノードまたはクラスターで処理された TransactGetItems リクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
ItemCacheHits	ノードまたはクラスターによってキャッシュから項目が返された回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
ItemCacheMisses	項目がノードキャッシュまたはクラスターキャッシュに存在せず、DynamoDB から取得する必要があった回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
QueryCacheHits	ノードキャッシュまたはクラスターキャッシュからクエリ結果が返された回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
QueryCacheMisses	クエリ結果がノードキャッシュまたはクラスターキャッシュに存在せず、DynamoDB から取得する必要があった回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
ScanCacheHits	ノードキャッシュまたはクラスターキャッシュからスキャン結果が返された回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
ScanCacheMisses	スキャン結果がノードキャッシュまたはクラスターキャッシュに存在せず、DynamoDB から取得する必要があった回数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
TotalRequestCount	ノードまたはクラスターによって処理されたリクエストの合計数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
ErrorRequestCount	ノードまたはクラスターによって報告されたユーザーエラーの原因となったリクエストの合計数。ノードまたはクラスターによってスロットリングされたリクエストが含まれます。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
ThrottledRequestCount	ノードまたはクラスターによってスロットリングされたリクエストの合計数。DynamoDB によってスロットリングされたリクエストは含まれません。DynamoDB メトリクス を使用してモニタリングできます。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
FaultRequestCount	ノードまたはクラスターによって報告された内部エラーの原因となったリクエストの総数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
FailedRequestCount	ノードまたはクラスターによって報告されたエラーの原因となったリクエストの合計数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
QueryRequestCount	ノードまたはクラスターによって処理されたクエリリクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
ScanRequestCount	ノードまたはクラスターによって処理されたスキャンリクエストの数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
ClientConnections	クライアントから行われたノードまたはクラスターへの同時接続の数。 単位: Count 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum
EstimatedDbSize	ノードまたはクラスターによって項目キャッシュおよびクエリキャッシュにキャッシュされるデータ量の近似。 単位: Bytes 有効な統計: • Minimum • Maximum • Average

メトリクス	説明
EvictedSize	新しくリクエストされたデータ用のスペースを作るために、ノードまたはクラスターによって エビクトされたデータの量。ミス率が上昇し、この指標も増加している場合は、ワーキングセットが増加したと考えられます。より大きなノードタイプのクラスターへの切り替えを検討する必要があります。 単位: Bytes 有効な統計: • Minimum • Maximum • Average • Sum
CPUCreditUsage	CPU 使用率に関してノードで消費される CPU クレジットの数。1 つの CPU クレジットは、1 個の vCPU が 100% の使用率で 1 分間実行されること、または、vCPU、使用率、時間の同等の組み合わせ (たとえば、1 個の vCPU が 50% の使用率で 2 分間実行されるか、2 個の vCPU が 25% の使用率で 2 分間実行される) に相当します。 CPU クレジットメトリクスは、5 分間隔でのみ利用可能です。5 分を超える期間を指定する場合は、Average の代わりに Sum 統計を使用します。 単位: Credits (vCPU-minutes) 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
CPUCreditBalance	ノードが起動または開始後に蓄積した獲得 CPU クレジットの数。 クレジットは、獲得後にクレジット残高に蓄積され、消費されるとクレジット残高から削除されます。クレジット残高には、DAX ノードサイズによって決まる上限があります。制限に到達すると、獲得された新しいクレジットはすべて破棄されます。 CPUCreditBalance のクレジットは、ノードがそのベースライン CPU 使用率を超えてバーストするために消費できます。 単位: Credits (vCPU-minutes) 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
CPUSurplusCreditBalance	CPUCreditBalance 値がゼロの場合に DAX ノードによって消費された余剰クレジットの数。 CPUSurplusCreditBalance 値は獲得した CPU クレジットによって支払われます。余剰クレジットの数が、24 時間にノードが獲得できるクレジットの最大数を超えている場合、最大数を超えて消費された余剰クレジットに対しては料金が発生します。 単位: Credits (vCPU-minutes) 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

メトリクス	説明
CPUSurplusCreditsCharged	獲得 CPU クレジットにより支払われないために追加料金が発生した消費済み余剰クレジットの数。 消費済み余剰クレジットは、消費された余剰クレジットがノードで 24 時間に獲得できる最大クレジット数を超えている場合に課金されます。最大数を越えて消費された余剰クレジットは、時間の最後またはノードが終了したときに課金されます。 単位: Credits (vCPU-minutes) 有効な統計: • Minimum • Maximum • Average • SampleCount • Sum

> **Note**
>
> CPUCreditUsage、CPUCreditBalance、CPUSurplusCreditBalance、および CPUSurplusCreditsCharged メトリクスは T3 ノードでのみ使用できます。

DAX メトリクスのディメンション

DAX のメトリクスは、アカウント、クラスター ID、またはクラスター ID とノード ID の組み合わせの値によって修飾されます。CloudWatch コンソールを使用して、以下の表に示すいずれかのディメンションに従って DAX データを取得できます。

ディメンション	説明
Account	アカウント内のすべてのノードで集計された統計情報を提供します。
ClusterId	データをクラスターに制限します。
ClusterId, NodeId	データをクラスター内のノードに制限します。

DAX をモニタリングする CloudWatch アラームを作成します

アラームの状態が変わったときに Amazon Simple Notification Service (Amazon SNS) メッセージを送信する Amazon CloudWatch のアラームを作成することができます。指定した期間中、1 つのアラームが 1 つのメトリクスを監視します。このアラームは、複数の期間にわたる一定のしきい値とメトリクスの値の関係性に基づき、1 つ以上のアクションを実行します。アクションは、Amazon SNS のトピックまたは Auto Scaling のポリシーに送信される通知です。アラームは、持続している状態変化に対してのみアクションを呼び出します。CloudWatch アラームは、特定の状態にあるという理由だけではアクションを呼び出しません。状態が変わって、変わった状態が指定期間にわたって維持される必要があります。

クエリキャッシュミスに関する通知を受け取る方法は?

1. Amazon SNS トピック (`arn:aws:sns:us-west-2:522194210714:QueryMissAlarm`) を作成します。

詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon Simple Notification Service の設定」を参照してください。

2. アラームを作成します。

```
aws cloudwatch put-metric-alarm \
    --alarm-name QueryCacheMissesAlarm  \
    --alarm-description "Alarm over query cache misses" \
    --namespace AWS/DAX \
    --metric-name QueryCacheMisses  \
    --dimensions Name=ClusterID,Value=myCluster \
    --statistic Sum \
    --threshold 8 \
    --comparison-operator GreaterThanOrEqualToThreshold \
    --period 60 \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-west-2:522194210714:QueryMissAlarm
```

3. アラームのテストを行います。

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
 "initializing" --state-value ALARM
```

> **Note**
>
> しきい値をアプリケーションに合わせた値に増減できます。また、CloudWatch Metric Math を使用して、キャッシュミス率メトリクスを定義し、そのメトリクスに対してアラームを設定できます。

リクエストがクラスターで内部エラーを起こした場合は、どのようにして通知されますか?

1. Amazon SNS トピック (`arn:aws:sns:us-west-2:123456789012:notify-on-system-errors`) を作成します。

詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon Simple Notification Service の設定」を参照してください。

2. アラームを作成します。

```
aws cloudwatch put-metric-alarm \
    --alarm-name FaultRequestCountAlarm \
    --alarm-description "Alarm when a request causes an internal error" \
    --namespace AWS/DAX \
    --metric-name FaultRequestCount \
    --dimensions Name=ClusterID,Value=myCluster \
    --statistic Sum \
    --threshold 0 \
    --comparison-operator GreaterThanThreshold \
    --period 60 \
    --unit Count \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. アラームのテストを行います。

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-reason
 "initializing" --state-value ALARM
```

本番モニタリング

さまざまなタイミングとロード条件でパフォーマンスを測定することにより、お客様の環境で通常の DAX のパフォーマンスのベースラインを確定する必要があります。DAX をモニタリングするには、モニタリングの履歴データを保存することを検討します。保存データを、最新のパフォーマンスデータと比較するベースラインとして使用し、通常のパフォーマンスのパターンやパフォーマンスの異常を検出して、問題への対応を検討することができます。

ベースラインを確立するには、最低でも、ロードテスト中と本番環境の両方で次の項目をモニタリングする必要があります。

- CPU 使用率とスロットル済みのリクエスト。これにより、クラスター内でより大きなノードタイプを使用する必要があるかどうかを判断できます。クラスターの CPU 使用率

は、CPUUtilization CloudWatch メトリクスで利用できます。このメトリクスの平均統計は、クラスター内のすべてのノードの平均 CPU 使用率ビューを提供します。クラスターのスケーリングの決定には、すべてのノードの最大使用率である最大統計を使用することをお勧めします。

> **Note**
>
> AWS は、CPUUtilization メトリクスの詳細度を改善しました。メトリクスの変更は、2024 年 5 月 17 日から 2024 年 6 月 22 日の間に行われます。

- オペレーションのレイテンシー (クライアント側で測定) は、アプリケーションのレイテンシー要件内で一貫して維持する必要があります。
- ErrorRequestCount、FaultRequestCount、および FailedRequestCount CloudWatch メトリクスからわかるように、エラー率は低いままにしておきます。
- ネットワークバイトの消費量。これにより、クラスターでより多くのノードを使用するか、より大きなノードタイプを使用するかを判断できます。消費量をモニタリングするには、CloudWatch で利用可能な BaselineNetworkBytesInUtilization と BaselineNetworkBytesOutUtilization のメトリクスにアラートを設定します。これにより、受信トラフィックと送信トラフィックそれぞれについて、インスタンスタイプで使用可能なネットワーク帯域幅の消費率を確認できます。
- キャッシュメモリの使用率と削除されたサイズ。これにより、クラスタのノードタイプに作業セットを保持するのに十分なメモリがあるかどうかを判断し、ない場合はより大きなノードタイプに切り替えることができます。

> **Note**
>
> キャッシュミスや書き込みが多数発生すると、キャッシュメモリの使用率が 100% まで増加し、可用性のダウンタイムが発生する可能性があります。

- クライアント接続。これを使用して、クラスターへの接続に原因がわからないスパイクがあるかどうかをモニターできます。

AWS CloudTrail を使用した DAX オペレーションのログ記録

Amazon DynamoDB Accelerator (DAX) は、DAX のユーザー、ロール、または AWS のサービスによって実行されたアクションをレコードするサービスである AWS CloudTrail と統合されています。

DAX と CloudTrail の詳細については、AWS CloudTrail を使用して DynamoDB オペレーションを口グに記録する で DynamoDB Accelerator (DAX) のセクションを参照してください。

DAX T3/T2 バーストインスタンス

DAX では、固定パフォーマンスインスタンス (R4、R5、R7 など) とバーストパフォーマンスインスタンス (T2 や T3 など) を選択できます。バーストパフォーマンスインスタンスは、ベースラインレベルの CPU パフォーマンスを提供しながら、必要に応じてバーストする機能を備えています。

ベースラインパフォーマンスとバースト機能は、CPU クレジットにより管理されます。バーストパフォーマンスインスタンスは、ワークロードがベースラインしきい値を下回ったときに、インスタンスサイズによって決定されるレートで CPU クレジットを継続的に蓄積します。これらのクレジットは、ワークロードが増加すると消費される可能性があります。CPU クレジットは、フル CPU パフォーマンスを 1 分間実現します。

多くのワークロードは、一貫して高レベルの CPU が必要になることはありませんが、必要なときに非常に高速な CPU にフルアクセスできるため、大きなメリットが得られます。バーストパフォーマンスインスタンスは、これらのユースケースに特化して設計されています。データベースで一貫して高い CPU パフォーマンスが必要な場合は、固定パフォーマンスのインスタンスを使用することをお勧めします。

DAX T2 インスタンスファミリー

DAX T2 インスタンスは、バースト可能な汎用パフォーマンスインスタンスを使用し、ベースラインレベルの CPU パフォーマンスを提供しながら、そのベースラインを超えるとバーストする機能を備えています。T2 インスタンスは価格予測可能性を必要とするテストおよび開発ワークロードに適しています。DAX T2 インスタンスはスタンダードモードに設定されます。したがって、インスタンスの蓄積されたクレジットが少なくなると、CPU 使用率は徐々にベースラインレベルまで下がります。スタンダードモードの詳細については、「Amazon EC2 ユーザーガイド」の「バーストパフォーマンスインスタンスのスタンダードモード」を参照してください。

DAX T3 インスタンスファミリー

DAX T3 インスタンスは、次世代のバースタブルな汎用インスタンスタイプで、ベースラインレベルの CPU パフォーマンスを提供しながら、いつでも必要なだけ CPU 使用をバーストできる機能を備えています。T3 インスタンスはコンピューティング、メモリ、ネットワークリソースを提供するので、一時的な使用スパイクが発生する CPU 使用率の中程度のワークロードに最適です。DAX T3 インスタンスは、無制限モードに設定されています。したがって、24 時間にわたってベースラインを

超えてバーストすることがあり、その場合は追加料金が発生します。無制限モードの詳細については、Amazon EC2 ユーザーガイドの「バーストパフォーマンスインスタンスの Unlimited モード」を参照してください。

DAX T3 インスタンスは、ワークロードが必要とする限り、高い CPU パフォーマンスを維持できます。ほとんどの汎用ワークロードでは、T3 インスタンスは追加料金なしで十分なパフォーマンスを提供します。24 時間のT3 インスタンスの平均 CPU 使用率がベースライン以下になった場合、1 時間ごとのインスタンス料金は自動的にすべての暫定スパイクをカバーします。

たとえば、`dax.t3.small` インスタンスは、1 時間あたり 24 CPU クレジットのレートで継続的にクレジットを受け取ります。この機能は、CPU コアの 20% に相当するベースラインパフォーマンスを提供します (20% x 60 分 = 12 分)。インスタンスが受け取ったクレジットを使用しない場合、それらは CPU クレジットバランスに最大 576 CPU クレジットまで格納されます。`t3.small` インスタンスはコアの 20% 以上にバーストする必要がある場合、このサージは CPU クレジットバランスから自動的に処理されます。

CPU クレジットバランスがゼロになると、DAX T2 インスタンスはベースラインパフォーマンスに制限されますが、DAX T3 インスタンスは CPU クレジットバランスがゼロの場合でも、ベースラインを超えてバーストする可能性があります。CPU 使用率がベースラインパフォーマンス以下である大部分のワークロードでは、`t3.small` の基本時間料金は、すべての CPU バーストをカバーします。インスタンスが CPU クレジットバランスがゼロになってから 24 時間にわたって平均 25% の CPU 使用率 (ベースラインを 5% 超過) で実行された場合、さらに 11.52 セント (9.6 セント/VCPU 時間 x 1 vCPU x 5% x 24 時間) が課金されます。料金の詳細については、「Amazon DynamoDB の料金表」を参照してください。

DAX のアクセスコントロール

DynamoDB アクセラレーター (DAX) は DynamoDB と連携して動作し、アプリケーションにキャッシングレイヤーをシームレスに追加するように設計されています。ただし、DAX および DynamoDB には、個別のアクセス制御のメカニズムがあります。どちらのサービスもそれぞれのセキュリティポリシーの実装に AWS Identity and Access Management (IAM) を使用しますが、セキュリティモデルは DAX と DynamoDB で異なります。

両方のセキュリティモデルを理解することを強くお勧めします。そうすることで、DAX を使用するアプリケーションに適切なセキュリティ手段を実装できます。

このセクションでは、DAX が提供するアクセスコントロールメカニズムを説明し、必要に合わせて調整できる IAM ポリシーの例を示します。

クラスター名内の機密データを保護する

DAX クラスターを作成するときは、顧客のアカウント番号などの機密性の高い識別情報をクラスター名に含めないでください。クラスター名はアカウント外に表示される場合があります。転送中の暗号化を有効にすると、DAX は AWS Certificate Manager (ACM) を使用してクラスターエンドポイントの Transport Layer Security (TLS) 証明書をプロビジョニングし、その証明書にはクラスター名が含まれます。

ACM は、発行されたすべてのパブリック証明書をパブリック証明書透明性 (CT) ログに自動的に記録します。また、ブラウザのポリシーの要件により、この記録をオプトアウトすることはできません。その結果、パブリック CT ログには、ACM が証明書を発行または更新した後の TLS が有効な DAX クラスターの名前が含まれます。リソースの命名に関する一般的なガイダンスについては、「DynamoDB におけるデータ保護」を参照してください。CT ログ記録の詳細については、「AWS Certificate Manager ユーザーガイド」の「証明書の透明性ログ記録」を参照してください。

DynamoDB では、個別の DynamoDB リソースでユーザーが実行できるアクションを制限する IAM ポリシーを作成できます。たとえば、特定の DynamoDB テーブルに対して、ユーザーに読み込み専用アクションのみを許可するユーザーロールを作成できます。(詳しくは、Amazon DynamoDB の Identity and Access Management を参照してください)。対して、DAX のセキュリティモデルでは、クラスターのセキュリティおよびクラスターがユーザーに代わって実行する DynamoDB API アクション機能が中心です。

Warning

現在 IAM ロールおよびポリシーを使用して DynamoDB テーブルのデータへのアクセスを制限している場合、DAX を使用することで、これらのポリシーが覆される場合があります。例えば、あるユーザーが、DAX 経由では DynamoDB テーブルにアクセスできても、DynamoDB に直接アクセスすると同じテーブルに対する明示的なアクセスができないという場合があります。詳細については、「Amazon DynamoDB の Identity and Access Management」を参照してください。

DAX では、DynamoDB のデータに対してユーザーレベルの分離を実施していません。代わりに、ユーザーは DAX クラスターにアクセスする際に、そのクラスターの IAM ポリシーのアクセス権限を継承します。そのため、DAX 経由で DynamoDB テーブルにアクセスする際、有効なアクセスコントロールは DAX クラスターの IAM ポリシーのアクセス許可のみです。他のアクセス権限は認識されません。

隔離が必要な場合は、追加の DAX クラスターを作成して、クラスターごとに IAM ポリシーを限定することをお勧めします。たとえば、複数の DAX クラスターを作成し、各クラスターに 1 つのテーブルのみに対してアクセスを許可することもできます。

DAX 用の IAM サービスロール

DAX クラスターを作成するときに、クラスターを IAM ロールと関連付ける必要があります。これは、クラスターのサービスロールと呼ばれます。

DAXCluster01 という名前の新しい DAX クラスターを作成するとします。DAXServiceRole というサービスロールを作成し、そのロールを DAXCluster01 と関連付けます。DAXServiceRole のポリシーは、DAXCluster01 と対話するユーザーの代理で DAXCluster01 が実行できる DynamoDB アクションを定義します。

サービスロールを作成する際に、DAXServiceRole と DAX サービス自体の信頼関係を指定する必要があります。信頼関係は、どのエンティティがロールを引き受けアクセス権限を利用できるかを決定します。以下は、DAXServiceRole の信頼関係のドキュメントの例です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "dax.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

この信頼関係では、DAX クラスターが DAXServiceRole を引き受け、ユーザーに代わって DynamoDB API コールを実行します。

許可される DynamoDB API アクションは、DAXServiceRole にアタッチした IAM ポリシードキュメントに記載されています。以下に、ポリシードキュメントの例を示します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DaxAccessPolicy",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
            ]
        }
    ]
}
```

このポリシーでは、DAX は、DynamoDB テーブルに対して必要な DynamoDB API アクションを実行できます。`dynamodb:DescribeTable` アクションは、DAX がテーブルに関するメタデータを維持するために必要です。その他のアクションは、テーブル内の項目に対して実行される読み込みおよび書き込みアクションです。`Books` という名前のテーブルは us-west-2 リージョンにあり、AWS アカウント ID `123456789012` によって所有されています。

> **Note**
>
> DAX は、クロスサービスアクセス中に混乱した代理問題を防止するメカニズムをサポートしています。詳細については、IAM ユーザーガイド の 混乱した代理問題 を参照してください。

DAX クラスターのアクセスを許可する IAM ポリシー

DAX クラスターを作成したら、ユーザーが DAX クラスターにアクセスできるように、ユーザーにアクセス許可を付与する必要があります。

例えば、Alice というユーザーに DAXCluster01 へのアクセス許可を付与するとします。最初に、受信者がアクセスできる DAX クラスターと DAX API アクションを定義する IAM ポリシー (AliceAccessPolicy) を作成します。次に、このポリシーをユーザー Alice にアタッチして、アクセスを付与します。

次に示すポリシードキュメントは、受取人に DAXCluster01 のフルアクセスを付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "dax:*"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ]
        }
    ]
}
```

ポリシードキュメントは DAX クラスターへのアクセスを許可しますが、DynamoDB に対するアクセス権限は付与しません。(DynamoDB のアクセス権限は、DAX サービスロールによって付与されます。)

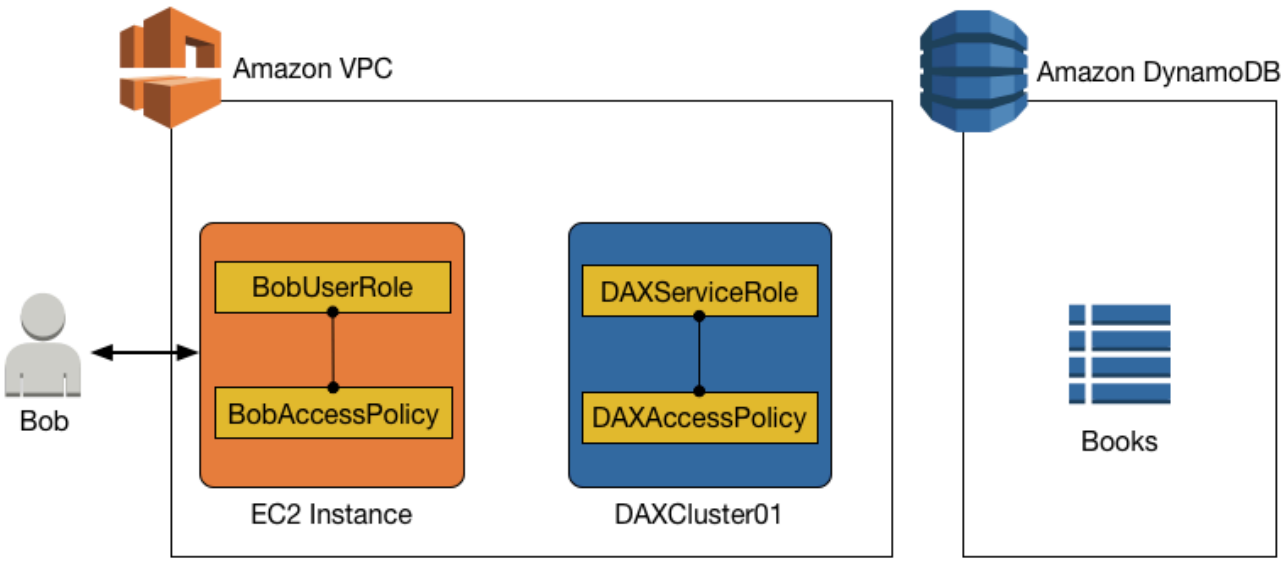
ユーザー Alice 用に、まず上記のポリシードキュメントを持った `AliceAccessPolicy` を作成します。次に、ポリシーを Alice にアタッチします。

> **Note**
>
> ポリシーをユーザーにアタッチする代わりに、IAM ロールにアタッチできます。この方法では、そのロールを引き受けるユーザーのすべてに、ポリシーで定義したアクセス権限があります。

ユーザーポリシーは、DAX サービスロールとともに、受信者が DAX 経由でアクセスできる DynamoDB リソースおよび API アクションを決定します。

導入事例: DynamoDB と DAX にアクセスする

次のシナリオは、DAX で使用する IAM ポリシーについての理解を深めるために役立ちます。(このシナリオは、このセクションの他の部分でも参照されます)。次の図は、このシナリオの大まかな概要を示しています。



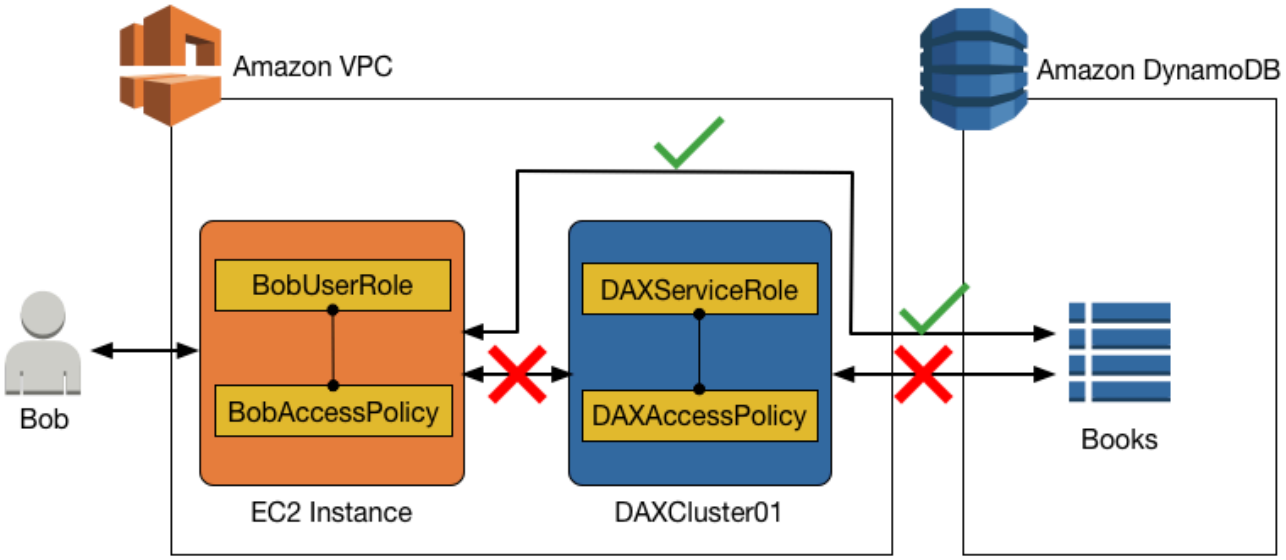
このシナリオには、次のエンティティがあります。

- ユーザー (Bob)。
- IAM ロール (`BobUserRole`)。Bob は実行時にこのロールを引き受けます。

- IAM ポリシー (BobAccessPolicy)。このポリシーは、BobUserRole にアタッチされています。BobAccessPolicy では、BobUserRole がアクセスできる DynamoDB リソースと DAX リソースが定義されます。
- DAX クラスター (DAXCluster01)。
- IAM サービスロール (DAXServiceRole)。このロールにより、DAXCluster01 は DynamoDB にアクセスできます。
- IAM ポリシー (DAXAccessPolicy)。このポリシーは、DAXServiceRole にアタッチされています。DAXAccessPolicy では、DAXCluster01 がアクセスできる DynamoDB API およびリソースが定義されます。
- DynamoDB テーブル (Books)。

BobAccessPolicy と DAXAccessPolicy のポリシーステートメントの組み合わせにより、Bob が Books に対して何ができるかが決まります。たとえば、Books に (DynamoDB エンドポイントを使用して) 直接アクセスできる、(DAX クラスターを使用して) 間接的にアクセスできる、またはその両方などです。Books からデータを読み込む、Books にデータを書き込む、またはその両方が可能な場合もあります。

DynamoDB へのアクセスと DAX へのアクセス防止



DynamoDB テーブルに直接アクセスすることを許可する一方で、DAX クラスターを使用した間接的なアクセスを防止することもできます。DynamoDB への直接アクセスについては、BobUserRole のアクセス権限は (ロールにアタッチされている) BobAccessPolicy によって決まります。

DynamoDB (のみ) への読み込み専用アクセス

Bob は BobUserRole で DynamoDB にアクセスできます。このロールにアタッチされた IAM ポリシー (BobAccessPolicy) は、BobUserRole がアクセスできる DynamoDB テーブルと、BobUserRole が呼び出すことができる API を決定します。

BobAccessPolicy に対して、次のポリシードキュメントを検討してください。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

このドキュメントが BobAccessPolicy にアタッチされていると、BobUserRole は DynamoDB エンドポイントにアクセスし、Books テーブルで読み込み専用のオペレーションを実行できるようになります。

DAX はこのポリシーには含まれていないため、DAX 経由のアクセスは拒否されます。

DynamoDB (のみ) への読み込み/書き込みアクセス

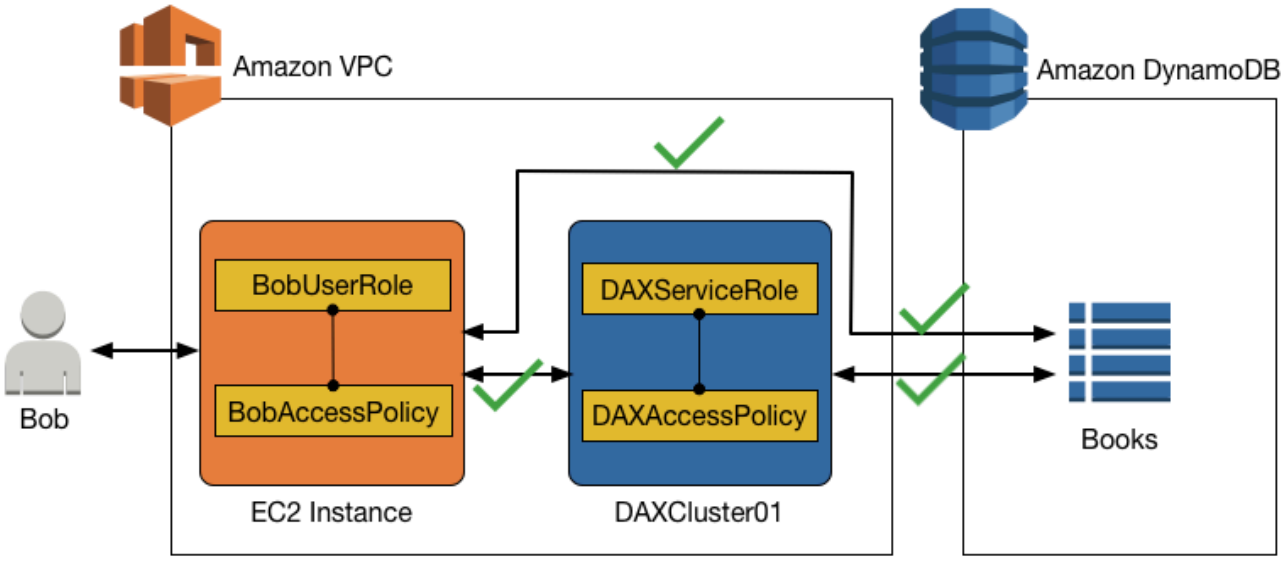
BobUserRole に DynamoDB への読み込み/書き込みアクセスが必要な場合、次のポリシーを使用できます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

DAX はこのポリシーには含まれていないため、DAX 経由のアクセスは拒否されます。

DynamoDB および DAX へのアクセス



DAX クラスターに対するアクセスを許可するには、IAM ポリシーに DAX 固有のアクションを含める必要があります。

次の DAX 固有のアクションは、DynamoDB API の類似した名前のアクションに対応しています。

- `dax:GetItem`
- `dax:BatchGetItem`
- `dax:Query`
- `dax:Scan`
- `dax:PutItem`
- `dax:UpdateItem`
- `dax:DeleteItem`
- `dax:BatchWriteItem`
- `dax:ConditionCheckItem`

`dax:EnclosingOperation` 条件キーについても同様です。

DynamoDB への読み込み専用アクセスと DAX への読み込み専用アクセス

Bob には、DynamoDB と DAX の両方から Books テーブルへの読み込み専用アクセスが必要であるとします。次のポリシー (BobUserRole にアタッチされている) により、このアクセスが付与されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

ポリシーには、DAX アクセス用のステートメント (DAXAccessStmt) と DynamoDBaccess 用の別のステートメント (DynamoDBAccessStmt) があります。これらのステートメントにより、Bob は、GetItem、BatchGetItem、Query、Scan リクエストを DAXCluster01 に送信できるようになります。

ただし、DAXCluster01 のサービスロールにも、DynamoDB の Books テーブルへの読み込み専用のアクセスが必要になります。DAXServiceRole にアタッチされた以下の IAM ポリシーは、この条件を満たします。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

DynamoDB への読み込み/書き込みアクセスと DAX 経由の読み込み専用アクセス

特定のユーザーロールに、DAX 経由での読み込み専用アクセスを許可したまま、DynamoDB テーブルへの読み込み/書き込みアクセスを提供できます。

Bob の場合、BobUserRole の IAM ポリシーは、Books テーブルに対する DynamoDB の読み込み/書き込みアクションを許可しながら、DAXCluster01 を介した読み込み専用アクションもサポートする必要があります。

BobUserRole に対する次のポリシードキュメントの例では、このアクセスが付与されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

さらに、DAXServiceRole には、DAXCluster01 が Books テーブルに対して読み込み専用のアクションを実行できるようにする IAM ポリシーが必要になります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
```

```
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

DynamoDB への読み込み/書き込みアクセスと DAX への読み込み/書き込みアクセス

次に、Bob に DynamoDB から直接、または DAXCluster01 からの間接的な Books テーブルへの読み込み/書き込みアクセスが必要であるとします。BobAccessPolicy にアタッチされている次のポリシードキュメントでは、このアクセスが付与されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {
            "Sid": "DynamoDBAccessStmt",
```

```
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

さらに、DAXServiceRole には、DAXCluster01 が Books テーブルに対する読み込み/書き込みアクションを許可する IAM ポリシーが必要になります。

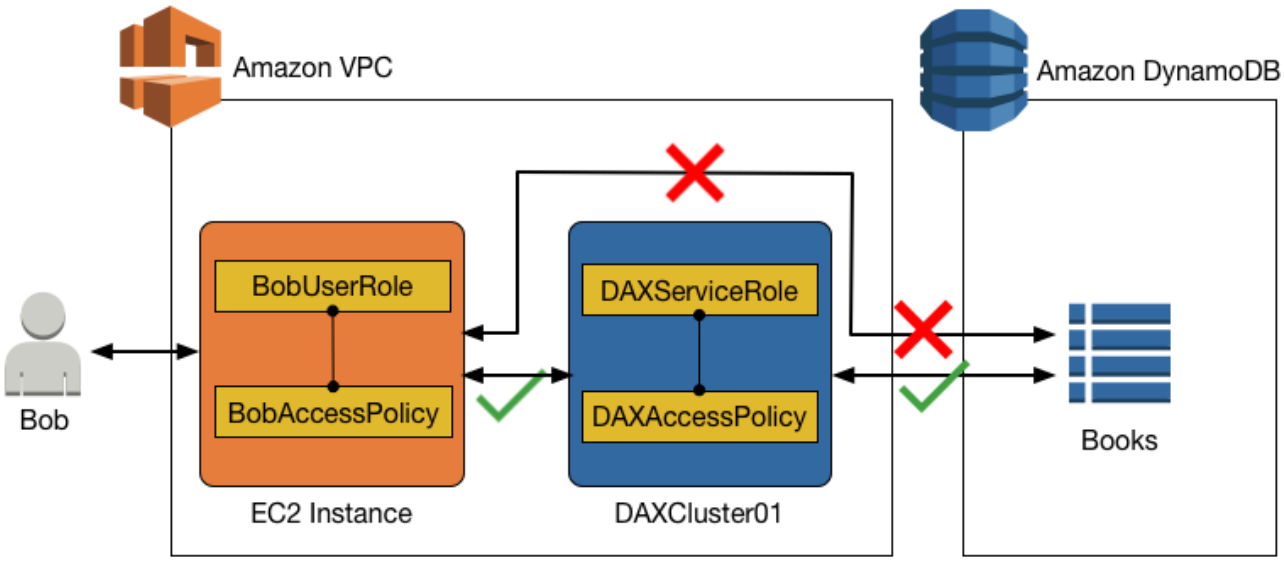
JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
```

```
        }
    ]
}
```

DAX 経由での DynamoDB へのアクセスの許可と DynamoDB への直接アクセスの防止

このシナリオでは、Bob は Books テーブルに DAX 経由でアクセスできますが、DynamoDB の Books テーブルに直接アクセスすることはできません。したがって、Bob が DAX にアクセスすると、他の場合ではアクセスできない DynamoDB テーブルへのアクセスも取得します。DAX サービスロール用の IAM ポリシーを設定する際に、ユーザーアクセスポリシー経由で DAX クラスターへのアクセスを付与されたユーザーには、そのポリシーで指定されたテーブルへのアクセスも付与されることに注意してください。この場合、`BobAccessPolicy` には `DAXAccessPolicy` で指定されたテーブルへのアクセスが付与されます。



現在 IAM ロールおよびポリシーを使用して DynamoDB テーブルおよびデータへのアクセスを制限している場合、DAX を使用することで、これらのポリシーが覆される場合があります。次のポリシーでは、Bob は DynamoDB テーブルに DAX 経由でアクセスできますが、DynamoDB の同じテーブルに明示的に直接アクセスすることはできません。

BobAccessPolicy にアタッチされている次のポリシードキュメント (BobUserRole) では、このアクセスが付与されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        }
    ]
}
```

このアクセスポリシーには、DynamoDB に直接アクセスするアクセス許可がありません。

BobAccessPolicy と次の DAXAccessPolicy により、BobUserRole は、DynamoDB テーブル Books にアクセスできるようになります。ただし、BobUserRole は、Books テーブルに直接アクセスすることはできません。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
```

```
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

この例に示すように、ユーザーアクセスポリシーおよび DAX クラスターアクセスポリシーのアクセスコントロールを設定する場合は、エンドツーエンドのアクセスを完全に理解し、最小権限の原則が守られていることを確認してください。また、DAX クラスターへのアクセスをユーザーに付与することで、以前に確立されたアクセスコントロールポリシーが覆されないことを確認してください。

保管時の DAX 暗号化

Amazon DynamoDB Accelerator (DAX) 保管時の暗号化は、基になるストレージへの不正アクセスからデータを保護することで、データ保護のレイヤーを追加します。組織のポリシー、業界や政府の規制、またはコンプライアンス要件によって、データを保護するために保管時の暗号化の使用が求められる場合があります。暗号化を使用すると、クラウドにデプロイされたアプリケーションデータの安全性を向上できます。

保管時の暗号化を使用して、DAX によってディスクに保持されるデータが 256 ビット Advanced Encryption Standard (AES-256 暗号化とも呼ばれる) を使用して暗号化されます。DAX は、プライマリノードからリードレプリカへの変更の伝播の一部として、データをディスクに書き込みます。

DAX 保管時の暗号化には、クラスターの暗号化に使用される単一サービスデフォルトキーを管理する AWS Key Management Service (AWS KMS) を自動的に統合します。DAX クラスターを作成するときにサービスデフォルトキーが存在しない場合、AWS KMS は自動的に新しい AWS マネージドキーを作成します。このキーは、この先作成するクラスターの暗号化に使用されます。AWS KMS

は、安全で可用性の高いハードウェアとソフトウェアを組み合わせて、クラウド向けに拡張されたキー管理システムを提供します。

データが暗号化されると、DAX はパフォーマンスの影響を最小限に抑えながら、データの復号を透過的に処理します。暗号化を使用するためにアプリケーションを変更する必要はありません。

> **Note**
>
> DAX はすべての DAX オペレーションについて AWS KMS を呼び出すことはしません。DAX は、クラスター起動時にのみキーを使用します。アクセス権限が取り消された場合でも、DAX はクラスターがシャットダウンされるまで引き続きデータにアクセスできます。顧客が指定した AWS KMS キーはサポートされません。

保管時の DAX 暗号化は、次のクラスターノードタイプに使用できます。

ファミリー	ノードの種類
メモリ最適化 (R4、R5、R7)	dax.r4.large
	dax.r4.xlarge
	dax.r4.2xlarge
	dax.r4.4xlarge
	dax.r4.8xlarge
	dax.r4.16xlarge
	dax.r5.large
	dax.r5.xlarge
	dax.r5.2xlarge
	dax.r5.4xlarge
	dax.r5.8xlarge
	dax.r5.12xlarge

ファミリー	ノードの種類
	dax.r5.16xlarge
	dax.r5.24xlarge
	dax.r7i.large
	dax.r7i.xlarge
	dax.r7i.2xlarge
	dax.r7i.4xlarge
	dax.r7i.8xlarge
	dax.r7i.12xlarge
	dax.r7i.16xlarge
	dax.r7i.24xlarge
汎用 (T2)	dax.t2.small
	dax.t2.medium
汎用 (T3)	dax.t3.small
	dax.t3.medium

> **Important**
>
> 保管時の DAX 暗号化は `dax.r3.*` ノードタイプではサポートされません。

クラスターが作成された後は、保管時の暗号化を有効または無効にすることはできません。作成時に有効でなかった場合には、保管時の暗号化を有効にするためにクラスターを再度作成する必要があります。

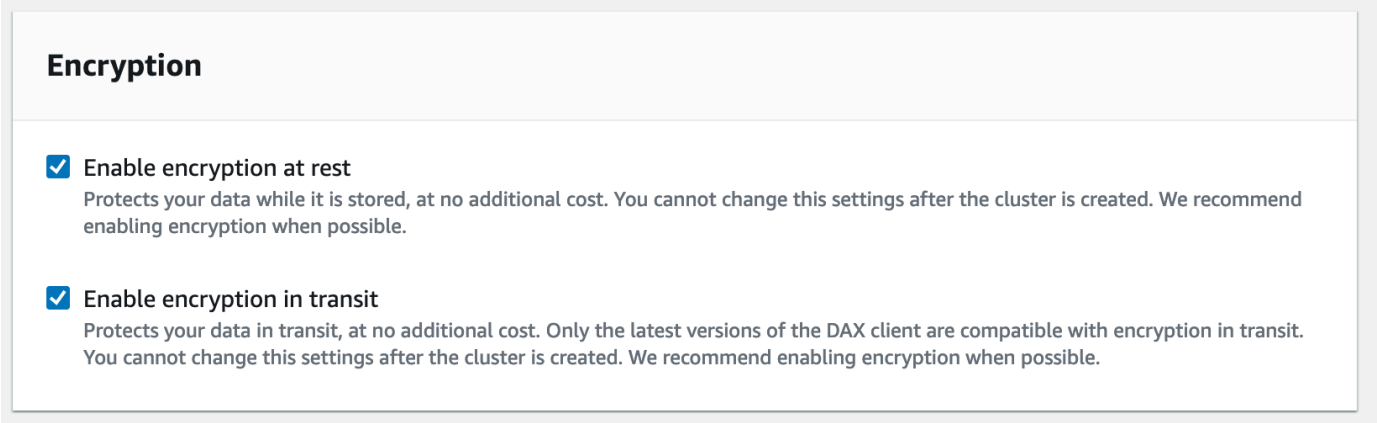
DAX 保管時の暗号化は、追加コストなしで提供されます (AWS KMS 暗号化キーの利用料金適用)。料金に関する詳細については、「Amazon DynamoDB の料金表」を参照してください。

AWS マネジメントコンソール を使用した保管時の暗号化の有効化

コンソールを使用して、次の手順に従ってテーブルに対する保管時の DAX 暗号化を有効にします。

保管時の DAX 暗号化を有効にするには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインの、[DAX] の [クラスター] を選択します。
3. [Create cluster (クラスターの作成)] を選択します。
4. [クラスター名] に、クラスターの短縮名を入力します。クラスターのすべてのノードに [ノードタイプ] を選択して、クラスターサイズに、**3** ノードを使用します。
5. [暗号化] で、[暗号化の有効化] が選択されていることを確認します。



6. IAM ロール、サブネットグループ、セキュリティグループ、およびクラスター設定を選択した後、[クラスターの起動] を選択します。

クラスターが暗号化されていることを確認するには、[クラスター] ペインでクラスターの詳細を参照します。暗号化が [有効] である必要があります。

転送時の DAX 暗号化

Amazon DynamoDB Accelerator (DAX) は、アプリケーションと DAX クラスター間でデータを転送する際の暗号化をサポートするため、厳しい暗号化要件を持つアプリケーションでも DAX を使用できます。

転送中に暗号化を選択するかどうかにかかわらず、アプリケーションと DAX クラスター間のトラフィックは Amazon VPC に残ります。このトラフィックは、クラスターのノードに添付されている

VPC 内のプライベート IP を使用して、Elastic Network Interface にルーティングされます。信頼境界として VPC を使用すると、セキュリティグループ、ネットワーク ACL を使用したサブネットセグメンテーション、VPC フローログなどのスタンダードツールを使用することで、データのセキュリティを大幅に制御できます。転送中の DAX 暗号化では、このベースラインレベルの機密性が高まり、アプリケーションとクラスター間のすべてのリクエストとレスポンスがトランスポートレベルセキュリティ (TLS) によって暗号化され、クラスターへの接続はクラスター x509 証明書の検証によって認証されます。DAX によってディスクに書き込まれたデータも、DAX クラスターを作成する際に保管時の暗号化 を選択すれば暗号化されます。

DAXなら、転送中の暗号化を簡単に使用できます。新しいクラスターを作成するときにこのオプションを選択するだけです。そして最近のバージョンの DAX クライアントアプリケーションに保管します。転送中に暗号化を使用するクラスターは、暗号化されていないトラフィックをサポートしないため、アプリケーションを誤って設定したり、暗号化をバイパスしたりすることはありません。DAX クライアントは、接続を確立するときに、クラスターの x509 証明書を使用してクラスターのアイデンティティを認証し、DAX リクエストが意図した場所に確実に送信されるようにします。DAX クラスターの作成方法はすべて (AWS マネジメントコンソール、AWS CLI、すべての SDK、CloudFormation)、転送中の暗号化をサポートします。

転送中の暗号化は、既存の DAX クラスターでは有効にできません。既存の DAX アプリケーションで転送中の暗号化を使用するには、転送中の暗号化を有効にした新しいクラスターを作成し、アプリケーションのトラフィックをそのクラスターに移行してから、古いクラスターを削除します。

DAX のサービスにリンクされた IAM ロールの使用

この Amazon DynamoDB Accelerator (DAX) では、AWS Identity and Access Management (IAM) のサービスにリンクされたロールが使用されます。サービスにリンクされたロールは、DAX に直接リンクされた一意のタイプの IAM ロールです。サービスにリンクされたロールは、DAX で事前定義されており、ユーザーの代わりに必要なサービスから他の AWS のサービスを呼び出すための、すべてのアクセス許可が付与されています。

サービスにリンクされたロールを使用することで、必要なアクセス許可を手動で追加する必要がなくなるため、DAX の設定が簡単になります。サービスにリンクされたロールのアクセス許可は DAX により定義され、他に特別な定義がある場合を除き、DAX のみがそのロールを引き受けることができます。定義されたアクセス権限には、信頼ポリシーとアクセス権限ポリシーが含まれます。このアクセス許可ポリシーを他の IAM エンティティにアタッチすることはできません。

ロールを削除するには、まず関連リソースを削除します。これにより、DAX リソースに対するアクセス権限が意図せずに削除されることがなくなり、リソースを保護できます。

サービスにリンクされたロールをサポートしているその他のサービスの詳細については、IAM ユーザーガイドの「IAM と連携する AWS のサービス」を参照してください。[サービスにリンクされたロール] 列が [はい] になっているサービスを見つけます。該当するサービスにリンクされたロールに関するドキュメントを表示するには、[あり] リンクを選択します。

トピック

- DAX のサービスにリンクされたロールにおけるアクセス許可
- DAX でサービスにリンクされたロールの作成
- DAX のサービスにリンクされたロールの編集
- DAX でサービスにリンクされたロールの削除

DAX のサービスにリンクされたロールにおけるアクセス許可

DAX では、AWSServiceRoleForDAX という名前のサービスにリンクされたロールを使用します。このロールにより、ユーザーの DAX クラスターに代わってサービスを呼び出すことを、DAX に許可します。

> **Important**
>
> サービスにリンクされたロール AWSServiceRoleForDAX により、DAX クラスターのセットアップと保守が簡単になります。ただし、このロールを使用する前に、各クラスターに DynamoDB へのアクセス許可を付与する必要があります。詳細については、「」を参照してくださいDAX のアクセスコントロール

AWSServiceRoleForDAX サービスリンクロールは、ロールの引き受けについて以下のサービスを信頼します。

- dax.amazonaws.com

ロールのアクセス許可ポリシーは、DAX に対し、指定したリソースで以下のアクションを実行することを許可します。

- ec2 でのアクション:
  - AuthorizeSecurityGroupIngress
  - CreateNetworkInterface

- CreateSecurityGroup
- DeleteNetworkInterface
- DeleteSecurityGroup
- DescribeAvailabilityZones
- DescribeNetworkInterfaces
- DescribeSecurityGroups
- DescribeSubnets
- DescribeVpcs
- ModifyNetworkInterfaceAttribute
- RevokeSecurityGroupIngress

サービスにリンクされたロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) に許可するには、アクセス許可を設定する必要があります。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの許可」を参照してください。

IAM エンティティが AWSServiceRoleForDAX サービスにリンクされたロールを作成するには

以下のポリシーステートメントを IAM エンティティのアクセス許可に追加します。

```
{
    "Effect": "Allow",
    "Action": [
        "iam:CreateServiceLinkedRole"
    ],
    "Resource": "*",
    "Condition": {"StringLike": {"iam:AWSServiceName": "dax.amazonaws.com"}}
}
```

DAX でサービスにリンクされたロールの作成

サービスにリンクされたロールを手動で作成する必要はありません。サービスにリンクされたロールは、クラスターの作成時に DAX により自動的に作成されます。

> **Important**
>
> 2018 年 2 月 28日より前に DAX サービスを使用していた場合は、サービスにリンクされたロールのサポートが開始された時点で、AWSServiceRoleForDAX ロールが DAX によって

アカウントに作成されています。詳細については、IAM ユーザーガイドの「AWS アカウントに新しいロールが表示される」を参照してください。

サービスにリンクされたこのロールを削除したが、再作成する必要がある場合は、同じプロセスで、アカウントにロールを再作成することができます。インスタンスまたはクラスターを作成するたびに、(ユーザーに代わり) DAX により、サービスにリンクされたロールが再作成されます。

DAX のサービスにリンクされたロールの編集

DAX では、サービスにリンクされたロール AWSServiceRoleForDAX を編集することはできません。サービスにリンクされたロールを作成すると、多くのエンティティによってロールが参照される可能性があるため、ロール名を変更することはできません。ただし、IAM を使用してロールの説明を編集することはできます。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの編集」を参照してください。

DAX でサービスにリンクされたロールの削除

サービスリンクロールを必要とする機能またはサービスが不要になった場合には、そのロールを削除することをお勧めします。そうすることで、積極的にモニタリングまたは保守されていない未使用のエンティティを排除できます。ただし、サービスにリンクされたロールを削除する前に、すべての DAX クラスターを削除する必要があります。

サービスにリンクされたロールのクリーンアップ

IAM を使用してサービスにリンクされたロールを削除するには、まずそのロールにアクティブなセッションがないことを確認し、そのロールで使用されているリソースをすべて削除する必要があります。

サービスにリンクされたロールにアクティブなセッションがあるかどうかを、IAM コンソールで確認するには

1. AWS マネジメントコンソール にサインインして、IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. IAM コンソールのナビゲーションペインで [ロール] を選択します。次に、AWSServiceRoleForDAX ロールの名前 (チェックボックスではありません) を選択します。
3. 選択したロールの [Summary] ページで、[Access Advisor] タブを選択します。
4. [Access Advisor] タブで、サービスにリンクされたロールの最新のアクティビティを確認します。

> **Note**
>
> DAX で AWSServiceRoleForDAX ロールが使用されているかどうかが不明な場合は、このロールの削除を試みることができます。サービスがロールを使用している場合、削除は失敗し、ロールが使用されているリージョンを表示できます。ロールが使用されている場合は、そのロールを削除する前に DAX クラスターを削除する必要があります。サービスにリンクされたロールのセッションを取り消すことはできません。

AWSServiceRoleForDAX ロールを削除する場合は、最初にすべての DAX クラスターを削除する必要があります。

すべての DAX クラスターの削除

以下のいずれかの手順を使用して、各 DAX クラスターを削除します。

DAX クラスターを削除するには (コンソール)

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインの、[DAX] で、[クラスター]を選択します。
3. [アクション] を選択して、[削除] を選択します。
4. [Delete cluster confirmation (クラスターの削除の確認)] ボックスで、[削除] を選択します。

DAX クラスターを削除するには (AWS CLI)

AWS CLI コマンドリファレンスの「delete-cluster」を参照してください。

DAX クラスターを削除するには (API)

「Amazon DynamoDB API リファレンス」の「DeleteCluster」を参照してください。

サービスにリンクされたロールの削除

IAM を使用して、サービスにリンクされたロールを手動で削除するには

AWSServiceRoleForDAX サービスにリンクされたロールを削除するには、IAM コンソール、IAM CLI、または IAM API を使用します。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールの削除」を参照してください。

AWS アカウント間での DAX へのアクセス

ある AWS アカウント (アカウント A) で DynamoDB アクセラレーター (DAX) クラスターが実行されていて、別の AWS アカウント (アカウント B) の Amazon Elastic Compute Cloud (Amazon EC2) インスタンスから DAX クラスターにアクセスできる必要があるとします。このチュートリアルでは、アカウント B の IAM ロールを持つ EC2 インスタンスをアカウント B で起動することでこれを実現します。次に、EC2 インスタンスの一時的なセキュリティ認証情報を使用して、アカウント A から IAM ロールを引き受けます。最後に、IAM ロールを引き受けることで得られた一時的なセキュリティ認証情報を使用し、アカウント A の DAX クラスターに対して Amazon VPC ピアリング接続を介してアプリケーション呼び出しを行います。これらのタスクを実行するには、両方の AWS アカウントで管理アクセスが必要です。

> **Important**
>
> DAX クラスターに、別のアカウントから DynamoDB テーブルにアクセスさせることはできません。

トピック

- IAM のセットアップ
- VPC をセットアップする
- クロスアカウントアクセスを許可するように DAX クライアントを変更します

IAM のセットアップ

1. 次のコンテンツを含む `AssumeDaxRoleTrust.json` という名前のテキストファイルを作成します。これにより、Amazon EC2 がお客様に代わって作業できるようになります。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "ec2.amazonaws.com"
```

```
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

2. アカウント B で、インスタンスの起動時に Amazon EC2 が使用できるロールを作成します。

```
aws iam create-role \
    --role-name AssumeDaxRole \
    --assume-role-policy-document file://AssumeDaxRoleTrust.json
```

3. 次のコンテンツを含む AssumeDaxRolePolicy.json という名前のテキストファイルを作成します。これにより、アカウント B の EC2 インスタンスで実行されているコードが、アカウント A の IAM ロールを引き受けることができるようになります。*accountA* は、アカウント A の実際の ID に置き換えます。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Resource": "arn:aws:iam::111122223333:role/DaxCrossAccountRole"
        }
    ]
}
```

4. 作成したロールにそのポリシーを追加します。

```
aws iam put-role-policy \
    --role-name AssumeDaxRole \
    --policy-name AssumeDaxRolePolicy \
    --policy-document file://AssumeDaxRolePolicy.json
```

5. インスタンスプロファイルを作成して、インスタンスがロールを使用できるようにします。

```
aws iam create-instance-profile \
    --instance-profile-name AssumeDaxInstanceProfile
```

6. ロールをインスタンスプロファイルに関連付けます。

```
aws iam add-role-to-instance-profile \
    --instance-profile-name AssumeDaxInstanceProfile \
    --role-name AssumeDaxRole
```

7. 次の内容を含む DaxCrossAccountRoleTrust.json という名前のテキストファイルを作成します。これにより、アカウント B がアカウント A ロールを引き受けることができるようになります。*accountB* は、アカウント B の実際の ID に置き換えます。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/AssumeDaxRole"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

8. アカウント A で、アカウント B が引き受けることができるロールを作成します。

```
aws iam create-role \
    --role-name DaxCrossAccountRole \
    --assume-role-policy-document file://DaxCrossAccountRoleTrust.json
```

9. DAX クラスターへのアクセスを許可する DaxCrossAccountPolicy.json という名前のテキストファイルを作成します。*dax-cluster-arn* は、DAX クラスターの正しい Amazon リソースネーム (ARN) に置き換えます。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-east-1:111122223333:cache/dax-
cluster-name"
        }
    ]
}
```

10. アカウント A で、ロールにポリシーを追加します。

```
aws iam put-role-policy \
    --role-name DaxCrossAccountRole \
    --policy-name DaxCrossAccountPolicy \
    --policy-document file://DaxCrossAccountPolicy.json
```

VPC をセットアップする

1. アカウント A の DAX クラスターのサブネットグループを見つけます。*cluster-name* は、アカウント B がアクセスする必要のある DAX クラスターの名前に置き換えます。

```
aws dax describe-clusters \
    --cluster-name cluster-name
    --query 'Clusters[0].SubnetGroup'
```

2. その *subnet-group* を使用して、クラスターの VPC を見つけます。

```
aws dax describe-subnet-groups \
    --subnet-group-name subnet-group \
    --query 'SubnetGroups[0].VpcId'
```

3. その *vpc-id* を使用して、VPC の CIDR を見つけます。

```
aws ec2 describe-vpcs \
    --vpc vpc-id \
    --query 'Vpcs[0].CidrBlock'
```

4. アカウント B から、前のステップで見つかったものとは異なる重複しない CIDR を使用して VPC を作成します。次に、少なくとも 1 つのサブネットを作成します。AWS マネジメントコンソール の VPC 作成ウィザードまたは AWS CLI のいずれかを使用できます。
5. アカウント B から、「VPC ピアリング接続の作成と使用」の説明に従って、アカウント A VPC へのピアリング接続をリクエストします。アカウント A から接続を受け入れます。
6. アカウント B から、新しい VPC のルーティングテーブルを見つけます。*vpc-id* は、アカウント B で作成した VPC の ID に置き換えます。

```
aws ec2 describe-route-tables \
    --filters 'Name=vpc-id,Values=vpc-id' \
    --query 'RouteTables[0].RouteTableId'
```

7. アカウント A の CIDR 宛てのトラフィックを VPC ピアリング接続に送信するルートを追加します。必ず、各*############*は、アカウントの正しい値に置き換えます。

```
aws ec2 create-route \
    --route-table-id accountB-route-table-id \
    --destination-cidr accountA-vpc-cidr \
    --vpc-peering-connection-id peering-connection-id
```

8. アカウント A から、以前に見つけた *vpc-id* を使用して DAX クラスターのルートテーブルを見つけます。

```
aws ec2 describe-route-tables \
    --filters 'Name=vpc-id, Values=accountA-vpc-id' \
    --query 'RouteTables[0].RouteTableId'
```

9. アカウント A から、アカウント B の CIDR 宛てのトラフィックを VPC ピアリング接続に送信するルートを追加します。各*############*は、アカウントの正しい値に置き換えます。

```
aws ec2 create-route \
    --route-table-id accountA-route-table-id \
    --destination-cidr accountB-vpc-cidr \
    --vpc-peering-connection-id peering-connection-id
```

10. アカウント B から、前のステップで作成した VPC で EC2 インスタンスを起動します。それに AssumeDaxInstanceProfile を指定します。AWS マネジメントコンソール または AWS CLI いずれかの Launch Wizard を使用できます。インスタンスのセキュリティグループをメモしておきます。
11. アカウント A から、DAX クラスターが使用するセキュリティグループを見つけます。*cluster-name* を DAX クラスターの名前に置き換えることを記憶します。

```
aws dax describe-clusters \
    --cluster-name cluster-name \
    --query 'Clusters[0].SecurityGroups[0].SecurityGroupIdentifier'
```

12. アカウント B で作成した EC2 インスタンスのセキュリティグループからのインバウンドトラフィックを許可するように、DAX クラスターのセキュリティグループを更新します。*#######*は、アカウントの正しい値に置き換えることを記憶します。

```
aws ec2 authorize-security-group-ingress \
    --group-id accountA-security-group-id \
    --protocol tcp \
    --port 8111 \
    --source-group accountB-security-group-id \
    --group-owner accountB-id
```

この時点で、アカウント B の EC2 インスタンスのアプリケーションは、インスタンスプロファイルを使用して arn:aws:iam::*accountA-id*:role/DaxCrossAccountRole ロールを引き受け、DAX クラスターを使用できます。

クロスアカウントアクセスを許可するように DAX クライアントを変更します

> **Note**
>
> AWS Security Token Service (AWS STS) 認証情報は一時的な認証情報です。自動的に更新を処理するクライアントもあれば、認証情報を更新するための追加のロジックを必要とするクライアントもあります。該当するマニュアルの指示に従うことをお勧めします。

Java

このセクションでは、既存の DAX クライアントコードを変更して、クロスアカウント DAX アクセスを許可する方法について説明します。DAX クライアントコードがまだない場合は、「Java および DAX」チュートリアルで実際のコード例を見つけることができます。

1. 次のインポートを追加します。

```
import com.amazonaws.auth.STSAssumeRoleSessionCredentialsProvider;
import com.amazonaws.services.securitytoken.AWSSecurityTokenService;
import
 com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClientBuilder;
```

2. AWS STS から認証情報プロバイダを取得し、DAX クライアントオブジェクトを作成します。必ず、各############は、アカウントの正しい値に置き換えます。

```
AWSSecurityTokenService awsSecurityTokenService =
 AWSSecurityTokenServiceClientBuilder
     .standard()
     .withRegion(region)
     .build();

STSAssumeRoleSessionCredentialsProvider credentials =  new
 STSAssumeRoleSessionCredentialsProvider.Builder("arn:aws:iam::accountA:role/
RoleName", "TryDax")
     .withStsClient(awsSecurityTokenService)
     .build();

DynamoDB client = AmazonDaxClientBuilder.standard()
    .withRegion(region)
    .withEndpointConfiguration(dax_endpoint)
    .withCredentials(credentials)
    .build();
```

.NET

このセクションでは、既存の DAX クライアントコードを変更して、クロスアカウント DAX アクセスを許可する方法について説明します。DAX クライアントコードがまだない場合は、「.NET および DAX」チュートリアルで実際のコード例を見つけることができます。

1. AWSSDK.SecurityToken NuGet パッケージをソリューションに追加します。

```
<PackageReference Include="AWSSDK.SecurityToken" Version="latest version" />
```

2. SecurityToken および SecurityToken.Model パッケージを使用します。

```
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;
```

3. AmazonSimpleTokenService から一時的な認証情報を取得し、ClusterDaxClient オブジェクトを作成します。必ず、各############は、アカウントの正しい値に置き換えます。

```
IAmazonSecurityTokenService sts = new AmazonSecurityTokenServiceClient();

var assumeRoleResponse = sts.AssumeRole(new AssumeRoleRequest
{
    RoleArn = "arn:aws:iam::accountA:role/RoleName",
                RoleSessionName = "TryDax"
});

Credentials credentials = assumeRoleResponse.Credentials;

var clientConfig = new DaxClientConfig(dax_endpoint, port)
{
    AwsCredentials = assumeRoleResponse.Credentials

};

var client = new ClusterDaxClient(clientConfig);
```

Go

このセクションでは、既存の DAX クライアントコードを変更して、クロスアカウント DAX アクセスを許可する方法について説明します。DAX クライアントコードがまだない場合は、GitHub で実際のコード例を見つけることができます。

1. AWS STS およびセッションパッケージをインポートします。

```
import (
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sts"
```

```
    "github.com/aws/aws-sdk-go/aws/credentials/stscreds"
)
```

2. AmazonSimpleTokenService から一時的な認証情報を取得し、DAX クライアントオブジェクトを作成します。必ず、各############は、アカウントの正しい値に置き換えます。

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region)},
)
if err != nil {
    return nil, err
}

stsClient := sts.New(sess)
arp := &stscreds.AssumeRoleProvider{
                Duration:     900 * time.Second,
                ExpiryWindow: 10 * time.Second,
                RoleARN:      "arn:aws:iam::accountA:role/role_name",
                Client:       stsClient,
                RoleSessionName: "session_name",
        }cfg := dax.DefaultConfig()

cfg.HostPorts = []string{dax_endpoint}
cfg.Region = region
cfg.Credentials = credentials.NewCredentials(arp)
daxClient := dax.New(cfg)
```

Python

このセクションでは、既存の DAX クライアントコードを変更して、クロスアカウント DAX アクセスを許可する方法について説明します。DAX クライアントコードがまだない場合は、「Python および DAX」チュートリアルで実際のコード例を見つけることができます。

1. boto3 をインポートします。

```
import boto3
```

2. sts から一時的な認証情報を取得し、AmazonDaxClient オブジェクトを作成します。必ず、各############は、アカウントの正しい値に置き換えます。

```
sts = boto3.client('sts')
stsresponse =
 sts.assume_role(RoleArn='arn:aws:iam::accountA:role/
RoleName',RoleSessionName='tryDax')
credentials = botocore.session.get_session()['Credentials']

dax = amazondax.AmazonDaxClient(session, region_name=region,
 endpoints=[dax_endpoint], aws_access_key_id=credentials['AccessKeyId'],
 aws_secret_access_key=credentials['SecretAccessKey'],
 aws_session_token=credentials['SessionToken'])
client = dax
```

Node.js

このセクションでは、既存の DAX クライアントコードを変更して、クロスアカウント DAX アクセスを許可する方法について説明します。DAX クライアントコードがまだない場合は、「Node.js および DAX」チュートリアルで実際のコード例を見つけることができます。必ず、各############は、アカウントの正しい値に置き換えます。

```
const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');
const region = 'region';
const endpoints = [daxEndpoint1, ...];

const getCredentials = async() => {
  return new Promise((resolve, reject) => {
    const sts = new AWS.STS();
    const roleParams = {
      RoleArn: 'arn:aws:iam::accountA:role/RoleName',
      RoleSessionName: 'tryDax',
    };
    sts.assumeRole(roleParams, (err, session) => {
      if(err) {
        reject(err);
      } else {
        resolve({
          accessKeyId: session.Credentials.AccessKeyId,
          secretAccessKey: session.Credentials.SecretAccessKey,
          sessionToken: session.Credentials.SessionToken,
        });
      }
```

```
    });
  });
};

const createDaxClient = async() => {
  const credentials = await getCredentials();
  const daxClient = new AmazonDaxClient({endpoints: endpoints, region: region,
 accessKeyId: credentials.accessKeyId, secretAccessKey: credentials.secretAccessKey,
 sessionToken: credentials.sessionToken});
  return new AWS.DynamoDB.DocumentClient({service: daxClient});
};

createDaxClient().then((client) => {
  client.get(...);
  ...
}).catch((error) => {
  console.log('Caught an error: ' + error);
});
```

DAX クラスターサイジングガイド

このガイドでは、Amazon Amazon DynamoDB Accelerator (DAX) クラスターの適切なサイズとノードタイプを、アプリケーションのために選択する際のアドバイスを提供します。以下の手順では、アプリケーションでの DAX トラフィックの見積もりと、クラスター設定の選択、およびそのテストを行うためのチュートリアルが得られます。

既に DAX クラスターが存在し、そのクラスターに適切な数とサイズのノードが備わっているどうかを評価したい場合は、「DAX クラスターのスケーリング」を参照してください。

トピック

- 概要
- トラフィックの見積もり
- 負荷テスト

概要

新しい DAX クラスターを作成する場合も、既存のクラスターをメンテナンスしている場合も、ワークロードに合わせてクラスターを適切にスケーリングすることが重要です。時間が経過し、アプリ

ケーションのワークロードが変化するにつれて、スケーリングに関する決定を定期的に再検討して、それらが引き続き適切であることを確認する必要があります。

通常、このプロセスは以下の手順に従います。

1. トラフィックの見積もり。このステップでは、アプリケーションから DAX に送信されるトラフィックのボリュームや、そのトラフィックの性質 (そのオペレーションが読み込みか書き込みか)、および予想されるキャッシュヒット率についての推定を行います。
2. 負荷テスト: この手順では、クラスターを作成し、前のステップからの見積もりをミラーリングするクラスターにトラフィックを送信します。適切なクラスター設定が見つかるまで、この手順を繰り返します。
3. 本番稼働モニタリング。実稼働のアプリケーションで DAX を使用する場合には、クラスターをモニタリングしながら、時間的なワークロードの変動に応じてそのクラスターが正しくスケーリングされていることを、継続的に検証する必要があります。

トラフィックの見積もり

一般的な DAX ワークロードの特性は、以下の 3 つの主要素により表されます。

- キャッシュヒットレート
- 1 秒あたりの読み込みキャパシティーユニット (RCU)
- 1 秒あたりの書き込みキャパシティーユニット (WCU)

キャッシュヒットレートの見積もり

既存の DAX クラスターについては、`ItemCacheHits` および `ItemCacheMisses` の Amazon CloudWatch メトリクスを使用することでキャッシュヒット率を確認できます。キャッシュヒット率は `ItemCacheHits / (ItemCacheHits + ItemCacheMisses)` に等しくなります。ワークロードに `Query` または `Scan` オペレーションが含まれている場合は、`QueryCacheHits`、`QueryCacheMisses`、`ScanCacheHits`、および `ScanCacheMisses` の各メトリクスも確認する必要があります。キャッシュヒット率はアプリケーションごとに異なり、クラスターの有効期限 (TTL) 設定から大きな影響を受けます。DAX を使用するアプリケーションでの標準的なキャッシュヒット率は、85 ~ 95% です。

読み込みキャパシティユニットと書き込みキャパシティユニットの見積もり

アプリケーション用の DynamoDB テーブルが既に存在する場合は、ConsumedReadCapacityUnits および ConsumedWriteCapacityUnits の CloudWatch メトリクスを確認します。Sum 統計を使用して、期間の秒数で除算します。

DAX クラスターも既に存在する場合、DynamoDB の ConsumedReadCapacityUnits メトリクスでは、キャッシュミスのみが計算されることに注意してください。DAX クラスターで処理される 1 秒あたりの読み込みキャパシティーユニットの数を把握するには、このメトリクスからの数値をキャッシュミス率 (つまり、1 - キャッシュヒット率) で割ります。

まだ DynamoDB テーブルを用意していない場合は、「読み込みおよび書き込みのキャパシティユニット」のドキュメントを参照して、アプリケーションの推定リクエスト率、リクエストあたりのアクセスした項目数、および項目サイズに基づいて、トラフィックを見積もります。

トラフィックの見積もりを行う際、将来の増大と予想されるピークと予想外のピークを計画して、トラフィックの増加に十分なヘッドルームがクラスターにあることを確認します。

負荷テスト

トラフィックを見積もった後の次の手順は、負荷をかけてクラスター設定をテストすることです。

1. 最初の負荷テストでは、dax.r4.large ノードタイプ、最もコストの低い固定パフォーマンス、メモリ最適化ノードタイプから開始することをお勧めします。
2. 耐障害性の高いクラスターには、3 つのアベイラビリティーゾーンにまたがって分散される少なくとも 3 つのノードが必要です。この場合、アベイラビリティーゾーンが使用できなくなると、有効なアベイラビリティーゾーンの数が 3 分の 1 減少します。最初の負荷テストでは、3 ノードクラスター内の 1 つのアベイラビリティーゾーンの障害をシミュレートする 2 ノードクラスターから開始することをお勧めします。
3. 負荷テスト期間中、持続的なトラフィック (前のステップで見積もったもの) をテストクラスターに駆動します。
4. 負荷テスト中にクラスターのパフォーマンスをモニタリングします。

理想的には、負荷テスト中に駆動するトラフィックプロファイルは、アプリケーションの実際のトラフィックと可能な限り類似している必要があります。これには、オペレーションの分散 (GetItem 70%、Query 25%、PutItem 5% など)、各オペレーションのリクエスト率、リクエストごとにアクセスされた項目数、項目サイズの分散が含まれます。アプリケーションの予想キャッシュヒット率と

同様のキャッシュヒット率を達成するには、テストトラフィックでのキーの分散に細心の注意を払います。

> **Note**
>
> T2 ノードタイプ (`dax.t2.small` および `dax.t2.medium`) をロードテストする際は注意してください。T2 ノードタイプは、ノードの CPU クレジットバランスに応じて時間の経過とともに変化する、バースト可能な CPU パフォーマンスを提供します。T2 ノードで実行されている DAX クラスターのオペレーションが正常に見える場合でも、いずれかのノードがそのインスタンスのパフォーマンスベースラインを上回ってバーストする場合には、そのノードは蓄積 CPU クレジット残高を消費しています。クレジットで実行中のノードが少ない場合、パフォーマンスは次第にベースラインのパフォーマンスレベルに下がります。

負荷テストにより、DAX クラスターをモニタリングして、そこで使用しているノードタイプが適切なノードタイプであるかどうかを判断します。さらに、負荷テスト中にリクエストレートとキャッシュヒットレートをモニタリングして、テストインフラストラクチャが意図したトラフィック量を実際に駆動していることを確認する必要があります。

選択したクラスターインスタンスタイプのネットワークバイト消費量に注意する必要があります。Amazon EC2 インスタンスで使用可能なベースライン帯域幅を超えると、クラスターがアプリケーションのワークロードに耐えられなくなり、スケーリングする必要が出てくるかもしれません。

負荷テストにより、選択したクラスター設定ではアプリケーションのワークロードを維持できないことが示された場合、特にクラスター内のプライマリノードでの高い CPU 使用率や、高い削除率、あるいは高いキャッシュメモリー使用率が見られる場合には、より大きなノードタイプに切り替える必要があります。ヒット率が一貫して高く、書き込みトラフィックに対する読み込みトラフィックの比率が高い場合は、ノードをクラスターに追加することを検討してください。どのような場合により大きなノードタイプを使用するか (垂直スケーリング)、ノードを追加するか (水平スケーリング) の詳細なガイダンスについては、「DAX クラスターのスケーリング」を参照してください。

クラスター設定を変更した後、負荷テストを繰り返す必要があります。

DynamoDB テーブルのデータモデリング

データモデリングに進む前に、DynamoDB の基礎を理解することが重要です。DynamoDB は、柔軟なスキーマを可能にする key-value NoSQL データベースです。各項目のキー属性以外の一連のデータ属性は、均一または個別のいずれかにすることができます。DynamoDB のキースキーマは、パーティションキーで項目を一意に識別する単純なプライマリキー形式であるか、パーティションキーとソートキーの組み合わせで項目を一意に定義する複合プライマリキー形式のいずれかです。パーティションキーは、データの物理的な場所を特定して取得するためにハッシュ化されます。そのため、データを均等に分散させるには、カーディナリティが高く、水平方向にスケーラブルな属性をパーティションキーとして選択することが重要です。ソートキー属性は、キースキーマではオプションです。ソートキーを使用すると、1 対多リレーションシップをモデル化し、DynamoDB で項目コレクションを作成できます。ソートキーは範囲キーとも呼ばれます。ソートキーを使用して項目コレクション内の項目をソートし、柔軟な範囲ベースの操作を可能にすることもできます。

DynamoDB のキースキーマの詳細とベストプラクティスについては、以下を参照してください。

- the section called “DynamoDB におけるパーティションとデータ分散”
- the section called “パーティションキーの設計”
- the section called “ソートキーの設計”
- 適切な DynamoDB パーティションキーの選択

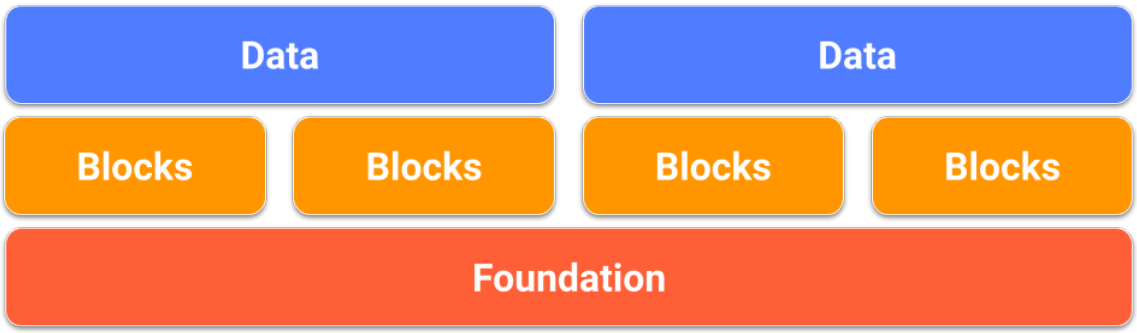
DynamoDB で追加のクエリパターンをサポートするには、多くの場合、セカンダリインデックスが必要になります。セカンダリインデックスは、同じデータがベーステーブルとは異なるキースキーマで整理されているシャドウテーブルです。ローカルセカンダリインデックス (LSI) は、ベーステーブルと同じパーティションキーを共有し、代替のソートキーを使用することでベーステーブルの容量を共有できます。グローバルセカンダリインデックス (GSI) では、ベーステーブルとは異なるパーティションキーと異なるソートキー属性を使用できます。つまり、GSI のスループット管理はベーステーブルとは独立しています。

セカンダリインデックスとベストプラクティスの詳細については、以下を参照してください。

- the section called “セカンダリインデックスの操作”
- the section called “セカンダリインデックス”

ここで、データモデリングをもう少し詳しく見てみましょう。高度に最適化された柔軟なスキーマを設計するプロセスは、DynamoDB に限らず、どの NoSQL データベースであっても習得が難しいス

キルと言えます。このモジュールの目標は、ユースケースから本番環境へと導く、スキーマ設計のメンタルフローチャートの開発を支援することにあります。まず、あらゆる設計の基本となる、シングルテーブル設計とマルチテーブル設計の選択について簡単に説明します。次に、アプリケーションの組織上またはパフォーマンス上のさまざまな結果を達成するために使用できる多様な設計パターン (構成要素) を確認します。最後に、ユースケースや業界別に異なる完全なスキーマ設計パッケージを紹介します。



トピック

- 項目コレクション - DynamoDB で一対多リレーションシップをモデル化する方法
- DynamoDB のデータモデリングの基盤
- DynamoDB のデータモデリングの構成要素
- DynamoDB のデータモデリングスキーマ設計パッケージ
- DynamoDB でリレーショナルデータをモデル化するためのベストプラクティス

項目コレクション - DynamoDB で一対多リレーションシップをモデル化する方法

DynamoDB の項目コレクションは、同じパーティションキー値を共有する項目のグループです。これは、これらの項目が関連していることを意味します。項目コレクションは、DynamoDB で 1 対多リレーションシップをモデル化する主要なメカニズムです。項目コレクションは、複合プライマリキーを使用するように設定されたテーブルまたはインデックスにのみ存在できます。

> **Note**
>
> 項目コレクションは、ベーステーブルまたはセカンダリインデックスのいずれかに存在できます。項目コレクションが特にインデックスとやり取りする方法の詳細については、「ローカルセカンダリインデックス内の項目コレクション」を参照してください。

3 つの異なるユーザーとそのゲーム内インベントリを示す次のテーブルを考えてみます。

Primary key		Attributes		
Partition key: PK	Sort key: SK			
account1234	inventory::armor	data		
		{"armor": [{"name" : "Pauldron of the Paladin", "type" : "chest", "gear score" : 545 }, { "name" : "Greaves of the Ranger", "type" : "sword", "gear score" : 382}]}		
	inventory::weapons	data		
		{"weapons" : [{ "name" : "Sword of the Ancients", "type" : "sword", "gear score" : 320}]}		
	login-data	pw	state	last-login
		d1e8a70b5ccab1dc2f56bbf7e99f064a660c08e361a35751b9c483c88943d082	Active	1649276737
account1387	info	data		
		{"email" : "bot123@gmail.com"}		
	inventory::armor	data		
		{"armor": [{"name" : "Pauldron of the Paladin", "type" : "chest", "gear score" : 545 }, { "name" : "Greaves of the Ranger", "type" : "sword", "gear score" : 382}]}		
	login-data	pw	state	last-login
		k2g8jk0m5ppab1dc2f56bbf7e99f064a660c08e361a35751b9c464r23943i082	Banned	1649456737
account1138	info	data		
		{"email" : "luh-3417@gmail.com" }		
	login-data	pw	state	last-login
		88A41A9A62B11CCC8C120861928765A3EA41DEB9EAFE261D90F619473B89A2D4	Active	14275516966

各コレクションの一部の項目において、ソートキーは、`inventory::armor`、`inventory::weapon`、`info` など、データをグループ化するための情報を連結したものです。項目コレクションごとに、これらの属性の異なる組み合わせをソートキーとして使用できます。ユーザー `account1234` には `inventory::weapons` 項目がありますが、ユーザー `account1387` にはありません (まだ見つけていないため)。ユーザー `account1138` は、ソートキーとして 2 つの項目のみ使用しています (まだインベントリがないため)。他のユーザーは 3 つを使用しています。

DynamoDB では、これらの項目コレクションから項目を選択的に取得して、次の操作を実行できます。

- 特定のユーザーからすべての項目を取得する。
- 特定のユーザーから 1 つの項目のみを取得する。
- 特定のユーザーに属する特定タイプのすべての項目を取得する。

項目コレクションでデータを整理してクエリを高速化する

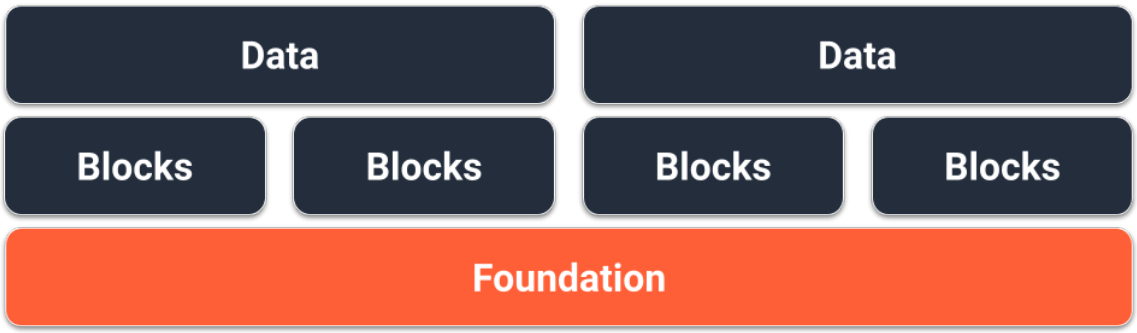
この例では、これらの 3 つの項目コレクションの各項目は、ゲームとプレイヤーのアクセスパターンに基づいて、この例で選択されたデータモデルにおけるプレイヤーを表しています。どのようなデータがゲームに必要であるか。いつ必要であるか。どのくらいの頻度で必要であるか。この方法で実行する場合のコストはどれくらいか。これらのデータモデリングの決定は、これらの質問に対する回答に基づいて行われました。

このゲームでは、武器のインベントリのページと鎧のページが別個にプレイヤーに提示されます。プレイヤーがインベントリを開くと、最初に武器が表示されます。これは、このページを非常に高速にロードする必要があるためであり、以降のインベントリページはその後にロードできます。これらの各項目タイプは、プレイヤーが獲得するゲーム内項目が増えるに従って肥大化する可能性があるため、データベースでは各インベントリページをプレイヤーの項目コレクションの独立した項目にしています。

次のセクションでは、`Query` オペレーションを使用して項目コレクションとやり取りする方法についてより詳しく説明します。

DynamoDB のデータモデリングの基盤

このセクションでは、基盤レイヤーについて、シングルテーブルとマルチテーブルという 2 種類のテーブル設計を検討します。



シングルテーブル設計の基盤

DynamoDB スキーマの基盤の 1 つとして、シングルテーブル設計を選択できます。シングルテーブル設計とは、複数のタイプ (エンティティ) のデータを単一の DynamoDB テーブルに保存できるパターンです。複数のテーブルやテーブル間の複雑な関係を維持する必要をなくすことで、データアクセスパターンを最適化し、パフォーマンスを向上させ、コストを削減することを目的としています。これが可能なのは、DynamoDB は、同じパーティションキーを持つ項目 (項目コレクションと呼ばれます) をいずれも同じパーティションに保存するためです。この設計では、異なる種類のデータが同じテーブル内に項目として保存され、各項目は一意のソートキーで識別されます。

Primary key		Attributes	
Partition key: PK	Sort key: SK		
UserID	Address#USA#CA#LA#90029	data	GSI-SK
		"Street Address"	Default
	Cart#ACTIVE#Coffee	data	GSI-SK
		CoffeeSKU	2019-11-27T103324
	Cart#ACTIVE#Spice	data	GSI-SK
		SpiceSKU	2019-11-28T091245
	Cart#SAVED#Cocoa	data	GSI-SK
		CocoaSKU	2019-11-28T125642
	OrderHistory#OrderUID	data	GSI-SK
		{Order:DataMap}	2019-10-08T132612
	ProfileName	data	
		"Paul Atreides"	
	Store#StoreUID	data	GSI-SK
		Los Angeles	Active

利点:

- データの局所性により、1 回のデータベース呼び出しで複数のエンティティタイプのクエリをサポートします。
- 読み取りにかかる全体的な財務コストとレイテンシーコストを削減します。
  - 合計 4 KB 未満の 2 つの項目に対する単一のクエリは、結果整合性のある 0.5 RCU になります。
  - 合計 4 KB 未満の 2 つの項目に対する 2 つのクエリは、結果整合性のある 1 RCU になります (それぞれ 0.5 RCU)。
  - 2 つの別々のデータベース呼び出しを返すまでの時間は、1 回の呼び出しよりも平均して長くなります。
- 管理するテーブルの数が減ります。
  - アクセス許可を複数の IAM ロールやポリシーにわたって管理する必要はありません。
  - テーブルの容量管理はすべてのエンティティにわたって平均化され、通常は消費パターンがより予測可能になります。
  - モニタリングに必要なアラームの数が減ります。
  - カスタマーマネージド暗号化キーは、1 つのテーブルでローテーションするだけで済みます

- テーブルへのトラフィックをスムーズにします。
  - 複数の使用パターンを同じテーブルに集約することで、全体的な使用がよりスムーズになる傾向があり (株価指数のパフォーマンスが個々の株式よりもスムーズになる傾向があるように)、プロビジョニングモードテーブルでより高い使用率を達成するのに適しています。

欠点:

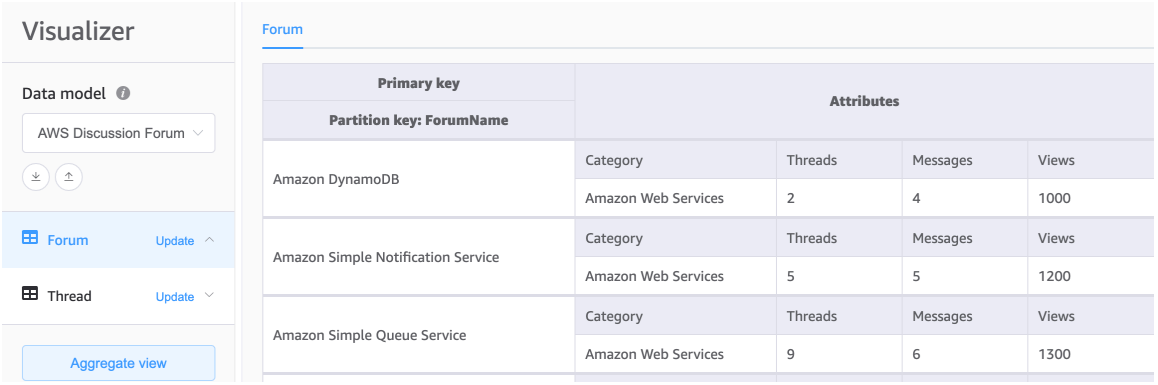
- リレーショナルデータベースと比べて逆説的な設計であるため、ラーニングカーブが急になる場合があります。
- データ要件はすべてのエンティティタイプにわたって一貫している必要があります。
  - バックアップは全部かゼロかのどちらかであるため、ミッションクリティカルでないデータは、別のテーブルに保存することを検討してください。
  - テーブルの暗号化はすべての項目間で共有されます。テナントごとに暗号化要件が異なるマルチテナントアプリケーションでは、クライアント側の暗号化が必要になります
  - 履歴データと運用データが混在するテーブルでは、低頻度アクセスストレージクラスを有効にしてもあまりメリットがありません。詳細については、DynamoDB テーブルクラスを参照してください。
- エンティティのサブセットのみを処理する必要がある場合でも、すべての変更されたデータは DynamoDB Streams に伝播されます。
  - Lambda を使用する場合は、Lambda イベントフィルターのおかげで請求額に影響しませんが、Kinesis Consumer Libary を使用する場合は、追加費用が発生します。
- GraphQL を使用する場合、シングルテーブル設計の実装はより難しくなります。
- Java の `DynamoDBMapper` や Enhanced Client などの上位レベルの SDK クライアントを使用する場合、同じ応答内の項目が異なるクラスに関連付けられる可能性があるため、結果の処理が難しくなりがちです。

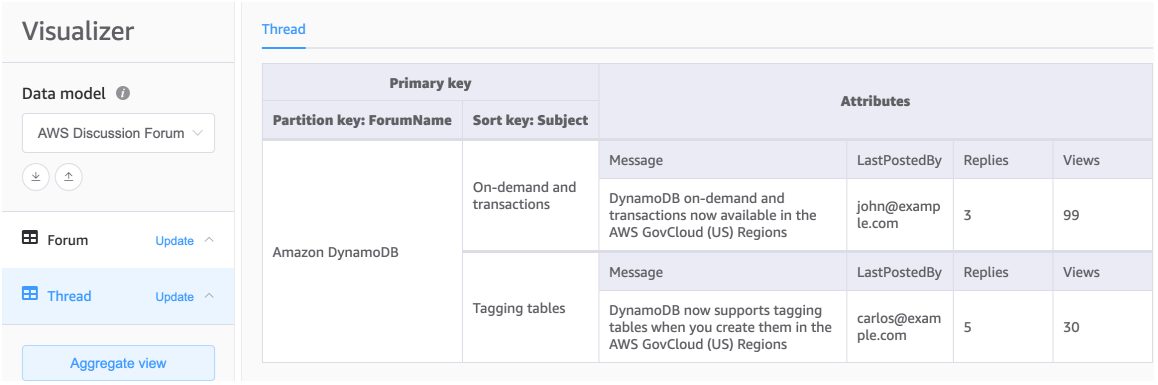
どのようなときに使うか:

単一テーブル設計は、複数のエンティティタイプを一緒に頻繁にクエリするアプリケーションや、異なるデータ型間の関係を維持する必要があるアプリケーションに適しています。これは、アクセスパターンがデータの局所性の恩恵を受ける場合や、複数のテーブルを管理するオーバーヘッドを最小限に抑えたい場合に特に効果的です。

マルチテーブル設計の基盤

DynamoDB スキーマの別の基盤として、マルチテーブル設計を選択できます。マルチテーブル設計は、DynamoDB のテーブルごとに 1 つのタイプ (エンティティ) のデータを保存する従来のデータベース設計に近いパターンです。各テーブル内のデータは引き続きパーティションキー別に整理されるため、単一のエンティティタイプ内のパフォーマンスとスケーラビリティは最適化されますが、複数のテーブルにわたるクエリは個別に実行する必要があります。





利点:

- シングルテーブル設計に慣れていないユーザーでも簡単に設計できます。
- 各リゾルバーが単一のエンティティ (テーブル) にマッピングされるため、GraphQL リゾルバーの実装が容易になります。
- エンティティタイプ別に異なるデータ要件が許容されます。
  - ミッションクリティカルな個々のテーブルのバックアップを作成できます
  - テーブル暗号化はテーブルごとに管理できます。テナントごとに暗号化要件が異なるマルチテナントアプリケーションでは、個別のテナントテーブルにより、お客様ごとに独自の暗号化キーを使用できます。

- 低頻度アクセスストレージクラスは、履歴データを含むテーブルでのみ有効にして、コスト削減のメリットを最大限に実現できます。詳細については、DynamoDB テーブルクラスを参照してください。
- 各テーブルには独自の変更データストリームがあり、単一のモノリシックプロセッサではなく、項目の種類ごとに専用の Lambda 関数を設計できます。

欠点:

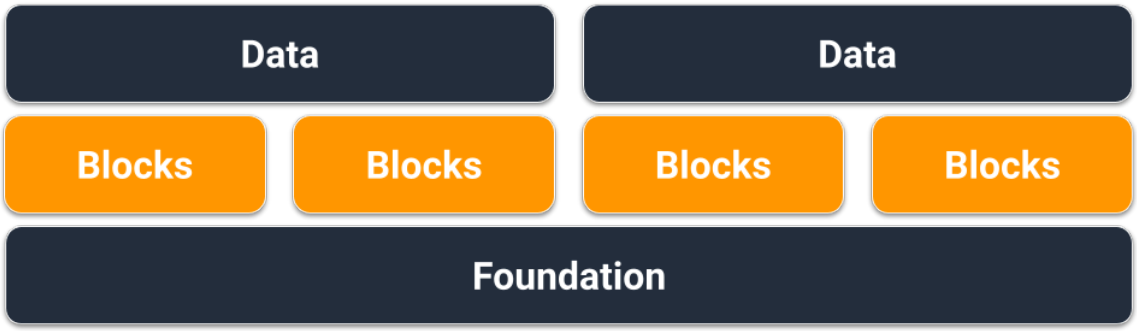
- 複数のテーブルにわたるデータを必要とするアクセスパターンの場合は、DynamoDB からの複数回の読み取りが必要になり、クライアントコードでのデータの処理/結合が必要になることがあります。
- 複数のテーブルの操作とモニタリングには、より多くの CloudWatch アラームが必要であり、各テーブルを個別にスケールする必要があります
- 各テーブルのアクセス許可を個別に管理する必要があります。今後テーブルを追加する場合は、必要な IAM ロールやポリシーの変更が必要になります。

どのようなときに使うか:

アプリケーションのアクセスパターンで複数のエンティティやテーブルをまとめてクエリする必要がない場合は、マルチテーブル設計が適切かつ十分なアプローチです。

DynamoDB のデータモデリングの構成要素

このセクションでは、アプリケーションで使用できる設計パターンを提供するビルディングブロックレイヤーについて説明します。



トピック

- 複合ソートキー構成要素

- マルチテナンシー構成要素
- スパースインデックスの構成要素
- Time to Live (有効期限) 構成要素
- Time to Live (アーカイブ用) 構成要素
- 垂直パーティショニング構成要素
- 書き込みシャーディング構成要素

複合ソートキー構成要素

NoSQL と言えば、非リレーショナルと考えがちです。ただし、DynamoDB の場合は、スキーマにリレーションシップを配置できない理由はなく、リレーショナルデータベースやその外部キーと見た目が異なるだけです。DynamoDB でデータの論理階層を構築するために使用できる最も重要なパターンの 1 つは、複合ソートキーです。最も一般的な設計スタイルは、階層の各レイヤー (親レイヤー > 子レイヤー > 孫レイヤー) をハッシュタグで区切ることです。例えば、`PARENT#CHILD#GRANDCHILD#ETC`。

パーティションキー: PK	ソートキー: SK
UserId	CART#ACTIVE#Apples
UserId	CART#ACTIVE#Bananas
UserId	CART#SAVED#Oranges
UserId	CART#SAVED#Pears
UserId	WISH#VEGGIES#Carrots

DynamoDB のパーティションキーには、データをクエリするために常に正確な値が必要です。一方、ソートキーには、バイナリツリーをたどる場合と同じように、左から右に部分条件を適用できます。

上の例では、e コマースストアのショッピングカートを、ユーザーセッションをまたいで維持する必要があります。ユーザーはログインするたびに、あとで買うために保存した商品を含むショッピングカート全体を確認したい場合があります。ただし、レジに進むときは、アクティブなカート内の商品のみをロードして購入できるようにする必要があります。これらの `KeyConditions` は、どちら

もカートのソートキーを明示的に要求するため、追加のウィッシュリストのデータは読み取り時に DynamoDB によって単に無視されます。保存した商品とアクティブな商品はどちらも同じカートの一部ですが、アプリケーションの異なる部分ごとに異なる方法で扱う必要があります。この場合、アプリケーションの部分ごとに必要なデータのみを取得する最適な方法は、ソートキーのプレフィックスに `KeyCondition` を適用することです。

この構成要素の主な特徴

- 関連項目は相互にローカルに保存されるため、データへのアクセスが効率的になります。
- `KeyCondition` 式を使用すると、階層のサブセットを選択的に取得できるため、無駄な RCU がなくなります。
- アプリケーションの異なる部分ごとに異なるプレフィックスを適用して項目を保存できるため、項目の上書きや書き込みの競合を防ぐことができます。

マルチテナンシー構成要素

多くのお客様が DynamoDB を使用してマルチテナントアプリケーションのデータをホストしています。このようなシナリオでは、1 つのテナントからのすべてのデータをテーブルの独自の論理パーティションに保持するようにスキーマを設計する必要があります。これには、項目コレクションという概念を活用します。項目コレクションとは、DynamoDB テーブル内で同じパーティションキーを持つすべての項目を表す用語です。DynamoDB におけるマルチテナンシーへのアプローチの詳細については、「DynamoDB のマルチテナンシー」を参照してください。

パーティションキー: PK	ソートキー: SK	ImageURL
UserOne	PhotoID1	https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserOne	PhotoID2	https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserTwo	PhotoID3	https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]
UserTwo	PhotoID4	https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]

パーティションキー: PK	ソートキー: SK	ImageURL
UserThree	PhotoID5	https://s3.amazonaws.com/[BUCKET-NAME]/[FILE-NAME].[FILE-TYPE]

この例では、ユーザー数が数千人に及ぶ可能性がある写真ホスティングサイトを運営しています。各ユーザーは、最初は自分のプロファイルにのみ写真をアップロードし、デフォルトでは他のユーザーの写真を見ることはできません。各ユーザーからの API コールの承認に分離レベルを追加して、各自のパーティションのデータのみを要求していることを確認するのが理想的ですが、スキーマレベルでは一意のパーティションキーで十分です。

この構成要素の主な特徴

- 1 人のユーザーやテナントが読み取ることができるデータ量は、各自のパーティション内の項目の合計量と同じ量に制限されます。
- アカウントの閉鎖やコンプライアンス要求に伴うテナントのデータの削除は、適切かつ安価に行うことができます。単に、パーティションキーがテナント ID と等しいクエリを実行し、返されたプライマリキーごとに `DeleteItem` 操作を実行します。

> **Note**
>
> マルチテナンシーを念頭に置いて設計されているため、1 つのテーブル全体でさまざまな暗号化キープロバイダーを使用してデータを安全に分離できます。AWSAmazon DynamoDB 用の Database Encryption SDK を使用すると、DynamoDB ワークロードにクライアント側の暗号化を組み込むことができます。属性レベルの暗号化を実行すると、特定の属性値を DynamoDB テーブルに保存する前に暗号化したり、データベース全体を事前に復号せずに暗号化された属性を検索したりできます。

スパースインデックスの構成要素

アクセスパターンによっては、まれな項目や、ステータスを受け取る項目 (応答のエスカレーションが必要) に一致する項目の検索が必要になる場合があります。これらの項目については、データセット全体を定期的にクエリするのではなく、グローバルセカンダリインデックス (GSI) を利用できま

す。GSI ではデータがまばらに読み込まれるという事実が役立ちます。つまり、インデックスに定義されている属性を持つベーステーブルの項目だけがインデックスにレプリケートされます。

パーティションキー: DeviceID	ソートキー: State#Date	オペレーター	日付	EscalatedTo
d#12345	NORMAL#2020-04-24T14:55:00	Liz	2020-04-24	
d#12345	WARNING1#2020-04-24T14:45:00	Liz	2020-04-24	
d#12345	WARNING1#2020-04-24T14:50:00	Liz	2020-04-24	
d#54321	NORMAL#2020-04-11T06:00:00	Liz	2020-04-11	
d#54321	NORMAL#2020-04-11T09:30:00	Sue	2020-04-11	
d#54321	WARNING2#2020-04-11T09:25:00	Sue	2020-04-11	
d#54321	WARNING3#2020-04-11T05:55:00	Liz	2020-04-11	
d#11223	WARNING4#2020-04-27T16:10:00	Sue	2020-04-27	
d#11223	WARNING4#2020-04-27T16:15:00	Sue	2020-04-27	Sara

パーティションキー: EscalatedTo	ソートキー: State#Date	DeviceID	オペレーター
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue

この例は、現場の各デバイスが定期的にステータスを報告している IOT のユースケースを示しています。ほとんどのレポートでは、デバイスから何の問題もないことが報告されると予想されますが、場合によっては障害が発生し、修理技術者へのエスカレーションが必要になる場合があります。エスカレーションを含むレポートでは、EscalatedTo 属性が項目に追加されますが、それ以外の場合は追加されません。この例の GSI は EscalatedTo でパーティション化されており、GSI はベーステーブルからキーを引き継ぐため、どの DeviceID がいつ障害を報告したかは依然として確認できます。

DynamoDB の場合、読み取りは書き込みよりも安価であり、スパースインデックスは、特定の種類の項目のインスタンスがめったに発生しないが、これらを見つけるための読み取りをよく行うユースケースでは非常に強力なツールです。

この構成要素の主な特徴

- スパース GSI の書き込みコストとストレージコストは、キーパターンに一致する項目にのみ適用されるため、GSI のコストは、すべての項目をレプリケートする他の GSI よりも大幅に低くなる可能性があります。
- 複合ソートキーを引き続き使用して、目的のクエリに一致する項目をさらに絞り込むこともできます。例えば、ソートキーにタイムスタンプを使用して、過去 X 分間 (`SK > 5 minutes ago, ScanIndexForward: False`) に報告された障害のみを表示できます。

Time to Live (有効期限) 構成要素

ほとんどのデータは、特定の期限まで、プライマリデータストアに保持する価値があると考えられます。DynamoDB からのデータのエージングアウトを容易にするために、Time to Live (TTL) と呼ばれる機能があります。TTL 機能を使用すると、エポックタイムスタンプ (経過済み) を持つ項目のために、モニタリングする必要がある特定の属性をテーブルレベルで定義できます。これにより、期限切れのレコードをテーブルから無料で削除できます。

> **Note**
>
> グローバルテーブルバージョン 2019.11.21 (現行) を使用しており、Time to Live 機能も使用している場合、DynamoDB は TTL による削除をすべてのレプリカテーブルにレプリケートします。最初の TTL による削除の場合、TTL の有効期限切れが発生したリージョンでは、書き込みキャパシティが消費されません。ただし、レプリカテーブルにレプリケートされた TTL による削除の場合、各レプリカリージョンではレプリケートされた書き込みキャパシティが消費され、該当する料金が適用されます。

パーティションキー: PK	ソートキー: MessageTimestamp	TTL	メッセージ
UserId	2030-06-30T12:12:12	1909570332	Hello
UserId	2030-06-30T12:17:22	1909570647	DynamoDB
UserId	2030-06-30T12:22:27	1909570947	TTL

この例は、存続期間の短いメッセージをユーザーが作成できるように設計したアプリケーションを示しています。DynamoDB でメッセージを作成すると、アプリケーションコードによって TTL 属性が 7 日後の日付に設定されます。約 7 日後に、DynamoDB はこれらの項目のエポックタイムスタンプが過去のものであることを確認し、削除します。

TTL による削除は無料であるため、この機能を使用してテーブルから履歴データを削除することを強くお勧めします。これにより、毎月の全体的なストレージ料金が削減され、クエリによって取得されるデータが少なくなるため、ユーザーの読み取りコストも削減される可能性があります。TTL はテーブルレベルで有効になっていますが、どの項目またはエンティティに対して TTL 属性を作成するか、またエポックタイムスタンプをどれだけ将来に設定するかはユーザーが指定できます。

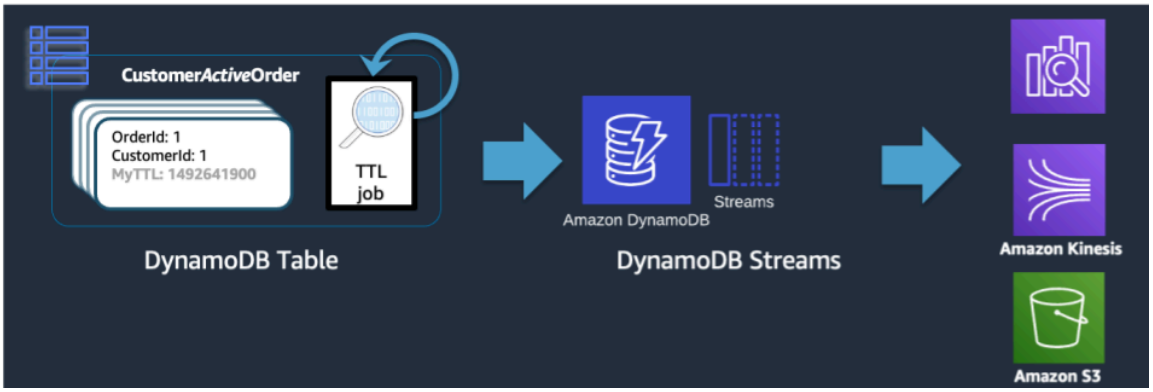
この構成要素の主な特徴

- TTL による削除はバックグラウンドで実行され、テーブルのパフォーマンスには影響しません
- TTL は約 6 時間ごとに実行される非同期プロセスですが、期限切れのレコードの削除には 48 時間超かかる場合があります。
  - 古いデータを 48 時間以内にクリーンアップする必要がある場合は、ロックレコードや状態管理などのユースケースで TTL による削除に依存しないでください。

- TTL 属性には有効な属性名を指定できますが、値は数値型でなければなりません。

Time to Live (アーカイブ用) 構成要素

TTL は DynamoDB から古いデータを削除する効果的なツールですが、多くのユースケースでは、データのアーカイブをプライマリデータストアよりも長期間保存する必要があります。この場合、TTL によるレコードの時間指定削除を利用して、期限切れのレコードを長期データストアにプッシュできます。



DynamoDB が TTL を使用して削除を実行すると、その操作は引き続き `Delete` イベントとして DynamoDB Stream にプッシュされます。ただし、DynamoDB の TTL 自体が削除を実行する場合は、`principal:dynamodb` のストリームレコードの属性を利用できます。DynamoDB Stream への Lambda サブスクライバーを使用すると、DynamoDB プリンシパル属性にのみイベントフィルターを適用することで、このフィルターに一致するすべてのレコードが Amazon Glacier などのアーカイブストアにプッシュされるようになります。

この構成要素の主な特徴

- 履歴項目に対する DynamoDB の低レイテンシーの読み取りが不要になったら、それらを Amazon Glacier などのコールドストレージサービスに移行することで、ストレージコストを大幅に削減すると同時に、ユースケースのデータコンプライアンスのニーズを満たすことができます。
- データが Amazon S3 に保存されている場合、Amazon Athena や Redshift Spectrum などのコスト効率の高い分析ツールを使用してデータの履歴分析を行うことができます。

垂直パーティショニング構成要素

ドキュメントモデルデータベースに精通しているユーザーなら、すべての関連データを 1 つの JSON ドキュメントに保存するという考え方に慣れているでしょう。DynamoDB は JSON データ

型をサポートしていますが、ネストされた JSON に対する KeyConditions の実行はサポートしていません。KeyConditions は、ディスクから読み取るデータの量とクエリが実際に消費する RCU の数を決定するものであるため、これに伴って大規模な非効率が生じる可能性があります。DynamoDB の書き込みと読み取りをより適切に最適化するには、ドキュメントの個々のエンティティを個別の DynamoDB 項目に分割することをお勧めします。これは垂直パーティショニングとも呼ばれます。

例えば、ユーザーのプロファイル、ストア、ショッピングカート、配送先住所、および注文履歴をまとめて保存する次の単一の JSON ドキュメントを考えてみます。

```
{
    "UserProfile": {
        "FirstName": "Paul",
        "LastName": "Atreides",
        "DateJoined": "1965-08-01"
    },
    "Store": {
        "store_id": "STOREUID",
        "city": "Los Angeles",
        "zip_code": "90029"
    },
    "ShoppingCart": [
        { "Spice": { "SKU": "SpiceSKU", "CategoryID": "FictionalSpice", "DateAdded":
 "2019-06-11" } },
        { "EspressoBeans": { "SKU": "CaffeineSKU", "CategoryID": "FOODANDDRINK",
 "DateAdded": "2019-06-10" } }
    ],
    "ShippingAddress": {
        "street_address": "1234 Arrakis Dr",
        "city": "Los Angeles",
        "zip_code": "90029",
        "status": "default"
    },
    "OrderHistory#OrderUID": {
        "ProductA": "SKU_A",
        "ProductB": "SKU_B",
        "DateOrdered": "2018-09-28"
    }
}
```

垂直パーティショニングを使用すると、単一のドキュメントは同じパーティションキー (`UserID`) を共有する個々の項目に分割されますが、ソートキープレフィックスを使用して各エンティティを識別します。次の表は、個別の項目として保存されている同じデータを示しています。

パーティションキー: UserID	ソートキー: SK	属性
UserId	UserProfile	FirstName: Paul, LastName: Atreides, DateJoined: 1965-08-01
UserId	Store#STOREUID	city: Los Angeles, zip_code: 90,029
UserId	Cart#ACTIVE#Spice	SKU: SpiceSKU, CategoryID: FictionalSpice, DateAdded: 2019-06-11
UserId	Cart#ACTIVE#EspressoBeans	SKU: CaffeineSKU, CategoryID: FOODANDDRINK, DateAdded: 2019-06-10
UserId	Address#default	street_address: 1,234 Arrakis Dr, city: Los Angeles, zip_code: 90,029, status: default
UserId	OrderHistory#OrderUID	ProductA: SKU_A, ProductB: SKU_B, DateOrdered: 2018-09-28

上記の垂直パーティショニングは、1 つのテーブル設計を示す重要な実例ですが、必要に応じて複数のテーブルに実装することもできます。DynamoDB は書き込みを 1 KB 単位で請求するため、理想的には各項目が 1 KB 未満になるようにドキュメントを分割します。

この構成要素の主な特徴

- データ関係の階層はソートキーのプレフィックスによって維持されるため、必要に応じて単一のドキュメント構造をクライアント側で再構築できます。
- データ構造の個々のコンポーネントを個別に更新できるため、小さな項目の更新はわずか 1 WCU で済みます。
- ソートキー `BeginsWith` を使用すると、アプリケーションは 1 回のクエリで類似したデータを取得し、読み取りコストを集約して総コスト/レイテンシーを削減できます。

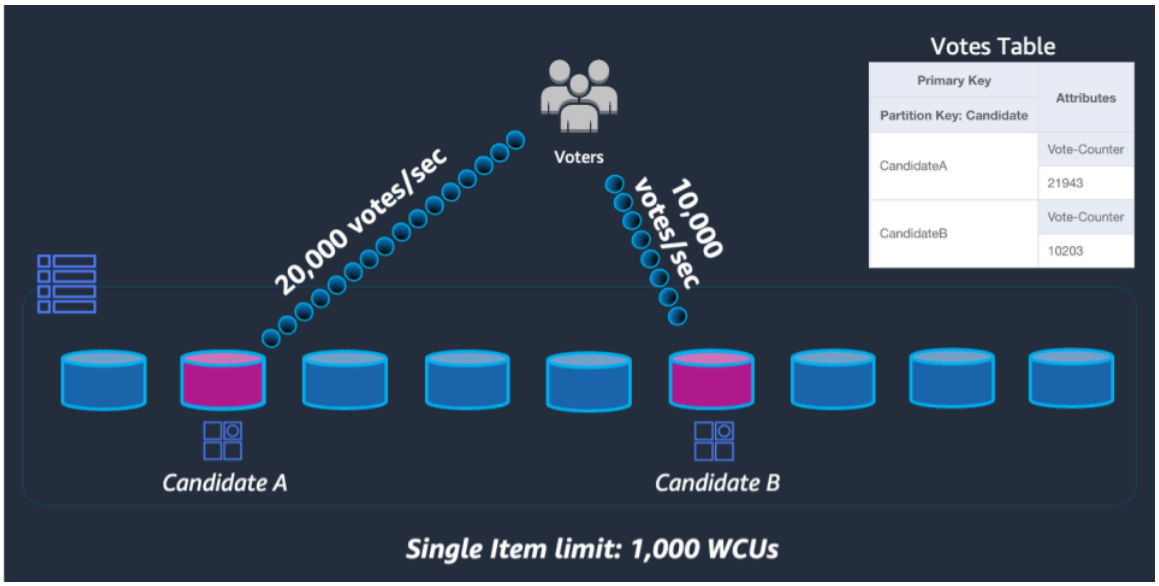
- サイズの大きいドキュメントは、DynamoDB の個別項目のサイズ制限である 400 KB を簡単に超えてしまう可能性があり、垂直パーティショニングはこの制限を回避するのに役立ちます。

書き込みシャーディング構成要素

DynamoDB に設定されている数少ないハード制限の 1 つは、単一の物理パーティションが 1 秒あたりに維持できるスループットに対する制限です (必ずしも単一のパーティションキーとは限りません)。現在、これらの制限は以下のとおりです。

- 1000 WCU (または 1 秒あたり 1000 <=1KB 項目の書き込み) および 3000 RCU (または 1 秒あたり 3000 <=4KB の読み取り) の強力な整合性のある読み込み、または
- 1 秒あたり 6000 <=4KB の結果整合性のある読み込み

テーブルに対するリクエストがこれらの制限のいずれかを超えると、エラーが `ThroughputExceededException` のクライアント SDK に送り返されます。これは一般的にスロットリングと呼ばれます。この制限を超える読み取り操作を必要とするユースケースは、ほとんどの場合、DynamoDB の前に読み取りキャッシュを配置することで最適に処理されます。ただし、書き込み操作には書き込みシャーディングと呼ばれるスキーマレベルの設計が必要です。



パーティションキー: Candidate	Vote-Counter	最終更新日
CandidateA#1	10238	2019-09-30T11:35:53
CandidateA#2	8452	2019-09-30T11:35:53

パーティションキー: Candidate	Vote-Counter	最終更新日
CandidateA#3	9148	2019-09-30T11:35:53
CandidateA#4	11092	2019-09-30T11:35:53

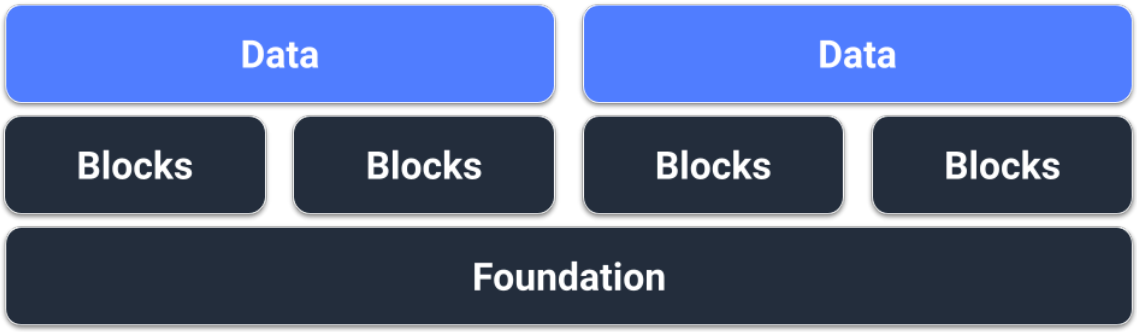
この問題を解決するには、アプリケーションの `UpdateItem` コードで各候補者のパーティションキーの末尾にランダムな整数を追加します。ランダム整数ジェネレータの範囲の上限は、特定の候補者の 1 秒あたりの予想書き込み量を 1000 で割った値と一致するか、それを超える必要があります。1 秒あたり 20,000 の投票数をサポートするには、rand(0,19) のようになります。データは個別の論理パーティションに保存されているため、読み取り時にデータを結合し直す必要があります。投票総数はリアルタイムである必要はないため、X 分ごとにすべての投票パーティションを読み取るようにスケジュールした Lambda 関数では、候補者ごとに集計をときどき行い、ライブ読み取り用の 1 つの投票集計レコードに書き戻すことができます。

この構成要素の主な特徴

- 特定のパーティションキーに対する書き込みスループットがきわめて高く、これを避けられないユースケースでは、書き込み操作を複数の DynamoDB パーティションに人為的に分散できます。
- GSI でのスロットリングはベーステーブルへの書き込み操作にバックプレッシャーを与えるため、カーディナリティの低いパーティションキーを持つ GSI でも、このパターンを利用する必要があります。

DynamoDB のデータモデリングスキーマ設計パッケージ

DynamoDB のデータモデリングスキーマ設計パッケージについて説明します。これには、ソーシャルネットワーク、ゲームプロファイル、苦情管理、定期払い、デバイスステータス、オンラインショップのユースケース、アクセスパターン、最終スキーマ設計が含まれます。



前提条件

DynamoDB のスキーマを設計する前に、まずスキーマがサポートする必要のあるユースケースに関する前提条件となるデータを収集する必要があります。リレーショナルデータベースとは異なり、DynamoDB はデフォルトでシャーディングされます。つまり、データはバックグラウンドで複数のサーバーに保存されるため、データの局所性を考慮して設計することが重要です。スキーマ設計ごとに次のリストを作成する必要があります。

- エンティティのリスト (ER 図)
- 各エンティティの推定ボリュームとスループット
- サポートが必要なアクセスパターン (クエリと書き込み)
- データ保持要件

トピック

- DynamoDB でのソーシャルネットワークのスキーマ設計
- DynamoDB でのゲームプロファイルスキーマの設計
- DynamoDB の苦情管理システムのスキーマ設計
- DynamoDB での定期払いスキーマの設計
- DynamoDB でのデバイスステータス更新のモニタリング
- DynamoDB をオンラインショップのデータストアとして使用する

DynamoDB でのソーシャルネットワークのスキーマ設計

ソーシャルネットワークのビジネスユースケース

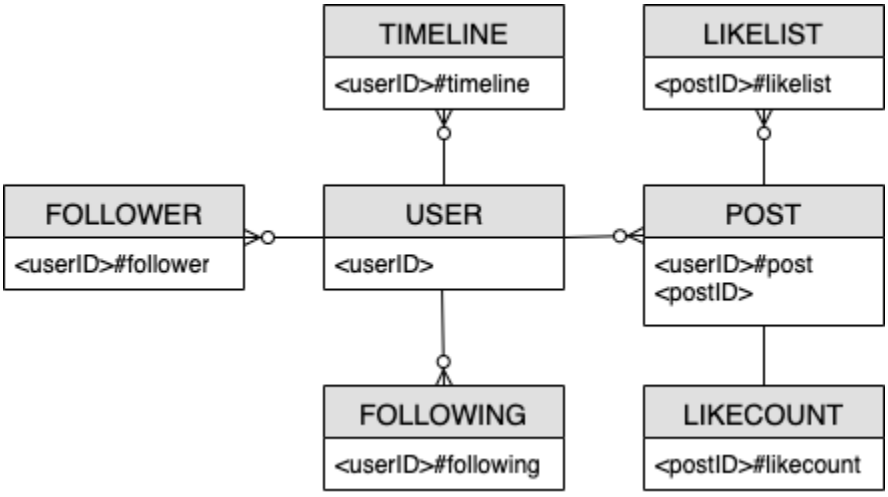
このユースケースでは、DynamoDB をソーシャルネットワークとして使用する方法について説明します。ソーシャルネットワークは、さまざまなユーザーが相互に交流できるオンラインサービスです。これから設計するソーシャルネットワークでは、ユーザーの投稿、フォロワー、フォローしている相手、およびフォローしている相手による投稿で構成されるタイムラインをユーザーが表示できるようにします。このスキーマ設計のアクセスパターンは以下のとおりです。

- 特定のユーザー ID のユーザー情報を取得する。
- 特定のユーザー ID のフォロワーリストを取得する。
- 特定のユーザー ID のフォローリストを取得する。
- 特定のユーザー ID の投稿リストを取得する。

- 特定の投稿 ID による投稿に「いいね!」したユーザーのリストを取得する。
- 特定の投稿 ID の「いいね!」数を取得する。
- 特定のユーザー ID のタイムラインを取得する。

ソーシャルネットワークエンティティ関係図

これは、ソーシャルネットワークのスキーマ設計に使用するエンティティ関係図 (ERD) です。



ソーシャルネットワークのアクセスパターン

これらは、ソーシャルネットワークのスキーマ設計のために検討するアクセスパターンです。

- `getUserInfoByUserID`
- `getFollowerListByUserID`
- `getFollowingListByUserID`
- `getPostListByUserID`
- `getUserLikesByPostID`
- `getLikeCountByPostID`
- `getTimelineByUserID`

ソーシャルネットワークのスキーマ設計の進化

DynamoDB は NoSQL データベースであるため、結合 (複数のデータベースのデータを結合する操作) は実行できません。DynamoDB に慣れていないお客様は、必要がない場合でも、リレーショナルデータベース管理システム (RDBMS) の設計理念 (エンティティごとにテーブルを作成するなど) を

DynamoDB に適用する可能性があります。DynamoDB のシングルテーブル設計の目的は、アプリケーションのアクセスパターンに従って事前に結合された形式でデータを書き込み、追加の計算を行わずにそのデータをすぐに使用することにあります。詳細については、「DynamoDB のシングルテーブル設計とマルチテーブル設計」を参照してください。

それでは、すべてのアクセスパターンに対処するためにスキーマ設計をどのように進化させるかをステップ別に見ていきましょう。

ステップ 1: アクセスパターン 1 (**getUserInfoByUserID**) に対処する

特定のユーザーに関する情報を取得するには、キー条件を PK=<userID> にしてベーステーブルに Query を実行する必要があります。クエリ操作では、結果をページ分割できます。これは、ユーザーに多数のフォロワーがいる場合に便利です。Query の詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

この例では、ユーザーの「count」と「info」の 2 種類のデータを追跡します。ユーザーの「count」は、ユーザーのフォロワー数、フォローしている相手の数、相手が作成した投稿の数を反映します。ユーザーの「info」は、名前などの個人情報を反映します。

これら 2 種類のデータは、以下の 2 つの項目で表されます。ソートキー (SK) に「count」が含まれている項目は、「info」が含まれている項目よりも変更される可能性が高くなります。DynamoDB は、更新前と更新後に表示される項目のサイズを考慮し、消費されるプロビジョニングされたスループットに、これらの項目サイズの大きい方を反映します。したがって、項目の属性の一部だけを更新した場合でも、UpdateItem は、プロビジョニングされたスループットの総量 (前後の項目サイズの大きい方) を消費します。単一の Query 操作で項目を取得し、UpdateItem を使用して既存の数値属性の加算または減算ができます。

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...

ステップ 2: アクセスパターン 2 (**getFollowerListByUserID**) に対処する

特定のユーザーのフォロワーリストを取得するには、キー条件を PK=<userID>#follower にしてベーステーブルに Query を実行する必要があります。

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				

ステップ 3: アクセスパターン 3 (**getFollowingListByUserID**) に対処する

特定のユーザーがフォローしている相手のリストを取得するには、キー条件を PK=<userID>#following にしてベーステーブルに Query を実行する必要があります。次に、TransactWriteItems オペレーションを使用して複数のリクエストをグループ化し、以下のことを実行できます。

- ユーザー A をユーザー B のフォロワーリストに追加し、ユーザー B のフォロワー数を 1 つ増やします。
- ユーザー B をユーザー A のフォロワーリストに追加し、ユーザー A のフォロワー数を 1 つ増やします

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				

ステップ 4: アクセスパターン 4 (**getPostListByUserID**) に対処する

特定のユーザーが作成した投稿のリストを取得するには、キー条件を PK=<userID>#post にしてベーステーブルに Query を実行する必要があります。ここで注意すべき重要な点の 1 つは、ユー

ザーの postID はインクリメンタルでなければならないということです。つまり、2 番目の PostID 値は 1 番目の PostID 値よりも大きくなければなりません (ユーザーは自分の投稿を並べ替えて表示したいはずです)。これを行うには、Universally Unique Lexicographically Sortable Identifier (ULID) などの時間値に基づいて postID を生成できます。

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	

ステップ 5: アクセスパターン 5 (**`getUserLikesByPostID`**) に対処する

特定のユーザーの投稿に「いいね!」を付けた人のリストを取得するには、キー条件を `PK=<postID>#likelist` にしてベーステーブルに `Query` を実行する必要があります。このアプローチは、アクセスパターン 2 (`getFollowerListByUserID`) とアクセスパターン 3 (`getFollowingListByUserID`) でフォロワーリストとフォローしている相手のリストを取得するときに使用したのと同じパターンです。

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				

ステップ 6: アクセスパターン 6 (**`getLikeCountByPostID`**) に対処する

特定の投稿の「いいね!」の数を取得するには、キー条件を PK=<postID>#likecount にしてベーステーブルに `GetItem` 操作を実行する必要があります。このアクセスパターンでは、パーティションのスループットが 1 秒あたり 1000 WCU を超えるとスロットリングが発生するため、フォロワーの多いユーザー (有名人など) が投稿を作成するたびにスロットリングの問題が発生する可能性があります。この問題は DynamoDB が原因ではなく、DynamoDB がソフトウェアスタックの最後にあるため、DynamoDB に現れるだけです。

すべてのユーザーに「いいね!」の数を同時に表示することが本当に必要か、それとも時間の経過と共に徐々に表示できるかを評価する必要があります。一般的に、投稿の「いいね!」の数はすぐに 100% 正確である必要はありません。この戦略を実装するには、アプリケーションと DynamoDB の間にキューを置き、更新が定期的に行われるようにします。

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			

ステップ 7: アクセスパターン 7 (**getTimelineByUserID**) に対処する

特定のユーザーのタイムラインを取得するには、キー条件を PK=<userID>#timeline にしてベーステーブルに Query 操作を実行する必要があります。ユーザーのフォロワーが自分の投稿を同期的に表示する必要があるシナリオを考えてみましょう。ユーザーが投稿を書くたびに、フォロワーリストが読み取られ、useIDと postID がすべてのフォロワーのタイムラインキーにゆっくりと入力されます。次に、アプリケーションを起動すると、Query 操作でタイムラインキーを読み取り、新しい項目に対して BatchGetItem 操作を使用して userID と postID の組み合わせをタイムライン画面に入力できます。API コールでタイムラインを読み取ることはできませんが、投稿が頻繁に編集される可能性がある場合、これはより費用対効果の高いソリューションです。

タイムラインは最近の投稿を表示する場所なので、古い投稿をクリーンアップする方法が必要です。WCU を使用して削除する代わりに、DynamoDB の TTL 機能を使用して無料で削除できます。

Primary key		Attributes			
Partition key: PK	Sort key: SK				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			
u#12345#timeline	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getUserInfoByUserID	ベーステーブル	クエリ	PK=<userID>		

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getFollowerListByUserID	ベーステーブル	クエリ	PK=<userID>#follower		
getFollowingListByUserID	ベーステーブル	クエリ	PK=<userID>#following		
getPostListByUserID	ベーステーブル	クエリ	PK=<userID>#post		
getUserLikesByPostID	ベーステーブル	クエリ	PK=<postID>#likelist		
getLikeCountByPostID	ベーステーブル	GetItem	PK=<postID>#likecount		
getTimelineByUserID	ベーステーブル	クエリ	PK=<userID>#timeline		

ソーシャルネットワークの最終スキーマ

これが最終的なスキーマ設計です。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の DynamoDB の例を参照してください。

ベーステーブル:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
u#12345	"count"	follower#	following#	post#	
		3000000000	971	4945	
	"info"	name	content	imageUrl	etc
		hyuklee	My name is Hyuk Lee	s3://....	...
u#12345#follower	u#23456				
	u#34567				
	u#45678				
u#12345#following	u#56789				
	u#67890				
	u#78912				
u#12345#post	p#12345	content	imageUrl	timestamp	
		content for a post	s3://....	1571827560	
	p#23456	content	imageUrl	timestamp	
		content for a post	s3://....	1571827561	
p#12345#likelist	u#23456				
	u#34567				
	u#45678				
p#12345#likecount	"count"	etc			
		100			
u#12345#timeline	p#34567#u#56789	ttl			
		1571827560			
	p#45678#u#67890	ttl			
		1571827560			
	p#56789#u#78901	ttl			
		1571827560			

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。

3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB でのゲームプロファイルスキーマの設計

ゲームプロファイルのビジネスユースケース

このユースケースでは、DynamoDB を使用してゲームシステムのプレイヤープロファイルを保存する方法について説明します。ユーザー (この場合はプレイヤー) は、多くの最新のゲーム、特にオンラインゲームを操作する前に、プロファイルを作成する必要があります。ゲームプロファイルには通常、以下が含まれます。

- ユーザー名などの基本情報
- 項目や装備などのゲームデータ
- タスクやアクティビティなどのゲーム記録
- 友達リストなどのソーシャル情報

このアプリケーションのきめ細かなデータクエリアクセス要件を満たすため、プライマリキー (パーティションキーとソートキー) には汎用名 (PK と SK) を使用するため、以下に示すように、さまざまなタイプの値でオーバーロードできます。

このスキーマ設計のアクセスパターンは以下のとおりです。

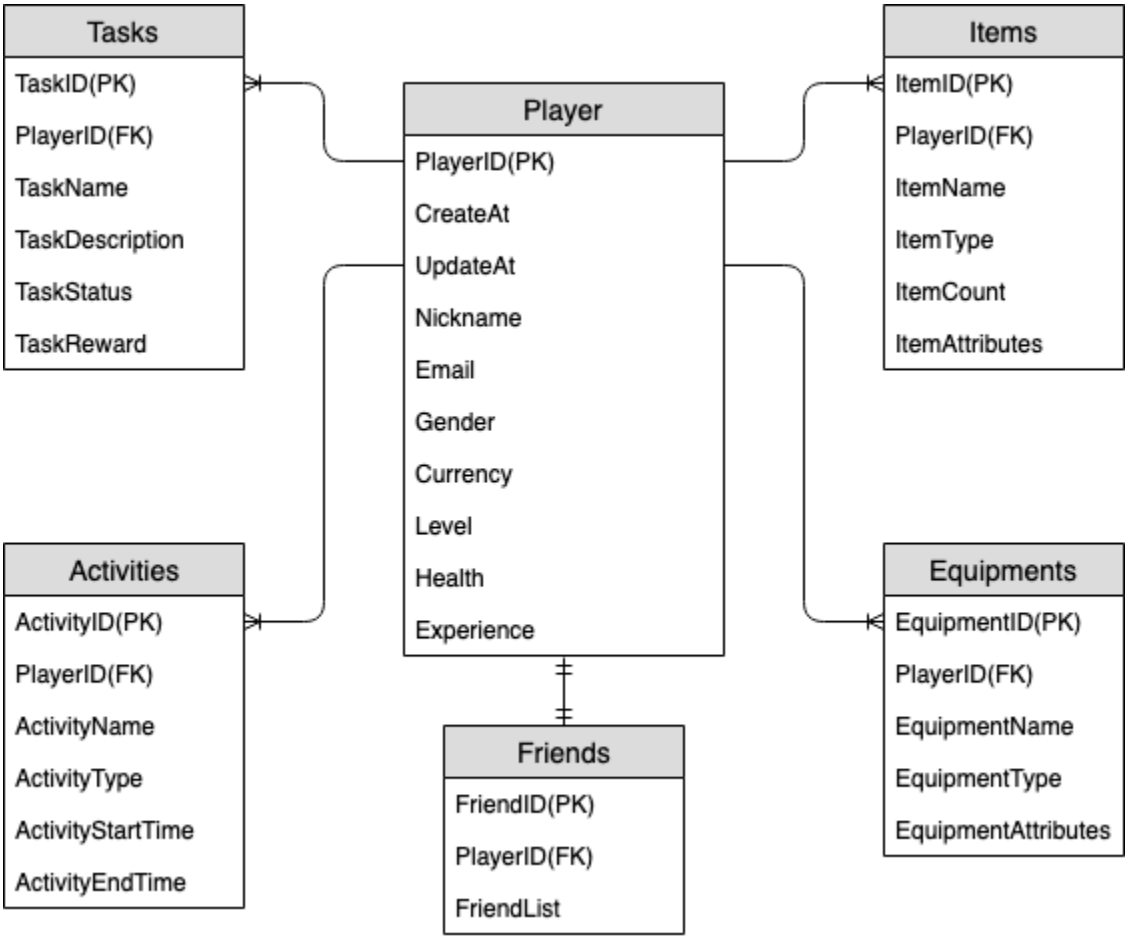
- ユーザーの友達リストを取得する。
- プレイヤーのすべての情報を取得する。
- ユーザーの項目リストを取得する。
- ユーザーの項目リストから特定の項目を取得する。
- ユーザーのキャラクターを更新する。
- ユーザーの項目数を更新する。

ゲームプロファイルのサイズは、ゲームによって異なります。大きな属性値は圧縮することで、DynamoDB の項目制限内に収め、コストを削減できます。スループット管理戦略は、プレイヤー数、1 秒あたりにプレイされるゲーム数、ワークロードの季節性などのさまざまな要因によって

異なります。新しくリリースされたゲームの場合、通常、プレイヤーの数や人気度は不明であるため、オンデマンドスループットモードから始めます。

ゲームプロファイルエンティティ関係図

次に示すのは、ゲームプロファイルのスキーマ設計に使用するエンティティ関係図 (ERD) です。



ゲームプロファイルのアクセスパターン

これらは、ソーシャルネットワークのスキーマ設計のために検討するアクセスパターンです。

- `getPlayerFriends`
- `getPlayerAllProfile`
- `getPlayerAllItems`
- `getPlayerSpecificItem`
- `updateCharacterAttributes`
- `updateItemCount`

ゲームプロファイルスキーマ設計の進化

上記の ERD から、これは 1 対多リレーションシップタイプのデータモデリングであることがわかります。DynamoDB では、1 対多のデータモデルを項目コレクションに整理できます。これは、複数のテーブルを作成して外部キーでリンクする従来のリレーショナルデータベースとは異なります。項目コレクションは、同じパーティションキー値を共有していてもソートキー値が異なる項目のグループです。項目コレクション内の各項目には、他の項目と区別する固有のソートキー値があります。これを念頭に置いて、エンティティタイプごとに HASH 値と RANGE 値のパターンを使用してみましょう。

まず、PK や SK などの汎用名を使用して、さまざまなタイプのエンティティを同じテーブルに保存し、モデルを将来にわたって使用できるようにします。読みやすくするために、データのタイプを示すプレフィックスを含めたり、Entity_type または Type という任意の属性を含めたりできます。現在の例では、player で始まる文字列を PK として player_ID を保存し、SK のプレフィックスとして entity name# を使用します。さらに、このデータがどのエンティティタイプであるかを示す Type 属性を追加します。これにより、将来的にはより多くのエンティティタイプの保存をサポートできるようになり、GSI オーバーローディングやスパース GSI などの高度なテクノロジーを使用して、より多くのアクセスパターンに対応できるようになります。

アクセスパターンの実装を始めましょう。プレイヤーの追加や装備の追加などのアクセスパターンは PutItem 操作を通じて実現できるため、無視して構いません。このドキュメントでは、上記の一般的なアクセスパターンに焦点を当てます。

ステップ 1: アクセスパターン 1 (**getPlayerFriends**) に対処する

このステップではアクセスパターン 1 (getPlayerFriends) に対処します。現在の設計の場合、友達の関係はシンプルで、ゲーム内の友達の数は少なくなっています。わかりやすくするために、リストデータ型を使用して友達リストを保存します (1:1 モデリング)。この設計では、GetItem を使用してこのアクセスパターンに対処します。GetItem 操作では、特定の項目を取得するためのパーティションキーとソートキーの値を明示的に指定します。

ただし、ゲームに多数の友達がいて、友達間の関係が複雑な場合 (招待コンポーネントと承諾コンポーネントの両方で友達の関係が双方向であるなど)、友達リストのサイズを無制限にスケールするには、多対多リレーションシップを使用して各友達を個別に保存する必要があります。また、友達関係の変更に伴って複数の項目を同時に操作する必要がある場合は、DynamoDB トランザクションを使用して複数のアクションをグループ化し、単一のオールオアナッシングの TransactWriteItems 操作または TransactGetItems 操作として送信できます。

Primary key		Attributes	
Partition key: PK	Sort key: SK		
player001	FRIENDS#player001	Type	FriendList
		Friends	[{"M": {"FriendId": {"S":"player002"},"FriendName": {"S":"Alice"}}}, {"M":{"FriendId": {"S":"player003"},"FriendName": {"S":"Bob"}}}]

ステップ 2: アクセスパターン 2 (**getPlayerAllProfile**),3 (**getPlayerAllItems**)、4 (**getPlayerSpecificItem**) に対処する

このステップを使用して、アクセスパターン 2 (getPlayerAllProfile)、3 (getPlayerAllItems)、4 (getPlayerSpecificItem) に対処します。これら 3 つのアクセスパターンに共通しているのは、Query 操作を使用する範囲クエリです。クエリの範囲によっては、実際の開発でよく使用されるキー条件およびフィルター式を使用します。

Query 操作では、パーティションキーとして 1 つの値を指定し、このパーティションキー値を持つすべての項目を取得します。このようにしてアクセスパターン 2 (getPlayerAllProfile) を実装します。オプションで、ソートキー条件式、つまりテーブルから読み取る項目を決定する文字列を追加できます。アクセスパターン 3 (getPlayerAllItems) は、ソートキー begins_with ITEMS# というキー条件を追加することで実装します。さらに、アプリケーション側の開発を簡略化するために、フィルター式を使用してアクセスパターン 4 (getPlayerSpecificItem) を実装できます。

Weapon カテゴリの項目をフィルタリングするフィルター式を使用した疑似コードの例を次に示します。

```
filterExpression: "ItemType = :itemType"
expressionAttributeValues: {":itemType": "Weapon"}
```

Primary key		Attributes				
Partition key: PK	**Sort key: SK**					
player001	ITEMS#001	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Health Potion	Consumable	5	{"M":{"HP":{"N":"50"}}}
	ITEMS#002	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Armor of the Knight	Armor	1	{"M":{"DEF":{"N":"100"}}}
	ITEMS#003	Type	ItemName	ItemType	ItemCount	ItemAttributes
		Item	Sword of the Dragon	Weapon	1	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}

> **Note**
>
> フィルター式は、クエリの完了後、結果がクライアントに返される前に適用されます。したがって、Query は、フィルター式の有無にかかわらず、同じ量の読み取りキャパシティを消費します。

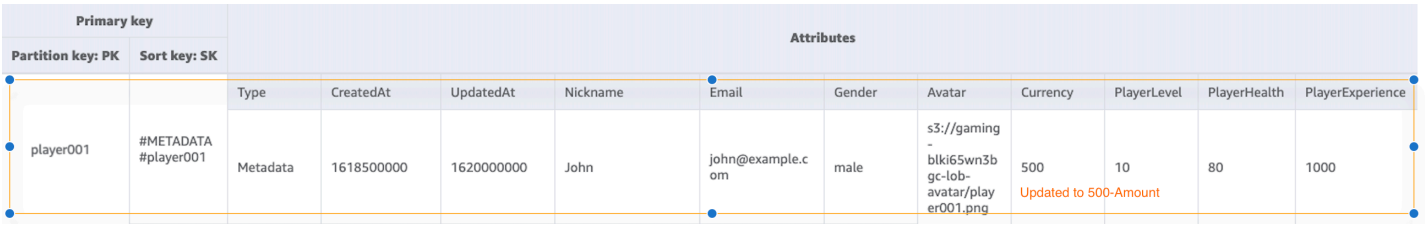
アクセスパターンとして、大きなデータセットをクエリし、大量のデータをフィルタリングして一部のデータのみを保持する場合、適切なアプローチは DynamoDB パーティションキーとソートキーをより効果的に設計することです。例えば、前述の特定の ItemType を取得する例で、各プレイヤーに多数の項目があり、特定の ItemType をクエリするのが一般的なアクセスパターンである場合は、ItemType を複合キーとして SK に取り込む方が効率的です。データモデルは ITEMS#ItemType#ItemId のようになります。

ステップ 3: アクセスパターン 5 (**updateCharacterAttributes**) および 6 (**updateItemCount**) に対処する

このステップを使用して、アクセスパターン 5 (updateCharacterAttributes) と 6 (updateItemCount) に対処します。プレイヤーが通貨を減らしたり、項目内の特定の武器の数量を変更したりするなど、キャラクターを変更する必要がある場合は、UpdateItem を使用してこれらのアクセスパターンを実装します。プレイヤーの通貨を更新しながら最低額を下回らないようにするには、the section called “CLI の例” を追加して残高が最低額以上である場合にのみ残高を減らすことができます。擬似コードの例を次に示します。

```
UpdateExpression: "SET currency = currency - :amount"
```

```
ConditionExpression: "currency >= :minAmount"
```



Primary key		Attributes									
Partition key: PK	Sort key: SK	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
player001	#METADATA #player001	Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-blki65wn3bgc-lob-avatar/player001.png	500 Updated to 500-Amount	10	80	1000

DynamoDB で開発し、アトミックカウンタを使用してインベントリを減らす場合、楽観的ロックを使用することで冪等性を確保できます。アトミックカウンタの擬似コードの例を次に示します。

```
UpdateExpression: "SET ItemCount = ItemCount - :incr"
expression-attribute-values: '{":incr":{"N":"1"}}'
```

Primary key		Attributes				
Partition key: PK	Sort key: SK	Type	ItemName	ItemType	ItemCount	ItemAttributes
player001	ITEMS#001	Item	Health Potion	Consumable	5 Updated to 4	{"M":{"HP":{"N":"50"}}}

さらに、プレイヤーが通貨で項目を購入するシナリオでは、プロセス全体で通貨の差し引きと項目の追加を同時に行う必要があります。DynamoDB トランザクションを使用すると、複数のアクションをグループ化し、単一の全部かゼロかの TransactWriteItems 操作または TransactGetItems 操作として送信できます。TransactWriteItems は、単一の全部かゼロかの操作で最大 100 の書き込みアクションをグループ化する、同期かつ冪等性の書き込み操作です。アクションはアトミックに完了するため、すべてが成功するか、どれも成功しません。トランザクションは、通貨の重複や消滅のリスクを排除するのに役立ちます。トランザクションの詳細については、「DynamoDB トランザクションの例」を参照してください。

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getPlayerFriends	ベーステーブル	GetItem	PK=PlayerID	SK=“FRIENDS#playerID”	

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getPlayerAllProfile	ベーステーブル	クエリ	PK=PlayerID		
getPlayerAllItems	ベーステーブル	クエリ	PK=PlayerID	SK begins_with “ITEMS#”	
getPlayerSpecificItem	ベーステーブル	クエリ	PK=PlayerID	SK begins_with “ITEMS#”	filterExpression: "ItemType = :itemType" expressionAttributeValues: { ":itemType": "Weapon" }
updateCharacterAttributes	ベーステーブル	UpdateItem	PK=PlayerID	SK=“#METADATA#playerID”	UpdateExpression: "SET currency = currency - :amount" ConditionExpression: "currency >= :minAmount"

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
updateItemCount	ベーステーブル	UpdateItem	PK=PlayerID	SK =“ITEMS#ItemID”	update-expression: "SET ItemCount = ItemCount - :incr" expression-attribute-values : '{":incr": {"N":"1"}}'

ゲームプロファイルの最終スキーマ

これが最終的なスキーマ設計です。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の DynamoDB の例を参照してください。

ベーステーブル:

Primary key		Attributes										
Partition key: PK	Sort key: SK											
player001	#METADATA #player001	Type	CreatedAt	UpdatedAt	Nickname	Email	Gender	Avatar	Currency	PlayerLevel	PlayerHealth	PlayerExperience
		Metadata	1618500000	1620000000	John	john@example.com	male	s3://gaming-blki65wn3bgc-lob-avatar/player001.png	500	10	80	1000
	ACTIVITY#001	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647475199	Hunting Trip	{"M":{"Gold":{"N":"50"},"XP":{"N":"200"}}}	1647388800	Hunting					
	ACTIVITY#002	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647647999	Mining Adventure	{"M":{"Gold":{"N":"1000"},"XP":{"N":"500"}}}	1647561600	Mining					
	ACTIVITY#003	Type	ActivityEndTime	ActivityName	ActivityReward	ActivityStartTime	ActivityType					
		Activity	1647820799	Arena Challenge	{"M":{"Gold":{"N":"2000"},"XP":{"N":"1000"}}}	1647734400	Arena					
	EQUIPMENTS#001	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Sword of the Dragon	Weapon	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}							
	EQUIPMENTS#001EQUIPMENTS#002	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Armor of the Knight	Armor	{"M":{"DEF":{"N":"100"}}}							
	EQUIPMENTS#003	Type	EquipmentName	EquipmentType	EquipmentAttributes							
		Equipment	Ring of the Mage	Accessory	{"M":{"SP":{"N":"50"}}}							
	FRIENDS#player001	Type	FriendList									
		Friends	[{"M":{"FriendId":{"S":"player002"},"FriendName":{"S":"Alice"}}},{"M":{"FriendId":{"S":"player003"},"FriendName":{"S":"Bob"}}}]									
	ITEMS#001	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Health Potion	Consumable	5	{"M":{"HP":{"N":"50"}}}						
	ITEMS#002	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Armor of the Knight	Armor	1	{"M":{"DEF":{"N":"100"}}}						
	ITEMS#003	Type	ItemName	ItemType	ItemCount	ItemAttributes						
		Item	Sword of the Dragon	Weapon	1	{"M":{"ATK":{"N":"100"},"DEF":{"N":"50"}}}						
	TASK#001	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Find the Lost TreasureFind the Lost Treasure	Get clues from a lost adventurer and find the lost treasure.	InProgress	{"M":{"Gold":{"N":"100"},"XP":{"N":"50"}}}						
	TASK#002	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Defeat Magic Monsters	Go to the Magic Forest and defeat three magic monsters.	Completed	{"M":{"Gold":{"N":"200"},"XP":{"N":"100"}}}						
	TASK#003	Type	TaskName	TaskDescription	TaskStatus	TaskReward						
		Task	Rescue the Princess	Go to the Demon King's Castle and rescue the princess who is being held captive by the Demon King.	Available	{"M":{"Gold":{"N":"500"},"XP":{"N":"200"}}}						

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。
3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB の苦情管理システムのスキーマ設計

苦情管理システムのビジネスユースケース

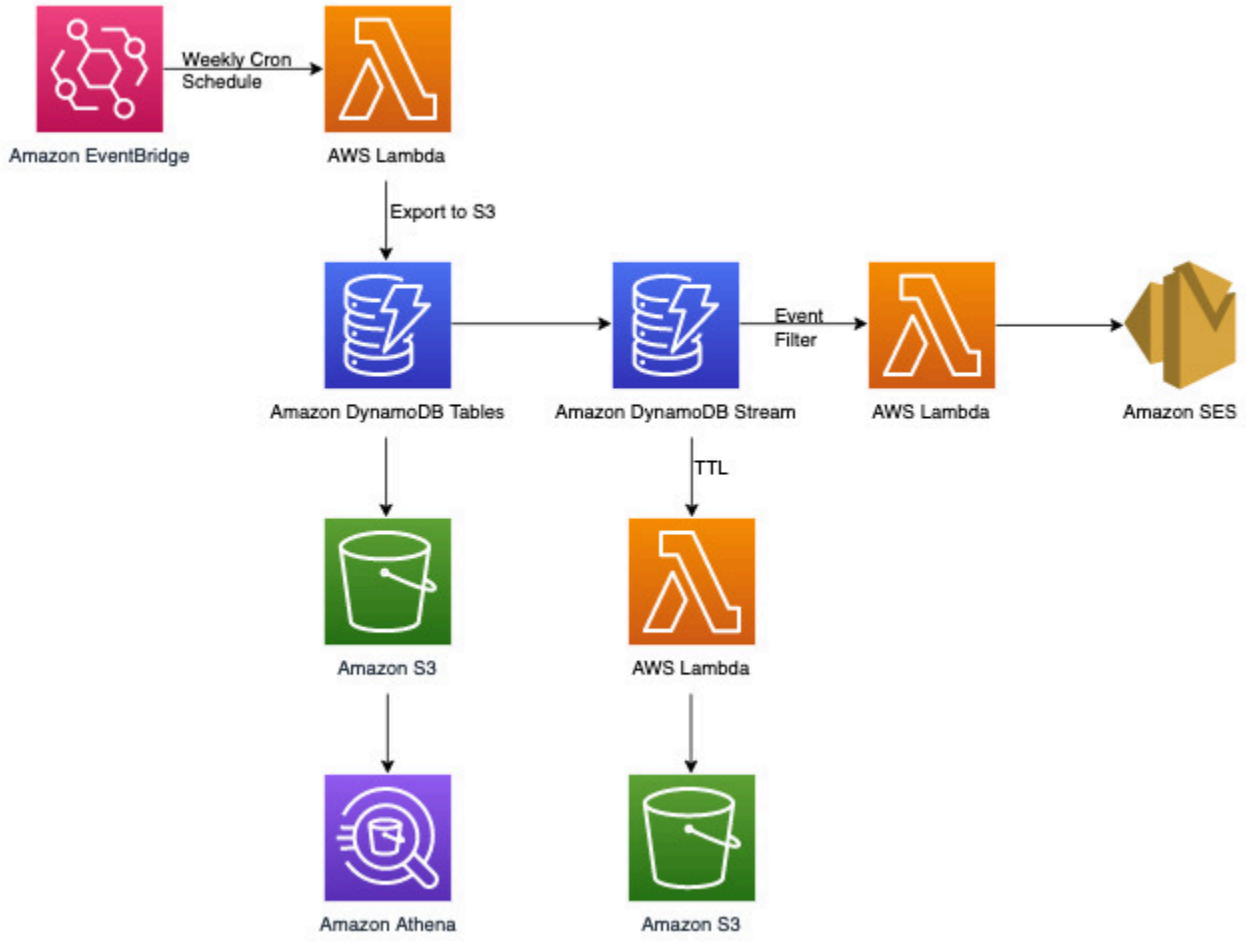
DynamoDB は、苦情管理システム (またはコンタクトセンター) のユースケースに最適なデータベースです。関連するほとんどのアクセスパターンが key-value ベースのトランザクションルックアップであるためです。このシナリオの一般的なアクセスパターンは、次のとおりです。

- 苦情を作成および更新する
- 苦情をエスカレートする
- 苦情に関するコメントを作成および閲覧する
- 顧客からのすべての苦情を取得する
- エージェントからのすべてのコメントを取得し、すべてのエスカレーションを取得する

一部のコメントには、苦情または解決策を説明する添付ファイルが付いている場合があります。これらはすべて key-value アクセスパターンですが、苦情に新しいコメントが追加されたときに通知を送信したり、分析クエリを実行して重大度 (またはエージェントのパフォーマンス) 別に苦情の分布を毎週調べたりするなど、追加の要件が伴う場合があります。ライフサイクル管理やコンプライアンスに関連するその他の要件としては、苦情を 3 年間記録した後に苦情データをアーカイブすることが挙げられます。

苦情管理システムのアーキテクチャ図

次の図は、苦情管理システムのアーキテクチャ図を示しています。この図は、苦情管理システムが使用するさまざまな AWS のサービス統合を示しています。



後述の DynamoDB データモデリングセクションで扱う key-value トランザクションアクセスパターンとは別に、非トランザクション要件が 3 つあります。上のアーキテクチャ図は、次の 3 つのワークフローに分けることができます。

1. 苦情に新しいコメントが追加されたときに通知を送信する
2. 週次データに対して分析クエリを実行する
3. 3 年より前のデータをアーカイブする

それぞれを詳しく見ていきましょう。

苦情に新しいコメントが追加されたときに通知を送信する

この要件を満たすには、次のワークフローを使用できます。



DynamoDB ストリームは、DynamoDB テーブルに対するすべての書き込みアクティビティを記録する変更データキャプチャメカニズムです。これらの変更の一部またはすべてに応じてトリガーされるように Lambda 関数を設定できます。Lambda トリガーにイベントフィルターを設定して、ユースケースに関係のないイベントを除外できます。この例では、フィルターを使用して、新しいコメントが追加されたときにのみ Lambda をトリガーし、関連する E メール ID (AWS Secret Manager またはその他の認証情報ストアから取得可能) に通知を送信できます。

週次データに対して分析クエリを実行する

DynamoDB は、オンライントランザクション処理 (OLTP) を主眼とするワークロードに適しています。分析が必要な他の 10～20% のアクセスパターンについては、マネージド機能の Amazon S3 へのエクスポートを使用してデータを S3 にエクスポートできます。この機能を使用しても、DynamoDB テーブルのライブトラフィックには影響がありません。次のワークフローをご覧ください。



Amazon EventBridge を使用すると、AWS Lambda をスケジュールどおりにトリガーできます。これにより、Lambda を定期的に呼び出すように cron 式を設定できます。Lambda は `ExportToS3` API コールを呼び出し、DynamoDB データを S3 に保存します。この S3 データ

に Amazon Athena などの SQL エンジンからアクセスし、テーブルのライブトランザクションワークロードには影響を与えることなく、DynamoDB データに対して分析クエリを実行できます。重大度レベル別に苦情数を調べる Athena クエリの例は、次のようになります。

```
SELECT Item.severity.S as "Severity", COUNT(Item) as "Count"
FROM "complaint_management"."data"
WHERE NOT Item.severity.S = ''
GROUP BY Item.severity.S ;
```

この Athena クエリは次の結果を返します。

Results (3)

Search rows

#	Severity	Count
1	P3	1
2	P2	2
3	P1	1

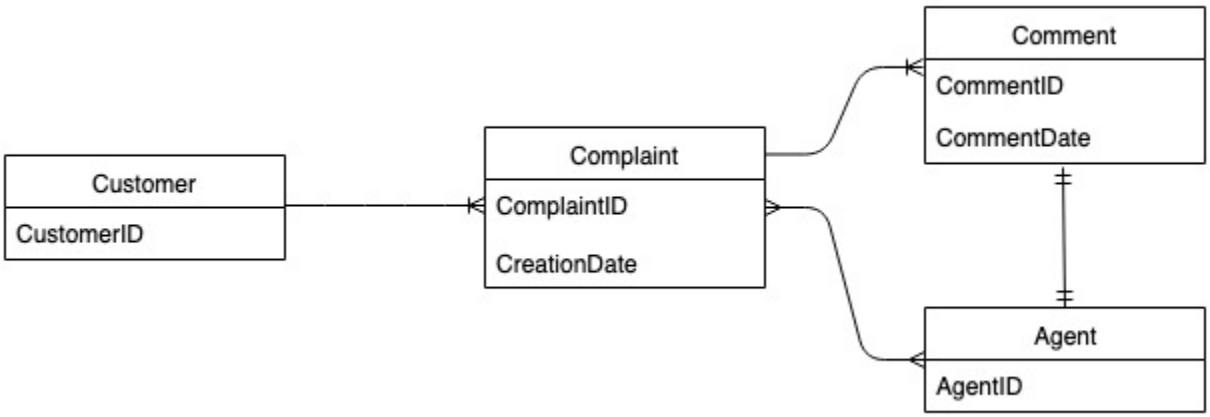
3 年より前のデータをアーカイブする

DynamoDB の Time to Live (TTL) 機能を使用して、追加費用なしで、DynamoDB テーブルから古いデータを削除できます (ただし、2019.11.21 (現行) バージョンのグローバルテーブルレプリカは除きます。TTL 削除が他のリージョンにレプリケートされると書き込み容量が消費されるためです)。このデータを DynamoDB ストリームで表示し、使用できます。その後に、Amazon S3 にアーカイブできます。この要件のワークフローは次のとおりです。



苦情管理システムのエンティティ関係図

次に示すのは、苦情管理システムのスキーマ設計に使用するエンティティ関係図 (ERD) です。



苦情管理システムのアクセスパターン

以下は、苦情管理システムのスキーマ設計で検討するアクセスパターンです。

1. createComplaint
2. updateComplaint
3. updateSeveritybyComplaintID
4. getComplaintByComplaintID
5. addCommentByComplaintID
6. getAllCommentsByComplaintID
7. getLatestCommentByComplaintID
8. getAComplaintbyCustomerIDAndComplaintID
9. getAllComplaintsByCustomerID
10. escalateComplaintByComplaintID
11. getAllEscalatedComplaints
12. getEscalatedComplaintsByAgentID (新しいものから古いものへの順)
13. getCommentsByAgentID (2 つの日付間)

苦情管理システムのスキーマ設計の進化

これは苦情管理システムであるため、ほとんどのアクセスパターンはプライマリエンティティとしての苦情を中心に展開します。`ComplaintID` のカーディナリティが高いと、基盤となるパーティションにデータが均等に分散されます。また、特定したアクセスパターンの最も一般的な検索条件となり

ます。したがって、ComplaintID は、このデータセットのパーティションキー候補として適しています。

ステップ 1: アクセスパターン 1 (**createComplaint**)、2 (**updateComplaint**)、3 (**updateSeveritybyComplaintID**)、4 (**getComplaintByComplaintID**) に対処する

「metadata」(または「AA」) という汎用ソートキー値を使用して、CustomerID、State、Severity、CreationDate などの苦情固有の情報を保存できます。PK=ComplaintID と SK="metadata" でシングルトンオペレーションを使用して、以下を行います。

1. PutItem で新しい苦情を作成する
2. UpdateItem で苦情メタデータの重大度やその他のフィールドを更新する
3. GetItem で苦情のメタデータを取得する

Primary key		Attributes				
Partition key: PK	Sort key: SK					
Complaint1321	metadata	customer_id	current_state	creation_time	severity	complaint_description
		custXYZ	assigned	2023-05-10T15:58:00	P2	<Complaint Description>

ステップ 2: アクセスパターン 5 (**addCommentByComplaintID**) に対処する

このアクセスパターンでは、苦情と苦情に関するコメントとの間に 1 対多リレーションシップモデルが必要です。ここでは、垂直パーティショニング手法を通じてソートキーを使用し、さまざまなタイプのデータを含む項目コレクションを作成します。アクセスパターン 6 (getAllCommentsByComplaintID) と 7 (getLatestCommentByComplaintID) に注目すると、コメントを時間順に並べ替える必要があることがわかります。また、複数のコメントを同時に受け取ることができるため、複合ソートキー手法を使用して時間と CommentID をソートキー属性に追加できます。

このようなコメント競合の可能性に対処する他のオプションとしては、タイムスタンプの粒度を増やすか、Comment_ID を使用する代わりに増分数値をサフィックスとして追加します。この場合、範囲ベースの操作を可能にするために、コメントに対応する項目のソートキー値の前に「comm#」を付けます。

また、苦情メタデータの currentState に、新しいコメントを追加したときの状態が反映されることを確認する必要があります。コメントの追加は、エージェントへの苦情の割り当てや苦情の解決などを示す場合があります。コメントの追加と現在の状態の更新をオールオアナッシング方式で苦情メ

タデータにバンドルするには、TransactWriteItems API を使用します。結果として得られるテーブルの状態は次のようになります。

Primary key		Attributes				
Partition key: PK	Sort key: SK					
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID
		comm3	2023-05-10T16:00:00	investigating	<Comment text>	AgentB
	metadata	customer_id	current_state	creation_time	severity	complaint_description
		custXYZ	investigating	2023-05-10T15:58:00	P2	<Complaint Description>

テーブルにさらにデータを追加し、PK とは別のフィールドとして ComplaintID も追加しましょう。ComplaintID に追加のインデックスが必要な場合に備えて、モデルを将来的に保証するためです。また、一部のコメントには添付ファイルが含まれている場合があることに注意してください。添付ファイルは、Amazon Simple Storage Service に保存し、その参照または URL のみを DynamoDB で管理します。コストとパフォーマンスを最適化するために、トランザクションデータベースはできるだけリーンな状態に保つことがベストプラクティスです。これで、データは次のようになります。

Primary key		Attributes					
Partition key: PK	Sort key: SK						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

ステップ 3: アクセスパターン 6 (**getAllCommentsByComplaintID**) と 7 (**getLatestCommentByComplaintID**) に対処する

苦情に関するすべてのコメントを取得するには、ソートキーに対する query オペレーションで begins_with 条件を使用できます。メタデータエントリを読み取るために追加の読み取り容量を消費したり、関連する結果をフィルタリングするためのオーバーヘッドを生じたりせずに、このようなソートキー条件を設定することで、必要なものだけを読み取ることができます。例えば、SK および PK=Complaint123 begins_with comm# を使用したクエリオペレーションは、メタデータエントリをスキップして、以下を返します。

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

パターン 7 (getLatestCommentByComplaintID) では苦情に関する最新のコメントが必要であるため、次の 2 つのクエリパラメータを追加しましょう。

1. 結果を降順にソートするために False に設定する ScanIndexForward
2. 最新のコメントを (1つだけ) 取得するために 1 に設定する Limit

アクセスパターン 6 (getAllCommentsByComplaintID) と同じように、ソートキー条件として begins_with comm# を使用してメタデータエントリを省略します。これで、クエリオペレーションで PK=Complaint123 と SK=begins_with comm# に加えて、ScanIndexForward=False と Limit を使用し、この設計でアクセスパターン 7 を実行できるようになりました。結果として、次のターゲット項目が返されます。

Partition key: PK	Sort key: SK	Attributes					
	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
Complaint123	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

さらにダミーデータをテーブルに追加してみましょう。

Primary key		Attributes					
Partition key: PK	**Sort key: SK**						
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA	
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID	
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB	
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>
Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text		
		comm4	2022-12-31T19:32:00	waiting	<comm text>		
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
Complaint0987	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>

ステップ 4: アクセスパターン 8 (**getAComplaintbyCustomerIDAndComplaintID**) と 9 (**getAllComplaintsByCustomerID**) に対処する

アクセスパターン 8 (getAComplaintbyCustomerIDAndComplaintID) と 9 (getAllComplaintsByCustomerID) は、新しい検索条件 CustomerID を導入します。これを既存のテーブルから取得するには、すべてのデータを読み取って当の CustomerID の関連項目をフィルタリングするために、コストの高い Scan が必要になります。この検索をより効

率的にするには、CustomerID をパーティションキーとしてグローバルセカンダリインデックス (GSI) を作成できます。顧客と苦情の 1 対多リレーションシップおよびアクセスパターン 9 (getAllComplaintsByCustomerID) を考慮すると、ComplaintID がソートキーの適切な候補となります。

GSI のデータは次のようになります。

Primary key		Attributes					
Partition key: customer_id	Sort key: complaint_id						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

この GSI のクエリ例はアクセスパターン 8 (getAComplaintbyCustomerIDAndComplaintID) で customer_id=custXYZ、sort key=Complaint1321 となります。結果は次のようになります。

Primary key		Attributes					
Partition key: customer_id	Sort key: complaint_id						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
custXYZ							
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

アクセスパターン 9 (getAllComplaintsByCustomerID) でお客様の苦情をすべて取得する場合、GSI のクエリはパーティションキー条件が customer_id=custXYZ になります。結果は次のようになります。

Primary key		Attributes					
Partition key: customer_id	Sort key: complaint_id						
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>

ステップ 5: アクセスパターン 10 (**escalateComplaintByComplaintID**) に対処する

このアクセスでは、エスカレーションのディメンションを導入します。苦情をエスカレーションするには、UpdateItem を使用して escalated_to や escalation_time などの属性を既存の苦情メタデータ項目に追加します。DynamoDB のスキーマ設計は柔軟であるため、キー以外の属性のセットをさまざまな項目間で均一にすることも、個別にすることもできます。次の例を参照してください。

UpdateItem with PK=Complaint1444, SK=metadata

Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text				
		comm4	2022-12-31T19:32:00	waiting	<comm text>				
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC		
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07

ステップ 6: アクセスパターン 11 (**getAllEscalatedComplaints**) と 12 (**getEscalatedComplaintsByAgentID**) に対処する

データセット全体からエスカレーションされると予想される苦情はほんの一握りです。したがって、エスカレーション関連の属性にインデックスを作成すると、効率的な検索と費用対効果の高い GSI ストレージが実現します。これを実現するには、スパースインデックス手法を活用できます。パーティションキーを `escalated_to`、ソートキーを `escalation_time` とする GSI は、次のようになります。

Primary key		Attributes							
Partition key: escalated_to	**Sort key: escalation_time**								
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

アクセスパターン 11 (`getAllEscalatedComplaints`) ですべてのエスカレーションされた苦情を取得するには、この GSI を単にスキャンします。このスキャンは、GSI のサイズにより、パフォーマンスとコスト効率が向上することに注意してください。特定のエージェント (アクセスパターン 12 (`getEscalatedComplaintsByAgentID`)) でエスカレーションされた苦情を取得する場合は、パーティションキーを `escalated_to=agentID` とし、`ScanIndexForward` を `False` に設定して新しいものから古いものへの順に取得します。

ステップ 7: アクセスパターン 13 (**`getCommentsByAgentID`**) に対処する

最後のアクセスパターンでは、新しいディメンション `AgentID` によるルックアップを実行する必要があります。また、2 つの日付間のコメントを読み取るには時間ベースの順序付けも必要であるため、パーティションキーを `agent_id`、ソートキーを `comm_date` として GSI を作成します。この GSI のデータは次のようになります。

Primary key		Attributes					
Partition key: agentID	**Sort key: comm_date**						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

この GSI のクエリの例は `partition key agentID=AgentA` および `sort key=comm_date between (2023-04-30T12:30:00, 2023-05-01T09:00:00)` となり、その結果は次のようになります。

Primary key		Attributes					
Partition key: agentID	Sort key: comm_date						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
createComplaint	ベーステーブル	PutItem	PK=complaint_id	SK=metadata	
updateComplaint	ベーステーブル	UpdateItem	PK=complaint_id	SK=metadata	
updateSeveritybyComplaintID	ベーステーブル	UpdateItem	PK=complaint_id	SK=metadata	
getComplaintByComplaintID	ベーステーブル	GetItem	PK=complaint_id	SK=metadata	

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
addCommentByComplaintID	ベーステーブル	TransactWriteItems	PK=complaint_id	SK=metadata, SK=comm#comm_date#comm_id	
getAllCommentsByComplaintID	ベーステーブル	クエリ	PK=complaint_id	SK begins_with "comm#"	
getLatestCommentByComplaintID	ベーステーブル	クエリ	PK=complaint_id	SK begins_with "comm#"	scan_index_forward=False, Limit 1
getAComplaintbyCustomerIDAndComplaintID	Customer_complaint_GSI	クエリ	customer_id=customer_id	complaint_id = complaint_id	
getAllComplaintsByCustomerID	Customer_complaint_GSI	クエリ	customer_id=customer_id	該当なし	
escalateComplaintByComplaintID	ベーステーブル	UpdateItem	PK=complaint_id	SK=metadata	
getAllEscalatedComplaints	Escalations_GSI	Scan	該当なし	該当なし	

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getEscalatedComplaintsByAgentID (新しいものから古いものへの順)	Escalations_GSI	クエリ	escalated_to=agent_id	該当なし	scan_index_forward=False
getCommentsByAgentID (2 つの日付間)	Agents_Comments_GSI	クエリ	agent_id=agent_id	SK between (date1, date2)	

苦情管理システムの最終スキーマ

最終的なスキーマ設計は次のとおりです。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の DynamoDB の例を参照してください。

ベーステーブル

Primary key		Attributes							
Partition key: PK	**Sort key: SK**								
Complaint123	comm#2023-04-30T12:00:24#comm1	comm_id	comm_date	complaint_state	comm_text	agentID			
		comm1	2023-04-30T12:00:24	investigating	<comm text>	AgentA			
	comm#2023-04-30T12:35:54#comm2	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
		comm2	2023-04-30T12:35:54	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]	AgentA		
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description		
		custABC	Complaint123	resolved	2023-04-30T12:00:00	P2	<description text>		
Complaint1321	comm#2023-05-10T16:00:00#comm3	comm_id	comm_date	complaint_state	comm_text	agentID			
		comm3	2023-05-10T16:00:00	investigating	<comm text>	AgentB			
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB	2023-05-15T14:00:00
Complaint1444	comm#2022-12-31T19:32:00#comm4	comm_id	comm_date	complaint_state	comm_text				
		comm4	2022-12-31T19:32:00	waiting	<comm text>				
	comm#2022-12-31T19:40:00#comm5	comm_id	comm_date	complaint_state	comm_text	attachments	agentID		
		comm5	2022-12-31T19:40:00	assigned	<comm text>	["s3://URL_for_attachment1"]	AgentC		
	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07
Complaint0987	metadata	customer_id	complaint_id	current_state	creation_time	severity	complaint_description		
		custXYZ	Complaint0987	assigned	2023-06-10T12:30:08	P3	<description text>		

Customer_Complaint_GSI

Primary key		Attributes							
Partition key: customer_id	**Sort key: complaint_id**								
custABC	Complaint123	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint123	metadata	resolved	2023-04-30T12:00:00	P2	<description text>		
custXY32	Complaint1444	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1444	metadata	assigned	2022-12-31T19:39:57	P1	<description text>	AgentB	2023-01-03T04:00:07
custXYZ	Complaint0987	PK	SK	current_state	creation_time	severity	complaint_description		
		Complaint0987	metadata	assigned	2023-06-10T12:30:08	P3	<description text>		
	Complaint1321	PK	SK	current_state	creation_time	severity	complaint_description	escalated_to	escalation_time
		Complaint1321	metadata	investigating	2023-05-10T15:58:00	P2	<descr_text>	AgentB	2023-05-15T14:00:00

Escalations_GSI

Primary key		Attributes							
Partition key: escalated_to	Sort key: escalation_time								
AgentB	2023-01-03T04:00:07	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1444	metadata	custXY32	Complaint1444	assigned	2022-12-31T19:39:57	P1	<description text>
	2023-05-15T14:00:00	PK	SK	customer_id	complaint_id	current_state	creation_time	severity	complaint_description
		Complaint1321	metadata	custXYZ	Complaint1321	investigating	2023-05-10T15:58:00	P2	<descr_text>

Agents_Comments_GSI

Primary key		Attributes					
Partition key: agentID	Sort key: comm_date						
AgentA	2023-04-30T12:00:24	PK	SK	comm_id	complaint_state	comm_text	
		Complaint123	comm#2023-04-30T12:00:24#comm1	comm1	investigating	<comm text>	
	2023-04-30T12:35:54	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint123	comm#2023-04-30T12:35:54#comm2	comm2	resolved	<comm text>	["s3://URL_for_attachment1","s3://URL_for_attachment2"]
AgentB	2023-05-10T16:00:00	PK	SK	comm_id	complaint_state	comm_text	
		Complaint1321	comm#2023-05-10T16:00:00#comm3	comm3	investigating	<comm text>	
AgentC	2022-12-31T19:40:00	PK	SK	comm_id	complaint_state	comm_text	attachments
		Complaint1444	comm#2022-12-31T19:40:00#comm5	comm5	assigned	<comm text>	["s3://URL_for_attachment1"]

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。
3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB での定期払いスキーマの設計

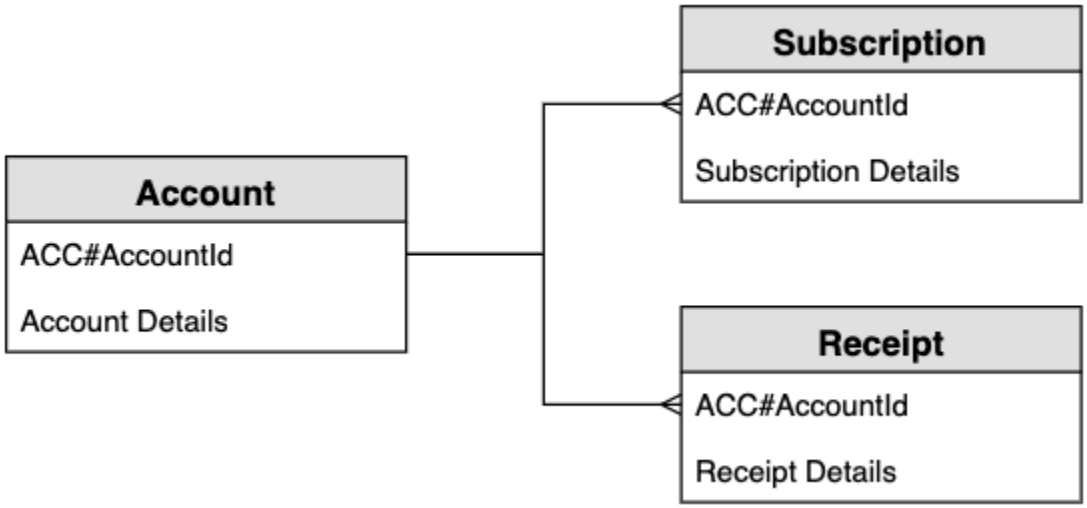
定期払いのビジネスユースケース

このユースケースでは、DynamoDB を使用して定期払いシステムを実装する方法について説明します。データモデルには、アカウント、サブスクリプション、領収書のエンティティがあります。このユースケースの詳細は次のとおりです。

- アカウントごとに複数のサブスクリプションを持つことができます。
- サブスクリプションは、次の支払いを処理する必要があるときは `NextPaymentDate`、E メールによるリマインダーをお客様に送信するときは `NextReminderDate` です。
- サブスクリプションには、支払いが処理されると、保存および更新される項目があります (平均の項目サイズは約 1 KB で、スループットはアカウントとサブスクリプションの数によって異なります)。
- 支払いプロセッサは、処理の一環として領収書も作成します。領収書は、テーブルに保存され、TTL 属性を使用して一定期間後に期限切れになるように設定されます。

定期払いエンティティ関係図

これは、定期払いシステムのスキーマ設計に使用するエンティティ関係図 (ERD) です。



定期払いシステムのアクセスパターン

これらは、定期払いシステムのスキーマ設計で検討するアクセスパターンです。

1. `createSubscription`

2. createReceipt
3. updateSubscription
4. getDueRemindersByDate
5. getDuePaymentsByDate
6. getSubscriptionsByAccount
7. getReceiptsByAccount

定期払いのスキーマ設計

一般的な名前である PK や SK は、アカウント、サブスクリプション、領収書などの異なる種類のエンティティを同じテーブルに保存するためのキー属性に使用します。ユーザーは最初にサブスクリプションを作成します。サブスクリプションにより、ユーザーは製品の代金として毎月同じ日に金額を支払うことに同意します。月内のどの日に支払いを処理するかは、ユーザーが選択できます。支払いを処理する前にリマインダーも送信されます。このアプリケーションは、毎日 2 つのバッチジョブを実行することで機能します。1 つのバッチジョブでは、当日を期限とするリマインダーを送信し、もう 1 つのバッチジョブでは、当日を支払期日とするすべての支払いを処理します。

ステップ 1: アクセスパターン 1 (**createSubscription**) に対処する

アクセスパターン 1 (createSubscription) を使用してサブスクリプションを最初に作成し、SKU、NextPaymentDate、NextReminderDate、PaymentDetails などの詳細を設定します。このステップでは、1 つのサブスクリプションで 1 つのアカウントに限り、テーブルの状態を表示します。項目コレクションには複数のサブスクリプションを含めることができるため、これは 1 対多リレーションシップになります。

Primary key		Attributes									
Partition key: PK	Sort key: SK										
		Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	1970-01-01T00:00:00.000Z	2023-05-28	1970-01-01T00:00:00.000Z	2023-05-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

ステップ 2: アクセスパターン 2 (**createReceipt**) と 3 (**updateSubscription**) に対処する

アクセスパターン 2 (createReceipt) を使用して領収書項目を作成します。毎月支払いを処理すると、支払いプロセッサは領収書をベーステーブルに書き戻します。項目コレクションには複数の領収書を含めることができるため、これは 1 対多リレーションシップです。支払いプロセッサは、サブスクリプション項目 (アクセスパターン 3 (updateSubscription)) も更新し、翌月の NextReminderDate または NextPaymentDate に変更します。

Primary key		Attributes									
Partition key: PK	Sort key: SK										
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999	Email	SKU	ProcessedDate	ProcessedAmount	TTL					
		s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200					
	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

ステップ 3: アクセスパターン 4 (**getDueRemindersByDate**) に対処する

アプリケーションは、当日の支払いのリマインダーをバッチで処理します。そのため、アプリケーションは別のディメンション (アカウントではなく日付) でサブスクリプションにアクセスする必要があります。これは、グローバルセカンダリインデックス (GSI) に適したユースケースです このステップでは、インデックス GSI-1 を追加し、NextReminderDate を GSI パーティションキーとして使用します。すべての項目をレプリケートする必要はありません。この GSI はスパースインデックスであり、領収書項目はレプリケートされません。また、すべての属性を射影する必要はなく、属性の一部を含めるだけで済みます。次の図に示す GSI-1 のスキーマは、アプリケーションがリマインダー E メールを送信するために必要な情報を提供します。

Primary key		Attributes				
Partition key: NextReminderDate	Sort key: LastReminderDate					
2023-06-21	2023-05-21T14:15:39.247Z	SK	PK	SKU	Email	NextPaymentDate
		SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28

ステップ 4: アクセスパターン 5 (**getDuePaymentsByDate**) に対処する

アプリケーションは、リマインダーの場合と同じように、当日の支払いをバッチで処理します。このステップでは GSI-2 を追加し、NextPaymentDate を GSI パーティションキーとして使用します。すべての項目をレプリケートする必要はありません。GSI はスパースインデックスであり、領収書項目はレプリケートされません。次の図は GSI-2 のスキーマを示しています。

Primary key		Attributes						
Partition key: NextPaymentDate	Sort key: LastPaymentDate							
2023-06-28	2023-05-18T14:15:39.247Z	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
		ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}

ステップ 5: アクセスパターン 6 (**getSubscriptionsByAccount**) と 7 (**getReceiptsByAccount**) に対処する

アプリケーションは、ベーステーブルでアカウント識別子 (PK) をターゲットとするクエリを使用して、アカウントのすべてのサブスクリプションを取得できます。また、範囲演算子を使用して SK が「SUB#」で始まるすべての項目を取得できます。また、アプリケーションは、同じクエ

リ構造を使用してすべての領収書を取得し、範囲演算子を使用して SK が「REC#」で始まるすべての項目を取得できます。これにより、アクセスパターン 6 (getSubscriptionsByAccount) と 7 (getReceiptsByAccount) に対処できます。これらのアクセスパターンをアプリケーションで使用することで、ユーザーは現在のサブスクリプションと過去 6 か月間の領収書を確認できます。このステップでは、テーブルスキーマに変更はありません。アクセスパターン 6 (getSubscriptionsByAccount) でサブスクリプション項目だけをターゲットにする方法は次の表で確認できます。

Primary key		Attributes									
Partition key: PK	Sort key: SK										
	REC#12023-05-28T14:15:39.24#SKU#999	Email	SKU	ProcessedDate	ProcessedAmount	TTL					
		s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200					
ACC#123	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値
CreateSubscription	ベーステーブル	PutItem	ACC#account_id	SUB#<SUBID>#SKU<SKUID>
createReceipt	ベーステーブル	PutItem	ACC#account_id	REC#<ReceiptDate>#SKU<SKUID>
updateSubscription	ベーステーブル	UpdateItem	ACC#account_id	SUB#<SUBID>#SKU<SKUID>
getDueRemindersByDate	GSI-1	クエリ	<NextReminderDate>	
getDuePaymentsByDate	GSI-2	クエリ	<NextPaymentDate>	

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値
getSubscriptionsByAccount	ベーステーブル	クエリ	ACC#account_id	SK begins_with “SUB#”
getReceiptsByAccount	ベーステーブル	クエリ	ACC#account_id	SK begins_with “REC#”

Recurring payments final schema

最終的なスキーマ設計は次のとおりです。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の DynamoDB の例を参照してください。

ベーステーブル

Primary key		Attributes										
Partition key: PK	Sort key: SK											
ACC#123	REC#12023-05-28T14:15:39.24#SKU#999	Email	SKU	ProcessedDate	ProcessedAmount	TTL						
		s@s.com	999	2023-05-28T14:15:39.247Z	12.99	1700318200						
	SUB#123#SKU#999	Email	PaymentDay	PaymentAmount	LastPaymentDate	NextPaymentDate	LastReminderDate	NextReminderDate	SKU	PaymentDetails	CreatedDate	
		s@s.com	28	12.99	2023-05-18T14:15:39.247Z	2023-06-28	2023-05-21T14:15:39.247Z	2023-06-21	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}	2023-05-18T09:41:25.856Z	

GSI-1

Primary key		Attributes				
Partition key: NextReminderDate	Sort key: LastReminderDate	SK	PK	SKU	Email	NextPaymentDate
2023-06-21	2023-05-21T14:15:39.247Z	SUB#123#SKU#999	ACC#123	999	s@s.com	2023-06-28

GSI-2

Primary key		Attributes						
Partition key: NextPaymentDate	Sort key: LastPaymentDate	PK	SK	Email	PaymentDay	PaymentAmount	SKU	PaymentDetails
2023-06-28	2023-05-18T14:15:39.247Z	ACC#123	SUB#123#SKU#999	s@s.com	28	12.99	999	{"default-card":{"S":"1234123412341234"},"default-address":{"S":"12 Bridge Street, Birmingham, B12 7ST"}}

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。
3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB でのデバイスステータス更新のモニタリング

このユースケースでは、DynamoDB を使用して、DynamoDB 内でデバイスのステータスの更新 (またはデバイスのステータスの変更) をモニタリングする方法について説明しています。

ユースケース

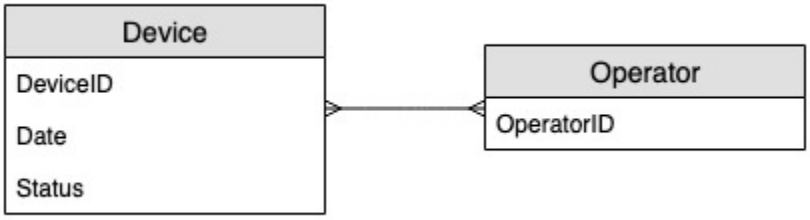
IoT のユースケース (スマートファクトリーなど) では、多くのデバイスをオペレーターがモニタリングする必要があり、オペレーターはステータスやログをモニタリングシステムに定期的に送信します。デバイスに問題が発生すると、デバイスのステータスが [正常] から [警告] に変わります。デバイスの異常な動作の重大度やタイプによって、さまざまなログレベルまたはステータスがあります。その後、システムはデバイスをチェックするオペレーターを割り当て、オペレーターは必要に応じて問題を上司にエスカレーションします。

このシステムの典型的なアクセスパターンには次のものがあります。

- デバイスのログエントリを作成する
- 特定のデバイスステータスのすべてのログを取得し、最新のログを最初に表示する
- 2 つの日付間の特定のオペレーターのすべてのログを取得する
- 特定のスーパーバイザーのエスカレーションされたログをすべて取得する
- 特定のスーパーバイザーの特定のデバイスステータスに関するすべてのエスカレーションログを取得
- 特定の日付における特定のスーパーバイザーの特定のデバイスステータスを含むすべてのエスカレーションされたログを取得する

エンティティ関係図

これは、デバイスステータスの更新をモニタリングするために使用するエンティティ関係図 (ERD) です。



アクセスパターン

デバイスのステータス更新をモニタリングする方法について説明します。

1. `createLogEntryForSpecificDevice`
2. `getLogsForSpecificDevice`
3. `getWarningLogsForSpecificDevice`
4. `getLogsForOperatorBetweenTwoDates`
5. `getEscalatedLogsForSupervisor`
6. `getEscalatedLogsWithSpecificStatusForSupervisor`
7. `getEscalatedLogsWithSpecificStatusForSupervisorForDate`

スキーマ設計の進化

ステップ 1: アクセスパターン 1 (**`createLogEntryForSpecificDevice`**) と 2 (**`getLogsForSpecificDevice`**) に対処する

デバイス追跡システムのスケーリングの単位は、個々のデバイスです。このシステムでは、`deviceID` はデバイスを一意に識別します。これにより、`deviceID` がパーティションキーの適切な候補となります。各デバイスは定期的に (例えば 5 分おきに) 情報を追跡システムに送信します。この順序によって、日付が論理的なソート基準となり、したがってソートキーになります。この場合のサンプルデータは次のようになります。

Primary key		Attributes
Partition key: DeviceID	Sort key: State#Date	
d#12345	2020-04-24T14:40:00	State
		WARNING1
	2020-04-24T14:45:00	State
		WARNING1
	2020-04-24T14:50:00	State
		WARNING1
	2020-04-24T14:55:00	State
		NORMAL
d#54321	2020-04-11T05:50:00	State
		WARNING3
	2020-04-11T05:55:00	State
		WARNING3
	2020-04-11T06:00:00	State
		NORMAL
	2020-04-11T09:25:00	State
		WARNING2
	2020-04-11T09:30:00	State
		NORMAL
d#11223	2020-04-27T16:10:00	State
		WARNING4
	2020-04-27T16:15:00	State
		WARNING4

特定のデバイスのログエントリを取得するには、パーティションキー DeviceID="d#12345" を使用してクエリオペレーションを実行します。

ステップ 2: アクセスパターン 3 (**getWarningLogsForSpecificDevice**) に対処する

State は非キー属性であるため、現在のスキーマでアクセスパターン 3 に対処するにはフィルター式が必要になります。DynamoDB では、フィルター式はキー条件式を使用してデータを読み取った後に適用されます。例えば、d#12345 の警告ログを取得する場合、パーティションキー DeviceID="d#12345" を使用したクエリオペレーションは、上記のテーブルから 4 つの項目を読み取り、警告ステータスなしで 1 つの項目をフィルターで除外します。このアプローチは大規模な場合は効率的ではありません。フィルター式は、除外される項目の割合が低い場合や、クエリの実行頻度が低い場合に、クエリ対象の項目を除外するのに適した方法です。ただし、テーブルから多くの

項目が取得され、その大半の項目がフィルターで除外される場合は、より効率的に実行されるようにテーブル設計を進化させ続けることができます。

複合ソートキーを使用して、このアクセスパターンを処理する方法を変更してみましょう。ソートキーを State#Date に変更したサンプルデータを DeviceStateLog_3.json からインポートできます。このソートキーは、属性 State、#、および Date から構成されます。この例では、# は区切り文字として使用されています。これで、データは次のようになります。

Primary key	
Partition key: DeviceID	**Sort key: State#Date**
d#12345	NORMAL#2020-04-24T14:55:00
	WARNING1#2020-04-24T14:40:00
	WARNING1#2020-04-24T14:45:00
	WARNING1#2020-04-24T14:50:00

デバイスの警告ログのみを取得する場合は、このスキーマの方がクエリの対象が絞り込まれます。クエリのキー条件にはパーティションキー DeviceID="d#12345" とソートキー State#Date begins_with “WARNING” が使用されます。このクエリは、警告ステータスの関連する 3 つの項目のみを読み取ります。

ステップ 3: アクセスパターン 4 (**getLogsForOperatorBetweenTwoDates**) に対処する

Operator 属性がサンプルデータとともに DeviceStateLog テーブルに追加された DeviceStateLog_4.jsonD をインポートできます。

Primary key		Attributes		
Partition key: DeviceID	**Sort key: State#Date**			
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State
		Liz	2020-04-24T14:55:00	NORMAL
	WARNING1#2020-04-24T14:40:00	Operator	Date	State
		Liz	2020-04-24T14:40:00	WARNING1
	WARNING1#2020-04-24T14:45:00	Operator	Date	State
		Liz	2020-04-24T14:45:00	WARNING1
	WARNING1#2020-04-24T14:50:00	Operator	Date	State
		Liz	2020-04-24T14:50:00	WARNING1
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State
		Liz	2020-04-11T06:00:00	NORMAL
	NORMAL#2020-04-11T09:30:00	Operator	Date	State
		Sue	2020-04-11T09:30:00	NORMAL
	WARNING2#2020-04-11T09:25:00	Operator	Date	State
		Sue	2020-04-11T09:25:00	WARNING2
	WARNING3#2020-04-11T05:50:00	Operator	Date	State
		Sue	2020-04-11T05:50:00	WARNING3
	WARNING3#2020-04-11T05:55:00	Operator	Date	State
		Liz	2020-04-11T05:55:00	WARNING3
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State
		Sue	2020-04-27T16:10:00	WARNING4
	WARNING4#2020-04-27T16:15:00	Operator	Date	State
		Sue	2020-04-27T16:15:00	WARNING4

Operator は現在パーティションキーではないため、OperatorID に基づいてこのテーブルのキーと値を直接検索する方法はありません。OperatorID にグローバルセカンダリインデックスを使用

して新しい項目コレクションを作成する必要があります。アクセスパターンでは日付に基づく検索が必要なため、日付がグローバルセカンダリインデックス (GSI) のソートキー属性になります。GSI は現在次のようになっています。

Primary key		Attributes		
Partition key: Operator	Sort key: Date			
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

アクセスパターン 4 (getLogsForOperatorBetweenTwoDates) の場合、2020-04-11T05:58:00 と 2020-04-24T14:50:00 の間のパーティションキー OperatorID=Liz とソートキー Date を使用してこの GSI をクエリできます。

Primary key		Attributes		
Partition key: Operator	**Sort key: Date**			
	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
Liz	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

ステップ 4: アクセスパターン 5 (**getEscalatedLogsForSupervisor**)、6 (**getEscalatedLogsWithSpecificStatusForSupervisor**)、および 7 (**getEscalatedLogsWithSpecificStatusForSupervisorForDate**) に対処する

スパースインデックスを使用して、これらのアクセスパターンに対処します。

グローバルセカンダリインデックスはデフォルトではスパースであるため、インデックスのプライマリキー属性を含むベーステーブルの項目だけが実際にインデックスに表示されます。これは、モデル化するアクセスパターンに関係のない項目を除外するもう 1 つの方法です。

`EscalatedTo` 属性がサンプルデータとともに `DeviceStateLog` テーブルに追加された DeviceStateLog_6.json をインポートできます。前述のように、すべてのログがスーパーバイザーにエスカレーションされるわけではありません。

Primary key		Attributes			
Partition key: DeviceID	**Sort key: State#Date**				
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State	
		Liz	2020-04-24T14:55:00	NORMAL	
	WARNING1#2020-04-24T14:40:00	Operator	Date	State	
		Liz	2020-04-24T14:40:00	WARNING1	
	WARNING1#2020-04-24T14:45:00	Operator	Date	State	
		Liz	2020-04-24T14:45:00	WARNING1	
	WARNING1#2020-04-24T14:50:00	Operator	Date	State	
		Liz	2020-04-24T14:50:00	WARNING1	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State	
		Liz	2020-04-11T06:00:00	NORMAL	
	NORMAL#2020-04-11T09:30:00	Operator	Date	State	
		Sue	2020-04-11T09:30:00	NORMAL	
	WARNING2#2020-04-11T09:25:00	Operator	Date	State	
		Sue	2020-04-11T09:25:00	WARNING2	
	WARNING3#2020-04-11T05:50:00	Operator	Date	State	
		Sue	2020-04-11T05:50:00	WARNING3	
	WARNING3#2020-04-11T05:55:00	Operator	Date	State	
		Liz	2020-04-11T05:55:00	WARNING3	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State	
		Sue	2020-04-27T16:10:00	WARNING4	
	WARNING4#2020-04-27T16:15:00	Operator	Date	State	EscalatedTo
		Sue	2020-04-27T16:15:00	WARNING4	Sara

これで、EscalatedTo がパーティションキー、State#Date がソートキーである新しい GSI を作成できるようになりました。EscalatedTo 属性と State#Date 属性の両方を持つ項目のみがインデックスに表示されることに注意してください。

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date				
		DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

その他のアクセスパターンは、以下のようにまとめられています。

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
createLogEntryForSpecificDevice	ベーステーブル	PutItem	DeviceID=deviceId	State#Date=state#date	
getLogsForSpecificDevice	ベーステーブル	クエリ	DeviceID=deviceId	State#Date begins_with "state1#"	ScanIndexForward = False
getWarningLogsForSpecificDevice	ベーステーブル	クエリ	DeviceID=deviceId	State#Date begins_with "WARNING"	
getLogsForOperatorBetweenTwoDates	GSI-1	クエリ	Operator=operatorName	Date between date1 and date2	
getEscalatedLogsForSupervisor	GSI-2	クエリ	EscalatedTo=supervisorName		

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値	その他の条件/フィルター
getEscalatedLogsWithSpecificStatusForSupervisor	GSI-2	クエリ	EscalatedTo=supervisorName	State#Date begins_with "state1#"	
getEscalatedLogsWithSpecificStatusForSupervisorForDate	GSI-2	クエリ	EscalatedTo=supervisorName	State#Date begins_with "state1#date1"	

最終スキーマ

最終的なスキーマ設計は次のとおりです。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の DynamoDB の例を参照してください。

ベーステーブル

Primary key		Attributes			
Partition key: DeviceID	**Sort key: State#Date**				
d#12345	NORMAL#2020-04-24T14:55:00	Operator	Date	State	
		Liz	2020-04-24T14:55:00	NORMAL	
	WARNING1#2020-04-24T14:40:00	Operator	Date	State	
		Liz	2020-04-24T14:40:00	WARNING1	
	WARNING1#2020-04-24T14:45:00	Operator	Date	State	
		Liz	2020-04-24T14:45:00	WARNING1	
	WARNING1#2020-04-24T14:50:00	Operator	Date	State	
		Liz	2020-04-24T14:50:00	WARNING1	
d#54321	NORMAL#2020-04-11T06:00:00	Operator	Date	State	
		Liz	2020-04-11T06:00:00	NORMAL	
	NORMAL#2020-04-11T09:30:00	Operator	Date	State	
		Sue	2020-04-11T09:30:00	NORMAL	
	WARNING2#2020-04-11T09:25:00	Operator	Date	State	
		Sue	2020-04-11T09:25:00	WARNING2	
	WARNING3#2020-04-11T05:50:00	Operator	Date	State	
		Sue	2020-04-11T05:50:00	WARNING3	
	WARNING3#2020-04-11T05:55:00	Operator	Date	State	
		Liz	2020-04-11T05:55:00	WARNING3	
d#11223	WARNING4#2020-04-27T16:10:00	Operator	Date	State	
		Sue	2020-04-27T16:10:00	WARNING4	
	WARNING4#2020-04-27T16:15:00	Operator	Date	State	EscalatedTo
		Sue	2020-04-27T16:15:00	WARNING4	Sara

GSI-1

Primary key		Attributes		
Partition key: Operator	**Sort key: Date**			
Liz	2020-04-11T05:55:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:55:00	WARNING3
	2020-04-11T06:00:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T06:00:00	NORMAL
	2020-04-24T14:40:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:40:00	WARNING1
	2020-04-24T14:45:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:45:00	WARNING1
	2020-04-24T14:50:00	DeviceID	State#Date	State
		d#12345	WARNING1#2020-04-24T14:50:00	WARNING1
	2020-04-24T14:55:00	DeviceID	State#Date	State
		d#12345	NORMAL#2020-04-24T14:55:00	NORMAL
Sue	2020-04-11T05:50:00	DeviceID	State#Date	State
		d#54321	WARNING3#2020-04-11T05:50:00	WARNING3
	2020-04-11T09:25:00	DeviceID	State#Date	State
		d#54321	WARNING2#2020-04-11T09:25:00	WARNING2
	2020-04-11T09:30:00	DeviceID	State#Date	State
		d#54321	NORMAL#2020-04-11T09:30:00	NORMAL
	2020-04-27T16:10:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:10:00	WARNING4
	2020-04-27T16:15:00	DeviceID	State#Date	State
		d#11223	WARNING4#2020-04-27T16:15:00	WARNING4

GSI-2

Primary key		Attributes			
Partition key: EscalatedTo	Sort key: State#Date	DeviceID	Operator	Date	State
Sara	WARNING4#2020-04-27T16:15:00	d#11223	Sue	2020-04-27T16:15:00	WARNING4

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。
3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB をオンラインショップのデータストアとして使用する

このユースケースでは、オンラインショップ (または e ストア) のデータストアとして DynamoDB を使用する方法について説明しています。

ユースケース

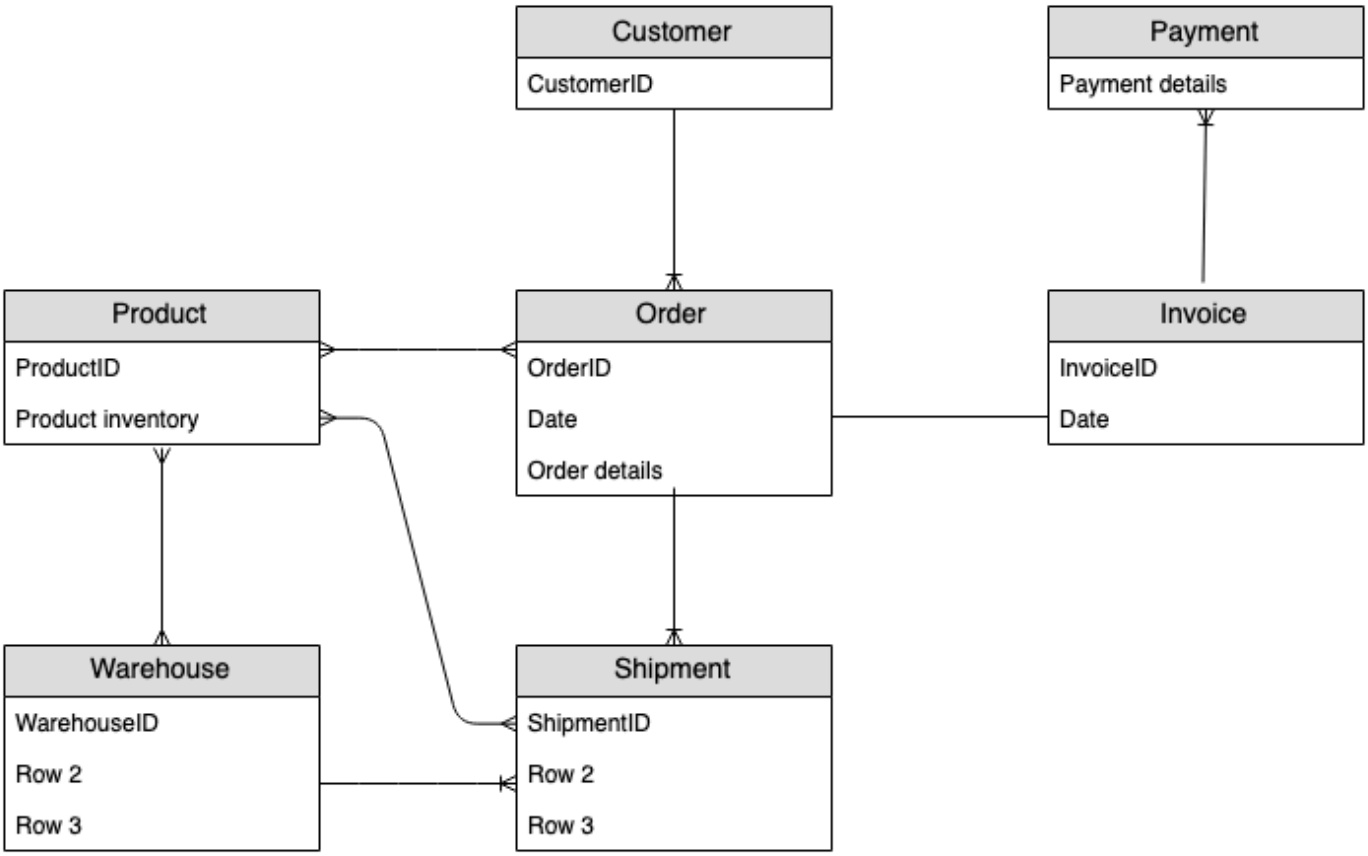
オンラインストアでは、ユーザーはさまざまな製品を閲覧し、最終的に購入することができます。生成された請求書に基づいて、お客様はディスカウントコードまたはギフトカードを使用して支払い、残りの金額をクレジットカードで支払うことができます。購入した商品は、複数の倉庫から集荷され、指定された住所に発送されます。オンラインストアの一般的なアクセスパターンは以下のとおりです。

- 指定された customerId のお客様を取得する

- 指定された productId の製品を取得する
- 指定された warehouseId の倉庫を取得する
- productId で全倉庫の商品在庫を取得する
- 指定された OrderId の注文を取得する
- 指定された OrderID のすべての商品を取得する
- 指定された OrderID の請求書を取得する
- 指定された OrderID のすべての出荷情報を取得する
- 指定された日付範囲における指定された productId のすべての注文を取得する
- 指定された invoiceId の請求書を取得する
- 指定された InvoiceID のすべての支払いを取得する
- 指定された shipmentId の出荷詳細を取得する
- 指定された warehouseId のすべての出荷情報を取得する
- 指定された warehouseId のすべての商品の在庫を取得する
- 指定された日付範囲における指定された customerId のすべての請求書を取得する
- 指定された日付範囲で指定された customerId で注文されたすべての製品を取得する

エンティティ関係図

これは、オンラインショップのデータストアとして DynamoDB をモデル化するために使用するエンティティ関係図 (ERD) です。



アクセスパターン

以上が、DynamoDB をオンラインショップのデータストアとして利用する場合のアクセスパターンです。

1. `getCustomerByCustomerId`
2. `getProductByProductId`
3. `getWarehouseByWarehouseId`
4. `getProductInventoryByProductId`
5. `getOrderDetailsByOrderId`
6. `getProductByOrderId`
7. `getInvoiceByOrderId`
8. `getShipmentByOrderId`
9. `getOrderByProductIdForDateRange`
10. `getInvoiceByInvoiceId`

11getPaymentByInvoiceId

12getShipmentDetailsByShipmentId

13getShipmentByWarehouseId

14getProductInventoryByWarehouseId

15getInvoiceByCustomerIdForDateRange

16getProductsByCustomerIdForDateRange

スキーマ設計の進化

DynamoDB 用の NoSQL Workbench を使用して、AnOnlineShop_1.json をインポートし、AnOnlineShop という新しいデータモデルと OnlineShop という新しいテーブルを作成します。パーティションキーとソートキーには一般的な名前 PK と SK を使用することに注意してください。これは、さまざまなタイプのエンティティを同じテーブルに保存するための方法です。

ステップ 1: アクセスパターン 1 (**getCustomerByCustomerId**) に対処する

AnOnlineShop_2.json をインポートして、アクセスパターン 1 (getCustomerByCustomerId) を処理します。エンティティの中には他のエンティティとリレーションシップを持たないものもあるため、それらには同じ値の PK と SK を使用します。データ例では、customerId を後で追加される他のエンティティから区別するために、キーに接頭辞 c# が使われていることに注意してください。この方法は他のエンティティでも繰り返されます。

このアクセスパターンに対処するために、GetItem オペレーションを PK=customerId と SK=customerId で使用できます。

ステップ 2: アクセスパターン 2 (**getProductByProductId**) に対処する

AnOnlineShop_3.json をインポートして、product エンティティのアクセスパターン 2 (getProductByProductId) に対処します。製品エンティティには p# というプレフィックスが付けられ、同じソートキー属性が customerID と productID の保存にも使われています。一般的な命名規則と垂直パーティショニングによって、このような項目コレクションを作成し、効果的な単一テーブル設計を実現できます。

このアクセスパターンに対処するため、GetItem オペレーションは PK=productId および SK=productId で使用できます。

ステップ 3: アクセスパターン 3 (**getWarehouseByWarehouseId**) に対処する

AnOnlineShop_4.json をインポートして、warehouse エンティティのアクセスパターン 3 (getWarehouseByWarehouseId) に対処します。現在、customer、product、および warehouse エンティティが同じテーブルに追加されています。これらはプレフィックスと EntityType 属性を使用して区別されます。タイプ属性 (またはプレフィックスの命名) は読みやすさを向上させます。さまざまなエンティティの英数字 ID を同じ属性に保存するだけでは、読みやすさに影響が出ます。これらの識別子がないと、あるエンティティを別のエンティティと区別するのが難しくなります。

このアクセスパターンに対処するため、GetItem オペレーションは PK=warehouseId および SK=warehouseId で使用できます。

ベーステーブル –

Primary key		Attributes		
Partition key: PK	Sort key: SK			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
p#12345	p#12345	EntityType	Detail	Price
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}	

ステップ 4: アクセスパターン 4 (**getProductInventoryByProductId**) に対処する

AnOnlineShop_5.json をインポートしてアクセスパターン 4 (getProductInventoryByProductId) に対処します。warehouseItem エンティティは、各倉庫の商品数を追跡するために使用されます。この項目は通常、倉庫に商品が追加されたり、倉庫から商品が削除されたりすると更新されます。ERD にあるように、product と warehouse の間には多対多リレーションシップがあります。ここで、product から warehouse への 1 対多リレーションシップは warehouseItem のようにモデル化されます。その後、warehouse から product への 1 対多リレーションシップもモデル化されます。

アクセスパターン 4 は、PK=ProductId および SK begins_with “w#“ に対するクエリで対処できます。

ソートキーに適用できる begins_with( ) およびその他の式の詳細については、「キー条件式」を参照してください。

ベーステーブル –

Primary key		Attributes		
Partition key: PK	Sort key: SK			
c#12345	c#12345	EntityType	Email	Name
		customer	samaneh@example.com	Samaneh Utter
	p#12345	EntityType	Detail	Price
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100
p#12345	w#12345	EntityType	Quantity	
		warehouseItem	50	
w#12345	w#12345	EntityType	Address	
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}	

ステップ 5: アクセスパターン 5 (**getOrderDetailsByOrderId**) と 6 (**getProductByOrderId**) に対処する

AnOnlineShop_6.json をインポートして、さらに customer、product、および warehouse の項目をテーブルに追加します。次に、AnOnlineShop_7.json をインポートして、アクセスパターン 5 (getOrderDetailsByOrderId) および 6 (getProductByOrderId) に対処できる order 用項目コレクションを構築します。orderItem エンティティとしてモデル化された order と product の間の 1 対多リレーションシップを確認できます。

アクセスパターン 5 (getOrderDetailsByOrderId) に対処するには、PK=orderId を使用してテーブルをクエリします。これにより、customerId および注文された製品を含む注文に関するすべての情報が提供されます。

ベーステーブル –

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

アクセスパターン 6 (getProductByOrderId) に対処するには、order 内の製品のみを読み取る必要があります。これを達成するために、PK=orderId および SK begins_with “p#” を使用してテーブルをクエリします。

ベーステーブル –

Primary key		Attributes		
Partition key: PK	Sort key: SK			
	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
o#12345	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

ステップ 6: アクセスパターン 7 (**getInvoiceByOrderId**) に対処する

AnOnlineShop_8.json をインポートして、アクセスパターン 7 (getInvoiceByOrderId) を処理する注文項目コレクションに invoice エンティティを追加します。このアクセスパターンに対処するために、PK=orderId および SK begins_with “i#” でクエリオペレーションを使用できます。

ベーステーブル –

Primary key		Attributes		
Partition key: PK	Sort key: SK			
o#12345	c#12345	EntityType	Date	
		order	2020-06-21T19:10:00	
	i#55443	EntityType	Amount	Date
		invoice	400	2020-06-21T19:18:00
	p#12345	EntityType	Price	Quantity
		orderItem	100	2
	p#99887	EntityType	Price	Quantity
		orderItem	40	5

ステップ 7: アクセスパターン 8 (**`getShipmentByOrderId`**) に対処する

AnOnlineShop_9.json をインポートして、アクセスパターン 8 (`getShipmentByOrderId`) を処理する注文項目コレクションに `shipment` エンティティを追加します。単一テーブルの設計にさらに多くの種類のエンティティを追加することで、同じ垂直分割モデルを拡張しています。注文項目コレクションに、`order` エンティティが `shipment`、`orderItem`、および `invoice` エンティティと持つさまざまな関係がどのように含まれているかに注意してください。

`orderId` による出荷を取得するには、`PK=orderId` と `SK begins_with “sh#”` でクエリオペレーションを実行します。

ベーステーブル –

Primary key		Attributes				
Partition key: PK	Sort key: SK					
o#12345	c#12345	EntityType	Date			
		order	2020-06-21T19:10:00			
	i#55443	EntityType	Amount	Date		
		invoice	400	2020-06-21T19:18:00		
	p#12345	EntityType	Price	Quantity		
		orderItem	100	2		
	p#99887	EntityType	Price	Quantity		
		orderItem	40	5		
	sh#88899	EntityType	Address	Type	Date	Warehouseld
		shipment	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00	w#12376
	sh#98765	EntityType	Address	Type	Date	Warehouseld
		shipment	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00	w#12345

ステップ 8: アクセスパターン 9 (**`getOrderByProductIdForDateRange`**) に対処する

前のステップで、注文項目コレクションを作成しました。このアクセスパターンには新しいルックアップディメンション (`ProductID` および `Date`) があり、テーブル全体をスキャンし、関連するレコードをフィルタリングして、ターゲット項目を取得する必要があります。このアクセスパターンに対処するには、グローバルセカンダリインデックス (GSI) を作成する必要があります。AnOnlineShop_10.json をインポートし、複数の注文項目コレクションから `orderItem` データを取得できる GSI を使用して、新しい項目コレクションを作成します。データには `GSI1-PK` と `GSI1-SK` が含まれ、それぞれ `GSI1` のパーティションキーとソートキーになります。

DynamoDB は、GSI のキー属性を含む項目をテーブルから GSI に自動的に入力します。GSI に追加の挿入を手動で行う必要はありません。

アクセスパターン 9 に対処するには、GSI1-PK=productId および GSI1SK between (date1, date2) を使用して GSI1 でクエリを実行します。

ベーステーブル –

Primary key		Attributes				
Partition key: PK	Sort key: SK					
o#12345	p#12345	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#12345	2020-06-21T19:18:00	100	2
	p#99887	EntityType	GSI1-PK	GSI1-SK	Price	Quantity
		orderItem	p#99887	2020-06-21T19:20:00	40	5

GSI1:

Primary key		Attributes				
Partition key: GSI1-PK	Sort key: GSI1-SK					
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#12345	orderItem	2	100
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	Quantity	Price
		o#12345	p#99887	orderItem	5	40

ステップ 9: アクセスパターン 10 (**getInvoiceByInvoiceId**) と 11 (**getPaymentByInvoiceId**) に対処する

AnOnlineShop_11.json をインポートして、アクセスパターン 10 (getInvoiceByInvoiceId) と 11 (getPaymentByInvoiceId) に対処します。どちらも invoice に関連しています。これらは 2 つの異なるアクセスパターンですが、同じキー条件を使用して実現されます。Payments は、invoice エンティティのマップデータタイプを持つ属性として定義されます。

> **Note**
>
> GSI1-PK と GSI1-SK は、さまざまなエンティティに関する情報を保存するためにオーバーロードされているため、同じ GSI から複数のアクセスパターンに対応できます。GSI オーバーロードの詳細については、「DynamoDB のグローバルセカンダリインデックスの多重定義」を参照してください。

アクセスパターン 10 と 11 に対処するには、`GSI1-PK=invoiceId` および `GSI1-SK=invoiceId` を使用して `GSI1` に対してクエリを実行します。

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
i#55443	i#55443	o#12345	i#55443	invoice	c#12345	i#2020-06-21T19:18:00	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00

ステップ 10: アクセスパターン 12 (**`getShipmentDetailsByShipmentId`**) と 13 (**`getShipmentByWarehouseId`**) に対処する

AnOnlineShop_12.json をインポートして、アクセスパターン 12 (`getShipmentDetailsByShipmentId`) と 13 (`getShipmentByWarehouseId`) に対処します。

1 回のクエリオペレーションで注文に関するすべての詳細を取得できるように、`shipmentItem` エンティティがベーステーブルの注文項目コレクションに追加されることに注意してください。

ベーステーブル –

Primary key		Attributes							
Partition key: PK	Sort key: SK								
o#12345	sh#88899	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#88899	sh#88899	w#12376	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
	sh#98765	EntityType	GSI1-PK	GSI1-SK	GSI2-PK	GSI2-SK	Address	Type	Date
		shipment	sh#98765	sh#98765	w#12345	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00
	shp#12345	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#99887	3				
	shp#54321	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#88899	p#99887	2				
	shp#55555	EntityType	GSI1-PK	GSI1-SK	Quantity				
		shipmentItem	sh#98765	p#12345	2				

GSI1 パーティションとソートキーは、shipment と shipmentItem の間の 1 対多リレーションシップをモデル化するために既に使用されています。アクセスパターン 12 (getShipmentDetailsByShipmentId) に対処するには、GSI1-PK=shipmentId および GSI1-SK=shipmentId を使用して GSI1 に対してクエリを実行します。

GSI1:

Primary key		Attributes							
Partition key: GSI1-PK	Sort key: GSI1-SK								
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
sh#88899	sh#88899	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#88899	shipment	w#12376	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
	p#12345	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
sh#98765	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00

アクセスパターン 13 (`getShipmentByWarehouseId`) の `warehouse` と `shipment` の間の新しい 1 対多リレーションシップをモデル化するには、別の GSI (`GSI2`) を作成する必要があります。アク

セスパターンに対処するには、GSI2-PK=warehouseId および GSI2-SK begins_with “sh#” を使用して GSI2 に対してクエリを実行します。

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
w#12376	sh#88899	o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T08:20:00
		PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
w#12345	sh#98765	o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country": {"S":"Sweden"},"County": {"S":"Vastra Gotaland"},"City": {"S":"Goteborg"},"Street": {"S":"Slanbarsvagen"},"Number": {"S":"34"},"ZipCode": {"S":"41787"}}	Express	2020-06-22T10:20:00

ステップ 11: アクセスパターン 14 (**getProductInventoryByWarehouseId**)、15 (**getInvoiceByCustomerIdForDateRange**)、および 16 (**getProductsByCustomerIdForDateRange**) に対処する

AnOnlineShop_13.json をインポートして、次のアクセスパターンセットに関連するデータを追加します。アクセスパターン 14 (getProductInventoryByWarehouseId) に対処するには、GSI2-PK=warehouseId および GSI2-SK begins_with “p#” を使用して GSI2 に対してクエリを実行します。

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

アクセスパターン 15 (`getInvoiceByCustomerIdForDateRange`) に対処するには、`GSI2-PK=customerId` および `GSI2-SK between (i#date1, i#date2)` を使用して GSI2 に対してクエリを実行します。

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

アクセスパターン 16 (`getProductsByCustomerIdForDateRange`) に対処するには、`GSI2-PK=customerId` および `GSI2-SK between (p#date1, p#date2)` を使用して GSI2 に対してクエリを実行します。

GSI2:

Primary key		Attributes							
Partition key: GSI2-PK	Sort key: GSI2-SK								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
c#12345	i#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	p#2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	p#2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	

Note

NoSQL Workbench では、ファセットは DynamoDB に対するアプリケーションのさまざまなデータアクセスパターンを表します。ファセットを使用すると、ファセットの制約を満たさないレコードを表示することなく、テーブル内のデータのサブセットを表示できます。プロビジョンドキャパシティの引き上げがファセットはビジュアルデータモデリングツールと見なされ、アクセスパターンのモデリングを純粋に補助するものであるため、DynamoDB で使用できるコンストラクトとしては存在しません。

AnOnlineShop_facets.json をインポートして、このユースケースのファセットを確認します。

すべてのアクセスパターンと各アクセスパターンにスキーマ設計で対処する方法を次の表にまとめています。

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値
getCustomerByCustomerId	ベーステーブル	GetItem	PK=customerId	SK=customerId
getProductByProductId	ベーステーブル	GetItem	PK=productId	SK=productId
getWarehouseByWarehouseId	ベーステーブル	GetItem	PK=warehouseId	SK=warehouseId
getProductInventoryByProductId	ベーステーブル	クエリ	PK=productId	SK begins_with "w#"
getOrderDetailsByOrderId	ベーステーブル	クエリ	PK=orderId	
getProductByOrderId	ベーステーブル	クエリ	PK=orderId	SK begins_with "p#"
getInvoiceByOrderId	ベーステーブル	クエリ	PK=orderId	SK begins_with "i#"
getShipmentByOrderId	ベーステーブル	クエリ	PK=orderId	SK begins_with "sh#"
getOrderByProductIdForDateRange	GSI1	クエリ	PK=productId	SK between date1 and date2

アクセスパターン	ベーステーブル/GSI/LSI	Operation	パーティションキー値	ソートキー値
getInvoiceByInvoiceId	GSI1	クエリ	PK=invoiceId	SK=invoiceId
getPaymentByInvoiceId	GSI1	クエリ	PK=invoiceId	SK=invoiceId
getShipmentDetailsByShipmentId	GSI1	クエリ	PK=shipmentId	SK=shipmentId
getShipmentByWarehouseId	GSI2	クエリ	PK=warehouseId	SK begins_with "sh#"
getProductInventoryByWarehouseId	GSI2	クエリ	PK=warehouseId	SK begins_with "p#"
getInvoiceByCustomerIdForDateRange	GSI2	クエリ	PK=customerId	SK between i#date1 and i#date2
getProductsByCustomerIdForDateRange	GSI2	クエリ	PK=customerId	SK between p#date1 and p#date2

オンラインショップの最終スキーマ

最終的なスキーマ設計は次のとおりです。このスキーマ設計を JSON ファイルとしてダウンロードするには、GitHub の「DynamoDB 設計パターン」を参照してください。

ベーステーブル –

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
c#12345	c#12345	EntityType	Email	Name	
		customer	samaneh@example.com	Samaneh	
c#23456	c#23456	EntityType	Email	Name	
		customer	kathleen@example.com	Kathleen	
c#54321	c#54321	EntityType	Email	Name	
		customer	henrik@example.com	Henrik	
p#12345	p#12345	EntityType	Detail	Price	
		product	{"Name":{"S":"Options Open"},"Description":{"S":"The latest album"}}	100	
	w#12345	EntityType	GSI2-PK	GSI2-SK	Quantity
		warehouseItem	w#12345	p#12345	50
p#99887	p#99887	EntityType	Detail	Price	
		product	{"Name":{"S":"The Book"},"Description":{"S":"The best book ever"}}	40	
	w#12345	EntityType	GSI2-PK	GSI2-SK	Quantity
		warehouseItem	w#12345	p#99887	4
	w#12376	EntityType	Quantity		
		warehouseItem	4		
w#12345	w#12345	EntityType	Address		
		warehouse	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"MainStreet"},"Number":{"S":"20"},"ZipCode":{"S":"41111"}}		
	w#12376	EntityType	Address		
		warehouse	{"Country":{"S":"Sweden"},"County":		

GSI1

Primary key		Attributes							
Partition key: GSI1-PK	**Sort key: GSI1-SK**								
p#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#12345	orderItem	c#12345	2020-06-21T19:18:00	100	2	
p#99887	2020-06-21T19:20:00	PK	SK	EntityType	GSI2-PK	GSI2-SK	Price	Quantity	
		o#12345	p#99887	orderItem	c#12345	2020-06-21T19:20:00	40	5	
i#55443	i#55443	PK	SK	EntityType	GSI2-PK	GSI2-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	c#12345	2020-06-21T19:18:00	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
sh#88899	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#54321	shipmentItem	2				
	sh#88899	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#88899	shipment	w#12376	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00
sh#98765	p#12345	PK	SK	EntityType	Quantity				
		o#12345	shp#55555	shipmentItem	2				
	p#99887	PK	SK	EntityType	Quantity				
		o#12345	shp#12345	shipmentItem	3				
	sh#98765	PK	SK	EntityType	GSI2-PK	GSI2-SK	Address	Type	Date
		o#12345	sh#98765	shipment	w#12345	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00

GSI2

Primary key		Attributes							
Partition key: GSI2-PK	**Sort key: GSI2-SK**								
w#12345	p#12345	PK	SK	EntityType	Quantity				
		p#12345	w#12345	warehouseItem	50				
	p#99887	PK	SK	EntityType	Quantity				
		p#99887	w#12345	warehouseItem	4				
	sh#98765	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#98765	shipment	sh#98765	sh#98765	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T10:20:00
c#12345	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#12345	orderItem	p#12345	2020-06-21T19:18:00	100	2	
	2020-06-21T19:18:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Detail	Amount	Date
		o#12345	i#55443	invoice	i#55443	i#55443	{"Payments":{"L":[{"M":{"Type":{"S":"GiftCard"},"Amount":{"N":"100"},"Data":{"S":"GiftCard data here..."}}},{"M":{"Type":{"S":"MasterCard"},"Amount":{"N":"300"},"Data":{"S":"Payment data here..."}}}]}}	400	2020-06-21T19:18:00
	2020-06-21T19:20:00	PK	SK	EntityType	GSI1-PK	GSI1-SK	Price	Quantity	
		o#12345	p#99887	orderItem	p#99887	2020-06-21T19:20:00	40	5	
w#12376	sh#88899	PK	SK	EntityType	GSI1-PK	GSI1-SK	Address	Type	Date
		o#12345	sh#88899	shipment	sh#88899	sh#88899	{"Country":{"S":"Sweden"},"County":{"S":"Vastra Gotaland"},"City":{"S":"Goteborg"},"Street":{"S":"Slanbarsvagen"},"Number":{"S":"34"},"ZipCode":{"S":"41787"}}	Express	2020-06-22T08:20:00

このスキーマ設計での NoSQL Workbench の使用

この最終スキーマを、DynamoDB のデータモデリング、データ視覚化、クエリ開発機能を提供するビジュアルツールである NoSQL Workbench にインポートして、新しいプロジェクトを詳しく調べたり編集したりできます。使用を開始するには、次の手順に従います。

1. NoSQL Workbench をダウンロードします。詳細については、「the section called “ダウンロード”」を参照してください。
2. 上記の JSON スキーマファイルをダウンロードします。このファイルは既に NoSQL Workbench モデル形式になっています。
3. JSON スキーマファイルを NoSQL Workbench にインポートします。詳細については、「the section called “既存のモデルのインポート”」を参照してください。
4. NOSQL Workbench にインポートしたら、データモデルを編集できます。詳細については、「the section called “既存モデルの編集”」を参照してください。

DynamoDB でリレーショナルデータをモデル化するためのベストプラクティス

このセクションでは、Amazon DynamoDB でリレーショナルデータをモデル化するためのベストプラクティスを説明します。まず、従来のデータモデリングのコンセプトを紹介します。次に、従来のリレーショナルデータベース管理システムではなく DynamoDB を使用する利点について説明します。つまり JOIN オペレーションが不要になり、オーバーヘッドが削減されるということです。

このセクションでは、効率的にスケールする DynamoDB テーブルを設計する方法と、DynamoDB でリレーショナルデータをモデル化する方法の例を説明します。

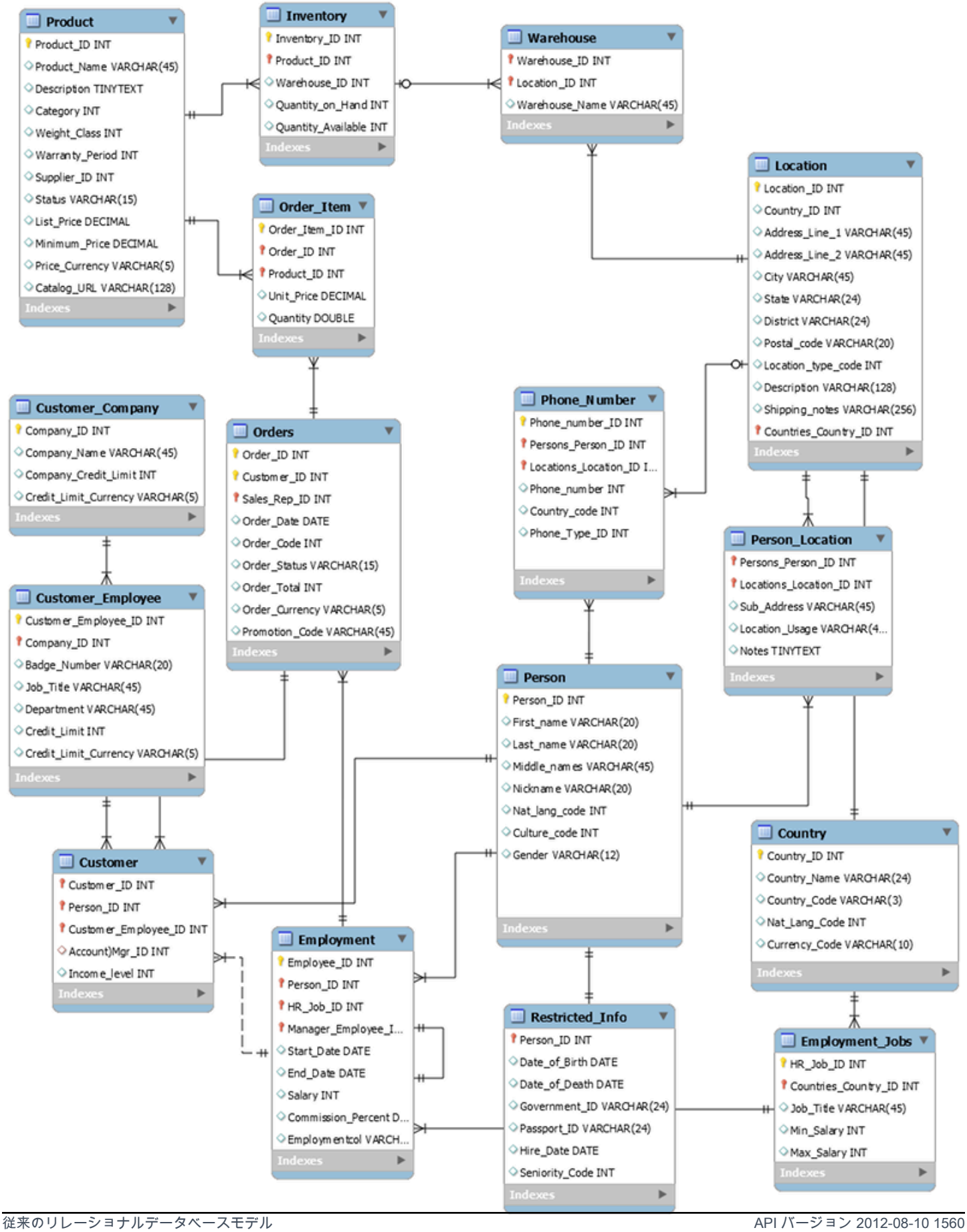
トピック

- 従来のリレーショナルデータベースモデル
- DynamoDB によって JOIN オペレーションが不要になる理由
- DynamoDB トランザクションが書き込みプロセスのオーバーヘッドを排除する方法
- DynamoDB でリレーショナルデータをモデル化するための最初のステップ
- DynamoDB でリレーショナルデータをモデル化する例

従来のリレーショナルデータベースモデル

従来のリレーショナルデータベース管理システム (RDBMS) は、正規化されたリレーショナル構造でデータを保存します。リレーショナルデータモデルの目的は、(正規化によって) データの重複を減らして参照整合性を維持し、データの異常を削減することです。

次のスキーマは、一般的な注文入力アプリケーションのリレーショナルデータ型の一例です。このアプリケーションは、理論上の製造元の運用およびビジネスサポートシステムを支える人事スキーマをサポートします。


Product
Product_ID INT
Product_Name VARCHAR(45)
Description TINYTEXT
Category INT
Weight_Class INT
Warranty_Period INT
Supplier_ID INT
Status VARCHAR(15)
List_Price DECIMAL
Minimum_Price DECIMAL
Price_Currency VARCHAR(5)
Catalog_URL VARCHAR(128)
Indexes
Inventory
Inventory_ID INT
Product_ID INT
Warehouse_ID INT
Quantity_on_Hand INT
Quantity_Available INT
Indexes
Warehouse
Warehouse_ID INT
Location_ID INT
Warehouse_Name VARCHAR(45)
Indexes
Location
Location_ID INT
Country_ID INT
Address_Line_1 VARCHAR(45)
Address_Line_2 VARCHAR(45)
City VARCHAR(45)
State VARCHAR(24)
District VARCHAR(24)
Postal_code VARCHAR(20)
Location_type_code INT
Description VARCHAR(128)
Shipping_notes VARCHAR(256)
Countries_Country_ID INT
Indexes
Order_Item
Order_Item_ID INT
Order_ID INT
Product_ID INT
Unit_Price DECIMAL
Quantity DOUBLE
Indexes
Customer_Company
Company_ID INT
Company_Name VARCHAR(45)
Company_Credit_Limit INT
Credit_Limit_Currency VARCHAR(5)
Indexes
Orders
Order_ID INT
Customer_ID INT
Sales_Rep_ID INT
Order_Date DATE
Order_Code INT
Order_Status VARCHAR(15)
Order_Total INT
Order_Currency VARCHAR(5)
Promotion_Code VARCHAR(45)
Indexes
Phone_Number
Phone_number_ID INT
Persons_Person_ID INT
Locations_Location_ID I...
Phone_number INT
Country_code INT
Phone_Type_ID INT
Indexes
Person_Location
Persons_Person_ID INT
Locations_Location_ID INT
Sub_Address VARCHAR(45)
Location_Usage VARCHAR(4...
Notes TINYTEXT
Indexes
Customer_Employee
Customer_Employee_ID INT
Company_ID INT
Badge_Number VARCHAR(20)
Job_Title VARCHAR(45)
Department VARCHAR(45)
Credit_Limit INT
Credit_Limit_Currency VARCHAR(5)
Indexes
Person
Person_ID INT
First_name VARCHAR(20)
Last_name VARCHAR(20)
Middle_names VARCHAR(45)
Nickname VARCHAR(20)
Nat_lang_code INT
Culture_code INT
Gender VARCHAR(12)
Indexes
Country
Country_ID INT
Country_Name VARCHAR(24)
Country_Code VARCHAR(3)
Nat_Lang_Code INT
Currency_Code VARCHAR(10)
Indexes
Customer
Customer_ID INT
Person_ID INT
Customer_Employee_ID INT
Account)Mgr_ID INT
Income_level INT
Indexes
Employment
Employee_ID INT
Person_ID INT
HR_Job_ID INT
Manager_Employee_I...
Start_Date DATE
End_Date DATE
Salary INT
Commission_Percent D...
Employmentcol VARCH...
Indexes
Restricted_Info
Person_ID INT
Date_of_Birth DATE
Date_of_Death DATE
Government_ID VARCHAR(24)
Passport_ID VARCHAR(24)
Hire_Date DATE
Seniority_Code INT
Indexes
Employment_Jobs
HR_Job_ID INT
Countries_Country_ID INT
Job_Title VARCHAR(45)
Min_Salary INT
Max_Salary INT
Indexes

非リレーショナルデータベースサービスである DynamoDB には、従来のリレーショナルデータベース管理システムと比較して多くの利点があります。

DynamoDB によって JOIN オペレーションが不要になる理由

RDBMS は、構造クエリ言語 (SQL) を使用してデータをアプリケーションに返します。データ型の正規化により、このようなクエリでは通常、JOIN 演算子を使用して 1 つ以上のテーブルのデータを結合する必要があります。

例えば、各項目を出荷するすべてのウェアハウスで在庫数量でソートされた発注書項目のリストを生成するには、前のスキーマに対して次の SQL クエリを発行します。

```
SELECT * FROM Orders
  INNER JOIN Order_Items ON Orders.Order_ID = Order_Items.Order_ID
  INNER JOIN Products ON Products.Product_ID = Order_Items.Product_ID
  INNER JOIN Inventories ON Products.Product_ID = Inventories.Product_ID
  ORDER BY Quantity_on_Hand DESC
```

この種の SQL クエリは、データにアクセスするための柔軟な API を提供できますが、大量の処理が必要です。クエリを結合するたびに、クエリのランタイムの複雑さが増します。これは、各テーブルのデータをステージングしてからアセンブルして結果セットを返す必要があるためです。

クエリの実行時間に影響するその他の要因は、テーブルのサイズおよび結合される列にインデックスがあるかどうかです。前述のクエリは、複数のテーブルにわたって複雑なクエリを実行し、結果として生成されるセットをソートします。

JOINs の必要性をなくすことが NoSQL データモデリングの中核となります。これが、Amazon.com をサポートするために Amazon が DynamoDB を構築した理由であり、DynamoDB があらゆる規模で一貫したパフォーマンスを提供できる理由です。SQL クエリと JOINs のランタイムの複雑さを考慮すると、RDBMS のパフォーマンスは規模に応じて一定ではありません。そのため、お客様のアプリケーションが大きくなるにつれてパフォーマンスの問題が発生します。

データを正規化することでディスクに保存されるデータの量は削減されますが、多くの場合、パフォーマンスに影響する最も制約の厳しいリソースは CPU 時間とネットワーク遅延です。

DynamoDB は、JOINs を削除し(かつデータの非正規化を促進し)、データベースアーキテクチャを最適化して、項目に 1 回のリクエストでアプリケーションクエリに完全に応答します。これによって、両方の制約を最小限に抑えます。これらの特性により、DynamoDB は規模にかかわらず 1 桁のミリ秒単位のパフォーマンスを実現できます。これは、一般的なアクセスパターンでは、DynamoDB オペレーションの実行時の複雑さが一定であるためです。

DynamoDB トランザクションが書き込みプロセスのオーバーヘッドを排除する方法

RDBMS の速度を低下させるもう 1 つの要因は、正規化されたスキーマに書き込むためにトランザクションを使用することです。例に示す通り、ほとんどのオンライントランザクション処理 (OLTP) アプリケーションで使用されるリレーショナルデータ構造は、RDBMS に格納されているときに複数の論理テーブルに分割して分散させる必要があります。

そのため、アプリケーションで書き込み中のオブジェクトを読み込もうとした場合に発生する可能性のある競合状態およびデータの整合性の問題を回避するためには、ACID 準拠のトランザクションフレームワークが必要です。こうしたトランザクションフレームワークをリレーショナルスキーマと組み合わせると、書き込みプロセスにかなりのオーバーヘッドが発生する可能性があります。

DynamoDB にトランザクションを実装すると、RDBMS で一般的に見られるスケーリングの問題が防止されます。DynamoDB では、トランザクションを 1 回の API 呼び出しとして発行し、その 1 回のトランザクションでアクセスできる項目の数を制限することで、これを行っています。トランザクションの実行時間が長いと、トランザクションがクローズされないため、データのロックが長期間、または永続的に保持され、運用上の問題が発生する可能性があります。

DynamoDB でのこのような問題を防ぐため、トランザクションは `TransactWriteItems` と `TransactGetItems` の 2 つの異なる API オペレーションを使用して実装されました。これらの API オペレーションには、RDBMS で一般的な開始および終了セマンティクスがありません。さらに、DynamoDB では、同様に長時間実行されるトランザクションを防ぐため、1 回のトランザクション内で 100 項目のアクセス制限を設けています。DynamoDB トランザクションの詳細については、「トランザクションでの使用」を参照してください。

こうした理由から、高トラフィックのクエリに対して低レイテンシーの応答が必要な場合は、NoSQL システムを利用すると、一般的に技術的および経済的な効果がもたらされます。Amazon DynamoDB では、リレーショナルシステムのスケーラビリティを制限する問題を回避できます。

RDBMS のパフォーマンスは、通常、以下の理由から適切にスケーリングできません。

- 高価な結合を使用して、クエリ結果の必要なビューを再構成します。
- データを正規化し、複数のクエリを必要とする複数のテーブルに格納して、ディスクに書き込みます。
- 一般的に、ACID 準拠のトランザクションシステムのパフォーマンスコストが発生します。

DynamoDB は、次の理由により適切に拡張されます。

- スキーマの柔軟性により、DynamoDB は複雑な階層データを 1 つの項目内に格納できます。
- 複合キー設計では、関連する項目を同じテーブルの範囲内に格納できます。
- トランザクションは 1 回のオペレーションで実行されます。オペレーションの実行が長時間に及ぶのを避けるため、アクセスできる項目の上限数は 100 となっています。

データストアに対するクエリは、多くの場合次のような形式で、非常に簡単になります。

```
SELECT * FROM Table_X WHERE Attribute_Y = "somevalue"
```

DynamoDB は、以前の例の RDBMS と比較して、リクエストされたデータを返す作業がはるかに少なくなっています。

DynamoDB でリレーショナルデータをモデル化するための最初のステップ

> **Note**
>
> NoSQL 設計では、RDBMS 設計とは異なる考え方が必要です。RDBMS の場合は、アクセスパターンを考慮せずに正規化されたデータモデルを作成できます。その後、新しい課題とクエリの要件が発生したら、そのデータモデルを拡張することができます。対照的に、Amazon DynamoDB の場合は答えが必要な質問が分かるまで、スキーマの設計を開始しないでください。ビジネス上の問題とアプリケーションのユースケースを理解することが極めて重要です。

効率的に拡張する DynamoDB テーブルの設計を開始するには、サポートする必要のあるオペレーションおよびビジネスサポートシステム (OSS/BSS) で必要とされるアクセスパターンを特定するために、まずいくつかの手順を実行する必要があります。

- 新しいアプリケーションの場合は、アクティビティや目的に関するユーザーストーリーを確認します。特定するさまざまなユースケースを文書化し、必要なアクセスパターンを分析します。
- 既存のアプリケーションでは、クエリログを分析して、ユーザーが現在どのようにシステムを使用しているか、主要なアクセスパターンを調べます。

このプロセスが完了したら、次のようなリストが表示されます。

注文入力アプリケーションのアクセスパターン

パターン #	アクセスパターンの説明
1	従業員 ID で従業員の詳細を検索する
2	従業員名で従業員の詳細をクエリする
3	従業員の電話番号を検索する (複数可)
4	顧客の電話番号を検索する (複数可)
5	日付範囲内の顧客の注文を取得する
6	日付範囲内のすべての OPEN の注文を表示する
7	最近雇用したすべての従業員を確認する
8	倉庫のすべての従業員を検索する
9	製品の注文中のすべての項目を取得する
10	すべての倉庫の製品在庫を取得する
11	アカウント担当者別に顧客を取得する
12	アカウント担当者別に注文を取得する
13	役職を持つ従業員を取得する
14	商品および倉庫別に在庫を取得する
15	商品在庫総数を取得する

実際のアプリケーションでは、リストはさらに長くなる場合があります。しかし、このコレクションは、本番環境で見つかる可能性のあるクエリパターンの複雑さの範囲を表します。

DynamoDB スキーマ設計の最新のアプローチでは、集約指向の原則を使用し、厳格なエンティティの境界ではなくアクセスパターンに基づいてデータをグループ化します。このアプローチでは、複数の設計パターンを考慮します。

- 単一テーブル設計 - 複合ソートキー、オーバーロードされたグローバルセカンダリインデックス、隣接リストパターンを使用して、複数のエンティティタイプを 1 つのテーブルに保存します
- マルチテーブル設計 - 独立した運用特性を持ち、アクセス相関が低いエンティティに個別のテーブルを使用し、エンティティ間のクエリには戦略的な GSI を使用します
- 集約設計 - 常に一緒にアクセスされる場合に関連するデータを埋め込む (Order + OrderItems)、または関係を識別するために項目コレクションを使用します (Product + Inventory)

これらのアプローチの選択は、特定のアクセスパターン、データ特性、運用要件によって異なります。これらの要素を使用してデータを構造化することで、アプリケーションは、テーブルまたはインデックスの単一のクエリを使用して、特定のアクセスパターンに必要なものを取得できます。

> **Note**
>
> 単一テーブル設計とマルチテーブル設計の選択は、特定の要件によって異なります。単一テーブル設計は、エンティティのアクセス相関性が高く、運用特性が類似している場合に適しています。エンティティに独立した運用要件や異なるアクセスパターンがある場合、または明確な運用上の境界が必要な場合は、マルチテーブル設計が推奨されます。このガイドの例では、戦略的集約と非正規化によるマルチテーブルアプローチを示しています。

NoSQL Workbench for DynamoDB を使用してパーティションキー設計を視覚化する方法については、「NoSQL Workbench を使用したデータモデルの構築」を参照してください。

DynamoDB でリレーショナルデータをモデル化する例

この例では、Amazon DynamoDB でリレーショナルデータをモデル化する方法について説明します。DynamoDB テーブルの設計は、「リレーショナルモデル化」に示されているリレーショナルオーダーのエントリスキーマに対応しています。この設計では、単一の隣接リストではなく複数の特殊なテーブルを使用し、戦略的 GSI を活用してすべてのアクセスパターンを効率的に処理しながら、明確な運用上の境界を提供します。

設計アプローチでは、集約指向の原則を使用し、厳格なエンティティ境界ではなくアクセスパターンに基づいてデータをグループ化します。主な設計上の決定事項には、アクセス相関が低いエンティティに個別のテーブルを使用すること、常に一緒にアクセスされる場合に関連データを埋め込むこと、関係を特定するための項目コレクションを使用することが含まれます。

次のテーブルとそれに付随するインデックスは、リレーショナルオーダーのエントリスキーマをサポートしています。

従業員テーブルの設計

従業員テーブルには、従業員情報が項目ごとに 1 つのエンティティとして保存され、従業員を直接検索できるように最適化し、戦略的 GSI を通じて複数のクエリパターンがサポートされます。このテーブルは、独立した運用特性を持ち、エンティティ間のアクセス相関が低いエンティティに対して個別のテーブルを設計する原則を示しています。

このテーブルでは、各従業員が個別のエンティティであるため、ソートキーなしでシンプルなパーティションキー (employee_id) を使用します。4 つの GSI により、さまざまな属性による効率的なクエリが可能になります。

- EmployeeByName GSI - すべての従業員属性で INCLUDE 射影を使用して、従業員の詳細を名前で完全に取得し、employee_id をソートキーとして重複する可能性のある名前を処理します。
- EmployeeByWarehouse GSI - 必須属性 (名前、job_title、hire_date) のみを含む INCLUDE 射影を使用して、倉庫ベースのクエリをサポートしながらストレージコストを最小限に抑えます
- EmployeeByJobTitle GSI - レポートと組織分析のための INCLUDE 射影を使用してロールベースのクエリを有効にします
- EmployeeByHireDate GSI - hire_date をソートキーとする静的パーティションキー値「EMPLOYEE」を使用して、最近の採用者に対して効率的な日付範囲クエリを有効にします。従業員の追加/更新は通常 1,000 WCU 未満であるため、ホットパーティションの問題が発生することなく、単一のパーティションで書き込み負荷を処理できます。

従業員テーブル - ベーステーブル構造

employee_id (PK)	名前	phone_number	Warehouse_id	job_title	hire_date	entity_type
emp_001	John Smith	["+1-555-0101"]	wh_sea	マネージャー	2024-03-15	EMPLOYEES
emp_002	Jane Doe	["+1-555-0102", "+1-555-0103"]	wh_sea	アソシエイト	2025-01-10	EMPLOYEES
emp_003	Bob Wilson	["+1-555-0104"]	wh_pdx	アソシエイト	2025-06-20	EMPLOYEES

employee_id (PK)	名前	phone_number	Warehouse_id	job_title	hire_date	entity_type
emp_004	Alice Brown	["+1-555-0105"]	wh_pdx	スーパーバイザー	2023-11-05	EMPLOYEES
emp_005	Charlie Davis	["+1-555-0106"]	wh_sea	アソシエイト	2025-12-01	EMPLOYEES

EmployeeByName GSI - 従業員名クエリのサポート

名前 (GSI-PK)	employee_id (GSI-SK)	phone_number	Warehouse_id	job_title	hire_date
Alice Brown	emp_004	["+1-555-0105"]	wh_pdx	スーパーバイザー	2023-11-05
Bob Wilson	emp_003	["+1-555-0104"]	wh_pdx	アソシエイト	2025-06-20
Charlie Davis	emp_005	["+1-555-0106"]	wh_sea	アソシエイト	2025-12-01
Jane Doe	emp_002	["+1-555-0102", "+1-555-0103"]	wh_sea	アソシエイト	2025-01-10
John Smith	emp_001	["+1-555-0101"]	wh_sea	マネージャー	2024-03-15

EmployeeByWarehouse GSI - 倉庫クエリのサポート

Warehouse_id (GSI-PK)	employee_id (GSI-SK)	名前	job_title	hire_date
wh_pdx	emp_003	Bob Wilson	アソシエイト	2025-06-20

Warehouse_id (GSI-PK)	employee_id (GSI-SK)	名前	job_title	hire_date
wh_pdx	emp_004	Alice Brown	スーパーバイザー	2023-11-05
wh_sea	emp_001	John Smith	マネージャー	2024-03-15
wh_sea	emp_002	Jane Doe	アソシエイト	2025-01-10
wh_sea	emp_005	Charlie Davis	アソシエイト	2025-12-01

EmployeeByJobTitle GSI - 役職クエリのサポート

job_title (GSI-PK)	employee_id (GSI-SK)	名前	Warehouse_id	hire_date
アソシエイト	emp_002	Jane Doe	wh_sea	2025-01-10
アソシエイト	emp_003	Bob Wilson	wh_pdx	2025-06-20
アソシエイト	emp_005	Charlie Davis	wh_sea	2025-12-01
マネージャー	emp_001	John Smith	wh_sea	2024-03-15
スーパーバイザー	emp_004	Alice Brown	wh_pdx	2023-11-05

EmployeeByHireDate GSI - 最近の採用者クエリのサポート

entity_type (GSI-PK)	hire_date (GSI-SK)	employee_id	名前	Warehouse_id
EMPLOYEES	2023-11-05	emp_004	Alice Brown	wh_pdx
EMPLOYEES	2024-03-15	emp_001	John Smith	wh_sea
EMPLOYEES	2025-01-10	emp_002	Jane Doe	wh_sea

entity_type (GSI-PK)	hire_date (GSI-SK)	employee_id	名前	Warehouse_id
EMPLOYEES	2025-06-20	emp_003	Bob Wilson	wh_pdx
EMPLOYEES	2025-12-01	emp_005	Charlie Davis	wh_sea

顧客テーブルの設計

顧客テーブルは、Account_rep_id を戦略的に非正規化して顧客情報を維持し、アカウント担当者のクエリを効率的に実行できるようにします。この設計の選択により、わずかなストレージオーバーヘッドと引き換えにクエリパフォーマンスが向上し、顧客データとアカウント担当者データの結合が不要になります。

このテーブルは、リスト属性を使用して顧客ごとに複数の電話番号をサポートしており、DynamoDB のスキーマの柔軟性を示しています。単一の GSI により、アカウント担当者のワークフローが有効になります。

- CustomerByAccountRep GSI - 名前と E メール属性を含む INCLUDE 射影を使用して、完全な顧客レコードの取得を必要とせずに、アカウント担当者の顧客管理をサポートします。

顧客テーブル - ベーステーブル構造

customer_id (PK)	名前	phone_number	E メール	account_rep_id
cust_001	Acme Corp	["+1-555-1001"]	contact@acme.com	rep_001
cust_002	TechStart Inc	["+1-555-1002", "+1-555-1003"]	info@techstart.com	rep_001
cust_003	Global Traders	["+1-555-1004"]	sales@globaltraders.com	rep_002
cust_004	BuildRight LLC	["+1-555-1005"]	orders@buildright.com	rep_002

customer_id (PK)	名前	phone_number	E メール	account_rep_id
cust_005	FastShip Co	["+1-555-1006"]	support@fastship.com	rep_003

CustomerByAccountRep GSI - アカウント担当者クエリのサポート

account_rep_id (GSI-PK)	customer_id (GSI-SK)	名前	E メール
rep_001	cust_001	Acme Corp	contact@acme.com
rep_001	cust_002	TechStart Inc	info@techstart.com
rep_002	cust_003	Global Traders	sales@globaltraders.com
rep_002	cust_004	BuildRight LLC	orders@buildright.com
rep_003	cust_005	FastShip Co	support@fastship.com

注文テーブルの設計

注文テーブルは、注文ヘッダーと注文項目に個別の項目を含む垂直パーティショニングを使用します。この設計により、すべての注文コンポーネントを同じパーティション内に維持しながら、効率的な製品ベースのクエリが可能になり、効率的にアクセスできるようになります。各注文は複数の項目で構成されます。

- 注文ヘッダー - PK=order_id、SK=order_id の注文メタデータが含まれます
- 注文項目 - PK=order_id、SK=product_id を持つ個々の明細項目。直接の製品クエリを有効にします

> **Note**
>
> この垂直パーティショニングアプローチは、埋め込み注文項目のシンプルさと引き換えに、クエリの柔軟性を高めます。各注文項目は個別の DynamoDB 項目になり、すべての注文データを同じパーティション内に維持しながら効率的な製品ベースのクエリを実行し、単一のリクエストで効率的に取得できるようになります。

このテーブルには、顧客検索を必要とせずに直接アカウント担当者クエリを有効にするための account_rep_id (顧客テーブルから複製) の戦略的な非正規化が含まれています。高スループットの書き込みシナリオの場合、OPEN の注文には、複数のパーティションにわたる書き込みシャーディングを有効にするためのステータスとシャード属性が含まれます。

4 つの GSI は、最適化された射影を使用してさまざまなクエリパターンをサポートします。

- OrderByCustomerDate GSI - 注文の概要と項目の詳細を含む INCLUDE 射影を使用して、日付範囲フィルタリングによる顧客の注文履歴をサポートします
- OpenOrdersByDate GSI (スパース、シャード) - 5 つのシャードを持つマルチ属性パーティションキー (ステータス + シャード) を使用して、5,000 WPS (1 秒あたりの書き込み) をパーティション全体に分散します (各パーティションに 1,000 WPS、DynamoDB の パーティション制限 1,000 WCU に一致)。OPEN の注文のみをインデックスします (全体の 20%)。これにより、GSI ストレージコストを削減できます。クライアント側で結果をマージし、5 つのシャードすべてにわたって並列クエリを実行する必要があります
- OrderByAccountRep GSI - 完全な注文詳細がないアカウント担当者のワークフローをサポートするために、注文概要属性を含む INCLUDE 射影を使用します
- ProductInOrders GSI - OrderItem レコード (PK=order_id、SK=product_id) から作成されたこの GSI により、特定の製品を含むすべての注文を検索するクエリが可能になります。製品需要分析のために、注文コンテキスト (customer_id、order_date、数量) を含む INCLUDE 射影を使用します

注文テーブル - ベーステーブル構造 (垂直パーティショニング)

PK	SK	customer_id	order_date	ステータス	account_rep_id	quantity	料金	シャード
ord_001	ord_001	cust_001	2025-11-15	クローズ	rep_001			

PK	SK	customer_id	order_date	ステータス	account_rep_id	quantity	料金	シャード
ord_001	prod_100					5	25.00	
ord_002	ord_002	cust_001	2025-12-20	OPEN	rep_001			0
ord_002	prod_101					10	15.00	
ord_003	ord_003	cust_002	2026-01-05	OPEN	rep_001			2
ord_003	prod_100					3	25.00	

OrderByCustomerDate GSI - 顧客注文クエリのサポート

customer_id (GSI-PK)	order_date (GSI-SK)	order_id	ステータス	total_amount	order_items	シャード
cust_001	2025-11-15	ord_001	クローズ	225.00	[{product_id: "prod_100", qty: 5}]	
cust_001	2025-12-20	ord_002	OPEN	150.00	[{product_id: "prod_101", qty: 10}]	0
cust_002	2026-01-05	ord_003	OPEN	175.00	[{product_id: "prod_100", qty: 3}]	2
cust_003	2025-10-10	ord_004	クローズ	250.00	[{product_id: "prod_101", qty: 5}]	
cust_004	2026-01-03	ord_005	OPEN	200.00	[{product_id: "prod_100", qty: 20}]	1

OpenOrdersByDate GSI (スパース、シャード) - 高スループットの Open 注文クエリのサポート

ステータス (GSI-PK-1)	シャード (GSI-PK-2)	order_date (SK)	order_id	customer_id	account_rep_id	order_items	total_amount
OPEN	0	2025-12-20	ord_002	cust_001	rep_001	[{product_id: "prod_101", qty: 10}]	150.00
OPEN	1	2026-01-03	ord_005	cust_004	rep_002	[{product_id: "prod_100", qty: 20}]	200.00
OPEN	2	2026-01-05	ord_003	cust_002	rep_001	[{product_id: "prod_100", qty: 3}]	175.00

OrderByAccountRep GSI - アカウント担当者の注文クエリのサポート

account_rep_id (GSI-PK)	order_date (GSI-SK)	order_id	customer_id	ステータス	total_amount
rep_001	2025-11-15	ord_001	cust_001	クローズ	225.00
rep_001	2025-12-20	ord_002	cust_001	OPEN	150.00
rep_001	2026-01-05	ord_003	cust_002	OPEN	175.00
rep_002	2025-10-10	ord_004	cust_003	クローズ	250.00
rep_002	2026-01-03	ord_005	cust_004	OPEN	200.00

ProductInOrders GSI - 製品注文クエリのサポート

product_id (GSI-PK)	order_id (GSI-SK)	customer_id	order_date	quantity
prod_100	ord_001	cust_001	2025-11-15	5
prod_100	ord_003	cust_002	2026-01-05	3
prod_101	ord_002	cust_001	2025-12-20	10

製品テーブルの設計

製品テーブルは、項目コレクションパターンを使用して、製品メタデータと在庫データの両方を同じパーティション内に保存します。この設計では、製品と在庫間の識別関係を活用します。親製品がないと在庫は存在できません。製品メタデータに SK=product_id を、在庫項目に SK=warehouse_id を使用し、PK=product_id を設定すると、個別の在庫テーブルと GSI が不要になり、コストが約 50% 削減されます。

このパターンにより、個々の倉庫在庫 (複合キーを持つ GetItem) と製品のすべての倉庫在庫 (パーティションキーに対するクエリ) の両方に対する効率的なクエリが可能になります。製品メタデータ項目の total_inventory 属性は、合計在庫数をすばやく検索するための非正規化集計を提供します。

製品テーブル - ベーステーブル構造 (項目コレクションパターン)

product_id (PK)	Warehouse_id (SK)	product_name	category	unit_price	inventory_quantity	total_inventory
prod_100	prod_100	ウィジェット A	ハードウェア	25.00		500
prod_100	wh_sea				200	
prod_100	wh_pdx				150	
prod_100	wh_atl				150	
prod_101	prod_101	ガジェット B	電子機器	50.00		300

product_id (PK)	Warehouse_id (SK)	product_name	category	unit_price	inventory_quantity	total_inventory
prod_101	wh_sea				100	
prod_101	wh_pdx				200	

各テーブルは、必要なアクセスパターンを効率的にサポートするように、特定のグローバルセカンダリインデックス (GSI) で設計されています。この設計では、戦略的な非正規化とスパースインデックス作成を備えた集約指向の原則を使用して、パフォーマンスとコストの両方を最適化します。

主要な設計の最適化には以下が含まれます。

- スパース GSI - OpenOrdersByDate は OPEN の注文 (全体の 20%) のみをインデックス化するため、GSI ストレージコストを削減できます。
- 項目コレクションパターン - 製品テーブルは PK=product_id、SK=warehouse_id を使用して在庫を保存し、個別の在庫テーブルを排除します。
- Order + OrderItems 集約 - アクセス相関が 100% であるため、単一の項目として埋め込まれます
- 戦略的な非正規化 - 効率的なクエリのために注文テーブルで account_rep_id を重複させる

最後に、前に定義したアクセスパターンを再度使用することができます。次のテーブルは、戦略的 GSIs を備えたマルチテーブル設計を使用して、各アクセスパターンを効率的にサポートする方法を示しています。各パターンは、直接キー検索または単一の GSI クエリのいずれかを使用するため、コストのかかるスキャンを回避し、あらゆる規模で一貫したパフォーマンスを提供します。

S 番号	アクセスパターン	クエリの状態
1	従業員 ID で従業員の詳細を検索する	従業員テーブル: GetItem(employee_id="emp_001")
2	従業員名で従業員の詳細をクエリする	EmployeeByName GSI: Query(name="John Smith")
3	従業員の電話番号を検索する (複数可)	従業員テーブル: GetItem(employee_id="emp_001")

S 番号	アクセスパターン	クエリの状態
4	顧客の電話番号を検索する (複数可)	顧客テーブル: GetItem(customer_id="cust_001")
5	日付範囲内の顧客の注文を取得する	OrderByCustomerDate GSI: Query(customer_id="cust_001", order_date BETWEEN "2025-01-01" AND "2025-12-31")
6	日付範囲内のすべての OPEN の注文を表示する	OpenOrdersByDate GSI: マルチ属性 PK (status="OPEN + shard=0-4) を使用して 5 つのシャードを並列にクエリし、SK=order_date BETWEEN "2025-01-01" AND "2025-12-31"、結果をマージします
7	最近雇用したすべての従業員を確認する	EmployeeByHireDate GSI: Query(entity_type="EMPLOYEE", hire_date >= "2025-01-01")
8	倉庫のすべての従業員を検索する	EmployeeByWarehouse GSI: Query(warehouse_id="wh_sea")
9	製品の注文中のすべての項目を取得する	ProductInOrders GSI: Query(product_id="prod_100")
10	すべての倉庫の製品在庫を取得する	製品テーブル: Query(product_id="prod_100")
11	アカウント担当者別に顧客を取得する	CustomerByAccountRep GSI: Query(account_rep_id="rep_001")

S 番号	アクセスパターン	クエリの状態
12	アカウント担当者別に注文を取得する	OrderByAccountRep GSI: Query(account_rep_id="rep_001")
13	役職を持つ従業員を取得する	EmployeeByJobTitle GSI: Query(job_title="Manager")
14	商品および倉庫別に在庫を取得する	製品テーブル: GetItem(product_id="prod_100", warehouse_id="wh_sea")
15	商品在庫総数を取得する	製品テーブル: GetItem(product_id="prod_100", warehouse_id="prod_100")

リレーショナルデータベースから DynamoDB への移行

リレーショナルデータベースを DynamoDB に移行する際には、確実に成功させるための綿密な計画が必要です。このガイドでは、このプロセスの仕組み、利用できるツール、考えられる移行戦略を評価して要件に合うものを選択する方法を説明します。

トピック

- DynamoDB に移行する理由
- リレーショナルデータベースを DynamoDB に移行する際の考慮事項
- DynamoDB への移行の仕組みを理解する
- DynamoDB への移行に役立つツール
- DynamoDB に移行するための適切な戦略の選択
- DynamoDB へのオフライン移行を実行する
- DynamoDB へのハイブリッド移行を実行する
- 各テーブルを 1 対 1 で移行して DynamoDB へのオンライン移行を実行する
- カスタムステージングテーブルを使用して DynamoDB へのオンライン移行を実行する

DynamoDB に移行する理由

Amazon DynamoDB への移行は、企業や組織に幅広い魅力的なメリットをもたらします。データベースの移行先に DynamoDB を選択することで得られる主なメリットは次のとおりです。

- スケーラビリティ: DynamoDB は、大量のワークロードを処理し、データ量とトラフィックの増加に合わせてシームレスにスケールできるように設計されています。DynamoDB を使用すると、需要に応じてデータベースを簡単にスケールアップまたはスケールダウンできるため、アプリケーションのパフォーマンスを損なうことなくトラフィックの急増に対応できます。
- パフォーマンス: DynamoDB では低レイテンシーでのデータアクセスが可能であるため、並外れた速度でデータをアプリケーションに取得して処理できます。分散型アーキテクチャにより、読み取りと書き込みの操作が複数のノードに分散され、高いリクエストレートでも 1 桁ミリ秒単位の応答時間を一貫して実現できます。
- フルマネージド: DynamoDB は、AWS が提供するフルマネージドサービスです。つまり、プロビジョニング、設定、パッチ適用、バックアップ、スケーリングなど、データベース管理の運用面は AWS が処理するということです。これにより、企業はデータベースの管理タスクよりもアプリケーションの開発に集中できます。

- サーバーレスアーキテクチャ: DynamoDB は、DynamoDB オンデマンドと呼ばれるサーバーレスモデルをサポートしています。このモデルでは、事前にキャパシティをプロビジョニングすることなく、アプリケーションによって行われる実際の読み取りおよび書き込みリクエストに対してのみ料金が発生します。このリクエストベースの課金モデルでは、使用したリソースに対してのみ支払いを行うことになり、キャパシティのプロビジョニングや監視の必要がないため、コスト効率が向上し、運用上のオーバーヘッドも最小限に抑えられます。
- NoSQL の柔軟性: 従来のリレーショナルデータベースとは異なり、DynamoDB は NoSQL データモデルを採用しているため、スキーマ設計に柔軟性があります。DynamoDB では、構造化データ、半構造化データ、非構造化データを保存でき、変化を続ける多様なデータ型の処理に最適です。この柔軟性により、開発サイクルが短縮され、変化するビジネス要件に容易に適応できます。
- 優れた可用性と耐久性: DynamoDB では、リージョン内の複数のアベイラビリティーゾーンにデータをレプリケーションして、優れた可用性とデータ耐久性を確保します。レプリケーション、フェイルオーバー、リカバリが自動的に処理されるため、データ損失やサービス中断のリスクが最小限に抑えられます。DynamoDB では、最大 99.999% の可用性 SLA が提供されています。
- セキュリティとコンプライアンス: DynamoDB と AWS Identity and Access Management の統合によって、きめ細かいアクセスコントロールが実現します。保管中も転送中もデータを暗号化してセキュリティを確保します。また、DynamoDB は HIPAA、PCI DSS、GDPR などのさまざまなコンプライアンス標準に準拠しているため、規制要件も満たすことができます。
- AWS エコシステムとの統合: DynamoDB は AWS エコシステムの一部として、AWS Lambda、CloudFormation、AWS AppSync などの他の AWS サービスとシームレスに統合します。この統合により、サーバーレスアーキテクチャの構築、Infrastructure as Code の活用、リアルタイムのデータ主導型アプリケーションの作成が可能になります。

リレーショナルデータベースを DynamoDB に移行する際の考慮事項

リレーショナルデータベースシステムと NoSQL データベースにはそれぞれ異なる長所と短所があります。これらの相違点により、2 つのシステム間でデータベース設計が異なるものになります。

タスクの種類	リレーショナルデータベース	NoSQL データベース
データベースのクエリ	リレーショナルデータベースでは、データは柔軟にクエリできますが、クエリは比較的コストが高く、トラ	一方、DynamoDB のような NoSQL データベースでは、データは限られた数の方法で効率的にクエリできます

タスクの種類	リレーショナルデータベース	NoSQL データベース
	フィックが多い状況ではスケールがうまくいかない場合があります (「DynamoDB でリレーショナルデータをモデル化するための最初のステップ」を参照)。リレーショナルデータベースアプリケーションでは、ストアドプロシージャ、SQL サブクエリ、一括更新クエリ、集計クエリにビジネスロジックを実装する場合があります。	が、その範囲外では、クエリは高コストで低速になりがちです。DynamoDB への書き込みはシングルトンです。以前にストアドプロシージャで実行されていたアプリケーションのビジネスロジックは、Amazon EC2 や AWS Lambda などのホストで実行されるカスタムコードによって DynamoDB の外部で実行されるようにリファクタリングする必要があります。
データベースの設計	実装の詳細やパフォーマンスを気にせずに柔軟に設計できます。クエリの最適化は一般的にスキーマ設計には影響しませんが、正規化は重要です。	最も一般的で重要なクエリをできるだけ速く、安価にするために、具体的にスキーマを設計します。データ構造は、ビジネスユースケースの特定の要件に合わせて調整されています。

NoSQL データベースの設計には、リレーショナルデータベース管理システム (RDBMS) の設計とは異なる考え方が必要です。RDBMS の場合は、アクセスパターンを考慮せずに正規化されたデータモデルを作成できます。その後、新しい課題とクエリの要件が発生したら、そのデータモデルを拡張することができます。各タイプのデータを独自のテーブルに整理できます。

NoSQL の設計では、回答すべき質問が判明したら、DynamoDB のスキーマを設計できます。ビジネス上の問題とアプリケーションの読み取り/書き込みパターンを理解することが不可欠です。さらに、DynamoDB アプリケーションで維持するテーブルをできるだけ少なくする必要もあります。テーブルの数が少なくなると、スケーラビリティが向上し、必要なアクセス権限の管理が少なくなり、DynamoDB アプリケーションのオーバーヘッドが削減されます。また、バックアップコストを全体的に低く抑えるのにも役立ちます。

DynamoDB のリレーショナルデータをモデル化し、フロントエンドアプリケーションの新しいバージョンを構築するタスクについては、別のトピックで説明します。このガイドは、DynamoDB を使用するために構築された新しいバージョンのアプリケーションがあることを前提としていますが、カットオーバー中に履歴データを移行して同期する最適な方法を決定する必要はあります。

サイズに関する考慮事項

DynamoDB テーブルに格納する各項目 (行) の最大サイズは 400 KB です。詳細については、「the section called “クォータ”」を参照してください。項目のサイズは、項目のすべての属性名と属性値の合計サイズによって決まります。詳細については、「the section called “項目サイズと形式”」を参照してください。

アプリケーションで DynamoDB のサイズ制限よりも多くのデータを格納する必要がある場合は、項目を項目コレクションに分割するか、項目データを圧縮するか、項目をオブジェクトとして Amazon Simple Storage Service (Amazon S3) に格納して Amazon S3 オブジェクト識別子を DynamoDB 項目に格納します。「the section called “大きな項目”」を参照してください。項目の更新にかかるコストは、項目のフルサイズに基づいて決まります。既存の項目を頻繁に更新する必要があるワークロードでは、1 ~ 2 KB の小さい項目の方が、大きい項目よりも更新にかかるコストが低くなります。項目コレクションの詳細については、「the section called “項目コレクションの操作”」を参照してください。

パーティションとソートキーの属性、その他のテーブル設定、項目のサイズと構造、およびセカンダリインデックスを作成するかどうかを選択するときは、必ず DynamoDB のモデリングに関するドキュメントと「the section called “コスト最適化”」のガイドを確認してください。DynamoDB の機能と制限の範囲内でコスト効率よく DynamoDB ソリューションを運用できるように、移行計画を必ずテストしてください。

DynamoDB への移行の仕組みを理解する

利用可能な移行ツールを確認する前に、DynamoDB による書き込みの処理方法を検討してください。

デフォルトの最も一般的な書き込み操作は、単一の `PutItem` API オペレーションです。ループ内で `PutItem` オペレーションを実行してデータセットを処理できます。DynamoDB は実質的に無制限の同時接続をサポートしているため、MapReduce や Spark などの大規模なマルチスレッドのロードルーチンを設定して実行できると仮定すると、書き込み速度はターゲットテーブルのキャパシティ (これも通常は無制限) によってのみ制限されます。

DynamoDB にデータをロードするときは、ローダーの書き込み速度を把握することが重要です。ロードする項目 (行) のサイズが 1 KB 以下の場合、この速度は単に 1 秒あたりの項目数となります。これで、この速度を処理するのに十分な WCU (書き込みキャパシティユニット) をターゲットテーブルにプロビジョニングできます。ローダーがプロビジョニングされたキャパシティを 1 秒でも超えた場合、追加のリクエストはスロットリングされるか、完全に拒否される可能性があります。スロットリングは、DynamoDB コンソールの [モニタリング] タブにある CloudWatch チャートで確認できます。

2 つ目のオペレーションは、`BatchWriteItem` という関連 API を使用して実行できます。`BatchWriteItem` では、最大 25 件の書き込みリクエストを 1 つの API コールにまとめることができます。これらはサービスによって受信され、テーブルへの個別の `PutItem` リクエストとして処理されます。現在、`BatchWriteItem` を選択すると、`PutItem` によるシングルトン呼び出し時に、AWS SDK に備わっている自動再試行のメリットを享受できません。そのため、エラー (スロットリング例外など) が発生した場合は、`BatchWriteItem` への応答呼び出しで失敗した書き込みのリストを探す必要があります。CloudWatch スロットリングチャートで警告が検出された場合のスロットリング警告の処理については、「the section called “スロットリング”」を参照してください。

3 つ目のタイプのデータインポートには、DynamoDB の S3 からのインポート機能を使います。この機能により、Amazon S3 で大規模なデータセットをステージングし、そのデータを新しいテーブルに自動的にインポートするよう DynamoDB に指示できます。インポートは即時では行われず、データセットのサイズに比例した時間がかかりますが、ETL プラットフォームやカスタム DynamoDB コードが必要ないので便利です。DynamoDB は、インポートによって作成された新しいテーブルにデータをロードします。現在、既存のテーブルにデータをロードすることはできません。DynamoDB は、変換なしでデータをそのままインポートします。`PutItem` と同様に、アップストリームプロセスが必要であり、選択した形式でデータを Amazon S3 バケットに書き込みます。

DynamoDB への移行に役立つツール

データを DynamoDB に移行するために使用できる一般的な移行ツールと ETL ツールがいくつかあります。

Amazon は、AWS Database Migration Service (DMS)、AWS Glue、Amazon EMR、Amazon Managed Streaming for Apache Kafka など、移行に使用できるデータツールを多数提供しています。これらのツールはすべてダウンタイム移行に使用可能であり、リレーショナルデータベースの変更データキャプチャ (CDC) 機能を活用してオンライン移行をサポートできます。ツールを選択する場合は、組織が各ツールに関して持っているスキルセットと経験と共に、各ツールの機能、パフォーマンス、コストを考慮することが役立ちます。

多くのお客様は、移行プロセス用のカスタムデータ変換を構築するために、独自の移行スクリプトとジョブを作成することを選択しています。大量の書き込みトラフィックや定期的な大規模一括ロードジョブを伴う大容量の DynamoDB テーブルを運用する予定がある場合は、書き込みトラフィックが多いときの DynamoDB の動作に詳しくなるために、移行コードを自分で書くこともできます。スロットリング処理や効率的なテーブルプロビジョニングなどのシナリオは、プロジェクトの早い段階で実際に移行を行う際に経験できます。

DynamoDB に移行するための適切な戦略の選択

大規模なリレーショナルデータベースアプリケーションは、100 以上のテーブルを使用し、複数の異なるアプリケーション機能をサポートしている場合があります。大規模な移行を行う場合は、アプリケーションを小さなコンポーネントまたはマイクロサービスに分割して、テーブルの小さなセットを一度に移行することを検討してください。その後、追加のコンポーネントを段階的に DynamoDB に移行できます。

移行戦略を選択する際、さまざまな要因に基づいて 1 つのソリューションを決定する場合があります。次の決定ツリーは、要件とリソースに応じて利用可能なオプションを簡略化したものです。ここでは、概念について簡単に説明します (詳細はこのガイドの後半で説明します)。

- オフライン移行: 移行時にある程度のダウンタイムがアプリケーションで許容される場合は、移行プロセスが簡素化されます。
- ハイブリッド移行: この方法では、読み取りは許可するが書き込みは許可しない、読み取りと挿入は許可するが更新と削除は許可しないなど、移行時に部分的な稼働時間を確保できます。
- オンライン移行: 移行時にダウンタイムを一切許容しないアプリケーションは移行しにくいため、綿密な計画とカスタム開発が必要になる場合があります。この場合に重要なのは、カスタム移行プロセスを構築するコストと、カットオーバー時のダウンタイムによってビジネスに生じる損失を見積もり、比較検討することです。

If	And	THEN
データ移行を実行するために、メンテナンスウィンドウ中にアプリケーションをしばらく停止してもかまわない場合。これはオフライン移行です。		AWS DMS でフルロードタスクを使用してオフライン移行を実行してください。必要に応じて、SQL の `VIEW` でソースデータを事前に形成します。

If	And	THEN
移行時に、アプリケーションを読み取り専用モードで実行できればよいという場合。これはハイブリッド移行です。		アプリケーションまたはソースデータベース内での書き込みを無効にします。AWS DMS でフルロードタスクを使用してオフライン移行を実行してください。
移行時に、アプリケーションで更新や削除ができなくても、読み取りや新規レコードの挿入ができればよいという場合。これはハイブリッド移行です。	アプリケーション開発スキルがあり、すべての新しいレコードについて DynamoDB への書き込みを含む二重書き込みを実行するために、既存のリレーショナルアプリを更新できる場合。	AWS DMS でフルロードタスクを使用してオフライン移行を実行してください。同時に、既存のアプリを読み取りと二重書き込みが可能なバージョンにしてデプロイします。
ダウンタイムを最小限に抑えて移行する必要がある場合。これはオンライン移行です。	• スキーマを大幅に変更せずに、ソーステーブルを 1 対 1 で DynamoDB に移行する場合。	AWS DMS でオンラインデータ移行を実行してください。一括ロードタスクを実行し、続いて CDC 同期タスクを実行します。
ダウンタイムを最小限に抑えて移行する必要がある場合。これはオンライン移行です。	• スタックされたスキーマまたは単一テーブルの理念に従って、ソーステーブルをより少ない数の DynamoDB テーブルに結合する場合。 • バックエンドデータベース開発のスキルがあり、SQL ホストに予備のキャパシティがある場合。	SQL データベース内に NoSQL 対応テーブルを作成してください。JOIN、UNION、VIEW、トリガー、ストアドプロシージャを使用してデータの入力と同期を行います。

If	And	THEN
ダウンタイムを最小限に抑えて移行する必要がある場合。これはオンライン移行です。	• 単一テーブルの理念に従って、ソーステーブルをより少ない数の DynamoDB テーブルに結合する場合。例えば、次のようになります。 • バックエンドデータベース開発のスキルがなく、SQL ホストに予備のキャパシティもない場合。	ハイブリッドまたはオフラインの移行アプローチを検討してください。
ダウンタイムを最小限に抑えて移行する必要がある場合。これはオンライン移行です。	過去のトランザクションデータの移行は省略しても問題ない、または移行の代わりに Amazon S3 にアーカイブしてもよい場合。いくつかの小さな静的テーブルを移行するだけでよい場合。	スクリプトを記述するか、任意の ETL ツールを使用してテーブルを移行してください。必要に応じて、SQL の `VIEW` でソースデータを事前に形成します。

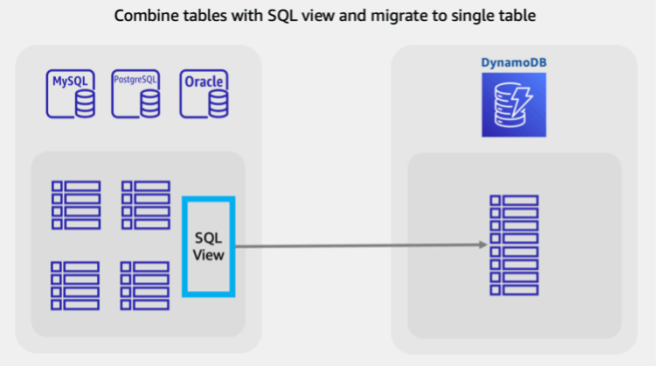
DynamoDB へのオフライン移行を実行する

オフライン移行は、移行時にダウンタイムを許容できる場合に適しています。リレーショナルデータベースは通常、メンテナンスとパッチ適用のために毎月少なくともある程度のダウンタイムを必要とし、ハードウェアのアップグレードやメジャーリリースのアップグレードではダウンタイムがさらに長くなることもあります。

Amazon S3 は移行時のステージングエリアとして使用できます。CSV (カンマ区切り値) または DynamoDB JSON 形式で保存されたデータは、DynamoDB の S3 からのインポート機能を使用して新しい DynamoDB テーブルに自動的にインポートできます。

独自の NoSQL アクセスパターンを活用するために複数のテーブルを結合することが望ましい場合があります (4 つのレガシーテーブルを 1 つの DynamoDB テーブルに変換するなど)。通常、単一のキーと値のドキュメントに対するリクエスト、または事前にグループ化された項目コレクションに対するクエリは、複数テーブルの結合を実行する SQL データベースよりもレイテンシーが短くなります。ただし、これによって移行タスクはより困難になります。SQL ビューを使ってソースデータ

ベース内で処理を行い、4 つのテーブルすべてを 1 つにまとめた単一のデータセットを作成することもできます。



このビューでは、テーブルを非正規化されたテーブルに JOIN することも、SQL の UNION を使用してエンティティを正規化されたままでテーブルをスタックすることもできます。リレーショナルデータの再形成に関する重要な決定事項については、こちらのビデオで説明しています。オフライン移行では、ビューを使用してテーブルを結合することが、DynamoDB の単一テーブルスキーマのデータを形成するのに最適な方法です。

プラン

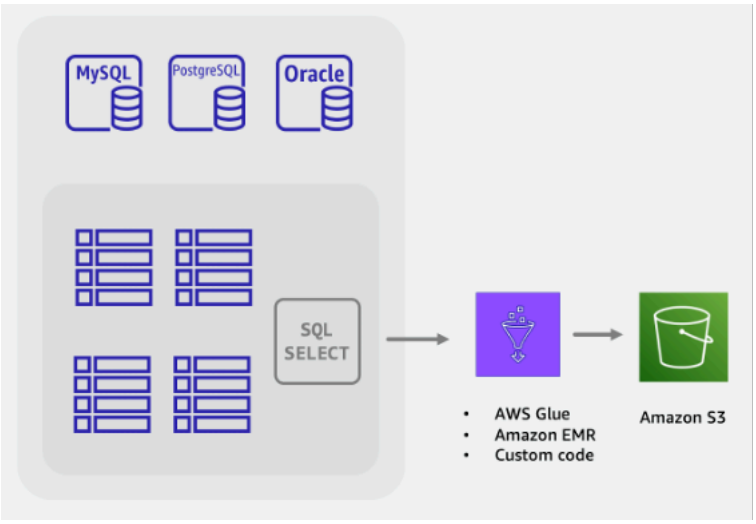
Amazon S3 を使用してオフライン移行を実行する

ツール

- SQL データを抽出して変換し、そのデータを S3 バケットに保存する次のような ETL ジョブ
  - AWS Database Migration Service は、履歴データを一括ロードし、変更データキャプチャ (CDC) レコードを処理してソーステーブルとターゲットテーブルを同期できるサービスです。
  - AWS Glue
  - Amazon EMR
  - 独自のカスタムコード
- DynamoDB の S3 からのインポート機能

オフライン移行ステップ

1. SQL データベースにクエリを実行し、テーブルデータを DynamoDB JSON または CSV 形式に変換して、S3 バケットに保存する ETL ジョブを構築します。



2. DynamoDB の S3 からのインポート機能が呼び出され、新しいテーブルを作成して、S3 バケットからデータを自動的にロードします。

完全なオフライン移行はシンプルで簡単ですが、アプリケーションの所有者やユーザーにはおそらくあまり好まれません。移行時にアプリケーションのサービスがまったく提供されないのではなく、提供するサービスのレベルが下がるだけであれば、ユーザーも助かるでしょう。

オフライン移行時に書き込みだけを無効にして、読み取りは通常どおり継続できるようにする機能を追加できます。こうすることで、アプリケーションのユーザーは、リレーショナルデータの移行中でも既存のデータを安全に参照してクエリを実行できます。この方法が適している場合は、引き続きハイブリッド移行についてお読みください。

DynamoDB へのハイブリッド移行を実行する

読み取りおよび書き込み操作はどのデータベースアプリケーションでも実行されますが、ハイブリッド移行またはオンライン移行を計画する際には、実行される書き込み操作の種類を考慮する必要があります。データベースへの書き込みは、挿入、更新、削除の 3 つのバケットに分類できます。アプリケーションによっては、削除をすぐに処理する必要がない場合があります。これらのアプリケーションは、削除を月末などの一括クリーンアッププロセスに延期できます。こうした種類のアプリケーションは、部分的な稼働時間を確保しながら、より簡単に移行できます。

計画

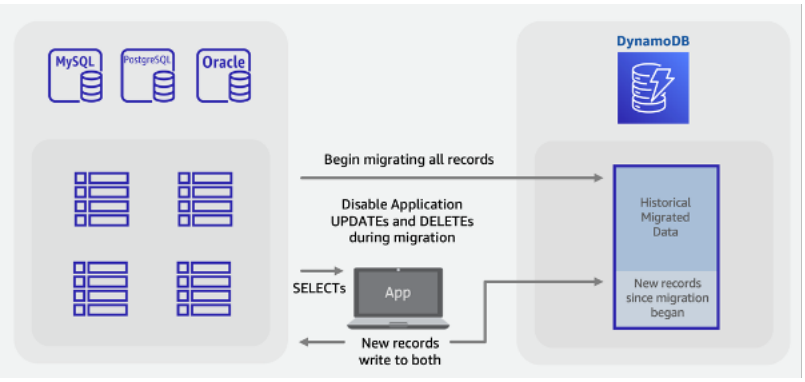
アプリケーションの二重書き込みによるオンライン/オフラインのハイブリッド移行を実行する

ツール

- SQL データを抽出して変換し、そのデータを S3 バケットに保存する次のような ETL ジョブ
  - AWS DMS
  - AWS Glue
  - Amazon EMR
  - 独自のカスタムコード

ハイブリッド移行ステップ

1. ターゲットの DynamoDB テーブルを作成します。このテーブルには、過去のバルクデータと新しいライブデータの両方が格納されます。
2. レガシーアプリケーションで削除と更新を無効にし、すべての挿入が SQL データベースと DynamoDB の両方への二重書き込みとして実行されるようにして、新しいバージョンのアプリケーションを作成します。
3. ETL ジョブまたは AWS DMS タスクを開始して、既存のデータをバックフィルすると同時に、新しいバージョンのアプリケーションをデプロイします。
4. バックフィルジョブが完了すると、DynamoDB には既存のレコードと新しいレコードがすべて揃い、アプリケーションのカットオーバーの準備が整います。



> **Note**
>
> バックフィルジョブでは SQL から DynamoDB に直接書き込みが行われます。オフライン移行の例のように S3 インポート機能を使用することはできません。この機能で作成される新しいテーブルは、DynamoDB がデータをロードするまで有効にならないためです。

各テーブルを 1 対 1 で移行して DynamoDB へのオンライン移行を実行する

多くのリレーショナルデータベースには、変更データキャプチャ (CDC) と呼ばれる機能があり、ユーザーは特定の時点より前または後にテーブルに加えられた変更のリストをリクエストできます。CDC は内部ログを使用してこの機能を有効にするため、テーブルにタイムスタンプ列がなくても動作します。

SQL テーブルのスキーマを NoSQL データベースに移行する際、データを結合して再形成し、テーブルの数を減らすことができます。そうすることで、データを 1 か所に集めることができ、複数ステップの読み取り操作において関連データを手動で結合する必要がなくなります。ただし、単一テーブルのデータ形成は必ずしも必要ではなく、テーブルを 1 対 1 で DynamoDB に移行することもあります。こうした 1 対 1 のテーブル移行は、この種の移行をサポートする一般的な ETL ツールを使ってソースデータベースの CDC 機能を活用すれば、それほど複雑ではありません。各行のデータは新しい形式に変換することも可能ですが、各テーブルの範囲は変わりません。

SQL テーブルを 1 対 1 で DynamoDB に移行することを検討する場合は、DynamoDB がサーバー側の結合をサポートしないという点に注意してください。複数のテーブルのデータを組み合わせるには、アプリケーションにロジックを追加する必要があります。

計画

AWS DMS を使用して各テーブルの DynamoDB へのオンライン移行を実行する

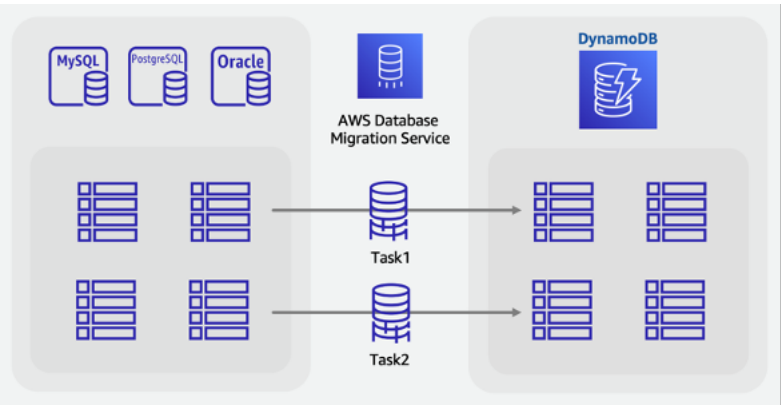
ツール

- AWS DMS (DMS)

オンライン移行ステップ

1. ソーススキーマ内の移行するテーブルを特定します。
2. ソースと同じキー構造で同数のテーブルを DynamoDB で作成します。
3. AWS DMS にレプリケーションサーバーを作成し、ソースとターゲットのエンドポイントを設定します。
4. 行ごとの必要な変換 (連結列、日付の ISO-8601 文字列形式への変換など) を定義します。
5. [フルロードと変更データキャプチャ (CDC)] で、テーブルごとに移行タスクを作成します。
6. レプリケーションフェーズの処理が始まるまで、これらのタスクを監視します。

7. この時点で、検証監査をすべて実行すれば、DynamoDB へ読み取りと書き込みを行うアプリケーションにユーザーを切り替えることができます。



カスタムステージングテーブルを使用して DynamoDB へのオンライン移行を実行する

上のオフライン移行シナリオと同様に、独自の NoSQL アクセスパターンを活用するためにテーブルを結合することを選択できます (4 つのレガシーテーブルを 1 つの DynamoDB テーブルに変換するなど)。SQL の `VIEW` を使ってソースデータベース内で処理を行い、4 つのテーブルすべてを 1 つにまとめた単一のデータセットを作成することもできます。

ただし、変化するライブデータを伴うオンライン移行では、CDC 機能は活用できません。`VIEW` 内ではサポートされないためです。テーブルに最終更新日時のタイムスタンプ列があり、その列が `VIEW` に組み込まれている場合は、その列を使用して同期による一括ロードを実行するカスタム ETL ジョブを構築できます。

この課題に対する新しいアプローチとしては、ビュー、ストアドプロシージャ、トリガーといった標準の SQL 機能を使用して、最終的に必要な DynamoDB NoSQL 形式で新しい SQL テーブルを作成します。

データベースサーバーに予備の容量がある場合は、移行の開始前に、この単一のステージングテーブルを作成できます。そのためには、既存のテーブルからの読み取り、必要に応じたデータ変換、新しいステージングテーブルへの書き込みを行うストアドプロシージャを作成します。テーブルの変更をステージングテーブルにリアルタイムで複製するための一連のトリガーを追加できます。会社のポリシーでトリガーが許可されていない場合は、ストアドプロシージャを変更しても同様の結果が得られる可能性があります。データを書き込むプロシージャに数行のコードを加え、同じ変更をステージングテーブルに追加で書き込みます。

このステージングテーブルをレガシーアプリケーションテーブルと完全に同期された場所に置いておくと、ライブ移行の出発点として最適です。このテーブルでは、AWS DMS など、データベース CDC を使ってライブ移行を実行するツールが使用できるようになっています。この方法のメリットは、リレーショナルデータベースエンジンで利用可能なよく知られた SQL スキルと機能を使用できることです。

計画

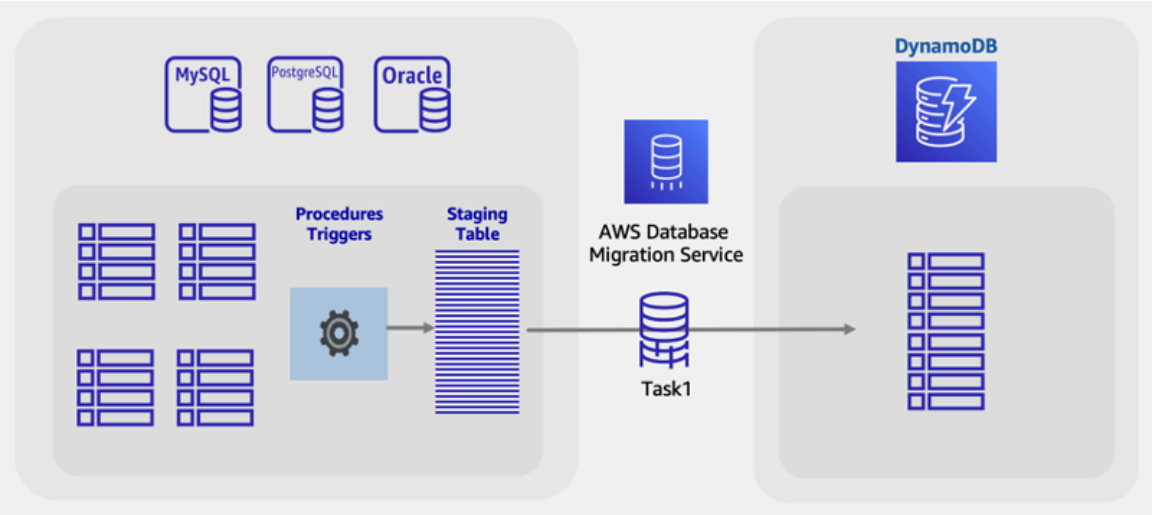
AWS DMS と SQL ステージングテーブルを使用してオンライン移行を実行する

ツール

- カスタム SQL ストアドプロシージャまたはトリガー
- AWS DMS

オンライン移行ステップ

1. ソースのリレーショナルデータベースエンジン内に、予備のディスク容量と処理容量がある程度あることを確認します。
2. タイムスタンプまたは CDC 機能を有効にして、SQL データベースに新しいステージングテーブルを作成します。
3. ストアドプロシージャを記述して実行し、既存のリレーショナルテーブルデータをステージングテーブルにコピーします。
4. トリガーをデプロイするか、既存のプロシージャを変更して、既存のテーブルに通常の書き込みを実行しながら新しいステージングテーブルに二重書き込みが行われるようにします。
5. AWS DMS を実行して、このソーステーブルをターゲットの DynamoDB テーブルに移行して同期します。



このガイドでは、ダウンタイムを最小限に抑えることと、一般的なデータベースツールや手法を使用することに重点を置いて、リレーショナルデータベースのデータを DynamoDB に移行する際の考慮事項とアプローチをいくつか紹介しました。詳細については次を参照してください:

- AWS DMS ユーザーガイド
- AWS Glue ユーザーガイド
- RDBMS から DynamoDB への移行のベストプラクティス

DynamoDB 用の NoSQL Workbench

Amazon DynamoDB 用の NoSQL Workbench は、最新のデータベース開発および運用向けのクロスプラットフォームのクライアント側 GUI アプリケーションです。Windows、macOS、Linux で使用できます。NoSQL Workbench を使用すると、DynamoDB データモデルを設計し、アクセスパターンを実際の DynamoDB オペレーションとして定義し、サンプルデータを使用して検証できます。さらに、データモデルをプロジェクトに整理することもできます。NoSQL Workbench には DynamoDB local が含まれており、データモデルをクラウドにコミットする前にテーブルとインデックスをテストできます。DynamoDB local とその要件の詳細については、「DynamoDB local (ダウンロード可能バージョン) のセットアップ」を参照してください。

データモデラー

NoSQL Workbench for DynamoDB を使用すると、新しいプロジェクトを最初から開始することも、ユースケースに合ったサンプルプロジェクトを使用することもできます。次に、テーブルとグローバルセカンダリインデックスを設計し、属性を定義して、サンプルデータを設定します。アクセスパターンを実際の DynamoDB オペレーション (PutItem、UpdateItem、Query など) として視覚化し、設定されたサンプルデータに対してこれらのオペレーションを実行して、アクセスパターンが意図したとおりに動作することを検証し、検証結果に基づいてデータモデルを調整することもできます。最後に、検証後、さらなるテストと本番環境での使用のために、モデルを DynamoDB local または AWS アカウントのいずれかにコミットします。コラボレーションでは、設計済みのデータモデルをインポートおよびエクスポートできます。詳細については、「NoSQL Workbench を使用したデータモデルの構築」を参照してください。

オペレーションビルダー

NoSQL Workbench は、クエリを開発およびテストするための豊富なグラフィカルユーザーインターフェイスを提供します。オペレーションビルダーを使用して、データセットを表示、探索、およびクエリできます。構造化オペレーションビルダーは、射影式、条件式をサポートし、複数の言語でサンプルコードを生成します。ある Amazon DynamoDB アカウントのテーブルのクローンを別のリージョンの別のアカウントで直接作成できます。また、DynamoDB Local と Amazon DynamoDB アカウントの間でテーブルを直接クローンして、開発環境間におけるテーブルのキースキーマ (およびオプションで GSI スキーマと項目) のコピーを高速化することもできます。詳細については、「NoSQL Workbench を使用したデータセットの探索とオペレーションの構築」を参照してください。

次の動画では、NoSQL Workbench を使用したデータモデリングの概念について詳しく説明します。

トピック

- DynamoDB 用の NoSQL Workbench のダウンロード
- NoSQL Workbench を使用したデータモデルの構築
- NoSQL Workbench を使用したデータセットの探索とオペレーションの構築
- NoSQL Workbench のサンプルデータモデル
- NoSQL Workbench のリリース履歴

DynamoDB 用の NoSQL Workbench のダウンロード

Amazon DynamoDB 用の NoSQL Workbench および DynamoDB local をダウンロードするには、以下の手順に従います。

NoSQL ワークベンチおよび DynamoDB local をダウンロードするには

- オペレーティングシステムに適したバージョンの NoSQL Workbench をダウンロードします。

オペレーティングシステム	ダウンロードリンク
macOS (Intel)	macOS (Intel) 用のダウンロード
macOS (Apple シリコン)	macOS (Apple シリコン) 用のダウンロード
Windows	Windows 用のダウンロード
Linux	Linux 用のダウンロード

> **Note**
>
> NoSQL Workbench には、インストールプロセスの一部として DynamoDB local が含まれています。
> DynamoDB local を実行するには、Java ランタイム環境 (JRE) バージョン 17.x 以降が必要です。

> **Note**
>
> NoSQL Workbench は、Ubuntu 12.04、Fedora 21、Debian 8、またはこれらの Linux ディストリビューションの新しいバージョンをサポートしています。
> Ubuntu のインストールには、`libfuse2` と `curl` という 2 つの前提条件となるソフトウェアが必要です。
> Ubuntu 22.04 では、libfuse2 はデフォルトでインストールされなくなりました。これを解決するには、`sudo add-apt-repository universe && sudo apt install libfuse2` を実行して最新の Ubuntu バージョンを実行してインストールしてください。
> cURL の場合は、`sudo apt update && sudo apt upgrade && sudo apt install curl` を実行します。

NoSQL Workbench を使用したデータモデルの構築

Amazon DynamoDB 用 NoSQL Workbench のデータモデラーツールを使用して、新しいデータモデルを構築したり、アプリケーションのデータアクセスパターンを満たす既存のデータモデルに基づいてモデルを設計したりできます。データモデラーには、使用開始に役立つサンプルデータモデルがいくつか含まれています。

トピック

- 新しいデータモデルの作成
- 既存のデータモデルのインポート
- 既存のデータモデルの編集
- データモデルへのサンプルデータの追加
- アクセスパターンの追加と検証
- CSV ファイルからサンプルデータのインポート
- ファセット
- 集計ビューを使用したデータモデル内のすべてのテーブルの表示
- データモデルのエクスポート
- DynamoDB へのデータモデルのコミット

新しいデータモデルの作成

以下のステップに従って、NoSQL Workbench を使用して Amazon DynamoDB に新しいデータモデルを作成します。

新しいデータモデルを作成するには

1. NoSQL Workbench を開き、メイン画面で [モデルを手動で作成する] を選択します。

   最初のテーブルの空の設定を含む新しいページが開きます。NoSQL Workbench は、すべての新しいデータモデルをデフォルト名 (untitled-2 など) で作成し、[ドラフト] プロジェクトフォルダに追加します。

2. [テーブル設定画面] で、次のように指定します。

   - [Table name (テーブル名)] — テーブルの一意な名前を入力します。
   - [パーティションキー] – パーティションキー名を入力し、そのタイプを指定します。オプションで、サンプルデータを生成するためのより詳細なデータ型形式を選択することもできます。
   - [ソートキー] を追加する場合は、ソートキー名とそのタイプを指定します。オプションで、サンプルデータを生成するためのより詳細なデータ型形式を選択することができます。

   > **Note**
   >
   > プライマリキーの設計、パーティションキーの効果的な設計と使用、およびソートキーの使用の詳細については、以下を参照してください。
   >
   > - プライマリキー
   > - DynamoDB でパーティションキーを効率的に設計し、使用するためのベストプラクティス。
   > - ソートキーを使用してデータを DynamoDB で整理するためのベストプラクティス

3. 他の属性を追加して、モデルとアクセスパターンをより明確に検証できます。その他の属性を追加するには。

   - [属性の追加] を選択します。
   - 属性名とタイプを指定します。
   - オプションで、サンプルデータを生成するためのより詳細なデータ型形式を選択することができます。

4. グローバルセカンダリインデックスを追加する場合は、[Add global secondary index (グローバルセカンダリインデックスの追加)] を選択します。[Global secondary index name (グローバルセカンダリインデックス名)]、[Partition key (パーティションキー)] 属性、および [Projection type (プロジェクションタイプ)] を指定します。

   DynamoDB でグローバルセカンダリインデックスを使用する詳細方法については、「グローバルセカンダリインデックス」を参照してください。

5. オプションで、[ファセットを追加する]。ファセットは NoSQL Workbench の仮想コンストラクトです。これは DynamoDB の機能的コンストラクトではありません。NoSQL Workbench のファセットは、テーブル内のデータのサブセットのみを使用して、DynamoDB に対するアプリケーションのさまざまなデータアクセスパターンを視覚化するのに役立ちます。

   > **Note**
   >
   > アプリケーションが DynamoDB のデータにアクセスする方法を視覚化するには、ファセットではなく、アクセスパターンの追加と検証 を使用することをお勧めします。アクセスパターンは実際のデータベースインタラクションを反映し、ユースケースに適したデータモデルを構築するのに役立ちますが、ファセットは非機能的な視覚化です。

   [Add facet (ファセットの追加)] を選択します。次を指定します:

   - Facet name (ファセット名)。
   - このファセットビューを区別するのに役立つパーティションキーエイリアス。
   - テーブルに [ソートキー] を指定した場合の [ソートキーエイリアス]。
   - このファセットの一部である [属性] を選択します。

   さらにファセットを追加する場合は、このステップを繰り返します。

6. 最後に、[保存] を選択してテーブルを作成します。

7. 他の [テーブル] または [グローバルセカンダリインデックス] が必要な場合は、先ほど作成したテーブルの上にある [+] アイコンを選択します。

既存のデータモデルのインポート

Amazon DynamoDB 用 NoSQL Workbench を使用して、既存のモデルをインポートおよび変更することでデータモデルを構築できます。データモデルは、NoSQL Workbench モデル形式または AWS CloudFormation JSON テンプレート形式のいずれかでインポートできます。

データモデルをインポートするには

1. NoSQL Workbench を開き、メイン画面で [モデルのインポート] を選択します。NoSQL Workbench モデル形式または CloudFormation JSON テンプレートのいずれかをインポートできます。
2. インポートしたモデルを既存のプロジェクトに追加したり、新しいプロジェクトを作成したりできます。別のプロジェクトを選択しない場合、モデルはドラフトプロジェクトフォルダに追加されます。
3. [デスクトップを参照] を選択し、コンピュータからインポートするモデルを選択します。
4. モデルがインポートされたら、アクセスパターンの追加、テーブルまたはインデックスの編集、データをローカルで変更することができます。

既存のデータモデルの編集

既存のモデルを編集するには

1. 既存のモデルの変更を開始するには、モデラーページでモデルを開きます。
2. リソースセレクターパネルには、[テーブル] と [グローバルセカンダリインデックス] のリストが表示されます。
3. [テーブル] または [グローバルセカンダリインデックス] を編集するには、まずリソースセレクターパネルでその名前を選択し、上部のアクションアイコンを使用します。使用可能なアクションは、[削除]、[複製]、[編集] です。
4. [モデルの詳細] を編集する場合は、モデル名の横にある 3 つのドットアイコンを選択します。
5. そこから、[モデルの詳細を編集] を選択し、それに応じて情報を編集できます。
6. 同じメニューからモデルを [複製]、[移動]、[削除]、[エクスポート] することもできます。
7. 別のモデルに変更するには、メイン画面を再度開くか、モデル選択ドロップダウンを使用します。

データモデルへのサンプルデータの追加

サンプルデータを自動生成するには

1. NoSQL Workbench を開き、メイン画面で、サンプルデータを追加するモデルの名前を選択します。
2. メインコンテンツツールバーの追加アクション (3 つのドットアイコン) を選択し、[サンプルデータの追加] を選択します。
3. 生成するサンプルデータの項目数を入力し、[確認] を選択します。
4. サンプルデータを自動生成すると、1～5,000 行のデータを生成して、それをすぐに使用できます。きめ細かいサンプルデータ型を指定して、設計やテストのニーズに基づいた現実的なデータを作成できます。現実的なデータを生成する機能を使用するには、データモデラーで属性のサンプルデータ型フォーマットを指定する必要があります。サンプルデータ型フォーマットの指定については、「新しいデータモデルの作成」を参照してください。

サンプルデータを 1 項目ずつ追加するには

1. 編集するモデルを開き、サンプルデータを追加するテーブルを選択します。メインコンテンツツールバーの追加アクション (3 つのドットアイコン) を選択し、[データの編集] を選択します。
2. 行を追加、削除、編集できるようになりました。必要な変更を行ったら、[保存] を選択します。

アクセスパターンの追加と検証

Amazon DynamoDB 用の NoSQL Workbench を使用して、アクセスパターンを作成、保存、検証できます。

> **Note**
>
> 適切なアクセスパターンを識別する方法の詳細については、「データアクセスパターンの特定」を参照してください。

アクセスパターンを作成するには

1. NoSQL Workbench を開き、メイン画面で、アクセスパターンを追加するモデルの名前を選択します。

2. 左側で、[アクセスパターン] タブを選択し、[+] アイコンを選択します。
3. 次の画面で、[名前]、オプションの [説明]、アクセスパターンの [タイプ]、アクセスパターンをテストする [テーブル] または [グローバルセカンダリインデックス] を指定します。

> **Note**
>
> NoSQL Workbench は現在、アクセスパターンに対して次のオペレーションをサポートしていま
> す。`Scan`、`Query`、`GetItem`、`PutItem`、`UpdateItem`、`DeleteItem`。Amazon DynamoDB は、より広範なオペレーションをサポートしています。

4. アクセスパターンを作成したら、[検証] タブに切り替えて、データモデルがアクセスパターンに対して期待される結果を返すように設計されているかどうかを確認できます。テーブルのサンプルデータを自動生成する方法の詳細については、「データモデルへのサンプルデータの追加」を参照してください。アクセスパターンのタイプによって、サポートされる入力パラメータも異なります。

> **Note**
>
> アクセスパターンを検証するために、NoSQL Workbench は、テーブルとインデックスがメモリに保存されたポート `8001` (デフォルト) で別の DynamoDB local データベースを起動します。
>
> - NoSQL Workbench は、モデルからのサンプルデータを一時テーブルに自動的に追加します。
> - サンプルデータまたはデータモデル自体を編集すると、NoSQL Workbench は一時テーブルを更新します。
> - この一時データベースは、アプリケーションを閉じると消去されます。

アクセスパターンを編集するには

1. NoSQL Workbench を開き、メイン画面で、アクセスパターンを編集するモデルの名前を選択します。
2. 左側で、[アクセスパターン] タブを選択します。
3. アクセスパターンを編集するには、左側のリストから選択します。
4. 上部のバーで、[編集] を選択します。

CSV ファイルからサンプルデータのインポート

CSV ファイル内に既存のサンプルデータがある場合は、それを NoSQL Workbench にインポートすることができます。これにより、行単位で入力することなく、サンプルデータをモデルにすばやく入力できるようになります。

CSV ファイルの列名は、データモデルの属性名と一致する必要がありますが、同じ順序である必要はありません。例えば、データモデルに `LoginAlias`、`FirstName`、`LastName` と呼ばれる属性がある場合の場合、CSV 列は `LastName`、`FirstName`、`LoginAlias` になる可能性があります。

CSV ファイルから一度に最大 150 行をインポートできます。

CSV ファイルから NoSQL Workbench にデータをインポートするには

1. CSV データを [テーブル] にインポートするには、まずリソースパネルでテーブル名を選択し、次にメインコンテンツツールバーの追加アクション (3 つのドットアイコン) を選択します。
2. [サンプルデータのインポート] を選択します。
3. CSV ファイルを選択し、[開く] を選択します。CSV データがテーブルに追加されます。

ファセット

NoSQL Workbench では、ファセットを使用すると、ファセットの制約を満たさないレコードを表示することなく、テーブル内のデータのサブセットを表示できます。ファセットはビジュアルデータモデリングツールと見なされ、アクセスパターンのモデリングを純粋に補助するものであるため、DynamoDB で使用できる構造としては存在しません。

> **Note**
>
> アプリケーションが DynamoDB のデータにアクセスする方法を視覚化するには、ファセットではなく、アクセスパターンの追加と検証 を使用することをお勧めします。アクセスパターンは実際のデータベースインタラクションを反映し、ユースケースに適したデータモデルを構築するのに役立ちますが、ファセットは非機能的な視覚化です。

ファセットを作成するには

1. リソースセレクターパネルで、編集する [テーブル] を選択します。
2. 上部のバーで、[編集] を選択します。

3. [ファセットフィルター] セクションまで下にスクロールします。
4. [Add facet (ファセットの追加)] を選択します。次を指定します:

   - Facet name (ファセット名)。
   - このファセットビューを区別するのに役立つパーティションキーエイリアス。
   - テーブルに [ソートキー] を指定した場合の [ソートキーエイリアス]。
   - このファセットの一部である [属性] を選択します。

   さらにファセットを追加する場合は、このステップを繰り返します。

集計ビューを使用したデータモデル内のすべてのテーブルの表示

Amazon DynamoDB 用の NoSQL Workbench の集計ビューを使用して、データモデル内の全てのテーブルとインデックスを視覚化できます。各テーブルについて、以下の情報が表示されます。

- テーブルの列名
- サンプルデータ
- テーブルに関連付けられているすべてのグローバルセカンダリインデックス。NoSQL Workbench は、インデックスごとに次の情報を表示します。
  - インデックス列名
  - サンプルデータ

すべてのテーブル情報を表示するには

1. NoSQL Workbench を開き、メイン画面で、開くモデルの名前を選択します。
2. 上部のバーで、[集計ビュー] を選択します。すべてのテーブルとインデックスのデータが同じ画面に表示されます。

集約ビューをイメージとしてエクスポートするには

1. [集計ビュー] を選択し、3 つのドットアイコンを選択し、[集計ビューのエクスポート] を選択します。
2. すべてのテーブルとインデックスの PNG イメージを含むアーカイブがダウンロードできるようになります。

データモデルのエクスポート

Amazon DynamoDB 用 NoSQL Workbench を使用してデータモデルを作成した後、モデルを NoSQL Workbench モデル形式または AWS CloudFormation JSON テンプレート形式で保存およびエクスポートできます。

データモデルをエクスポートするには

1. NoSQL Workbench を開き、メイン画面で、編集するモデルの名前を選択します。
2. データモデル名の横にある 3 つのドットアイコンを選択して、[モデルをエクスポート] を選択します。
3. データモデルを NoSQL Workbench モデル形式または CloudFormation JSON テンプレート形式でエクスポートするかどうかを選択します。

   - NoSQL Workbench を使用してモデルを他のチームメンバーと共有したり、後で NoSQL Workbench にインポートしたりする場合は、NoSQL Workbench モデル形式を選択します。
   - モデルを AWS に直接デプロイしたり、コードとしてのインフラストラクチャワークフローに統合したりする場合は、CloudFormation JSON テンプレート形式を選択します。

DynamoDB へのデータモデルのコミット

データモデルに満足したら、モデルを Amazon DynamoDB にコミットできます。

> **Note**
>
> - このアクションにより、データモデルで表されるテーブルとグローバルセカンダリインデックス用のサーバー側リソースが AWS に作成されます。
> - NoSQL Workbench は、デフォルトでオンデマンド容量を持つテーブルとインデックスを作成します。

データモデルを DynamoDB にコミットするには

1. NoSQL Workbench を開き、メイン画面で、コミットするモデルの名前をクリックします。
2. 上部のバーで、[コミット] を選択します。
3. 既存の接続を選択するか、[新しい接続を追加] を選択して新しい接続を作成します。

- 新しい接続を追加するには、以下の情報を指定します。
  - アカウントエイリアス
  - AWS リージョン
  - アクセスキー ID
  - シークレットアクセスキー

  アクセスキーを取得する方法の詳細については、「AWS アクセスキーの取得」を参照してください。
- オプションで次の値を指定できます。
  - セッショントークン
  - IAM ロール ARN

4. DynamoDB local を使用する場合。
   1. [ローカル接続] タブを選択します。
   2. [新しい接続を追加] を選択します。
   3. [Connection name (接続名)] と [Port (ポート)] を指定します。

   > **Note**
   >
   > DynamoDB local を使用するには、NoSQL Workbench 画面の左下にある [DynamoDB local] トグルを使用して有効にします。

5. [コミット] を選択します。

NoSQL Workbench を使用したデータセットの探索とオペレーションの構築

Amazon DynamoDB 用 NoSQL Workbench は、クエリを開発およびテストするための豊富なグラフィカルユーザーインターフェイスを提供します。NoSQL Workbench のオペレーションビルダーを使用して、ライブデータセットを表示、探索、およびクエリできます。構造化オペレーションビルダーは、射影式、条件式をサポートし、複数の言語でサンプルコードを生成します。ある Amazon DynamoDB アカウントのテーブルのクローンを別のリージョンの別のアカウントで直接作成できます。また、DynamoDB Local と Amazon DynamoDB アカウントの間で直接テーブルのクローンを

作成すれば、テーブルのキースキーマ (オプションで GSI スキーマと項目) を開発環境間で迅速にコピーできます。オペレーションビルダーでは、最大 50 の DynamoDB データオペレーションを保存できます。

トピック

- ライブデータセットへの接続
- 複雑なオペレーションの構築
- NoSQL Workbench を使用したテーブルのクローン作成
- CSV ファイルへのデータのエクスポート

ライブデータセットへの接続

NoSQL Workbench を使用して Amazon DynamoDB テーブルに接続するには、まず AWS アカウントに接続する必要があります。

データベースへの接続を追加するには

1. NoSQL Workbench の左側のナビゲーションペインで [Operation builder (オペレーションビルダー)] アイコンを選択します。
2. [接続を追加] を選択します。
3. 以下の情報を指定します。

   - 接続名
   - AWS リージョン
   - アクセスキー ID
   - シークレットアクセスキー

   アクセスキーを取得する方法の詳細については、「AWS アクセスキーの取得」を参照してください。

   オプションで、次の値を指定できます。

   - セッショントークン
   - IAM ロール ARN
4. [接続] を選択します。

無料利用枠アカウントにサインアップせず、DynamoDB Local (ダウンロード可能バージョン) を使用する場合:

a. 接続画面の [Local] (ローカル) タブを選択します
b. 以下の情報を指定します。
   - 接続名
   - ポート
c. [接続] ボタンを選択します。

> **Note**
>
> DynamoDB Local に接続するには、ターミナルを使用して DynamoDB Local を手動で起動するか (「Deploying DynamoDB local on your computer」を参照)、NoSQL Workbench ナビゲーションメニューの DDB Local トグルを使用して DynamoDB Local を直接起動します。接続ポートが DynamoDB Local ポートと同じであることを確認してください。

5. 作成した接続で、[Open (開く)] を選択します。

DynamoDB データベースに接続すると、使用可能なテーブルのリストが左ペインに表示されます。テーブルのいずれかを選択すると、テーブルに保存されているデータのサンプルが返されます。

選択したテーブルに対してクエリを実行できるようになりました。

テーブルに対してクエリを実行するには、オペレーションの構築に関する次のセクション「複雑なオペレーションの構築」を参照してください。

複雑なオペレーションの構築

Amazon DynamoDB 用 NoSQL Workbench のオペレーションビルダーは、複雑なデータプレーンオペレーションが実行できる視覚的なインターフェイスを提供します。プロジェクション式と条件式のサポートが含まれています。オペレーションを作成したら、後で使用するために保存できます (最大 50 個のオペレーションを保存できます)。その後、[Saved Operations] (保存されたオペレーション) メニューで、頻繁に使用するデータプレーンオペレーションのリストを参照し、それらを使用して自動的に新しいオペレーションを設定および構築できます。これらのオペレーションのサンプルコードを複数の言語で生成することもできます。

NoSQL Workbench では、DynamoDB ステートメント用の PartiQL の構築をサポートしています。これにより、SQL 互換のクエリ言語を使用して DynamoDB とやり取りできます。NoSQL Workbench では、DynamoDB CRUD API オペレーションの構築もサポートしています。

NoSQL Workbench を使用してオペレーションを構築するには、左側のナビゲーションペインで [Operation builder] (オペレーションビルダー) アイコンを選択します。

トピック

- PartiQL ステートメントの構築
- API オペレーションの構築

PartiQL ステートメントの構築

NoSQL Workbench を使用して PartiQL for DynamoDB ステートメントを構築するには、NoSQL Workbench UI の上部にある [PartiQL エディタ] を選択します。

オペレーションビルダーでは、次の PartiQL ステートメントタイプを作成できます。

トピック

- シングルトンステートメント
- 取引
- バッチ

シングルトンステートメント

PartiQL ステートメントのコードを実行または生成するには、次の操作を行います。

1. ウィンドウの上部にある [PartiQL エディタ] を選択します。
2. 有効な PartiQL ステートメントを入力します。
3. ステートメントでパラメータを使用する場合
    a. [オプションのリクエストパラメータ] を選択します。
    b. [パラメータの追加] を選択します。
    c. 属性のタイプと値を入力します。
    d. パラメータを追加する場合は、ステップ b および c を繰り返します。
4. コードを生成する場合は、[Generate code (コードの生成)] を選択します。

表示されたタブから目的の言語を選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。

5. オペレーションをすぐに実行する場合は、[実行] をクリックします。
6. このオペレーションを後で使用するために保存する場合は、[Save operation] (保存オペレーション) を選択します。次に、オペレーションの名前を入力して [Save] (保存) を選択します。

取引

PartiQL トランザクションのコードを実行または生成するには、次の操作を行います。

1. [その他のオペレーション] ドロップダウンで、[PartiQLTransaction] を選択します。
2. [Add a new statement] (新しいステートメントの追加) を選択します。
3. 有効な PartiQL ステートメントを入力します。

> **Note**
>
> 読み込みおよび書き込みオペレーションは、同じ PartiQL トランザクションのリクエストではサポートされていません。INSERT、UPDATE、および DELETE ステートメントでは、同じリクエストに SELECT ステートメントを指定することはできません。詳細については、「DynamoDB 用 PartiQL を使用したトランザクションの実行」を参照してください。

4. ステートメントでパラメータを使用する場合
   a. [Optional request parameters] (オプションのリクエストパラメータ) を選択します。
   b. [パラメータの追加] を選択します。
   c. 属性のタイプと値を入力します。
   d. パラメータを追加する場合は、ステップ b および c を繰り返します。
5. さらにステートメントを追加する場合は、ステップ 2 ~ 4 を繰り返します。
6. コードを生成する場合は、[コードの生成] を選択します。

   表示されたタブから目的の言語を選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。
7. オペレーションをすぐに実行する場合は、[実行] をクリックします。

8. このオペレーションを後で使用するために保存する場合は、[Save operation] (保存オペレーション) を選択します。次に、オペレーションの名前を入力して [Save] (保存) を選択します。

バッチ

PartiQL バッチのコードを実行または生成するには、次の操作を行います。

1. [その他のオペレーション] ドロップダウンで、[PartiQLBatch] を選択します。
2. [Add a new statement] (新しいステートメントの追加) を選択します。
3. 有効な PartiQL ステートメントを入力します。

> **Note**
>
> 読み込みおよび書き込みオペレーションは、同じ PartiQL バッチリクエストではサポートされていません。つまり、SELECT ステートメントを INSERT、UPDATE、DELETE ステートメントと同じリクエストに含めることはできません。同じ項目への書き込みオペレーションは許可されていません。BatchGetItem オペレーションと同様に、シングルトン読み込みオペレーションのみがサポートされています。スキャンおよびクエリオペレーションはサポートされていません。詳細については、「DynamoDB 用 PartiQL を使用した batch オペレーションの実行」を参照してください。

4. ステートメントでパラメータを使用する場合

   a. [オプションのリクエストパラメータ] を選択します。

   b. [パラメータの追加] を選択します。

   c. 属性のタイプと値を入力します。

   d. パラメータを追加する場合は、ステップ b および c を繰り返します。

5. さらにステートメントを追加する場合は、ステップ 2～4 を繰り返します。
6. コードを生成する場合は、[コードの生成] を選択します。

   表示されたタブから目的の言語を選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。

7. オペレーションをすぐに実行する場合は、[実行] をクリックします。
8. このオペレーションを後で使用するために保存する場合は、[Save operation] (保存オペレーション) を選択します。次に、オペレーションの名前を入力して [Save] (保存) を選択します。

API オペレーションの構築

NoSQL Workbench を使用して DynamoDB CRUD API を構築するには、NoSQL Workbench ユーザーインターフェイスの左側で [オペレーションビルダー] を選択します。

次に [開く] を選択して、接続を選択します。

オペレーションビルダーでは、以下の操作を実行できます。

- Delete Table
- Create Table
- Update Table

- Put Item
- Update Item
- Delete Item
- Query
- Scan
- Transact Get Items
- Transact Write Items

テーブルを削除する

`Delete Table` オペレーションを実行するには、次の操作を行います。

1. [テーブル] セクションで、削除するテーブルを見つけます。
2. 横方向の省略記号メニューで、[テーブルを削除] を選択します。
3. [テーブル名] を入力して、テーブルを削除することを確認します。
4. [削除] を選択します。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Delete table」を参照してください。

GSI を削除

`Delete GSI` オペレーションを実行するには、次の操作を行います。

1. [テーブル] セクションで、削除するテーブルの GSI を見つけます。
2. 横方向の省略記号メニューで、[GSI を削除] を選択します。
3. [GSI 名] を入力して、GSI を削除することを確認します。
4. [削除] を選択します。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Delete table」を参照してください。

テーブルの作成

`Create Table` オペレーションを実行するには、次の操作を行います。

1. [テーブル] セクションの横にある [+] アイコンを選択します。
2. 目的のテーブル名を入力します。
3. パーティションキーを作成します。
4. オプション: ソートキーを作成します。
5. キャパシティー設定をカスタマイズするには、[デフォルトのキャパシティー設定を使用] チェックボックスをオフにします。
   - これで、プロビジョンドキャパシティまたはオンデマンドキャパシティを選択できます。

     [Provisioned] (プロビジョンド) を選択すると、最小と最大の読み取りおよび書き込みのキャパシティユニットを設定できます。自動スケーリングを有効または無効にすることもできます。
   - テーブルの現在の設定が [オンデマンド] になっている場合は、プロビジョンドスループットを指定できません。
   - [オンデマンド] から [プロビジョンドスループット] に切り替えると、すべての GSI に自動スケーリング (最小: 1、最大: 10、ターゲット: 70%) が自動的に適用されます。
6. このテーブルを GSI なしで作成するには、[GSI をスキップして作成] を選択します。オプションで、[次へ] を選択して、この新しいテーブルで GSI を作成できます。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Create table」を参照してください。

GSI の作成

`Create GSI` オペレーションを実行するには、次の操作を行います。

1. GSI を追加する先のテーブルを見つけます。
2. 横方向の省略記号メニューで、[GSI の作成] を選択します。
3. [インデックス名] で、GSI に名前を付けます。
4. パーティションキーを作成します。
5. オプション: ソートキーを作成します。
6. ドロップダウンから [射影タイプ] オプションを選択します。
7. [GSI の作成] を選択します。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Create table」を参照してください。

テーブルを更新する

`Update Table` オペレーションでテーブルのキャパシティー設定を更新するには、次の操作を行います。

1. キャパシティー設定を更新するテーブルを見つけます。
2. 横方向の省略記号メニューで、[キャパシティー設定を更新] を選択します。
3. [プロビジョンド] または [オンデマンドキャパシティー] を選択します。

   [プロビジョンド] を選択すると、最小と最大の読み込みおよび書き込みキャパシティーユニットを設定できます。自動スケーリングを有効または無効にすることもできます。
4. [更新] を選択します。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Update table」を参照してください。

GSI の更新

`Update Table` オペレーションで GSI のキャパシティー設定を更新するには、次の操作を行います。

> **Note**
>
> デフォルトでは、グローバルセカンダリインデックスは基本テーブルのキャパシティー設定を継承します。基本テーブルがプロビジョンドキャパシティーモードである場合にのみ、グローバルセカンダリインデックスを別のキャパシティーモードに設定できます。プロビジョ

ニングモードのテーブルでグローバルセカンダリインデックスを作成するときには、そのインデックスに対して予想されるワークロードに応じた読み込み/書き込みキャパシティーユニットを指定する必要があります。詳細については、「グローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項」を参照してください。

1. キャパシティー設定を更新する GSI を見つけます。
2. 横方向の省略記号メニューで、[キャパシティー設定を更新] を選択します。
3. これで、プロビジョンドキャパシティまたはオンデマンドキャパシティを選択できます。

   [プロビジョンド] を選択すると、最小と最大の読み込みおよび書き込みキャパシティーユニットを設定できます。自動スケーリングを有効または無効にすることもできます。
4. [更新] を選択します。

このオペレーションの詳細については、「Amazon DynamoDB API リファレンス」の「Update table」を参照してください。

項目を配置する

`Put Item` オペレーションを使用して項目を作成します。`Put Item` オペレーションのコードを実行または生成するには、次の操作を行います。

1. 項目を作成するテーブルを見つけます。
2. [アクション] ドロップダウンで、[項目の作成] を選択します。
3. パーティションキー値を入力します。
4. ソートキーの値を入力します (存在する場合)。
5. キーではない属性を指定する場合は、次の操作を行います。

   a. [+ その他の属性を追加] を選択します。

   b. [属性の名前]、[タイプ]、[値] を指定します。
6. `Put Item` オペレーションを成功させるために条件式を満たす必要がある場合は、次の手順を実行します。

   a. [Condition (条件)] を選択します。

   b. 属性名、比較演算子、属性タイプ、および属性値を指定します。

   c. 他の条件が必要な場合は、[Condition (条件)] を再度選択します。

詳細については、「DynamoDB 条件式 CLI の例」を参照してください。

7. コードを生成する場合は、[コードの生成] を選択します。

   表示されたタブから目的の言語を選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。

8. オペレーションをすぐに実行する場合は、[実行] をクリックします。
9. このオペレーションを後で使用するために保存する場合は、[Save operation] (オペレーションの保存) を選択し、オペレーションの名前を入力して、[Save] (保存) を選択します。

このオペレーションの詳細については、Amazon DynamoDB API リファレンスの PutItem を参照してください。

項目を更新する

`Update Item` オペレーションのコードを実行または生成するには、次の操作を行います。

1. 項目を更新するテーブルを見つけます。
2. 項目を選択します。
3. 選択した式の属性名と属性値を入力します。
4. さらに式を追加する場合は、[式を更新] ドロップダウンリストで別の式を選択し、[+] アイコンを選択します。
5. `Update Item` オペレーションを成功させるために条件式を満たす必要がある場合は、次の手順を実行します。

   a. [Condition (条件)] を選択します。
   b. 属性名、比較演算子、属性タイプ、および属性値を指定します。
   c. 他の条件が必要な場合は、[Condition (条件)] を再度選択します。

   詳細については、「DynamoDB 条件式 CLI の例」を参照してください。

6. コードを生成する場合は、[Generate code (コードの生成)] を選択します。

   使用する言語のタブを選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。

7. オペレーションをすぐに実行する場合は、[実行] をクリックします。

8. このオペレーションを後で使用するために保存する場合は、[Save operation] (オペレーションの保存) を選択し、オペレーションの名前を入力して、[Save] (保存) を選択します。

このオペレーションの詳細については、Amazon DynamoDB API リファレンスの UpdateItem を参照してください。

項目を削除する

`Delete Item` オペレーションを実行するには、次の操作を行います。

1. 項目を削除するテーブルを見つけます。
2. 項目を選択します。
3. [アクション] ドロップダウンで、[項目の削除] を選択します。
4. [削除] を選択して、項目を削除することを確認します。

このオペレーションの詳細については、Amazon DynamoDB API リファレンスの DeleteItem を参照してください。

項目の複製

同じ属性を持つ新しい項目を作成することで、項目を複製できます。項目を複製するには、次の操作を行います。

1. 項目を複製するテーブルを見つけます。
2. 項目を選択します。
3. [アクション] ドロップダウンで、[項目の複製] を選択します。
4. 新しいパーティションキーを指定します。
5. 新しいソートキーを指定します (必要な場合)。
6. [実行] を選択します。

このオペレーションの詳細については、Amazon DynamoDB API リファレンスの DeleteItem を参照してください。

クエリ

`Query` オペレーションのコードを実行または生成するには、次の操作を行います。

1. NoSQL Workbench UI の上部で、[クエリ] を選択します。
2. パーティションキーの値を指定します。
3. `Query` オペレーションにソートキーが必要な場合は、以下の操作を実行します。

   a. [ソートキー] を選択します。

   b. 比較演算子と属性値を指定します。
4. [クエリ] を選択して、このクエリオペレーションを実行します。さらにオプションが必要な場合は、[その他のオプション] チェックボックスをオンにして、次の手順に進みます。
5. オペレーション結果とともにすべての属性を返す必要がない場合は、[Projection expression (プロジェクション式)] を選択します。
6. [+] アイコンを選択します。
7. クエリ結果とともに返される属性を入力します。
8. さらに属性が必要な場合は、[+] を選択します。
9. `Query` オペレーションを成功させるために条件式を満たす必要がある場合は、次の手順を実行します。

   a. [Condition (条件)] を選択します。

   b. 属性名、比較演算子、属性タイプ、および属性値を指定します。

   c. 他の条件が必要な場合は、[Condition (条件)] を再度選択します。

   詳細については、「DynamoDB 条件式 CLI の例」を参照してください。
10. コードを生成する場合は、[Generate code (コードの生成)] を選択します。

    使用する言語のタブを選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。
11. オペレーションをすぐに実行する場合は、[実行] をクリックします。
12. このオペレーションを後で使用するために保存する場合は、[Save operation] (オペレーションの保存) を選択し、オペレーションの名前を入力して、[Save] (保存) を選択します。

このオペレーションの詳細については、Amazon DynamoDB API リファレンスのクエリを参照してください。

Scan

`Scan` オペレーションのコードを実行または生成するには、次の操作を行います。

1. NoSQL Workbench UI の上部で、[スキャン] を選択します。
2. [スキャン] ボタンを選択して、この基本的なスキャンオペレーションを実行します。さらにオプションが必要な場合は、[その他のオプション] チェックボックスをオンにして、次の手順に進みます。
3. 属性名を指定して、スキャン結果をフィルタリングします。
4. オペレーション結果とともにすべての属性を返す必要がない場合は、[Projection expression (プロジェクション式)] を選択します。
5. スキャンオペレーションを成功させるために条件式を満たす必要がある場合は、次の手順を実行します。

   a. [Condition (条件)] を選択します。

   b. 属性名、比較演算子、属性タイプ、および属性値を指定します。

   c. 他の条件が必要な場合は、[Condition (条件)] を再度選択します。

   詳細については、「DynamoDB 条件式 CLI の例」を参照してください。
6. コードを生成する場合は、[Generate code (コードの生成)] を選択します。

   使用する言語のタブを選択します。これで、このコードをコピーしてアプリケーションで使用できるようになります。
7. オペレーションをすぐに実行する場合は、[実行] をクリックします。
8. このオペレーションを後で使用するために保存する場合は、[Save operation] (オペレーションの保存) を選択し、オペレーションの名前を入力して、[Save] (保存) を選択します。

TransactGetItems

`TransactGetItems` オペレーションのコードを実行または生成するには、次の操作を行います。

1. NoSQL Workbench UI の上部にある [その他のオペレーション] ドロップダウンで、[TransactGetItems] を選択します。
2. [TransactGetItem] の近くにある [+] アイコンを選択します。
3. パーティションキーを指定します。
4. ソートキーを指定します (必要な場合)。
5. [実行] を選択してオペレーションを実行するか、[オペレーションを保存] を選択して保存するか、[コードを生成] を選択してオペレーションのコードを生成します。

トランザクションの詳細については、「Amazon DynamoDB Transactions」を参照してください。

TransactWriteItems

`TransactWriteItems` オペレーションのコードを実行または生成するには、次の操作を行います。

1. NoSQL Workbench UI の上部にある [その他のオペレーション] ドロップダウンで、[TransactWriteItems] を選択します。
2. [アクション] ドロップダウンで、オペレーションを選択します。
3. [TransactWriteItem] の近くにある [+] アイコンを選択します。
4. [アクション] ドロップダウンで、実行するオペレーションを選択します。

   - `DeleteItem` の場合、項目を削除する オペレーションの手順に従います。
   - `PutItem` の場合、項目を配置する オペレーションの手順に従います。
   - `UpdateItem` の場合、項目を更新する オペレーションの手順に従います。

   アクションの順序を変更するには、左側のリストでアクションを選択し、上矢印または下矢印を選択してリスト内で上下に移動します。

   アクションを削除するには、リストでアクションを選択してから、[削除] (ごみ箱) アイコンを選択します。
5. [実行] を選択してオペレーションを実行するか、[オペレーションを保存] を選択して保存するか、[コードを生成] を選択してオペレーションのコードを生成します。

トランザクションの詳細については、「Amazon DynamoDB Transactions」を参照してください。

NoSQL Workbench を使用したテーブルのクローン作成

テーブルのクローンを作成すると、テーブルのキースキーマ (オプションで GSI スキーマと項目) が開発環境間でコピーされます。DynamoDB Local と Amazon DynamoDB アカウント間でテーブルのクローンを作成できます。また、あるアカウントのテーブルのクローンを別のリージョンのアカウントに作成して、迅速に実験を行うこともできます。

テーブルのクローンを作成するには

1. オペレーションビルダーで、接続とリージョンを選択します (DynamoDB Local ではリージョンの選択はできません)。

2. DynamoDB に接続したら、テーブルを参照し、クローンを作成するテーブルを選択します。
3. 横方向の省略記号メニューで、[クローン作成] オプションを選択します。
4. クローンの保存先の詳細を入力します。

   a. 接続を選択します。

   b. リージョンを選択します (リージョンは DynamoDB Local では使用できません)。

   c. 新しいテーブル名を入力します。

   d. クローン作成のオプションを選択します。

      i. [キースキーマ] はデフォルトで選択されており、選択を解除することはできません。デフォルトでは、テーブルのクローンを作成するとプライマリキーとソートキー (使用可能な場合) がコピーされます。

      ii. クローン対象のテーブルに GSI がある場合は、デフォルトで [GSI スキーマ] が選択されます。テーブルのクローンを作成すると、GSI プライマリキーとソートキー (使用可能な場合) がコピーされます。[GSI スキーマ] の選択を解除して、GSI スキーマのクローン作成は省略することもできます。テーブルのクローンを作成すると、ベーステーブルの容量設定が GSI の容量設定としてコピーされます。オペレーションビルダーで `UpdateTable` オペレーションを使用して、クローン作成の完了後にテーブルの GSI 容量設定を更新できます。

5. クローンを作成する項目の数を入力します。キースキーマとオプションで GSI スキーマのみをクローン作成する場合は、[クローンする項目] の値を 0 のままにしておくことができます。クローンを作成できる項目の最大数は 5000 です。
6. キャパシティモードを選択します。

   a. デフォルトでは、[オンデマンドモード]が選択されています。DynamoDB オンデマンドには、読み取りおよび書き込みリクエストのリクエストごとの支払い料金が用意されているため、使用した分だけ課金されます。詳細については、「DynamoDB On-demand mode」を参照してください。

   b. [プロビジョンドモード]では、アプリケーションに必要な 1 秒あたりの読み込みと書き込みの回数を指定します。Auto Scaling を使用すると、トラフィックの変更に応じて、テーブルのプロビジョンドキャパシティーを自動的に調整できます。詳細については、「DynamoDB Provisioned mode」を参照してください。

7. [クローン] を選択してクローン作成を開始します。
8. クローン作成のプロセスはバックグラウンドで実行されます。テーブルのクローン作成のステータスが変化すると、[Operation builder] タブに通知が表示されます。このステータスを表示する

には、[Operation builder] タブを選択し、矢印ボタンを選択します。矢印ボタンは、メニューサイドバーの下部にあるテーブルのクローン作成ステータスのウィジェットにあります。

CSV ファイルへのデータのエクスポート

クエリの結果を Operation Builder から CSV ファイルにエクスポートできます。これにより、スプレッドシートにデータをロードしたり、好みのプログラミング言語でデータ処理ができるようになります。

CSV へエクスポート

1. Operation Builder で、スキャンやクエリなどの任意の操作を実行します。

   > **Note**
   >
   > - 読み取り API オペレーションと PartiQL ステートメントの結果のみを CSV ファイルにエクスポートできます。トランザクション読み取りステートメントの結果はエクスポートできません。
   > - 現時点では、結果を 1 ページずつ CSV ファイルにエクスポートできます。結果のページが複数ある場合は、各ページを個別にエクスポートする必要があります。

2. 結果からエクスポートする項目を選択します。
3. [アクション] ドロップダウンで、[CSV としてエクスポート] を選択します。
4. CSV ファイルのファイル名と場所を選択し、[Save] (保存) を選択します。

NoSQL Workbench のサンプルデータモデル

モデラーのホームページには、NoSQL Workbench に付属のサンプルモデルが多数表示されます。このセクションでは、これらのモデルとその可能性のある用途について説明します。

トピック

- 従業員データモデル
- ディスカッションフォーラムデータモデル
- ミュージックライブラリデータモデル
- スキーリゾートデータモデル

- クレジットカードオファーデータモデル
- ブックマークデータモデル

従業員データモデル

このデータモデルは、入門モデルです。これは、一意のエイリアス、名、姓、職名、マネージャー、およびスキルなどの従業員の基本的な詳細を表します。

このデータモデルは、複数のスキルを持つなどの複雑な属性を扱うなど、いくつかの技術を示しています。このモデルは、セカンダリインデックス DirectReports によって達成されたマネージャーとその部下の従業員による 1 対多リレーションシップの例でもあります。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- `Employee` というテーブルによって容易になる、従業員のログインエイリアスを使用した従業員レコードの取得。
- `Name` という Employee テーブルのグローバルセカンダリインデックスによって容易になる、名前による従業員の検索。
- `DirectReports` という従業員のテーブルのグローバルセカンダリインデックスによって容易になる、マネージャーのログインエイリアスを使用したマネージャーの直接レポートの取得。

ディスカッションフォーラムデータモデル

このデータモデルは、ディスカッションフォーラムを表します。このモデルを使用すると、お客様は開発者コミュニティとやり取りし、質問をしたり、他のお客さま顧客の投稿に応答することができます。各 AWS サービスには専用フォーラムがあります。フォーラムにメッセージを投稿することで、誰でも新しいディスカッションスレッドを開始できます。各スレッドは任意の数の返信を受け取ります。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- `Forum` というテーブルによって容易になる、フォーラムの名前を使用したフォーラムのレコードの取得。
- `Thread` というテーブルによって容易になる、フォーラムの特定のスレッドまたはすべてのスレッドの取得。
- `PostedBy-Message-Index` という Reply テーブルのグローバルセカンダリインデックスによって容易になる、投稿ユーザーの E メールアドレスを使用した返信の検索。

ミュージックライブラリデータモデル

このデータモデルは、大量の楽曲コレクションを持ち、ダウンロード数の多い楽曲をほぼリアルタイムで紹介するミュージックライブラリです。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- Songs というテーブルによって容易になる曲レコードの取得。
- Songs というテーブルによって容易になる、曲の特定のダウンロードレコードまたはすべてのダウンロードレコードの取得。
- Song というテーブルによって容易になる、曲の特定の月別ダウンロード数レコードまたはすべての月のダウンロード数レコードの取得
- Songs というテーブルによって容易になる曲のすべてのレコード (曲のレコード、ダウンロードレコード、月別ダウンロード数レコードを含む) の取得。
- DownloadsByMonth という Songs テーブルのグローバルセカンダリインデックスによって容易になる、ほとんどのダウンロード曲の検索。

スキーリゾートデータモデル

このデータモデルは、毎日収集される各スキーリフトについて大量のデータを収集するスキーリゾートを表します。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- SkiLifts というテーブルによって容易になる、指定されたスキーリフトやリゾート全体の動的および静的なすべてのデータの取得。
- SkiLifts というテーブルによって容易になる、特定の日付のスキーリフトまたはリゾート全体のすべての動的データ (一意のリフトライダー、積雪、雪崩の危険、リフト状況を含む) の取得。
- SkiLifts というテーブルによって容易になる、特定のスキーリフトのすべての静的データ (リフトが経験豊富なライダーのみを対象とする場合、リフトが上昇する垂直フィート、リフトのライディング時間を含む) の取得。
- SkiLiftsByRiders という SkiLifts テーブルのグローバルセカンダリインデックスによって容易になる、一意のライダーの総数によってソートされる特定のスキーリフトまたはリゾート全体について記録されたデータの日付の取得。

クレジットカードオファーデータモデル

このデータモデルは、クレジットカードオファーアプリケーションで使用されます。

クレジットカードプロバイダは、時間の経過とともにオファーを生成します。これらのオファーには、手数料なしの残高振替、与信限度額の増加、金利の低減、キャッシュバック、航空会社マイルが含まれます。お客様がこれらのオファーを承認または拒否すると、それに応じてそれぞれのオファーステータスが更新されます。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- メインテーブルによって容易になる、AccountId を使用した取引先レコードの取得。
- セカンダリインデックス AccountIndex によって容易になる、いくつかの予測項目を持つすべてのアカウントの取得。
- メインテーブルによって容易になる、AccountId を使用したアカウントおよびそれらのアカウントに関連付けられているすべてのオファーレコードの取得。
- メインテーブルによって容易になる、AccountId および OfferId を使用した取引先およびそれらの取引先に関連付けられた特定のオファーレコードの取得。
- セカンダリインデックス ACCEPTED/DECLINED によって容易になる、OfferType、AccountId、および OfferType を使用した取引先に関連付けられた特定の Status のすべての GSI1 オファーレコードの取得
- メインテーブルによって容易になる、OfferId を使用したオファーおよび関連するオファー項目レコードの取得。

ブックマークデータモデル

このデータモデルは、お客様のストアブックマークに使用されます。

お客様は多くのブックマークを持つことができ、ブックマークは多くのお客様に属することができます。このデータモデルは、多対多の関係を表します。

このデータモデルによって容易になるアクセスパターンは次のとおりです。

- customerId による単一のクエリで、ブックマークだけでなくお客様データも返せるようになりました。
- クエリ ByEmail インデックスは、E メールアドレスごとにお客様データを返します。ブックマークは、このインデックスによって取得されないことに注意してください。

- クエリ ByUrl インデックスは、URL ごとにブックマークデータを取得します。同じ URL を複数のお客様がブックマークできるため、このインデックスのソートキーとして customerId があることに注意してください。
- クエリ ByCustomerFolder インデックスは、各お客様のフォルダごとにブックマークを取得します。

NoSQL Workbench のリリース履歴

NoSQL Workbench クライアントツールの各リリースにおける重要な変更点を次の表に示します。

バージョン	変更	説明	日付
3.20.2	バグ修正	DynamoDB local が起動に失敗するかキャンセルされた場合のエラー処理を改善し、DynamoDB アクセスパターンのデータモデラーを修正しました。	2026 年 4 月 6 日
3.20.1	メンテナンスリリース	安定性とリリース設定の修正。	2026 年 2 月 17 日
3.20.0	DynamoDB 用の新しいデータモデラー	DynamoDB 用のデータモデラーのユーザーエクスペリエンスが更新されました。DynamoDB 用のデータモデラーがアクセスパターンをサポートするようになりました。	2026 年 2 月 16 日
3.13.5	デフォルトのテーブル設定のキャパシテ	デフォルト設定でテーブルを作成すると、DynamoDB はプ	2025 年 2 月 24 日

バージョン	変更	説明	日付
	ィモードがオンデマンドに変更	ロビジョニングされたキャパシティモードの代わりにオンデマンドキャパシティモードを使用するテーブルを作成します。	
3.13.0	NoSQL Workbench オペレーションビルダーの改善点	NoSQL Workbench には、ダークモードのネイティブサポートが含まれるようになりました。オペレーションビルダーのテーブルと項目のオペレーションを改善しました。項目の結果とオペレーションビルダーのリクエスト情報は JSON 形式で利用できます。	2024 年 4 月 24 日

バージョン	変更	説明	日付
3.12.0	NoSQL Workbench によるテーブルのクローン作成と消費されたキャパシティの返却	DynamoDB Local と DynamoDB ウェブサービスアカウント間、または DynamoDB ウェブサービスアカウント間でテーブルをクローンして、開発の反復を迅速に行えるようになりました。Operation builder を使用してオペレーションを実行した後に消費された RCU または WCU を表示します。CSV ファイルからのインポート時にデータが上書きされる問題を修正しました。	2024 年 2 月 26 日
3.11.0	DynamoDB Local の機能強化	ビルトイン DynamoDB Local インスタンスを起動するときにポートを指定できるようになりました。NoSQL Workbench を管理者権限なしで Windows にインストールできるようになりました。データモデルテンプレートを更新しました。	2024 年 1 月 17 日

バージョン	変更	説明	日付
3.10.0	Apple シリコンのネイティブサポート	NoSQL Workbench で、Apple シリコンを搭載した Mac をネイティブにサポートするようになりました。Number 型の属性にサンプルデータ生成形式を設定できるようになりました。	2023 年 12 月 5 日
3.9.0	データモデラーの改善	Visualizer では、既存のテーブルを上書きするオプションを使用して、データモデルを DynamoDB local にコミットできるようになりました。	2023 年 11 月 3 日
3.8.0	サンプルデータ生成	NoSQL Workbench は DynamoDB データ型のサンプルデータの生成をサポートするようになりました。	2023 年 9 月 25 日
3.6.0	オペレーションビルダーの改善	オペレーションビルダーの接続管理が改善されました。データモデラーの属性名は、データを削除せずに変更できるようになりました。その他のバグ修正。	2023 年 4 月 11 日

バージョン	変更	説明	日付
3.5.0	新しい AWS リージョン向けのサポート	NoSQL Workbench は、ap-south-2、ap-southeast-3、ap-southeast-4、eu-central-2、eu-south-2、me-central-1、および me-west-1 リージョンのサポートを開始しました。	2023 年 2 月 23 日
3.4.0	DynamoDB local のサポート	NoSQL Workbench は、インストールプロセスの一部として、DynamoDB local のインストールをサポートするようになりました。	2022 年 12 月 6 日
3.3.0	コントロールプレーンオペレーションのサポート	Operation Builder でコントロールプレーンオペレーションがサポートされるようになりました。	2022 年 6 月 1 日
3.2.0	CSV のインポートとエクスポート	ビジュアライザーツールの CSV ファイルからサンプルデータをインポートできるようになりました。また、Operation Builder クエリの結果を CSV ファイルにエクスポートすることもできます。	2021 年 10 月 11 日

バージョン	変更	説明	日付
3.1.0	オペレーションの保存	NoSQL Workbench のオペレーションビルダーで、後で使用するためにオペレーションを保存することがサポートされるようになりました。	2021 年 7 月 12 日
3.0.0	容量設定と CloudFormation のインポート/エクスポート	Amazon DynamoDB 用の NoSQL Workbench は、テーブルの読み込み/書き込み容量モードの指定をサポートするようになり、CloudFormation 形式でデータモデルをインポートおよびエクスポートできるようになりました。	2021 年 4 月 21 日
2.2.0	PartiQL のサポート	Amazon DynamoDB 用の NoSQL Workbench が、DynamoDB 用 PartiQL ステートメントを構築するためのサポートを追加します。	2020 年 12 月 4 日

バージョン	変更	説明	日付
1.1.0	Linux のサポート。	Amazon DynamoDB 用の NoSQL Workbench は Linux—Ubuntu、Fedora および Debian でサポートされています。	2020 年 5 月 4 日
1.0.0	NoSQL Workbench for Amazon DynamoDB – GA。	Amazon DynamoDB 用 NoSQL Workbench の一般提供が開始されました。	2020 年 3 月 2 日
0.4.1	IAM ロールと一時的なセキュリティ認証情報のサポート。	Amazon DynamoDB 用の NoSQL Workbench が、AWS Identity and Access Management (IAM) ロールと一時的なセキュリティ認証情報のサポートを追加。	2019 年 12 月 19 日
0.3.1	DynamoDB local (ダウンロード可能バージョン) のサポート。	NoSQL Workbench では、DynamoDB local (ダウンロード可能バージョン) に接続して、DynamoDB テーブルを設計、作成、クエリ、管理できるようになりました。	2019年11月8日

バージョン	変更	説明	日付
0.2.1	NoSQL Workbench プレビュー版がリリースされました。	これは DynamoDB 用 NoSQL Workbench の初版リリースです。NoSQL Workbench を使用して、DynamoDB テーブルを設計、作成、クエリ、および管理します。	2019 年 9 月 16 日

DynamoDB のバックアップと復元

DynamoDB では、災害イベントから DynamoDB データを保護するためのオンデマンドバックアップとポイントインタイムリカバリ (PITR) バックアップ、さらに長期保存のためのデータアーカイブを利用できます。本番アプリケーションのパフォーマンスや可用性に影響を与えることなく、テーブル内の数メガバイトから数百テラバイトまでのデータをバックアップできます。すべてのバックアップは自動的に暗号化およびカタログ化され、簡単に検出できます。

オンデマンドバックアップを使用すると、DynamoDB が保存および管理するテーブルのスナップショットバックアップを作成できます。バックアップのサイズと期間に基づいて料金が請求されます。オンデマンドバックアップを使用すると、DynamoDB テーブル全体を、バックアップ作成時の正確な状態に復元できます。

DynamoDB オンデマンドバックアップを作成および管理するためのオプションは 2 つあります。

- DynamoDB
- AWS Backup

DynamoDB オンデマンドバックアップ機能を使用し、テーブルの完全なバックアップを作成して、長期保存し、規制コンプライアンスの要件に合わせてアーカイブすることができます。テーブルのデータは、AWS マネジメントコンソール または単一の API コールを使用して、いつでもバックアップおよび復元できます。

ポイントインタイムリカバリ (PITR) バックアップは DynamoDB のフルマネージド機能であり、1 秒あたり最大 35 日間のリカバリポイントを提供します。継続的バックアップであるポイントインタイムリカバリを使用するには、DynamoDB テーブルでポイントインタイムリカバリ (PITR) を有効にします。テーブルで PITR が有効になっている間は、DynamoDB テーブルのサイズに応じて料金が請求されます。DynamoDB テーブルに対してポイントインタイムリカバリ (PITR) を有効にすると、データは継続的にバックアップされます。これにより、その時点での元のテーブルと正確に同じ状態の新しい DynamoDB テーブルが作成され、PITR リカバリ期間内の特定の時点にテーブルを復元できるようになります。

ポイントインタイムリカバリを使用することで、DynamoDB テーブルを、偶発的な書き込みや削除のオペレーションから保護できます。ポイントインタイムリカバリを有効化すれば、オンデマンドバックアップの作成、維持、スケジュールを心配する必要はありません。たとえば、テストスクリプトで、誤って本稼働環境の DynamoDB テーブルに書き込みを行ったとします。

ポイントインタイムリカバリでは、過去 35 日間の任意の時点にテーブルを復元できます。リカバリ期間は、1 ~ 35 日間の任意の値を設定できます。ポイントインタイムリカバリの有効化後、現在の時刻の 5 分前から設定したリカバリ期間までの任意の時点に復元できます。DynamoDB では、テーブルの増分バックアップが維持されます。

また、ポイントインタイムリカバリオペレーションが、パフォーマンスや API レイテンシーに影響を及ぼすことはありません。

DynamoDB テーブルをポイントインタイムに復元するには、AWS マネジメントコンソール、 AWS Command Line Interface (AWS CLI)、または DynamoDB API を使用します。ポイントインタイムリカバリプロセスは、新しいテーブルに復元されます。

使用可能な AWS リージョンおよび料金に関する詳細については、Amazon DynamoDB 料金表を参照してください。

> **Note**
>
> - DynamoDB バックアップでは、タグ付けと属性ベースのアクセスコントロール (ABAC) はサポートされていません。バックアップで ABAC を使用するには、AWS Backup を使用することをお勧めします。
> - タグは復元されたテーブルには保持されません。ポリシーでタグベースの条件を使用する前に、復元されたテーブルにタグを追加する必要があります。

次の動画では、バックアップと復元の概念を紹介し、ポイントインタイムリカバリについて詳しく説明します。

バックアップと復元

トピック

- DynamoDB のポイントインタイムバックアップ
- DynamoDB のオンデマンドバックアップおよび復元の使用
- Amazon DynamoDB のバックアップの請求について
- DynamoDB でテーブルを復元する
- DynamoDB での AWS Backup の使用
- Amazon DynamoDB ワークロードのディザスタリカバリ戦略の選択

DynamoDB のポイントインタイムバックアップ

Amazon DynamoDB ポイントインタイムリカバリ (PITR) は、DynamoDB テーブルデータの継続的バックアップを自動的に行います。ポイントインタイムリカバリ (PITR) バックアップは DynamoDB のフルマネージド機能であり、1 秒あたり最大 35 日間のリカバリポイントを提供します。ポイントインタイムリカバリを有効化すれば、オンデマンドバックアップの作成、維持、スケジュールを心配する必要はありません。このセクションでは、DynamoDB でのプロセスの動作の概要を説明します。

[開始する前に]

Amazon DynamoDB テーブルでポイントインタイムリカバリ (PITR) を有効にする前に、以下の点を考慮します。

- `RecoveryPeriodinDays` を設定すると、継続的バックアップを行う期間を短縮できます。デフォルトでは、`RecoveryPeriodinDays` は 35 です。ただし、1 から 35 までの任意の値に設定できます。料金はテーブルとローカルセカンダリインデックスのサイズに基づいているため、`RecoveryPeriodinDays` を短縮しても、PITR の料金には影響しません。
- ポイントインタイムリカバリを無効にし、後にテーブル上で再度有効にした場合は、そのテーブルで復元可能な開始時間をリセットします。そのため、`LatestRestorableDateTime` を使用して、すぐにそのテーブルを復元できます。
- ポイントインタイムリカバリは、グローバルテーブルのローカルレプリカごとに有効にすることができます。テーブルを復元すると、バックアップは、グローバルテーブルに含まれていない独立したテーブルに復元されます。グローバルテーブルバージョン 2019.11.21 (現行) を使用している場合、復元されたテーブルから新しいグローバルテーブルを作成できます。詳細については、「DynamoDB グローバルテーブルの仕組み」を参照してください。
- また、送信元テーブルが存在する別のリージョンに復元済みテーブルが作成されるように、AWS リージョン全体で DynamoDB テーブルデータを復元することもできます。AWS 商用リージョン、AWS 中国リージョン、および AWS GovCloud (米国) リージョン間でクロスリージョン復元を実行できます。送信元リージョンから転送したデータと、送信先リージョンの新しいテーブルの復元に対してのみ料金が発生します。
- AWS CloudTrailポイントインタイムリカバリのコンソールと API のアクションはすべて、 に記録され、ログ記録、継続的モニタリング、監査に使用されます。詳細については、「AWS CloudTrail を使用して DynamoDB オペレーションをログに記録する」を参照してください。

トピック

- DynamoDB でポイントインタイムリカバリを有効にする

DynamoDB でポイントインタイムリカバリを有効にする

Amazon DynamoDB ポイントインタイムリカバリ (PITR) は、DynamoDB テーブルデータを自動バックアップする機能です。このセクションでは、DynamoDB でのプロセスの動作の概要を説明します。

> **Note**
>
> DynamoDB は、テーブルデータやローカルセカンダリインデックスなど、各 DynamoDB テーブルのサイズに基づいて PITR の料金を請求します。復旧期間を (35 日から 1 日などに) 変更しても、料金は削減されません。選択した復旧期間に関係なく、コストは変わりません。設定した最長リカバリ期間は、PITR をオンにした場合に請求される料金には影響しません。DynamoDB は、バックアップ料金を決定するために、PITR がオンになっているテーブルのサイズを継続的にモニタリングします。各テーブルの PITR をオフにするまでは、PITR の使用に対して料金が請求されます。

トピック

- ポイントインタイムリカバリの有効化
- PITR を有効にする (コンソール)
- PITR を有効にする (AWS CLI)
- PITR を有効にする (CloudFormation)
- PITR を有効にする (API)
- リカバリ期間
- PITR を編集する
- PITR が有効になっているテーブルを削除する

ポイントインタイムリカバリの有効化

ポイントインタイムリカバリは、AWS マネジメントコンソール、AWS Command Line Interface (AWS CLI)、または DynamoDB API を使用して有効にできます。有効化されると、ポイントインタイムリカバリは、明示的に無効化するまで、連続バックアップを行います。

ポイントインタイムリカバリを有効にしたら、EarliestRestorableDateTime および LatestRestorableDateTime 内の任意の時点に復元できます。LatestRestorableDateTime は、通常、現在時刻の 5 分前です。詳細については、「DynamoDB テーブルを特定の時点に復元」を参照してください。

> **Note**
>
> ポイントインタイムリカバリプロセスは、必ず新しいテーブルに復元されます。

PITR を有効にする (コンソール)

DynamoDB コンソールを使用して PITR を有効にするには

1. DynamoDB コンソールに移動します。
2. 左のナビゲーションから [テーブル] を選択し、DynamoDB テーブルを選択します。
3. [バックアップ] タブの [ポイントインタイムリカバリ] オプションで、[編集] を選択します。
4. [ポイントインタイムリカバリを有効にする] をクリックします。
5. バックアップリカバリ期間では、1 ~ 35 の値を選択します。これが、継続的バックアップが復元可能な最長期間を示します。

PITR を有効にする (AWS CLI)

> **Note**
>
> AWS CLI コマンドの実行中にエラーが発生した場合は、「AWS CLI エラーのトラブルシューティング」を参照してください。最新の AWS CLI バージョンを使用していることを確認します。

point-in-time-recovery-specification 設定をオンにして update-continuous-backups コマンドを実行します。

```
aws dynamodb update-continuous-backups \
--table-name Music \
--point-in-time-recovery-specification
 PointInTimeRecoveryEnabled=true,RecoveryPeriodInDays=35
```

PITR を有効にする (CloudFormation)

`PointInTimeRecoverySpecification` プロパティをオンにして AWS::DynamoDB ::Table リソースを使用します。

```
Resources:
  iotCatalog:
    Type: AWS::DynamoDB::Table
      Properties:
      ...
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true
        RecoveryPeriodInDays: 35
```

リクエストの構文例:

```
{
   "PointInTimeRecoverySpecification": {
      "PointInTimeRecoveryEnabled": boolean,
      "RecoveryPeriodInDays: number
   },
   "TableName": "string"
}
```

PITR を有効にする (API)

`PointInTimeRecoverySpecification` パラメータをオンにして UpdateContinuousBackups API オペレーションを実行します。

リクエストの構文例:

```
{
   "PointInTimeRecoverySpecification": {
      "PointInTimeRecoveryEnabled": boolean,
      "RecoveryPeriodInDays" : number
   },
   "TableName": "string"
}
```

レスポンスの構文例 :

```
{
```

```
   "ContinuousBackupsDescription": {
      "ContinuousBackupsStatus": "string",
      "PointInTimeRecoveryDescription": {
         "PointInTimeRecoveryStatus": "string",
         "EarliestRestorableDateTime": number,
         "RecoveryPeriodInDays": number,
         "LatestRestorableDateTime": number
      }
   }
}
```

> **Note**
>
> ContinuousBackupsStatus は常に DynamoDB テーブルに対して ENABLED であり、無効にすることはできません。ENABLED という値は、ポイントインタイムリカバリが有効になっていることを意味するものではありません。テーブルを特定の時点に復元できるかどうかは、PointInTimeRecoveryStatus によって個別に制御されます。これは、PITR を有効にするまでデフォルトで DISABLED になります。したがって、新しく作成されたテーブルは、ContinuousBackupsStatus を ENABLED、PointInTimeRecoveryStatus を DISABLED と報告します。

Python:

```
import boto3

dynamodb = boto3.client('dynamodb')

response = dynamodb.update_continuous_backups(
    TableName=<table_name>,
    PointInTimeRecoverySpecification={
        'PointInTimeRecoveryEnabled': True,
        'RecoveryPeriodInDays': 35
    }
)
```

リカバリ期間

継続的バックアップのリカバリ期間は、1 ~ 35 日間の任意の数に設定できます。この RecoveryPeriodInDays により、継続的バックアップが維持される期間が決まります。例えば、

この値を 30 日に設定すると、テーブルを過去 30 日間の任意の時点に復元することしかできなくなります。

> **Note**
>
> DynamoDB は、テーブルデータやローカルセカンダリインデックスなど、各 DynamoDB テーブルのサイズに基づいて PITR の料金を請求します。設定した最長リカバリ期間は、PITR をオンにした場合に請求される料金には影響しません。料金の詳細については、「DynamoDB 料金」を参照してください。

PITR を編集する

テーブルの PITR 設定を編集すると、リカバリ期間を変更できます。リカバリ期間を変更して、以前の設定値より高い値に引き上げても、`EarliestRestorePoint` は直ちには変更されません。復旧期間はローリングウィンドウ方式であるため、DynamoDB は新しく引き上げられた期間に達するまで自動バックアップを引き続き実行します。リカバリ期間を変更して、以前に設定した値よりも低い値に引き下げると、`EarliestRestorePoint` はリカバリ期間に合わせて直ちに低減し、新しい設定値の範囲外の継続的なバックアップは復元できなくなります。

PITR が有効になっているテーブルを削除する

ポイントインタイムリカバリが有効になっているテーブルを削除すると、DynamoDB はシステムバックアップと呼ばれるバックアップスナップショットを自動的に作成し、それを 35 日間保持します (追加コストは発生しません)。システムバックアップを使用して、削除したテーブルを削除前の状態に復元できます。システムバックアップはすべて、*`table-name`* `$DeletedTableBackup` という標準的な命名規則に従います。

> **Note**
>
> ポイントインタイムリカバリが有効になっているテーブルが削除されると、システムバックアップを使用して、削除されたテーブルを特定の時点、つまり削除直前の時点に復元できます。システムバックアップはテーブルの削除時に作成され、テーブルが削除される直前のテーブルのスナップショットです。

DynamoDB のオンデマンドバックアップおよび復元の使用

Amazon DynamoDB では、スタンドアロンのオンデマンドバックアップと復元機能をサポートしています。これらの機能は、AWS Backup を使用するかどうかに関係なく利用できます。

DynamoDB オンデマンドバックアップ機能を使用し、テーブルの完全なバックアップを作成して、規制コンプライアンスの要件を満たすために長期間の保存とアーカイブを行うことができます。テーブルのデータは、AWS マネジメントコンソールでワンクリックするか、1 回の API 呼び出しで、バックアップおよび復元することができます。バックアップおよび復元アクションを実行しても、テーブルのパフォーマンスや可用性に影響を及ぼすことはありません。

コンソール、AWS コマンドラインインターフェース (AWS CLI)、または DynamoDB API を使用してテーブルバックアップを作成できます。詳細については、「DynamoDB テーブルのバックアップ」を参照してください。

バックアップからテーブル復元する方法については、「バックアップからの DynamoDB テーブルの復元」を参照してください。

DynamoDB を使用した DynamoDB テーブルのバックアップと復元の仕組み

DynamoDB オンデマンドバックアップ機能を使用して、Amazon DynamoDB テーブルの完全バックアップを作成できます。この機能は、AWS Backup とは関係なく利用できます。このセクションでは、DynamoDB のバックアップおよび復元プロセス中に発生することの概要について説明します。

バックアップ

DynamoDB を使用してオンデマンドバックアップを作成すると、リクエストのタイムマーカーがカタログ化されます。このバックアップは、前回のテーブル全体のスナップショットへのリクエスト時間までにすべての変更を適用して、非同期的に作成されます。DynamoDB のバックアップリクエストは瞬時に処理され、数分以内に復元できる状態になります。

> **Note**
>
> オンデマンドバックアップを作成する度に、テーブルデータ全体がバックアップされます。オンデマンドバックアップの実行可能数に制限はありません。

DynamoDB では、テーブルのプロビジョンされたスループットを消費することなく、バックアップすることができます。

DynamoDB バックアップでは、項目間の因果整合性は保証されません。ただし、バックアップの更新間のスキューは、通常 1 秒未満です。

バックアップ進行中は、以下を行うことができません。

- バックアップオペレーションの一時停止またはキャンセル。
- バックアップのソーステーブルの削除。
- テーブルのバックアップ中におけるテーブルのバックアップの無効化。

スケジューリングスクリプトとクリーンアップジョブを作成したくない場合は、AWS Backup を使用して DynamoDB テーブルのスケジュールと保存ポリシーを含むバックアップ計画を作成できます。AWS Backup はバックアップを実行し、有効期限が切れると削除します。詳細については、「AWS Backup デベロッパーガイド」を参照してください。

AWS Backup に加えて、AWS Lambda 関数を使用して、定期的または以降のバックアップをスケジュールできます。詳細については、ブログ記事「Amazon DynamoDB のオンデマンドバックアップをスケジュールするサーバーレスソリューション」を参照してください。

コンソールを使用している場合、AWS Backup を使用して作成されたバックアップは、[Backup type (バックアップタイプ)] が `AWS_BACKUP` に設定された状態で [Backups (バックアップ)] タブに一覧表示されます。

> **Note**
>
> DynamoDB コンソールを使用して、[Backup type (バックアップタイプ)] が `AWS_BACKUP` でマークされたバックアップを削除することはできません。これらのバックアップを管理するには、AWS Backup コンソールを使用します。

バックアップを実行する方法については、「DynamoDB テーブルのバックアップ」を参照してください。

復元

テーブルは、テーブルのプロビジョニングされたスループットを消費することなく復元します。DynamoDB バックアップからテーブル全体を復元することも、送信先テーブルの設定を構成することもできます。復元を実行するときに、次のテーブル設定を変更できます。

- グローバルセカンダリインデックス (GSI)
- ローカルセカンダリインデックス (LSI)
- 請求モード
- プロビジョニングされた読み込みおよび書き込みキャパシティ
- 暗号化設定

> **Important**
>
> テーブル全体を復元する場合、送信先テーブルはバックアップがリクエストされた時間に記録されたように、送信元テーブルと同じプロビジョニングされた読み込みキャパシティユニットおよびプロビジョニングされた書き込みキャパシティユニットを使用して設定されます。復元プロセスでは、ローカルセカンダリインデックスおよびグローバルセカンダリインデックスも復元されます。

また、バックアップが存在する別のリージョンに復元済みテーブルが作成されるように、AWS リージョン全体で DynamoDB テーブルデータを復元することもできます。AWS 商用リージョン、AWS 中国リージョン、および AWS GovCloud (米国) リージョン間でクロスリージョン復元を実行できます。送信元リージョンから転送したデータと、送信先リージョンの新しいテーブルの復元に対してのみ料金が発生します。

新しく復元されるテーブルで、一部またはすべてのセカンダリインデックスの作成を除外すると、復元はより高速でコスト効率が高くなります。

以下は、復元されたテーブルで手動で設定する必要があります。

- Auto Scaling ポリシー
- AWS Identity and Access Management (IAM) ポリシー
- Amazon CloudWatch メトリクスおよびアラーム
- タグ
- ストリーム設定

- 有効期限 (TTL) 設定
- 削除保護設定
- ポイントインタイムリカバリ (PITR) 設定

テーブルデータ全体は、バックアップから新しいテーブルにのみ復元することができます。復元されたテーブルに書き込むことができるのは、アクティブになってからです。

> **Note**
>
> 復元オペレーション中は既存のテーブルを上書きすることはできません。

復元時間は、テーブルの構成 (テーブルのサイズ、基礎となるパーティションの数など) およびその他の関連する変数に直接関係しています。災害対策を計画する際のベストプラクティスは、平均復元完了時間を定期的に記録し、これらの時間が目標復旧時間全体にどのように影響するかを確認することです。

復元を実行する方法については、「バックアップからの DynamoDB テーブルの復元」を参照してください。

IAM ポリシーを使用してアクセスコントロールできます。詳細については、「DynamoDB バックアップおよび復元での IAM の使用」を参照してください。

バックアップおよび復元を行うコンソールと API のアクションはすべて、AWS CloudTrail にキャプチャおよび記録され、ログ記録、継続的モニタリング、監査に使用されます。

DynamoDB テーブルのバックアップ

このセクションでは、Amazon DynamoDB コンソールまたは AWS Command Line Interface を使用してテーブルをバックアップする方法について説明します。

テーブルバックアップの作成 (コンソール)

`MusicBackup` を使用して既存のテーブル `Music` のバックアップ AWS マネジメントコンソール を作成するには、以下のステップに従います。

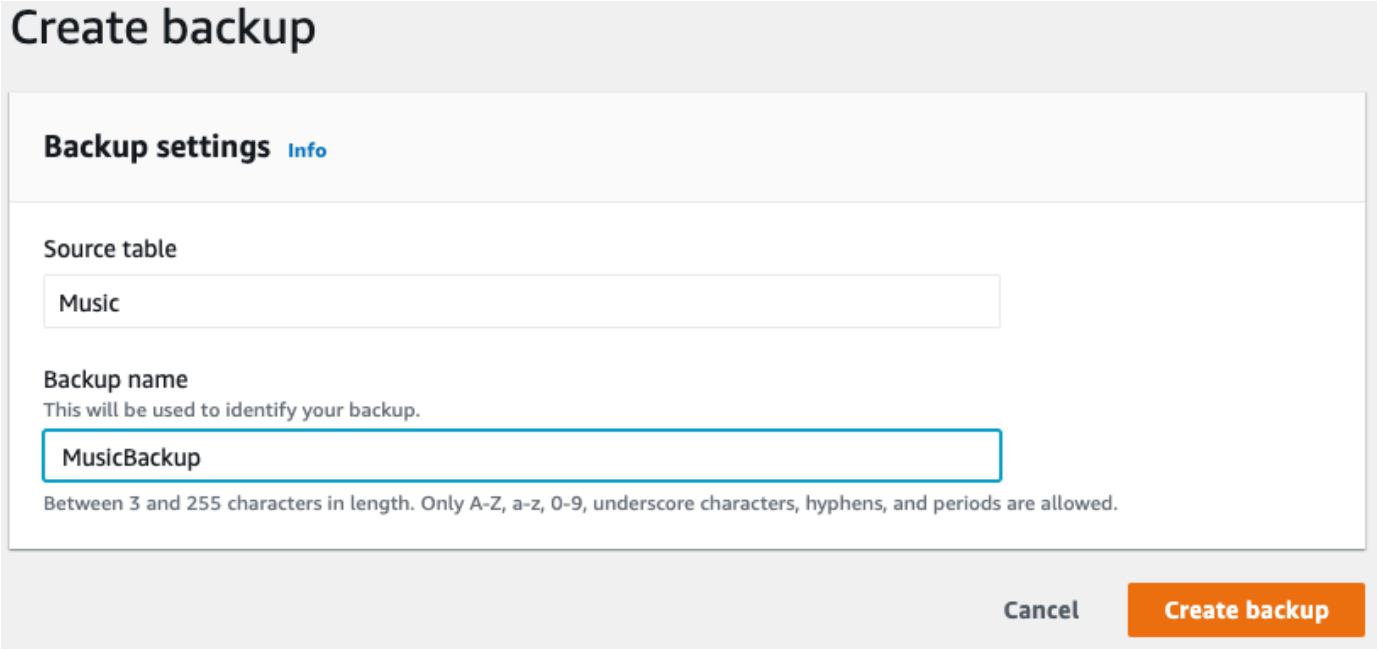
テーブルのバックアップを作成するには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。

2. バックアップを作成するには、次のいずれかを行います。

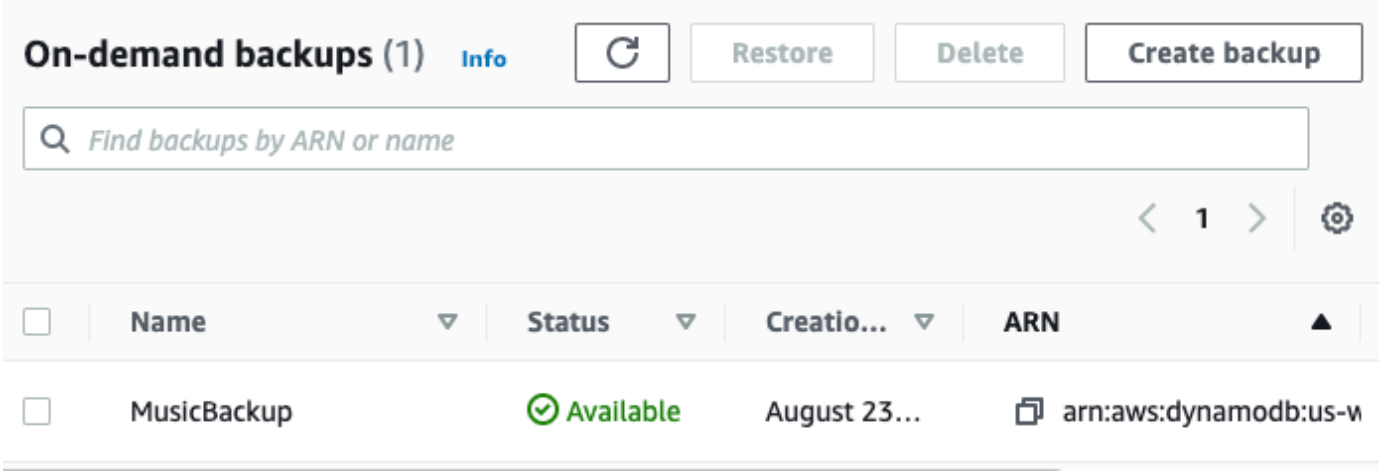
   - `Music` テーブルの [Backups] (バックアップ) タブで、[Create backup] (バックアップの作成) を選択します。
   - コンソールの左側のナビゲーションペインで、[Backups (バックアップ)] を選択します。続いて、[Create backup (バックアップを作成)] を選択します。

3. `Music` がテーブル名であることを確認し、バックアップ名として **`MusicBackup`** を入力します。次に、[バックアップを作成] を選択してバックアップを作成します。



> **Note**
>
> ナビゲーションペインの [Backups (バックアップ)] セクションでバックアップを作成した場合、テーブルは事前に選択されていません。バックアップ用のソーステーブルは、手動で選択する必要があります。

バックアップ作成中、バックアップステータスは [作成中] に設定されます。バックアップが完了すると、バックアップステータスは [利用可能] に変わります。



テーブルバックアップの作成 (AWS CLI)

Music を使用して既存のテーブル AWS CLI のバックアップを作成するには、以下のステップに従います。

テーブルのバックアップを作成するには

- MusicBackup テーブルのバックアップ (Music) を作成します。

```
aws dynamodb create-backup --table-name Music \
 --backup-name MusicBackup
```

バックアップ作成中、バックアップステータスは CREATING に設定されます。

```
{
    "BackupDetails": {
        "BackupName": "MusicBackup",
        "BackupArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc",
        "BackupStatus": "CREATING",
        "BackupCreationDateTime": 1489602797.149
    }
}
```

バックアップが完了すると、その BackupStatus が AVAILABLE に変更されます。これを確認するには、describe-backup コマンドを使用します。入力値 (backup-arn) は、前のステップの出力から、または list-backups コマンドを使用して取得できます。

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-b308cd7d
```

バックアップを追跡するには、list-backups コマンドを使用できます。これにより、CREATING ステータスまたは AVAILABLE ステータスのバックアップがすべて表示されます。

```
aws dynamodb list-backups
```

バックアップのソーステーブルに関する情報を確認するには、list-backups コマンドおよび describe-backup コマンドが便利です。

バックアップからの DynamoDB テーブルの復元

このセクションでは、 Amazon DynamoDB コンソールまたは AWS Command Line Interface (AWS CLI) を使用してバックアップからテーブルを復元する方法について説明します。

> **Note**
>
> AWS CLI を使用する場合は、最初に設定する必要があります。詳細については、「DynamoDB にアクセスする」を参照してください。

バックアップからのテーブルの復元 (コンソール)

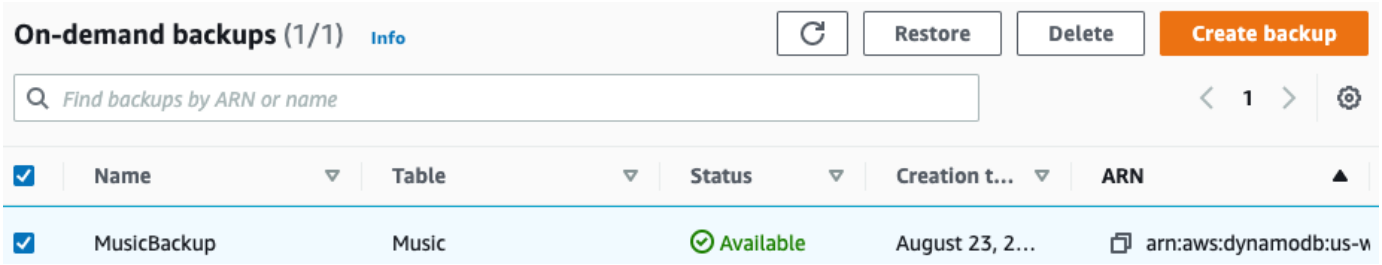
以下の手順では、Music チュートリアルで作成された MusicBackup ファイルを使用して DynamoDB テーブルのバックアップ を復元する方法について説明します。

> **Note**
>
> この手順では、Music が既に存在していないことを前提として、これを MusicBackup ファイルを使用してリストアする方法を示します。

バックアップからテーブルを復元するには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。

2. コンソールの左側のナビゲーションペインで、[Backups (バックアップ)] を選択します。
3. バックアップのリストで、[MusicBackup] を選択します。



4. [復元] を選択します。
5. 新しいテーブルの名前を **Music** と入力します。バックアップ名およびその他のバックアップの詳細を確認します。続いて、[Restore table (テーブルの復元)] を選択して復元プロセスを開始します。

> **Note**
>
> テーブルは、同じ AWS リージョンまたはバックアップが存在する別のリージョンに復元できます。新しく復元されるテーブルで、セカンダリインデックスの作成を除外することもできます。また、別の暗号化モードを指定することもできます。
> バックアップから復元されたテーブルは、常に DynamoDB 標準テーブルクラスを使用して作成されます。

Restore table from backup Info

Restoring a table from a backup will restore it as a new table.

Restore settings

Name of restored table

This name will identify your restored table.

Music

Between 3 and 255 characters in length. Only A–Z, a–z, 0–9, underscore characters, hyphens, and periods allowed.

Secondary indexes

- (●) Restore the entire table
  Your restored table will include all local and global secondary indexes.
- ( ) Restore the table without secondary indexes
  Your restored table will exclude all local and global secondary indexes. Restoring this way can be faster and more cost efficient.

Destination AWS Region

- (●) Same Region (Oregon)
  Restore the table to the same Region as the original table.
- ( ) Cross-Region
  Restore the table to a different Region for greater redundancy but with higher data transfer costs.

▼ Encryption at rest - *optional*

All user data stored in Amazon DynamoDB is fully encrypted at rest. By default, Amazon DynamoDB manages the encryption key, and you are not charged any fee for using it.

Encryption key management Info

- (●) Owned by Amazon DynamoDB
  The key is owned and managed by DynamoDB. You are not charged an additional fee for using this customer master key (CMK).
- ( ) AWS managed CMK
  The key is stored in your account and is managed by AWS Key Management Service (AWS KMS). AWS KMS charges apply.
- ( ) Stored in your account, and owned and managed by you
  Choose a key that is owned and managed by you, and stored in AWS KMS.

ⓘ The time it takes to restore a table from a backup can vary and is based on multiple variables. After your table is restored from the backup, you might need to reapply configuration settings. Learn more

Cancel Restore

復元中のテーブルのステータスは、[Creating (作成中)] と表示されます。復元プロセスが終了すると、Music テーブルのステータスは [アクティブ] に変わります。

バックアップからのテーブルの復元 (AWS CLI)

AWS CLI チュートリアルで作成されている Music を使用して、MusicBackup で DynamoDB テーブルのバックアップ テーブルを復元するには、以下の手順を実行します。

バックアップからテーブルを復元するには

1. list-backups コマンドを使用して復元するバックアップを確認します。この例では MusicBackup を使用します。

```
aws dynamodb list-backups
```

バックアップの詳細をさらに取得するには、describe-backup コマンドを使用します。前のステップから入力の backup-arn を取得できます。

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d
```

2. バックアップからテーブルを復元します。この場合、MusicBackup は Music テーブルを同じ AWS リージョンに復元します。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d
```

3. カスタムテーブル設定で、バックアップからテーブルを復元します。この場合、MusicBackup は Music テーブルを復元し、復元したテーブルの暗号化モードを指定します。

> **Note**
>
> sse-specification-override パラメータは、sse-specification-override コマンドで使用される CreateTable パラメータと同じ値を使用します。詳細については、「DynamoDB での暗号化テーブルの管理」を参照してください。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

テーブルは、バックアップが存在する別の AWS リージョンに復元できます。

> **Note**
>
> - `sse-specification-override` パラメータは、クロスリージョン復元には必須ですが、送信元テーブルと同じリージョンに復元する場合はオプションです。
> - コマンドラインからクロスリージョン復元を実行する場合は、デフォルトの AWS リージョンを希望のリージョンに設定する必要があります。詳細については、「AWS Command Line Interface ユーザーガイド」の「コマンドラインオプション」を参照してください。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581080476474-e177ebe2 \
--sse-specification-override Enabled=true,SSEType=KMS
```

復元済みテーブルの請求モードとプロビジョニングされたスループットをオーバーライドできます。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d \
--billing-mode-override PAY_PER_REQUEST
```

復元済みテーブルで、一部またはすべてのセカンダリインデックスの作成を除外できます。

> **Note**
>
> 復元済みテーブルで、一部またはすべてのセカンダリインデックスの作成を除外すると、復元はより高速でコスト効率が高くなります。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01581081403719-db9c1f91 \
--global-secondary-index-override '[]' \
--sse-specification-override Enabled=true,SSEType=KMS
```

> **Note**
>
> 提供されたセカンダリインデックスは、既存のインデックスに一致します。復元時に新しいインデックスを作成することはできません。

さまざまなオーバーライドの組み合わせを使用できます。たとえば、次のように単一のグローバルセカンダリインデックスを使用すると同時に、プロビジョニングされたスループットを変更できます。

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:eu-west-1:123456789012:table/Music/
backup/01581082594992-303b6239 \
--billing-mode-override PROVISIONED \
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \
--global-secondary-index-override IndexName=singers-
index,KeySchema=["{AttributeName=SingerName,KeyType=HASH}"],Projection="{ProjectionType=KEYS_
 \
--sse-specification-override Enabled=true,SSEType=KMS
```

復元を確認するには、`describe-table` コマンドで `Music` テーブルを指定します。

```
aws dynamodb describe-table --table-name Music
```

バックアップから復元中のテーブルのステータスは、[Creating (作成中)] と表示されます。復元プロセスが終了すると、Music テーブルのステータスは [アクティブ] に変わります。

> **Important**
>
> 復元中は、IAM ロールのポリシーを変更または削除しないでください。行った場合、予期しない動作が発生する場合があります。たとえば、テーブルの復元中にテーブルの書き込み権限を削除したとします。この場合、基本となる RestoreTableFromBackup オペレーションを使用しても、復元されたデータをテーブルに書き込むことはできません。
> 復元オペレーションが完了したら、IAM ロールポリシーを変更または削除できます。
> 復元先のターゲットテーブルにアクセスするための送信元 IP の制限を含む IAM ポリシーでは、プリンシパルによって直接行われたリクエストにのみその制限が適用されるように、aws:ViaAWSService キーを false に設定する必要があります。そうしないと、復元はキャンセルされます。
> バックアップが AWS マネージドキー またはカスタマーマネージドキーで暗号化されている場合、復元中にキーを無効にしたり削除したりすると復元が失敗します。
> 復元操作が完了したら、復元されたテーブルの暗号化キーを変更し、古いキーを無効化または削除できます。

DynamoDB テーブルのバックアップの削除

このセクションでは、AWS マネジメントコンソール または AWS Command Line Interface (AWS CLI) を使用して Amazon DynamoDB テーブルバックアップを削除する方法について説明します。

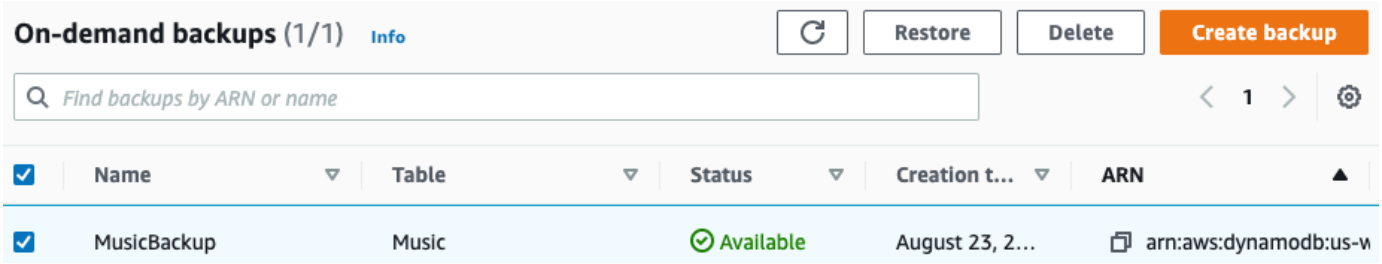
> **Note**
>
> AWS CLI を使用する場合は、最初に設定する必要があります。詳細については、「の使用 AWS CLI」を参照してください。

テーブルバックアップの削除 (コンソール)

次の手順では、コンソールを使用し、MusicBackup チュートリアルで作成した DynamoDB テーブルのバックアップ を削除する方法を示します。

バックアップを削除するには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups (バックアップ)] を選択します。
3. バックアップのリストで、[MusicBackup] を選択します。



4. [削除] を選択します。「**delete**」と入力してから、[削除] を選択して、バックアップの削除を確認します。

テーブルバックアップの削除 (AWS CLI)

次の例では、`Music` を使用して、既存のテーブルである AWS CLI テーブルのバックアップを削除します。

```
aws dynamodb delete-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc
```

DynamoDB バックアップおよび復元での IAM の使用

AWS Identity and Access Management (IAM) を使用すれば、一部リソースの Amazon DynamoDB バックアップアクションと復元アクションを制限することができます。`CreateBackup` および `RestoreTableFromBackup` API はテーブルごとにオペレーションを行います。

DynamoDB での IAM ポリシーの詳細な使用については、「DynamoDB のアイデンティティベースのポリシー」を参照してください。

次に、DynamoDB で特定のバックアップ機能と復元機能を設定するために使用できる IAM ポリシーの例を示します。

例 1: CreateBackup および RestoreTableFromBackup アクションを許可する

以下の IAM ポリシーは、すべてのテーブルで `CreateBackup` および `RestoreTableFromBackup` DynamoDB アクションを適用する許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateBackup",
                "dynamodb:RestoreTableFromBackup",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "*"
        }
    ]
}
```

Important

DynamoDB RestoreTableFromBackup アクセス許可はソースバックアップに必要であり、復元機能にはターゲットテーブルに対する DynamoDB の読み取りおよび書き込みアクセス許可が必要です。
RestoreTableToPointInTime アクセス許可はソーステーブルに必要であり、復元機能にはターゲットテーブルに対する DynamoDB の読み取りおよび書き込みアクセス許可が必要です。

例 2: CreateBackup アクションを許可し、RestoreTableFromBackup アクションを拒否する

以下の IAM ポリシーは、CreateBackup アクションの許可を付与し、RestoreTableFromBackup アクションを拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateBackup"],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": ["dynamodb:RestoreTableFromBackup"],
            "Resource": "*"
        }

    ]
}
```

例 3: ListBackups アクションを許可し、CreateBackup および RestoreTableFromBackup アクションを拒否する

以下の IAM ポリシーは、ListBackups アクションの許可を付与し、CreateBackup アクションおよび RestoreTableFromBackup アクションを拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
```

```
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateBackup",
                "dynamodb:RestoreTableFromBackup"
            ],
            "Resource": "*"
        }

    ]
}
```

例 4: ListBackups を許可し、DeleteBackup を拒否する

以下の IAM ポリシーは、`ListBackups` アクションの許可を付与し、`DeleteBackup` アクションを拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": ["dynamodb:DeleteBackup"],
            "Resource": "*"
        }

    ]
}
```

例 5: すべてのリソースに対する RestoreTableFromBackup および DescribeBackup を許可し、特定のバックアップに対する DeleteBackup を拒否する

以下の IAM ポリシーは、`RestoreTableFromBackup` および `DescribeBackup` アクションの許可を付与し、特定のバックアップリソースに対する `DeleteBackup` アクションを拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeBackup",
                "dynamodb:RestoreTableFromBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:DeleteBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489173575360-b308cd7d"
        }
    ]
```

```
}
```

> **Important**
>
> DynamoDB RestoreTableFromBackup アクセス許可はソースバックアップに必要であり、復元機能にはターゲットテーブルに対する DynamoDB の読み取りおよび書き込みアクセス許可が必要です。
> RestoreTableToPointInTime アクセス許可はソーステーブルに必要であり、復元機能にはターゲットテーブルに対する DynamoDB の読み取りおよび書き込みアクセス許可が必要です。

例 6: 特定のテーブルに対する CreateBackup を許可する

以下の IAM ポリシーは、`Movies` テーブルに対する `CreateBackup` アクションの許可のみを付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateBackup"],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/Movies"
            ]
        }
    ]
}
```

例 7: ListBackups を許可する

以下の IAM ポリシーは、`ListBackups` アクションに対する許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:ListBackups"],
            "Resource": "*"
        }
    ]
}
```

> **Important**
>
> テーブル別に ListBackups アクションに対する許可を付与することはできません。

例 8: AWS Backup 機能へのアクセス許可

高度な機能を備えたバックアップを完了するための StartAwsBackupJob アクションと、そのバックアップを正常に復元するための dynamodb:RestoreTableFromAwsBackup アクションに対する API アクセス許可が必要になります。

次の IAM ポリシーでは、高度な機能と復元を使用してバックアップをトリガーするためのアクセス許可を AWS Backup に付与します。また、テーブルが暗号化されている場合、ポリシーは AWS KMS キーにアクセスする必要があります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DescribeQueryScanBooksTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:StartAwsBackupJob",
                "dynamodb:DescribeTable",
```

```
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Books"
        },
        {
            "Sid": "AllowRestoreFromAwsBackup",
            "Effect": "Allow",
            "Action": [
                "dynamodb:RestoreTableFromAwsBackup"
            ],
            "Resource": "*"
        }
    ]
}
```

例 9: 特定のソーステーブルの RestoreTableToPointInTime を拒否する

以下の IAM ポリシーは、特定のソーステーブルに対する `RestoreTableToPointInTime` アクションの許可を拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:RestoreTableToPointInTime"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music"
        }
    ]
}
```

例 10: 特定のソーステーブルのすべてのバックアップに対して RestoreTableFromBackup を拒否する

以下の IAM ポリシーは、特定のソーステーブルのすべてのバックアップに対する `RestoreTableToPointInTime` アクションの許可を拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:RestoreTableFromBackup"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/*"
        }
    ]
}
```

Amazon DynamoDB のバックアップの請求について

このガイドでは、DynamoDB のバックアップの請求の仕組みについて説明します。以降のセクションでは、全体のコストに影響を与えるさまざまな要素を詳述し、明確な説明と実用的な例を提供します。

DynamoDB では、災害イベントから DynamoDB データを保護するためのオンデマンドバックアップとポイントインタイムリカバリ (PITR) バックアップ、さらに長期保存のためのデータアーカイブを利用できます。

仕組み

DynamoDB のオンデマンドバックアップは毎月請求されます。月の特定の日にバックアップを取得した場合、そのバックアップに対する料金はその月の残りの日数に基づいて計算され、1 回分の料金として課金されます (例: 27 日にバックアップを作成した場合、その月の残り数日分の料金が計算され、27 日の 1 回分の料金として適用されます)。

以前に取得したバックアップを以降の月も保持する場合、そのバックアップに対する月額料金が 1 日に適用されます。月末より前にバックアップが削除された場合、料金は実際の使用量に基づいて調整されます。

例えば、7 月 27 日にバックアップを作成し、8 月中維持した場合、そのバックアップには以下の料金が発生します。

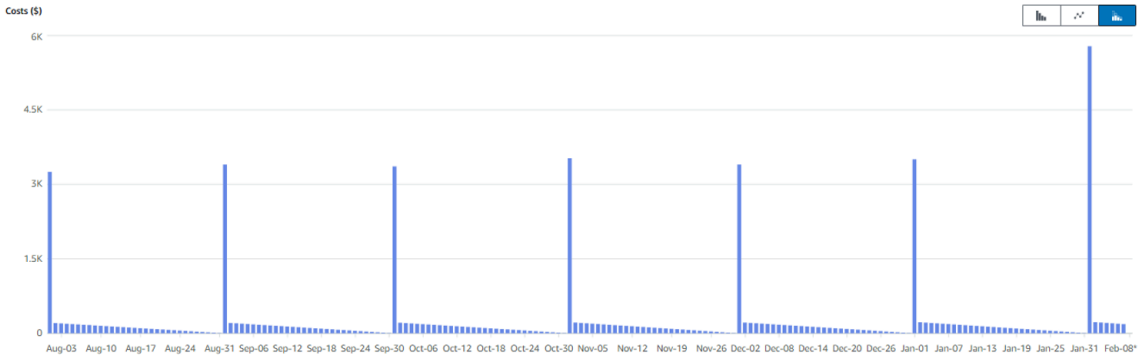
- 7 月の残りの日に対する 7 月 27 日の料金
- 8 月全体に対する 8 月 1 日の料金
- バックアップを維持する翌月以降の 1 日の料金

DynamoDB テーブルのバックアップが維持されている場合、DynamoDB (Region)-TimedBackupStorage-ByteHrs 使用量の費用のメトリクスが、毎月 1 日に異常に高いように見えることがあります。さらに、新しい月の開始時にこのメトリクスをチェックし、以前の請求サイクルと比較すると、使用量の大幅な急増が見られる場合があります。これは仕様です。毎月 1 日に、既存の DynamoDB バックアップには、月全体の使用料が適用されます。その月に削除された DynamoDB バックアップは、実際の使用量を反映するために、使用コストが日割り計算されます。その結果、1 日に適用された料金が 1 か月を通じて減少することがあります。これは、保持ポリシーによる有効期限の適用や、引き継がれたバックアップに対する手動削除が行われるためです。これは、以下のシナリオで詳しく説明します。

同様に、新しいバックアップを作成すると、月内により小さいスパイクが発生し、作成日から月の残りの分までの使用料が作成日に適用されます。

DynamoDB バックアップの請求例

以下は、月の初めに Cost Explorer に表示される可能性のある例です。



2 月 1 日が、前月に比べて使用量が急増しています。これが発生する理由を詳しく見てみましょう。

DynamoDB 料金ページから：

「毎月請求されるバックアップストレージの合計サイズは、DynamoDB テーブルのすべてのバックアップの合計です。DynamoDB は、オンデマンドバックアップのサイズを 1 か月間継続的にモニタリングして、バックアップ料金を決定します」

これが、請求書が毎月 1 日に一貫して使用量の急増を示している理由です。新しい月に入る既存のバックアップには、1 日に 1 か月分の料金が適用されます。つまり、DynamoDB バックアップが 300 個ある状態で新しい月に入ると、300 個すべてのバックアップについて、その月の 1 日に 1 か月分の料金が適用されます。

当月中に作成した新しいバックアップには、作成日から月末までの使用料が発生します。

バックアップを月半ばに削除した場合はどうなりますか?

考慮すべきシナリオをいくつか示します。

1. 前月からのバックアップを当月の 15 日に削除した場合、バックアップの使用料は、依然として初日に適用されますが、使用量は前月適用の 1 か月分ではなく、実際の使用量に調整されます。次の例で、詳しく説明します。
2. 月の途中でバックアップを作成すると、残月分の使用料が作成日に適用されます。ただし、月末前にバックアップを削除すると、使用料はバックアップがアクティブだった日数分だけに調整され、依然として作成日に適用されます。

なぜ当月の使用量が月の初日に前月よりも大幅に高く見えるのですか? また、バックアップを削除した場合はどうなりますか ?

この重要な 2 つの質問に答えるために、次の情報を使用してシナリオ例をセットアップします。

- 月の長さ: 30 日
- DynamoDB バックアップ頻度: 10/日、300/月
- DynamoDB バックアップ保持ポリシー: 30 日間
- DynamoDB バックアップあたりのコスト: 1 日 2 ドル、月 60 ドル
- 前月 1 日の合計 (`TimedBackupStorage-ByteHrs`、当月 1 日にチェック): 9,300 ドル
- 前月の合計 (`TimedBackupStorage-ByteHrs`): 18,600 ドル
- 現在の月初合計 (`TimedBackupStorage-ByteHrs`、1 日に確認): 18,000 ドル
- 月単位の DynamoDB 使用量の変更: なし

上記の情報を見ると、前月に 300 件のバックアップが作成され、30 日間保持するポリシーが設定されていることがわかります。新しい月の 1 日には、これらのバックアップのリカバリ期間はまだ終了していないため、すべてそのまま残ります。しかし、以下に示すように、日が経つにつれて最も古いバックアップセットが次第に削除されていくことになります。

DynamoDB テーブルバックアップの削除

新しい月	1 日目	2 日目	3 日目	4 日目	5 日目
前月のバックアップの持ち越し合計	300	290	280	270	260

- 1 日には、1 回のバックアップあたり月額 60 USD で 300 件のバックアップがあり、合計 18,000 USD の `TimedBackupStorage-ByteHrs` が適用されます。これは、前月の合計が 18,600 ドルであったのとは対照的です。
- 2 日に、これらのバックアップのうち 10 個が期限切れになり、削除されます。この場合、これらのバックアップに適用される料金は、想定使用量ではなく実際の使用量に調整されます。その結果、1日には 600 ドル (10 個のバックアップ × 30日) として適用されていた料金が、20 ドル (10 個のバックアップ × 1 日) に調整されます。
- 翌日には次の 10 個のバックアップが期限切れとなり削除され、その使用期間が 30 日から 2 日に短縮されることで、料金は 40 ドル (10個のバックアップ × 2日) に減額されます

日が経つごとに、前月よりも大きかった料金の急増が次第に縮小していくのがわかります。これを 1 か月全体に拡大すると、次のことが確認できます。

DynamoDB バックアップ料金 (月の初日) の進行状況

10 個のブロックで 300 個のバックアップ	1 日	10 日	20 日	30 日
ブロック 1	600 USD	20 USD	20 USD	20 USD
ブロック 2	600 USD	40 USD	40 USD	40 USD
ブロック 3	600 USD	60 USD	60 USD	60 USD
ブロック 4	600 USD	80 USD	80 USD	80 USD

10 個のブロックで 300 個のバックアップ	1 日	10 日	20 日	30 日
ブロック 5	600 USD	100 USD	100 USD	100 USD
ブロック 6	600 USD	120 USD	120 USD	120 USD
ブロック 7	600 USD	140 USD	140 USD	140 USD
ブロック 8	600 USD	160 USD	160 USD	160 USD
ブロック 9	600 USD	180 USD	180 USD	180 USD
ブロック 10	600 USD	600 USD	$200	$200
ブロック 11	600 USD	600 USD	220 USD	220 USD
ブロック 12	600 USD	600 USD	240 USD	240 USD
ブロック 13	600 USD	600 USD	260 USD	260 USD
ブロック 14	600 USD	600 USD	280 USD	280 USD
ブロック 15	600 USD	600 USD	$300	$300
ブロック 16	600 USD	600 USD	320 USD	320 USD
ブロック 17	600 USD	600 USD	340 USD	340 USD
ブロック 18	600 USD	600 USD	360 USD	360 USD
ブロック 19	600 USD	600 USD	380 USD	380 USD
ブロック 20	600 USD	600 USD	600 USD	400 USD
ブロック 21	600 USD	600 USD	600 USD	420 USD
ブロック 22	600 USD	600 USD	600 USD	440 USD
ブロック 23	600 USD	600 USD	600 USD	460 USD
ブロック 24	600 USD	600 USD	600 USD	480 USD

10 個のブロックで 300 個のバックアップ	1 日	10 日	20 日	30 日
ブロック 25	600 USD	600 USD	600 USD	500 USD
ブロック 26	600 USD	600 USD	600 USD	520 USD
ブロック 27	600 USD	600 USD	600 USD	540 USD
ブロック 28	600 USD	600 USD	600 USD	560 USD
ブロック 29	600 USD	600 USD	600 USD	580 USD
ブロック 30	600 USD	600 USD	600 USD	600 USD
月の初日の合計 (ドル)	18,000 USD	13,500 USD	10,400 USD	9,300 USD

毎日新しいブロックが削除されると、その使用量は 1 か月分の全期間ではなく、実際に存在していた日数に基づいて調整されます。その結果、1日に確認された料金は、月末までに当初の18,000 ドルから想定されていた 9,300 ドルにまで減少します。この金額に、月内に新しく作成されたバックアップ (上記と似た請求テーブルを持つが、パターンは逆) が加算されることで、前月の18,600 ドルに近い月額費用となります。

DynamoDB でテーブルを復元する

DynamoDB テーブルを PITR バックアップまたはオンデマンドバックアップから復元するには、AWS マネジメントコンソール、AWS コマンドラインインターフェイス (AWS CLI)、または DynamoDB API を使用できます。リカバリプロセスでは、新しい DynamoDB テーブルに復元します。

ポイントインタイムリカバリを使用したテーブルの復元

テーブルは、`EarliestRestoreableDateTime` までの任意の時点に復元できます。

> **Important**
>
> ポイントインタイムリカバリを無効にし、後でテーブルに対して再度有効にした場合は、テーブルを復元できる開始時刻がリセットされます。その結

果、LatestRestorableDateTime を使用したテーブルの即時復元しかできなくなります。

ポイントインタイムリカバリを使用して復元すると、DynamoDB は、選択した日時 (day:hour:minute:second) の状態を反映して、テーブルデータを新しいテーブルとして復元します。テーブルは、テーブルのプロビジョニングされたスループットを消費することなく復元します。ポイントインタイムリカバリを使用してテーブル全体を復元するか、送信先テーブルの設定を構成することができます。復元済みテーブルで次のテーブル設定を変更できます。

- グローバルセカンダリインデックス (GSI)
- ローカルセカンダリインデックス (LSI)
- 請求モード
- プロビジョニングされた読み込みおよび書き込みキャパシティ
- 暗号化設定

⚠ Important

テーブル全体の復元を実行すると、送信先テーブルには、バックアップが要求されたときに送信元テーブルが持っていたのと同じプロビジョニングされた読み込みキャパシティユニットと書き込みキャパシティユニットが設定されます。たとえば、テーブルのプロビジョニングされたスループットが 50 読み込みキャパシティーユニットおよび 50 書き込みキャパシティーユニットに最近下げられたとします。その後、このテーブルの状態を 3 週間前に復元し、その時点で、プロビジョニングされたスループットは 100 読み込みキャパシティーユニットと 100 書き込みキャパシティーユニットに設定されました。この場合、DynamoDB はテーブルデータをその時点からのプロビジョンされたスループット (100 読み込み容量単位および 100 書き込み容量単位) を利用してその時点のものに復元します。

また、AWS リージョン間で DynamoDB テーブルデータを復元し、ソーステーブルがあるリージョンとは別のリージョンに、復元したテーブルを作成することもできます。AWS 商用リージョン、AWS 中国リージョン、および AWS GovCloud (US) 間でクロスリージョンの復元を実行できます。送信元リージョンから転送したデータと、送信先リージョンの新しいテーブルの復元に対してのみ料金が発生します。

> **Note**
>
> 送信元または送信先リージョンがアジアパシフィック (香港) または中東 (バーレーン) の場合、クロスリージョンの復元はサポートされません。

復元済みテーブルで、一部またはすべてのインデックスの作成を除外すると、復元はより高速でコスト効率が高くなります。以下は、復元されたテーブルで手動で設定する必要があります。

- 自動スケーリングポリシー
- AWS Identity and Access Management ポリシー
- Amazon CloudWatch Events のメトリクスとアラーム
- CloudWatch Contributor Insights の設定
- タグ
- ストリーム設定
- 有効期限 (TTL) 設定
- 削除保護設定
- ポイントインタイムリカバリ設定

テーブルの復元にかかる時間は、複数の要因に基づきます。テーブルのサイズに常に関連するとは限りません。

DynamoDB テーブルを特定の時点に復元

Amazon DynamoDB ポイントインタイムリカバリ (PITR) は、DynamoDB テーブルデータを連続バックアップする機能です。テーブルをポイントインタイムに復元するには、DynamoDB コンソール、または AWS Command Line Interface (AWS CLI) を使用します。ポイントインタイムリカバリプロセスは、新しいテーブルに復元されます。

復元では常に新しいテーブルが作成されるため、復元の実行中もソーステーブルの読み取りと書き込みは完全に使用できます。復元にかかる時間は、DynamoDB が新しいテーブルで再構築するデータ量とセカンダリインデックスの数によって異なります。テーブルが大きいほど、またインデックスが多いほど時間がかかり、完了までの所要時間は保証されていません。復元を高速化するために、一部またはすべてのセカンダリインデックスを新しいテーブルから除外できます。

復元では、復元ポイントのソーステーブルの設定から新しいテーブルに、テーブルデータ、グローバルセカンダリインデックス、ローカルセカンダリインデックス、プロビジョニングされた読み込みお

よび書き込みキャパシティ、暗号化設定が再作成されます。復元ではこれらの設定が引き継がれないため、復元が完了したら、自動スケーリングポリシー、AWS Identity and Access Management ポリシー、CloudWatch メトリクスとアラーム、タグ、DynamoDB Streams 設定、有効期限設定、ポイントインタイムリカバリ設定を新しいテーブルに自身で設定する必要があります。

AWS CLI を使用する場合は、最初に設定する必要があります。詳細については、「DynamoDB にアクセスする」を参照してください。

トピック

- DynamoDB テーブルを特定の時点に復元 (コンソール)
- テーブルを特定の時点に復元する (AWS CLI)

DynamoDB テーブルを特定の時点に復元 (コンソール)

DynamoDB コンソールを使用して、`Music` という既存のテーブルを特定の時点に復元する方法を次の例に示します。

> **Note**
>
> この手順は、ポイントインタイムリカバリを有効にしていることを前提としています。`Music` テーブルで有効にするには、[バックアップ] タブの [ポイントインタイムリカバリ (PITR)] セクションで、[編集] を選択してから、[ポイントインタイムリカバリを有効にする] の横にあるチェックボックスをオンにします。

テーブルをポイントインタイムに復元するには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. テーブルのリストで、[`Music`] テーブルを選択します。
4. `Music` テーブルの [バックアップ] タブの [ポイントインタイムリカバリ (PITR)] セクションで、[復元] を選択します。
5. 新しいテーブルの名前に **`MusicMinutesAgo`** と入力します。

> **Note**
>
> テーブルは、同じ AWS リージョンまたは送信元テーブルが存在する別のリージョンに復元できます。復元済みテーブルで、セカンダリインデックスの作成を除外することもできます。また、別の暗号化モードを指定することもできます。

6. 復元可能な時刻を確認するには、[復元日時] を [最も早い] に設定します。次に、[復元] を選択して復元プロセスを開始します。

   復元中のテーブルのステータスは、[Restoring (復元中)] と表示されます。復元プロセスが終了すると、`MusicMinutesAgo` テーブルのステータスは [アクティブ] に変わります。

テーブルを特定の時点に復元する (AWS CLI)

AWS CLI を使用して、`Music` という既存のテーブルをポイントインタイムに復元する方法を次の手順に示します。

> **Note**
>
> この手順は、ポイントインタイムリカバリを有効にしていることを前提としています。この機能を `Music` テーブルに対して有効にするには、次のコマンドを実行します。
>
> ```
> aws dynamodb update-continuous-backups \
>     --table-name Music \
>     --point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
> ```

テーブルをポイントインタイムに復元するには

1. `Music` でポイントインタイムリカバリが有効になっていることを確認するには、`describe-continuous-backups` コマンドを使用します。

   ```
   aws dynamodb describe-continuous-backups \
       --table-name Music
   ```

   連続バックアップ (テーブル作成時は自動的に有効になる) とポイントインタイムリカバリは有効化されています。

```
{
    "ContinuousBackupsDescription": {
        "PointInTimeRecoveryDescription": {
            "PointInTimeRecoveryStatus": "ENABLED",
            "EarliestRestorableDateTime": 1519257118.0,
            "LatestRestorableDateTime": 1520018653.01
        },
        "ContinuousBackupsStatus": "ENABLED"
    }
}
```

2. テーブルをポイントインタイムに復元します。この場合、Music テーブルは同じ AWS リージョンの LatestRestorableDateTime (最大 5 分前) に復元されます。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time
```

> **Note**
>
> 特定の時点に復元することもできます。そのためには、--restore-date-time 引数を使用してコマンドを実行し、タイムスタンプを指定します。設定したリカバリ期間内の任意の時点を指定できます。この値は 1 ~ 35 日の間の任意の値に設定できます。たとえば、次のコマンドでは EarliestRestorableDateTime にテーブルを復元します。
>
> ```
> aws dynamodb restore-table-to-point-in-time \
>     --source-table-name Music \
>     --target-table-name MusicEarliestRestorableDateTime \
>     --no-use-latest-restorable-time \
>     --restore-date-time 1519257118.0
> ```
>
> 特定の時点に復元する場合の --no-use-latest-restorable-time 引数の指定はオプションです。

3. カスタムテーブル設定で、テーブルを特定時点に復元します。この場合、Music テーブルは、LatestRestorableDateTime に復元されます (5 分前まで)。

   復元済みテーブルに対して、次のように別の暗号化モードを指定できます。

> **Note**
>
> sse-specification-override パラメータは、sse-specification-override コマンドで使用される CreateTable パラメータと同じ値を使用します。詳細については、「DynamoDB での暗号化テーブルの管理」を参照してください。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

テーブルは、送信元テーブルが存在する別の AWS リージョンから復元できます。

> **Note**
>
> - sse-specification-override パラメータは、クロスリージョン復元には必須ですが、送信元テーブルと同じリージョンに復元する場合はオプションです。
> - クロスリージョン復元には、source-table-arn パラメータを指定する必要があります。
> - コマンドラインからクロスリージョン復元を実行する場合は、デフォルトの AWS リージョンを希望のリージョンに設定する必要があります。詳細については、「AWS Command Line Interface ユーザーガイド」の「コマンドラインオプション」を参照してください。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

復元済みテーブルの請求モードとプロビジョニングされたスループットをオーバーライドできます。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --billing-mode-override PAY_PER_REQUEST
```

復元済みテーブルで、一部またはすべてのセカンダリインデックスの作成を除外できます。

> **Note**
>
> 新しく復元されるテーブルで、一部またはすべてのセカンダリインデックスの作成を除外すると、復元はより高速でコスト効率が高くなります。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --use-latest-restorable-time \
    --global-secondary-index-override '[]'
```

さまざまなオーバーライドの組み合わせを使用できます。たとえば、次のように単一のグローバルセカンダリインデックスを使用すると同時に、プロビジョニングされたスループットを変更できます。

```
aws dynamodb restore-table-to-point-in-time \
    --source-table-name Music \
    --target-table-name MusicMinutesAgo \
    --billing-mode-override PROVISIONED \
    --provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100
 \
    --global-secondary-index-override IndexName=singers-
index,KeySchema=["{AttributeName=SingerName,KeyType=HASH}"],Projection="{ProjectionType=KEY
 \
    --sse-specification-override Enabled=true,SSEType=KMS \
    --use-latest-restorable-time
```

復元を確認するには、describe-table コマンドで MusicEarliestRestorableDateTime テーブルを指定します。

```
aws dynamodb describe-table --table-name MusicEarliestRestorableDateTime
```

復元中のテーブルは、ステータスが [Creating (作成中)]、復元中に [true] として表示されています。復元プロセスが終了すると、MusicEarliestRestorableDateTime テーブルのステータスは [アクティブ] に変わります。

> **Important**
>
> 復元中は、復元を目的とした、IAM エンティティ (例: ユーザー、グループ、ロール) を付与する AWS Identity and Access Management (IAM) ポリシーの変更や削除を行わないでください。行った場合、予期しない動作が発生する場合があります。たとえば、テーブルの復元中にテーブルの書き込み権限を削除したとします。この場合、基本となる RestoreTableToPointInTime オペレーションを使用しても、復元されたデータをテーブルに書き込むことはできません。復元先のターゲットテーブルにアクセスするための送信元 IP 制限を含む IAM ポリシーでも、同様の問題が発生する可能性があります。
> 復元オペレーション完了後は、アクセス権限の変更または削除のみ行うことができます。

DynamoDB での AWS Backup の使用

Amazon DynamoDB は、AWS Backup の強化されたバックアップ機能を通じて、規制コンプライアンスとビジネス継続性の要件を満たすのに役立ちます。AWS Backup は、フルマネージドのデータ保護サービスであり、AWS サービス間、クラウド内、およびオンプレミスでのバックアップの一元化と自動化を容易にします。このサービスを使用すると、バックアップポリシーを設定し、AWS リソースのアクティビティを 1 か所でモニタリングできます。AWS Backup を使用するには、肯定的にする必要があります。積極的にオプトインする必要があります。オプトインの選択は特定のアカウントと AWS リージョンに適用されるため、同じアカウントを使用して複数のリージョンにオプトインする必要がある場合があります。詳細については、AWSBackup デベロッパーガイドを参照してください。

Amazon DynamoDBは、AWS Backup とネイティブに統合されています。AWS Backup を使用して、DynamoDB オンデマンドバックアップを自動的にスケジュール、コピー、タグ付け、ライフサイクルを実行することができます。これらのバックアップは DynamoDB コンソールから引き続き表示および復元できます。DynamoDB コンソール、API、および AWS コマンドラインインターフェイス (AWS CLI) を使用して、DynamoDB テーブルの自動バックアップを有効にします。

> **Note**
>
> DynamoDB を介して作成されたバックアップは変更されません。バックアップを作成するのに、現在の DynamoDB ワークフローをまだ使用できます。

AWS Backup で利用できる拡張バックアップ機能は次のとおりです。

スケジュールバックアップ - バックアッププランを使用して、DynamoDB テーブルの定期的にスケジュールするバックアップを設定できます。

クロスアカウントとクロスリージョンのコピー - バックアップを異なる AWS リージョンまたはアカウントの別のバックアップボールトに自動的にコピーできます。これにより、データ保護要件をサポートできるようになります。

コールドストレージ階層化 - バックアップを削除したり、古いストレージに移行したりするライフサイクルルールを実施するようにバックアップを設定できます。これにより、バックアップコストを最適化できます。

タグ - 請求およびコスト配分のために、バックアップに自動的にタグを付けることができます。

暗号化 — AWS Backup で管理されている DynamoDB オンデマンドバックアップが AWS Backup ボールトに保存されるようになりました。これにより、DynamoDB テーブルの暗号化キーから独立した AWS KMS key を使用してバックアップを暗号化して保護できます。

バックアップの監査 — AWS Backup Audit Manager を使用して、AWS Backup ポリシーのコンプライアンスを監査し、定義したコントロールにまだ準拠していないバックアップアクティビティとリソースを検索できます。また、バックアップガバナンスのために、日次レポートとオンデマンドレポートの監査証跡を自動的に生成することもできます。

WORM モデルを使用した安全なバックアップ — AWS Backup Vault Lock を使用して、バックアップのWrite-Once-Read-Many（WORM）設定を有効にすることができます。AWS Backup Vault Lock を使用すると、不注意な削除オペレーションや悪意のある削除オペレーション、バックアップ保持期間の変更、ライフサイクル設定の更新からバックアップを保護する保護レイヤーを追加できます。詳細については、AWS BackupVault Lock を参照してください。

これらの拡張バックアップ機能は、すべての AWS リージョンで利用できます。これらの機能の詳細については、AWS Backup デベロッパーガイドを参照してください。

トピック

- AWS Backup を使用した DynamoDB テーブルのバックアップと復元: 仕組み
- AWS Backup Backup を使用した DynamoDB テーブルのバックアップの作成
- AWS Backup を使用した DynamoDB テーブルのバックアップのコピー
- AWS Backup からの DynamoDB テーブルのバックアップの復元。
- AWS Backup を使用して DynamoDB テーブルのバックアップを削除
- AWS Backup によって管理されるオンデマンドバックアップと DynamoDB の使用上の注意の違い

AWS Backup を使用した DynamoDB テーブルのバックアップと復元: 仕組み

オンデマンドバックアップといった特徴を使用して、Amazon DynamoDB テーブルの完全バックアップを作成できます。このセクションでは、バックアップおよび復元プロセス中の概要について示します。

バックアップ

AWS Backup を使用してオンデマンドバックアップを作成する場合、リクエストのタイムマーカーがカタログ化されます。このバックアップは、前回のテーブル全体のスナップショットへのリクエスト時間までにすべての変更を適用して、非同期的に作成されます。

オンデマンドバックアップを作成する度に、テーブルデータ全体がバックアップされます。オンデマンドバックアップの実行可能数に制限はありません。

> **Note**
>
> DynamoDB バックアップとは異なり、AWS Backup で作成されるバックアップは即時に行われません。

バックアップ進行中は、以下を行うことができません。

- バックアップオペレーションの一時停止またはキャンセル。
- バックアップのソーステーブルの削除。
- テーブルのバックアップ中におけるテーブルのバックアップの無効化。

AWS Backup は、自動バックアップスケジュール、保持管理、およびライフサイクル管理を提供します。これにより、カスタムスクリプトや手動プロセスが不要になります。 AWS Backup はバックアップを行い、期限がくるとそれを削除します。詳細については、AWS Backup デベロッパーガイドを参照してください。

コンソールを使用している場合、AWS Backup を使用して作成されたバックアップは、[Backup type (バックアップタイプ)] が AWS_BACKUP に設定された状態で [Backups (バックアップ)] タブに一覧表示されます。

> **Note**
>
> DynamoDB コンソールを使用して、[Backup type (バックアップタイプ)] が AWS_BACKUP でマークされたバックアップを削除することはできません。これらのバックアップを管理するには、AWS Backup コンソールを使用します。

バックアップを実行する方法については、「DynamoDB テーブルのバックアップ」を参照してください。

復元

テーブルは、テーブルのプロビジョニングされたスループットを消費することなく復元します。DynamoDB バックアップからテーブル全体を復元することも、送信先テーブルの設定を構成することもできます。復元を実行するときに、次のテーブル設定を変更できます。

- 暗号化設定

> **Important**
>
> テーブル全体の復元を実行すると、送信先テーブルには、バックアップが要求されたときに送信元テーブルが持っていたのと同じプロビジョニングされた読み込みキャパシティユニットと書き込みキャパシティユニットが設定されます。復元プロセスでは、ローカルセカンダリインデックスおよびグローバルセカンダリインデックスも復元されます。

DynamoDB テーブルデータのバックアップを別の AWS リージョンにコピーしてから、その新しいリージョンで復元することができます。AWS 商用リージョン、AWS 中国リージョン、および AWS GovCloud (米国) リージョン間でバックアップをコピーして復元できます。送信元リージョンからコ

ピーしたデータと、送信先リージョンの新しいテーブルに復元したデータに対してのみ料金が発生します。

AWS Backupは、元のインデックスをすべて含むテーブルを復元します。

以下は、復元されたテーブルで手動で設定する必要があります。

- Auto Scaling ポリシー
- AWS Identity and Access Management (IAM) ポリシー
- Amazon CloudWatch メトリクスおよびアラーム
- タグ
- ストリーム設定
- 有効期限 (TTL) 設定
- 削除保護設定
- ポイントインタイムリカバリ (PITR) 設定

テーブルデータ全体は、バックアップから新しいテーブルにのみ復元することができます。復元されたテーブルに書き込むことができるのは、アクティブになってからです。

> **Note**
>
> AWS Backup 復元は非破壊的です。復元オペレーション中は既存のテーブルを上書きすることはできません。

復元時間は、テーブルの構成 (テーブルのサイズ、基礎となるパーティションの数など) およびその他の関連する変数に直接関係しています。災害対策を計画する際のベストプラクティスは、平均復元完了時間を定期的に記録し、これらの時間が目標復旧時間全体にどのように影響するかを確認することです。

復元を実行する方法については、「バックアップからの DynamoDB テーブルの復元」を参照してください。

IAM ポリシーを使用してアクセスコントロールできます。詳細については、「DynamoDB バックアップおよび復元での IAM の使用」を参照してください。

バックアップおよび復元を行うコンソールと API のアクションはすべて、AWS CloudTrail にキャプチャおよび記録され、ログ記録、継続的モニタリング、監査に使用されます。

AWS Backup Backup を使用した DynamoDB テーブルのバックアップの作成

このセクションでは、AWS Backup をオンにし DynamoDB テーブルからオンデマンドおよびスケジュールでバックアップを作成する方法について説明します。

トピック

- AWS Backup 機能をオンにする
- オンデマンドバックアップ
- スケジュールバックアップ

AWS Backup 機能をオンにする

DynamoDB で使用するには、AWS Backup をオンにする必要があります。

AWS Backup をオンにするには、次の手順を実行します。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups] (バックアップ) を選択します。
3. [Backup Settings] (バックアップの設定) ウィンドウで、[Turn on] (オンにする) を選択します。
4. 確認画面が表示されます。[Turn on features] (オンにする) を選択します。

AWS BackupDynamoDB テーブルで 機能を使用できるようになりました。

オンにした AWS Backup 機能をオフにするには、次の手順を実行します。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups] (バックアップ) を選択します。
3. [Backup Settings] (バックアップの設定) ウィンドウで、[Turn off] (オフにする) を選択します。
4. 確認画面が表示されます。[Turn off features] (オフにする) を選択します。

AWS Backup 機能をオンまたはオフにできない場合は、AWS 管理者がこれらのアクションを実行する必要がある場合があります。

オンデマンドバックアップ

DynamoDB テーブルのオンデマンドバックアップを作成するには、次の手順を実行します。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups] (バックアップ) を選択します。
3. [Create backup] (バックアップの作成) を選択します。
4. ドロップダウンメニューから [Create an on-demand backup] (オンデマンドバックアップを作成) を選択します。
5. ウォームストレージおよびその他の基本機能を備えた AWS Backup によって管理されるバックアップを作成するには、[Default Settings] (デフォルト設定) を選択します。コールドストレージに移行できるバックアップを作成するか、またはAWS Backup の代わりに DynamoDB 機能を使用してバックアップを作成するには、[Customize settings] (設定のカスタマイズ) を選択します。

   代わりに以前の DynamoDB 機能を使用してバックアップを作成するには、[Customize settings] (設定のカスタマイズ) を選択し、次に [Backup with DynamoDB] (DynamoDB でバックアップ) を選択します。
6. 設定が完了したら、[Create backup] (バックアップを作成) を選択します。

スケジュールバックアップ

バックアップをスケジュールするには、次の手順を実行します。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups] (バックアップ) を選択します。
3. 表示されるドロップダウンメニューから [AWS Backup を使用してバックアップをスケジュール] を選択します。
4. AWS Backup に移動して、バックアップ計画を作成します。

AWS Backup を使用した DynamoDB テーブルのバックアップのコピー

現在のバックアップのコピーを作成できます。バックアップは、オンデマンドで複数の AWSアカウントAWSまたはリージョンにコピーすることも、スケジュールしたバックアッププランの一部とし

てあるいは自動的にコピーすることもできます。また、Amazon DynamoDB Encryption Client のクロスアカウントのコピーやクロスリージョンのコピーのシーケンスを自動化することもできます。

クロスリージョンレプリケーションは、ビジネス継続性またはコンプライアンス要件があり、本番稼働用データから最小限の距離だけ離してバックアップを保存する場合に特に役立ちます。

クロスアカウントバックアップは、運用上またはセキュリティ上の理由からバックアップを組織内の 1 つまたは複数の AWS アカウントに確実にコピーする必要がある場合に便利です。元のバックアップを誤って削除してしまった場合は、コピー先のアカウントからコピー元のアカウントにバックアップをコピーし復元できます。これを行うには、Organizations サービスで同じ組織に属する 2 つのアカウントが必要です。

コピーでは、特に指定しない限り、ソースバックアップの設定を継承します。ただし、新しいコピーの「Never」の期限切れを指定した場合は例外があります。この設定では、新しいコピーはソースの有効期限を継承します。新しいバックアップコピーを永続的なコピーにする場合は、ソースバックアップを期限切れにならないように設定するか、新しいコピーの作成から 100 年後を有効期限に指定します。

> **Note**
>
> 別のアカウントにコピーする場合は、まずそのアカウントからのアクセス許可が必要です。

バックアップをコピーするには、以下の操作を行います。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups] (バックアップ) を選択します。
3. コピーするバックアップの横にあるチェックボックスを選択します。
   - コピーするバックアップがグレーアウトしている場合は、AWS Backup で高度な機能を有効にする必要があります。次に、新しいバックアップを作成します。これで、この新しいバックアップを他のリージョンおよびアカウントにコピーし、今後、他の新しいバックアップをコピーできます。
4. [Copy] (コピー) を選択します。
5. バックアップを別のアカウントまたはリージョンにコピーする場合は、[Copy the recovery point to another destination] (リカバリポイントを別の送信先にコピーする) の横にあるチェックボックスをオンにします。次に、アカウント内の別のリージョンにコピーするか、別のリージョンの別のアカウントにコピーするかを選択します。

> **Note**
>
> バックアップを別のリージョンまたはアカウントに復元するには、まずそのリージョンまたはアカウントにバックアップをコピーしておく必要があります。

6. ファイルをコピーする目的のボールドを選択します。必要に応じて、新しいバックアップボールトを作成することもできます。
7. [Copy backup] (バックアップをコピー) を選択します。

AWS Backup からの DynamoDB テーブルのバックアップの復元。

このセクションでは、AWS Backup から DynamoDB テーブルのバックアップを復元する方法について説明します。

トピック

- AWS Backup から DynamoDB テーブルの復元
- DynamoDB テーブルを別のリージョンまたはアカウントに復元

AWS Backup から DynamoDB テーブルの復元

AWS Backup から DynamoDB テーブルを復元するには、次の手順を実行します。

1. AWS マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. [Backups] (バックアップ) タブを選択します。
4. 復元したい以前のバックアップの横にあるチェックボックスを選択します。
5. [復元] を選択します。[Restore table from backup] (バックアップからのテーブルの復元) 画面が表示されます。
6. 新しく復元されたテーブルの名前、この新しいテーブルの暗号化、復元の暗号化に使用するキー、およびその他のオプションを入力します。
7. 完了したら、[Restore] (復元) を選択します。

DynamoDB テーブルを別のリージョンまたはアカウントに復元

DynamoDB テーブルを別のリージョンまたはアカウントに復元するには、まずその新しいリージョンまたはアカウントにバックアップをコピーする必要があります。別のアカウントにコピーするには、まずそのアカウントからアクセス許可を付与される必要があります。DynamoDB バックアップを新しいリージョンまたはアカウントにコピーした後、前のセクションのプロセスを使用して復元できます。

AWS Backup を使用して DynamoDB テーブルのバックアップを削除

このセクションでは、AWS Backup を使用して DynamoDB テーブルのバックアップを削除する方法について説明します。

AWS Backup の機能で作成された DynamoDB バックアップは、AWS Backup ボールトに保存されます。

この種のバックアップを削除するには、次の手順を実行します。

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Backups (バックアップ)] を選択します。
3. 次の画面で、[AWS Backup にアクセス] を選択します。

   AWS Backup コンソール が開きます。AWS Backup コンソール でのバックアップの削除についての詳細は、「AWS Backup デベロッパーガイド」の「バックアップの削除」を参照してください。

   AWS Backup の詳細については、「AWS 規範ガイダンス」の「AWS Backup を使用したバックアップとリカバリ」を参照してください。

AWS Backup によって管理されるオンデマンドバックアップと DynamoDB の使用上の注意の違い

このセクションでは、AWS Backup が管理するオンデマンドバックアップと DynamoDB の技術的な違いについて説明します。

AWS Backup には DynamoDB とは異なるワークフローや動作があります。具体的には次のとおりです。

暗号化 - AWS Backup プランで作成されたバックアップは、AWS Backup サービスが管理するキーを使用して暗号化されたボールトに保存されます。ボールトには、セキュリティを強化するためのアクセス制御ポリシーがあります。

Backup ARN - AWS Backup が作成したバックアップファイルには AWS Backup ARN が含まれるようになり、ユーザーアクセス許可モデルに影響を与える可能性があります。バックアップリソース名 (ARN) が `arn:aws:dynamodb` から `arn:aws:backup` に変更されます。

バックアップの削除 - AWS Backup で作成したバックアップは、AWS Backup ボールトからのみ削除できます。DynamoDB コンソールから AWS Backup ファイルを削除することはできません。

バックアッププロセス - DynamoDB バックアップとは異なり、AWS Backup で作成されるバックアップは即時に行われません。

請求 - AWS Backup 機能を使用した DynamoDB テーブルのバックアップは、AWS Backup から請求されます。

IAM ロール - IAM ロールを介してアクセスを管理する場合は、次の新しいアクセス許可を使用して新しい IAM ロールを設定する必要もあります。

```
"dynamodb:StartAwsBackupJob",
"dynamodb:RestoreTableFromAwsBackup"
```

`dynamodb:StartAwsBackupJob` は AWS Backup 機能を使用してバックアップを完了させるために必要で、`dynamodb:RestoreTableFromAwsBackup` は AWS Backup 機能を使用して作成されたバックアップから復元するために必要です。

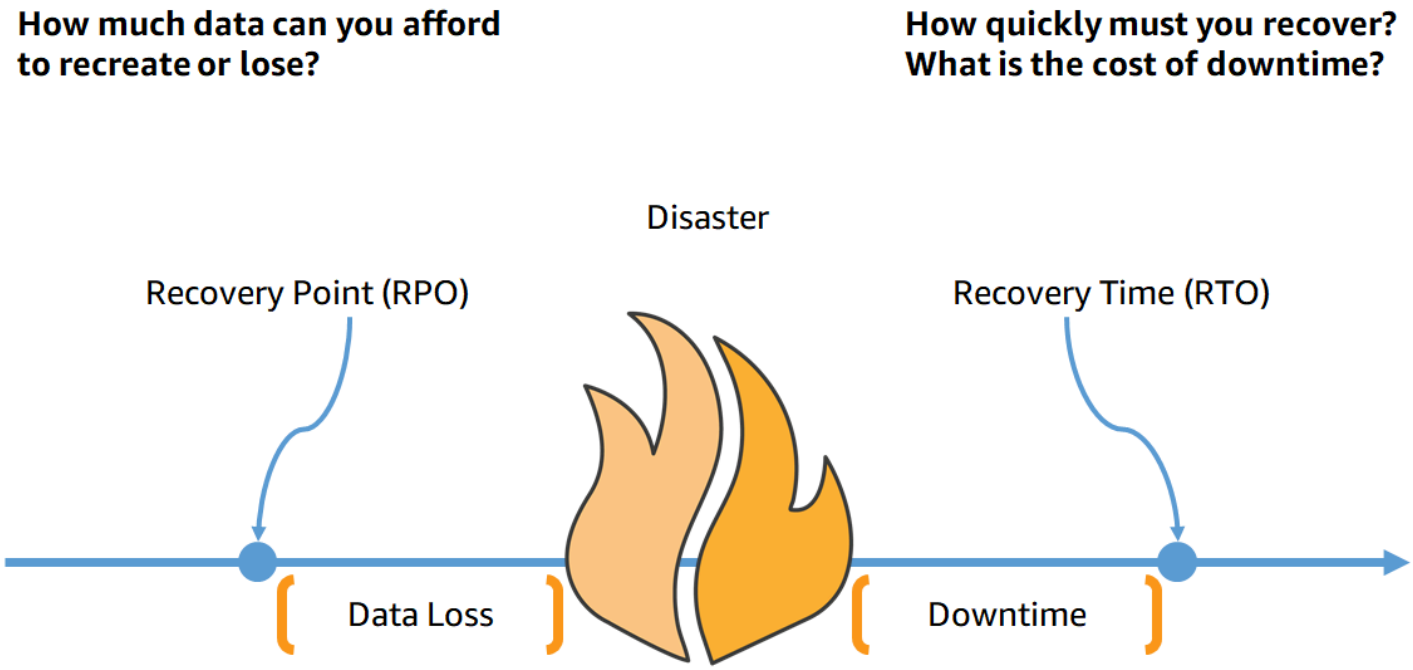
完全な IAM ポリシーでこれらのアクセス許可を確認するには、「DynamoDB バックアップおよび復元での IAM の使用」の例 8 を参照してください。

Amazon DynamoDB ワークロードのディザスタリカバリ戦略の選択

DynamoDB DR 戦略を選択する前に、ビジネス復旧要件を定義する必要があります。この初期分析により、重要なビジネス継続性のニーズを確実に満たすと同時に、コストのかかるオーバーエンジニアリングを回避できます。最適なアプローチは、復旧速度、データ損失耐性、実装の複雑さ、運用コストの 4 つの重要な要素のバランスを取ります。

DR 戦略を形成する 2 つの主要なメトリクスについて説明します。

- 目標復旧時間 (RTO) – これは、サービスの中断から復旧までの最大許容遅延時間です。RTO は「どのくらいの時間、停止を許容できるか?」という質問への回答になります。これは、ビジネス要件に応じて、ゼロ (継続的な可用性が必要) から数時間までの範囲にすることができます。
- 目標復旧時点 (RPO) – これは、最後に実行されたデータ復旧時点からの最大許容時間を表します。RPO は「どのくらいのデータ損失を許容できるか?」という質問への回答になります。例えば、RPO が 1 時間の場合、DR ソリューションでは、インシデント開始前の 1 時間以内の時点にデータを復旧できる必要があります。



DynamoDB DR 戦略の選択は、次の要因に基づいて行う必要があります。

- サービス中断によるビジネスへの影響の分析
- さまざまな災害シナリオのリスク評価
- 各 DR オプションの実装コスト
- データ保護と可用性に関する規制要件

DynamoDB DR アプローチを選択するための決定ポイント

適切な DynamoDB DR アプローチを選択するときは、特定のビジネス要件、リスク許容度、規制要件、および予算の制約を評価する必要があります。

サービス停止中の収益損失が追加の運用コストを大幅に上回るために、ダウンタイムがゼロであることが要件となる場合、マルチリージョンの強力な整合性 (MRSC) テーブルを持つグローバルテーブルなどのアーキテクチャが、即時の復旧機能を提供します。これらのソリューションは運用コストが高くなりますが、即時のフェイルオーバーと継続的な可用性を提供するため、ダウンタイムが収益源に直接影響するミッションクリティカルなアプリケーションには不可欠です。

柔軟な復旧要件を持つ予算意識の高い組織では、復旧シナリオ中にアプリケーションがダウンタイムを何時間も許容できるのであれば、よりコスト効率の良いソリューションを使用できます。オンデマンドバックアップ、AWS Backup、または Amazon Simple Storage Service のエクスポート戦略は、より長い復旧時間を許容する一方で、運用コストを大幅に削減します。これらのアプローチは、ビジネス継続性が重要であるが、即時の復旧をコスト面で正当化できないアプリケーションに適していて、保護とコスト最適化のバランスを取ることができます。

規制コンプライアンスを満たすには、監査への即時の対応と適切なデータ保持ポリシーを確保する自動化されたソリューションが必要です。クロスリージョンレプリケーションを備えた AWS Backup は、より高いコスト構造を維持しながら、包括的なコンプライアンスのオートメーションを提供します。このアプローチは、規制フレームワークに必要なドキュメント、保持ポリシー、地理的分散を提供すると同時に、手動のコンプライアンスオーバーヘッドを削減し、組織全体で一貫したバックアップ手順を確保するために役立ちます。

偶発的な削除やデータ破損などの人的エラーシナリオに対する保護やポイントインタイムリカバリ (PITR) を削除保護と組み合わせることで、単一のリージョン内で継続的な保護が可能になります。このソリューションは運用上のミスに対する優れた保護を提供し、合理的なコストを維持できますが、単一リージョンの制限を受け入れ、リージョンの停止やインフラストラクチャの障害を伴う実際の DR シナリオに対する追加の戦略を計画する必要があります。

ディザスタリカバリオプション

DynamoDB には、テーブルデータをバックアップおよび復元するためのオプションが用意されており、さまざまな障害シナリオに対する耐障害性を提供します。異なるニーズに対応する 4 つの主要な DR ソリューションがあります。

- 継続的なデータ保護のためのポイントインタイムリカバリ (PITR)
- 柔軟な保護のためのオンデマンドバックアップと復元
- 定期的な自動保護のためのスケジュールされたバックアップと復元
- ミッションクリティカルなワークロードの即時復旧のためのグローバルテーブル

バックアップタイプ	バックアップオプション	機能	RPO	RTO	バックアップの可用性	毎月のバックアップコスト	1 回限りの復元コスト
継続的なデータ保護	PITR	最大 35 日間保持される継続的なバックアップにより、設定されたバックアップ期間で数秒以内に PITR が有効になります。	秒	分	単一 AWS リージョンのみ	1 GB/月あたり 0.20 USD	1 GB あたり 0.15 USD
オンデマンド保護	オンデマンドバックアップと復元	オンデマンドプロセス	時間	分	1 つの AWS リージョン	1 GB/月あたり 0.10 USD	1 GB あたり 0.15 USD
スケジュールされたバックアップと復元	AWS Backup	柔軟なスケジューリングオプション (時間単位～月単位)	時間	分	クロスリージョンバックアップをサポート	1 GB/月あたり 0.10 USD	1 GB あたり 0.15 USD
	Amazon S3 のエクスポートとイ	PITR 期間内のフルデータまたは増	分	時間	バケットへのクロスリージョン	1 GB あたり 0.10 USD	1 GB あたり 0.15 USD

バックアップタイプ	バックアップオプション	機能	RPO	RTO	バックアップの可用性	毎月のバックアップコスト	1 回限りの復元コスト
	ンポート (PITR が有効)	分データのバケットへのエクスポート			バックアップが可能		
ミッションクリティカルなワークロードの即時復旧	グローバルテーブル	フルマネージドソリューション、マルチリージョン、マルチアクティブテーブル	秒	Zero	ミリ秒以内のプライマリリージョンからセカンダリリージョンへのデータ変更の自動伝播	レプリケートされた書き込みリクエストユニット 100 万あたり 0.625 USD	0
	MRSC テーブル	強力な整合性を持つフルマネージドソリューション、マルチリージョン、マルチアクティブテーブル	Zero	ゼロ	プライマリリージョンからセカンダリリージョンへのデータ変更のリアルタイムでの自動伝播	レプリケートされた書き込みリクエストユニット 100 万あたり 0.625 USD	0

> **Note**
>
> 上記の表に示すコストは、オンデマンドキャパシティーの DynamoDB 料金に基づくバージニア北部リージョンの料金です。

PITR を使用した継続的なデータ保護

インベントリ管理、製品カタログ、注文処理に DynamoDB を使用する e コマース小売業者の例を挙げます。ブラックフライデーイベントの準備中に、開発チームが誤って製品カタログを削除しました。PITR を使用すれば、何千ものレコードをインシデント前の任意の正確な時点に 30 分で復元できるため、何百万 USD もの潜在的な収益損失を防ぐことができます。

PITR を有効にすると、DynamoDB はテーブルデータを 1 秒あたりの詳細度で自動的にバックアップします。テーブルは、1 ~ 35 日の範囲で設定された復旧期間内の任意の秒単位の時点に復元できます。注目すべきことに、保持期間を短縮してもコストは減少しないため、保持期間を延長してもコストの増減はありません。PITR 機能は、最小限の RPO (秒) と合理的な RTO (分単位 ~ 時間単位) できめ細かな復旧機能を提供するため、さまざまな DR 要件に適しています。実際のシナリオでは、PITR の最小限の RPO と合理的な RTO が、人為的ミスやアプリケーションの障害から保護するか、厳格な規制要件を満たすかにかかわらず、潜在的なビジネス上の障害を管理可能な復旧オペレーションに変える方法を示しています。

詳細については、「DynamoDB でポイントインタイムリカバリを有効にする」を参照してください。CloudFormation、AWS CLI、および DynamoDB API を使用して PITR を設定し、テーブルを復元する方法について詳しく説明しています。

オンデマンドバックアップと復元を使用した柔軟な保護

大規模な電子医療記録 (EHR) のシステム移行を計画している医療サービス提供者を例に挙げます。移行プロセス全体でマイルストーンのバックアップを使用すると、厳格な HIPAA 要件を満たすと同時に、法務チームに変更不可能な監査証拠を提供できます。移行中にコンプライアンスの問題が発生した場合、正確なデータ処理手順を示し、準拠したチェックポイントに復元して、患者のデータ保護と規制遵守を実現できます。

DynamoDB オンデマンドバックアップ機能を使用すると、必要に応じてバックアップを作成し、テーブルを復元できます。オンデマンドバックアップは非同期的に動作し、バックアップリクエストが行われた時点までのすべての変更をキャプチャします。オンデマンドバックアップ機能により、データ保護戦略を正確に管理しながら、さまざまな運用要件とコンプライアンス要件をサポートできます。

オンデマンドバックアップと復元は、同じリージョンシナリオとクロスリージョンシナリオの両方で柔軟なテーブル復元機能を提供し、汎用性の高い復旧オプションを選択できます。ビジネス継続性を強化するために、DynamoDB はテーブルサイズに関係なく瞬時にバックアップを完了します。また、同じリージョンの復元は復旧時間を最小限に抑え、運用上の可用性を最大化するための最適なパフォーマンスを提供します。

日次バックアップ戦略を実装することで、信頼性の高い 24 時間の RPO を達成し、数分～数時間の予測可能な RTO で補完できます。この構造化されたアプローチをデータ保護に使用することで、バックアップと復旧の戦略を効果的に計画、実行しながら、ビジネスの運営を維持できます。

DynamoDB のバックアップおよび復元オペレーションの詳細については、「DynamoDB テーブルのバックアップ」および「バックアップからの DynamoDB テーブルの復元」を参照してください。

スケジュールされたバックアップと復元を使用した定期的な自動保護

DynamoDB は、スケジュールされたワークフローを通じて自動化できる複数の保護メカニズムを提供します。これには、運用を復旧するのためのカスタマイズ可能なバックアッププランを使用する AWS Backup や、長期アーカイブ、クロスリージョンのデータ移行、継続的な運用のための Amazon S3 エクスポート機能が含まれます。これらの自動化された定期的な保護スケジュールを実装することで、復旧目標とストレージコストのバランスを取りながら、手動による介入なしで一貫したデータ保護を実現できます。

AWS Backup およびバックアッププランを使用したバックアップと復元

Amazon DynamoDB ベースの不正検出システムにエンタープライズグレードの保護を追加する必要がある、15 か国で業務を行っている多国籍銀行の例を挙げます。この保護では、継続的な可用性を確保しながら、管轄区域全体の多様な規制要件に適応できる必要があります。

この銀行は、AWS Backup のマルチリージョンレプリケーションとクロスアカウント機能を使用して、保護アーキテクチャを実装しました。このアーキテクチャは、さまざまなビジネスユニットで厳格なアクセスコントロールを維持しながら、リージョン間で重要なトランザクションデータを自動的にレプリケートします。3 か国の規制当局からコンプライアンス監査のために過去のトランザクションデータを同時に要求された際に、この銀行の標準化されたバックアップポリシーと詳細な監査証跡により、数週間ではなく数時間以内に完全なドキュメントを提供することができました。

AWS Backup を使用すると、特定の AWS アカウント内でデータの定期的なバックアップを取ることができます。バックアッププランを使用すると、これらのバックアップを異なるリージョンにオンデマンドでコピーすることも、スケジュールされたバックアッププランの一部として自動的にコピーすることもできます。バックアッププランには柔軟なスケジューリングオプションが用意されているた

め、ニーズに合わせて、1 時間ごと、12 時間ごと、毎日、毎週、毎月のいずれかの頻度を選択できます。DynamoDB バックアップは、選択した頻度に基づいてリージョン間でコピーされます。

AWS Backup Vault Lock は、Write-Once Read-Many (WORM) バックアップを適用して、バックアップボールトのバックアップ (復旧ポイント) を不注意によるアクションや悪意のあるアクションから保護するとともに、ランサムウェア攻撃に対する保護を提供します。RPO は 1 時間、RTO はテーブルサイズに応じて数分 ~ 数時間まで達成できます。

AWS Backup が DynamoDB とどのように統合されるかを理解するとともに、バックアップ機能およびベストプラクティスについて学習するには、「DynamoDB での AWS Backup の使用」を参照してください。

Amazon Simple Storage Service (Amazon S3) のエクスポートとインポート

99.99% の可用性を維持しながら、複数の管轄区域にわたるトランザクションデータの規制要件を満たす必要があるグローバルな金融サービス会社を例に挙げます。この企業は、Amazon S3 のエクスポートおよびインポート機能を中心に据えてエンタープライズデータ戦略を構築し、クロスリージョンデータレプリケーションを実装して 12 か国のデータ主権法に準拠し、リージョン間フェイルオーバーのメカニズムを確立することができました。欧州で障害が発生した場合、これらのシステムはフェイルオーバー手順を自動的に実行し、取引の中断をゼロに抑えた継続的な運用の実現を支援します。

この企業は、Amazon S3 へのエクスポート/インポート機能を使用して、12 リージョンすべての管轄区域固有の Amazon S3 バケットへの毎日のフルバックアップと 1 時間ごとの増分エクスポートを作成し、現地のデータレジデンシー要件に準拠できるようにしました。各リージョンテーブルは、同じ管轄区域内の暗号化されたバケットにトランザクションデータを自動的にエクスポートします。また、クロスリージョンレプリケーションにより重要なデータセットの冗長性が高まりました。エクスポートは読み込みキャパシティーユニット (RCU) を消費せず、テーブルのパフォーマンスや可用性にも影響しません。この企業が運用しているリージョンのいずれかに障害が発生した場合、自動復旧手順によって代替リージョンに最新のエクスポートを復元し、完全な取引業務を再開することができます。このエクスポート/インポート戦略は、イミュータブルな監査証跡による規制コンプライアンスと堅牢な DR 機能の両方を提供します。同社は、利用可能な任意のリージョンの新しいテーブルに復元できる完全なトランザクション履歴を保持しながら、継続的なオペレーションを維持できるため、リージョンのインフラストラクチャ障害時のデータ損失とダウンタイムを最小限に抑えることができます。

DynamoDB の Amazon S3 へのエクスポート機能により、PITR 期間内の特定時点の DynamoDB テーブルからデータをエクスポートできます。テーブルのサイズに応じて、数分の RPO と、数分 ~ 数時間の RTO を実現できます。

エクスポートの設定に関する詳細な実装ガイダンスについては、「DynamoDB でのテーブルエクスポートのリクエスト」を参照してください。復元オペレーションとデータのインポートについては、「DynamoDB でのテーブルインポートのリクエスト」を参照してください。

グローバルテーブルを使用したミッションクリティカルなワークロードの即時復旧

DynamoDB グローバルテーブルは、プライマリリージョンの中断時に堅牢なフェイルオーバー機能を提供します。アプリケーションはトラフィックを正常なリージョンに即座にリダイレクトし、運用を続行できます。グローバルテーブルには、マルチリージョンの結果整合性 (MREC) およびマルチリージョンの強力な整合性 (MRSC) の 2 つのオプションがあります。

マルチリージョンの結果整合性

北米、欧州、アジアパシフィックの何百万もの顧客にサービスを提供する多国籍の e コマースサービス企業が、適切なグローバルインフラストラクチャがないと運用上の大きな課題に直面する可能性があるとします。この企業は、海外の顧客に対する長い決済時間、複数の管轄区域にわたる複雑なコンプライアンス要件、リージョン停止中の頻繁な中断に苦慮する可能性があります。このような場合は、DynamoDB グローバルテーブルを実装することで、業務を変革できます。東京の顧客に対し、超低レイテンシーのローカルデータアクセスを通じて、ニューヨークの顧客と同じ非常に高速な決済時間を実現できます。激しい気象現象によってプライマリリージョンが中断された場合でも、アプリケーションはお客様に気付かれることなくシームレスに別のリージョンにフェイルオーバーできます。このアプローチにより、高可用性が維持され、グローバルな読み取り/書き込みレイテンシーが改善し、すべてのリージョンで規制コンプライアンスへの準拠がサポートされます。また、現地の要件を満たしながら、インフラストラクチャを世界中で確実に拡大できることを確信したうえで、自信を持って新しい市場に進出できます。

DynamoDB グローバルテーブルでは、フルマネージドで、マルチリージョン、マルチアクティブなデータベースオプションが提供されます。このソリューションは、大規模に拡張されたグローバルアプリケーションからの読み込み/書き込みのパフォーマンスを高速に、かつローカルに実現するよう設計されています。DR の目的で、テーブルを使用可能にするプライマリリージョンと DR リージョンを指定できます。DynamoDB は、進行中のデータ変更をプライマリリージョンから DR レプリカに自動的に伝播します。MREC セットアップでは、数秒の RPO とゼロ RTO を実現できます。

DynamoDB グローバルテーブルの作成と管理に関するガイダンスについては、「グローバルテーブル - DynamoDB の複数リージョンレプリケーション」を参照してください。

マルチリージョンの強力な整合性

3 つの大陸で毎日数百万件の取引を処理する、主要なデジタルバンキングプラットフォームを例に挙げます。このプラットフォームでは、ピーク時の取引時間中にリージョンで大規模な停止が発生す

る可能性があります。緊急に手動の DR 手順で対応し、数百万 USD 規模の損失危機に直面することなく、アーキテクチャは数秒以内に DR をアクティブ化し、中断について顧客に気付かれずに、トラフィックを正常なリージョンにシームレスにリダイレクトします。銀行のモバイルアプリはローンの申し込みをリアルタイムで処理し続け、取引プラットフォームは高頻度取引に対してミリ秒の応答時間を維持し、顧客の口座残高はすべてのリージョンで完全に同期されたままになります。

マルチリージョンの強力な整合性のグローバルテーブルを使用し、RPO を 0 にしてアプリケーションを構築できます。RPO が 0 の場合、アプリケーションの中断によってトラフィックを別のリージョンに移行する必要がある場合でも、アプリケーションは最新バージョンの DynamoDB データを読み取ることができます。

> **Note**
>
> DR の目的で、2 つのレプリカと 1 つの監視リージョンを使用できます。

MRSC グローバルテーブル作成の詳細については、「グローバルテーブルの仕組み」を参照してください。

AWS SDK を使用した DynamoDB のコード例

以下は、AWS ソフトウェア開発キット (SDK) で DynamoDB を使用する方法を説明するコード例です。

基本 は、重要なオペレーションをサービス内で実行する方法を示すコード例です。

アクションはより大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。アクションは個々のサービス機能を呼び出す方法を示していますが、コンテキスト内のアクションは、関連するシナリオで確認できます。

シナリオは、1 つのサービス内から、または他の AWS のサービスと組み合わせて複数の関数を呼び出し、特定のタスクを実行する方法を示すコード例です。

AWS コミュニティへの貢献は、AWS 間の複数のチームによって作成および維持されている例です。フィードバックを提供するには、リンクされたリポジトリで提供されているメカニズムを使用します。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

コードの例

- AWS SDK を使用した DynamoDB の基本的な例
  - Hello DynamoDB
  - AWS SDK を使用した DynamoDB の基本の学習
  - AWS SDK を使用した DynamoDB 向けアクション
    - AWS SDK で BatchExecuteStatement を使用する
    - AWS SDK または CLI で BatchGetItem を使用する
    - AWS SDK または CLI で BatchWriteItem を使用する
    - AWS SDK または CLI で CreateTable を使用する
    - AWS SDK または CLI で DeleteItem を使用する
    - AWS SDK または CLI で DeleteTable を使用する
    - AWS SDK または CLI で DescribeTable を使用する
    - AWS SDK または CLI で DescribeTimeToLive を使用する
    - AWS SDK で ExecuteStatement を使用する

    - AWS SDK または CLI で GetItem を使用する
    - AWS SDK または CLI で ListTables を使用する
    - AWS SDK または CLI で PutItem を使用する
    - AWS SDK または CLI で Query を使用する
    - AWS SDK または CLI で Scan を使用する
    - AWS SDK または CLI で UpdateItem を使用する
    - AWS SDK または CLI で UpdateTable を使用する
    - AWS SDK または CLI で UpdateTimeToLive を使用する
- AWS SDK を使用した DynamoDB のシナリオ
    - AWS SDK を使用して DAX で DynamoDB の読み取りを高速化する
    - AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスの高度なシナリオを操作する
    - DynamoDB テーブルにデータを送信するアプリケーションを構築する
    - AWS SDK を使用して、DynamoDB で複数の値を 1 つの属性と比較する
    - AWS SDK を使用して TTL を含む DynamoDB 項目を条件付きで更新する
    - AWS SDK を使用してローカル DynamoDB インスタンスに接続する
    - AWS SDKを使用した DynamoDB のカウント式演算子
    - COVID-19 データを追跡する API Gateway REST API を作成する
    - ステップ関数でメッセンジャーアプリケーションを作成する
    - ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成
    - AWS SDK を使用してグローバルセカンダリインデックスを含む DynamoDB テーブルを作成する
    - AWS SDK を使用してウォームスループット設定で DynamoDB テーブルを作成する
    - DynamoDB データを追跡するウェブアプリケーションを作成する
    - API Gateway で WebSocket チャットアプリケーションを作成する
    - AWS SDK を使用して TTL を含む DynamoDB 項目を作成する
    - AWS SDK を使用して、DynamoDB でマルチリージョンの強力な整合性を備えたグローバルテーブルを作成して管理する
    - AWS SDK を使用した MREC を示す DynamoDB グローバルテーブルの作成と管理
    - AWS SDK で PartiQL DELETE ステートメントを使用して DynamoDB データを削除する
    - AWS SDK を使用して、Amazon Rekognition で画像内の PPE を検出する

- NoSQL データベースの開始方法
- AWS SDK で PartiQL INSERT ステートメントを使用して DynamoDB データを挿入する
- ブラウザからの Lambda 関数の呼び出し
- AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスを管理する
- AWS Command Line Interface v2 を使用して DynamoDB リソースベースのポリシーを管理する
- AWS SDK を使用した Amazon DynamoDB のパフォーマンスのモニタリング
- AWS SDK を使用して高度な DynamoDB クエリオペレーションを実行する
- AWS SDK を使用して DynamoDB でリストオペレーションを実行する
- AWS SDK を使用して DynamoDB でマップオペレーションを実行する
- AWS SDK を使用して DynamoDB でセットオペレーションを実行する
- PartiQL ステートメントのバッチと AWS SDK を使用して DynamoDB テーブルにクエリを実行する
- PartiQL と AWS SDK を使用して DynamoDB テーブルに対してクエリを実行する
- AWS SDK でグローバルセカンダリインデックスを使用して DynamoDB テーブルをクエリする
- AWS SDK で begins_with 条件を使用して DynamoDB テーブルをクエリする
- AWS SDK でソートキーの日付範囲を使用して DynamoDB テーブルをクエリする
- AWS SDK で複雑なフィルター式を使用して DynamoDB テーブルをクエリする
- AWS SDK で動的フィルター式を使用して DynamoDB テーブルをクエリする
- AWS SDK でフィルター式と制限を使用して DynamoDB テーブルをクエリする
- AWS SDK でネストされた属性を使用して DynamoDB テーブルをクエリする
- AWS SDK でページ分割を使用して DynamoDB テーブルをクエリする
- AWS SDK で強力な整合性のある読み込みを使用して DynamoDB テーブルをクエリする
- AWS SDK で PartiQL SELECT ステートメントを使用して DynamoDB データをクエリする
- AWS SDK を使用して、DynamoDB テーブルの TTL 項目に対してクエリを実行する
- AWS SDK で日付と時刻のパターンを使用して DynamoDB テーブルをクエリする
- AWS SDK を使用して、EXIF とその他の画像情報を保存する
- AWS Command Line Interface v2 を使用して DynamoDB の属性ベースのアクセス制御を設定する
- AWS SDK を使用した DynamoDB の式順序の更新を理解する
- AWS SDK を使用してウォームスループットで DynamoDB テーブル設定を更新する

- AWS SDK を使用して TTL を含む DynamoDB 項目を更新する
- AWS SDK で PartiQL UPDATE ステートメントを使用して DynamoDB データを更新する
- API Gateway を使用して Lambda 関数を呼び出す
- Step Functions を使用して Lambda 関数を呼び出す
- AWS SDK を使用して DynamoDB のドキュメントモデルを使用する
- AWS SDK を使用して DynamoDB 用の高レベルのオブジェクト永続性モデルを使用する
- AWS SDK を使用して DynamoDB でアトミックカウンターオペレーションを使用する
- AWS SDK を使用して DynamoDB で条件付きオペレーションを使用する
- AWS SDK を使用して DynamoDB で式の属性名を使用する
- スケジュールされたイベントを使用した Lambda 関数の呼び出し
- AWS Command Line Interface v2 を使用して DynamoDB ローカルセカンダリインデックスを操作する
- AWS Command Line Interface v2 を使用して DynamoDB Streams と Time-to-Live を操作する
- AWS Command Line Interface v2 を使用して、DynamoDB グローバルテーブルと最終的な一貫性を備えたマルチリージョンレプリケーション (MREC) を操作する
- AWS Command Line Interface v2 を使用して DynamoDB リソースのタグ付けを操作する
- AWS Command Line Interface v2 を使用して DynamoDB テーブルの暗号化を操作する

- DynamoDB 用のサーバーレスサンプル
  - DynamoDB トリガーから Lambda 関数を呼び出す
  - DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする
- AWSDynamoDB の コミュニティへの貢献
  - サーバーレスアプリケーションの構築とテスト

AWS SDK を使用した DynamoDB の基本的な例

以下のコード例は、AWS SDK で Amazon DynamoDB の基本を使用する方法を示しています。

例

- Hello DynamoDB
- AWS SDK を使用した DynamoDB の基本の学習
- AWS SDK を使用した DynamoDB 向けアクション
  - AWS SDK で BatchExecuteStatement を使用する

- AWS SDK または CLI で BatchGetItem を使用する
- AWS SDK または CLI で BatchWriteItem を使用する
- AWS SDK または CLI で CreateTable を使用する
- AWS SDK または CLI で DeleteItem を使用する
- AWS SDK または CLI で DeleteTable を使用する
- AWS SDK または CLI で DescribeTable を使用する
- AWS SDK または CLI で DescribeTimeToLive を使用する
- AWS SDK で ExecuteStatement を使用する
- AWS SDK または CLI で GetItem を使用する
- AWS SDK または CLI で ListTables を使用する
- AWS SDK または CLI で PutItem を使用する
- AWS SDK または CLI で Query を使用する
- AWS SDK または CLI で Scan を使用する
- AWS SDK または CLI で UpdateItem を使用する
- AWS SDK または CLI で UpdateTable を使用する
- AWS SDK または CLI で UpdateTimeToLive を使用する

Hello DynamoDB

次のコード例は、DynamoDB の使用を開始する方法を示しています。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace DynamoDBActions;

/// <summary>
/// A simple example that demonstrates basic DynamoDB operations.
/// </summary>
public class HelloDynamoDB
{
    /// <summary>
    /// HelloDynamoDB lists the existing DynamoDB tables for the default user.
    /// </summary>
    /// <param name="args">Command line arguments</param>
    /// <returns>Async task.</returns>
    static async Task Main(string[] args)
    {
        // Set up dependency injection for Amazon DynamoDB.
        using var host =
 Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonDynamoDB>()
            )
            .Build();

        // Now the client is available for injection.
        var dynamoDbClient = host.Services.GetRequiredService<IAmazonDynamoDB>();

        try
        {
            var request = new ListTablesRequest();
            var tableNames = new List<string>();

            var paginatorForTables =
 dynamoDbClient.Paginators.ListTables(request);

            await foreach (var tableName in paginatorForTables.TableNames)
            {
                tableNames.Add(tableName);
            }

            Console.WriteLine("Welcome to the DynamoDB Hello Service example. " +
                              "\nLet's list your DynamoDB tables:");
            tableNames.ForEach(table =>
            {
                Console.WriteLine($"Table: {table}");
```

```
            });
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB service error occurred while
 listing tables. {ex.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while listing tables.
 {ex.Message}");
        }
    }
}
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListTables」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

CMakeLists.txt CMake ファイルのコード。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS dynamodb)

# Set this project's name.
project("hello_dynamodb")

# Set the C++ standard to use to build this target.
```

```
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_dynamodb.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_dynamodb.cpp ソースファイルのコード。

```
#include <aws/core/Aws.h>
#include <aws/dynamodb/DynamoDBClient.h>
#include <aws/dynamodb/model/ListTablesRequest.h>
#include <iostream>
```

```
/*
 *  A "Hello DynamoDB" starter application which initializes an Amazon DynamoDB
 (DynamoDB) client and lists the
 *  DynamoDB tables.
 *
 *  main function
 *
 *  Usage: 'hello_dynamodb'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.

    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::DynamoDB::DynamoDBClient dynamodbClient(clientConfig);
        Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
        listTablesRequest.SetLimit(50);
        do {
            const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
 dynamodbClient.ListTables(
                    listTablesRequest);
            if (!outcome.IsSuccess()) {
                std::cout << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
                result = 1;
                break;
            }

            for (const auto &tableName: outcome.GetResult().GetTableNames()) {
                std::cout << tableName << std::endl;
            }

            listTablesRequest.SetExclusiveStartTableName(
                    outcome.GetResult().GetLastEvaluatedTableName());
```

```
        } while (!listTablesRequest.GetExclusiveStartTableName().empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「ListTables」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListTables {
    public static void main(String[] args) {
        System.out.println("Listing your Amazon DynamoDB tables:\n");
```

```
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        listAllTables(ddb);
        ddb.close();
    }

    public static void listAllTables(DynamoDbClient ddb) {
        boolean moreTables = true;
        String lastName = null;

        while (moreTables) {
            try {
                ListTablesResponse response = null;
                if (lastName == null) {
                    ListTablesRequest request =
ListTablesRequest.builder().build();
                    response = ddb.listTables(request);
                } else {
                    ListTablesRequest request = ListTablesRequest.builder()
                            .exclusiveStartTableName(lastName).build();
                    response = ddb.listTables(request);
                }

                List<String> tableNames = response.tableNames();
                if (tableNames.size() > 0) {
                    for (String curName : tableNames) {
                        System.out.format("* %s\n", curName);
                    }
                } else {
                    System.out.println("No tables found!");
                    System.exit(0);
                }

                lastName = response.lastEvaluatedTableName();
                if (lastName == null) {
                    moreTables = false;
                }

            } catch (DynamoDbException e) {
                System.err.println(e.getMessage());
                System.exit(1);
            }
```

```
        }
        System.out.println("\nDone!");
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「ListTables」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

AWS SDK for JavaScript で DynamoDB を使用する方法の詳細については、「JavaScript による DynamoDB のプログラミング」を参照してください。

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response.TableNames.join("\n"));
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「ListTables」を参照してください。

Python

SDK for Python v4

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
This is a test output. Examples for AWS SDK for Python
for AWS DynamoDB will be available here soon.
```

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import boto3

# Create a DynamoDB client using the default credentials and region
dynamodb = boto3.client("dynamodb")

# Initialize a paginator for the list_tables operation
paginator = dynamodb.get_paginator("list_tables")

# Create a PageIterator from the paginator
page_iterator = paginator.paginate(Limit=10)

# List the tables in the current AWS account
print("Here are the DynamoDB tables in your account:")

# Use pagination to list all tables
table_names = []
```

```
for page in page_iterator:
    for table_name in page.get("TableNames", []):
        print(f"- {table_name}")
        table_names.append(table_name)

if not table_names:
    print("You don't have any DynamoDB tables in your account.")
else:
    print(f"\nFound {len(table_names)} tables.")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「ListTables」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
require 'aws-sdk-dynamodb'
require 'logger'

# DynamoDBManager is a class responsible for managing DynamoDB operations
# such as listing all tables in the current AWS account.
class DynamoDBManager
  def initialize(client)
    @client = client
    @logger = Logger.new($stdout)
  end

  # Lists and prints all DynamoDB tables in the current AWS account.
  def list_tables
    @logger.info('Here are the DynamoDB tables in your account:')
```

```
    paginator = @client.list_tables(limit: 10)
    table_names = []

    paginator.each_page do |page|
      page.table_names.each do |table_name|
        @logger.info("- #{table_name}")
        table_names << table_name
      end
    end

    if table_names.empty?
      @logger.info("You don't have any DynamoDB tables in your account.")
    else
      @logger.info("\nFound #{table_names.length} tables.")
    end
  end
end

if $PROGRAM_NAME == __FILE__
  dynamodb_client = Aws::DynamoDB::Client.new
  manager = DynamoDBManager.new(dynamodb_client)
  manager.list_tables
end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「ListTables」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した DynamoDB の基本の学習

次のコード例は、以下を実行する方法を示しています。

- 映画データを保持できるテーブルを作成する。
- テーブルに 1 つの映画を入れ、取得して更新する。
- サンプル JSON ファイルから映画データをテーブルに書き込む。
- 特定の年にリリースされた映画を照会する。

- 一定期間内に公開された映画をスキャンします。
- テーブルからムービーを削除し、テーブルを削除します。

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// <summary>
/// This example application performs the following basic Amazon DynamoDB
/// functions:
///     CreateTableAsync
///     PutItemAsync
///     UpdateItemAsync
///     BatchWriteItemAsync
///     GetItemAsync
///     DeleteItemAsync
///     Query
///     Scan
///     DeleteItemAsync.
/// </summary>
public class DynamoDbBasics
{
    public static bool IsInteractive = true;

    // Separator for the console display.
    private static readonly string SepBar = new string('-', 80);

    /// <summary>
    /// The main entry point for the DynamoDB Basics example application.
    /// </summary>
    /// <param name="args">Command line arguments.</param>
    /// <returns>A task representing the asynchronous operation.</returns>
    public static async Task Main(string[] args)
    {
```

```
        // Set up dependency injection for Amazon DynamoDB.
        using var host =
Microsoft.Extensions.Hosting.Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonDynamoDB>()
                    .AddTransient<DynamoDbWrapper>())
            .Build();

        // Now the wrapper is available for injection.
        var dynamoDbWrapper =
host.Services.GetRequiredService<DynamoDbWrapper>();

        var tableName = "movie_table";

        var movieFileName = @"movies.json";

        DisplayInstructions();

        // Create a new table and wait for it to be active.
        Console.WriteLine($"Creating the new table: {tableName}");

        var success = await dynamoDbWrapper.CreateMovieTableAsync(tableName);

        Console.WriteLine(success
            ? $"\nTable: {tableName} successfully created."
            : $"\nCould not create {tableName}.");

        WaitForEnter();

        // Add a single new movie to the table.
        var newMovie = new Movie
        {
            Year = 2021,
            Title = "Spider-Man: No Way Home",
        };

        success = await dynamoDbWrapper.PutItemAsync(newMovie, tableName);
        if (success)
        {
            Console.WriteLine($"Added {newMovie.Title} to the table.");
        }
        else
        {
            Console.WriteLine("Could not add movie to table.");
```

```
        }

        WaitForEnter();

        // Update the new movie by adding a plot and rank.
        var newInfo = new MovieInfo
        {
            Plot = "With Spider-Man's identity now revealed, Peter asks" +
                   "Doctor Strange for help. When a spell goes wrong, dangerous"
+
                   "foes from other worlds start to appear, forcing Peter to" +
                   "discover what it truly means to be Spider-Man.",
            Rank = 9,
        };

        success = await dynamoDbWrapper.UpdateItemAsync(newMovie, newInfo,
tableName);
        if (success)
        {
            Console.WriteLine($"Successfully updated the movie:
{newMovie.Title}");
        }
        else
        {
            Console.WriteLine("Could not update the movie.");
        }

        WaitForEnter();

        // Add a batch of movies to the DynamoDB table from a list of
        // movies in a JSON file.
        var itemCount = await dynamoDbWrapper.BatchWriteItemsAsync(movieFileName,
tableName);
        Console.WriteLine($"Added {itemCount} movies to the table.");

        WaitForEnter();

        // Get a movie by key. (partition + sort)
        var lookupMovie = new Movie
        {
            Title = "Jurassic Park",
            Year = 1993,
        };
```

```
        Console.WriteLine("Looking for the movie \"Jurassic Park\".");
        var item = await dynamoDbWrapper.GetItemAsync(lookupMovie, tableName);
        if (item?.Count > 0)
        {
            dynamoDbWrapper.DisplayItem(item);
        }
        else
        {
            Console.WriteLine($"Couldn't find {lookupMovie.Title}");
        }

        WaitForEnter();

        // Delete a movie.
        var movieToDelete = new Movie
        {
            Title = "The Town",
            Year = 2010,
        };

        success = await dynamoDbWrapper.DeleteItemAsync(tableName,
movieToDelete);

        if (success)
        {
            Console.WriteLine($"Successfully deleted {movieToDelete.Title}.");
        }
        else
        {
            Console.WriteLine($"Could not delete {movieToDelete.Title}.");
        }

        WaitForEnter();

        // Use Query to find all the movies released in 2010.
        int findYear = 2010;
        Console.WriteLine($"Movies released in {findYear}");
        var queryCount = await dynamoDbWrapper.QueryMoviesAsync(tableName,
findYear);
        Console.WriteLine($"Found {queryCount} movies released in {findYear}");

        WaitForEnter();

        // Use Scan to get a list of movies from 2001 to 2011.
```

```
        int startYear = 2001;
        int endYear = 2011;
        var scanCount = await dynamoDbWrapper.ScanTableAsync(tableName,
startYear, endYear);
        Console.WriteLine($"Found {scanCount} movies released between {startYear}
and {endYear}");

        WaitForEnter();

        // Delete the table.
        success = await dynamoDbWrapper.DeleteTableAsync(tableName);

        if (success)
        {
            Console.WriteLine($"Successfully deleted {tableName}");
        }
        else
        {
            Console.WriteLine($"Could not delete {tableName}");
        }

        Console.WriteLine("The DynamoDB Basics example application is
complete.");

        WaitForEnter();
    }

    /// <summary>
    /// Displays the description of the application on the console.
    /// </summary>
    private static void DisplayInstructions()
    {
        if (!IsInteractive)
        {
            return;
        }

        Console.Clear();
        Console.WriteLine();
        Console.Write(new string(' ', 28));
        Console.WriteLine("DynamoDB Basics Example");
        Console.WriteLine(SepBar);
        Console.WriteLine("This demo application shows the basics of using
DynamoDB with the AWS SDK.");
```

```
        Console.WriteLine(SepBar);
        Console.WriteLine("The application does the following:");
        Console.WriteLine("\t1. Creates a table with partition: year and
 sort:title.");
        Console.WriteLine("\t2. Adds a single movie to the table.");
        Console.WriteLine("\t3. Adds movies to the table from moviedata.json.");
        Console.WriteLine("\t4. Updates the rating and plot of the movie that was
 just added.");
        Console.WriteLine("\t5. Gets a movie using its key (partition + sort).");
        Console.WriteLine("\t6. Deletes a movie.");
        Console.WriteLine("\t7. Uses QueryAsync to return all movies released in
 a given year.");
        Console.WriteLine("\t8. Uses ScanAsync to return all movies released
 within a range of years.");
        Console.WriteLine("\t9. Finally, it deletes the table that was just
 created.");
        WaitForEnter();
    }

    /// <summary>
    /// Simple method to wait for the Enter key to be pressed.
    /// </summary>
    private static void WaitForEnter()
    {
        if (IsInteractive)
        {
            Console.WriteLine("\nPress <Enter> to continue.");
            Console.WriteLine(SepBar);
            _ = Console.ReadLine();
        }
    }
}
```

挿入されたクライアントをテーブルオペレーションに使用します。

```
using System.Text.Json;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
```

```
namespace DynamoDBActions;

/// <summary>
/// Methods of this class perform Amazon DynamoDB operations.
/// </summary>
public class DynamoDbWrapper
{
    private readonly IAmazonDynamoDB _amazonDynamoDB;

    /// <summary>
    /// Constructor for the DynamoDbWrapper class.
    /// </summary>
    /// <param name="amazonDynamoDB">The injected DynamoDB client.</param>
    public DynamoDbWrapper(IAmazonDynamoDB amazonDynamoDB)
    {
        _amazonDynamoDB = amazonDynamoDB;
    }
```

ムービーデータを含めるテーブルを作成します。

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
 CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
```

```
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };

            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
```

```
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
 table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
 {tableName}. {ex.Message}");
            throw;
        }
    }
```

1 つのムービーをテーブルに追加します。

```
    /// <summary>
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
```

```
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };

            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
{ex.Message}");
            throw;
        }
    }
```

テーブルの 1 つの項目を更新します。

```
    /// <summary>
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
```

```
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };

            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
```

```
        {
            Console.WriteLine($"Table {tableName} or item was not found.
 {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
 item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while updating item.
 {ex.Message}");
            throw;
        }
    }
```

ムービーテーブルから 1 つの項目を取得します。

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
```

```
            var request = new GetItemRequest
            {
                Key = key,
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
{ex.Message}");
            throw;
        }
    }
```

項目のバッチをムービーテーブルに書き込みます。

```
    /// <summary>
    /// Loads the contents of a JSON file into a list of movies to be
    /// added to the DynamoDB table.
    /// </summary>
    /// <param name="movieFileName">The name of the JSON file.</param>
    /// <returns>A generic list of movie objects.</returns>
    public List<Movie> ImportMovies(string movieFileName)
    {
        var moviesList = new List<Movie>();
```

```
        if (!File.Exists(movieFileName))
        {
            return moviesList;
        }

        using var sr = new StreamReader(movieFileName);
        string json = sr.ReadToEnd();
        var allMovies = JsonSerializer.Deserialize<List<Movie>>(
            json,
            new JsonSerializerOptions
            {
                PropertyNameCaseInsensitive = true
            });

        // Now return the first 250 entries.
        if (allMovies != null && allMovies.Any())
        {
            moviesList = allMovies.GetRange(0, 250);
        }
        return moviesList;
    }

    /// <summary>
    /// Writes 250 items to the movie table.
    /// </summary>
    /// <param name="movieFileName">A string containing the full path to
    /// the JSON file containing movie data.</param>
    /// <param name="tableName">The name of the table to write items to.</param>
    /// <returns>A long integer value representing the number of movies
    /// imported from the JSON file.</returns>
    public async Task<long> BatchWriteItemsAsync(
        string movieFileName, string tableName)
    {
        try
        {
            var movies = ImportMovies(movieFileName);
            if (!movies.Any())
            {
                Console.WriteLine("Couldn't find the JSON file with movie
data.");
                return 0;
            }

            var context = new DynamoDBContextBuilder()
```

```
                // Optional call to provide a specific instance of
IAmazonDynamoDB
                .WithDynamoDBClient(() => _amazonDynamoDB)
                .Build();

            var movieBatch = context.CreateBatchWrite<Movie>(
                new BatchWriteConfig()
                {
                    OverrideTableName = tableName
                });
            movieBatch.AddPutItems(movies);

            Console.WriteLine("Adding imported movies to the table.");
            await movieBatch.ExecuteAsync();

            return movies.Count;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table was not found during batch write operation.
{ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred during batch
write operation. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred during batch write operation.
{ex.Message}");
            throw;
        }
    }
```

テーブルから 1 つの項目を削除します。

```
    /// <summary>
```

```
    /// Deletes a single item from a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table from which the item
    /// will be deleted.</param>
    /// <param name="movieToDelete">A movie object containing the title and
    /// year of the movie to delete.</param>
    /// <returns>A Boolean value indicating the success or failure of the
    /// delete operation.</returns>
    public async Task<bool> DeleteItemAsync(
        string tableName,
        Movie movieToDelete)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = movieToDelete.Title },
                ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
            };

            var request = new DeleteItemRequest { TableName = tableName, Key =
key, };

            await _amazonDynamoDB.DeleteItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting item.
{ex.Message}");
            throw;
        }
```

```
    }
```

特定の年にリリースされたムービーのテーブルにクエリを実行します。

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public async Task<int> QueryMoviesAsync(string tableName, int year)
{
    try
    {
        var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
            .AddHashKey("year", DynamoDBEntryType.Numeric)
            .AddRangeKey("title", DynamoDBEntryType.String)
            .Build();

        var filter = new QueryFilter("year", QueryOperator.Equal, year);

        Console.WriteLine("\nFind movies released in: {year}:");

        var config = new QueryOperationConfig()
        {
            Limit = 10, // 10 items per page.
            Select = SelectValues.SpecificAttributes,
            AttributesToGet = new List<string>
            {
                "title",
                "year",
            },
            ConsistentRead = true,
            Filter = filter,
        };

        // Value used to track how many movies match the
        // supplied criteria.
```

```
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while querying movies.
{ex.Message}");
            throw;
        }
    }
```

数年にわたってリリースされたムービーのテーブルをスキャンします。

```
    /// <summary>
    /// Scans the table for movies released between the specified years.
    /// </summary>
```

```
    /// <param name="tableName">The name of the table to scan.</param>
    /// <param name="startYear">The starting year for the range.</param>
    /// <param name="endYear">The ending year for the range.</param>
    /// <returns>The number of movies found in the specified year range.</
returns>
    public async Task<int> ScanTableAsync(
        string tableName,
        int startYear,
        int endYear)
    {
        try
        {
            var request = new ScanRequest
            {
                TableName = tableName,
                ExpressionAttributeNames = new Dictionary<string, string>
                {
                    { "#yr", "year" },
                },
                ExpressionAttributeValues = new Dictionary<string,
 AttributeValue>
                {
                    { ":y_a", new AttributeValue { N = startYear.ToString() } },
                    { ":y_z", new AttributeValue { N = endYear.ToString() } },
                },
                FilterExpression = "#yr between :y_a and :y_z",
                ProjectionExpression = "#yr, title, info.actors[0],
 info.directors, info.running_time_secs",
                Limit = 10 // Set a limit to demonstrate using the
 LastEvaluatedKey.
            };

            // Keep track of how many movies were found.
            int foundCount = 0;

            var response = new ScanResponse();
            do
            {
                response = await _amazonDynamoDB.ScanAsync(request);
                foundCount += response.Items.Count;
                response.Items.ForEach(i => DisplayItem(i));
                request.ExclusiveStartKey = response.LastEvaluatedKey;
            }
            while (response?.LastEvaluatedKey?.Count > 0);
```

```
            return foundCount;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while scanning
 table. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while scanning table.
 {ex.Message}");
            throw;
        }
    }
```

ムービーテーブルを削除します。

```
    /// <summary>
    /// Deletes a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);
```

```
            Console.WriteLine($"Table {response.TableDescription.TableName}
successfully deleted.");
            return true;

        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
{tableName}. {ex.Message}");
            return false;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDB の開始シナリオ。

```
###############################################################################
# function dynamodb_getting_started_movies
#
# Scenario to create an Amazon DynamoDB table and perform a series of operations
 on the table.
#
# Returns:
#       0 - If successful.
#       1 - If an error occurred.
###############################################################################
function dynamodb_getting_started_movies() {

  source ./dynamodb_operations.sh

  key_schema_json_file="dynamodb_key_schema.json"
  attribute_definitions_json_file="dynamodb_attr_def.json"
  item_json_file="movie_item.json"
  key_json_file="movie_key.json"
  batch_json_file="batch.json"
  attribute_names_json_file="attribute_names.json"
  attributes_values_json_file="attribute_values.json"

  echo_repeat "*" 88
  echo
  echo "Welcome to the Amazon DynamoDB getting started demo."
  echo
  echo_repeat "*" 88
  echo

  local table_name
  echo -n "Enter a name for a new DynamoDB table: "
```

```
  get_input
  table_name=$get_input_result

  echo '[
  {"AttributeName": "year", "KeyType": "HASH"},
   {"AttributeName": "title", "KeyType": "RANGE"}
  ]' >"$key_schema_json_file"

  echo '[
  {"AttributeName": "year", "AttributeType": "N"},
   {"AttributeName": "title", "AttributeType": "S"}
  ]' >"$attribute_definitions_json_file"

  if dynamodb_create_table -n "$table_name" -a "$attribute_definitions_json_file"
\
    -k "$key_schema_json_file" 1>/dev/null; then
    echo "Created a DynamoDB table named $table_name"
  else
    errecho "The table failed to create. This demo will exit."
    clean_up
    return 1
  fi

  echo "Waiting for the table to become active...."

  if dynamodb_wait_table_active -n "$table_name"; then
    echo "The table is now active."
  else
    errecho "The table failed to become active. This demo will exit."
    cleanup "$table_name"
    return 1
  fi

  echo
  echo_repeat "*" 88
  echo

  echo -n "Enter the title of a movie you want to add to the table: "
  get_input
  local added_title
  added_title=$get_input_result

  local added_year
  get_int_input "What year was it released? "
```

```
  added_year=$get_input_result

  local rating
  get_float_input "On a scale of 1 - 10, how do you rate it? " "1" "10"
  rating=$get_input_result

  local plot
  echo -n "Summarize the plot for me: "
  get_input
  plot=$get_input_result

  echo '{
    "year": {"N" :"'"$added_year"'"},
    "title": {"S" :  "'"$added_title"'"},
    "info": {"M" : {"plot": {"S" : "'"$plot"'"}, "rating":
{"N" :"'"$rating"'"} } }
   }' >"$item_json_file"

  if dynamodb_put_item -n "$table_name" -i "$item_json_file"; then
    echo "The movie '$added_title' was successfully added to the table
'$table_name'."
  else
    errecho "Put item failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi

  echo
  echo_repeat "*" 88
  echo

  echo "Let's update your movie '$added_title'."
  get_float_input "You rated it $rating, what new rating would you give it? " "1"
"10"
  rating=$get_input_result

  echo -n "You summarized the plot as '$plot'."
  echo "What would you say now? "
  get_input
  plot=$get_input_result

  echo '{
    "year": {"N" :"'"$added_year"'"},
    "title": {"S" :  "'"$added_title"'"}
```

```
    }' >"$key_json_file"

  echo '{
    ":r": {"N" :"'"$rating"'"},
    ":p": {"S" : "'"$plot"'"}
   }' >"$item_json_file"

  local update_expression="SET info.rating = :r, info.plot = :p"

  if dynamodb_update_item -n "$table_name" -k "$key_json_file" -e
"$update_expression" -v "$item_json_file"; then
    echo "Updated '$added_title' with new attributes."
  else
    errecho "Update item failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi

  echo
  echo_repeat "*" 88
  echo

  echo "We will now use batch write to upload 150 movie entries into the table."

  local batch_json
  for batch_json in movie_files/movies_*.json; do
    echo "{ \"$table_name\" : $(<"$batch_json") }" >"$batch_json_file"
    if dynamodb_batch_write_item -i "$batch_json_file" 1>/dev/null; then
      echo "Entries in $batch_json added to table."
    else
      errecho "Batch write failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi
  done

  local title="The Lord of the Rings: The Fellowship of the Ring"
  local year="2001"

  if get_yes_no_input "Let's move on...do you want to get info about '$title'?
(y/n) "; then
    echo '{
  "year": {"N" :"'"$year"'"},
  "title": {"S" :  "'"$title"'"}
```

```
  }' >"$key_json_file"
    local info
    info=$(dynamodb_get_item -n "$table_name" -k "$key_json_file")

    # shellcheck disable=SC2181
    if [[ ${?} -ne 0 ]]; then
      errecho "Get item failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi

    echo "Here is what I found:"
    echo "$info"
  fi

  local ask_for_year=true
  while [[ "$ask_for_year" == true ]]; do
    echo "Let's get a list of movies released in a given year."
    get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"
    year=$get_input_result
    echo '{
    "#n": "year"
    }' >"$attribute_names_json_file"

    echo '{
    ":v": {"N" :"'"$year"'"}
    }' >"$attributes_values_json_file"

    response=$(dynamodb_query -n "$table_name" -k "#n=:v" -a
"$attribute_names_json_file" -v "$attributes_values_json_file")

    # shellcheck disable=SC2181
    if [[ ${?} -ne 0 ]]; then
      errecho "Query table failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi

    echo "Here is what I found:"
    echo "$response"

    if ! get_yes_no_input "Try another year? (y/n) "; then
      ask_for_year=false
    fi
```

```
  done

  echo "Now let's scan for movies released in a range of years. Enter a year: "
  get_int_input "Enter a year between 1972 and 2018: " "1972" "2018"
  local start=$get_input_result

  get_int_input "Enter another year: " "1972" "2018"
  local end=$get_input_result

  echo '{
    "#n": "year"
    }' >"$attribute_names_json_file"

  echo '{
    ":v1": {"N" : "'"$start"'"},
    ":v2": {"N" : "'"$end"'"}
    }' >"$attributes_values_json_file"

  response=$(dynamodb_scan -n "$table_name" -f "#n BETWEEN :v1 AND :v2" -a
"$attribute_names_json_file" -v "$attributes_values_json_file")

  # shellcheck disable=SC2181
  if [[ ${?} -ne 0 ]]; then
    errecho "Scan table failed. This demo will exit."
    clean_up "$table_name"
    return 1
  fi

  echo "Here is what I found:"
  echo "$response"

  echo
  echo_repeat "*" 88
  echo

  echo "Let's remove your movie '$added_title' from the table."

  if get_yes_no_input "Do you want to remove '$added_title'? (y/n) "; then
    echo '{
  "year": {"N" :"'"$added_year"'"},
  "title": {"S" :  "'"$added_title"'"}
  }' >"$key_json_file"

    if ! dynamodb_delete_item -n "$table_name" -k "$key_json_file"; then
```

```
      errecho "Delete item failed. This demo will exit."
      clean_up "$table_name"
      return 1
    fi
  fi

  if get_yes_no_input "Do you want to delete the table '$table_name'? (y/n) ";
 then
    if ! clean_up "$table_name"; then
      return 1
    fi
  else
    if ! clean_up; then
      return 1
    fi
  fi

  return 0
}
```

このシナリオで使用される DynamoDB 関数。

```
###############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.
```

```
 ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_create_table"
   echo "Creates an Amazon DynamoDB table with on-demand billing."
   echo " -n table_name  -- The name of the table to create."
   echo " -a attribute_definitions -- JSON file path of a list of attributes and
their types."
   echo " -k key_schema -- JSON file path of a list of attributes and their key
types."
   echo ""
 }

 # Retrieve the calling parameters.
 while getopts "n:a:k:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     a) attribute_definitions="${OPTARG}" ;;
     k) key_schema="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$attribute_definitions" ]]; then
   errecho "ERROR: You must provide an attribute definitions json file path the
-a parameter."
   usage
   return 1
```

```
  fi

  if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
 parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi

  return 0
}

###############################################################################
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#
#  Response:
#       - TableStatus:
#     And:
#       0 - Table is active.
```

```
#       1 - If it fails.
###############################################################################
function dynamodb_describe_table {
  local table_name
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_describe_table"
    echo "Describe the status of a DynamoDB table."
    echo "  -n table_name  -- The name of the table."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  local table_status
    table_status=$(
      aws dynamodb describe-table \
        --table-name "$table_name" \
        --output text \
```

```
        --query 'Table.TableStatus'
    )

   local error_code=${?}

    if [[ $error_code -ne 0 ]]; then
      aws_cli_error_log "$error_code"
      errecho "ERROR: AWS reports describe-table operation failed.$table_status"
      return 1
    fi

  echo "$table_status"

  return 0
}

###############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }
```

```
  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
```

```
    return 1
  fi

  return 0

}

###############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_update_item"
    echo "Update an item in a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to update."
    echo " -e update expression  -- An expression that defines one or more
 attributes to be updated."
    echo " -v values  -- Path to json file containing the update values."
    echo ""
  }
```

```
while getopts "n:k:e:v:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    e) update_expression="${OPTARG}" ;;
    v) values="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?)
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi
if [[ -z "$update_expression" ]]; then
  errecho "ERROR: You must provide an update expression with the -e parameter."
  usage
  return 1
fi

if [[ -z "$values" ]]; then
  errecho "ERROR: You must provide a values json file path the -v parameter."
  usage
  return 1
fi

iecho "Parameters:\n"
iecho "    table_name:   $table_name"
```

```
  iecho "    keys:   $keys"
  iecho "    update_expression:   $update_expression"
  iecho "    values:   $values"

  response=$(aws dynamodb update-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --update-expression "$update_expression" \
    --expression-attribute-values file://"$values")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}

########################################################################
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
#       -i item  -- Path to json file containing the items to write.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
########################################################################
function dynamodb_batch_write_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
```

```
    echo " -i item  -- Path to json file containing the items to write."
    echo ""
  }
  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""

  response=$(aws dynamodb batch-write-item \
    --request-items file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  return 0
}
```

```
###############################################################################
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
  }
  query=""
  while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
```

```
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

if [[ -n "$query" ]]; then
  response=$(aws dynamodb get-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --output text \
    --query "$query")
else
  response=$(
    aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text
  )
fi

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
  aws_cli_error_log $error_code
  errecho "ERROR: AWS reports get-item operation failed.$response"
  return 1
fi

if [[ -n "$query" ]]; then
```

```
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
 query inserts on some strings.
  else
    echo "$response"
  fi

  return 0
}

###############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_query() {
  local table_name key_condition_expression attribute_names attribute_values
 projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_query"
    echo "Query a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k key_condition_expression -- The key condition expression."
    echo " -a attribute_names -- Path to JSON file containing the attribute
 names."
    echo " -v attribute_values -- Path to JSON file containing the attribute
 values."
```

```
    echo " [-p projection_expression]  -- Optional projection expression."
    echo ""
  }

  while getopts "n:k:a:v:p:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) key_condition_expression="${OPTARG}" ;;
      a) attribute_names="${OPTARG}" ;;
      v) attribute_values="${OPTARG}" ;;
      p) projection_expression="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$key_condition_expression" ]]; then
    errecho "ERROR: You must provide a key condition expression with the -k
parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_values" ]]; then
```

```
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values")
  else
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}

###############################################################################
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -f filter_expression  -- The filter expression.
#       -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names.
```

```
#       -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_scan() {
  local table_name filter_expression expression_attribute_names
 expression_attribute_values projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_scan"
    echo "Scan a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -f filter_expression  -- The filter expression."
    echo " -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names."
    echo " -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values."
    echo " [-p projection_expression]  -- Optional projection expression."
    echo ""
  }

  while getopts "n:f:a:v:p:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      f) filter_expression="${OPTARG}" ;;
      a) expression_attribute_names="${OPTARG}" ;;
      v) expression_attribute_values="${OPTARG}" ;;
      p) projection_expression="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
```

```
        usage
        return 1
        ;;
    esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$filter_expression" ]]; then
   errecho "ERROR: You must provide a filter expression with the -f parameter."
   usage
   return 1
 fi

 if [[ -z "$expression_attribute_names" ]]; then
   errecho "ERROR: You must provide expression attribute names with the -a
parameter."
   usage
   return 1
 fi

 if [[ -z "$expression_attribute_values" ]]; then
   errecho "ERROR: You must provide expression attribute values with the -v
parameter."
   usage
   return 1
 fi

 if [[ -z "$projection_expression" ]]; then
   response=$(aws dynamodb scan \
     --table-name "$table_name" \
     --filter-expression "$filter_expression" \
     --expression-attribute-names file://"$expression_attribute_names" \
     --expression-attribute-values file://"$expression_attribute_values")
 else
   response=$(aws dynamodb scan \
     --table-name "$table_name" \
     --filter-expression "$filter_expression" \
     --expression-attribute-names file://"$expression_attribute_names" \
```

```
      --expression-attribute-values file://"$expression_attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}

###############################################################################
# function dynamodb_delete_item
#
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_item() {
  local table_name keys response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_delete_item"
    echo "Delete an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to delete."
```

```
  echo ""
}
while getopts "n:k:h" option; do
  case "${option}" in
    n) table_name="${OPTARG}" ;;
    k) keys="${OPTARG}" ;;
    h)
      usage
      return 0
      ;;
    \?)
      echo "Invalid parameter"
      usage
      return 1
      ;;
  esac
done
export OPTIND=1

if [[ -z "$table_name" ]]; then
  errecho "ERROR: You must provide a table name with the -n parameter."
  usage
  return 1
fi

if [[ -z "$keys" ]]; then
  errecho "ERROR: You must provide a keys json file path the -k parameter."
  usage
  return 1
fi

iecho "Parameters:\n"
iecho "    table_name:   $table_name"
iecho "    keys:   $keys"
iecho ""

response=$(aws dynamodb delete-item \
  --table-name "$table_name" \
  --key file://"$keys")

local error_code=${?}

if [[ $error_code -ne 0 ]]; then
  aws_cli_error_log $error_code
```

```
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
  fi

  return 0

}

###############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
```

```
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""

  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

このシナリオで使用されるユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}
```

```
##############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
##############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

##############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
##############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi
```

```
  return 0
}
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    {
        Aws::Client::ClientConfiguration clientConfig;
        //  1. Create a table with partition: year (N) and sort: title (S).
(CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::dynamodbGettingStartedScenario(clientConfig);

            // 9. Delete the table. (DeleteTable)
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
```

```
        }
    }

//! Scenario to modify and query a DynamoDB table.
/*!
  \sa dynamodbGettingStartedScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::dynamodbGettingStartedScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;
    std::cout << "Welcome to the Amazon DynamoDB getting started demo." <<
 std::endl;
    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;

    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie.
    Aws::String title;
    float rating;
    int year;
    Aws::String plot;
    {
        title = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        year = askQuestionForInt("What year was it released? ");
        rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
 it? ",
                                          1, 10);
        plot = askQuestion("Summarize the plot for me: ");

        Aws::DynamoDB::Model::PutItemRequest putItemRequest;
        putItemRequest.SetTableName(MOVIE_TABLE_NAME);

        putItemRequest.AddItem(YEAR_KEY,

 Aws::DynamoDB::Model::AttributeValue().SetN(year));
        putItemRequest.AddItem(TITLE_KEY,

 Aws::DynamoDB::Model::AttributeValue().SetS(title));
```

```
        // Create attribute for the info map.
        Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        ratingAttribute->SetN(rating);
        infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        plotAttribute->SetS(plot);
        infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);

        putItemRequest.AddItem(INFO_KEY, infoMapAttribute);

        Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
                putItemRequest);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add an item: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
              << std::endl;

    // 3. Update the rating and plot of the movie by using an update expression.
    {
        rating = askQuestionForFloatRange(
                Aws::String("\nLet's update your movie.\nYou rated it  ") +
                std::to_string(rating)
                + ", what new rating would you give it? ", 1, 10);
        plot = askQuestion(Aws::String("You summarized the plot as '") + plot +
                           "'.\nWhat would you say now? ");

        Aws::DynamoDB::Model::UpdateItemRequest request;
        request.SetTableName(MOVIE_TABLE_NAME);
        request.AddKey(TITLE_KEY,
Aws::DynamoDB::Model::AttributeValue().SetS(title));
```

```
        request.AddKey(YEAR_KEY,
Aws::DynamoDB::Model::AttributeValue().SetN(year));
        std::stringstream expressionStream;
        expressionStream << "set " << INFO_KEY << "." << RATING_KEY << " =:r, "
                         << INFO_KEY << "." << PLOT_KEY << " =:p";
        request.SetUpdateExpression(expressionStream.str());
        request.SetExpressionAttributeValues({
                                                 {":r",
Aws::DynamoDB::Model::AttributeValue().SetN(
                                                         rating)},
                                                 {":p",
Aws::DynamoDB::Model::AttributeValue().SetS(
                                                         plot)}
                                             });

        request.SetReturnValues(Aws::DynamoDB::Model::ReturnValue::UPDATED_NEW);

        const Aws::DynamoDB::Model::UpdateItemOutcome &result =
dynamoClient.UpdateItem(
                request);
        if (!result.IsSuccess()) {
            std::cerr << "Error updating movie " + result.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;

    // 4. Put 250 movies in the table from moviedata.json.
    {
        std::cout << "Adding movies from a json file to the database." <<
std::endl;
        const size_t MAX_SIZE_FOR_BATCH_WRITE = 25;
        const size_t MOVIES_TO_WRITE = 10 * MAX_SIZE_FOR_BATCH_WRITE;
        Aws::String jsonString = getMovieJSON();
        if (!jsonString.empty()) {
            Aws::Utils::Json::JsonValue json(jsonString);
            Aws::Utils::Array<Aws::Utils::Json::JsonView> movieJsons =
json.View().AsArray();
            Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;

            // To add movies with a cross-section of years, use an appropriate
increment
```

```
            // value for iterating through the database.
            size_t increment = movieJsons.GetLength() / MOVIES_TO_WRITE;
            for (size_t i = 0; i < movieJsons.GetLength(); i += increment) {
                writeRequests.push_back(Aws::DynamoDB::Model::WriteRequest());
                Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
putItems = movieJsonViewToAttributeMap(
                        movieJsons[i]);
                Aws::DynamoDB::Model::PutRequest putRequest;
                putRequest.SetItem(putItems);
                writeRequests.back().SetPutRequest(putRequest);
                if (writeRequests.size() == MAX_SIZE_FOR_BATCH_WRITE) {
                    Aws::DynamoDB::Model::BatchWriteItemRequest request;
                    request.AddRequestItems(MOVIE_TABLE_NAME, writeRequests);
                    const Aws::DynamoDB::Model::BatchWriteItemOutcome &outcome =
dynamoClient.BatchWriteItem(
                            request);
                    if (!outcome.IsSuccess()) {
                        std::cerr << "Unable to batch write movie data: "
                                  << outcome.GetError().GetMessage()
                                  << std::endl;
                        writeRequests.clear();
                        break;
                    }
                    else {
                        std::cout << "Added batch of " << writeRequests.size()
                                  << " movies to the database."
                                  << std::endl;
                    }
                    writeRequests.clear();
                }
            }
        }
    }

    std::cout << std::setfill('*') << std::setw(ASTERISK_FILL_WIDTH) << " "
              << std::endl;

    // 5. Get a movie by Key (partition + sort).
    {
        Aws::String titleToGet("King Kong");
        Aws::String answer = askQuestion(Aws::String(
                "Let's move on...Would you like to get info about '" + titleToGet
+
                "'? (y/n) "));
```

```
        if (answer == "y") {
            Aws::DynamoDB::Model::GetItemRequest request;
            request.SetTableName(MOVIE_TABLE_NAME);
            request.AddKey(TITLE_KEY,

Aws::DynamoDB::Model::AttributeValue().SetS(titleToGet));
            request.AddKey(YEAR_KEY,
Aws::DynamoDB::Model::AttributeValue().SetN(1933));

            const Aws::DynamoDB::Model::GetItemOutcome &result =
dynamoClient.GetItem(
                    request);
            if (!result.IsSuccess()) {
                std::cerr << "Error " << result.GetError().GetMessage();
            }
            else {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = result.GetResult().GetItem();
                if (!item.empty()) {
                    std::cout << "\nHere's what I found:" << std::endl;
                    printMovieInfo(item);
                }
                else {
                    std::cout << "\nThe movie was not found in the database."
                              << std::endl;
                }
            }
        }
    }

    // 6. Use Query with a key condition expression to return all movies
    //    released in a given year.
    Aws::String doAgain = "n";
    do {
        Aws::DynamoDB::Model::QueryRequest req;

        req.SetTableName(MOVIE_TABLE_NAME);

        // "year" is a DynamoDB reserved keyword and must be replaced with an
        // expression attribute name.
        req.SetKeyConditionExpression("#dynobase_year = :valueToMatch");
        req.SetExpressionAttributeNames({{"#dynobase_year", YEAR_KEY}});

        int yearToMatch = askQuestionForIntRange(
```

```
                "\nLet's get a list of movies released in"
                " a given year. Enter a year between 1972 and 2018 ",
                1972, 2018);
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
        attributeValues.emplace(":valueToMatch",
                                Aws::DynamoDB::Model::AttributeValue().SetN(
                                        yearToMatch));
        req.SetExpressionAttributeValues(attributeValues);

        const Aws::DynamoDB::Model::QueryOutcome &result =
dynamoClient.Query(req);
        if (result.IsSuccess()) {
            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
            if (!items.empty()) {
                std::cout << "\nThere were " << items.size()
                          << " movies in the database from "
                          << yearToMatch << "." << std::endl;
                for (const auto &item: items) {
                    printMovieInfo(item);
                }
                doAgain = "n";
            }
            else {
                std::cout << "\nNo movies from " << yearToMatch
                          << " were found in the database"
                          << std::endl;
                doAgain = askQuestion(Aws::String("Try another year? (y/n) "));
            }
        }
        else {
            std::cerr << "Failed to Query items: " <<
result.GetError().GetMessage()
                      << std::endl;
        }

    } while (doAgain == "y");

    //  7. Use Scan to return movies released within a range of years.
    //     Show how to paginate data using ExclusiveStartKey. (Scan +
FilterExpression)
    {
```

```
        int startYear = askQuestionForIntRange("\nNow let's scan a range of years
"
                                               "for movies in the database. Enter
a start year: ",
                                               1972, 2018);
        int endYear = askQuestionForIntRange("\nEnter an end year: ",
                                             startYear, 2018);
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
        do {
            Aws::DynamoDB::Model::ScanRequest scanRequest;
            scanRequest.SetTableName(MOVIE_TABLE_NAME);
            scanRequest.SetFilterExpression(
                    "#dynobase_year >= :startYear AND #dynobase_year
<= :endYear");
            scanRequest.SetExpressionAttributeNames({{"#dynobase_year",
YEAR_KEY}});

            Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
attributeValues;
            attributeValues.emplace(":startYear",
                                    Aws::DynamoDB::Model::AttributeValue().SetN(
                                            startYear));
            attributeValues.emplace(":endYear",
                                    Aws::DynamoDB::Model::AttributeValue().SetN(
                                            endYear));
            scanRequest.SetExpressionAttributeValues(attributeValues);

            if (!exclusiveStartKey.empty()) {
                scanRequest.SetExclusiveStartKey(exclusiveStartKey);
            }

            const Aws::DynamoDB::Model::ScanOutcome &result = dynamoClient.Scan(
                    scanRequest);
            if (result.IsSuccess()) {
                const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetResult().GetItems();
                if (!items.empty()) {
                    std::stringstream stringStream;
                    stringStream << "\nFound " << items.size() << " movies in one
scan."
                                 << " How many would you like to see? ";
                    size_t count = askQuestionForInt(stringStream.str());
                    for (size_t i = 0; i < count && i < items.size(); ++i) {
```

```
                        printMovieInfo(items[i]);
                    }
                }
                else {
                    std::cout << "\nNo movies in the database between " <<
startYear <<
                              " and " << endYear << "." << std::endl;
                }

                exclusiveStartKey = result.GetResult().GetLastEvaluatedKey();
                if (!exclusiveStartKey.empty()) {
                    std::cout << "Not all movies were retrieved. Scanning for
more."
                              << std::endl;
                }
                else {
                    std::cout << "All movies were retrieved with this scan."
                              << std::endl;
                }
            }
            else {
                std::cerr << "Failed to Scan movies: "
                          << result.GetError().GetMessage() << std::endl;
            }
        } while (!exclusiveStartKey.empty());
    }

    // 8. Delete a movie. (DeleteItem)
    {
        std::stringstream stringStream;
        stringStream << "\nWould you like to delete the movie " << title
                     << " from the database? (y/n) ";
        Aws::String answer = askQuestion(stringStream.str());
        if (answer == "y") {
            Aws::DynamoDB::Model::DeleteItemRequest request;
            request.AddKey(YEAR_KEY,
Aws::DynamoDB::Model::AttributeValue().SetN(year));
            request.AddKey(TITLE_KEY,
                           Aws::DynamoDB::Model::AttributeValue().SetS(title));
            request.SetTableName(MOVIE_TABLE_NAME);

            const Aws::DynamoDB::Model::DeleteItemOutcome &result =
dynamoClient.DeleteItem(
                    request);
```

```
            if (result.IsSuccess()) {
                std::cout << "\nRemoved \"" << title << "\" from the database."
                          << std::endl;
            }
            else {
                std::cerr << "Failed to delete the movie: "
                          << result.GetError().GetMessage()
                          << std::endl;
            }
        }
    }

    return true;
}

//! Routine to convert a JsonView object to an attribute map.
/*!
  \sa movieJsonViewToAttributeMap()
  \param jsonView: Json view object.
  \return map: Map that can be used in a DynamoDB request.
 */
Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
AwsDoc::DynamoDB::movieJsonViewToAttributeMap(
        const Aws::Utils::Json::JsonView &jsonView) {
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> result;

    if (jsonView.KeyExists(YEAR_KEY)) {
        result[YEAR_KEY].SetN(jsonView.GetInteger(YEAR_KEY));
    }
    if (jsonView.KeyExists(TITLE_KEY)) {
        result[TITLE_KEY].SetS(jsonView.GetString(TITLE_KEY));
    }
    if (jsonView.KeyExists(INFO_KEY)) {
        Aws::Map<Aws::String, const
 std::shared_ptr<Aws::DynamoDB::Model::AttributeValue>> infoMap;
        Aws::Utils::Json::JsonView infoView = jsonView.GetObject(INFO_KEY);
        if (infoView.KeyExists(RATING_KEY)) {
            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> attributeValue
 = std::make_shared<Aws::DynamoDB::Model::AttributeValue>();
            attributeValue->SetN(infoView.GetDouble(RATING_KEY));
            infoMap.emplace(std::make_pair(RATING_KEY, attributeValue));
        }
        if (infoView.KeyExists(PLOT_KEY)) {
```

```
            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> attributeValue
 = std::make_shared<Aws::DynamoDB::Model::AttributeValue>();
            attributeValue->SetS(infoView.GetString(PLOT_KEY));
            infoMap.emplace(std::make_pair(PLOT_KEY, attributeValue));
        }

        result[INFO_KEY].SetM(infoMap);
    }

    return result;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
        request.AddKeySchema(yearKeySchema);
```

```
        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
                std::cerr << "Failed to create table: "
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
                  << std::endl;
    }

    return true;
```

```
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
```

```
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query

- Scan
- UpdateItem

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

対話型シナリオを実行してテーブルを作成し、そのテーブルに対してアクションを実行します。

```
import (
 "context"
 "fmt"
 "log"
 "strings"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunMovieScenario is an interactive example that shows you how to use the AWS
 SDK for Go
// to create and use an Amazon DynamoDB table that stores data about movies.
//
//  1. Create a table that can hold movie data.
//  2. Put, get, and update a single movie in the table.
//  3. Write movie data to the table from a sample JSON file.
//  4. Query for movies that were released in a given year.
//  5. Scan for movies that were released in a range of years.
//  6. Delete a movie from the table.
//  7. Delete the table.
//
```

```
// This example creates a DynamoDB service client from the specified sdkConfig so
 that
// you can replace it with a mocked or stubbed config for unit testing.
//
// It uses a questioner from the `demotools` package to get input during the
 example.
// This package can be found in the ..\..\demotools folder of this repo.
//
// The specified movie sampler is used to get sample data from a URL that is
 loaded
// into the named table.
func RunMovieScenario(
 ctx context.Context, sdkConfig aws.Config, questioner demotools.IQuestioner,
 tableName string,
 movieSampler actions.IMovieSampler) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon DynamoDB getting started demo.")
 log.Println(strings.Repeat("-", 88))

 tableBasics := actions.TableBasics{TableName: tableName,
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig)}

 exists, err := tableBasics.TableExists(ctx)
 if err != nil {
  panic(err)
 }
 if !exists {
  log.Printf("Creating table %v...\n", tableName)
  _, err = tableBasics.CreateMovieTable(ctx)
  if err != nil {
   panic(err)
  } else {
   log.Printf("Created table %v.\n", tableName)
  }
 } else {
  log.Printf("Table %v already exists.\n", tableName)
 }
```

```
var customMovie actions.Movie
customMovie.Title = questioner.Ask("Enter a movie title to add to the table:",
 demotools.NotEmpty{})
customMovie.Year = questioner.AskInt("What year was it released?",
 demotools.NotEmpty{}, demotools.InIntRange{Lower: 1900, Upper: 2030})
customMovie.Info = map[string]interface{}{}
customMovie.Info["rating"] = questioner.AskFloat64(
 "Enter a rating between 1 and 10:",
 demotools.NotEmpty{}, demotools.InFloatRange{Lower: 1, Upper: 10})
customMovie.Info["plot"] = questioner.Ask("What's the plot? ",
 demotools.NotEmpty{})
err = tableBasics.AddMovie(ctx, customMovie)
if err == nil {
 log.Printf("Added %v to the movie table.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Let's update your movie. You previously rated it %v.\n",
customMovie.Info["rating"])
customMovie.Info["rating"] = questioner.AskFloat64(
 "What new rating would you give it?",
 demotools.NotEmpty{}, demotools.InFloatRange{Lower: 1, Upper: 10})
log.Printf("You summarized the plot as '%v'.\n", customMovie.Info["plot"])
customMovie.Info["plot"] = questioner.Ask("What would you say now?",
 demotools.NotEmpty{})
attributes, err := tableBasics.UpdateMovie(ctx, customMovie)
if err == nil {
 log.Printf("Updated %v with new values.\n", customMovie.Title)
 for _, attVal := range attributes {
  for valKey, val := range attVal {
   log.Printf("\t%v: %v\n", valKey, val)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting movie data from %v and adding 250 movies to the table...\n",
 movieSampler.GetURL())
movies := movieSampler.GetSampleMovies()
written, err := tableBasics.AddMovieBatch(ctx, movies, 250)
if err != nil {
 panic(err)
} else {
 log.Printf("Added %v movies to the table.\n", written)
```

```
}

show := 10
if show > written {
 show = written
}
log.Printf("The first %v movies in the table are:", show)
for index, movie := range movies[:show] {
 log.Printf("\t%v. %v\n", index+1, movie.Title)
}
movieIndex := questioner.AskInt(
 "Enter the number of a movie to get info about it: ",
 demotools.InIntRange{Lower: 1, Upper: show},
)
movie, err := tableBasics.GetMovie(ctx, movies[movieIndex-1].Title,
movies[movieIndex-1].Year)
if err == nil {
 log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

log.Println("Let's get a list of movies released in a given year.")
releaseYear := questioner.AskInt("Enter a year between 1972 and 2018: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018},
)
releases, err := tableBasics.Query(ctx, releaseYear)
if err == nil {
 if len(releases) == 0 {
  log.Printf("I couldn't find any movies released in %v!\n", releaseYear)
 } else {
  for _, movie = range releases {
   log.Println(movie)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Println("Now let's scan for movies released in a range of years.")
startYear := questioner.AskInt("Enter a year: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018})
endYear := questioner.AskInt("Enter another year: ",
 demotools.InIntRange{Lower: 1972, Upper: 2018})
releases, err = tableBasics.Scan(ctx, startYear, endYear)
if err == nil {
```

```
 if len(releases) == 0 {
  log.Printf("I couldn't find any movies released between %v and %v!\n",
startYear, endYear)
 } else {
  log.Printf("Found %v movies. In this list, the plot is <nil> because "+
   "we used a projection expression when scanning for items to return only "+
   "the title, year, and rating.\n", len(releases))
  for _, movie = range releases {
   log.Println(movie)
  }
 }
}
log.Println(strings.Repeat("-", 88))

var tables []string
if questioner.AskBool("Do you want to list all of your tables? (y/n) ", "y") {
 tables, err = tableBasics.ListTables(ctx)
 if err == nil {
  log.Printf("Found %v tables:", len(tables))
  for _, table := range tables {
   log.Printf("\t%v", table)
  }
 }
}
log.Println(strings.Repeat("-", 88))

log.Printf("Let's remove your movie '%v'.\n", customMovie.Title)
if questioner.AskBool("Do you want to delete it from the table? (y/n) ", "y") {
 err = tableBasics.DeleteMovie(ctx, customMovie)
}
if err == nil {
 log.Printf("Deleted %v.\n", customMovie.Title)
}

if questioner.AskBool("Delete the table, too? (y/n)", "y") {
 err = tableBasics.DeleteTable(ctx)
} else {
 log.Println("Don't forget to delete the table when you're done or you might " +
  "incur charges on your account.")
}
if err == nil {
 log.Printf("Deleted table %v.\n", tableBasics.TableName)
}
```

```
 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
```

```
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

DynamoDB アクションを呼び出す構造体およびメソッドを作成します。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists(ctx context.Context) (bool, error) {
```

```
 exists := true
 _, err := basics.DynamoDbClient.DescribeTable(
  ctx, &dynamodb.DescribeTableInput{TableName: aws.String(basics.TableName)},
 )
 if err != nil {
  var notFoundEx *types.ResourceNotFoundException
  if errors.As(err, &notFoundEx) {
   log.Printf("Table %v does not exist.\n", basics.TableName)
   err = nil
  } else {
   log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
 basics.TableName, err)
  }
  exists = false
 }
 return exists, err
}



// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
  TableName:   aws.String(basics.TableName),
```

```
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}



// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables(ctx context.Context) ([]string, error) {
 var tableNames []string
 var output *dynamodb.ListTablesOutput
 var err error
 tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
 &dynamodb.ListTablesInput{})
 for tablePaginator.HasMorePages() {
  output, err = tablePaginator.NextPage(ctx)
  if err != nil {
   log.Printf("Couldn't list tables. Here's why: %v\n", err)
   break
  } else {
   tableNames = append(tableNames, output.TableNames...)
  }
 }
 return tableNames, err
}



// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
```

```
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}



// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
```

```
    }
   }
  }
  return attributeMap, err
}



// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
func (basics TableBasics) AddMovieBatch(ctx context.Context, movies []Movie,
 maxMovies int) (int, error) {
 var err error
 var item map[string]types.AttributeValue
 written := 0
 batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
 start := 0
 end := start + batchSize
 for start < maxMovies && start < len(movies) {
  var writeReqs []types.WriteRequest
  if end > len(movies) {
   end = len(movies)
  }
  for _, movie := range movies[start:end] {
   item, err = attributevalue.MarshalMap(movie)
   if err != nil {
    log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
 movie.Title, err)
   } else {
    writeReqs = append(
     writeReqs,
     types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
    )
   }
  }
  _, err = basics.DynamoDbClient.BatchWriteItem(ctx,
 &dynamodb.BatchWriteItemInput{
   RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}})
  if err != nil {
   log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
 basics.TableName, err)
  } else {
   written += len(writeReqs)
```

```
  }
  start = end
  end += batchSize
 }

 return written, err
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
```

```
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}



// Scan gets all movies in the DynamoDB table that were released in a range of
 years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(ctx context.Context, startYear int, endYear int)
 ([]Movie, error) {
 var movies []Movie
 var err error
 var response *dynamodb.ScanOutput
```

```
 filtEx := expression.Name("year").Between(expression.Value(startYear),
 expression.Value(endYear))
 projEx := expression.NamesList(
  expression.Name("year"), expression.Name("title"),
 expression.Name("info.rating"))
 expr, err :=
 expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
 if err != nil {
  log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
 } else {
  scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
 &dynamodb.ScanInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   FilterExpression:          expr.Filter(),
   ProjectionExpression:      expr.Projection(),
  })
  for scanPaginator.HasMorePages() {
   response, err = scanPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't scan for movies released between %v and %v. Here's why:
 %v\n",
     startYear, endYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}



// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(ctx context.Context, movie Movie) error {
```

```
 _, err := basics.DynamoDbClient.DeleteItem(ctx, &dynamodb.DeleteItemInput{
  TableName: aws.String(basics.TableName), Key: movie.GetKey(),
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
 return err
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDB テーブルを作成します。

```
// Create a table with a Sort key.
public static void createTable(DynamoDbClient ddb, String tableName) {
    DynamoDbWaiter dbWaiter = ddb.waiter();
    ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

    // Define attributes.
    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("year")
        .attributeType("N")
        .build());

    attributeDefinitions.add(AttributeDefinition.builder()
        .attributeName("title")
        .attributeType("S")
        .build());

    ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
    KeySchemaElement key = KeySchemaElement.builder()
        .attributeName("year")
        .keyType(KeyType.HASH)
        .build();

    KeySchemaElement key2 = KeySchemaElement.builder()
        .attributeName("title")
        .keyType(KeyType.RANGE)
        .build();

    // Add KeySchemaElement objects to the list.
    tableKey.add(key);
    tableKey.add(key2);
```

```
        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

サンプルの JSON ファイルをダウンロードして抽出するヘルパー関数を作成します。

```
    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();

        DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
```

```
        ObjectNode currentNode;
        int t = 0;
        while (iter.hasNext()) {
            // Only add 200 Movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            Movies movies = new Movies();
            movies.setYear(year);
            movies.setTitle(title);
            movies.setInfo(info);

            // Put the data into the Amazon DynamoDB Movie table.
            mappedTable.putItem(movies);
            t++;
        }
    }
```

テーブルから項目を取得します。

```
    public static void getItem(DynamoDbClient ddb) {

        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put("year", AttributeValue.builder()
            .n("1933")
            .build());

        keyToGet.put("title", AttributeValue.builder()
            .s("King Kong")
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
            .tableName("Movies")
            .build();

        try {
```

```
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();

            if (returnedItem != null) {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");

                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            } else {
                System.out.format("No item found with the key %s!\n", "year");
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

完全な例です。

```
/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 * <p>
 * This Java example performs these tasks:
 * <p>
 * 1. Creates the Amazon DynamoDB Movie table with partition and sort key.
 * 2. Puts data into the Amazon DynamoDB table from a JSON document using the
 * Enhanced client.
 * 3. Gets data from the Movie table.
 * 4. Adds a new item.
 * 5. Updates an item.
 * 6. Uses a Scan to query items using the Enhanced client.
```

```
 * 7. Queries all items where the year is 2013 using the Enhanced Client.
 * 8. Deletes the table.
 */

public class Scenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws IOException {
        String tableName = "Movies";
        String fileName = "../../../resources/sample_files/movies.json";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the Amazon DynamoDB example scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println(
            "1. Creating an Amazon DynamoDB table named Movies with a key named
 year and a sort key named title.");
        createTable(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Loading data into the Amazon DynamoDB table.");
        loadData(ddb, tableName, fileName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. Getting data from the Movie table.");
        getItem(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Putting a record into the Amazon DynamoDB
 table.");
        putRecord(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
```

```
        System.out.println("5. Updating a record.");
        updateTableItem(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Scanning the Amazon DynamoDB table.");
        scanMovies(ddb, tableName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Querying the Movies released in 2013.");
        queryTable(ddb);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("8. Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        System.out.println(DASHES);

        ddb.close();
    }

    // Create a table with a Sort key.
    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());

        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();
```

```
        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE)
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Query the table.
    public static void queryTable(DynamoDbClient ddb) {
        try {
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
```

```
            DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
            QueryConditional queryConditional = QueryConditional
                .keyEqualTo(Key.builder()
                    .partitionValue(2013)
                    .build());

            // Get items in the table and write out the ID value.
            Iterator<Movies> results =
custTable.query(queryConditional).items().iterator();
            String result = "";

            while (results.hasNext()) {
                Movies rec = results.next();
                System.out.println("The title of the movie is " +
rec.getTitle());
                System.out.println("The movie information  is " + rec.getInfo());
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Scan the table.
    public static void scanMovies(DynamoDbClient ddb, String tableName) {
        System.out.println("******* Scanning all movies.\n");
        try {
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

            DynamoDbTable<Movies> custTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
            Iterator<Movies> results = custTable.scan().items().iterator();
            while (results.hasNext()) {
                Movies rec = results.next();
                System.out.println("The movie title is " + rec.getTitle());
                System.out.println("The movie year is " + rec.getYear());
            }

        } catch (DynamoDbException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String tableName, String
fileName) throws IOException {
        DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
            .dynamoDbClient(ddb)
            .build();

        DynamoDbTable<Movies> mappedTable = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
        ObjectNode currentNode;
        int t = 0;
        while (iter.hasNext()) {
            // Only add 200 Movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            Movies movies = new Movies();
            movies.setYear(year);
            movies.setTitle(title);
            movies.setInfo(info);

            // Put the data into the Amazon DynamoDB Movie table.
            mappedTable.putItem(movies);
            t++;
        }
    }

    // Update the record to include show only directors.
    public static void updateTableItem(DynamoDbClient ddb, String tableName) {
        HashMap<String, AttributeValue> itemKey = new HashMap<>();
```

```
        itemKey.put("year", AttributeValue.builder().n("1933").build());
        itemKey.put("title", AttributeValue.builder().s("King Kong").build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put("info", AttributeValueUpdate.builder()
            .value(AttributeValue.builder().s("{\"directors\":[\"Merian C. Cooper
\",\"Ernest B. Schoedsack\"]")
                .build())
            .action(AttributeAction.PUT)
            .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(itemKey)
            .attributeUpdates(updatedValues)
            .build();

        try {
            ddb.updateItem(request);
        } catch (ResourceNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        System.out.println("Item was updated!");
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
 {
        DeleteTableRequest request = DeleteTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
```

```
    }

    public static void putRecord(DynamoDbClient ddb) {
        try {
            DynamoDbEnhancedClient enhancedClient =
DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();

            DynamoDbTable<Movies> table = enhancedClient.table("Movies",
TableSchema.fromBean(Movies.class));

            // Populate the Table.
            Movies record = new Movies();
            record.setYear(2020);
            record.setTitle("My Movie2");
            record.setInfo("no info");
            table.putItem(record);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Added a new movie to the table.");
    }

    public static void getItem(DynamoDbClient ddb) {

        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put("year", AttributeValue.builder()
            .n("1933")
            .build());

        keyToGet.put("title", AttributeValue.builder()
            .s("King Kong")
            .build());

        GetItemRequest request = GetItemRequest.builder()
            .key(keyToGet)
            .tableName("Movies")
            .build();

        try {
```

```
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();

            if (returnedItem != null) {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");

                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            } else {
                System.out.format("No item found with the key %s!\n", "year");
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { readFileSync } from "node:fs";
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";

/**
 * This module is a convenience library. It abstracts Amazon DynamoDB's data type
 * descriptors (such as S, N, B, and BOOL) by marshalling JavaScript objects into
 * AttributeValue shapes.
 */
import {
  BatchWriteCommand,
  DeleteCommand,
  DynamoDBDocumentClient,
  GetCommand,
  PutCommand,
  UpdateCommand,
  paginateQuery,
  paginateScan,
} from "@aws-sdk/lib-dynamodb";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";
```

```
const dirname = dirnameFromMetaUrl(import.meta.url);
const tableName = getUniqueName("Movies");
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);

export const main = async () => {
  /**
   * Create a table.
   */

  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "year",
        // 'N' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "N",
      },
      { AttributeName: "title", AttributeType: "S" },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [
      // The way your data is accessed determines how you structure your keys.
      // The movies table will be queried for movies by year. It makes sense
      // to make year our partition (HASH) key.
      { AttributeName: "year", KeyType: "HASH" },
      { AttributeName: "title", KeyType: "RANGE" },
    ],
  });
```

```
  log("Creating a table.");
  const createTableResponse = await client.send(createTableCommand);
  log(`Table created: ${JSON.stringify(createTableResponse.TableDescription)}`);

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");

  /**
   * Add a movie to the table.
   */

  log("Adding a single movie to the table.");
  // PutCommand is the first example usage of 'lib-dynamodb'.
  const putCommand = new PutCommand({
    TableName: tableName,
    Item: {
      // In 'client-dynamodb', the AttributeValue would be required (`year: { N:
1981 }`)
      // 'lib-dynamodb' simplifies the usage ( `year: 1981` )
      year: 1981,
      // The preceding KeySchema defines 'title' as our sort (RANGE) key, so
'title'
      // is required.
      title: "The Evil Dead",
      // Every other attribute is optional.
      info: {
        genres: ["Horror"],
      },
    },
  });
  await docClient.send(putCommand);
  log("The movie was added.");

  /**
   * Get a movie from the table.
   */

  log("Getting a single movie from the table.");
  const getCommand = new GetCommand({
    TableName: tableName,
    // Requires the complete primary key. For the movies table, the primary key
```

```
    // is only the id (partition key).
    Key: {
      year: 1981,
      title: "The Evil Dead",
    },
    // Set this to make sure that recent writes are reflected.
    // For more information, see https://docs.aws.amazon.com/amazondynamodb/
latest/developerguide/HowItWorks.ReadConsistency.html.
    ConsistentRead: true,
  });
  const getResponse = await docClient.send(getCommand);
  log(`Got the movie: ${JSON.stringify(getResponse.Item)}`);

  /**
   * Update a movie in the table.
   */

  log("Updating a single movie in the table.");
  const updateCommand = new UpdateCommand({
    TableName: tableName,
    Key: { year: 1981, title: "The Evil Dead" },
    // This update expression appends "Comedy" to the list of genres.
    // For more information on update expressions, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.UpdateExpressions.html
    UpdateExpression: "set #i.#g = list_append(#i.#g, :vals)",
    ExpressionAttributeNames: { "#i": "info", "#g": "genres" },
    ExpressionAttributeValues: {
      ":vals": ["Comedy"],
    },
    ReturnValues: "ALL_NEW",
  });
  const updateResponse = await docClient.send(updateCommand);
  log(`Movie updated: ${JSON.stringify(updateResponse.Attributes)}`);

  /**
   * Delete a movie from the table.
   */

  log("Deleting a single movie from the table.");
  const deleteCommand = new DeleteCommand({
    TableName: tableName,
    Key: { year: 1981, title: "The Evil Dead" },
  });
```

```
await docClient.send(deleteCommand);
log("Movie deleted.");

/**
 * Upload a batch of movies.
 */

log("Adding movies from local JSON file.");
const file = readFileSync(
  `${dirname}../../../../resources/sample_files/movies.json`,
);
const movies = JSON.parse(file.toString());
// chunkArray is a local convenience function. It takes an array and returns
// a generator function. The generator function yields every N items.
const movieChunks = chunkArray(movies, 25);
// For every chunk of 25 movies, make one BatchWrite request.
for (const chunk of movieChunks) {
  const putRequests = chunk.map((movie) => ({
    PutRequest: {
      Item: movie,
    },
  }));

  const command = new BatchWriteCommand({
    RequestItems: {
      [tableName]: putRequests,
    },
  });

  await docClient.send(command);
}
log("Movies added.");

/**
 * Query for movies by year.
 */

log("Querying for all movies from 1981.");
const paginatedQuery = paginateQuery(
  { client: docClient },
  {
    TableName: tableName,
    //For more information about query expressions, see
```

```
      // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Query.html#Query.KeyConditionExpressions
      KeyConditionExpression: "#y = :y",
      // 'year' is a reserved word in DynamoDB. Indicate that it's an attribute
      // name by using an expression attribute name.
      ExpressionAttributeNames: { "#y": "year" },
      ExpressionAttributeValues: { ":y": 1981 },
      ConsistentRead: true,
    },
  );
  /**
   * @type { Record<string, any>[] };
   */
  const movies1981 = [];
  for await (const page of paginatedQuery) {
    movies1981.push(...page.Items);
  }
  log(`Movies: ${movies1981.map((m) => m.title).join(", ")}`);

  /**
   * Scan the table for movies between 1980 and 1990.
   */

  log("Scan for movies released between 1980 and 1990");
  // A 'Scan' operation always reads every item in the table. If your design
 requires
  // the use of 'Scan', consider indexing your table or changing your design.
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-
scan.html
  const paginatedScan = paginateScan(
    { client: docClient },
    {
      TableName: tableName,
      // Scan uses a filter expression instead of a key condition expression.
 Scan will
      // read the entire table and then apply the filter.
      FilterExpression: "#y between :y1 and :y2",
      ExpressionAttributeNames: { "#y": "year" },
      ExpressionAttributeValues: { ":y1": 1980, ":y2": 1990 },
      ConsistentRead: true,
    },
  );
  /**
   * @type { Record<string, any>[] };
```

```
   */
  const movies1980to1990 = [];
  for await (const page of paginatedScan) {
    movies1980to1990.push(...page.Items);
  }
  log(
    `Movies: ${movies1980to1990
      .map((m) => `${m.title} (${m.year})`)
      .join(", ")}`,
  );

  /**
   * Delete the table.
   */

  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  log(`Deleting table ${tableName}.`);
  await client.send(deleteTableCommand);
  log("Table deleted.");
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDB テーブルを作成します。

```
suspend fun createScenarioTable(
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
```

```
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        println("The table was successfully created
 ${response.tableDescription?.tableArn}")
    }
}
```

サンプルの JSON ファイルをダウンロードして抽出するヘルパー関数を作成します。

```
// Load data into the table.
suspend fun loadData(
    tableName: String,
    fileName: String,
) {
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode

    var t = 0
    while (iter.hasNext()) {
        if (t == 50) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}
```

```
suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()
    val strVal = year.toString()
    // Add all content to the table.
    itemValues["year"] = AttributeValue.N(strVal)
    itemValues["title"] = AttributeValue.S(title)
    itemValues["info"] = AttributeValue.S(info)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println("Added $title to the Movie table.")
    }
}
```

テーブルから項目を取得します。

```
suspend fun getMovie(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.N(keyVal)
    keyToGet["title"] = AttributeValue.S("King Kong")

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }
```

```
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

完全な例です。

```
suspend fun main() {
    val tableName = "Movies"
    val fileName = "../../../resources/sample_files/movies.json"
    val partitionAlias = "#a"

    println("Creating an Amazon DynamoDB table named Movies with a key named id
 and a sort key named title.")
    createScenarioTable(tableName, "year")
    loadData(tableName, fileName)
    getMovie(tableName, "year", "1933")
    scanMovies(tableName)
    val count = queryMovieTable(tableName, "year", partitionAlias)
    println("There are $count Movies released in 2013.")
    deletIssuesTable(tableName)
}

suspend fun createScenarioTable(
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
```

```
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        println("The table was successfully created
 ${response.tableDescription?.tableArn}")
    }
}

// Load data into the table.
suspend fun loadData(
    tableName: String,
    fileName: String,
) {
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode
```

```
    var t = 0
    while (iter.hasNext()) {
        if (t == 50) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()
        putMovie(tableName, year, title, info)
        t++
    }
}

suspend fun putMovie(
    tableNameVal: String,
    year: Int,
    title: String,
    info: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()
    val strVal = year.toString()
    // Add all content to the table.
    itemValues["year"] = AttributeValue.N(strVal)
    itemValues["title"] = AttributeValue.S(title)
    itemValues["info"] = AttributeValue.S(info)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println("Added $title to the Movie table.")
    }
}

suspend fun getMovie(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
```

```
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.N(keyVal)
    keyToGet["title"] = AttributeValue.S("King Kong")

    val request =
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}

suspend fun deletIssuesTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}

suspend fun queryMovieTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = "year"

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.N("2013")
```

```
    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}

suspend fun scanMovies(tableNameVal: String) {
    val request =
        ScanRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.scan(request)
        response.items?.forEach { item ->
            item.keys.forEach { key ->
                println("The key name is $key\n")
                println("The value is ${item[key]}")
            }
        }
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem

- PutItem
- Query
- Scan
- UpdateItem

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例ではサポートファイルを使用しているため、PHP 例の README.md ファイルのガイダンスをお読みください。

```
namespace DynamoDb\Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;
use DynamoDb\DynamoDBService;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;

class GettingStartedWithDynamoDB
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the Amazon DynamoDB getting started demo using PHP!
\n");
        echo("---------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDBService();
```

```
        $tableName = "ddb_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
$tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);

        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
|| $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
```

```
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $key = [
            'Item' => [
                'title' => [
                    'S' => $movieName,
                ],
                'year' => [
                    'N' => $movieYear,
                ],
            ]
        ];
        $attributes = ["rating" =>
            [
                'AttributeName' => 'rating',
                'AttributeType' => 'N',
                'Value' => $rating,
            ],
            'plot' => [
                'AttributeName' => 'plot',
                'AttributeType' => 'S',
                'Value' => $plot,
            ]
        ];
        $service->updateItemAttributesByKey($tableName, $key, $attributes);
        echo "Movie added and updated.";

        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);


        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
 {$movie['Item']['year']['N']}.\n";
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
```

```
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
$rating);

        $movie = $service->getItemByKey($tableName, $key);
        echo "Ok, you have rated {$movie['Item']['title']['S']} as a
{$movie['Item']['rating']['N']}\n";

        $service->deleteItemByKey($tableName, $key);
        echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
```

```
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDB テーブルをカプセル化するクラスを作成します。

```
from decimal import Decimal
from io import BytesIO
import json
import logging
import os
from pprint import pprint
import requests
from zipfile import ZipFile
import boto3
from boto3.dynamodb.conditions import Key
from botocore.exceptions import ClientError
from question import Question

logger = logging.getLogger(__name__)

class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
```

```
                "Kelly Preston",
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def exists(self, table_name):
    """
    Determines whether a table exists. As a side effect, stores the table in
    a member variable.

    :param table_name: The name of the table to check.
    :return: True when the table exists; otherwise, False.
    """
    try:
        table = self.dyn_resource.Table(table_name)
        table.load()
        exists = True
    except ClientError as err:
        if err.response["Error"]["Code"] == "ResourceNotFoundException":
            exists = False
        else:
            logger.error(
                "Couldn't check for existence of %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
    else:
        self.table = table
    return exists
```

```
    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table


    def list_tables(self):
        """
        Lists the Amazon DynamoDB tables for the current account.

        :return: The list of tables.
        """
        try:
```

```
            tables = []
            for table in self.dyn_resource.tables.all():
                print(table.name)
                tables.append(table)
        except ClientError as err:
            logger.error(
                "Couldn't list tables. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return tables


    def write_batch(self, movies):
        """
        Fills an Amazon DynamoDB table with the specified data, using the Boto3
        Table.batch_writer() function to put the items in the table.
        Inside the context manager, Table.batch_writer builds a list of
        requests. On exiting the context manager, Table.batch_writer starts
sending
        batches of write requests to Amazon DynamoDB and automatically
        handles chunking, buffering, and retrying.

        :param movies: The data to put in the table. Each item must contain at
least
                       the keys required by the schema that was specified when
the
                       table was created.
        """
        try:
            with self.table.batch_writer() as writer:
                for movie in movies:
                    writer.put_item(Item=movie)
        except ClientError as err:
            logger.error(
                "Couldn't load data into table %s. Here's why: %s: %s",
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

```
    def add_movie(self, title, year, plot, rating):
        """
        Adds a movie to the table.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :param plot: The plot summary of the movie.
        :param rating: The quality rating of the movie.
        """
        try:
            self.table.put_item(
                Item={
                    "year": year,
                    "title": title,
                    "info": {"plot": plot, "rating": Decimal(str(rating))},
                }
            )
        except ClientError as err:
            logger.error(
                "Couldn't add movie %s to table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def get_movie(self, title, year):
        """
        Gets movie data from the table for a specific movie.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :return: The data about the requested movie.
        """
        try:
            response = self.table.get_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't get movie %s from table %s. Here's why: %s: %s",
                title,
                self.table.name,
```

```
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Item"]


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]


    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
```

```
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
self.table.query(KeyConditionExpression=Key("year").eq(year))
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Items"]


    def scan_movies(self, year_range):
        """
        Scans for movies that were released in a range of years.
        Uses a projection expression to return a subset of data for each movie.

        :param year_range: The range of years to retrieve.
        :return: The list of movies released in the specified years.
        """
        movies = []
        scan_kwargs = {
            "FilterExpression": Key("year").between(
                year_range["first"], year_range["second"]
            ),
            "ProjectionExpression": "#yr, title, info.rating",
            "ExpressionAttributeNames": {"#yr": "year"},
        }
        try:
            done = False
            start_key = None
            while not done:
                if start_key:
                    scan_kwargs["ExclusiveStartKey"] = start_key
                response = self.table.scan(**scan_kwargs)
                movies.extend(response.get("Items", []))
                start_key = response.get("LastEvaluatedKey", None)
                done = start_key is None
        except ClientError as err:
```

```
            logger.error(
                "Couldn't scan for movies. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise

        return movies


    def delete_movie(self, title, year):
        """
        Deletes a movie from the table.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        """
        try:
            self.table.delete_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't delete movie %s. Here's why: %s: %s",
                title,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise


    def delete_table(self):
        """
        Deletes the table.
        """
        try:
            self.table.delete()
            self.table = None
        except ClientError as err:
            logger.error(
                "Couldn't delete table. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

サンプルの JSON ファイルをダウンロードして抽出するヘルパー関数を作成します。

```
def get_sample_movie_data(movie_file_name):
    """
    Gets sample movie data, either from a local file or by first downloading it
 from
    the Amazon DynamoDB developer guide.

    :param movie_file_name: The local file name where the movie data is stored in
 JSON format.
    :return: The movie data as a dict.
    """
    if not os.path.isfile(movie_file_name):
        print(f"Downloading {movie_file_name}...")
        movie_content = requests.get(
            "https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
samples/moviedata.zip"
        )
        movie_zip = ZipFile(BytesIO(movie_content.content))
        movie_zip.extractall()

    try:
        with open(movie_file_name) as movie_file:
            movie_data = json.load(movie_file, parse_float=Decimal)
    except FileNotFoundError:
        print(
            f"File {movie_file_name} not found. You must first download the file
 to "
            "run this demo. See the README for instructions."
        )
        raise
    else:
        # The sample file lists over 4000 movies, return only the first 250.
        return movie_data[:250]
```

対話型シナリオを実行してテーブルを作成し、そのテーブルに対してアクションを実行します。

```
def run_scenario(table_name, movie_file_name, dyn_resource):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB getting started demo.")
    print("-" * 88)

    movies = Movies(dyn_resource)
    movies_exists = movies.exists(table_name)
    if not movies_exists:
        print(f"\nCreating table {table_name}...")
        movies.create_table(table_name)
        print(f"\nCreated table {movies.table.name}.")

    my_movie = Question.ask_questions(
        [
            Question(
                "title", "Enter the title of a movie you want to add to the
 table: "
            ),
            Question("year", "What year was it released? ", Question.is_int),
            Question(
                "rating",
                "On a scale of 1 - 10, how do you rate it? ",
                Question.is_float,
                Question.in_range(1, 10),
            ),
            Question("plot", "Summarize the plot for me: "),
        ]
    )
    movies.add_movie(**my_movie)
    print(f"\nAdded '{my_movie['title']}' to '{movies.table.name}'.")
    print("-" * 88)

    movie_update = Question.ask_questions(
        [
            Question(
                "rating",
                f"\nLet's update your movie.\nYou rated it {my_movie['rating']},
 what new "
```

```
                f"rating would you give it? ",
                Question.is_float,
                Question.in_range(1, 10),
            ),
            Question(
                "plot",
                f"You summarized the plot as '{my_movie['plot']}'.\nWhat would
you say now? ",
            ),
        ]
    )
    my_movie.update(movie_update)
    updated = movies.update_movie(**my_movie)
    print(f"\nUpdated '{my_movie['title']}' with new attributes:")
    pprint(updated)
    print("-" * 88)

    if not movies_exists:
        movie_data = get_sample_movie_data(movie_file_name)
        print(f"\nReading data from '{movie_file_name}' into your table.")
        movies.write_batch(movie_data)
        print(f"\nWrote {len(movie_data)} movies into {movies.table.name}.")
    print("-" * 88)

    title = "The Lord of the Rings: The Fellowship of the Ring"
    if Question.ask_question(
        f"Let's move on...do you want to get info about '{title}'? (y/n) ",
        Question.is_yesno,
    ):
        movie = movies.get_movie(title, 2001)
        print("\nHere's what I found:")
        pprint(movie)
    print("-" * 88)

    ask_for_year = True
    while ask_for_year:
        release_year = Question.ask_question(
            f"\nLet's get a list of movies released in a given year. Enter a year
between "
            f"1972 and 2018: ",
            Question.is_int,
            Question.in_range(1972, 2018),
        )
        releases = movies.query_movies(release_year)
```

```
        if releases:
            print(f"There were {len(releases)} movies released in
{release_year}:")
            for release in releases:
                print(f"\t{release['title']}")
            ask_for_year = False
        else:
            print(f"I don't know about any movies released in {release_year}!")
            ask_for_year = Question.ask_question(
                "Try another year? (y/n) ", Question.is_yesno
            )
    print("-" * 88)

    years = Question.ask_questions(
        [
            Question(
                "first",
                f"\nNow let's scan for movies released in a range of years. Enter
a year: ",
                Question.is_int,
                Question.in_range(1972, 2018),
            ),
            Question(
                "second",
                "Now enter another year: ",
                Question.is_int,
                Question.in_range(1972, 2018),
            ),
        ]
    )
    releases = movies.scan_movies(years)
    if releases:
        count = Question.ask_question(
            f"\nFound {len(releases)} movies. How many do you want to see? ",
            Question.is_int,
            Question.in_range(1, len(releases)),
        )
        print(f"\nHere are your {count} movies:\n")
        pprint(releases[:count])
    else:
        print(
            f"I don't know about any movies released between {years['first']} "
            f"and {years['second']}."
        )
```

```
    print("-" * 88)

    if Question.ask_question(
        f"\nLet's remove your movie from the table. Do you want to remove "
        f"'{my_movie['title']}'? (y/n)",
        Question.is_yesno,
    ):
        movies.delete_movie(my_movie["title"], my_movie["year"])
        print(f"\nRemoved '{my_movie['title']}' from the table.")
    print("-" * 88)

    if Question.ask_question(f"\nDelete the table? (y/n) ", Question.is_yesno):
        movies.delete_table()
        print(f"Deleted {table_name}.")
    else:
        print(
            "Don't forget to delete the table when you're done or you might incur
 "
            "charges on your account."
        )

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            "doc-example-table-movies", "moviedata.json",
 boto3.resource("dynamodb")
        )
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

このシナリオでは、次のヘルパークラスを使用してコマンドプロンプトで質問します。

```
class Question:
    """
    A helper class to ask questions at a command prompt and validate and convert
    the answers.
    """
```

```
    def __init__(self, key, question, *validators):
        """
        :param key: The key that is used for storing the answer in a dict, when
                    multiple questions are asked in a set.
        :param question: The question to ask.
        :param validators: The answer is passed through the list of validators
until
                           one fails or they all pass. Validators may also
convert the
                           answer to another form, such as from a str to an int.
        """
        self.key = key
        self.question = question
        self.validators = Question.non_empty, *validators

    @staticmethod
    def ask_questions(questions):
        """
        Asks a set of questions and stores the answers in a dict.

        :param questions: The list of questions to ask.
        :return: A dict of answers.
        """
        answers = {}
        for question in questions:
            answers[question.key] = Question.ask_question(
                question.question, *question.validators
            )
        return answers

    @staticmethod
    def ask_question(question, *validators):
        """
        Asks a single question and validates it against a list of validators.
        When an answer fails validation, the complaint is printed and the
question
        is asked again.

        :param question: The question to ask.
        :param validators: The list of validators that the answer must pass.
        :return: The answer, converted to its final form by the validators.
        """
        answer = None
        while answer is None:
```

```
            answer = input(question)
            for validator in validators:
                answer, complaint = validator(answer)
                if answer is None:
                    print(complaint)
                    break
        return answer

    @staticmethod
    def non_empty(answer):
        """
        Validates that the answer is not empty.
        :return: The non-empty answer, or None.
        """
        return answer if answer != "" else None, "I need an answer. Please?"

    @staticmethod
    def is_yesno(answer):
        """
        Validates a yes/no answer.
        :return: True when the answer is 'y'; otherwise, False.
        """
        return answer.lower() == "y", ""

    @staticmethod
    def is_int(answer):
        """
        Validates that the answer can be converted to an int.
        :return: The int answer; otherwise, None.
        """
        try:
            int_answer = int(answer)
        except ValueError:
            int_answer = None
        return int_answer, f"{answer} must be a valid integer."

    @staticmethod
    def is_letter(answer):
        """
        Validates that the answer is a letter.
        :return The letter answer, converted to uppercase; otherwise, None.
        """
        return (
            answer.upper() if answer.isalpha() else None,
```

```
            f"{answer} must be a single letter.",
        )

    @staticmethod
    def is_float(answer):
        """
        Validate that the answer can be converted to a float.
        :return The float answer; otherwise, None.
        """
        try:
            float_answer = float(answer)
        except ValueError:
            float_answer = None
        return float_answer, f"{answer} must be a valid float."

    @staticmethod
    def in_range(lower, upper):
        """
        Validate that the answer is within a range. The answer must be of a type
that can
        be compared to the lower and upper bounds.
        :return: The answer, if it is within the range; otherwise, None.
        """

        def _validate(answer):
            return (
                answer if lower <= answer <= upper else None,
                f"{answer} must be between {lower} and {upper}.",
            )

        return _validate
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable

- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDB テーブルをカプセル化するクラスを作成します。

```
# Creates an Amazon DynamoDB table that can be used to store movie data.
# The table uses the release year of the movie as the partition key and the
# title as the sort key.
#
# @param table_name [String] The name of the table to create.
# @return [Aws::DynamoDB::Table] The newly created table.
def create_table(table_name)
  @table = @dynamo_resource.create_table(
    table_name: table_name,
    key_schema: [
      { attribute_name: 'year', key_type: 'HASH' }, # Partition key
      { attribute_name: 'title', key_type: 'RANGE' } # Sort key
    ],
    attribute_definitions: [
      { attribute_name: 'year', attribute_type: 'N' },
      { attribute_name: 'title', attribute_type: 'S' }
    ],
    billing_mode: 'PAY_PER_REQUEST'
  )
  @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
```

```
    @table
  rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
  end
```

サンプルの JSON ファイルをダウンロードして抽出するヘルパー関数を作成します。

```
  # Gets sample movie data, either from a local file or by first downloading it
 from
  # the Amazon DynamoDB Developer Guide.
  #
  # @param movie_file_name [String] The local file name where the movie data is
 stored in JSON format.
  # @return [Hash] The movie data as a Hash.
  def fetch_movie_data(movie_file_name)
    if !File.file?(movie_file_name)
      @logger.debug("Downloading #{movie_file_name}...")
      movie_content = URI.open(
        'https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
samples/moviedata.zip'
      )
      movie_json = ''
      Zip::File.open_buffer(movie_content) do |zip|
        zip.each do |entry|
          movie_json = entry.get_input_stream.read
        end
      end
    else
      movie_json = File.read(movie_file_name)
    end
    movie_data = JSON.parse(movie_json)
    # The sample file lists over 4000 movies. This returns only the first 250.
    movie_data.slice(0, 250)
  rescue StandardError => e
    puts("Failure downloading movie data:\n#{e}")
    raise
  end
```

対話型シナリオを実行してテーブルを作成し、そのテーブルに対してアクションを実行します。

```
 table_name = "doc-example-table-movies-#{rand(10**4)}"
 scaffold = Scaffold.new(table_name)
 dynamodb_wrapper = DynamoDBBasics.new(table_name)

 new_step(1, 'Create a new DynamoDB table if none already exists.')
 unless scaffold.exists?(table_name)
   puts("\nNo such table: #{table_name}. Creating it...")
   scaffold.create_table(table_name)
   print "Done!\n".green
 end

 new_step(2, 'Add a new record to the DynamoDB table.')
 my_movie = {}
 my_movie[:title] = CLI::UI::Prompt.ask('Enter the title of a movie to add to
the table. E.g. The Matrix')
 my_movie[:year] = CLI::UI::Prompt.ask('What year was it released? E.g.
1989').to_i
 my_movie[:rating] = CLI::UI::Prompt.ask('On a scale of 1 - 10, how do you rate
it? E.g. 7').to_i
 my_movie[:plot] = CLI::UI::Prompt.ask('Enter a brief summary of the plot. E.g.
A man awakens to a new reality.')
 dynamodb_wrapper.add_item(my_movie)
 puts("\nNew record added:")
 puts JSON.pretty_generate(my_movie).green
 print "Done!\n".green

 new_step(3, 'Update a record in the DynamoDB table.')
 my_movie[:rating] = CLI::UI::Prompt.ask("Let's update the movie you added with
a new rating, e.g. 3:").to_i
 response = dynamodb_wrapper.update_item(my_movie)
 puts("Updated '#{my_movie[:title]}' with new attributes:")
 puts JSON.pretty_generate(response).green
 print "Done!\n".green

 new_step(4, 'Get a record from the DynamoDB table.')
 puts("Searching for #{my_movie[:title]} (#{my_movie[:year]})...")
 response = dynamodb_wrapper.get_item(my_movie[:title], my_movie[:year])
 puts JSON.pretty_generate(response).green
 print "Done!\n".green

 new_step(5, 'Write a batch of items into the DynamoDB table.')
 download_file = 'moviedata.json'
 puts("Downloading movie database to #{download_file}...")
```

```
 movie_data = scaffold.fetch_movie_data(download_file)
 puts("Writing movie data from #{download_file} into your table...")
 scaffold.write_batch(movie_data)
 puts("Records added: #{movie_data.length}.")
 print "Done!\n".green

 new_step(5, 'Query for a batch of items by key.')
 loop do
   release_year = CLI::UI::Prompt.ask('Enter a year between 1972 and 2018, e.g.
1999:').to_i
   results = dynamodb_wrapper.query_items(release_year)
   if results.any?
     puts("There were #{results.length} movies released in #{release_year}:")
     results.each do |movie|
       print "\t #{movie['title']}".green
     end
     break
   else
     continue = CLI::UI::Prompt.ask("Found no movies released in
#{release_year}! Try another year? (y/n)")
     break unless continue.eql?('y')
   end
 end
 print "\nDone!\n".green

 new_step(6, 'Scan for a batch of items using a filter expression.')
 years = {}
 years[:start] = CLI::UI::Prompt.ask('Enter a starting year between 1972 and
2018:')
 years[:end] = CLI::UI::Prompt.ask('Enter an ending year between 1972 and
2018:')
 releases = dynamodb_wrapper.scan_items(years)
 if !releases.empty?
   puts("Found #{releases.length} movies.")
   count = Question.ask(
     'How many do you want to see? ', method(:is_int), in_range(1,
releases.length)
   )
   puts("Here are your #{count} movies:")
   releases.take(count).each do |release|
     puts("\t#{release['title']}")
   end
 else
   puts("I don't know about any movies released between #{years[:start]} "\
```

```
          "and #{years[:end]}.")
  end
  print "\nDone!\n".green

  new_step(7, 'Delete an item from the DynamoDB table.')
  answer = CLI::UI::Prompt.ask("Do you want to remove '#{my_movie[:title]}'? (y/
n) ")
  if answer.eql?('y')
    dynamodb_wrapper.delete_item(my_movie[:title], my_movie[:year])
    puts("Removed '#{my_movie[:title]}' from the table.")
    print "\nDone!\n".green
  end

  new_step(8, 'Delete the DynamoDB table.')
  answer = CLI::UI::Prompt.ask('Delete the table? (y/n)')
  if answer.eql?('y')
    scaffold.delete_table
    puts("Deleted #{table_name}.")
  else
    puts("Don't forget to delete the table when you're done!")
  end
  print "\nThanks for watching!\n".green
rescue Aws::Errors::ServiceError
  puts('Something went wrong with the demo.')
rescue Errno::ENOENT
  true
end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query

- Scan
- UpdateItem

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    " Create an Amazon Dynamo DB table.

    TRY.
        DATA(lo_session) = /aws1/cl_rt_session_aws=>create( cv_pfl ).
        DATA(lo_dyn) = /aws1/cl_dyn_factory=>create( lo_session ).
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                           iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                           iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
        DATA(oo_result) = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
 returning.
```

```
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
        " It throws exception if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
 { lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.

    " Describe table
    TRY.
        DATA(lo_table) = lo_dyn->describetable( iv_tablename = iv_table_name ).
        DATA(lv_tablename) = lo_table->get_table( )->ask_tablename( ).
        MESSAGE 'The table name is ' && lv_tablename TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table does not exist' TYPE 'E'.
    ENDTRY.

    " Put items into the table.
    TRY.
        DATA(lo_resp_putitem) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Jaws' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1975' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '7.5' }| ) ) )
          ) ).
        lo_resp_putitem = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s = 'Star
 Wars' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
```

```
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1978' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '8.1' }| ) ) )
          ) ).
        lo_resp_putitem = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = VALUE /aws1/
cl_dynattributevalue=>tt_putiteminputattributemap(
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Speed' ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '1994' }| ) ) )
            ( VALUE /aws1/cl_dynattributevalue=>ts_putiteminputattrmap_maprow(
              key = 'rating' value = NEW /aws1/cl_dynattributevalue( iv_n = |
{ '7.9' }| ) ) )
          ) ).
        " TYPE REF TO /AWSEX/CL_AWS1_dyn_PUT_ITEM_OUTPUT
        MESSAGE '3 rows inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.

    " Get item from table.
    TRY.
        DATA(lo_resp_getitem) = lo_dyn->getitem(
          iv_tablename                = iv_table_name
          it_key                      = VALUE /aws1/cl_dynattributevalue=>tt_key(
           ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
             key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 'Jaws' ) ) )
           ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
             key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n =
 '1975' ) ) )
          ) ).
        DATA(lt_attr) = lo_resp_getitem->get_item( ).
```

```
        DATA(lo_title) = lt_attr[ key = 'title' ]-value.
        DATA(lo_year) = lt_attr[ key = 'year' ]-value.
        DATA(lo_rating) = lt_attr[ key = 'year' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Query item from table.
    TRY.
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
              ( NEW /aws1/cl_dynattributevalue( iv_n = '1975' ) ) ).
        DATA(lt_keyconditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
            it_attributevaluelist = lt_attributelist
            iv_comparisonoperator = |EQ|
          ) ) ) ).
        DATA(lo_query_result) = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_keyconditions ).
        DATA(lt_items) = lo_query_result->get_items( ).
        READ TABLE lo_query_result->get_items( ) INTO DATA(lt_item) INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
        lo_year = lt_item[ key = 'year' ]-value.
        lo_rating = lt_item[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Scan items from table.
    TRY.
        DATA(lo_scan_result) = lo_dyn->scan( iv_tablename = iv_table_name ).
        lt_items = lo_scan_result->get_items( ).
        " Read the first item and display the attributes.
        READ TABLE lo_query_result->get_items( ) INTO lt_item INDEX 1.
        lo_title = lt_item[ key = 'title' ]-value.
```

```
        lo_year = lt_item[ key = 'year' ]-value.
        lo_rating = lt_item[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( ) TYPE 'I'.
        MESSAGE 'Movie year is: ' && lo_year->get_n( ) TYPE 'I'.
        MESSAGE 'Movie rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.

    " Update items from table.
    TRY.
        DATA(lt_attributeupdates) = VALUE /aws1/
cl_dynattrvalueupdate=>tt_attributeupdates(
          ( VALUE /aws1/cl_dynattrvalueupdate=>ts_attributeupdates_maprow(
          key = 'rating' value = NEW /aws1/cl_dynattrvalueupdate(
            io_value  = NEW /aws1/cl_dynattributevalue( iv_n = '7.6' )
            iv_action = |PUT| ) ) ) ).
        DATA(lt_key) = VALUE /aws1/cl_dynattributevalue=>tt_key(
          ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
            key = 'year' value = NEW /aws1/cl_dynattributevalue( iv_n =
 '1975' ) ) )
          ( VALUE /aws1/cl_dynattributevalue=>ts_key_maprow(
            key = 'title' value = NEW /aws1/cl_dynattributevalue( iv_s =
 '1980' ) ) ) ).
        DATA(lo_resp) = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = lt_key
          it_attributeupdates = lt_attributeupdates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.

    " Delete table.
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        lo_dyn->get_waiter( )->tablenotexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
```

```
        MESSAGE 'DynamoDB Table deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted as it is in use' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の以下のトピックを参照してください。
  - BatchWriteItem
  - CreateTable
  - DeleteItem
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - Scan
  - UpdateItem

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

SDK for Swift への DynamoDB 呼び出しを処理する Swift クラス。

```
import AWSDynamoDB
import Foundation

/// An enumeration of error codes representing issues that can arise when using
```

```
/// the `MovieTable` class.
enum MoviesError: Error {
    /// The specified table wasn't found or couldn't be created.
    case TableNotFound
    /// The specified item wasn't found or couldn't be created.
    case ItemNotFound
    /// The Amazon DynamoDB client is not properly initialized.
    case UninitializedClient
    /// The table status reported by Amazon DynamoDB is not recognized.
    case StatusUnknown
    /// One or more specified attribute values are invalid or missing.
    case InvalidAttributes
}

/// A class representing an Amazon DynamoDB table containing movie
/// information.
public class MovieTable {
    var ddbClient: DynamoDBClient?
    let tableName: String

    /// Create an object representing a movie table in an Amazon DynamoDB
    /// database.
    ///
    /// - Parameters:
    ///   - region: The optional Amazon Region to create the database in.
    ///   - tableName: The name to assign to the table. If not specified, a
    ///     random table name is generated automatically.
    ///
    /// > Note: The table is not necessarily available when this function
    /// returns. Use `tableExists()` to check for its availability, or
    /// `awaitTableActive()` to wait until the table's status is reported as
    /// ready to use by Amazon DynamoDB.
    ///
    init(region: String? = nil, tableName: String) async throws {
        do {
            let config = try await DynamoDBClient.DynamoDBClientConfiguration()
            if let region = region {
                config.region = region
            }

            self.ddbClient = DynamoDBClient(config: config)
            self.tableName = tableName

            try await self.createTable()
```

```
        } catch {
            print("ERROR: ", dump(error, name: "Initializing Amazon
DynamoDBClient client"))
            throw error
        }
    }

    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }


    /// Check to see if the table exists online yet.
```

```
    ///
    /// - Returns: `true` if the table exists, or `false` if not.
    ///
    func tableExists() async throws -> Bool {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
            }

            return description.tableName == self.tableName
        } catch {
            print("ERROR: tableExists:", dump(error))
            throw error
        }
    }


    ///
    /// Waits for the table to exist and for its status to be active.
    ///
    func awaitTableActive() async throws {
        while try (await self.tableExists() == false) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25
seconds to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
            } catch {
                print("Sleep error:", dump(error))
            }
        }

        while try (await self.getTableStatus() != .active) {
            do {
                let duration = UInt64(0.25 * 1_000_000_000) // Convert .25
seconds to nanoseconds.
                try await Task.sleep(nanoseconds: duration)
```

```
            } catch {
                print("Sleep error:", dump(error))
            }
        }
    }


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }


    /// Get the table's status.
    ///
    /// - Returns: The table status, as defined by the
    ///   `DynamoDBClientTypes.TableStatus` enum.
    ///
    func getTableStatus() async throws -> DynamoDBClientTypes.TableStatus {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DescribeTableInput(
                tableName: self.tableName
            )
            let output = try await client.describeTable(input: input)
            guard let description = output.table else {
                throw MoviesError.TableNotFound
```

```
            }
            guard let status = description.tableStatus else {
                throw MoviesError.StatusUnknown
            }
            return status
        } catch {
            print("ERROR: getTableStatus:", dump(error))
            throw error
        }
    }


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from:
jsonData)

            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item
request.

            let count = movieList.count
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
```

```
            }

            // For each chunk, create a list of write request records and
populate
            // them with `PutRequest` requests, each specifying one movie from
the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
    }


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }
```

```
            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()

            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    /// Given a movie's details, add a movie to the Amazon DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title as a `String`.
    ///   - year: The release year of the movie (`Int`).
    ///   - rating: The movie's rating if available (`Double`; default is
    ///     `nil`).
    ///   - plot: A summary of the movie's plot (`String`; default is `nil`,
    ///     indicating no plot summary is available).
    ///
    func add(title: String, year: Int, rating: Double? = nil,
             plot: String? = nil) async throws
    {
        do {
            let movie = Movie(title: title, year: year, rating: rating, plot:
plot)
            try await self.add(movie: movie)
        } catch {
            print("ERROR: add with fields:", dump(error))
            throw error
        }
    }


    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
```

```
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }


    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
```

```
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }


    /// Return an array of `Movie` objects released in the specified range of
```

```
    /// years.
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]?
= nil)
        async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y`
instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )

            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
```

```
                    print("Error: no items returned.")
                    continue
                }

                // Build an array of `Movie` objects for the returned items.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
                }
            }
            return movieList

        } catch {
            print("ERROR: getMovies with scan:", dump(error))
            throw error
        }
    }


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.
```

```
            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
[:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }


    /// Delete a movie, given its title and release year.
```

```
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
}
```

MovieTable クラスがムービーを表現するために使用する構造。

```
import Foundation
import AWSDynamoDB

/// The optional details about a movie.
public struct Details: Codable {
    /// The movie's rating, if available.
    var rating: Double?
    /// The movie's plot, if available.
    var plot: String?
}

/// A structure describing a movie. The `year` and `title` properties are
/// required and are used as the key for Amazon DynamoDB operations. The
```

```
/// `info` sub-structure's two properties, `rating` and `plot`, are optional.
public struct Movie: Codable {
    /// The year in which the movie was released.
    var year: Int
    /// The movie's title.
    var title: String
    /// A `Details` object providing the optional movie rating and plot
    /// information.
    var info: Details

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The year in which the movie was released (`Int`).
    ///   - rating: The movie's rating (optional `Double`).
    ///   - plot: The movie's plot (optional `String`)
    init(title: String, year: Int, rating: Double? = nil, plot: String? = nil) {
        self.title = title
        self.year = year

        self.info = Details(rating: rating, plot: plot)
    }

    /// Create a `Movie` object representing a movie, given the movie's
    /// details.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The year in which the movie was released (`Int`).
    ///   - info: The optional rating and plot information for the movie in a
    ///     `Details` object.
    init(title: String, year: Int, info: Details?){
        self.title = title
        self.year = year

        if info != nil {
            self.info = info!
        } else {
            self.info = Details(rating: nil, plot: nil)
        }
    }
```

```
    ///
    /// Return a new `MovieTable` object, given an array mapping string to Amazon
    /// DynamoDB attribute values.
    ///
    /// - Parameter item: The item information provided to the form used by
    ///   DynamoDB. This is an array of strings mapped to
    ///   `DynamoDBClientTypes.AttributeValue` values.
    init(withItem item: [Swift.String:DynamoDBClientTypes.AttributeValue]) throws
{
        // Read the attributes.

        guard let titleAttr = item["title"],
              let yearAttr = item["year"] else {
            throw MoviesError.ItemNotFound
        }
        let infoAttr = item["info"] ?? nil

        // Extract the values of the title and year attributes.

        if case .s(let titleVal) = titleAttr {
            self.title = titleVal
        } else {
            throw MoviesError.InvalidAttributes
        }

        if case .n(let yearVal) = yearAttr {
            self.year = Int(yearVal)!
        } else {
            throw MoviesError.InvalidAttributes
        }

        // Extract the rating and/or plot from the `info` attribute, if
        // they're present.

        var rating: Double? = nil
        var plot: String? = nil

        if infoAttr != nil, case .m(let infoVal) = infoAttr {
            let ratingAttr = infoVal["rating"] ?? nil
            let plotAttr = infoVal["plot"] ?? nil

            if ratingAttr != nil, case .n(let ratingVal) = ratingAttr {
                rating = Double(ratingVal) ?? nil
            }
```

```
            if plotAttr != nil, case .s(let plotVal) = plotAttr {
                plot = plotVal
            }
        }

        self.info = Details(rating: rating, plot: plot)
    }

    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
        item["info"] = .m(details)

        return item
    }
}
```

MovieTable クラスを使用して DynamoDB データベースにアクセスするプログラム。

```
import ArgumentParser
import ClientRuntime
import Foundation

import AWSDynamoDB


@testable import MovieList

extension String {
    // Get the directory if the string is a file path.
    func directory() -> String {
        guard let lastIndex = lastIndex(of: "/") else {
            print("Error: String directory separator not found.")
            return ""
        }
        return String(self[...lastIndex])
    }
}

struct ExampleCommand: ParsableCommand {
    @Argument(help: "The path of the sample movie data JSON file.")
    var jsonPath: String = #file.directory() + "../../../../../resources/
sample_files/movies.json"

    @Option(help: "The AWS Region to run AWS API calls in.")
    var awsRegion: String?

    @Option(
        help: ArgumentHelp("The level of logging for the Swift SDK to perform."),
        completion: .list([
            "critical",
            "debug",
            "error",
            "info",
            "notice",
            "trace",
            "warning"
        ])
```

```
    )
    var logLevel: String = "error"

    /// Configuration details for the command.
    static var configuration = CommandConfiguration(
        commandName: "basics",
        abstract: "A basic scenario demonstrating the usage of Amazon DynamoDB.",
        discussion: """
        An example showing how to use Amazon DynamoDB to perform a series of
        common database activities on a simple movie database.
        """
    )

    /// Called by ``main()`` to asynchronously run the AWS example.
    func runAsync() async throws {
        print("Welcome to the AWS SDK for Swift basic scenario for Amazon
DynamoDB!")

        //========================================================================
        // 1. Create the table. The Amazon DynamoDB table is represented by
        //    the `MovieTable` class.
        //========================================================================

        let tableName = "ddb-movies-sample-\(Int.random(in: 1 ... Int.max))"

        print("Creating table \"\(tableName)\"...")

        let movieDatabase = try await MovieTable(region: awsRegion,
                                                 tableName: tableName)

        print("\nWaiting for table to be ready to use...")
        try await movieDatabase.awaitTableActive()

        //========================================================================
        // 2. Add a movie to the table.
        //========================================================================

        print("\nAdding a movie...")
        try await movieDatabase.add(title: "Avatar: The Way of Water", year:
2022)
        try await movieDatabase.add(title: "Not a Real Movie", year: 2023)

        //========================================================================
        // 3. Update the plot and rating of the movie using an update
```

```
        //    expression.
        //=====================================================================

        print("\nAdding details to the added movie...")
        _ = try await movieDatabase.update(title: "Avatar: The Way of Water",
year: 2022,
                                           rating: 9.2, plot: "It's a sequel.")

        //=====================================================================
        // 4. Populate the table from the JSON file.
        //=====================================================================

        print("\nPopulating the movie database from JSON...")
        try await movieDatabase.populate(jsonPath: jsonPath)

        //=====================================================================
        // 5. Get a specific movie by key. In this example, the key is a
        //    combination of `title` and `year`.
        //=====================================================================

        print("\nLooking for a movie in the table...")
        let gotMovie = try await movieDatabase.get(title: "This Is the End",
year: 2013)

        print("Found the movie \"\(gotMovie.title)\", released in
\(gotMovie.year).")
        print("Rating: \(gotMovie.info.rating ?? 0.0).")
        print("Plot summary: \(gotMovie.info.plot ?? "None.")")

        //=====================================================================
        // 6. Delete a movie.
        //=====================================================================

        print("\nDeleting the added movie...")
        try await movieDatabase.delete(title: "Avatar: The Way of Water", year:
2022)

        //=====================================================================
        // 7. Use a query with a key condition expression to return all movies
        //    released in a given year.
        //=====================================================================

        print("\nGetting movies released in 1994...")
        let movieList = try await movieDatabase.getMovies(fromYear: 1994)
```

```
        for movie in movieList {
            print("    \(movie.title)")
        }

        //===========================================================================
        // 8. Use `scan()` to return movies released in a range of years.
        //===========================================================================

        print("\nGetting movies released between 1993 and 1997...")
        let scannedMovies = try await movieDatabase.getMovies(firstYear: 1993,
 lastYear: 1997)
        for movie in scannedMovies {
            print("    \(movie.title) (\(movie.year))")
        }

        //===========================================================================
        // 9. Delete the table.
        //===========================================================================

        print("\nDeleting the table...")
        try await movieDatabase.deleteTable()
    }
}

@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.runAsync()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}
```

- API の詳細については、AWS SDK for Swift API リファレンスの以下のトピックを参照してください。
  - BatchWriteItem

- CreateTable
- DeleteItem
- DeleteTable
- DescribeTable
- GetItem
- PutItem
- Query
- Scan
- UpdateItem

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した DynamoDB 向けアクション

以下は、AWS SDK を使用して個々の DynamoDB アクションを実行する方法を説明するコード例です。それぞれの例には、GitHub へのリンクがあり、そこにはコードの設定と実行に関する説明が記載されています。

これらは DynamoDB API を呼び出すもので、コンテキスト内で実行する必要がある大規模なプログラムからのコード抜粋です。アクションは AWS SDK を使用した DynamoDB のシナリオ のコンテキスト内で確認できます。

以下の例には、最も一般的に使用されるアクションのみ含まれています。詳細なリストについては、Amazon DynamoDB API リファレンスを参照してください。

例

- AWS SDK で BatchExecuteStatement を使用する
- AWS SDK または CLI で BatchGetItem を使用する
- AWS SDK または CLI で BatchWriteItem を使用する
- AWS SDK または CLI で CreateTable を使用する
- AWS SDK または CLI で DeleteItem を使用する
- AWS SDK または CLI で DeleteTable を使用する

- AWS SDK または CLI で DescribeTable を使用する
- AWS SDK または CLI で DescribeTimeToLive を使用する
- AWS SDK で ExecuteStatement を使用する
- AWS SDK または CLI で GetItem を使用する
- AWS SDK または CLI で ListTables を使用する
- AWS SDK または CLI で PutItem を使用する
- AWS SDK または CLI で Query を使用する
- AWS SDK または CLI で Scan を使用する
- AWS SDK または CLI で UpdateItem を使用する
- AWS SDK または CLI で UpdateTable を使用する
- AWS SDK または CLI で UpdateTimeToLive を使用する

AWS SDK で **BatchExecuteStatement** を使用する

次のサンプルコードは、BatchExecuteStatement を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- PartiQL DELETE を使用してデータを削除する
- PartiQL INSERT を使用してデータを挿入する
- PartiQL ステートメントのバッチを使用してテーブルにクエリを実行する
- PartiQL SELECT を使用してデータをクエリする
- PartiQL UPDATE を使用してデータを更新する

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

INSERT ステートメントのバッチを使用して項目を追加します。

```
        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table into which the movie
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
                                Parameters = new List<AttributeValue>
                                {
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
```

```
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
            {
                return null!;
            }

            using var sr = new StreamReader(movieFileName);
```

```
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
            }
            else
            {
                return null!;
            }
        }
```

SELECT ステートメントのバッチを使用して項目を取得します。

```
        /// <summary>
        /// Gets movies from the movie table by
        /// using an Amazon DynamoDB PartiQL SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year1">The year of the first movie.</param>
        /// <param name="year2">The year of the second movie.</param>
        /// <returns>True if successful.</returns>
        public static async Task<bool> GetBatch(
            string tableName,
            string title1,
            string title2,
            int year1,
            int year2)
        {
            var getBatch = $"SELECT * FROM {tableName} WHERE title = ? AND year
 = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
```

```
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            if (response.Responses.Count > 0)
            {
                response.Responses.ForEach(r =>
                {
                    if (r.Item.Any())
                    {

Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]}");
                    }
                });
                return true;
            }
            else
            {
                Console.WriteLine($"Couldn't find either {title1} or {title2}.");
                return false;
            }

        }
```

UPDATE ステートメントのバッチを使用して項目を更新します。

```
        /// <summary>
        /// Updates information for multiple movies.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// movies to be updated.</param>
        /// <param name="producer1">The producer name for the first movie
        /// to update.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year that the first movie was released.</
param>
        /// <param name="producer2">The producer name for the second
        /// movie to update.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year that the second movie was released.</
param>
        /// <returns>A Boolean value that indicates the success of the update.</
returns>
        public static async Task<bool> UpdateBatch(
            string tableName,
            string producer1,
            string title1,
            int year1,
            string producer2,
            string title2,
            int year2)
        {

            string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title
 = ? AND year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer1 },
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },
```

```
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer2 },
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
 BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

DELETE ステートメントのバッチを使用して項目を削除します。

```
        /// <summary>
        /// Deletes multiple movies using a PartiQL BatchExecuteAsync
        /// statement.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// moves that will be deleted.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year the first movie was released.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year the second movie was released.</param>
        /// <returns>A Boolean value indicating the success of the operation.</
returns>
        public static async Task<bool> DeleteBatch(
            string tableName,
            string title1,
            int year1,
            string title2,
            int year2)
        {
```

```
            string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「BatchExecuteStatement」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

INSERT ステートメントのバッチを使用して項目を追加します。

```
    // 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::vector<Aws::String> titles;
    std::vector<float> ratings;
    std::vector<int> years;
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
                "On a scale of 1 - 10, how do you rate it? ",
                1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/
n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
              << (titles.size() == 1 ? " movie " : " movies ")
              << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
```

```
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                  << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                  << INFO_KEY << "': ?}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));

            // Create attribute for the info map.
            Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            ratingAttribute->SetN(ratings[i]);
            infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            plotAttribute->SetS(plots[i]);
            infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
            attributes.push_back(infoMapAttribute);
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
```

```
            std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }
```

SELECT ステートメントのバッチを使用して項目を取得します。

```
    // 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (outcome.IsSuccess()) {
            const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();
```

```
            const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

            for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

                printMovieInfo(item);
            }
        }
        else {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

UPDATE ステートメントのバッチを使用して項目を更新します。

```
    // 4. Update the data for multiple movies using "Update" statements.
(BatchExecuteStatement)

    for (size_t i = 0; i < titles.size(); ++i) {
        ratings[i] = askQuestionForFloatRange(
                Aws::String("\nLet's update your the movie, \"") + titles[i] +
                ".\nYou rated it  " + std::to_string(ratings[i])
                + ", what new rating would you give it? ", 1, 10);
    }

    std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<
std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                  << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                  << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";
```

```
        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);
        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

DELETE ステートメントのバッチを使用して項目を削除します。

```
    // 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());
```

```
        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movies: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

- API の詳細については、AWS SDK for C++ API リファレンスの「BatchExecuteStatement」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例の関数レシーバー構造体を定義します。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

INSERT ステートメントのバッチを使用して項目を追加します。

```
// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
 to the
// DynamoDB table.
func (runner PartiQLRunner) AddMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year, movie.Info})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(fmt.Sprintf(
    "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
 runner.TableName)),
   Parameters: params,
  }
```

```
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
 err)
 }
 return err
}
```

SELECT ステートメントのバッチを使用して項目を取得します。

```
// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
 from
// the DynamoDB table by title and year.
func (runner PartiQLRunner) GetMovieBatch(ctx context.Context, movies []Movie)
 ([]Movie, error) {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 output, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 var outMovies []Movie
```

```
 if err != nil {
  log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
 } else {
  for _, response := range output.Responses {
   var movie Movie
   err = attributevalue.UnmarshalMap(response.Item, &movie)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    outMovies = append(outMovies, movie)
   }
  }
 }
 return outMovies, err
}
```

UPDATE ステートメントのバッチを使用して項目を更新します。

```
// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
 rating of
// multiple movies that already exist in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovieBatch(ctx context.Context, movies []Movie,
 ratings []float64) error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{ratings[index],
 movie.Title, movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
```

```
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

DELETE ステートメントのバッチを使用して項目を削除します。

```
// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
 movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「BatchExecuteStatement」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL を使用して項目のバッチを作成します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const breakfastFoods = ["Eggs", "Bacon", "Sausage"];
  const command = new BatchExecuteStatementCommand({
    Statements: breakfastFoods.map((food) => ({
      Statement: `INSERT INTO BreakfastFoods value {'Name':?}`,
      Parameters: [food],
    })),
```

```
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目のバッチを取得します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Teaspoons"],
        ConsistentRead: true,
      },
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Grams"],
        ConsistentRead: true,
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目のバッチを更新します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const eggUpdates = [
    ["duck", "fried"],
    ["chicken", "omelette"],
  ];
  const command = new BatchExecuteStatementCommand({
    Statements: eggUpdates.map((change) => ({
      Statement: "UPDATE Eggs SET Style=? where Variety=?",
      Parameters: [change[1], change[0]],
    })),
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目のバッチを削除します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
```

```
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Grape"],
      },
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Strawberry"],
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchExecuteStatement」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    public function getItemByPartiQLBatch(string $tableName, array $keys): Result
    {
        $statements = [];
        foreach ($keys as $key) {
            list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);
            $statements[] = [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ];
        }
```

```
        return $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => $statements,
        ]);
    }

    public function insertItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function updateItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function deleteItemByPartiQLBatch(string $statement, array
$parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }
```

- API の詳細についてはAWS SDK for PHP API リファレンスの「BatchExecuteStatement」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class PartiQLBatchWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statements, param_list):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statements: The batch of PartiQL statements.
        :param param_list: The batch of PartiQL parameters that are associated
 with
```

```
                            each statement. This list must be in the same order as
the
                            statements.
        :return: The responses returned from running the statements, if any.
        """
        try:
            output = self.dyn_resource.meta.client.batch_execute_statement(
                Statements=[
                    {"Statement": statement, "Parameters": params}
                    for statement, params in zip(statements, param_list)
                ]
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute batch of PartiQL statements because the
table "
                    "does not exist."
                )
            else:
                logger.error(
                    "Couldn't execute batch of PartiQL statements. Here's why:
%s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「BatchExecuteStatement」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL を使用して項目のバッチを読み取ります。

```
class DynamoDBPartiQLBatch
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Selects a batch of items from a table using PartiQL
  #
  # @param batch_titles [Array] Collection of movie titles
  # @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
  def batch_execute_select(batch_titles)
    request_items = batch_titles.map do |title, year|
      {
        statement: "SELECT * FROM \"#{@table.name}\" WHERE title=? and year=?",
        parameters: [title, year]
      }
    end
    @dynamodb.client.batch_execute_statement({ statements: request_items })
  end
```

PartiQL を使用して項目のバッチを削除します。

```
class DynamoDBPartiQLBatch
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
```

```
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Deletes a batch of items from a table using PartiQL
  #
  # @param batch_titles [Array] Collection of movie titles
  # @return [Aws::DynamoDB::Types::BatchExecuteStatementOutput]
  def batch_execute_write(batch_titles)
    request_items = batch_titles.map do |title, year|
      {
        statement: "DELETE FROM \"#{@table.name}\" WHERE title=? and year=?",
        parameters: [title, year]
      }
    end
    @dynamodb.client.batch_execute_statement({ statements: request_items })
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「BatchExecuteStatement」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **BatchGetItem** を使用する

次のサンプルコードは、BatchGetItem を使用する方法を説明しています。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;

namespace LowLevelBatchGet
{
    public class LowLevelBatchGet
    {
        private static readonly string _table1Name = "Forum";
        private static readonly string _table2Name = "Thread";

        public static async void
 RetrieveMultipleItemsBatchGet(AmazonDynamoDBClient client)
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
            {
                { _table1Name,
                  new KeysAndAttributes
                  {
                      Keys = new List<Dictionary<string, AttributeValue> >()
                      {
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                                S = "Amazon DynamoDB"
                            } }
                          },
                          new Dictionary<string, AttributeValue>()
                          {
                              { "Name", new AttributeValue {
                                S = "Amazon S3"
                            } }
                          }
                      }
                  }},
                {
                    _table2Name,
                    new KeysAndAttributes
                    {
                        Keys = new List<Dictionary<string, AttributeValue> >()
```

```
                        {
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 1"
                                  } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon DynamoDB"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "DynamoDB Thread 2"
                                  } }
                            },
                            new Dictionary<string, AttributeValue>()
                            {
                                { "ForumName", new AttributeValue {
                                      S = "Amazon S3"
                                  } },
                                { "Subject", new AttributeValue {
                                      S = "S3 Thread 1"
                                  } }
                            }
                        }
                    }
                }
            }
            };

            BatchGetItemResponse response;
            do
            {
                Console.WriteLine("Making request");
                response = await client.BatchGetItemAsync(request);

                // Check the response.
                var responses = response.Responses; // Attribute list in the
response.
```

```
                foreach (var tableResponse in responses)
                {
                    var tableResults = tableResponse.Value;
                    Console.WriteLine("Items retrieved from table {0}",
tableResponse.Key);
                    foreach (var item1 in tableResults)
                    {
                        PrintItem(item1);
                    }
                }

                // Any unprocessed keys? could happen if you exceed
ProvisionedThroughput or some other error.
                Dictionary<string, KeysAndAttributes> unprocessedKeys =
response.UnprocessedKeys;
                foreach (var unprocessedTableKeys in unprocessedKeys)
                {
                    // Print table name.
                    Console.WriteLine(unprocessedTableKeys.Key);
                    // Print unprocessed primary keys.
                    foreach (var key in unprocessedTableKeys.Value.Keys)
                    {
                        PrintItem(key);
                    }
                }

                request.RequestItems = unprocessedKeys;
            } while (response.UnprocessedKeys.Count > 0);
        }

        private static void PrintItem(Dictionary<string, AttributeValue>
attributeList)
        {
            foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
            {
                string attributeName = kvp.Key;
                AttributeValue value = kvp.Value;

                Console.WriteLine(
                    attributeName + " " +
                    (value.S == null ? "" : "S=[" + value.S + "]") +
                    (value.N == null ? "" : "N=[" + value.N + "]") +
                    (value.SS == null ? "" : "SS=[" + string.Join(",",
value.SS.ToArray()) + "]") +
```

```
                    (value.NS == null ? "" : "NS=[" + string.Join(",",
 value.NS.ToArray()) + "]")
                    );
            }

 Console.WriteLine("*********************************************");
        }

        static void Main()
        {
            var client = new AmazonDynamoDBClient();

            RetrieveMultipleItemsBatchGet(client);
        }
    }
}
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「BatchGetItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_batch_get_item
#
# This function gets a batch of items from a DynamoDB table.
#
# Parameters:
#       -i item  -- Path to json file containing the keys of the items to get.
#
#  Returns:
#       The items as json output.
```

```
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_batch_get_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_get_item"
    echo "Get a batch of items from a DynamoDB table."
    echo " -i item  -- Path to json file containing the keys of the items to
 get."
    echo ""
  }

  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  response=$(aws dynamodb batch-get-item \
    --request-items file://"$item")
  local error_code=${?}
```

```
  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-get-item operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
```

```
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、AWS CLI コマンドリファレンスの「BatchGetItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Batch get items from different Amazon DynamoDB tables.
/*!
  \sa batchGetItem()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::batchGetItem(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
```

```
    Aws::DynamoDB::Model::BatchGetItemRequest request;

    // Table1: Forum.
    Aws::String table1Name = "Forum";
    Aws::DynamoDB::Model::KeysAndAttributes table1KeysAndAttributes;

    // Table1: Projection expression.
    table1KeysAndAttributes.SetProjectionExpression("#n, Category, Messages,
#v");

    // Table1: Expression attribute names.
    Aws::Http::HeaderValueCollection headerValueCollection;
    headerValueCollection.emplace("#n", "Name");
    headerValueCollection.emplace("#v", "Views");
    table1KeysAndAttributes.SetExpressionAttributeNames(headerValueCollection);

    // Table1: Set key name, type, and value to search.
    std::vector<Aws::String> nameValues = {"Amazon DynamoDB", "Amazon S3"};
    for (const Aws::String &name: nameValues) {
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
        Aws::DynamoDB::Model::AttributeValue key;
        key.SetS(name);
        keys.emplace("Name", key);
        table1KeysAndAttributes.AddKeys(keys);
    }

    Aws::Map<Aws::String, Aws::DynamoDB::Model::KeysAndAttributes> requestItems;
    requestItems.emplace(table1Name, table1KeysAndAttributes);

    // Table2: ProductCatalog.
    Aws::String table2Name = "ProductCatalog";
    Aws::DynamoDB::Model::KeysAndAttributes table2KeysAndAttributes;
    table2KeysAndAttributes.SetProjectionExpression("Title, Price, Color");

    // Table2: Set key name, type, and value to search.
    std::vector<Aws::String> idValues = {"102", "103", "201"};
    for (const Aws::String &id: idValues) {
        Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> keys;
        Aws::DynamoDB::Model::AttributeValue key;
        key.SetN(id);
        keys.emplace("Id", key);
        table2KeysAndAttributes.AddKeys(keys);
    }
```

```
   requestItems.emplace(table2Name, table2KeysAndAttributes);

   bool result = true;
   do {  // Use a do loop to handle pagination.
       request.SetRequestItems(requestItems);
       const Aws::DynamoDB::Model::BatchGetItemOutcome &outcome =
dynamoClient.BatchGetItem(
               request);

       if (outcome.IsSuccess()) {
           for (const auto &responsesMapEntry:
outcome.GetResult().GetResponses()) {
               Aws::String tableName = responsesMapEntry.first;
               const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &tableResults = responsesMapEntry.second;
               std::cout << "Retrieved " << tableResults.size()
                         << " responses for table '" << tableName << "'.\n"
                         << std::endl;
               if (tableName == "Forum") {

                   std::cout << "Name | Category | Message | Views" <<
std::endl;
                   for (const Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
                       std::cout << item.at("Name").GetS() << " | ";
                       std::cout << item.at("Category").GetS() << " | ";
                       std::cout << (item.count("Message") == 0 ? "" : item.at(
                               "Messages").GetN()) << " | ";
                       std::cout << (item.count("Views") == 0 ? "" : item.at(
                               "Views").GetN()) << std::endl;
                   }
               }
               else {
                   std::cout << "Title | Price | Color" << std::endl;
                   for (const Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue> &item: tableResults) {
                       std::cout << item.at("Title").GetS() << " | ";
                       std::cout << (item.count("Price") == 0 ? "" : item.at(
                               "Price").GetN());
                       if (item.count("Color")) {
                           std::cout << " | ";
                           for (const
std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> &listItem: item.at(
```

```
                                    "Color").GetL())
                                std::cout << listItem->GetS() << " ";
                        }
                        std::cout << std::endl;
                    }
                }
                std::cout << std::endl;
            }

            // If necessary, repeat request for remaining items.
            requestItems = outcome.GetResult().GetUnprocessedKeys();
        }
        else {
            std::cerr << "Batch get item failed: " <<
 outcome.GetError().GetMessage()
                      << std::endl;
            result = false;
            break;
        }
    } while (!requestItems.empty());

    return result;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「BatchGetItem」を参照してください。

CLI

AWS CLI

テーブルから複数の項目を取得するには

次の batch-get-items の例では、3 つの GetItem リクエストのバッチを使用して MusicCollection テーブルから複数の項目を読み込み、この操作で使用された読み込み容量ユニットの数を取得します。このコマンドは AlbumTitle 属性のみを返します。

```
aws dynamodb batch-get-item \
    --request-items file://request-items.json \
    --return-consumed-capacity TOTAL
```

request-items.json の内容:

```
{
    "MusicCollection": {
        "Keys": [
            {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Call Me Today"}
            },
            {
                "Artist": {"S": "Acme Band"},
                "SongTitle": {"S": "Happy Day"}
            },
            {
                "Artist": {"S": "No One You Know"},
                "SongTitle": {"S": "Scared of My Shadow"}
            }
        ],
        "ProjectionExpression":"AlbumTitle"
    }
}
```

出力:

```
{
    "Responses": {
        "MusicCollection": [
            {
                "AlbumTitle": {
                    "S": "Somewhat Famous"
                }
            },
            {
                "AlbumTitle": {
                    "S": "Blue Sky Blues"
                }
            },
            {
                "AlbumTitle": {
                    "S": "Louder Than Ever"
                }
            }
        ]
```

```
    },
    "UnprocessedKeys": {},
    "ConsumedCapacity": [
        {
            "TableName": "MusicCollection",
            "CapacityUnits": 1.5
        }
    ]
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「バッチオペレーション」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「BatchGetItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

サービスクライアントを使用してバッチアイテムを取得する方法を示します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
```

```
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class BatchReadItems {
    public static void main(String[] args){
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        getBatchItems(dynamoDbClient, tableName);
    }

    public static void getBatchItems(DynamoDbClient dynamoDbClient, String
 tableName) {
        // Define the primary key values for the items you want to retrieve.
        Map<String, AttributeValue> key1 = new HashMap<>();
        key1.put("Artist", AttributeValue.builder().s("Artist1").build());

        Map<String, AttributeValue> key2 = new HashMap<>();
        key2.put("Artist", AttributeValue.builder().s("Artist2").build());

        // Construct the batchGetItem request.
        Map<String, KeysAndAttributes> requestItems = new HashMap<>();
        requestItems.put(tableName, KeysAndAttributes.builder()
            .keys(List.of(key1, key2))
            .projectionExpression("Artist, SongTitle")
            .build());

        BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
            .requestItems(requestItems)
            .build();
```

```
        // Make the batchGetItem request.
        BatchGetItemResponse batchGetItemResponse =
 dynamoDbClient.batchGetItem(batchGetItemRequest);

        // Extract and print the retrieved items.
        Map<String, List<Map<String, AttributeValue>>> responses =
 batchGetItemResponse.responses();
        if (responses.containsKey(tableName)) {
            List<Map<String, AttributeValue>> musicItems =
 responses.get(tableName);
            for (Map<String, AttributeValue> item : musicItems) {
                System.out.println("Artist: " + item.get("Artist").s() +
                    ", SongTitle: " + item.get("SongTitle").s());
            }
        } else {
            System.out.println("No items retrieved.");
        }
    }
}
```

サービスクライアントおよびページネーターを使用してバッチアイテムを取得する方法を示します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchGetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.KeysAndAttributes;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BatchGetItemsPaginator {

    public static void main(String[] args){
        final String usage = """

                Usage:
                    <tableName>
```

```
                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        getBatchItemsPaginator(dynamoDbClient, tableName) ;
    }

    public static void getBatchItemsPaginator(DynamoDbClient dynamoDbClient,
 String tableName) {
        // Define the primary key values for the items you want to retrieve.
        Map<String, AttributeValue> key1 = new HashMap<>();
        key1.put("Artist", AttributeValue.builder().s("Artist1").build());

        Map<String, AttributeValue> key2 = new HashMap<>();
        key2.put("Artist", AttributeValue.builder().s("Artist2").build());

        // Construct the batchGetItem request.
        Map<String, KeysAndAttributes> requestItems = new HashMap<>();
        requestItems.put(tableName, KeysAndAttributes.builder()
            .keys(List.of(key1, key2))
            .projectionExpression("Artist, SongTitle")
            .build());

        BatchGetItemRequest batchGetItemRequest = BatchGetItemRequest.builder()
            .requestItems(requestItems)
            .build();

        // Use batchGetItemPaginator for paginated requests.
        dynamoDbClient.batchGetItemPaginator(batchGetItemRequest).stream()
            .flatMap(response -> response.responses().getOrDefault(tableName,
 Collections.emptyList()).stream())
            .forEach(item -> {
                System.out.println("Artist: " + item.get("Artist").s() +
                    ", SongTitle: " + item.get("SongTitle").s());
            });
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「BatchGetItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「BatchGet」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { BatchGetCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchGetCommand({
    // Each key in this object is the name of a table. This example refers
    // to a Books table.
    RequestItems: {
      Books: {
        // Each entry in Keys is an object that specifies a primary key.
        Keys: [
          {
            Title: "How to AWS",
          },
          {
            Title: "DynamoDB for DBAs",
          },
        ],
        // Only return the "Title" and "PageCount" attributes.
        ProjectionExpression: "Title, PageCount",
```

```
      },
    },
  });

  const response = await docClient.send(command);
  console.log(response.Responses.Books);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchGetItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        { KEY_NAME: { N: "KEY_VALUE_1" } },
        { KEY_NAME: { N: "KEY_VALUE_2" } },
        { KEY_NAME: { N: "KEY_VALUE_3" } },
      ],
      ProjectionExpression: "KEY_NAME, ATTRIBUTE",
    },
  },
};
```

```
ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchGetItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: DynamoDB テーブルの「Music」と「Songs」から「Somewhere Down The Road」という SongTitle の項目を取得します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$keysAndAttributes = New-Object Amazon.DynamoDBv2.Model.KeysAndAttributes
$list = New-Object
 'System.Collections.Generic.List[System.Collections.Generic.Dictionary[String,
 Amazon.DynamoDBv2.Model.AttributeValue]]'
$list.Add($key)
$keysAndAttributes.Keys = $list

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
    'Songs' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
}

$batchItems = Get-DDBBatchItem -RequestItem $requestItem
```

```
$batchItems.GetEnumerator() | ForEach-Object {$PSItem.Value} | ConvertFrom-
DDBItem
```

出力する。

```
Name                           Value
----                           -----
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「BatchGetItem」を参照してください。

Tools for PowerShell V5

例 1: DynamoDB テーブルの「Music」と「Songs」から「Somewhere Down The Road」という SongTitle の項目を取得します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$keysAndAttributes = New-Object Amazon.DynamoDBv2.Model.KeysAndAttributes
$list = New-Object
 'System.Collections.Generic.List[System.Collections.Generic.Dictionary[String,
 Amazon.DynamoDBv2.Model.AttributeValue]]'
$list.Add($key)
$keysAndAttributes.Keys = $list

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
    'Songs' = [Amazon.DynamoDBv2.Model.KeysAndAttributes]$keysAndAttributes
```

```
}

$batchItems = Get-DDBBatchItem -RequestItem $requestItem
$batchItems.GetEnumerator() | ForEach-Object {$PSItem.Value} | ConvertFrom-
DDBItem
```

出力する。

```
Name                           Value
----                           -----
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
Artist                         No One You Know
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
CriticRating                   10
Genre                          Country
Price                          1.94
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「BatchGetItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import decimal
import json
import logging
import os
import pprint
```

```
import time
import boto3
from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
dynamodb = boto3.resource("dynamodb")

MAX_GET_SIZE = 100  # Amazon DynamoDB rejects a get batch larger than 100 items.


def do_batch_get(batch_keys):
    """
    Gets a batch of items from Amazon DynamoDB. Batches can contain keys from
    more than one table.

    When Amazon DynamoDB cannot process all items in a batch, a set of
 unprocessed
    keys is returned. This function uses an exponential backoff algorithm to
 retry
    getting the unprocessed keys until all are retrieved or the specified
    number of tries is reached.

    :param batch_keys: The set of keys to retrieve. A batch can contain at most
 100
                       keys. Otherwise, Amazon DynamoDB returns an error.
    :return: The dictionary of retrieved items grouped under their respective
             table names.
    """
    tries = 0
    max_tries = 5
    sleepy_time = 1  # Start with 1 second of sleep, then exponentially increase.
    retrieved = {key: [] for key in batch_keys}
    while tries < max_tries:
        response = dynamodb.batch_get_item(RequestItems=batch_keys)
        # Collect any retrieved items and retry unprocessed keys.
        for key in response.get("Responses", []):
            retrieved[key] += response["Responses"][key]
        unprocessed = response["UnprocessedKeys"]
        if len(unprocessed) > 0:
            batch_keys = unprocessed
            unprocessed_count = sum(
                [len(batch_key["Keys"]) for batch_key in batch_keys.values()]
            )
            logger.info(
```

```
                "%s unprocessed keys returned. Sleep, then retry.",
 unprocessed_count
            )
            tries += 1
            if tries < max_tries:
                logger.info("Sleeping for %s seconds.", sleepy_time)
                time.sleep(sleepy_time)
                sleepy_time = min(sleepy_time * 2, 32)
        else:
            break

    return retrieved
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「BatchGetItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Gets an array of `Movie` objects describing all the movies in the
    /// specified list. Any movies that aren't found in the list have no
    /// corresponding entry in the resulting array.
    ///
    /// - Parameters
    ///     - keys: An array of tuples, each of which specifies the title and
    ///       release year of a movie to fetch from the table.
    ///
    /// - Returns:
```

```
    ///     - An array of `Movie` objects describing each match found in the
    ///     table.
    ///
    /// - Throws:
    ///     - `MovieError.ClientUninitialized` if the DynamoDB client has not
    ///     been initialized.
    ///     - DynamoDB errors are thrown without change.
    func batchGet(keys: [(title: String, year: Int)]) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MovieError.ClientUninitialized
            }

            var movieList: [Movie] = []
            var keyItems: [[Swift.String: DynamoDBClientTypes.AttributeValue]] =
[]

            // Convert the list of keys into the form used by DynamoDB.

            for key in keys {
                let item: [Swift.String: DynamoDBClientTypes.AttributeValue] = [
                    "title": .s(key.title),
                    "year": .n(String(key.year))
                ]
                keyItems.append(item)
            }

            // Create the input record for `batchGetItem()`. The list of
requested
            // items is in the `requestItems` property. This array contains one
            // entry for each table from which items are to be fetched. In this
            // example, there's only one table containing the movie data.
            //
            // If we wanted this program to also support searching for matches
            // in a table of book data, we could add a second `requestItem`
            // mapping the name of the book table to the list of items we want to
            // find in it.
            let input = BatchGetItemInput(
                requestItems: [
                    self.tableName: .init(
                        consistentRead: true,
                        keys: keyItems
                    )
                ]
```

```
            )

            // Fetch the matching movies from the table.

            let output = try await client.batchGetItem(input: input)

            // Get the set of responses. If there aren't any, return the empty
            // movie list.

            guard let responses = output.responses else {
                return movieList
            }

            // Get the list of matching items for the table with the name
            // `tableName`.

            guard let responseList = responses[self.tableName] else {
                return movieList
            }

            // Create `Movie` items for each of the matching movies in the table
            // and add them to the `MovieList` array.

            for response in responseList {
                try movieList.append(Movie(withItem: response))
            }

            return movieList
        } catch {
            print("ERROR: batchGet", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「BatchGetItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **BatchWriteItem** を使用する

次のサンプルコードは、BatchWriteItem を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

項目のバッチをムービーテーブルに書き込みます。

```
/// <summary>
/// Loads the contents of a JSON file into a list of movies to be
/// added to the DynamoDB table.
/// </summary>
/// <param name="movieFileName">The name of the JSON file.</param>
/// <returns>A generic list of movie objects.</returns>
public List<Movie> ImportMovies(string movieFileName)
{
    var moviesList = new List<Movie>();
    if (!File.Exists(movieFileName))
    {
        return moviesList;
    }

    using var sr = new StreamReader(movieFileName);
    string json = sr.ReadToEnd();
    var allMovies = JsonSerializer.Deserialize<List<Movie>>(
        json,
        new JsonSerializerOptions
        {
```

```
                PropertyNameCaseInsensitive = true
            });

        // Now return the first 250 entries.
        if (allMovies != null && allMovies.Any())
        {
            moviesList = allMovies.GetRange(0, 250);
        }
        return moviesList;
    }

    /// <summary>
    /// Writes 250 items to the movie table.
    /// </summary>
    /// <param name="movieFileName">A string containing the full path to
    /// the JSON file containing movie data.</param>
    /// <param name="tableName">The name of the table to write items to.</param>
    /// <returns>A long integer value representing the number of movies
    /// imported from the JSON file.</returns>
    public async Task<long> BatchWriteItemsAsync(
        string movieFileName, string tableName)
    {
        try
        {
            var movies = ImportMovies(movieFileName);
            if (!movies.Any())
            {
                Console.WriteLine("Couldn't find the JSON file with movie
data.");
                return 0;
            }

            var context = new DynamoDBContextBuilder()
                // Optional call to provide a specific instance of
IAmazonDynamoDB
                .WithDynamoDBClient(() => _amazonDynamoDB)
                .Build();

            var movieBatch = context.CreateBatchWrite<Movie>(
                new BatchWriteConfig()
                {
                    OverrideTableName = tableName
                });
            movieBatch.AddPutItems(movies);
```

```
            Console.WriteLine("Adding imported movies to the table.");
            await movieBatch.ExecuteAsync();

            return movies.Count;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table was not found during batch write operation.
{ex.Message}");
            throw;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred during batch
write operation. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred during batch write operation.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「BatchWriteItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_batch_write_item
#
# This function writes a batch of items into a DynamoDB table.
#
# Parameters:
#       -i item  -- Path to json file containing the items to write.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_batch_write_item() {
  local item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_batch_write_item"
    echo "Write a batch of items into a DynamoDB table."
    echo " -i item  -- Path to json file containing the items to write."
    echo ""
  }
  while getopts "i:h" option; do
    case "${option}" in
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$item" ]]; then
    errecho "ERROR: You must provide an item with the -i parameter."
```

```
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""

  response=$(aws dynamodb batch-write-item \
    --request-items file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
```

```
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「BatchWriteItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Batch write items from a JSON file.
/*!
  \sa batchWriteItem()
  \param jsonFilePath: JSON file path.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

/*
 * The input for this routine is a JSON file that you can download from the
 following URL:
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
SampleData.html.
 *
 * The JSON data uses the BatchWriteItem API request syntax. The JSON strings are
 * converted to AttributeValue objects. These AttributeValue objects will then
 generate
 * JSON strings when constructing the BatchWriteItem request, essentially
 outputting
 * their input.
 *
 * This is perhaps an artificial example, but it demonstrates the APIs.
 */

bool AwsDoc::DynamoDB::batchWriteItem(const Aws::String &jsonFilePath,
                                      const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    std::ifstream fileStream(jsonFilePath);
```

```
   if (!fileStream) {
       std::cerr << "Error: could not open file '" << jsonFilePath << "'."
                 << std::endl;
   }

   std::stringstream stringStream;
   stringStream << fileStream.rdbuf();
   Aws::Utils::Json::JsonValue jsonValue(stringStream);

   Aws::DynamoDB::Model::BatchWriteItemRequest batchWriteItemRequest;
   Aws::Map<Aws::String, Aws::Utils::Json::JsonView> level1Map =
jsonValue.View().GetAllObjects();
   for (const auto &level1Entry: level1Map) {
       const Aws::Utils::Json::JsonView &entriesView = level1Entry.second;
       const Aws::String &tableName = level1Entry.first;
       // The JSON entries at this level are as follows:
       //  key - table name
       //  value - list of request objects
       if (!entriesView.IsListType()) {
           std::cerr << "Error: JSON file entry '"
                     << tableName << "' is not a list." << std::endl;
           continue;
       }

       Aws::Utils::Array<Aws::Utils::Json::JsonView> entries =
entriesView.AsArray();

       Aws::Vector<Aws::DynamoDB::Model::WriteRequest> writeRequests;
       if (AwsDoc::DynamoDB::addWriteRequests(tableName, entries,
                                              writeRequests)) {
           batchWriteItemRequest.AddRequestItems(tableName, writeRequests);
       }
   }

   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

   Aws::DynamoDB::Model::BatchWriteItemOutcome outcome =
dynamoClient.BatchWriteItem(
           batchWriteItemRequest);

   if (outcome.IsSuccess()) {
       std::cout << "DynamoDB::BatchWriteItem was successful." << std::endl;
   }
```

```
    else {
        std::cerr << "Error with DynamoDB::BatchWriteItem. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return outcome.IsSuccess();
}

//! Convert requests in JSON format to a vector of WriteRequest objects.
/*!
  \sa addWriteRequests()
  \param tableName: Name of the table for the write operations.
  \param requestsJson: Request data in JSON format.
  \param writeRequests: Vector to receive the WriteRequest objects.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::addWriteRequests(const Aws::String &tableName,
                                        const
 Aws::Utils::Array<Aws::Utils::Json::JsonView> &requestsJson,

 Aws::Vector<Aws::DynamoDB::Model::WriteRequest> &writeRequests) {
    for (size_t i = 0; i < requestsJson.GetLength(); ++i) {
        const Aws::Utils::Json::JsonView &requestsEntry = requestsJson[i];
        if (!requestsEntry.IsObject()) {
            std::cerr << "Error: incorrect requestsEntry type "
                      << requestsEntry.WriteReadable() << std::endl;
            return false;
        }

        Aws::Map<Aws::String, Aws::Utils::Json::JsonView> requestsMap =
 requestsEntry.GetAllObjects();

        for (const auto &request: requestsMap) {
            const Aws::String &requestType = request.first;
            const Aws::Utils::Json::JsonView &requestJsonView = request.second;

            if (requestType == "PutRequest") {
                if (!requestJsonView.ValueExists("Item")) {
                    std::cerr << "Error: item key missing for requests "
                              << requestJsonView.WriteReadable() << std::endl;
                    return false;
                }
```

```
                Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 attributes;
                if (!getAttributeObjectsMap(requestJsonView.GetObject("Item"),
                                            attributes)) {
                    std::cerr << "Error getting attributes "
                              << requestJsonView.WriteReadable() << std::endl;
                    return false;
                }

                Aws::DynamoDB::Model::PutRequest putRequest;
                putRequest.SetItem(attributes);
                writeRequests.push_back(
                        Aws::DynamoDB::Model::WriteRequest().WithPutRequest(
                                putRequest));
            }
            else {
                std::cerr << "Error: unimplemented request type '" << requestType
                          << "'." << std::endl;
            }
        }
    }

    return true;
}

//! Generate a map of AttributeValue objects from JSON records.
/*!
  \sa getAttributeObjectsMap()
  \param jsonView: JSONView of attribute records.
  \param writeRequests: Map to receive the AttributeValue objects.
  \return bool: Function succeeded.
 */
bool
AwsDoc::DynamoDB::getAttributeObjectsMap(const Aws::Utils::Json::JsonView
 &jsonView,
                                         Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue> &attributes) {
    Aws::Map<Aws::String, Aws::Utils::Json::JsonView> objectsMap =
 jsonView.GetAllObjects();
    for (const auto &entry: objectsMap) {
        const Aws::String &attributeKey = entry.first;
        const Aws::Utils::Json::JsonView &attributeJsonView = entry.second;

        if (!attributeJsonView.IsObject()) {
```

```
            std::cerr << "Error: attribute not an object "
                      << attributeJsonView.WriteReadable() << std::endl;
            return false;
        }

        attributes.emplace(attributeKey,

 Aws::DynamoDB::Model::AttributeValue(attributeJsonView));
    }

    return true;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「BatchWriteItem」を参照してください。

CLI

AWS CLI

テーブルに複数の項目を追加するには

次の batch-write-item の例は、3 つの PutItem リクエストのバッチを使用して、MusicCollection テーブルに 3 つの新しい項目を追加します。また、このオペレーションによって消費される書き込み容量ユニットの数、およびオペレーションによって変更されるアイテムコレクションに関する情報も要求します。

```
aws dynamodb batch-write-item \
    --request-items file://request-items.json \
    --return-consumed-capacity INDEXES \
    --return-item-collection-metrics SIZE
```

request-items.json の内容:

```
{
    "MusicCollection": [
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "No One You Know"},
```

```
                    "SongTitle": {"S": "Call Me Today"},
                    "AlbumTitle": {"S": "Somewhat Famous"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "Acme Band"},
                    "SongTitle": {"S": "Happy Day"},
                    "AlbumTitle": {"S": "Songs About Life"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Artist": {"S": "No One You Know"},
                    "SongTitle": {"S": "Scared of My Shadow"},
                    "AlbumTitle": {"S": "Blue Sky Blues"}
                }
            }
        }
    ]
}
```

出力:

```
{
    "UnprocessedItems": {},
    "ItemCollectionMetrics": {
        "MusicCollection": [
            {
                "ItemCollectionKey": {
                    "Artist": {
                        "S": "No One You Know"
                    }
                },
                "SizeEstimateRangeGB": [
                    0.0,
                    1.0
                ]
            },
```

```
            {
                "ItemCollectionKey": {
                    "Artist": {
                        "S": "Acme Band"
                    }
                },
                "SizeEstimateRangeGB": [
                    0.0,
                    1.0
                ]
            }
        ]
    },
    "ConsumedCapacity": [
        {
            "TableName": "MusicCollection",
            "CapacityUnits": 6.0,
            "Table": {
                "CapacityUnits": 3.0
            },
            "LocalSecondaryIndexes": {
                "AlbumTitleIndex": {
                    "CapacityUnits": 3.0
                }
            }
        }
    ]
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「バッチオペレーション」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「BatchWriteItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovieBatch adds a slice of movies to the DynamoDB table. The function sends
// batches of 25 movies to DynamoDB until all movies are added or it reaches the
// specified maximum.
func (basics TableBasics) AddMovieBatch(ctx context.Context, movies []Movie,
 maxMovies int) (int, error) {
 var err error
 var item map[string]types.AttributeValue
```

```
 written := 0
 batchSize := 25 // DynamoDB allows a maximum batch size of 25 items.
 start := 0
 end := start + batchSize
 for start < maxMovies && start < len(movies) {
  var writeReqs []types.WriteRequest
  if end > len(movies) {
   end = len(movies)
  }
  for _, movie := range movies[start:end] {
   item, err = attributevalue.MarshalMap(movie)
   if err != nil {
    log.Printf("Couldn't marshal movie %v for batch writing. Here's why: %v\n",
 movie.Title, err)
   } else {
    writeReqs = append(
     writeReqs,
     types.WriteRequest{PutRequest: &types.PutRequest{Item: item}},
    )
   }
  }
  _, err = basics.DynamoDbClient.BatchWriteItem(ctx,
 &dynamodb.BatchWriteItemInput{
   RequestItems: map[string][]types.WriteRequest{basics.TableName: writeReqs}})
  if err != nil {
   log.Printf("Couldn't add a batch of movies to %v. Here's why: %v\n",
 basics.TableName, err)
  } else {
   written += len(writeReqs)
  }
  start = end
  end += batchSize
 }

 return written, err
}
```

この例で使用している Movie struct を定義します。

```
import (
```

```
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「BatchWriteItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

サービスクライアントを使用して、多数の項目をテーブルに挿入します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.BatchWriteItemResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutRequest;
import software.amazon.awssdk.services.dynamodb.model.WriteRequest;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development environment,
 including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class BatchWriteItems {
    public static void main(String[] args){
        final String usage = """
```

```
                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music).\s
                """;

        String tableName = "Music";
        Region region = Region.US_EAST_1;
        DynamoDbClient dynamoDbClient = DynamoDbClient.builder()
            .region(region)
            .build();

        addBatchItems(dynamoDbClient, tableName);
    }

    public static void addBatchItems(DynamoDbClient dynamoDbClient, String
tableName) {
        // Specify the updates you want to perform.
        List<WriteRequest> writeRequests = new ArrayList<>();

        // Set item 1.
        Map<String, AttributeValue> item1Attributes = new HashMap<>();
        item1Attributes.put("Artist",
AttributeValue.builder().s("Artist1").build());
        item1Attributes.put("Rating", AttributeValue.builder().s("5").build());
        item1Attributes.put("Comments", AttributeValue.builder().s("Great
song!").build());
        item1Attributes.put("SongTitle",
AttributeValue.builder().s("SongTitle1").build());

writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item1Attri

        // Set item 2.
        Map<String, AttributeValue> item2Attributes = new HashMap<>();
        item2Attributes.put("Artist",
AttributeValue.builder().s("Artist2").build());
        item2Attributes.put("Rating", AttributeValue.builder().s("4").build());
        item2Attributes.put("Comments", AttributeValue.builder().s("Nice
melody.").build());
        item2Attributes.put("SongTitle",
AttributeValue.builder().s("SongTitle2").build());
```

```
 writeRequests.add(WriteRequest.builder().putRequest(PutRequest.builder().item(item2Attri

        try {
            // Create the BatchWriteItemRequest.
            BatchWriteItemRequest batchWriteItemRequest =
 BatchWriteItemRequest.builder()
                .requestItems(Map.of(tableName, writeRequests))
                .build();

            // Execute the BatchWriteItem operation.
            BatchWriteItemResponse batchWriteItemResponse =
 dynamoDbClient.batchWriteItem(batchWriteItemRequest);

            // Process the response.
            System.out.println("Batch write successful: " +
 batchWriteItemResponse);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

拡張クライアントを使用して、多数の項目をテーブルに挿入します。

```
import com.example.dynamodb.Customer;
import com.example.dynamodb.Music;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.Key;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;
import
 software.amazon.awssdk.enhanced.dynamodb.model.BatchWriteItemEnhancedRequest;
import software.amazon.awssdk.enhanced.dynamodb.model.WriteBatch;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
```

```
import java.time.ZoneOffset;

/*
 * Before running this code example, create an Amazon DynamoDB table named
 Customer with these columns:
 *   - id - the id of the record that is the key
 *   - custName - the customer name
 *   - email - the email value
 *   - registrationDate - an instant value when the item was added to the table
 *
 * Also, ensure that you have set up your development environment, including your
 credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class EnhancedBatchWriteItems {
        public static void main(String[] args) {
                Region region = Region.US_EAST_1;
                DynamoDbClient ddb = DynamoDbClient.builder()
                                .region(region)
                                .build();
                DynamoDbEnhancedClient enhancedClient =
 DynamoDbEnhancedClient.builder()
                                .dynamoDbClient(ddb)
                                .build();
                putBatchRecords(enhancedClient);
                ddb.close();
        }

        public static void putBatchRecords(DynamoDbEnhancedClient enhancedClient)
 {
                try {
                        DynamoDbTable<Customer> customerMappedTable =
 enhancedClient.table("Customer",
                                        TableSchema.fromBean(Customer.class));
                        DynamoDbTable<Music> musicMappedTable =
 enhancedClient.table("Music",
                                        TableSchema.fromBean(Music.class));
                        LocalDate localDate = LocalDate.parse("2020-04-07");
                        LocalDateTime localDateTime = localDate.atStartOfDay();
```

```
                        Instant instant =
localDateTime.toInstant(ZoneOffset.UTC);

                        Customer record2 = new Customer();
                        record2.setCustName("Fred Pink");
                        record2.setId("id110");
                        record2.setEmail("fredp@noserver.com");
                        record2.setRegistrationDate(instant);

                        Customer record3 = new Customer();
                        record3.setCustName("Susan Pink");
                        record3.setId("id120");
                        record3.setEmail("spink@noserver.com");
                        record3.setRegistrationDate(instant);

                        Customer record4 = new Customer();
                        record4.setCustName("Jerry orange");
                        record4.setId("id101");
                        record4.setEmail("jorange@noserver.com");
                        record4.setRegistrationDate(instant);

                        BatchWriteItemEnhancedRequest
batchWriteItemEnhancedRequest = BatchWriteItemEnhancedRequest
                                        .builder()
                                        .writeBatches(

WriteBatch.builder(Customer.class) // add items to the Customer

         // table

.mappedTableResource(customerMappedTable)

.addPutItem(builder -> builder.item(record2))

.addPutItem(builder -> builder.item(record3))

.addPutItem(builder -> builder.item(record4))
                                                                        .build(),

WriteBatch.builder(Music.class) // delete an item from the Music

      // table

.mappedTableResource(musicMappedTable)
```

```
 .addDeleteItem(builder -> builder.key(

       Key.builder().partitionValue(

                        "Famous Band")

                        .build()))
                                                                    .build())
                                      .build();

                        // Add three items to the Customer table and delete one
 item from the Music
                        // table.

 enhancedClient.batchWriteItem(batchWriteItemEnhancedRequest);
                        System.out.println("done");

                } catch (DynamoDbException e) {
                        System.err.println(e.getMessage());
                        System.exit(1);
                }
        }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「BatchWriteItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「BatchWrite」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";
import { readFileSync } from "node:fs";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const file = readFileSync(
    `${dirname}../../../../../resources/sample_files/movies.json`,
  );

  const movies = JSON.parse(file.toString());

  // chunkArray is a local convenience function. It takes an array and returns
  // a generator function. The generator function yields every N items.
  const movieChunks = chunkArray(movies, 25);

  // For every chunk of 25 movies, make one BatchWrite request.
  for (const chunk of movieChunks) {
    const putRequests = chunk.map((movie) => ({
      PutRequest: {
        Item: movie,
      },
    }));

    const command = new BatchWriteCommand({
      RequestItems: {
        // An existing table is required. A composite key of 'title' and 'year'
 is recommended
        // to account for duplicate titles.
        BatchWriteMoviesTable: putRequests,
```

```
      },
    });

    await docClient.send(command);
  }
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchWriteItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: [
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
```

```
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
    ],
  },
};

ddb.batchWriteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchWriteItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    public function writeBatch(string $TableName, array $Batch, int $depth = 2)
    {
        if (--$depth <= 0) {
            throw new Exception("Max depth exceeded. Please try with fewer batch
 items or increase depth.");
        }

        $marshal = new Marshaler();
        $total = 0;
```

```
        foreach (array_chunk($Batch, 25) as $Items) {
            foreach ($Items as $Item) {
                $BatchWrite['RequestItems'][$TableName][] = ['PutRequest' =>
 ['Item' => $marshal->marshalItem($Item)]];
            }
            try {
                echo "Batching another " . count($Items) . " for a total of " .
 ($total += count($Items)) . " items!\n";
                $response = $this->dynamoDbClient->batchWriteItem($BatchWrite);
                $BatchWrite = [];
            } catch (Exception $e) {
                echo "uh oh...";
                echo $e->getMessage();
                die();
            }
            if ($total >= 250) {
                echo "250 movies is probably enough. Right? We can stop there.
\n";
                break;
            }
        }
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「BatchWriteItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 新しい項目を作成する、または既存の項目を「Music 」DynamoDB テーブルおよび「Songs」DynamoDB テーブルの新しい項目で置き換えます。

```
$item = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
    AlbumTitle = 'Somewhat Famous'
    Price = 1.94
    Genre = 'Country'
    CriticRating = 10.0
} | ConvertTo-DDBItem
```

```
$writeRequest = New-Object Amazon.DynamoDBv2.Model.WriteRequest
$writeRequest.PutRequest = [Amazon.DynamoDBv2.Model.PutRequest]$item

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
    'Songs' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
}

Set-DDBBatchItem -RequestItem $requestItem
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「BatchWriteItem」を参照してください。

Tools for PowerShell V5

例 1: 新しい項目を作成する、または既存の項目を「Music 」DynamoDB テーブルおよび「Songs」DynamoDB テーブルの新しい項目で置き換えます。

```
$item = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
    AlbumTitle = 'Somewhat Famous'
    Price = 1.94
    Genre = 'Country'
    CriticRating = 10.0
} | ConvertTo-DDBItem

$writeRequest = New-Object Amazon.DynamoDBv2.Model.WriteRequest
$writeRequest.PutRequest = [Amazon.DynamoDBv2.Model.PutRequest]$item

$requestItem = @{
    'Music' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
    'Songs' = [Amazon.DynamoDBv2.Model.WriteRequest]($writeRequest)
}

Set-DDBBatchItem -RequestItem $requestItem
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「BatchWriteItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def write_batch(self, movies):
        """
        Fills an Amazon DynamoDB table with the specified data, using the Boto3
        Table.batch_writer() function to put the items in the table.
        Inside the context manager, Table.batch_writer builds a list of
        requests. On exiting the context manager, Table.batch_writer starts
sending
        batches of write requests to Amazon DynamoDB and automatically
        handles chunking, buffering, and retrying.

        :param movies: The data to put in the table. Each item must contain at
least
                       the keys required by the schema that was specified when
the
                       table was created.
        """
        try:
            with self.table.batch_writer() as writer:
                for movie in movies:
                    writer.put_item(Item=movie)
        except ClientError as err:
            logger.error(
                "Couldn't load data into table %s. Here's why: %s: %s",
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「BatchWriteItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Fills an Amazon DynamoDB table with the specified data. Items are sent in
  # batches of 25 until all items are written.
  #
  # @param movies [Enumerable] The data to put in the table. Each item must
 contain at least
  #                            the keys required by the schema that was specified
 when the
  #                            table was created.
  def write_batch(movies)
    index = 0
    slice_size = 25
    while index < movies.length
      movie_items = []
      movies[index, slice_size].each do |movie|
        movie_items.append({ put_request: { item: movie } })
      end
      @dynamo_resource.client.batch_write_item({ request_items: { @table.name =>
 movie_items } })
      index += slice_size
    end
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts(
      "Couldn't load data into table #{@table.name}. Here's why:"
```

```
    )
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「BatchWriteItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Populate the movie database from the specified JSON file.
    ///
    /// - Parameter jsonPath: Path to a JSON file containing movie data.
    ///
    func populate(jsonPath: String) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Create a Swift `URL` and use it to load the file into a `Data`
            // object. Then decode the JSON into an array of `Movie` objects.

            let fileUrl = URL(fileURLWithPath: jsonPath)
            let jsonData = try Data(contentsOf: fileUrl)

            var movieList = try JSONDecoder().decode([Movie].self, from:
 jsonData)
```

```
            // Truncate the list to the first 200 entries or so for this example.

            if movieList.count > 200 {
                movieList = Array(movieList[...199])
            }

            // Before sending records to the database, break the movie list into
            // 25-entry chunks, which is the maximum size of a batch item
request.

            let count = movieList.count
            let chunks = stride(from: 0, to: count, by: 25).map {
                Array(movieList[$0 ..< Swift.min($0 + 25, count)])
            }

            // For each chunk, create a list of write request records and
populate
            // them with `PutRequest` requests, each specifying one movie from
the
            // chunk. Once the chunk's items are all in the `PutRequest` list,
            // send them to Amazon DynamoDB using the
            // `DynamoDBClient.batchWriteItem()` function.

            for chunk in chunks {
                var requestList: [DynamoDBClientTypes.WriteRequest] = []

                for movie in chunk {
                    let item = try await movie.getAsItem()
                    let request = DynamoDBClientTypes.WriteRequest(
                        putRequest: .init(
                            item: item
                        )
                    )
                    requestList.append(request)
                }

                let input = BatchWriteItemInput(requestItems: [tableName:
requestList])
                _ = try await client.batchWriteItem(input: input)
            }
        } catch {
            print("ERROR: populate:", dump(error))
            throw error
        }
```

```
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「BatchWriteItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **CreateTable** を使用する

次のサンプルコードは、CreateTable を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- グローバルセカンダリインデックスの高度なシナリオ
- グローバルセカンダリインデックスを含むテーブルを作成する
- ウォームスループットを有効にしたテーブルを作成する
- MREC を示すグローバルテーブルの作成と管理
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- グローバルセカンダリインデックスを管理する
- リソースベースのポリシーを管理する
- 属性ベースのアクセス制御を設定する
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する
- ローカルセカンダリインデックスを操作する
- リソースのタグ付けを操作する
- Streams と Time-to-Live を操作する
- テーブルの暗号化を操作する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Creates a new Amazon DynamoDB table and then waits for the new
    /// table to become active.
    /// </summary>
    /// <param name="tableName">The name of the table to create.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> CreateMovieTableAsync(string tableName)
    {
        try
        {
            var response = await _amazonDynamoDB.CreateTableAsync(new
 CreateTableRequest
            {
                TableName = tableName,
                AttributeDefinitions = new List<AttributeDefinition>()
                {
                    new AttributeDefinition
                    {
                        AttributeName = "title",
                        AttributeType = ScalarAttributeType.S,
                    },
                    new AttributeDefinition
                    {
                        AttributeName = "year",
                        AttributeType = ScalarAttributeType.N,
                    },
                },
                KeySchema = new List<KeySchemaElement>()
                {
                    new KeySchemaElement
```

```
                    {
                        AttributeName = "year",
                        KeyType = KeyType.HASH,
                    },
                    new KeySchemaElement
                    {
                        AttributeName = "title",
                        KeyType = KeyType.RANGE,
                    },
                },
                BillingMode = BillingMode.PAY_PER_REQUEST,
            });

            // Wait until the table is ACTIVE and then report success.
            Console.Write("Waiting for table to become active...");

            var request = new DescribeTableRequest
            {
                TableName = response.TableDescription.TableName,
            };

            TableStatus status;

            int sleepDuration = 2000;

            do
            {
                Thread.Sleep(sleepDuration);

                var describeTableResponse = await
_amazonDynamoDB.DescribeTableAsync(request);
                status = describeTableResponse.Table.TableStatus;

                Console.Write(".");
            }
            while (status != "ACTIVE");

            return status == TableStatus.ACTIVE;
        }
        catch (ResourceInUseException ex)
        {
            Console.WriteLine($"Table {tableName} already exists. {ex.Message}");
            throw;
        }
```

```
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while creating
 table {tableName}. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while creating table
 {tableName}. {ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「CreateTable」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
#############################################################################
# function dynamodb_create_table
#
# This function creates an Amazon DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table to create.
#       -a attribute_definitions -- JSON file path of a list of attributes and
 their types.
#       -k key_schema -- JSON file path of a list of attributes and their key
 types.
#
```

```
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_create_table() {
  local table_name attribute_definitions key_schema response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_create_table"
    echo "Creates an Amazon DynamoDB table with on-demand billing."
    echo " -n table_name  -- The name of the table to create."
    echo " -a attribute_definitions -- JSON file path of a list of attributes and
 their types."
    echo " -k key_schema -- JSON file path of a list of attributes and their key
 types."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:a:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      a) attribute_definitions="${OPTARG}" ;;
      k) key_schema="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
```

```
    return 1
  fi

  if [[ -z "$attribute_definitions" ]]; then
    errecho "ERROR: You must provide an attribute definitions json file path the
 -a parameter."
    usage
    return 1
  fi

  if [[ -z "$key_schema" ]]; then
    errecho "ERROR: You must provide a key schema json file path the -k
 parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    attribute_definitions:   $attribute_definitions"
  iecho "    key_schema:   $key_schema"
  iecho ""

  response=$(aws dynamodb create-table \
    --table-name "$table_name" \
    --attribute-definitions file://"$attribute_definitions" \
    --billing-mode PAY_PER_REQUEST \
    --key-schema file://"$key_schema" )

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports create-table operation failed.$response"
    return 1
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
#############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
```

```
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「CreateTable」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Create an Amazon DynamoDB table.
/*!
  \sa createTable()
  \param tableName: Name for the DynamoDB table.
  \param primaryKey: Primary key for the DynamoDB table.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::createTable(const Aws::String &tableName,
                                   const Aws::String &primaryKey,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
```

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Creating table " << tableName <<
              " with a simple primary key: \"" << primaryKey << "\"." <<
 std::endl;

    Aws::DynamoDB::Model::CreateTableRequest request;

    Aws::DynamoDB::Model::AttributeDefinition hashKey;
    hashKey.SetAttributeName(primaryKey);
    hashKey.SetAttributeType(Aws::DynamoDB::Model::ScalarAttributeType::S);
    request.AddAttributeDefinitions(hashKey);

    Aws::DynamoDB::Model::KeySchemaElement keySchemaElement;
    keySchemaElement.WithAttributeName(primaryKey).WithKeyType(
            Aws::DynamoDB::Model::KeyType::HASH);
    request.AddKeySchema(keySchemaElement);

    Aws::DynamoDB::Model::ProvisionedThroughput throughput;
    throughput.WithReadCapacityUnits(5).WithWriteCapacityUnits(5);
    request.SetProvisionedThroughput(throughput);
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::CreateTableOutcome &outcome =
 dynamoClient.CreateTable(
              request);
    if (outcome.IsSuccess()) {
        std::cout << "Table \""
                  << outcome.GetResult().GetTableDescription().GetTableName() <<
                  " created!" << std::endl;
    }
    else {
        std::cerr << "Failed to create table: " <<
 outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「CreateTable」を参照してください。

CLI

AWS CLI

例 1: タグ付きのテーブルを作成するには

次の create-table の例では、指定された属性とキースキーマを使用して、MusicCollection という名前のテーブルを作成します。このテーブルはプロビジョニングされたスループットを使用し、保存時にはデフォルトの AWS 所有の CMK を使用して暗号化されます。またこのコマンドは、Owner キーと blueTeam 値を使用して、テーブルにタグを適用します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --tags Key=Owner,Value=blueTeam
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
```

```
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "CREATING",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": "2020-05-26T16:04:41.627000-07:00",
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 2: オンデマンドモードでテーブルを作成するには

次の例は、プロビジョニングされたスループットモードではなく、オンデマンドモードを使用して MusicCollection というテーブルを作成します。これは、ワークロードが予測できないテーブルに役立ちます。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:44:10.807000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 3: テーブルを作成してカスタマーマネージド CMK で暗号化するには

次の例では、MusicCollection という名前のテーブルを作成し、カスタマーマネージド CMK を使用して暗号化します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T11:12:16.431000-07:00",
```

```
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "SSEDescription": {
            "Status": "ENABLED",
            "SSEType": "KMS",
            "KMSMasterKeyArn": "arn:aws:kms:us-west-2:123456789012:key/abcd1234-
abcd-1234-a123-ab1234a1b234"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 4: ローカルセカンダリインデックスを持つテーブルを作成するには

次の例では、指定された属性とキースキーマを使用して、AlbumTitleIndex という名前のローカルセカンダリインデックスを持つ MusicCollection という名前のテーブルを作成します。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-
definitions AttributeName=Artist,AttributeType=S AttributeName=SongTitle,AttributeType=S
 \
    --key-
schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\": \"Artist\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\": \"AlbumTitle\",\"KeyType\":\"RANGE\"}
```

```
                ],
                \"Projection\": {
                    \"ProjectionType\": \"INCLUDE\",
                    \"NonKeyAttributes\": [\"Genre\", \"Year\"]
                }
            }
        ]"
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
```

```
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Genre",
                        "Year"
                    ]
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 5: グローバルセカンダリインデックスを持つテーブルを作成するには

次の例では、GameTitleIndex という名前のグローバルセカンダリインデックスを持つ GameScores という名前のテーブルを作成します。ベーステーブルには、パーティションキー UserId とソートキー GameTitle があり、特定のゲームの個々のユーザーのベストス

コアを効率的に見つけることができます。一方、GSI にはパーティションキー GameTitle とソートキー TopScore があり、特定のゲームの全体的な最高スコアをすばやく見つけることができます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-schema AttributeName=UserId,KeyType=HASH \
                 AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GameTitleIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"GameTitle\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"TopScore\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {
                    \"ProjectionType\":\"INCLUDE\",
                    \"NonKeyAttributes\":[\"UserId\"]
                },
                \"ProvisionedThroughput\": {
                    \"ReadCapacityUnits\": 10,
                    \"WriteCapacityUnits\": 5
                }
            }
        ]"
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
```

```
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-26T17:28:15.602000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
```

```
                        "UserId"
                    ]
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 6: 複数のグローバルセカンダリインデックスを持つテーブルを一度に作成するには

次の例では、2 つのグローバルセカンダリインデックスを持つ GameScores という名前のテーブルを作成します。GSI スキーマはコマンドラインではなくファイルを介して渡されます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --global-secondary-indexes file://gsi.json
```

gsi.json の内容:

```
[
    {
        "IndexName": "GameTitleIndex",
```

```
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "TopScore",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        }
    },
    {
        "IndexName": "GameDateIndex",
        "KeySchema": [
            {
                "AttributeName": "GameTitle",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "Date",
                "KeyType": "RANGE"
            }
        ],
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        }
    }
]
```

出力:

```
{
```

```
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Date",
                "AttributeType": "S"
            },
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "TopScore",
                "AttributeType": "N"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-08-04T16:40:55.524000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "GameTitleIndex",
```

```
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "TopScore",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 5
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameTitleIndex"
            },
            {
                "IndexName": "GameDateIndex",
                "KeySchema": [
                    {
                        "AttributeName": "GameTitle",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "Date",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
```

```
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/index/GameDateIndex"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 7: ストリームが有効なテーブルを作成するには

次の例では、DynamoDB ストリームを有効にした状態の GameScores という名前のテーブルを作成します。各アイテムの新しいイメージと古いイメージの両方がストリームに書き込まれます。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=NEW_AND_OLD_IMAGES
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
```

```
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2020-05-27T10:49:34.056000-07:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "LatestStreamLabel": "2020-05-27T17:49:34.056",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2020-05-27T17:49:34.056"
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB テーブルの基本的なオペレーション」を参照してください。

例 8: Keys-Only ストリームが有効なテーブルを作成するには

次の例では、DynamoDB ストリームを有効にした状態の GameScores という名前のテーブルを作成します。変更された項目のキー属性のみがストリームに書き込まれます。

```
aws dynamodb create-table \
    --table-name GameScores \
```

```
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=TRUE,StreamViewType=KEYS_ONLY
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:45:34.140000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
```

```
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "KEYS_ONLY"
        },
        "LatestStreamLabel": "2023-05-25T18:45:34.140",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/2023-05-25T18:45:34.140",
        "DeletionProtectionEnabled": false
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB Streams の変更データキャプチャ」を参照してください。

例 9: Standard Infrequent Access クラスでテーブルを作成するには

次の例は、GameScores という名前のテーブルを作成し、Standard-Infrequent Access (DynamoDB Standard-IA) テーブルクラスを割り当てます。このテーブルクラスは、ストレージが主なコストとなるように最適化されています。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --table-class STANDARD_INFREQUENT_ACCESS
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
```

```
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T18:33:07.581000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "TableClassSummary": {
            "TableClass": "STANDARD_INFREQUENT_ACCESS"
        },
        "DeletionProtectionEnabled": false
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブルクラス」を参照してください。

例 10: 削除保護を有効にしたテーブルを作成するには

次の例では、GameScores というテーブルを作成し、削除保護を有効にします。

```
aws dynamodb create-table \
    --table-name GameScores \
    --attribute-
definitions AttributeName=UserId,AttributeType=S AttributeName=GameTitle,AttributeType=S
 \
```

```
    --key-
schema AttributeName=UserId,KeyType=HASH AttributeName=GameTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --deletion-protection-enabled
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "GameTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "UserId",
                "AttributeType": "S"
            }
        ],
        "TableName": "GameScores",
        "KeySchema": [
            {
                "AttributeName": "UserId",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "GameTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "CREATING",
        "CreationDateTime": "2023-05-25T23:02:17.093000+00:00",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 10,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
        "TableId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
        "DeletionProtectionEnabled": true
    }
```

```
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「削除保護の使用」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「CreateTable」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// CreateMovieTable creates a DynamoDB table with a composite primary key defined
 as
// a string sort key named `title`, and a numeric partition key named `year`.
// This function uses NewTableExistsWaiter to wait for the table to be created by
// DynamoDB before it returns.
func (basics TableBasics) CreateMovieTable(ctx context.Context)
 (*types.TableDescription, error) {
 var tableDesc *types.TableDescription
 table, err := basics.DynamoDbClient.CreateTable(ctx, &dynamodb.CreateTableInput{
  AttributeDefinitions: []types.AttributeDefinition{{
   AttributeName: aws.String("year"),
   AttributeType: types.ScalarAttributeTypeN,
  }, {
   AttributeName: aws.String("title"),
   AttributeType: types.ScalarAttributeTypeS,
  }},
  KeySchema: []types.KeySchemaElement{{
   AttributeName: aws.String("year"),
   KeyType:       types.KeyTypeHash,
  }, {
   AttributeName: aws.String("title"),
   KeyType:       types.KeyTypeRange,
  }},
  TableName:   aws.String(basics.TableName),
  BillingMode: types.BillingModePayPerRequest,
 })
 if err != nil {
  log.Printf("Couldn't create table %v. Here's why: %v\n", basics.TableName, err)
 } else {
  waiter := dynamodb.NewTableExistsWaiter(basics.DynamoDbClient)
  err = waiter.Wait(ctx, &dynamodb.DescribeTableInput{
   TableName: aws.String(basics.TableName)}, 5*time.Minute)
  if err != nil {
   log.Printf("Wait for table exists failed. Here's why: %v\n", err)
  }
  tableDesc = table.TableDescription
  log.Printf("Ccreating table test")
 }
 return tableDesc, err
}
```

- API の詳細については、『AWS SDK for Go API リファレンス』の「CreateTable」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.BillingMode;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.OnDemandThroughput;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 * <p>
 * For more information, see the following documentation topic:
 * <p>
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class CreateTable {
```

```
   public static void main(String[] args) {
       final String usage = """

           Usage:
               <tableName> <key>

           Where:
               tableName - The Amazon DynamoDB table to create (for example,
Music3).
               key - The key for the Amazon DynamoDB table (for example,
Artist).
           """;

       if (args.length != 2) {
           System.out.println(usage);
           System.exit(1);
       }

       String tableName = args[0];
       String key = args[1];
       System.out.println("Creating an Amazon DynamoDB table " + tableName + "
with a simple primary key: " + key);
       Region region = Region.US_EAST_1;
       DynamoDbClient ddb = DynamoDbClient.builder()
           .region(region)
           .build();

       String result = createTable(ddb, tableName, key);
       System.out.println("New table is " + result);
       ddb.close();
   }

   public static String createTable(DynamoDbClient ddb, String tableName, String
key) {
       DynamoDbWaiter dbWaiter = ddb.waiter();
       CreateTableRequest request = CreateTableRequest.builder()
           .attributeDefinitions(AttributeDefinition.builder()
               .attributeName(key)
               .attributeType(ScalarAttributeType.S)
               .build())
           .keySchema(KeySchemaElement.builder()
               .attributeName(key)
               .keyType(KeyType.HASH)
               .build())
```

```
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
 scales based on traffic.
            .tableName(tableName)
            .build();

        String newTable;
        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
 dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            newTable = response.tableDescription().tableName();
            return newTable;

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「CreateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    BillingMode: "PAY_PER_REQUEST",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「CreateTable」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
```

```
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「CreateTable」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun createNewTable(
    tableNameVal: String,
    key: String,
): String? {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
```

```
            attributeName = key
            keyType = KeyType.Hash
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef)
            keySchema = listOf(keySchemaVal)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        var tableArn: String
        val response = ddb.createTable(request)
        ddb.waitUntilTableExists {
            // suspend call
            tableName = tableNameVal
        }
        tableArn = response.tableDescription!!.tableArn.toString()
        println("Table $tableArn is ready")
        return tableArn
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「CreateTable」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成します。

```
        $tableName = "ddb_demo_table_$uuid";
```

```
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

    public function createTable(string $tableName, array $attributes)
    {
        $keySchema = [];
        $attributeDefinitions = [];
        foreach ($attributes as $attribute) {
            if (is_a($attribute, DynamoDBAttribute::class)) {
                $keySchema[] = ['AttributeName' => $attribute->AttributeName,
'KeyType' => $attribute->KeyType];
                $attributeDefinitions[] =
                    ['AttributeName' => $attribute->AttributeName,
'AttributeType' => $attribute->AttributeType];
            }
        }

        $this->dynamoDbClient->createTable([
            'TableName' => $tableName,
            'KeySchema' => $keySchema,
            'AttributeDefinitions' => $attributeDefinitions,
            'ProvisionedThroughput' => ['ReadCapacityUnits' => 10,
'WriteCapacityUnits' => 10],
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「CreateTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ「Thread」という名前のテーブルを作成します。テーブル

の作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

例 2: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ Thread という名前のテーブルを作成します。ローカルセカンダリインデックスも定義されます。ローカルセカンダリインデックスのキーは、テーブルのプライマリハッシュキー (ForumName) から自動的に設定されます。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
```

```
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

例 3: この例では、単一のパイプラインを使用して、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキー、およびローカルセカンダリインデックスを持つ「Thread」という名前のテーブルを作成する方法を示します。TableSchema がパイプラインまたは -Schema パラメータから提供されない場合、Add-DDBKeySchema と Add-DDBIndexSchema によって新しい TableSchema オブジェクトが作成されます。

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「CreateTable」を参照してください。

Tools for PowerShell V5

例 1: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ「Thread」という名前のテーブルを作成します。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyType RANGE -KeyDataType "S"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {}
```

例 2: この例では、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキーを持つ Thread という名前のテーブルを作成します。ローカルセカンダリインデックスも定義されます。ローカルセカンダリインデックスのキーは、テーブルのプライマリハッシュキー (ForumName) から自動的に設定されます。テーブルの作成に使用したスキーマは、図のように各 cmdlet にパイプ処理するか、-Schema パラメータを使用して指定できます。

```
$schema = New-DDBTableSchema
$schema | Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S"
$schema | Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S"
$schema | Add-DDBIndexSchema -IndexName "LastPostIndex" -RangeKeyName
 "LastPostDateTime" -RangeKeyDataType "S" -ProjectionType "keys_only"
$schema | New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
```

```
LocalSecondaryIndexes  : {LastPostIndex}
```

例 3: この例では、単一のパイプラインを使用して、「ForumName」(キータイプハッシュ) と「Subject」(キータイプ範囲) で構成されるプライマリキー、およびローカルセカンダリインデックスを持つ「Thread」という名前のテーブルを作成する方法を示します。TableSchema がパイプラインまたは -Schema パラメータから提供されない場合、Add-DDBKeySchema と Add-DDBIndexSchema によって新しい TableSchema オブジェクトが作成されます。

```
New-DDBTableSchema |
  Add-DDBKeySchema -KeyName "ForumName" -KeyDataType "S" |
  Add-DDBKeySchema -KeyName "Subject" -KeyDataType "S" |
  Add-DDBIndexSchema -IndexName "LastPostIndex" `
                     -RangeKeyName "LastPostDateTime" `
                     -RangeKeyDataType "S" `
                     -ProjectionType "keys_only" |
  New-DDBTable -TableName "Thread" -ReadCapacity 10 -WriteCapacity 5
```

出力する。

```
AttributeDefinitions   : {ForumName, LastPostDateTime, Subject}
TableName              : Thread
KeySchema              : {ForumName, Subject}
TableStatus            : CREATING
CreationDateTime       : 10/28/2013 4:39:49 PM
ProvisionedThroughput  : Amazon.DynamoDBv2.Model.ProvisionedThroughputDescription
TableSizeBytes         : 0
ItemCount              : 0
LocalSecondaryIndexes  : {LastPostIndex}
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「CreateTable」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

映画データを格納するためのテーブルを作成します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def create_table(self, table_name):
        """
        Creates an Amazon DynamoDB table that can be used to store movie data.
        The table uses the release year of the movie as the partition key and the
        title as the sort key.

        :param table_name: The name of the table to create.
        :return: The newly created table.
        """
```

```
        try:
            self.table = self.dyn_resource.create_table(
                TableName=table_name,
                KeySchema=[
                    {"AttributeName": "year", "KeyType": "HASH"},  # Partition
key
                    {"AttributeName": "title", "KeyType": "RANGE"},  # Sort key
                ],
                AttributeDefinitions=[
                    {"AttributeName": "year", "AttributeType": "N"},
                    {"AttributeName": "title", "AttributeType": "S"},
                ],
                BillingMode='PAY_PER_REQUEST',
            )
            self.table.wait_until_exists()
        except ClientError as err:
            logger.error(
                "Couldn't create table %s. Here's why: %s: %s",
                table_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return self.table
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「CreateTable」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Creates an Amazon DynamoDB table that can be used to store movie data.
  # The table uses the release year of the movie as the partition key and the
  # title as the sort key.
  #
  # @param table_name [String] The name of the table to create.
  # @return [Aws::DynamoDB::Table] The newly created table.
  def create_table(table_name)
    @table = @dynamo_resource.create_table(
      table_name: table_name,
      key_schema: [
        { attribute_name: 'year', key_type: 'HASH' }, # Partition key
        { attribute_name: 'title', key_type: 'RANGE' } # Sort key
      ],
      attribute_definitions: [
        { attribute_name: 'year', attribute_type: 'N' },
        { attribute_name: 'title', attribute_type: 'S' }
      ],
      billing_mode: 'PAY_PER_REQUEST'
    )
    @dynamo_resource.client.wait_until(:table_exists, table_name: table_name)
    @table
  rescue Aws::DynamoDB::Errors::ServiceError => e
    @logger.error("Failed create table #{table_name}:\n#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「CreateTable」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn create_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<CreateTableOutput, Error> {
    let a_name: String = key.into();
    let table_name: String = table.into();

    let ad = AttributeDefinition::builder()
        .attribute_name(&a_name)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .map_err(Error::BuildError)?;

    let ks = KeySchemaElement::builder()
        .attribute_name(&a_name)
        .key_type(KeyType::Hash)
        .build()
        .map_err(Error::BuildError)?;

    let create_table_response = client
        .create_table()
        .table_name(table_name)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await;

    match create_table_response {
        Ok(out) => {
            println!("Added table {} with key {}", table, key);
```

```
            Ok(out)
        }
        Err(e) => {
            eprintln!("Got an error creating table:");
            eprintln!("{}", e);
            Err(Error::unhandled(e))
        }
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「CreateTable」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lt_keyschema) = VALUE /aws1/cl_dynkeyschemaelement=>tt_keyschema(
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'year'
                                              iv_keytype = 'HASH' ) )
          ( NEW /aws1/cl_dynkeyschemaelement( iv_attributename = 'title'
                                              iv_keytype = 'RANGE' ) ) ).
        DATA(lt_attributedefinitions) = VALUE /aws1/
cl_dynattributedefn=>tt_attributedefinitions(
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'year'
                                           iv_attributetype = 'N' ) )
          ( NEW /aws1/cl_dynattributedefn( iv_attributename = 'title'
                                           iv_attributetype = 'S' ) ) ).

        " Adjust read/write capacities as desired.
        DATA(lo_dynprovthroughput)  = NEW /aws1/cl_dynprovthroughput(
          iv_readcapacityunits = 5
          iv_writecapacityunits = 5 ).
```

```
        oo_result = lo_dyn->createtable(
          it_keyschema = lt_keyschema
          iv_tablename = iv_table_name
          it_attributedefinitions = lt_attributedefinitions
          io_provisionedthroughput = lo_dynprovthroughput ).
        " Table creation can take some time. Wait till table exists before
returning.
        lo_dyn->get_waiter( )->tableexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'DynamoDB Table' && iv_table_name && 'created.' TYPE 'I'.
        " This exception can happen if the table already exists.
      CATCH /aws1/cx_dynresourceinuseex INTO DATA(lo_resourceinuseex).
        DATA(lv_error) = |"{ lo_resourceinuseex->av_err_code }" -
{ lo_resourceinuseex->av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「CreateTable」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    ///
    /// Create a movie table in the Amazon DynamoDB data store.
    ///
    private func createTable() async throws {
        do {
            guard let client = self.ddbClient else {
```

```
                throw MoviesError.UninitializedClient
            }

            let input = CreateTableInput(
                attributeDefinitions: [
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"year", attributeType: .n),
                    DynamoDBClientTypes.AttributeDefinition(attributeName:
"title", attributeType: .s)
                ],
                billingMode: DynamoDBClientTypes.BillingMode.payPerRequest,
                keySchema: [
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "year",
keyType: .hash),
                    DynamoDBClientTypes.KeySchemaElement(attributeName: "title",
keyType: .range)
                ],
                tableName: self.tableName
            )
            let output = try await client.createTable(input: input)
            if output.tableDescription == nil {
                throw MoviesError.TableNotFound
            }
        } catch {
            print("ERROR: createTable:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「CreateTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **`DeleteItem`** を使用する

次のサンプルコードは、`DeleteItem` を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- 条件付きオペレーションを使用する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Deletes a single item from a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table from which the item
    /// will be deleted.</param>
    /// <param name="movieToDelete">A movie object containing the title and
    /// year of the movie to delete.</param>
    /// <returns>A Boolean value indicating the success or failure of the
    /// delete operation.</returns>
    public async Task<bool> DeleteItemAsync(
        string tableName,
        Movie movieToDelete)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = movieToDelete.Title },
                ["year"] = new AttributeValue { N =
movieToDelete.Year.ToString() },
            };

            var request = new DeleteItemRequest { TableName = tableName, Key =
key, };
```

```
            await _amazonDynamoDB.DeleteItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting item.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeleteItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_delete_item
#
```

```
# This function deletes an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_item() {
  local table_name keys response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_delete_item"
    echo "Delete an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to delete."
    echo ""
  }
  while getopts "n:k:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
```

```
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho ""

  response=$(aws dynamodb delete-item \
    --table-name "$table_name" \
    --key file://"$keys")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-item operation failed.$response"
    return 1
  fi

  return 0

}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
```

```
    echo "$@"
  fi
}

################################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
```

```
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Delete an item from an Amazon DynamoDB table.
/*!
  \sa deleteItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::deleteItem(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteItemRequest request;

    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));
```

```
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteItemOutcome &outcome =
 dynamoClient.DeleteItem(
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Item \"" << partitionValue << "\" deleted!" << std::endl;
    }
    else {
        std::cerr << "Failed to delete item: " << outcome.GetError().GetMessage()
                  << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
```

```
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「DeleteItem」を参照してください。

CLI

AWS CLI

例 1: 項目を削除するには

次の delete-item の例は、MusicCollection テーブルから項目を削除し、削除した項目とそのリクエストで使用された容量に関する詳細を取得します。

```
aws dynamodb delete-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-values ALL_OLD \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

key.json の内容:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Scared of My Shadow"}
}
```

出力:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Blue Sky Blues"
        },
        "Artist": {
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Scared of My Shadow"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 2.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

例 2: 条件付きで項目を削除するには

次の例では、ProductCategory が Sporting Goods または Gardening Supplies で、その価格が 500 および 600 の場合のみ、ProductCatalog テーブルから項目を削除します。削除された項目に関する詳細が返されます。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{"Id":{"N":"456"}}' \
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (#P
 between :lo and :hi)" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

names.json の内容:

```
{
    "#P": "Price"
}
```

values.json の内容:

```
{
    ":cat1": {"S": "Sporting Goods"},
    ":cat2": {"S": "Gardening Supplies"},
    ":lo": {"N": "500"},
    ":hi": {"N": "600"}
}
```

出力:

```
{
    "Attributes": {
        "Id": {
            "N": "456"
        },
        "Price": {
            "N": "550"
        },
        "ProductCategory": {
            "S": "Sporting Goods"
        }
```

```
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// DeleteMovie removes a movie from the DynamoDB table.
func (basics TableBasics) DeleteMovie(ctx context.Context, movie Movie) error {
 _, err := basics.DynamoDbClient.DeleteItem(ctx, &dynamodb.DeleteItemInput{
  TableName: aws.String(basics.TableName), Key: movie.GetKey(),
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}
```

```
// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスのの「DeleteItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
```

```
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DeleteItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyval>

                Where:
                    tableName - The Amazon DynamoDB table to delete the item from
 (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to delete
 (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Deleting item \"%s\" from %s\n", keyVal, tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }
```

```
    public static void deleteDynamoDBItem(DynamoDbClient ddb, String tableName,
 String key, String keyVal) {
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        DeleteItemRequest deleteReq = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(keyToGet)
                .build();

        try {
            ddb.deleteItem(deleteReq);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「DeleteItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「DeleteCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new DeleteCommand({
    TableName: "Sodas",
    Key: {
      Flavor: "Cola",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「DeleteItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルから項目を削除します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
```

```
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

DynamoDB ドキュメントクライアントを使用して、テーブルから項目を削除します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。

- API の詳細については、AWS SDK for JavaScript API リファレンスの「DeleteItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun deleteDynamoDBItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
        DeleteItemRequest {
            tableName = tableNameVal
            key = keyToGet
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteItem(request)
        println("Item with key matching $keyVal was deleted")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「DeleteItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        $key = [
            'Item' => [
                'title' => [
                    'S' => $movieName,
                ],
                'year' => [
                    'N' => $movieYear,
                ],
            ]
        ];

        $service->deleteItemByKey($tableName, $key);
        echo "But, bad news, this was a trap. That movie has now been deleted
because of your rating...harsh.\n";

    public function deleteItemByKey(string $tableName, array $key)
    {
        $this->dynamoDbClient->deleteItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「DeleteItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたキーと一致する DynamoDB 項目を削除します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem
Remove-DDBItem -TableName 'Music' -Key $key -Confirm:$false
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「DeleteItem」を参照してください。

Tools for PowerShell V5

例 1: 指定されたキーと一致する DynamoDB 項目を削除します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem
Remove-DDBItem -TableName 'Music' -Key $key -Confirm:$false
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「DeleteItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.
```

```
    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def delete_movie(self, title, year):
        """
        Deletes a movie from the table.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        """
        try:
            self.table.delete_item(Key={"year": year, "title": title})
        except ClientError as err:
            logger.error(
                "Couldn't delete movie %s. Here's why: %s: %s",
                title,
                err.response["Error"]["Code"],
```

```
                err.response["Error"]["Message"],
            )
            raise
```

項目が特定の条件を満たす場合にのみ削除されるように条件を指定できます。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def delete_underrated_movie(self, title, year, rating):
        """
        Deletes a movie only if it is rated below a specified value. By using a
        condition expression in a delete operation, you can specify that an item
 is
        deleted only when it meets certain criteria.

        :param title: The title of the movie to delete.
        :param year: The release year of the movie to delete.
        :param rating: The rating threshold to check before deleting the movie.
        """
        try:
            self.table.delete_item(
                Key={"year": year, "title": title},
                ConditionExpression="info.rating <= :val",
                ExpressionAttributeValues={":val": Decimal(str(rating))},
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
 "ConditionalCheckFailedException":
                logger.warning(
                    "Didn't delete %s because its rating is greater than %s.",
                    title,
                    rating,
                )
            else:
                logger.error(
                    "Couldn't delete movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
```

```
                    err.response["Error"]["Message"],
                )
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「DeleteItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Deletes a movie from the table.
  #
  # @param title [String] The title of the movie to delete.
  # @param year [Integer] The release year of the movie to delete.
  def delete_item(title, year)
    @table.delete_item(key: { 'year' => year, 'title' => title })
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete movie #{title}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「DeleteItem」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn delete_item(
    client: &Client,
    table: &str,
    key: &str,
    value: &str,
) -> Result<DeleteItemOutput, Error> {
    match client
        .delete_item()
        .table_name(table)
        .key(key, AttributeValue::S(value.into()))
        .send()
        .await
    {
        Ok(out) => {
            println!("Deleted item from table");
            Ok(out)
        }
        Err(e) => Err(Error::unhandled(e)),
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「DeleteItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lo_resp) = lo_dyn->deleteitem(
          iv_tablename                = iv_table_name
          it_key                      = it_key_input ).
        MESSAGE 'Deleted one item.' TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、AWS SDK for SAP ABAP API リファレンスの「DeleteItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB
```

```
    /// Delete a movie, given its title and release year.
    ///
    /// - Parameters:
    ///   - title: The movie's title.
    ///   - year: The movie's release year.
    ///
    func delete(title: String, year: Int) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            _ = try await client.deleteItem(input: input)
        } catch {
            print("ERROR: delete:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「DeleteItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DeleteTable** を使用する

次のサンプルコードは、DeleteTable を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- グローバルセカンダリインデックスを管理する
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Deletes a DynamoDB table.
    /// </summary>
    /// <param name="tableName">The name of the table to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteTableAsync(string tableName)
    {
        try
        {
            var request = new DeleteTableRequest
            {
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.DeleteTableAsync(request);

            Console.WriteLine($"Table {response.TableDescription.TableName}
 successfully deleted.");
            return true;

        }
```

```
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found and cannot be
deleted. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while deleting
table {tableName}. {ex.Message}");
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while deleting table
{tableName}. {ex.Message}");
            return false;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeleteTable」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_delete_table
#
# This function deletes a DynamoDB table.
#
# Parameters:
```

```
#       -n table_name  -- The name of the table to delete.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_delete_table() {
  local table_name response
  local option OPTARG # Required to use getopts command in a function.

  # bashsupport disable=BP5008
  function usage() {
    echo "function dynamodb_delete_table"
    echo "Deletes an Amazon DynamoDB table."
    echo " -n table_name  -- The name of the table to delete."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho ""
```

```
  response=$(aws dynamodb delete-table \
    --table-name "$table_name")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports delete-table operation failed.$response"
    return 1
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
```

```
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
########################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteTable」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Delete an Amazon DynamoDB table.
/*!
  \sa deleteTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteTable(const Aws::String &tableName,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(tableName);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「DeleteTable」を参照してください。

CLI

AWS CLI

テーブルを削除するには

以下の `delete-table` の例は `MusicCollection` テーブルを削除します。

```
aws dynamodb delete-table \
    --table-name MusicCollection
```

出力:

```
{
    "TableDescription": {
        "TableStatus": "DELETING",
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableName": "MusicCollection",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブルの削除」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteTable」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// DeleteTable deletes the DynamoDB table and all of its data.
func (basics TableBasics) DeleteTable(ctx context.Context) error {
 _, err := basics.DynamoDbClient.DeleteTable(ctx, &dynamodb.DeleteTableInput{
  TableName: aws.String(basics.TableName)})
 if err != nil {
  log.Printf("Couldn't delete table %v. Here's why: %v\n", basics.TableName, err)
 }
```

```
 return err
}
```

- API の詳細については、『AWS SDK for Go API リファレンス』の「DeleteTable」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class DeleteTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
```

```
                    tableName - The Amazon DynamoDB table to delete (for example,
 Music3).

                **Warning** This program will delete the table that you specify!
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Deleting the Amazon DynamoDB table %s...\n",
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
 {
        DeleteTableRequest request = DeleteTableRequest.builder()
                .tableName(tableName)
                .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンス の「DeleteTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「DeleteTable」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「DeleteTable」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun deleteDynamoDBTable(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
```

```
        println("$tableNameVal was deleted")
    }
}
```

- API の詳細については、『AWS SDK for Kotlin API リファレンス』の「DeleteTable」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    public function deleteTable(string $TableName)
    {
        $this->customWaiter(function () use ($TableName) {
            return $this->dynamoDbClient->deleteTable([
                'TableName' => $TableName,
            ]);
        });
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「DeleteTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたテーブルを削除します。操作を続行する前に確認画面が表示されます。

```
Remove-DDBTable -TableName "myTable"
```

例 2: 指定されたテーブルを削除します。操作を続行する前に確認画面は表示されません。

```
Remove-DDBTable -TableName "myTable" -Force
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「DeleteTable」を参照してください。

Tools for PowerShell V5

例 1: 指定されたテーブルを削除します。操作を続行する前に確認画面が表示されます。

```
Remove-DDBTable -TableName "myTable"
```

例 2: 指定されたテーブルを削除します。操作を続行する前に確認画面は表示されません。

```
Remove-DDBTable -TableName "myTable" -Force
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「DeleteTable」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
```

```
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def delete_table(self):
        """
        Deletes the table.
        """
        try:
            self.table.delete()
            self.table = None
        except ClientError as err:
            logger.error(
                "Couldn't delete table. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンス の「DeleteTable」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Deletes the table.
  def delete_table
    @table.delete
    @table = nil
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't delete table. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンス の「DeleteTable」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn delete_table(client: &Client, table: &str) ->
 Result<DeleteTableOutput, Error> {
    let resp = client.delete_table().table_name(table).send().await;

    match resp {
        Ok(out) => {
            println!("Deleted table");
            Ok(out)
        }
        Err(e) => Err(Error::Unhandled(e.into())),
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「DeleteTable」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        lo_dyn->deletetable( iv_tablename = iv_table_name ).
        " Wait till the table is actually deleted.
```

```
        lo_dyn->get_waiter( )->tablenotexists(
          iv_max_wait_time = 200
          iv_tablename     = iv_table_name ).
        MESSAGE 'Table ' && iv_table_name && ' deleted.' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table ' && iv_table_name && ' does not exist' TYPE 'E'.
      CATCH /aws1/cx_dynresourceinuseex.
        MESSAGE 'The table cannot be deleted since it is in use' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DeleteTable」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    ///
    /// Deletes the table from Amazon DynamoDB.
    ///
    func deleteTable() async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = DeleteTableInput(
                tableName: self.tableName
            )
            _ = try await client.deleteTable(input: input)
        } catch {
```

```
            print("ERROR: deleteTable:", dump(error))
            throw error
        }
    }
```

- API の詳細については、AWS SDK for Swift API リファレンス の「DeleteTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DescribeTable** を使用する

次のサンプルコードは、DescribeTable を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- MREC を示すグローバルテーブルの作成と管理
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する
- Streams と Time-to-Live を操作する
- テーブルの暗号化を操作する

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    private static async Task GetTableInformation()
    {
        Console.WriteLine("\n*** Retrieving table information ***");

        var response = await Client.DescribeTableAsync(new DescribeTableRequest
        {
            TableName = ExampleTableName
        });

        var table = response.Table;
        Console.WriteLine($"Name: {table.TableName}");
        Console.WriteLine($"# of items: {table.ItemCount}");

    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DescribeTable」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
###############################################################################
# function dynamodb_describe_table
#
# This function returns the status of a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#
#  Response:
#       - TableStatus:
#     And:
#       0 - Table is active.
```

```
#       1 - If it fails.
###############################################################################
function dynamodb_describe_table {
  local table_name
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_describe_table"
    echo "Describe the status of a DynamoDB table."
    echo "  -n table_name  -- The name of the table."
    echo ""
  }

  # Retrieve the calling parameters.
  while getopts "n:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  local table_status
    table_status=$(
      aws dynamodb describe-table \
        --table-name "$table_name" \
        --output text \
```

```
        --query 'Table.TableStatus'
    )

   local error_code=${?}

    if [[ $error_code -ne 0 ]]; then
      aws_cli_error_log "$error_code"
      errecho "ERROR: AWS reports describe-table operation failed.$table_status"
      return 1
    fi

  echo "$table_status"

  return 0
}
```

この例で使用されているユーティリティ関数。

```
##############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
##############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

##############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
##############################################################################
```

```
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「DescribeTable」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Describe an Amazon DynamoDB table.
/*!
  \sa describeTable()
  \param tableName: The DynamoDB table name.
  \param clientConfiguration: AWS client configuration.
```

```
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::describeTable(const Aws::String &tableName,
                                     const Aws::Client::ClientConfiguration
 &clientConfiguration) {
   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

   Aws::DynamoDB::Model::DescribeTableRequest request;
   request.SetTableName(tableName);

   const Aws::DynamoDB::Model::DescribeTableOutcome &outcome =
 dynamoClient.DescribeTable(
           request);

   if (outcome.IsSuccess()) {
       const Aws::DynamoDB::Model::TableDescription &td =
 outcome.GetResult().GetTable();
       std::cout << "Table name  : " << td.GetTableName() << std::endl;
       std::cout << "Table ARN   : " << td.GetTableArn() << std::endl;
       std::cout << "Status      : "
                 <<
 Aws::DynamoDB::Model::TableStatusMapper::GetNameForTableStatus(
                         td.GetTableStatus()) << std::endl;
       std::cout << "Item count  : " << td.GetItemCount() << std::endl;
       std::cout << "Size (bytes): " << td.GetTableSizeBytes() << std::endl;

       const Aws::DynamoDB::Model::ProvisionedThroughputDescription &ptd =
 td.GetProvisionedThroughput();
       std::cout << "Throughput" << std::endl;
       std::cout << "  Read Capacity : " << ptd.GetReadCapacityUnits() <<
 std::endl;
       std::cout << "  Write Capacity: " << ptd.GetWriteCapacityUnits() <<
 std::endl;

       const Aws::Vector<Aws::DynamoDB::Model::AttributeDefinition> &ad =
 td.GetAttributeDefinitions();
       std::cout << "Attributes" << std::endl;
       for (const auto &a: ad)
           std::cout << "  " << a.GetAttributeName() << " (" <<

 Aws::DynamoDB::Model::ScalarAttributeTypeMapper::GetNameForScalarAttributeType(
                           a.GetAttributeType()) <<
                   ")" << std::endl;
   }
```

```
    else {
        std::cerr << "Failed to describe table: " <<
 outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「DescribeTable」を参照してください。

CLI

AWS CLI

テーブルを記述するには

次の describe-table の例は、MusicCollection テーブルを記述します。

```
aws dynamodb describe-table \
    --table-name MusicCollection
```

出力:

```
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 5
        },
```

```
        "TableSizeBytes": 0,
        "TableName": "MusicCollection",
        "TableStatus": "ACTIVE",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1421866952.062
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「表の説明」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DescribeTable」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// TableExists determines whether a DynamoDB table exists.
func (basics TableBasics) TableExists(ctx context.Context) (bool, error) {
 exists := true
 _, err := basics.DynamoDbClient.DescribeTable(
  ctx, &dynamodb.DescribeTableInput{TableName: aws.String(basics.TableName)},
 )
 if err != nil {
  var notFoundEx *types.ResourceNotFoundException
  if errors.As(err, &notFoundEx) {
   log.Printf("Table %v does not exist.\n", basics.TableName)
   err = nil
  } else {
   log.Printf("Couldn't determine existence of table %v. Here's why: %v\n",
 basics.TableName, err)
  }
  exists = false
 }
 return exists, err
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「DescribeTable」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import
 software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughputDescription;
import software.amazon.awssdk.services.dynamodb.model.TableDescription;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DescribeTable {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table to get information
 about (for example, Music3).
                """;

        if (args.length != 1) {
```

```
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        System.out.format("Getting description for %s\n\n", tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        describeDymamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void describeDymamoDBTable(DynamoDbClient ddb, String
tableName) {
        DescribeTableRequest request = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

        try {
            TableDescription tableInfo = ddb.describeTable(request).table();
            if (tableInfo != null) {
                System.out.format("Table name  : %s\n", tableInfo.tableName());
                System.out.format("Table ARN   : %s\n", tableInfo.tableArn());
                System.out.format("Status      : %s\n", tableInfo.tableStatus());
                System.out.format("Item count  : %d\n", tableInfo.itemCount());
                System.out.format("Size (bytes): %d\n",
tableInfo.tableSizeBytes());

                ProvisionedThroughputDescription throughputInfo =
tableInfo.provisionedThroughput();
                System.out.println("Throughput");
                System.out.format("  Read Capacity : %d\n",
throughputInfo.readCapacityUnits());
                System.out.format("  Write Capacity: %d\n",
throughputInfo.writeCapacityUnits());

                List<AttributeDefinition> attributes =
tableInfo.attributeDefinitions();
                System.out.println("Attributes");
                for (AttributeDefinition a : attributes) {
```

```
                    System.out.format("  %s (%s)\n", a.attributeName(),
 a.attributeType());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("\nDone!");
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「DescribeTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { DescribeTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DescribeTableCommand({
    TableName: "Pastries",
  });

  const response = await client.send(command);
  console.log(`TABLE NAME: ${response.Table.TableName}`);
  console.log(`TABLE ITEM COUNT: ${response.Table.ItemCount}`);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「DescribeTable」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to retrieve the selected table descriptions
ddb.describeTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Table.KeySchema);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「DescribeTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定テーブルの詳細を返します。

```
Get-DDBTable -TableName "myTable"
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「DescribeTable」を参照してください。

Tools for PowerShell V5

例 1: 指定テーブルの詳細を返します。

```
Get-DDBTable -TableName "myTable"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「DescribeTable」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
```

```
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def exists(self, table_name):
        """
        Determines whether a table exists. As a side effect, stores the table in
        a member variable.

        :param table_name: The name of the table to check.
        :return: True when the table exists; otherwise, False.
        """
        try:
            table = self.dyn_resource.Table(table_name)
            table.load()
            exists = True
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                exists = False
            else:
                logger.error(
                    "Couldn't check for existence of %s. Here's why: %s: %s",
                    table_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
```

```
                raise
        else:
            self.table = table
        return exists
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「DescribeTable」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Determines whether a table exists. As a side effect, stores the table in
  # a member variable.
  #
  # @param table_name [String] The name of the table to check.
  # @return [Boolean] True when the table exists; otherwise, False.
  def exists?(table_name)
    @dynamo_resource.client.describe_table(table_name: table_name)
    @logger.debug("Table #{table_name} exists")
```

```
rescue Aws::DynamoDB::Errors::ResourceNotFoundException
  @logger.debug("Table #{table_name} doesn't exist")
  false
rescue Aws::DynamoDB::Errors::ServiceError => e
  puts("Couldn't check for existence of #{table_name}:\n")
  puts("\t#{e.code}: #{e.message}")
  raise
end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「DescribeTable」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
TRY.
    oo_result = lo_dyn->describetable( iv_tablename = iv_table_name ).
    DATA(lv_tablename) = oo_result->get_table( )->ask_tablename( ).
    DATA(lv_tablearn) = oo_result->get_table( )->ask_tablearn( ).
    DATA(lv_tablestatus) = oo_result->get_table( )->ask_tablestatus( ).
    DATA(lv_itemcount) = oo_result->get_table( )->ask_itemcount( ).
    MESSAGE 'The table name is ' && lv_tablename
        && '. The table ARN is ' && lv_tablearn
        && '. The tablestatus is ' && lv_tablestatus
        && '. Item count is ' && lv_itemcount TYPE 'I'.
  CATCH /aws1/cx_dynresourcenotfoundex.
    MESSAGE 'The table ' && lv_tablename && ' does not exist' TYPE 'E'.
ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DescribeTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DescribeTimeToLive** を使用する

次のサンプルコードは、DescribeTimeToLive を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- Streams と Time-to-Live を操作する

CLI

AWS CLI

テーブルの Time to Live 設定を表示するには

次の describe-time-to-live の例は、MusicCollection テーブルの Time to Live 設定を表示します。

```
aws dynamodb describe-time-to-live \
    --table-name MusicCollection
```

出力:

```
{
    "TimeToLiveDescription": {
        "TimeToLiveStatus": "ENABLED",
        "AttributeName": "ttl"
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「Time to Live (TTL)」を参照してください。

- API の詳細については、「AWS CLI Command Reference」の「DescribeTimeToLive」を参照してください。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して既存の DynamoDB テーブルの TTL 設定を記述します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DescribeTimeToLiveRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTimeToLiveResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.logging.Level;
import java.util.logging.Logger;

    public DescribeTimeToLiveResponse describeTTL(final String tableName, final
 Region region) {
        final DescribeTimeToLiveRequest request =
            DescribeTimeToLiveRequest.builder().tableName(tableName).build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.describeTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「DescribeTimeToLive」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して既存の DynamoDB テーブルの TTL 設定を記述します。

```
import { DynamoDBClient, DescribeTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const describeTTL = async (tableName, region) => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    try {
        const ttlDescription = await client.send(new
 DescribeTimeToLiveCommand({ TableName: tableName }));

        if (ttlDescription.TimeToLiveDescription.TimeToLiveStatus === 'ENABLED')
 {
            console.log("TTL is enabled for table %s.", tableName);
        } else {
            console.log("TTL is not enabled for table %s.", tableName);
        }

        return ttlDescription;
    } catch (e) {
        console.error(`Error describing table: ${e}`);
        throw e;
    }
}

// Example usage (commented out for testing)
// describeTTL('your-table-name', 'us-east-1');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「DescribeTimeToLive」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して既存の DynamoDB テーブルの TTL 設定を記述します。

```
import boto3
```

```
def describe_ttl(table_name, region):
    """
    Describes TTL on an existing table, as well as a region.

    :param table_name: String representing the name of the table
    :param region: AWS Region of the table - example `us-east-1`
    :return: Time to live description.
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        ttl_description = dynamodb.describe_time_to_live(TableName=table_name)
        print(
            f"TimeToLive for table {table_name} is status
 {ttl_description['TimeToLiveDescription']['TimeToLiveStatus']}"
        )

        return ttl_description
    except Exception as e:
        print(f"Error describing table: {e}")
        raise


# Enter your own table name and AWS region
describe_ttl("your-table-name", "us-east-1")
```

- API の詳細については、「AWS SDK for Python (Boto3) API Reference」の「DescribeTimeToLive」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で **ExecuteStatement** を使用する

次のサンプルコードは、ExecuteStatement を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- PartiQL DELETE を使用してデータを削除する

- PartiQL INSERT を使用してデータを挿入する
- PartiQL を使用してテーブルに対してクエリを実行する
- PartiQL SELECT を使用してデータをクエリする
- PartiQL UPDATE を使用してデータを更新する

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

INSERT ステートメントを使用して項目を追加します。

```
        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
```

```
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

SELECT ステートメントを使用して項目を取得します。

```
        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }
```

SELECT ステートメントを使用して項目の一覧を取得します。

```
        /// <summary>
```

```
        /// Retrieve multiple movies by year using a SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="year">The year the movies were released.</param>
        /// <returns></returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { N = year.ToString() },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }
```

UPDATE ステートメントを使用して項目を更新します。

```
        /// <summary>
        /// Updates a single movie in the table, adding information for the
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";
```

```
            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

DELETE ステートメントを使用して映画を 1 つ削除します。

```
        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
```

```
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ExecuteStatement」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

INSERT ステートメントを使用して項目を追加します。

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
    Aws::String title;
    float rating;
    int year;
    Aws::String plot;
    {
        title = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        year = askQuestionForInt("What year was it released? ");
        rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                          1, 10);
        plot = askQuestion("Summarize the plot for me: ");

        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
```

```
                << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                << INFO_KEY << "': ?}";

        request.SetStatement(sqlStream.str());

        // Create the parameter attributes.
        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

        Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        ratingAttribute->SetN(rating);
        infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

        std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                ALLOCATION_TAG.c_str());
        plotAttribute->SetS(plot);
        infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
        attributes.push_back(infoMapAttribute);
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add a movie: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }
```

SELECT ステートメントを使用して項目を取得します。

```
    //  3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
```

```
   {
       Aws::DynamoDB::Model::ExecuteStatementRequest request;
       std::stringstream sqlStream;
       sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                 << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

       request.SetStatement(sqlStream.str());

       Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
       request.SetParameters(attributes);

       Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
               request);

       if (!outcome.IsSuccess()) {
           std::cerr << "Failed to retrieve movie information: "
                     << outcome.GetError().GetMessage() << std::endl;
           return false;
       }
       else {
           // Print the retrieved movie information.
           const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

           const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

           if (items.size() == 1) {
               printMovieInfo(items[0]);
           }
           else {
               std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                         << " There should be only one movie." << std::endl;
           }
       }
   }
```

UPDATE ステートメントを使用して項目を更新します。

```
   //  4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
   {
       rating = askQuestionForFloatRange(
               Aws::String("\nLet's update your movie.\nYou rated it  ") +
               std::to_string(rating)
               + ", what new rating would you give it? ", 1, 10);

       Aws::DynamoDB::Model::ExecuteStatementRequest request;
       std::stringstream sqlStream;
       sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                 << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                 << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

       request.SetStatement(sqlStream.str());

       Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

       request.SetParameters(attributes);

       Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
               request);

       if (!outcome.IsSuccess()) {
           std::cerr << "Failed to update a movie: "
                     << outcome.GetError().GetMessage();
           return false;
       }
   }
```

DELETE ステートメントを使用して項目を削除します。

```
   // 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
   {
       Aws::DynamoDB::Model::ExecuteStatementRequest request;
       std::stringstream sqlStream;
       sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
```

```
                << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movie: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「ExecuteStatement」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例の関数レシーバー構造体を定義します。

```
import (
 "context"
 "fmt"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

INSERT ステートメントを使用して項目を追加します。

```
// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
 movie.Info})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
 }
 return err
}
```

SELECT ステートメントを使用して項目を取得します。

```
// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
 table by
// title and year.
func (runner PartiQLRunner) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 var movie Movie
 params, err := attributevalue.MarshalList([]interface{}{title, year})
 if err != nil {
  panic(err)
 }
 response, err := runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Items[0], &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}
```

SELECT ステートメントを使用して、項目のリストを取得し、結果を射影します。

```
// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
 DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
 results.
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(ctx context.Context, pageSize int32)
 ([]map[string]interface{}, error) {
 var output []map[string]interface{}
 var response *dynamodb.ExecuteStatementOutput
```

```
 var err error
 var nextToken *string
 for moreData := true; moreData; {
  response, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
   Statement: aws.String(
    fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
   Limit:     aws.Int32(pageSize),
   NextToken: nextToken,
  })
  if err != nil {
   log.Printf("Couldn't get movies. Here's why: %v\n", err)
   moreData = false
  } else {
   var pageOutput []map[string]interface{}
   err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    log.Printf("Got a page of length %v.\n", len(response.Items))
    output = append(output, pageOutput...)
   }
   nextToken = response.NextToken
   moreData = nextToken != nil
  }
 }
 return output, err
}
```

UPDATE ステートメントを使用して項目を更新します。

```
// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
 that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(ctx context.Context, movie Movie, rating
 float64) error {
 params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
```

```
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
 }
 return err
}
```

DELETE ステートメントを使用して項目を削除します。

```
// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
 DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
```

```
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「ExecuteStatement」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL を使用して項目を作成します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: `INSERT INTO Flowers value {'Name':?}`,
    Parameters: ["Rose"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目を取得します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "SELECT * FROM CloudTypes WHERE IsStorm=?",
    Parameters: [false],
    ConsistentRead: true,
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目を更新します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "UPDATE EyeColors SET IsRecessive=? where Color=?",
    Parameters: [true, "blue"],
```

```
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

PartiQL を使用して項目を削除します。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "DELETE FROM PaintColors where Name=?",
    Parameters: ["Purple"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスのExecuteStatement」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    public function insertItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => "$statement",
            'Parameters' => $parameters,
        ]);
    }

    public function getItemByPartiQL(string $tableName, array $key): Result
    {
        list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);

        return $this->dynamoDbClient->executeStatement([
            'Parameters' => $parameters,
            'Statement' => $statement,
        ]);
    }

    public function updateItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }

    public function deleteItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
```

```
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「ExecuteStatement」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class PartiQLWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statement, params):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statement: The PartiQL statement.
        :param params: The list of PartiQL parameters. These are applied to the
```

```
                        statement in the order they are listed.
        :return: The items returned from the statement, if any.
        """
        try:
            output = self.dyn_resource.meta.client.execute_statement(
                Statement=statement, Parameters=params
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute PartiQL '%s' because the table does not
exist.",
                    statement,
                )
            else:
                logger.error(
                    "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                    statement,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「ExecuteStatement」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL を使用して項目を 1 つ選択します。

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Gets a single record from a table using PartiQL.
  # Note: To perform more fine-grained selects,
  # use the Client.query instance method instead.
  #
  # @param title [String] The title of the movie to search.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def select_item_by_title(title)
    request = {
      statement: "SELECT * FROM \"#{@table.name}\" WHERE title=?",
      parameters: [title]
    }
    @dynamodb.client.execute_statement(request)
  end
```

PartiQL を使用して項目を更新します。

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Updates a single record from a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @param rating [Float] The new rating to assign the title.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def update_rating_by_title(title, year, rating)
    request = {
```

```
      statement: "UPDATE \"#{@table.name}\" SET info.rating=? WHERE title=? and
 year=?",
      parameters: [{ "N": rating }, title, year]
    }
    @dynamodb.client.execute_statement(request)
  end
```

PartiQL を使用して項目を 1 つ追加します。

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Adds a single record to a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @param plot [String] The plot of the movie.
  # @param rating [Float] The new rating to assign the title.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def insert_item(title, year, plot, rating)
    request = {
      statement: "INSERT INTO \"#{@table.name}\" VALUE {'title': ?, 'year': ?,
 'info': ?}",
      parameters: [title, year, { 'plot': plot, 'rating': rating }]
    }
    @dynamodb.client.execute_statement(request)
  end
```

PartiQL を使用して項目を 1 つ削除します。

```
class DynamoDBPartiQLSingle
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
```

```
    @dynamodb = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamodb.table(table_name)
  end

  # Deletes a single record from a table using PartiQL.
  #
  # @param title [String] The title of the movie to update.
  # @param year [Integer] The year the movie was released.
  # @return [Aws::DynamoDB::Types::ExecuteStatementOutput]
  def delete_item_by_title(title, year)
    request = {
      statement: "DELETE FROM \"#{@table.name}\" WHERE title=? and year=?",
      parameters: [title, year]
    }
    @dynamodb.client.execute_statement(request)
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「ExecuteStatement」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetItem** を使用する

次のサンプルコードは、GetItem を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Gets information about an existing movie from the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information about
    /// the movie to retrieve.</param>
    /// <param name="tableName">The name of the table containing the movie.</
param>
    /// <returns>A Dictionary object containing information about the item
    /// retrieved.</returns>
    public async Task<Dictionary<string, AttributeValue>> GetItemAsync(Movie
 newMovie, string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new GetItemRequest
            {
                Key = key,
                TableName = tableName,
            };

            var response = await _amazonDynamoDB.GetItemAsync(request);
            return response.Item;
        }
        catch (ResourceNotFoundException ex)
        {
```

```
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return new Dictionary<string, AttributeValue>();
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while getting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting item.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「GetItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_get_item
#
# This function gets an item from a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to get.
```

```
#       [-q query]  -- Optional JMESPath query expression.
#
#  Returns:
#       The item as text output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_get_item() {
  local table_name keys query response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_get_item"
    echo "Get an item from a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -k keys  -- Path to json file containing the keys that identify the
 item to get."
    echo " [-q query]  -- Optional JMESPath query expression."
    echo ""
  }
  query=""
  while getopts "n:k:q:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      k) keys="${OPTARG}" ;;
      q) query="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
```

```
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$keys" ]]; then
    errecho "ERROR: You must provide a keys json file path the -k parameter."
    usage
    return 1
  fi

  if [[ -n "$query" ]]; then
    response=$(aws dynamodb get-item \
      --table-name "$table_name" \
      --key file://"$keys" \
      --output text \
      --query "$query")
  else
    response=$(
      aws dynamodb get-item \
        --table-name "$table_name" \
        --key file://"$keys" \
        --output text
    )
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports get-item operation failed.$response"
    return 1
  fi

  if [[ -n "$query" ]]; then
    echo "$response" | sed "/^\t/s/\t//1" # Remove initial tab that the JMSEPath
 query inserts on some strings.
  else
    echo "$response"
  fi

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
```

```
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Get an item from an Amazon DynamoDB table.
/*!
  \sa getItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::getItem(const Aws::String &tableName,
                               const Aws::String &partitionKey,
                               const Aws::String &partitionValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::GetItemRequest request;

    // Set up the request.
    request.SetTableName(tableName);
```

```
    request.AddKey(partitionKey,
                   Aws::DynamoDB::Model::AttributeValue().SetS(partitionValue));

    // Retrieve the item's fields and values.
    const Aws::DynamoDB::Model::GetItemOutcome &outcome =
 dynamoClient.GetItem(request);
    if (outcome.IsSuccess()) {
        // Reference the retrieved fields/values.
        const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> &item =
 outcome.GetResult().GetItem();
        if (!item.empty()) {
            // Output each retrieved field and its value.
            for (const auto &i: item)
                std::cout << "Values: " << i.first << ": " << i.second.GetS()
                          << std::endl;
        }
        else {
            std::cout << "No item found with the key " << partitionKey <<
 std::endl;
        }
    }
    else {
        std::cerr << "Failed to get item: " << outcome.GetError().GetMessage();
    }

    return outcome.IsSuccess();
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「GetItem」を参照してください。

CLI

AWS CLI

例 1: テーブル内の項目を読み込むには

次の get-item の例は、MusicCollection テーブルから項目を取得します。テーブルにはハッシュおよび範囲プライマリキー (ArtistおよびSongTitle) があるため、これらの属性の両方を指定する必要があります。このコマンドは、オペレーションによって消費される読み込み容量に関する情報も要求します。

```
aws dynamodb get-item \
    --table-name MusicCollection \
    --key file://key.json \
    --return-consumed-capacity TOTAL
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

出力:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

例 2: 整合性のある読み込みを使用して項目を読み込むには

次の例では、強力な整合性のある読み込みを使用して MusicCollection テーブルから項目を読み込みます。

```
aws dynamodb get-item \
    --table-name MusicCollection \
```

```
    --key file://key.json \
    --consistent-read \
    --return-consumed-capacity TOTAL
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

出力:

```
{
    "Item": {
        "AlbumTitle": {
            "S": "Songs About Life"
        },
        "SongTitle": {
            "S": "Happy Day"
        },
        "Artist": {
            "S": "Acme Band"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

例 3: 項目の特定の属性を取得するには

次の例は、射影式を使用して目的のアイテムの 3 つの属性のみを取得します。

```
aws dynamodb get-item \
    --table-name ProductCatalog \
    --key '{"Id": {"N": "102"}}' \
    --projection-expression "#T, #C, #P" \
```

```
    --expression-attribute-names file://names.json
```

names.json の内容:

```
{
    "#T": "Title",
    "#C": "ProductCategory",
    "#P": "Price"
}
```

出力:

```
{
    "Item": {
        "Price": {
            "N": "20"
        },
        "Title": {
            "S": "Book 102 Title"
        },
        "ProductCategory": {
            "S": "Book"
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目の読み込み」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// GetMovie gets movie data from the DynamoDB table by using the primary
 composite key
// made of title and year.
func (basics TableBasics) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 movie := Movie{Title: title, Year: year}
 response, err := basics.DynamoDbClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key: movie.GetKey(), TableName: aws.String(basics.TableName),
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
 } else {
  err = attributevalue.UnmarshalMap(response.Item, &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
```

```
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
```

```
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「GetItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用して、テーブルから項目を取得します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To get an item from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, see the EnhancedGetItem example.
 */
public class GetItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal>

                Where:
                    tableName - The Amazon DynamoDB table from which an item is
 retrieved (for example, Music3).\s
                    key - The key used in the Amazon DynamoDB table (for example,
 Artist).\s
                    keyval - The key value that represents the item to get (for
 example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        System.out.format("Retrieving item \"%s\" from \"%s\"\n", keyVal,
 tableName);
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        getDynamoDBItem(ddb, tableName, key, keyVal);
        ddb.close();
    }

    public static void getDynamoDBItem(DynamoDbClient ddb, String tableName,
 String key, String keyVal) {
```

```
        HashMap<String, AttributeValue> keyToGet = new HashMap<>();
        keyToGet.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        GetItemRequest request = GetItemRequest.builder()
                .key(keyToGet)
                .tableName(tableName)
                .build();

        try {
            // If there is no matching item, GetItem does not return any data.
            Map<String, AttributeValue> returnedItem =
 ddb.getItem(request).item();
            if (returnedItem.isEmpty())
                System.out.format("No item found with the key %s!\n", key);
            else {
                Set<String> keys = returnedItem.keySet();
                System.out.println("Amazon DynamoDB table attributes: \n");
                for (String key1 : keys) {
                    System.out.format("%s: %s\n", key1,
 returnedItem.get(key1).toString());
                }
            }

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「GetItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「GetCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「GetItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルから項目を取得します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

DynamoDB ドキュメントクライアントを使用して、テーブルから項目を取得します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「GetItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun getSpecificItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
) {
    val keyToGet = mutableMapOf<String, AttributeValue>()
    keyToGet[keyName] = AttributeValue.S(keyVal)

    val request =
```

```
        GetItemRequest {
            key = keyToGet
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val returnedItem = ddb.getItem(request)
        val numbersMap = returnedItem.item
        numbersMap?.forEach { key1 ->
            println(key1.key)
            println(key1.value)
        }
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「GetItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        $movie = $service->getItemByKey($tableName, $key);
        echo "\nThe movie {$movie['Item']['title']['S']} was released in
{$movie['Item']['year']['N']}.\n";

    public function getItemByKey(string $tableName, array $key)
    {
        return $this->dynamoDbClient->getItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「GetItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目を返します。

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
SongTitle                      Somewhere Down The Road
Price                          1.94
Artist                         No One You Know
CriticRating                   9
AlbumTitle                     Somewhat Famous
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「GetItem」を参照してください。

Tools for PowerShell V5

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目を返します。

```
$key = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
} | ConvertTo-DDBItem

Get-DDBItem -TableName 'Music' -Key $key | ConvertFrom-DDBItem
```

出力する。

```
Name                            Value
----                            -----
Genre                           Country
SongTitle                       Somewhere Down The Road
Price                           1.94
Artist                          No One You Know
CriticRating                    9
AlbumTitle                      Somewhat Famous
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「GetItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
```

```
                "John C. Reilly"
            ]
        }
    }
"""

def __init__(self, dyn_resource):
    """
    :param dyn_resource: A Boto3 DynamoDB resource.
    """
    self.dyn_resource = dyn_resource
    # The table variable is set during the scenario in the call to
    # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
    self.table = None


def get_movie(self, title, year):
    """
    Gets movie data from the table for a specific movie.

    :param title: The title of the movie.
    :param year: The release year of the movie.
    :return: The data about the requested movie.
    """
    try:
        response = self.table.get_item(Key={"year": year, "title": title})
    except ClientError as err:
        logger.error(
            "Couldn't get movie %s from table %s. Here's why: %s: %s",
            title,
            self.table.name,
            err.response["Error"]["Code"],
            err.response["Error"]["Message"],
        )
        raise
    else:
        return response["Item"]
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「GetItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Gets movie data from the table for a specific movie.
  #
  # @param title [String] The title of the movie.
  # @param year [Integer] The release year of the movie.
  # @return [Hash] The data about the requested movie.
  def get_item(title, year)
    @table.get_item(key: { 'year' => year, 'title' => title })
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't get movie #{title} (#{year}) from table #{@table.name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「GetItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_item = lo_dyn->getitem(
          iv_tablename                = iv_table_name
          it_key                      = it_key ).
        DATA(lt_attr) = oo_item->get_item( ).
        DATA(lo_title) = lt_attr[ key = 'title' ]-value.
        DATA(lo_year) = lt_attr[ key = 'year' ]-value.
        DATA(lo_rating) = lt_attr[ key = 'rating' ]-value.
        MESSAGE 'Movie name is: ' && lo_title->get_s( )
          && 'Movie year is: ' && lo_year->get_n( )
          && 'Moving rating is: ' && lo_rating->get_n( ) TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「GetItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB
```

```
    /// Return a `Movie` record describing the specified movie from the Amazon
    /// DynamoDB table.
    ///
    /// - Parameters:
    ///   - title: The movie's title (`String`).
    ///   - year: The movie's release year (`Int`).
    ///
    /// - Throws: `MoviesError.ItemNotFound` if the movie isn't in the table.
    ///
    /// - Returns: A `Movie` record with the movie's details.
    func get(title: String, year: Int) async throws -> Movie {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = GetItemInput(
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                tableName: self.tableName
            )
            let output = try await client.getItem(input: input)
            guard let item = output.item else {
                throw MoviesError.ItemNotFound
            }

            let movie = try Movie(withItem: item)
            return movie
        } catch {
            print("ERROR: get:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「GetItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **ListTables** を使用する

次のサンプルコードは、ListTables を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 属性ベースのアクセス制御を設定する

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
private static async Task ListMyTables()
{
    Console.WriteLine("\n*** Listing tables ***");

    string lastTableNameEvaluated = null;
    do
    {
        var response = await Client.ListTablesAsync(new ListTablesRequest
        {
            Limit = 2,
            ExclusiveStartTableName = lastTableNameEvaluated
        });

        foreach (var name in response.TableNames)
        {
            Console.WriteLine(name);
        }
```

```
            lastTableNameEvaluated = response.LastEvaluatedTableName;
        } while (lastTableNameEvaluated != null);
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListTables」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_list_tables
#
# This function lists all the tables in a DynamoDB.
#
# Returns:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_list_tables() {
  response=$(aws dynamodb list-tables \
    --output text \
    --query "TableNames")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports batch-write-item operation failed.$response"
    return 1
  fi

  echo "$response" | tr -s "[:space:]" "\n"
```

```
  return 0
}
```

この例で使用されているユーティリティ関数。

```
################################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
################################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

################################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
```

```
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListTables」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! List the Amazon DynamoDB tables for the current AWS account.
/*!
  \sa listTables()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::listTables(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::ListTablesRequest listTablesRequest;
    listTablesRequest.SetLimit(50);
    do {
        const Aws::DynamoDB::Model::ListTablesOutcome &outcome =
 dynamoClient.ListTables(
                listTablesRequest);
```

```
        if (!outcome.IsSuccess()) {
            std::cout << "Error: " << outcome.GetError().GetMessage() <<
 std::endl;
            return false;
        }

        for (const auto &tableName: outcome.GetResult().GetTableNames())
            std::cout << tableName << std::endl;
        listTablesRequest.SetExclusiveStartTableName(
                outcome.GetResult().GetLastEvaluatedTableName());

    } while (!listTablesRequest.GetExclusiveStartTableName().empty());

    return true;
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「ListTables」を参照してください。

CLI

AWS CLI

例 1: テーブルを一覧表示するには

次の `list-tables` の例では、現在の AWS アカウントとリージョンに関連するすべてのテーブルを一覧表示します。

```
aws dynamodb list-tables
```

出力:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog",
        "Reply",
        "Thread"
    ]
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブル名のリスト化」を参照してください。

例 2: ページサイズを制限するには

次の例は、既存のすべてのテーブルのリストを返しますが、1 回の呼び出しで 1 つの項目のみを取得し、必要な場合は複数の呼び出しを実行してリスト全体を取得します。デフォルトのページサイズ (1000) を使用して大量のリソースに対してリストコマンドを実行する際に「タイムアウト」が発生する場合、ページサイズを制限してください。

```
aws dynamodb list-tables \
    --page-size 1
```

出力:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog",
        "Reply",
        "Thread"
    ]
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブル名のリスト化」を参照してください。

例 3: 返される項目の数を制限するには

次の例は、返される項目の数を 2 に制限します。レスポンスには、次の結果ページの取得に使用する NextToken 値が含まれます。

```
aws dynamodb list-tables \
    --max-items 2
```

出力:

```
{
    "TableNames": [
        "Forum",
        "ProductCatalog"
    ],
```

```
    "NextToken":
 "abCDeFGhiJKlmnOPqrSTuvwxYZ1aBCdEFghijK7LM51nOpqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9"
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブル名のリスト化」を参照してください。

例 4: 次の結果ページを取得するには

次のコマンドは、`list-tables` コマンドへの前の呼び出しの `NextToken` 値を使用して、次の結果ページを取得します。この場合のレスポンスには `NextToken` 値が含まれていないため、結果の最後のページに達したことがわかります。

```
aws dynamodb list-tables \
    --starting-
token abCDeFGhiJKlmnOPqrSTuvwxYZ1aBCdEFghijK7LM51nOpqRSTuv3WxY3ZabC5dEFGhI2Jk3LmnoPQ6RST9
```

出力:

```
{
    "TableNames": [
        "Reply",
        "Thread"
    ]
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「テーブル名のリスト化」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListTables」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// ListTables lists the DynamoDB table names for the current account.
func (basics TableBasics) ListTables(ctx context.Context) ([]string, error) {
 var tableNames []string
 var output *dynamodb.ListTablesOutput
 var err error
 tablePaginator := dynamodb.NewListTablesPaginator(basics.DynamoDbClient,
 &dynamodb.ListTablesInput{})
 for tablePaginator.HasMorePages() {
  output, err = tablePaginator.NextPage(ctx)
  if err != nil {
   log.Printf("Couldn't list tables. Here's why: %v\n", err)
   break
  } else {
   tableNames = append(tableNames, output.TableNames...)
  }
 }
 return tableNames, err
}
```

- API の詳細については、『AWS SDK for Go API リファレンス』の「ListTables」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ListTablesRequest;
import software.amazon.awssdk.services.dynamodb.model.ListTablesResponse;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListTables {
    public static void main(String[] args) {
        System.out.println("Listing your Amazon DynamoDB tables:\n");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        listAllTables(ddb);
        ddb.close();
```

```
    }

    public static void listAllTables(DynamoDbClient ddb) {
        boolean moreTables = true;
        String lastName = null;

        while (moreTables) {
            try {
                ListTablesResponse response = null;
                if (lastName == null) {
                    ListTablesRequest request =
 ListTablesRequest.builder().build();
                    response = ddb.listTables(request);
                } else {
                    ListTablesRequest request = ListTablesRequest.builder()
                            .exclusiveStartTableName(lastName).build();
                    response = ddb.listTables(request);
                }

                List<String> tableNames = response.tableNames();
                if (tableNames.size() > 0) {
                    for (String curName : tableNames) {
                        System.out.format("* %s\n", curName);
                    }
                } else {
                    System.out.println("No tables found!");
                    System.exit(0);
                }

                lastName = response.lastEvaluatedTableName();
                if (lastName == null) {
                    moreTables = false;
                }

            } catch (DynamoDbException e) {
                System.err.println(e.getMessage());
                System.exit(1);
            }
        }
        System.out.println("\nDone!");
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「ListTables」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「ListTables」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
```

```
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

// Call DynamoDB to retrieve the list of tables
ddb.listTables({ Limit: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err.code);
  } else {
    console.log("Table names are ", data.TableNames);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「ListTables」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun listAllTables() {
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.listTables(ListTablesRequest {})
        response.tableNames?.forEach { tableName ->
            println("Table name is $tableName")
        }
    }
}
```

- API の詳細については、『AWS SDK for Kotlin API リファレンス』の「ListTables」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
public function listTables($exclusiveStartTableName = "", $limit = 100)
{
    $this->dynamoDbClient->listTables([
        'ExclusiveStartTableName' => $exclusiveStartTableName,
        'Limit' => $limit,
    ]);
}
```

- API の詳細については、AWS SDK for PHP API リファレンスの「ListTables」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: すべてのテーブルの詳細を返し、サービスが他にテーブルがないことを知らせるまで自動で繰り返します。

```
Get-DDBTableList
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「ListTables」を参照してください。

Tools for PowerShell V5

例 1: すべてのテーブルの詳細を返し、サービスが他にテーブルがないことを知らせるまで自動で繰り返します。

```
Get-DDBTableList
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「ListTables」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
```

```
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def list_tables(self):
        """
        Lists the Amazon DynamoDB tables for the current account.

        :return: The list of tables.
        """
        try:
            tables = []
            for table in self.dyn_resource.tables.all():
                print(table.name)
                tables.append(table)
        except ClientError as err:
            logger.error(
                "Couldn't list tables. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return tables
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「ListTables」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルが存在するかどうかを確認します。

```
# Encapsulates an Amazon DynamoDB table of movie data.
class Scaffold
  attr_reader :dynamo_resource, :table_name, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table_name = table_name
    @table = nil
    @logger = Logger.new($stdout)
    @logger.level = Logger::DEBUG
  end

  # Determines whether a table exists. As a side effect, stores the table in
  # a member variable.
  #
  # @param table_name [String] The name of the table to check.
  # @return [Boolean] True when the table exists; otherwise, False.
  def exists?(table_name)
    @dynamo_resource.client.describe_table(table_name: table_name)
    @logger.debug("Table #{table_name} exists")
  rescue Aws::DynamoDB::Errors::ResourceNotFoundException
    @logger.debug("Table #{table_name} doesn't exist")
    false
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't check for existence of #{table_name}:\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「ListTables」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn list_tables(client: &Client) -> Result<Vec<String>, Error> {
    let paginator = client.list_tables().into_paginator().items().send();
    let table_names = paginator.collect::<Result<Vec<_>, _>>().await?;

    println!("Tables:");

    for name in &table_names {
        println!("  {}", name);
    }

    println!("Found {} tables", table_names.len());
    Ok(table_names)
}
```

テーブルが存在するかどうかを確認します。

```
pub async fn table_exists(client: &Client, table: &str) -> Result<bool, Error> {
    debug!("Checking for table: {table}");
    let table_list = client.list_tables().send().await;

    match table_list {
        Ok(list) => Ok(list.table_names().contains(&table.into())),
        Err(e) => Err(e.into()),
    }
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「ListTables」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_result = lo_dyn->listtables( ).
        " You can loop over the oo_result to get table properties like this.
        LOOP AT oo_result->get_tablenames( ) INTO DATA(lo_table_name).
          DATA(lv_tablename) = lo_table_name->get_value( ).
        ENDLOOP.
        DATA(lv_tablecount) = lines( oo_result->get_tablenames( ) ).
        MESSAGE 'Found ' && lv_tablecount && ' tables' TYPE 'I'.
      CATCH /aws1/cx_rt_service_generic INTO DATA(lo_exception).
        DATA(lv_error) = |"{ lo_exception->av_err_code }" - { lo_exception-
>av_err_msg }|.
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- API の詳細については、AWS SDK for SAP ABAP API リファレンスの「ListTables」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Get a list of the DynamoDB tables available in the specified Region.
    ///
    /// - Returns: An array of strings listing all of the tables available
    ///   in the Region specified when the session was created.
    public func getTableList() async throws -> [String] {
        let input = ListTablesInput(
        )
        return try await session.listTables(input: input)
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「ListTables」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **PutItem** を使用する

次のサンプルコードは、PutItem を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- グローバルセカンダリインデックスの高度なシナリオ
- TTL を含む項目を作成する
- MREC を示すグローバルテーブルの作成と管理
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- 条件付きオペレーションを使用する
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する

- Streams と Time-to-Live を操作する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Adds a new item to the table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing informtation for
    /// the movie to add to the table.</param>
    /// <param name="tableName">The name of the table where the item will be
 added.</param>
    /// <returns>A Boolean value that indicates the results of adding the item.</
returns>
    public async Task<bool> PutItemAsync(Movie newMovie, string tableName)
    {
        try
        {
            var item = new Dictionary<string, AttributeValue>
            {
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };

            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = item,
            };

            await _amazonDynamoDB.PutItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
```

```
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while putting
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while putting item.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「PutItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_put_item
#
# This function puts an item into a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -i item  -- Path to json file containing the item values.
```

```
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_put_item() {
  local table_name item response
  local option OPTARG # Required to use getopts command in a function.

  ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_put_item"
    echo "Put an item into a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -i item  -- Path to json file containing the item values."
    echo ""
  }

  while getopts "n:i:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      i) item="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$item" ]]; then
```

```
    errecho "ERROR: You must provide an item with the -i parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    item:   $item"
  iecho ""
  iecho ""

  response=$(aws dynamodb put-item \
    --table-name "$table_name" \
    --item file://"$item")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports put-item operation failed.$response"
    return 1
  fi

  return 0

}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
###############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

###############################################################################
```

```
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Put an item in an Amazon DynamoDB table.
/*!
  \sa putItem()
  \param tableName: The table name.
  \param artistKey: The artist key. This is the partition key for the table.
  \param artistValue: The artist value.
  \param albumTitleKey: The album title key.
  \param albumTitleValue: The album title value.
  \param awardsKey: The awards key.
  \param awardsValue: The awards value.
  \param songTitleKey: The song title key.
  \param songTitleValue: The song title value.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::putItem(const Aws::String &tableName,
                               const Aws::String &artistKey,
                               const Aws::String &artistValue,
                               const Aws::String &albumTitleKey,
                               const Aws::String &albumTitleValue,
                               const Aws::String &awardsKey,
                               const Aws::String &awardsValue,
                               const Aws::String &songTitleKey,
                               const Aws::String &songTitleValue,
                               const Aws::Client::ClientConfiguration
 &clientConfiguration) {
```

```
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::PutItemRequest putItemRequest;
    putItemRequest.SetTableName(tableName);

    putItemRequest.AddItem(artistKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            artistValue)); // This is the hash key.
    putItemRequest.AddItem(albumTitleKey,
 Aws::DynamoDB::Model::AttributeValue().SetS(
            albumTitleValue));
    putItemRequest.AddItem(awardsKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(awardsValue));
    putItemRequest.AddItem(songTitleKey,

 Aws::DynamoDB::Model::AttributeValue().SetS(songTitleValue));

    const Aws::DynamoDB::Model::PutItemOutcome outcome = dynamoClient.PutItem(
            putItemRequest);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully added Item!" << std::endl;
    }
    else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
```

```
                                        const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスのの「PutItem」を参照してください。

CLI

AWS CLI

例 1: テーブルに項目を追加するには

次の put-item の例は、MusicCollection テーブルに新しい項目を追加します。

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

item.json の内容:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Greatest Hits"}
}
```

出力:

```
{
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "No One You Know"
            }
        },
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

例 2: テーブル内の項目を条件付きで上書きするには

次の put-item の例は、MusicCollection テーブル内の既存の項目に Greatest Hits の値を持つ AlbumTitle 属性がある場合にのみ、その項目を上書きします。このコマンドは、その項目の以前の値を返します。

```
aws dynamodb put-item \
    --table-name MusicCollection \
    --item file://item.json \
    --condition-expression "#A = :A" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-values ALL_OLD
```

item.json の内容:

```
{
    "Artist": {"S": "No One You Know"},
    "SongTitle": {"S": "Call Me Today"},
    "AlbumTitle": {"S": "Somewhat Famous"}
}
```

names.json の内容:

```
{
    "#A": "AlbumTitle"
}
```

values.json の内容:

```
{
    ":A": {"S": "Greatest Hits"}
}
```

出力:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Greatest Hits"
        },
        "Artist": {
```

```
            "S": "No One You Know"
        },
        "SongTitle": {
            "S": "Call Me Today"
        }
    }
}
```

キーが存在する場合は、次のような出力が表示されます。

```
A client error (ConditionalCheckFailedException) occurred when calling the
 PutItem operation: The conditional request failed.
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
```

```
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovie adds a movie the DynamoDB table.
func (basics TableBasics) AddMovie(ctx context.Context, movie Movie) error {
 item, err := attributevalue.MarshalMap(movie)
 if err != nil {
  panic(err)
 }
 _, err = basics.DynamoDbClient.PutItem(ctx, &dynamodb.PutItemInput{
  TableName: aws.String(basics.TableName), Item: item,
 })
 if err != nil {
  log.Printf("Couldn't add item to table. Here's why: %v\n", err)
 }
 return err
}
```

この例で使用している Movie struct を定義します。

```


import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「PutItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルに項目を配置します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To place items into an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedPutItem example.
 */
public class PutItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <albumtitle> <albumtitleval>
 <awards> <awardsval> <Songtitle> <songtitleval>

                Where:
```

```
                    tableName - The Amazon DynamoDB table in which an item is
placed (for example, Music3).
                    key - The key used in the Amazon DynamoDB table (for example,
Artist).
                    keyval - The key value that represents the item to get (for
example, Famous Band).
                    albumTitle - The Album title (for example, AlbumTitle).
                    AlbumTitleValue - The name of the album (for example, Songs
About Life ).
                    Awards - The awards column (for example, Awards).
                    AwardVal - The value of the awards (for example, 10).
                    SongTitle - The song title (for example, SongTitle).
                    SongTitleVal - The value of the song title (for example,
Happy Day).
                **Warning** This program will  place an item that you specify
into a table!
                """;

        if (args.length != 9) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String albumTitle = args[3];
        String albumTitleValue = args[4];
        String awards = args[5];
        String awardVal = args[6];
        String songTitle = args[7];
        String songTitleVal = args[8];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        putItemInTable(ddb, tableName, key, keyVal, albumTitle, albumTitleValue,
awards, awardVal, songTitle,
                songTitleVal);
        System.out.println("Done!");
        ddb.close();
    }
```

```
    public static void putItemInTable(DynamoDbClient ddb,
            String tableName,
            String key,
            String keyVal,
            String albumTitle,
            String albumTitleValue,
            String awards,
            String awardVal,
            String songTitle,
            String songTitleVal) {

        HashMap<String, AttributeValue> itemValues = new HashMap<>();
        itemValues.put(key, AttributeValue.builder().s(keyVal).build());
        itemValues.put(songTitle,
 AttributeValue.builder().s(songTitleVal).build());
        itemValues.put(albumTitle,
 AttributeValue.builder().s(albumTitleValue).build());
        itemValues.put(awards, AttributeValue.builder().s(awardVal).build());

        PutItemRequest request = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();

        try {
            PutItemResponse response = ddb.putItem(request);
            System.out.println(tableName + " was successfully updated. The
 request id is "
                    + response.responseMetadata().requestId());

        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            System.err.println("Be sure that it exists and that you've typed its
 name correctly!");
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「PutItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「PutCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルに項目を配置します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

DynamoDB ドキュメントクライアントを使用して、テーブルに項目を配置します。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun putItemInTable(
    tableNameVal: String,
    key: String,
    keyVal: String,
    albumTitle: String,
```

```
    albumTitleValue: String,
    awards: String,
    awardVal: String,
    songTitle: String,
    songTitleVal: String,
) {
    val itemValues = mutableMapOf<String, AttributeValue>()

    // Add all content to the table.
    itemValues[key] = AttributeValue.S(keyVal)
    itemValues[songTitle] = AttributeValue.S(songTitleVal)
    itemValues[albumTitle] = AttributeValue.S(albumTitleValue)
    itemValues[awards] = AttributeValue.S(awardVal)

    val request =
        PutItemRequest {
            tableName = tableNameVal
            item = itemValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.putItem(request)
        println(" A new item was placed into $tableNameVal.")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「PutItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        echo "What's the name of the last movie you watched?\n";
```

```
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }

        $service->putItem([
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
            'TableName' => $tableName,
        ]);

    public function putItem(array $array)
    {
        $this->dynamoDbClient->putItem($array);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「PutItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 新しい項目を作成する、または既存の項目を新しい項目で置き換えます。

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
```

```
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「PutItem」を参照してください。

Tools for PowerShell V5

例 1: 新しい項目を作成する、または既存の項目を新しい項目で置き換えます。

```
$item = @{
  SongTitle = 'Somewhere Down The Road'
  Artist = 'No One You Know'
        AlbumTitle = 'Somewhat Famous'
        Price = 1.94
        Genre = 'Country'
        CriticRating = 9.0
} | ConvertTo-DDBItem
Set-DDBItem -TableName 'Music' -Item $item
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「PutItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
```

```
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def add_movie(self, title, year, plot, rating):
        """
        Adds a movie to the table.

        :param title: The title of the movie.
        :param year: The release year of the movie.
        :param plot: The plot summary of the movie.
        :param rating: The quality rating of the movie.
        """
        try:
            self.table.put_item(
                Item={
                    "year": year,
                    "title": title,
                    "info": {"plot": plot, "rating": Decimal(str(rating))},
                }
            )
        except ClientError as err:
            logger.error(
                "Couldn't add movie %s to table %s. Here's why: %s: %s",
```

```
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「PutItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Adds a movie to the table.
  #
  # @param movie [Hash] The title, year, plot, and rating of the movie.
  def add_item(movie)
    @table.put_item(
      item: {
        'year' => movie[:year],
        'title' => movie[:title],
        'info' => { 'plot' => movie[:plot], 'rating' => movie[:rating] }
      }
```

```
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't add movie #{title} to table #{@table.name}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「PutItem」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn add_item(client: &Client, item: Item, table: &String) ->
 Result<ItemOut, Error> {
    let user_av = AttributeValue::S(item.username);
    let type_av = AttributeValue::S(item.p_type);
    let age_av = AttributeValue::S(item.age);
    let first_av = AttributeValue::S(item.first);
    let last_av = AttributeValue::S(item.last);

    let request = client
        .put_item()
        .table_name(table)
        .item("username", user_av)
        .item("account_type", type_av)
        .item("age", age_av)
        .item("first_name", first_av)
        .item("last_name", last_av);

    println!("Executing request [{request:?}] to add item...");

    let resp = request.send().await?;
```

```
    let attributes = resp.attributes().unwrap();

    let username = attributes.get("username").cloned();
    let first_name = attributes.get("first_name").cloned();
    let last_name = attributes.get("last_name").cloned();
    let age = attributes.get("age").cloned();
    let p_type = attributes.get("p_type").cloned();

    println!(
        "Added user {:?}, {:?} {:?}, age {:?} as {:?} user",
        username, first_name, last_name, age, p_type
    );

    Ok(ItemOut {
        p_type,
        age,
        username,
        first_name,
        last_name,
    })
}
```

- API の詳細については、AWS SDK for Rust API リファレンスの「PutItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lo_resp) = lo_dyn->putitem(
          iv_tablename = iv_table_name
          it_item      = it_item ).
```

```
        MESSAGE '1 row inserted into DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
 TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「PutItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Add a movie specified as a `Movie` structure to the Amazon DynamoDB
    /// table.
    ///
    /// - Parameter movie: The `Movie` to add to the table.
    ///
    func add(movie: Movie) async throws {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Get a DynamoDB item containing the movie data.
            let item = try await movie.getAsItem()
```

```
            // Send the `PutItem` request to Amazon DynamoDB.

            let input = PutItemInput(
                item: item,
                tableName: self.tableName
            )
            _ = try await client.putItem(input: input)
        } catch {
            print("ERROR: add movie:", dump(error))
            throw error
        }
    }


    ///
    /// Return an array mapping attribute names to Amazon DynamoDB attribute
    /// values, representing the contents of the `Movie` record as a DynamoDB
    /// item.
    ///
    /// - Returns: The movie item as an array of type
    ///   `[Swift.String:DynamoDBClientTypes.AttributeValue]`.
    ///
    func getAsItem() async throws ->
[Swift.String:DynamoDBClientTypes.AttributeValue]  {
        // Build the item record, starting with the year and title, which are
        // always present.

        var item: [Swift.String:DynamoDBClientTypes.AttributeValue] = [
            "year": .n(String(self.year)),
            "title": .s(self.title)
        ]

        // Add the `info` field with the rating and/or plot if they're
        // available.

        var details: [Swift.String:DynamoDBClientTypes.AttributeValue] = [:]
        if (self.info.rating != nil || self.info.plot != nil) {
            if self.info.rating != nil {
                details["rating"] = .n(String(self.info.rating!))
            }
            if self.info.plot != nil {
                details["plot"] = .s(self.info.plot!)
            }
        }
```

```
        item["info"] = .m(details)

        return item
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「PutItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **Query** を使用する

次のサンプルコードは、Query を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- グローバルセカンダリインデックスの高度なシナリオ
- 複数の値を 1 つの属性と比較する
- NoSQL データベースの開始方法
- グローバルセカンダリインデックスを管理する
- 高度なクエリオペレーションを実行する
- begins_with 条件を使用してテーブルをクエリする
- 日付範囲を使用してテーブルをクエリする
- グローバルセカンダリインデックスを使用してテーブルをクエリする
- 複雑なフィルター式を使用してテーブルをクエリする
- 動的フィルター式を使用してテーブルをクエリする
- フィルター式と制限を使用してテーブルをクエリする
- ネストされた属性を使用してテーブルをクエリする
- ページ分割を使用してテーブルをクエリする
- 強力な整合性のある読み込みを使用してテーブルをクエリする

- TTL 項目のクエリ
- 日付と時刻のパターンを使用してテーブルをクエリする
- 式の属性名を使用する
- ローカルセカンダリインデックスを操作する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// <summary>
/// Queries the table for movies released in a particular year and
/// then displays the information for the movies returned.
/// </summary>
/// <param name="tableName">The name of the table to query.</param>
/// <param name="year">The release year for which we want to
/// view movies.</param>
/// <returns>The number of movies that match the query.</returns>
public async Task<int> QueryMoviesAsync(string tableName, int year)
{
    try
    {
        var movieTable = new TableBuilder(_amazonDynamoDB, tableName)
            .AddHashKey("year", DynamoDBEntryType.Numeric)
            .AddRangeKey("title", DynamoDBEntryType.String)
            .Build();

        var filter = new QueryFilter("year", QueryOperator.Equal, year);

        Console.WriteLine("\nFind movies released in: {year}:");

        var config = new QueryOperationConfig()
        {
            Limit = 10, // 10 items per page.
```

```
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
                {
                    "title",
                    "year",
                },
                ConsistentRead = true,
                Filter = filter,
            };

            // Value used to track how many movies match the
            // supplied criteria.
            var moviesFound = 0;

            var search = movieTable.Query(config);
            do
            {
                var movieList = await search.GetNextSetAsync();
                moviesFound += movieList.Count;

                foreach (var movie in movieList)
                {
                    DisplayDocument(movie);
                }
            }
            while (!search.IsDone);

            return moviesFound;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while querying
movies. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while querying movies.
{ex.Message}");
```

```
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「Query」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
###############################################################################
# function dynamodb_query
#
# This function queries a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k key_condition_expression -- The key condition expression.
#       -a attribute_names -- Path to JSON file containing the attribute names.
#       -v attribute_values -- Path to JSON file containing the attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_query() {
  local table_name key_condition_expression attribute_names attribute_values
 projection_expression response
  local option OPTARG # Required to use getopts command in a function.
```

```
 # ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_query"
   echo "Query a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k key_condition_expression -- The key condition expression."
   echo " -a attribute_names -- Path to JSON file containing the attribute
names."
   echo " -v attribute_values -- Path to JSON file containing the attribute
values."
   echo " [-p projection_expression]  -- Optional projection expression."
   echo ""
 }

 while getopts "n:k:a:v:p:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) key_condition_expression="${OPTARG}" ;;
     a) attribute_names="${OPTARG}" ;;
     v) attribute_values="${OPTARG}" ;;
     p) projection_expression="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$key_condition_expression" ]]; then
```

```
    errecho "ERROR: You must provide a key condition expression with the -k
parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_names" ]]; then
    errecho "ERROR: You must provide a attribute names with the -a parameter."
    usage
    return 1
  fi

  if [[ -z "$attribute_values" ]]; then
    errecho "ERROR: You must provide a attribute values with the -v parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values")
  else
    response=$(aws dynamodb query \
      --table-name "$table_name" \
      --key-condition-expression "$key_condition_expression" \
      --expression-attribute-names file://"$attribute_names" \
      --expression-attribute-values file://"$attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports query operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
```

```
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
###############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
```

```
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「Query」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Perform a query on an Amazon DynamoDB Table and retrieve items.
/*!
  \sa queryItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param projectionExpression: The projections expression, which is ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 * The partition key attribute is searched with the specified value. By default,
 all fields and values
 * contained in the item are returned. If an optional projection expression is
 * specified on the command line, only the specified fields and values are
 * returned.
 */
```

```
bool AwsDoc::DynamoDB::queryItems(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &projectionExpression,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
   Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
   Aws::DynamoDB::Model::QueryRequest request;

   request.SetTableName(tableName);

   if (!projectionExpression.empty()) {
       request.SetProjectionExpression(projectionExpression);
   }

   // Set query key condition expression.
   request.SetKeyConditionExpression(partitionKey + "= :valueToMatch");

   // Set Expression AttributeValues.
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue> attributeValues;
   attributeValues.emplace(":valueToMatch", partitionValue);

   request.SetExpressionAttributeValues(attributeValues);

   bool result = true;

   // "exclusiveStartKey" is used for pagination.
   Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
exclusiveStartKey;
   do {
       if (!exclusiveStartKey.empty()) {
           request.SetExclusiveStartKey(exclusiveStartKey);
           exclusiveStartKey.clear();
       }
       // Perform Query operation.
       const Aws::DynamoDB::Model::QueryOutcome &outcome =
dynamoClient.Query(request);
       if (outcome.IsSuccess()) {
           // Reference the retrieved items.
           const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
           if (!items.empty()) {
```

```
                std::cout << "Number of items retrieved from Query: " <<
 items.size()
                          << std::endl;
                // Iterate each item and print.
                for (const auto &item: items) {
                    std::cout
                            <<
 "*************************************************"
                            << std::endl;
                    // Output each retrieved field and its value.
                    for (const auto &i: item)
                        std::cout << i.first << ": " << i.second.GetS() <<
 std::endl;
                }
            }
            else {
                std::cout << "No item found in table: " << tableName <<
 std::endl;
            }

            exclusiveStartKey = outcome.GetResult().GetLastEvaluatedKey();
        }
        else {
            std::cerr << "Failed to Query items: " <<
 outcome.GetError().GetMessage();
            result = false;
            break;
        }
    } while (!exclusiveStartKey.empty());

    return result;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「Query」を参照してください。

CLI

AWS CLI

例 1: テーブルにクエリを実行するには

次の query の例では、MusicCollection テーブルの項目にクエリを実行します。テーブルにはハッシュおよび範囲プライマリキー (ArtistおよびSongTitle) がありますが、このクエリではハッシュキー値のみを指定します。「No One You Know」という名前のアーティストの曲タイトルが返されます。

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --return-consumed-capacity TOTAL
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 2: 強力な整合性のある読み込みを使用してテーブルにクエリを実行し、インデックスを降順で走査するには

次の例では、最初の例と同じクエリを実行しますが、結果は逆の順序で返され、強力な整合性のある読み込みが使用されます。

```
aws dynamodb query \
    --table-name MusicCollection \
    --projection-expression "SongTitle" \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --consistent-read \
    --no-scan-index-forward \
    --return-consumed-capacity TOTAL
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 1.0
    }
```

```
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 3: 特定の結果を除外するには

次の例は、MusicCollection をクエリしますが、AlbumTitle 属性に特定の値を含む結果を除外します。このフィルターは項目が読み込まれた後に適用されるため、ScannedCount または ConsumedCapacity には影響しないことに注意してください。

```
aws dynamodb query \
    --table-name MusicCollection \
    --key-condition-expression "#n1 = :v1" \
    --filter-expression "NOT (#n2 IN (:v2, :v3))" \
    --expression-attribute-names file://names.json \
    --expression-attribute-values file://values.json \
    --return-consumed-capacity TOTAL
```

values.json の内容:

```
{
    ":v1": {"S": "No One You Know"},
    ":v2": {"S": "Blue Sky Blues"},
    ":v3": {"S": "Greatest Hits"}
}
```

names.json の内容:

```
{
    "#n1": "Artist",
    "#n2": "AlbumTitle"
}
```

出力:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
```

```
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 4: 項目数だけを取得するには

次の例は、クエリに一致する項目数を取得しますが、項目自体は取得しません。

```
aws dynamodb query \
    --table-name MusicCollection \
    --select COUNT \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": null
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

例 5: インデックスをクエリするには

次の例は、ローカルセカンダリインデックス AlbumTitleIndex をクエリします。クエリは、ローカルセカンダリインデックスに射影されたベーステーブルのすべての属性を返します。ローカルセカンダリインデックスまたはグローバルセカンダリインデックスをクエリする場合は、table-name パラメータを使用してベーステーブルの名前も指定する必要があることに注意してください。

```
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumTitleIndex \
    --key-condition-expression "Artist = :v1" \
    --expression-attribute-values file://expression-attributes.json \
    --select ALL_PROJECTED_ATTRIBUTES \
    --return-consumed-capacity INDEXES
```

expression-attributes.json の内容:

```
{
    ":v1": {"S": "No One You Know"}
}
```

出力:

```
{
    "Items": [
        {
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Scared of My Shadow"
            }
        },
        {
```

```
            "AlbumTitle": {
                "S": "Somewhat Famous"
            },
            "Artist": {
                "S": "No One You Know"
            },
            "SongTitle": {
                "S": "Call Me Today"
            }
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 0.5,
        "Table": {
            "CapacityUnits": 0.0
        },
        "LocalSecondaryIndexes": {
            "AlbumTitleIndex": {
                "CapacityUnits": 0.5
            }
        }
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB のクエリオペレーション」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「Query」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Query gets all movies in the DynamoDB table that were released in the
 specified year.
// The function uses the `expression` package to build the key condition
 expression
// that is used in the query.
func (basics TableBasics) Query(ctx context.Context, releaseYear int) ([]Movie,
 error) {
 var err error
 var response *dynamodb.QueryOutput
 var movies []Movie
 keyEx := expression.Key("year").Equal(expression.Value(releaseYear))
 expr, err := expression.NewBuilder().WithKeyCondition(keyEx).Build()
 if err != nil {
  log.Printf("Couldn't build expression for query. Here's why: %v\n", err)
 } else {
  queryPaginator := dynamodb.NewQueryPaginator(basics.DynamoDbClient,
 &dynamodb.QueryInput{
   TableName:                 aws.String(basics.TableName),
```

```
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   KeyConditionExpression:    expr.KeyCondition(),
  })
  for queryPaginator.HasMorePages() {
   response, err = queryPaginator.NextPage(ctx)
   if err != nil {
    log.Printf("Couldn't query for movies released in %v. Here's why: %v\n",
 releaseYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「Query」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルに対してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To query items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client. See the EnhancedQueryRecords example.
 */
public class Query {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <partitionKeyName> <partitionKeyVal>

                Where:
                    tableName - The Amazon DynamoDB table to put the item in (for
 example, Music3).
```

```
                    partitionKeyName - The partition key name of the Amazon
 DynamoDB table (for example, Artist).
                    partitionKeyVal - The value of the partition key that should
 match (for example, Famous Band).
                """;

        if (args.length != 3) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String partitionKeyName = args[1];
        String partitionKeyVal = args[2];

        // For more information about an alias, see:
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Expressions.ExpressionAttributeNames.html
        String partitionAlias = "#a";

        System.out.format("Querying %s", tableName);
        System.out.println("");
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        int count = queryTable(ddb, tableName, partitionKeyName, partitionKeyVal,
 partitionAlias);
        System.out.println("There were " + count + "  record(s) returned");
        ddb.close();
    }

    public static int queryTable(DynamoDbClient ddb, String tableName, String
 partitionKeyName, String partitionKeyVal,
            String partitionAlias) {
        // Set up an alias for the partition key name in case it's a reserved
 word.
        HashMap<String, String> attrNameAlias = new HashMap<String, String>();
        attrNameAlias.put(partitionAlias, partitionKeyName);

        // Set up mapping of the partition name with the value.
        HashMap<String, AttributeValue> attrValues = new HashMap<>();
        attrValues.put(":" + partitionKeyName, AttributeValue.builder()
```

```
                .s(partitionKeyVal)
                .build());

        QueryRequest queryReq = QueryRequest.builder()
                .tableName(tableName)
                .keyConditionExpression(partitionAlias + " = :" +
 partitionKeyName)
                .expressionAttributeNames(attrNameAlias)
                .expressionAttributeValues(attrValues)
                .build();

        try {
            QueryResponse response = ddb.query(queryReq);
            return response.count();

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return -1;
    }
}
```

DynamoDbClient とセカンダリインデックスを使用してテーブルに対してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import java.util.HashMap;
import java.util.Map;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
```

```
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * Create the Movies table by running the Scenario example and loading the Movie
 * data from the JSON file. Next create a secondary
 * index for the Movies table that uses only the year column. Name the index
 * **year-index**. For more information, see:
 *
 * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html
 */
public class QueryItemsUsingIndex {
    public static void main(String[] args) {
        String tableName = "Movies";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        queryIndex(ddb, tableName);
        ddb.close();
    }

    public static void queryIndex(DynamoDbClient ddb, String tableName) {
        try {
            Map<String, String> expressionAttributesNames = new HashMap<>();
            expressionAttributesNames.put("#year", "year");
            Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
            expressionAttributeValues.put(":yearValue",
 AttributeValue.builder().n("2013").build());

            QueryRequest request = QueryRequest.builder()
                    .tableName(tableName)
                    .indexName("year-index")
                    .keyConditionExpression("#year = :yearValue")
                    .expressionAttributeNames(expressionAttributesNames)
                    .expressionAttributeValues(expressionAttributeValues)
                    .build();

            System.out.println("=== Movie Titles ===");
            QueryResponse response = ddb.query(request);
            response.items()
                    .forEach(movie ->
 System.out.println(movie.get("title").s()));
```

```
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「QueryCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
    ConsistentRead: true,
```

```
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Success", data.Items);
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun queryDynTable(
    tableNameVal: String,
    partitionKeyName: String,
    partitionKeyVal: String,
    partitionAlias: String,
): Int {
    val attrNameAlias = mutableMapOf<String, String>()
    attrNameAlias[partitionAlias] = partitionKeyName

    // Set up mapping of the partition name with the value.
    val attrValues = mutableMapOf<String, AttributeValue>()
    attrValues[":$partitionKeyName"] = AttributeValue.S(partitionKeyVal)

    val request =
        QueryRequest {
            tableName = tableNameVal
            keyConditionExpression = "$partitionAlias = :$partitionKeyName"
            expressionAttributeNames = attrNameAlias
            this.expressionAttributeValues = attrValues
        }
```

```
    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.query(request)
        return response.count
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「Query」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);

    public function query(string $tableName, $key)
    {
        $expressionAttributeValues = [];
        $expressionAttributeNames = [];
        $keyConditionExpression = "";
        $index = 1;
        foreach ($key as $name => $value) {
            $keyConditionExpression .= "#" . array_key_first($value) . " = :v
$index,";
            $expressionAttributeNames["#" . array_key_first($value)] =
 array_key_first($value);
            $hold = array_pop($value);
```

```
            $expressionAttributeValues[":v$index"] = [
                array_key_first($hold) => array_pop($hold),
            ];
        }
        $keyConditionExpression = substr($keyConditionExpression, 0, -1);
        $query = [
            'ExpressionAttributeValues' => $expressionAttributeValues,
            'ExpressionAttributeNames' => $expressionAttributeNames,
            'KeyConditionExpression' => $keyConditionExpression,
            'TableName' => $tableName,
        ];
        return $this->dynamoDbClient->query($query);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「Query」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定の SongTitle と Artist を含む DynamoDB 項目を返すクエリを呼び出します。

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
```

```
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「Query」を参照してください。

Tools for PowerShell V5

例 1: 指定の SongTitle と Artist を含む DynamoDB 項目を返すクエリを呼び出します。

```
$invokeDDBQuery = @{
    TableName = 'Music'
    KeyConditionExpression = ' SongTitle = :SongTitle and Artist = :Artist'
    ExpressionAttributeValues = @{
        ':SongTitle' = 'Somewhere Down The Road'
        ':Artist' = 'No One You Know'
    } | ConvertTo-DDBItem
}
Invoke-DDBQuery @invokeDDBQuery | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「Query」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

キー条件式を使用して項目に対してクエリを実行します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None
```

```
    def query_movies(self, year):
        """
        Queries for movies that were released in the specified year.

        :param year: The year to query.
        :return: The list of movies that were released in the specified year.
        """
        try:
            response =
 self.table.query(KeyConditionExpression=Key("year").eq(year))
        except ClientError as err:
            logger.error(
                "Couldn't query for movies released in %s. Here's why: %s: %s",
                year,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Items"]
```

項目に対してクエリを実行し、データのサブセットを返すように射影します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def query_and_project_movies(self, year, title_bounds):
        """
        Query for movies that were released in a specified year and that have
 titles
        that start within a range of letters. A projection expression is used
        to return a subset of data for each movie.

        :param year: The release year to query.
        :param title_bounds: The range of starting letters to query.
        :return: The list of movies.
        """
```

```
        try:
            response = self.table.query(
                ProjectionExpression="#yr, title, info.genres, info.actors[0]",
                ExpressionAttributeNames={"#yr": "year"},
                KeyConditionExpression=(
                    Key("year").eq(year)
                    & Key("title").between(
                        title_bounds["first"], title_bounds["second"]
                    )
                ),
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ValidationException":
                logger.warning(
                    "There's a validation error. Here's the message: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            else:
                logger.error(
                    "Couldn't query for movies. Here's why: %s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Items"]
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Queries for movies that were released in the specified year.
  #
  # @param year [Integer] The year to query.
  # @return [Array] The list of movies that were released in the specified year.
  def query_items(year)
    response = @table.query(
      key_condition_expression: '#yr = :year',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: { ':year' => year }
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't query for movies released in #{year}. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.items
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「Query」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

指定した年に作られた映画を検索します。

```
pub async fn movies_in_year(
    client: &Client,
    table_name: &str,
    year: u16,
) -> Result<Vec<Movie>, MovieError> {
    let results = client
        .query()
        .table_name(table_name)
        .key_condition_expression("#yr = :yyyy")
        .expression_attribute_names("#yr", "year")
        .expression_attribute_values(":yyyy",
 AttributeValue::N(year.to_string()))
        .send()
        .await?;

    if let Some(items) = results.items {
        let movies = items.iter().map(|v| v.into()).collect();
        Ok(movies)
    } else {
        Ok(vec![])
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「Query」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
```

```
        " Query movies for a given year .
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_year }| ) ) ).
        DATA(lt_key_conditions) = VALUE /aws1/cl_dyncondition=>tt_keyconditions(
          ( VALUE /aws1/cl_dyncondition=>ts_keyconditions_maprow(
          key = 'year'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |EQ|
          ) ) ) ).
        oo_result = lo_dyn->query(
          iv_tablename = iv_table_name
          it_keyconditions = lt_key_conditions ).
        DATA(lt_items) = oo_result->get_items( ).
        "You can loop over the results to get item attributes.
        LOOP AT lt_items INTO DATA(lt_item).
          DATA(lo_title) = lt_item[ key = 'title' ]-value.
          DATA(lo_year) = lt_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_result->get_count( ).
        MESSAGE 'Item count is: ' && lv_count TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「Query」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB
```

```
    /// Get all the movies released in the specified year.
    ///
    /// - Parameter year: The release year of the movies to return.
    ///
    /// - Returns: An array of `Movie` objects describing each matching movie.
    ///
    func getMovies(fromYear year: Int) async throws -> [Movie] {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = QueryInput(
                expressionAttributeNames: [
                    "#y": "year"
                ],
                expressionAttributeValues: [
                    ":y": .n(String(year))
                ],
                keyConditionExpression: "#y = :y",
                tableName: self.tableName
            )
            // Use "Paginated" to get all the movies.
            // This lets the SDK handle the 'lastEvaluatedKey' property in
"QueryOutput".

            let pages = client.queryPaginated(input: input)

            var movieList: [Movie] = []
            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
                    continue
                }

                // Convert the found movies into `Movie` objects and return an
array
                // of them.

                for item in items {
                    let movie = try Movie(withItem: item)
                    movieList.append(movie)
```

```
                }
            }
            return movieList
        } catch {
            print("ERROR: getMovies:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **Scan** を使用する

次のサンプルコードは、Scan を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- DAX で読み取りを高速化
- 複数の値を 1 つの属性と比較する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Scans the table for movies released between the specified years.
    /// </summary>
    /// <param name="tableName">The name of the table to scan.</param>
    /// <param name="startYear">The starting year for the range.</param>
    /// <param name="endYear">The ending year for the range.</param>
    /// <returns>The number of movies found in the specified year range.</
returns>
    public async Task<int> ScanTableAsync(
        string tableName,
        int startYear,
        int endYear)
    {
        try
        {
            var request = new ScanRequest
            {
                TableName = tableName,
                ExpressionAttributeNames = new Dictionary<string, string>
                {
                    { "#yr", "year" },
                },
                ExpressionAttributeValues = new Dictionary<string,
 AttributeValue>
                {
                    { ":y_a", new AttributeValue { N = startYear.ToString() } },
                    { ":y_z", new AttributeValue { N = endYear.ToString() } },
                },
                FilterExpression = "#yr between :y_a and :y_z",
                ProjectionExpression = "#yr, title, info.actors[0],
 info.directors, info.running_time_secs",
                Limit = 10 // Set a limit to demonstrate using the
 LastEvaluatedKey.
            };

            // Keep track of how many movies were found.
            int foundCount = 0;

            var response = new ScanResponse();
            do
            {
                response = await _amazonDynamoDB.ScanAsync(request);
                foundCount += response.Items.Count;
```

```
                response.Items.ForEach(i => DisplayItem(i));
                request.ExclusiveStartKey = response.LastEvaluatedKey;
            }
            while (response?.LastEvaluatedKey?.Count > 0);
            return foundCount;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} was not found. {ex.Message}");
            return 0;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while scanning
table. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while scanning table.
{ex.Message}");
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「Scan」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
```

```
# function dynamodb_scan
#
# This function scans a DynamoDB table.
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -f filter_expression  -- The filter expression.
#       -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names.
#       -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values.
#       [-p projection_expression]  -- Optional projection expression.
#
#  Returns:
#       The items as json output.
#  And:
#       0 - If successful.
#       1 - If it fails.
###############################################################################
function dynamodb_scan() {
  local table_name filter_expression expression_attribute_names
 expression_attribute_values projection_expression response
  local option OPTARG # Required to use getopts command in a function.

  # ######################################
  # Function usage explanation
  ######################################
  function usage() {
    echo "function dynamodb_scan"
    echo "Scan a DynamoDB table."
    echo " -n table_name  -- The name of the table."
    echo " -f filter_expression  -- The filter expression."
    echo " -a expression_attribute_names -- Path to JSON file containing the
 expression attribute names."
    echo " -v expression_attribute_values -- Path to JSON file containing the
 expression attribute values."
    echo " [-p projection_expression]  -- Optional projection expression."
    echo ""
  }

  while getopts "n:f:a:v:p:h" option; do
    case "${option}" in
      n) table_name="${OPTARG}" ;;
      f) filter_expression="${OPTARG}" ;;
```

```
      a) expression_attribute_names="${OPTARG}" ;;
      v) expression_attribute_values="${OPTARG}" ;;
      p) projection_expression="${OPTARG}" ;;
      h)
        usage
        return 0
        ;;
      \?)
        echo "Invalid parameter"
        usage
        return 1
        ;;
    esac
  done
  export OPTIND=1

  if [[ -z "$table_name" ]]; then
    errecho "ERROR: You must provide a table name with the -n parameter."
    usage
    return 1
  fi

  if [[ -z "$filter_expression" ]]; then
    errecho "ERROR: You must provide a filter expression with the -f parameter."
    usage
    return 1
  fi

  if [[ -z "$expression_attribute_names" ]]; then
    errecho "ERROR: You must provide expression attribute names with the -a
parameter."
    usage
    return 1
  fi

  if [[ -z "$expression_attribute_values" ]]; then
    errecho "ERROR: You must provide expression attribute values with the -v
parameter."
    usage
    return 1
  fi

  if [[ -z "$projection_expression" ]]; then
    response=$(aws dynamodb scan \
```

```
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values")
  else
    response=$(aws dynamodb scan \
      --table-name "$table_name" \
      --filter-expression "$filter_expression" \
      --expression-attribute-names file://"$expression_attribute_names" \
      --expression-attribute-values file://"$expression_attribute_values" \
      --projection-expression "$projection_expression")
  fi

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports scan operation failed.$response"
    return 1
  fi

  echo "$response"

  return 0
}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
###############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

###############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
```

```
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
################################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「Scan」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Scan an Amazon DynamoDB table.
/*!
  \sa scanTable()
  \param tableName: Name for the DynamoDB table.
  \param projectionExpression: An optional projection expression, ignored if
 empty.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */

bool AwsDoc::DynamoDB::scanTable(const Aws::String &tableName,
                                 const Aws::String &projectionExpression,
                                 const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);
    Aws::DynamoDB::Model::ScanRequest request;
    request.SetTableName(tableName);

    if (!projectionExpression.empty())
        request.SetProjectionExpression(projectionExpression);

    Aws::Vector<Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>>
 all_items;
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 last_evaluated_key; // Used for pagination;
    do {
        if (!last_evaluated_key.empty()) {
            request.SetExclusiveStartKey(last_evaluated_key);
        }
        const Aws::DynamoDB::Model::ScanOutcome &outcome =
 dynamoClient.Scan(request);
        if (outcome.IsSuccess()) {
```

```
            // Reference the retrieved items.
            const Aws::Vector<Aws::Map<Aws::String,
 Aws::DynamoDB::Model::AttributeValue>> &items = outcome.GetResult().GetItems();
            all_items.insert(all_items.end(), items.begin(), items.end());

            last_evaluated_key = outcome.GetResult().GetLastEvaluatedKey();
        }
        else {
            std::cerr << "Failed to Scan items: " <<
 outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }

    } while (!last_evaluated_key.empty());

    if (!all_items.empty()) {
        std::cout << "Number of items retrieved from scan: " << all_items.size()
                  << std::endl;
        // Iterate each item and print.
        for (const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 &itemMap: all_items) {
            std::cout << "*************************************************"
                      << std::endl;
            // Output each retrieved field and its value.
            for (const auto &itemEntry: itemMap)
                std::cout << itemEntry.first << ": " << itemEntry.second.GetS()
                          << std::endl;
        }
    }

    else {
        std::cout << "No items found in table: " << tableName << std::endl;
    }

    return true;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「Scan」を参照してください。

CLI

AWS CLI

テーブルをスキャンするには

次の scan の例は、MusicCollection テーブル全体をスキャンし、その結果をアーティスト「No One You Know」の曲に絞り込みます。各項目について、アルバムタイトルと曲タイトルのみが返されます。

```
aws dynamodb scan \
    --table-name MusicCollection \
    --filter-expression "Artist = :a" \
    --projection-expression "#ST, #AT" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json
```

expression-attribute-names.json の内容:

```
{
    "#ST": "SongTitle",
    "#AT":"AlbumTitle"
}
```

expression-attribute-values.json の内容:

```
{
    ":a": {"S": "No One You Know"}
}
```

出力:

```
{
    "Count": 2,
    "Items": [
        {
            "SongTitle": {
                "S": "Call Me Today"
            },
            "AlbumTitle": {
                "S": "Somewhat Famous"
            }
```

```
        },
        {
            "SongTitle": {
                "S": "Scared of My Shadow"
            },
            "AlbumTitle": {
                "S": "Blue Sky Blues"
            }
        }
    ],
    "ScannedCount": 3,
    "ConsumedCapacity": null
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「DynamoDB でのスキャンの使用」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「Scan」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
```

```
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// Scan gets all movies in the DynamoDB table that were released in a range of
 years
// and projects them to return a reduced set of fields.
// The function uses the `expression` package to build the filter and projection
// expressions.
func (basics TableBasics) Scan(ctx context.Context, startYear int, endYear int)
 ([]Movie, error) {
 var movies []Movie
 var err error
 var response *dynamodb.ScanOutput
 filtEx := expression.Name("year").Between(expression.Value(startYear),
 expression.Value(endYear))
 projEx := expression.NamesList(
  expression.Name("year"), expression.Name("title"),
 expression.Name("info.rating"))
 expr, err :=
 expression.NewBuilder().WithFilter(filtEx).WithProjection(projEx).Build()
 if err != nil {
  log.Printf("Couldn't build expressions for scan. Here's why: %v\n", err)
 } else {
  scanPaginator := dynamodb.NewScanPaginator(basics.DynamoDbClient,
 &dynamodb.ScanInput{
   TableName:                 aws.String(basics.TableName),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   FilterExpression:          expr.Filter(),
   ProjectionExpression:      expr.Projection(),
  })
  for scanPaginator.HasMorePages() {
   response, err = scanPaginator.NextPage(ctx)
   if err != nil {
```

```
    log.Printf("Couldn't scan for movies released between %v and %v. Here's why:
 %v\n",
     startYear, endYear, err)
    break
   } else {
    var moviePage []Movie
    err = attributevalue.UnmarshalListOfMaps(response.Items, &moviePage)
    if err != nil {
     log.Printf("Couldn't unmarshal query response. Here's why: %v\n", err)
     break
    } else {
     movies = append(movies, moviePage...)
    }
   }
  }
 }
 return movies, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
```

```
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「Scan」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用して Amazon DynamoDB テーブルをスキャンします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import java.util.Map;
import java.util.Set;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * To scan items from an Amazon DynamoDB table using the AWS SDK for Java V2,
 * its better practice to use the
 * Enhanced Client, See the EnhancedScanRecords example.
 */

public class DynamoDBScanItems {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <tableName>

                Where:
                    tableName - The Amazon DynamoDB table to get information from
 (for example, Music3).
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        Region region = Region.US_EAST_1;
```

```
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        scanItems(ddb, tableName);
        ddb.close();
    }

    public static void scanItems(DynamoDbClient ddb, String tableName) {
        try {
            ScanRequest scanRequest = ScanRequest.builder()
                    .tableName(tableName)
                    .build();

            ScanResponse response = ddb.scan(scanRequest);
            for (Map<String, AttributeValue> item : response.items()) {
                Set<String> keys = item.keySet();
                for (String key : keys) {
                    System.out.println("The key name is " + key + "\n");
                    System.out.println("The value is " + item.get(key).s());
                }
            }

        } catch (DynamoDbException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Scan」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「ScanCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, ScanCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ScanCommand({
    ProjectionExpression: "#Name, Color, AvgLifeSpan",
    ExpressionAttributeNames: { "#Name": "Name" },
    TableName: "Birds",
  });

  const response = await docClient.send(command);
  for (const bird of response.Items) {
    console.log(`${bird.Name} - (${bird.Color}, ${bird.AvgLifeSpan})`);
  }
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Scan」を参照してください。

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object.
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

const params = {
  // Specify which items in the results are returned.
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
  // Define the expression attribute value, which are substitutes for the values
 you want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: "SubTitle2" },
    ":s": { N: 1 },
    ":e": { N: 2 },
  },
  // Set the projection expression, which are the attributes that you want.
  ProjectionExpression: "Season, Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
    data.Items.forEach(function (element, index, array) {
      console.log(
        "printing",
        element.Title.S + " (" + element.Subtitle.S + ")"
      );
    });
```

```
  }
});
```

- 詳細については、「AWS SDK for JavaScript デベロッパーガイド」を参照してください。
- API の詳細については、AWS SDK for JavaScript API リファレンスの「Scan」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun scanItems(tableNameVal: String) {
    val request =
        ScanRequest {
            tableName = tableNameVal
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        val response = ddb.scan(request)
        response.items?.forEach { item ->
            item.keys.forEach { key ->
                println("The key name is $key\n")
                println("The value is ${item[key]}")
            }
        }
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「Scan」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    $yearsKey = [
        'Key' => [
            'year' => [
                'N' => [
                    'minRange' => 1990,
                    'maxRange' => 1999,
                ],
            ],
        ],
    ];
    $filter = "year between 1990 and 1999";
    echo "\nHere's a list of all the movies released in the 90s:\n";
    $result = $service->scan($tableName, $yearsKey, $filter);
    foreach ($result['Items'] as $movie) {
        $movie = $marshal->unmarshalItem($movie);
        echo $movie['title'] . "\n";
    }

public function scan(string $tableName, array $key, string $filters)
{
    $query = [
        'ExpressionAttributeNames' => ['#year' => 'year'],
        'ExpressionAttributeValues' => [
            ":min" => ['N' => '1990'],
            ":max" => ['N' => '1999'],
        ],
        'FilterExpression' => "#year between :min and :max",
        'TableName' => $tableName,
    ];
    return $this->dynamoDbClient->scan($query);
}
```

- API の詳細については、AWS SDK for PHP API リファレンスの「Scan」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: Music テーブルのすべての項目を返します。

```
Invoke-DDBScan -TableName 'Music' | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
Genre                          Country
Artist                         No One You Know
Price                          1.98
CriticRating                   8.4
SongTitle                      My Dog Spot
AlbumTitle                     Hey Now
```

例 2: Music テーブル内の CriticRating が 9 以上の項目を返します。

```
$scanFilter = @{
        CriticRating = [Amazon.DynamoDBv2.Model.Condition]@{
            AttributeValueList = @(@{N = '9'})
            ComparisonOperator = 'GE'
        }
    }
    Invoke-DDBScan -TableName 'Music' -ScanFilter $scanFilter | ConvertFrom-
DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「Scan」を参照してください。

Tools for PowerShell V5

例 1: Music テーブルのすべての項目を返します。

```
Invoke-DDBScan -TableName 'Music' | ConvertFrom-DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
Genre                          Country
Artist                         No One You Know
Price                          1.98
CriticRating                   8.4
SongTitle                      My Dog Spot
AlbumTitle                     Hey Now
```

例 2: Music テーブル内の CriticRating が 9 以上の項目を返します。

```
$scanFilter = @{
        CriticRating = [Amazon.DynamoDBv2.Model.Condition]@{
            AttributeValueList = @(@{N = '9'})
            ComparisonOperator = 'GE'
        }
    }
```

```
    Invoke-DDBScan -TableName 'Music' -ScanFilter $scanFilter | ConvertFrom-
DDBItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Country
Artist                         No One You Know
Price                          1.94
CriticRating                   9
SongTitle                      Somewhere Down The Road
AlbumTitle                     Somewhat Famous
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「Scan」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
```

```
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def scan_movies(self, year_range):
        """
        Scans for movies that were released in a range of years.
        Uses a projection expression to return a subset of data for each movie.

        :param year_range: The range of years to retrieve.
        :return: The list of movies released in the specified years.
        """
        movies = []
        scan_kwargs = {
            "FilterExpression": Key("year").between(
                year_range["first"], year_range["second"]
            ),
            "ProjectionExpression": "#yr, title, info.rating",
            "ExpressionAttributeNames": {"#yr": "year"},
        }
        try:
            done = False
            start_key = None
            while not done:
                if start_key:
                    scan_kwargs["ExclusiveStartKey"] = start_key
                response = self.table.scan(**scan_kwargs)
                movies.extend(response.get("Items", []))
                start_key = response.get("LastEvaluatedKey", None)
```

```
                done = start_key is None
        except ClientError as err:
            logger.error(
                "Couldn't scan for movies. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise

        return movies
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Scan」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Scans for movies that were released in a range of years.
  # Uses a projection expression to return a subset of data for each movie.
  #
  # @param year_range [Hash] The range of years to retrieve.
  # @return [Array] The list of movies released in the specified years.
  def scan_items(year_range)
```

```
    movies = []
    scan_hash = {
      filter_expression: '#yr between :start_yr and :end_yr',
      projection_expression: '#yr, title, info.rating',
      expression_attribute_names: { '#yr' => 'year' },
      expression_attribute_values: {
        ':start_yr' => year_range[:start], ':end_yr' => year_range[:end]
      }
    }
    done = false
    start_key = nil
    until done
      scan_hash[:exclusive_start_key] = start_key unless start_key.nil?
      response = @table.scan(scan_hash)
      movies.concat(response.items) unless response.items.empty?
      start_key = response.last_evaluated_key
      done = start_key.nil?
    end
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't scan for movies. Here's why:")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    movies
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「Scan」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
pub async fn list_items(client: &Client, table: &str, page_size: Option<i32>) ->
 Result<(), Error> {
    let page_size = page_size.unwrap_or(10);
    let items: Result<Vec<_>, _> = client
        .scan()
        .table_name(table)
        .limit(page_size)
        .into_paginator()
        .items()
        .send()
        .collect()
        .await;

    println!("Items in table (up to {page_size}):");
    for item in items? {
        println!("   {:?}", item);
    }

    Ok(())
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「Scan」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        " Scan movies for rating greater than or equal to the rating specified
        DATA(lt_attributelist) = VALUE /aws1/
cl_dynattributevalue=>tt_attributevaluelist(
            ( NEW /aws1/cl_dynattributevalue( iv_n = |{ iv_rating }| ) ) ).
```

```
        DATA(lt_filter_conditions) = VALUE /aws1/
cl_dyncondition=>tt_filterconditionmap(
          ( VALUE /aws1/cl_dyncondition=>ts_filterconditionmap_maprow(
          key = 'rating'
          value = NEW /aws1/cl_dyncondition(
          it_attributevaluelist = lt_attributelist
          iv_comparisonoperator = |GE|
          ) ) ) ).
        oo_scan_result = lo_dyn->scan( iv_tablename = iv_table_name
          it_scanfilter = lt_filter_conditions ).
        DATA(lt_items) = oo_scan_result->get_items( ).
        LOOP AT lt_items INTO DATA(lo_item).
          " You can loop over to get individual attributes.
          DATA(lo_title) = lo_item[ key = 'title' ]-value.
          DATA(lo_year) = lo_item[ key = 'year' ]-value.
        ENDLOOP.
        DATA(lv_count) = oo_scan_result->get_count( ).
        MESSAGE 'Found ' && lv_count && ' items' TYPE 'I'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「Scan」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Return an array of `Movie` objects released in the specified range of
    /// years.
```

```
    ///
    /// - Parameters:
    ///   - firstYear: The first year of movies to return.
    ///   - lastYear: The last year of movies to return.
    ///   - startKey: A starting point to resume processing; always use `nil`.
    ///
    /// - Returns: An array of `Movie` objects describing the matching movies.
    ///
    /// > Note: The `startKey` parameter is used by this function when
    ///   recursively calling itself, and should always be `nil` when calling
    ///   directly.
    ///
    func getMovies(firstYear: Int, lastYear: Int,
                   startKey: [Swift.String: DynamoDBClientTypes.AttributeValue]?
= nil)
        async throws -> [Movie]
    {
        do {
            var movieList: [Movie] = []

            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            let input = ScanInput(
                consistentRead: true,
                exclusiveStartKey: startKey,
                expressionAttributeNames: [
                    "#y": "year" // `year` is a reserved word, so use `#y`
instead.
                ],
                expressionAttributeValues: [
                    ":y1": .n(String(firstYear)),
                    ":y2": .n(String(lastYear))
                ],
                filterExpression: "#y BETWEEN :y1 AND :y2",
                tableName: self.tableName
            )

            let pages = client.scanPaginated(input: input)

            for try await page in pages {
                guard let items = page.items else {
                    print("Error: no items returned.")
```

```
                continue
            }

            // Build an array of `Movie` objects for the returned items.

            for item in items {
                let movie = try Movie(withItem: item)
                movieList.append(movie)
            }
        }
        return movieList

    } catch {
        print("ERROR: getMovies with scan:", dump(error))
        throw error
    }
}
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「Scan」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateItem** を使用する

次のサンプルコードは、UpdateItem を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 基本を学ぶ
- 項目の TTL を条件付きで更新する
- カウント式演算子
- MRSC グローバルテーブルの作成と管理
- NoSQL データベースの開始方法
- リストオペレーションを実行する

- マップオペレーションを実行する
- セットオペレーションを実行する
- 式順序の更新を理解する
- 項目を TTL を更新する
- アトミックカウンタオペレーションを使用する
- 条件付きオペレーションを使用する
- 式の属性名を使用する

.NET

SDK for .NET (v4)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Updates an existing item in the movies table.
    /// </summary>
    /// <param name="newMovie">A Movie object containing information for
    /// the movie to update.</param>
    /// <param name="newInfo">A MovieInfo object that contains the
    /// information that will be changed.</param>
    /// <param name="tableName">The name of the table that contains the movie.</
param>
    /// <returns>A Boolean value that indicates the success of the operation.</
returns>
    public async Task<bool> UpdateItemAsync(
        Movie newMovie,
        MovieInfo newInfo,
        string tableName)
    {
        try
        {
            var key = new Dictionary<string, AttributeValue>
            {
```

```
                ["title"] = new AttributeValue { S = newMovie.Title },
                ["year"] = new AttributeValue { N = newMovie.Year.ToString() },
            };
            var updates = new Dictionary<string, AttributeValueUpdate>
            {
                ["info.plot"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { S = newInfo.Plot },
                },

                ["info.rating"] = new AttributeValueUpdate
                {
                    Action = AttributeAction.PUT,
                    Value = new AttributeValue { N = newInfo.Rank.ToString() },
                },
            };

            var request = new UpdateItemRequest
            {
                AttributeUpdates = updates,
                Key = key,
                TableName = tableName,
            };

            await _amazonDynamoDB.UpdateItemAsync(request);
            return true;
        }
        catch (ResourceNotFoundException ex)
        {
            Console.WriteLine($"Table {tableName} or item was not found.
{ex.Message}");
            return false;
        }
        catch (AmazonDynamoDBException ex)
        {
            Console.WriteLine($"An Amazon DynamoDB error occurred while updating
item. {ex.Message}");
            throw;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while updating item.
{ex.Message}");
```

```
            throw;
        }
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「UpdateItem」を参照してください。

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
##############################################################################
# function dynamodb_update_item
#
# This function updates an item in a DynamoDB table.
#
#
# Parameters:
#       -n table_name  -- The name of the table.
#       -k keys  -- Path to json file containing the keys that identify the item
 to update.
#       -e update expression  -- An expression that defines one or more
 attributes to be updated.
#       -v values  -- Path to json file containing the update values.
#
#  Returns:
#       0 - If successful.
#       1 - If it fails.
##############################################################################
function dynamodb_update_item() {
  local table_name keys update_expression values response
  local option OPTARG # Required to use getopts command in a function.
```

```
 ######################################
 # Function usage explanation
 ######################################
 function usage() {
   echo "function dynamodb_update_item"
   echo "Update an item in a DynamoDB table."
   echo " -n table_name  -- The name of the table."
   echo " -k keys  -- Path to json file containing the keys that identify the
item to update."
   echo " -e update expression  -- An expression that defines one or more
attributes to be updated."
   echo " -v values  -- Path to json file containing the update values."
   echo ""
 }

 while getopts "n:k:e:v:h" option; do
   case "${option}" in
     n) table_name="${OPTARG}" ;;
     k) keys="${OPTARG}" ;;
     e) update_expression="${OPTARG}" ;;
     v) values="${OPTARG}" ;;
     h)
       usage
       return 0
       ;;
     \?)
       echo "Invalid parameter"
       usage
       return 1
       ;;
   esac
 done
 export OPTIND=1

 if [[ -z "$table_name" ]]; then
   errecho "ERROR: You must provide a table name with the -n parameter."
   usage
   return 1
 fi

 if [[ -z "$keys" ]]; then
   errecho "ERROR: You must provide a keys json file path the -k parameter."
   usage
   return 1
```

```
  fi
  if [[ -z "$update_expression" ]]; then
    errecho "ERROR: You must provide an update expression with the -e parameter."
    usage
    return 1
  fi

  if [[ -z "$values" ]]; then
    errecho "ERROR: You must provide a values json file path the -v parameter."
    usage
    return 1
  fi

  iecho "Parameters:\n"
  iecho "    table_name:   $table_name"
  iecho "    keys:   $keys"
  iecho "    update_expression:   $update_expression"
  iecho "    values:   $values"

  response=$(aws dynamodb update-item \
    --table-name "$table_name" \
    --key file://"$keys" \
    --update-expression "$update_expression" \
    --expression-attribute-values file://"$values")

  local error_code=${?}

  if [[ $error_code -ne 0 ]]; then
    aws_cli_error_log $error_code
    errecho "ERROR: AWS reports update-item operation failed.$response"
    return 1
  fi

  return 0

}
```

この例で使用されているユーティリティ関数。

```
###############################################################################
# function iecho
#
```

```
# This function enables the script to display the specified text only if
# the global variable $VERBOSE is set to true.
#############################################################################
function iecho() {
  if [[ $VERBOSE == true ]]; then
    echo "$@"
  fi
}

#############################################################################
# function errecho
#
# This function outputs everything sent to it to STDERR (standard error output).
#############################################################################
function errecho() {
  printf "%s\n" "$*" 1>&2
}

#############################################################################
# function aws_cli_error_log()
#
# This function is used to log the error messages from the AWS CLI.
#
# See https://docs.aws.amazon.com/cli/latest/topic/return-codes.html#cli-aws-
help-return-codes.
#
# The function expects the following argument:
#         $1 - The error code returned by the AWS CLI.
#
#  Returns:
#          0: - Success.
#
#############################################################################
function aws_cli_error_log() {
  local err_code=$1
  errecho "Error code : $err_code"
  if [ "$err_code" == 1 ]; then
    errecho "  One or more S3 transfers failed."
  elif [ "$err_code" == 2 ]; then
    errecho "  Command line failed to parse."
  elif [ "$err_code" == 130 ]; then
    errecho "  Process received SIGINT."
  elif [ "$err_code" == 252 ]; then
    errecho "  Command syntax invalid."
```

```
  elif [ "$err_code" == 253 ]; then
    errecho "  The system environment or configuration was invalid."
  elif [ "$err_code" == 254 ]; then
    errecho "  The service returned an error."
  elif [ "$err_code" == 255 ]; then
    errecho "  255 is a catch-all error."
  fi

  return 0
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateItem」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Update an Amazon DynamoDB table item.
/*!
  \sa updateItem()
  \param tableName: The table name.
  \param partitionKey: The partition key.
  \param partitionValue: The value for the partition key.
  \param attributeKey: The key for the attribute to be updated.
  \param attributeValue: The value for the attribute to be updated.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
  */

/*
 *  The example code only sets/updates an attribute value. It processes
 *  the attribute value as a string, even if the value could be interpreted
 *  as a number. Also, the example code does not remove an existing attribute
```

```
 *  from the key value.
 */

bool AwsDoc::DynamoDB::updateItem(const Aws::String &tableName,
                                  const Aws::String &partitionKey,
                                  const Aws::String &partitionValue,
                                  const Aws::String &attributeKey,
                                  const Aws::String &attributeValue,
                                  const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // *** Define UpdateItem request arguments.
    // Define TableName argument.
    Aws::DynamoDB::Model::UpdateItemRequest request;
    request.SetTableName(tableName);

    // Define KeyName argument.
    Aws::DynamoDB::Model::AttributeValue attribValue;
    attribValue.SetS(partitionValue);
    request.AddKey(partitionKey, attribValue);

    // Construct the SET update expression argument.
    Aws::String update_expression("SET #a = :valueA");
    request.SetUpdateExpression(update_expression);

    // Construct attribute name argument.
    Aws::Map<Aws::String, Aws::String> expressionAttributeNames;
    expressionAttributeNames["#a"] = attributeKey;
    request.SetExpressionAttributeNames(expressionAttributeNames);

    // Construct attribute value argument.
    Aws::DynamoDB::Model::AttributeValue attributeUpdatedValue;
    attributeUpdatedValue.SetS(attributeValue);
    Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
 expressionAttributeValues;
    expressionAttributeValues[":valueA"] = attributeUpdatedValue;
    request.SetExpressionAttributeValues(expressionAttributeValues);

    // Update the item.
    const Aws::DynamoDB::Model::UpdateItemOutcome &outcome =
 dynamoClient.UpdateItem(
            request);
    if (outcome.IsSuccess()) {
```

```
        std::cout << "Item was updated" << std::endl;
    } else {
        std::cerr << outcome.GetError().GetMessage() << std::endl;
        return false;
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
```

```
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「UpdateItem」を参照してください。

CLI

AWS CLI

例 1: テーブル内の項目を更新するには

次の update-item の例では、MusicCollection テーブルの項目を更新します。新しい属性 (Year) を追加して、AlbumTitle 属性を更新します。更新後に表示される項目内の属性はすべて、レスポンスで返されます。

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --return-values ALL_NEW \
    --return-consumed-capacity TOTAL \
    --return-item-collection-metrics SIZE
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

expression-attribute-names.json の内容:

```
{
    "#Y":"Year", "#AT":"AlbumTitle"
}
```

expression-attribute-values.json の内容:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
}
```

出力:

```
{
    "Attributes": {
        "AlbumTitle": {
            "S": "Louder Than Ever"
        },
        "Awards": {
            "N": "10"
        },
        "Artist": {
            "S": "Acme Band"
        },
        "Year": {
            "N": "2015"
        },
        "SongTitle": {
            "S": "Happy Day"
        }
    },
    "ConsumedCapacity": {
        "TableName": "MusicCollection",
        "CapacityUnits": 3.0
    },
    "ItemCollectionMetrics": {
        "ItemCollectionKey": {
            "Artist": {
                "S": "Acme Band"
            }
        },
```

```
        "SizeEstimateRangeGB": [
            0.0,
            1.0
        ]
    }
}
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

例 2: 項目を条件付きで更新するには

次の例は、既存の項目に Year 属性がない場合にのみ、MusicCollection テーブル内の項目を更新します。

```
aws dynamodb update-item \
    --table-name MusicCollection \
    --key file://key.json \
    --update-expression "SET #Y = :y, #AT = :t" \
    --expression-attribute-names file://expression-attribute-names.json \
    --expression-attribute-values file://expression-attribute-values.json  \
    --condition-expression "attribute_not_exists(#Y)"
```

key.json の内容:

```
{
    "Artist": {"S": "Acme Band"},
    "SongTitle": {"S": "Happy Day"}
}
```

expression-attribute-names.json の内容:

```
{
    "#Y":"Year",
    "#AT":"AlbumTitle"
}
```

expression-attribute-values.json の内容:

```
{
    ":y":{"N": "2015"},
    ":t":{"S": "Louder Than Ever"}
```

```
}
```

項目にすでに Year 属性がある場合、DynamoDB は次の出力を返します。

```
An error occurred (ConditionalCheckFailedException) when calling the UpdateItem
 operation: The conditional request failed
```

詳細については、「Amazon DynamoDB ディベロッパーガイド」の「項目を書き込みます」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateItem」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import (
 "context"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// TableBasics encapsulates the Amazon DynamoDB service actions used in the
 examples.
// It contains a DynamoDB service client that is used to act on the specified
 table.
```

```
type TableBasics struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// UpdateMovie updates the rating and plot of a movie that already exists in the
// DynamoDB table. This function uses the `expression` package to build the
 update
// expression.
func (basics TableBasics) UpdateMovie(ctx context.Context, movie Movie)
 (map[string]map[string]interface{}, error) {
 var err error
 var response *dynamodb.UpdateItemOutput
 var attributeMap map[string]map[string]interface{}
 update := expression.Set(expression.Name("info.rating"),
 expression.Value(movie.Info["rating"]))
 update.Set(expression.Name("info.plot"), expression.Value(movie.Info["plot"]))
 expr, err := expression.NewBuilder().WithUpdate(update).Build()
 if err != nil {
  log.Printf("Couldn't build expression for update. Here's why: %v\n", err)
 } else {
  response, err = basics.DynamoDbClient.UpdateItem(ctx,
 &dynamodb.UpdateItemInput{
   TableName:                 aws.String(basics.TableName),
   Key:                       movie.GetKey(),
   ExpressionAttributeNames:  expr.Names(),
   ExpressionAttributeValues: expr.Values(),
   UpdateExpression:          expr.Update(),
   ReturnValues:              types.ReturnValueUpdatedNew,
  })
  if err != nil {
   log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
  } else {
   err = attributevalue.UnmarshalMap(response.Attributes, &attributeMap)
   if err != nil {
    log.Printf("Couldn't unmarshall update response. Here's why: %v\n", err)
   }
  }
 }
 return attributeMap, err
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}
```

```
// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「UpdateItem」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DynamoDbClient を使用してテーブルで項目を更新します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.AttributeAction;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.AttributeValueUpdate;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import java.util.HashMap;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
```

```
 * To update an Amazon DynamoDB table using the AWS SDK for Java V2, its better
 * practice to use the
 * Enhanced Client, See the EnhancedModifyItem example.
 */
public class UpdateItem {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableName> <key> <keyVal> <name> <updateVal>

                Where:
                    tableName - The Amazon DynamoDB table (for example, Music3).
                    key - The name of the key in the table (for example, Artist).
                    keyVal - The value of the key (for example, Famous Band).
                    name - The name of the column where the value is updated (for
 example, Awards).
                    updateVal - The value used to update an item (for example,
 14).
                 Example:
                    UpdateItem Music3 Artist Famous Band Awards 14
                """;

        if (args.length != 5) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableName = args[0];
        String key = args[1];
        String keyVal = args[2];
        String name = args[3];
        String updateVal = args[4];

        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();
        updateTableItem(ddb, tableName, key, keyVal, name, updateVal);
        ddb.close();
    }

    public static void updateTableItem(DynamoDbClient ddb,
            String tableName,
```

```
            String key,
            String keyVal,
            String name,
            String updateVal) {

        HashMap<String, AttributeValue> itemKey = new HashMap<>();
        itemKey.put(key, AttributeValue.builder()
                .s(keyVal)
                .build());

        HashMap<String, AttributeValueUpdate> updatedValues = new HashMap<>();
        updatedValues.put(name, AttributeValueUpdate.builder()
                .value(AttributeValue.builder().s(updateVal).build())
                .action(AttributeAction.PUT)
                .build());

        UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(itemKey)
                .attributeUpdates(updatedValues)
                .build();

        try {
            ddb.updateItem(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("The Amazon DynamoDB table was updated!");
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

この例では、ドキュメントクライアントを使用して DynamoDB での項目の操作を簡略化しています。API の詳細については、「UpdateCommand」を参照してください。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun updateTableItem(
    tableNameVal: String,
    keyName: String,
    keyVal: String,
    name: String,
    updateVal: String,
) {
    val itemKey = mutableMapOf<String, AttributeValue>()
    itemKey[keyName] = AttributeValue.S(keyVal)

    val updatedValues = mutableMapOf<String, AttributeValueUpdate>()
    updatedValues[name] =
        AttributeValueUpdate {
            value = AttributeValue.S(updateVal)
            action = AttributeAction.Put
        }

    val request =
        UpdateItemRequest {
            tableName = tableNameVal
            key = itemKey
            attributeUpdates = updatedValues
        }

    DynamoDbClient.fromEnvironment { region = "us-east-1" }.use { ddb ->
        ddb.updateItem(request)
        println("Item in $tableNameVal was updated")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「UpdateItem」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        echo "What rating would you like to give {$movie['Item']['title']['S']}?
\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $service->updateItemAttributeByKey($tableName, $key, 'rating', 'N',
 $rating);

    public function updateItemAttributeByKey(
        string $tableName,
        array $key,
        string $attributeName,
        string $attributeType,
        string $newValue
    ) {
        $this->dynamoDbClient->updateItem([
            'Key' => $key['Item'],
            'TableName' => $tableName,
            'UpdateExpression' => "set #NV=:NV",
            'ExpressionAttributeNames' => [
                '#NV' => $attributeName,
            ],
            'ExpressionAttributeValues' => [
                ':NV' => [
                    $attributeType => $newValue
                ]
```

```
            ],
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「UpdateItem」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目のジャンル属性を「Rap」に設定します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Rap
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「UpdateItem」を参照してください。

Tools for PowerShell V5

例 1: パーティションキー SongTitle とソートキー Artist を含む DynamoDB 項目のジャンル属性を「Rap」に設定します。

```
$key = @{
    SongTitle = 'Somewhere Down The Road'
    Artist = 'No One You Know'
} | ConvertTo-DDBItem

$updateDdbItem = @{
    TableName = 'Music'
    Key = $key
    UpdateExpression = 'set Genre = :val1'
    ExpressionAttributeValue = (@{
        ':val1' = ([Amazon.DynamoDBv2.Model.AttributeValue]'Rap')
    })
}
Update-DDBItem @updateDdbItem
```

出力する。

```
Name                           Value
----                           -----
Genre                          Rap
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

更新式を使用して項目を更新します。

```
class Movies:
    """Encapsulates an Amazon DynamoDB table of movie data.

    Example data structure for a movie record in this table:
        {
            "year": 1999,
            "title": "For Love of the Game",
            "info": {
                "directors": ["Sam Raimi"],
                "release_date": "1999-09-15T00:00:00Z",
                "rating": 6.3,
                "plot": "A washed up pitcher flashes through his career.",
                "rank": 4987,
                "running_time_secs": 8220,
                "actors": [
                    "Kevin Costner",
                    "Kelly Preston",
                    "John C. Reilly"
                ]
            }
        }
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource
        # The table variable is set during the scenario in the call to
        # 'exists' if the table exists. Otherwise, it is set by 'create_table'.
        self.table = None


    def update_movie(self, title, year, rating, plot):
        """
        Updates rating and plot data for a movie in the table.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating: The updated rating to the give the movie.
        :param plot: The updated plot summary to give the movie.
        :return: The fields that were updated, with their new values.
        """
```

```
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating=:r, info.plot=:p",
                ExpressionAttributeValues={":r": Decimal(str(rating)), ":p":
 plot},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

算術演算を含む更新式を使用して、項目を更新します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def update_rating(self, title, year, rating_change):
        """
        Updates the quality rating of a movie in the table by using an arithmetic
        operation in the update expression. By specifying an arithmetic
 operation,
        you can adjust a value in a single request, rather than first getting its
        value and then setting its new value.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param rating_change: The amount to add to the current rating for the
 movie.
        :return: The updated rating.
        """
```

```
        try:
            response = self.table.update_item(
                Key={"year": year, "title": title},
                UpdateExpression="set info.rating = info.rating + :val",
                ExpressionAttributeValues={":val": Decimal(str(rating_change))},
                ReturnValues="UPDATED_NEW",
            )
        except ClientError as err:
            logger.error(
                "Couldn't update movie %s in table %s. Here's why: %s: %s",
                title,
                self.table.name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response["Attributes"]
```

特定の条件を満たす場合にのみ、項目を更新します。

```
class UpdateQueryWrapper:
    def __init__(self, table):
        self.table = table


    def remove_actors(self, title, year, actor_threshold):
        """
        Removes an actor from a movie, but only when the number of actors is
 greater
        than a specified threshold. If the movie does not list more than the
 threshold,
        no actors are removed.

        :param title: The title of the movie to update.
        :param year: The release year of the movie to update.
        :param actor_threshold: The threshold of actors to check.
        :return: The movie data after the update.
        """
        try:
            response = self.table.update_item(
```

```
                Key={"year": year, "title": title},
                UpdateExpression="remove info.actors[0]",
                ConditionExpression="size(info.actors) > :num",
                ExpressionAttributeValues={":num": actor_threshold},
                ReturnValues="ALL_NEW",
            )
        except ClientError as err:
            if err.response["Error"]["Code"] ==
"ConditionalCheckFailedException":
                logger.warning(
                    "Didn't update %s because it has fewer than %s actors.",
                    title,
                    actor_threshold + 1,
                )
            else:
                logger.error(
                    "Couldn't update movie %s. Here's why: %s: %s",
                    title,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return response["Attributes"]
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class DynamoDBBasics
```

```
  attr_reader :dynamo_resource, :table

  def initialize(table_name)
    client = Aws::DynamoDB::Client.new(region: 'us-east-1')
    @dynamo_resource = Aws::DynamoDB::Resource.new(client: client)
    @table = @dynamo_resource.table(table_name)
  end

  # Updates rating and plot data for a movie in the table.
  #
  # @param movie [Hash] The title, year, plot, rating of the movie.
  def update_item(movie)
    response = @table.update_item(
      key: { 'year' => movie[:year], 'title' => movie[:title] },
      update_expression: 'set info.rating=:r',
      expression_attribute_values: { ':r' => movie[:rating] },
      return_values: 'UPDATED_NEW'
    )
  rescue Aws::DynamoDB::Errors::ServiceError => e
    puts("Couldn't update movie #{movie[:title]} (#{movie[:year]}) in table
#{@table.name}\n")
    puts("\t#{e.code}: #{e.message}")
    raise
  else
    response.attributes
  end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「UpdateItem」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_output = lo_dyn->updateitem(
          iv_tablename        = iv_table_name
          it_key              = it_item_key
          it_attributeupdates = it_attribute_updates ).
        MESSAGE '1 item updated in DynamoDB Table' && iv_table_name TYPE 'I'.
      CATCH /aws1/cx_dyncondalcheckfaile00.
        MESSAGE 'A condition specified in the operation could not be evaluated.'
TYPE 'E'.
      CATCH /aws1/cx_dynresourcenotfoundex.
        MESSAGE 'The table or index does not exist' TYPE 'E'.
      CATCH /aws1/cx_dyntransactconflictex.
        MESSAGE 'Another transaction is using the item' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「UpdateItem」を参照してください。

Swift

SDK for Swift

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import AWSDynamoDB


    /// Update the specified movie with new `rating` and `plot` information.
    ///
    /// - Parameters:
    ///   - title: The title of the movie to update.
    ///   - year: The release year of the movie to update.
    ///   - rating: The new rating for the movie.
    ///   - plot: The new plot summary string for the movie.
    ///
    /// - Returns: An array of mappings of attribute names to their new
```

```
    ///   listing each item actually changed. Items that didn't need to change
    ///   aren't included in this list. `nil` if no changes were made.
    ///
    func update(title: String, year: Int, rating: Double? = nil, plot: String? =
nil) async throws
        -> [Swift.String: DynamoDBClientTypes.AttributeValue]?
    {
        do {
            guard let client = self.ddbClient else {
                throw MoviesError.UninitializedClient
            }

            // Build the update expression and the list of expression attribute
            // values. Include only the information that's changed.

            var expressionParts: [String] = []
            var attrValues: [Swift.String: DynamoDBClientTypes.AttributeValue] =
[:]

            if rating != nil {
                expressionParts.append("info.rating=:r")
                attrValues[":r"] = .n(String(rating!))
            }
            if plot != nil {
                expressionParts.append("info.plot=:p")
                attrValues[":p"] = .s(plot!)
            }
            let expression = "set \(expressionParts.joined(separator: ", "))"

            let input = UpdateItemInput(
                // Create substitution tokens for the attribute values, to ensure
                // no conflicts in expression syntax.
                expressionAttributeValues: attrValues,
                // The key identifying the movie to update consists of the
release
                // year and title.
                key: [
                    "year": .n(String(year)),
                    "title": .s(title)
                ],
                returnValues: .updatedNew,
                tableName: self.tableName,
                updateExpression: expression
            )
```

```
            let output = try await client.updateItem(input: input)

            guard let attributes: [Swift.String:
DynamoDBClientTypes.AttributeValue] = output.attributes else {
                throw MoviesError.InvalidAttributes
            }
            return attributes
        } catch {
            print("ERROR: update:", dump(error))
            throw error
        }
    }
```

- API の詳細については、「AWS SDK for Swift API リファレンス」の「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateTable** を使用する

次のサンプルコードは、UpdateTable を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- MREC を示すグローバルテーブルの作成と管理
- MRSC グローバルテーブルの作成と管理
- グローバルセカンダリインデックスを管理する
- テーブルのウォームスループット設定を更新する
- グローバルテーブルとマルチリージョンレプリケーションの最終的な一貫性 (MREC) を操作する
- テーブルの暗号化を操作する

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Update a DynamoDB table.
/*!
  \sa updateTable()
  \param tableName: Name for the DynamoDB table.
  \param readCapacity: Provisioned read capacity.
  \param writeCapacity: Provisioned write capacity.
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::updateTable(const Aws::String &tableName,
                                   long long readCapacity, long long
 writeCapacity,
                                   const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::cout << "Updating " << tableName << " with new provisioned throughput
 values"
              << std::endl;
    std::cout << "Read capacity : " << readCapacity << std::endl;
    std::cout << "Write capacity: " << writeCapacity << std::endl;

    Aws::DynamoDB::Model::UpdateTableRequest request;
    Aws::DynamoDB::Model::ProvisionedThroughput provisionedThroughput;

 provisionedThroughput.WithReadCapacityUnits(readCapacity).WithWriteCapacityUnits(
           writeCapacity);

 request.WithProvisionedThroughput(provisionedThroughput).WithTableName(tableName);

    const Aws::DynamoDB::Model::UpdateTableOutcome &outcome =
 dynamoClient.UpdateTable(
```

```
            request);
    if (outcome.IsSuccess()) {
        std::cout << "Successfully updated the table." << std::endl;
    } else {
        const Aws::DynamoDB::DynamoDBError &error = outcome.GetError();
        if (error.GetErrorType() == Aws::DynamoDB::DynamoDBErrors::VALIDATION &&
            error.GetMessage().find("The provisioned throughput for the table
 will not change") != std::string::npos) {
            std::cout << "The provisioned throughput for the table will not
 change." << std::endl;
        } else {
            std::cerr << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    return waitTableActive(tableName, dynamoClient);
}
```

テーブルがアクティブになるまで待機するコード。

```
//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
```

```
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「UpdateTable」を参照してください。

CLI

AWS CLI

例 1: テーブルの請求モードを変更するには

次の update-table 例は、MusicCollection テーブルにプロビジョニングされた読み取り/書き込みキャパシティを増やします。

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --billing-mode PROVISIONED \
    --provisioned-throughput ReadCapacityUnits=15,WriteCapacityUnits=10
```

出力:

```
{
```

```
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "UPDATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T13:18:18.921000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        }
```

```
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「テーブルの更新」を参照してください。

例 2: グローバルセカンダリインデックスを作成するには

次の例は、MusicCollection テーブルにグローバルセカンダリインデックスを追加します。

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=AlbumTitle,AttributeType=S \
    --global-secondary-index-updates file://gsi-updates.json
```

gsi-updates.json の内容:

```
[
    {
        "Create": {
            "IndexName": "AlbumTitle-index",
            "KeySchema": [
                {
                    "AttributeName": "AlbumTitle",
                    "KeyType": "HASH"
                }
            ],
            "ProvisionedThroughput": {
                "ReadCapacityUnits": 10,
                "WriteCapacityUnits": 10
            },
            "Projection": {
                "ProjectionType": "ALL"
            }
        }
    }
]
```

出力:

```
{
```

```
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "UPDATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
```

```
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "CREATING",
                "Backfilling": false,
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ]
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「テーブルの更新」を参照してください。

例 3: テーブルで DynamoDB Streams を有効化するには

次のコマンドは、`MusicCollection` テーブルで DynamoDB Streams を有効化します。

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_IMAGE
```

出力:

```
{
    "TableDescription": {
```

```
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "UPDATING",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
        "LocalSecondaryIndexes": [
```

```
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Year",
                        "Genre"
                    ]
                },
                "IndexSizeBytes": 139,
                "ItemCount": 2,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ],
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
```

```
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ],
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_IMAGE"
        },
        "LatestStreamLabel": "2020-07-28T21:53:39.112",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/stream/2020-07-28T21:53:39.112"
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「テーブルの更新」を参照してください。

例 4: サーバー側の暗号化を有効化するには

次の例は、MusicCollection テーブルでサーバー側の暗号化を有効します。

```
aws dynamodb update-table \
    --table-name MusicCollection \
    --sse-specification Enabled=true,SSEType=KMS
```

出力:

```
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
```

```
        ],
        "TableName": "MusicCollection",
        "KeySchema": [
            {
                "AttributeName": "Artist",
                "KeyType": "HASH"
            },
            {
                "AttributeName": "SongTitle",
                "KeyType": "RANGE"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": "2020-05-26T15:59:49.473000-07:00",
        "ProvisionedThroughput": {
            "LastIncreaseDateTime": "2020-07-28T12:59:17.537000-07:00",
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 15,
            "WriteCapacityUnits": 10
        },
        "TableSizeBytes": 182,
        "ItemCount": 2,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection",
        "TableId": "abcd0123-01ab-23cd-0123-abcdef123456",
        "BillingModeSummary": {
            "BillingMode": "PROVISIONED",
            "LastUpdateToPayPerRequestDateTime":
 "2020-07-28T13:14:48.366000-07:00"
        },
        "LocalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitleIndex",
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    },
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "RANGE"
                    }
                ],
                "Projection": {
```

```
                    "ProjectionType": "INCLUDE",
                    "NonKeyAttributes": [
                        "Year",
                        "Genre"
                    ]
                },
                "IndexSizeBytes": 139,
                "ItemCount": 2,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitleIndex"
            }
        ],
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "AlbumTitle-index",
                "KeySchema": [
                    {
                        "AttributeName": "AlbumTitle",
                        "KeyType": "HASH"
                    }
                ],
                "Projection": {
                    "ProjectionType": "ALL"
                },
                "IndexStatus": "ACTIVE",
                "ProvisionedThroughput": {
                    "NumberOfDecreasesToday": 0,
                    "ReadCapacityUnits": 10,
                    "WriteCapacityUnits": 10
                },
                "IndexSizeBytes": 0,
                "ItemCount": 0,
                "IndexArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/index/AlbumTitle-index"
            }
        ],
        "StreamSpecification": {
            "StreamEnabled": true,
            "StreamViewType": "NEW_IMAGE"
        },
        "LatestStreamLabel": "2020-07-28T21:53:39.112",
        "LatestStreamArn": "arn:aws:dynamodb:us-west-2:123456789012:table/
MusicCollection/stream/2020-07-28T21:53:39.112",
        "SSEDescription": {
```

```
            "Status": "UPDATING"
        }
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「テーブルの更新」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateTable」を参照してください。

PowerShell

Tools for PowerShell V4

例 1: 指定されたテーブルのプロビジョンされたスループットを更新します。

```
Update-DDBTable -TableName "myTable" -ReadCapacity 10 -WriteCapacity 5
```

- API の詳細については、AWS Tools for PowerShell コマンドレットリファレンス (V4) の「UpdateTable」を参照してください。

Tools for PowerShell V5

例 1: 指定されたテーブルのプロビジョンされたスループットを更新します。

```
Update-DDBTable -TableName "myTable" -ReadCapacity 10 -WriteCapacity 5
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス (V5)」の「UpdateTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateTimeToLive** を使用する

次のサンプルコードは、UpdateTimeToLive を使用する方法を説明しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- Streams と Time-to-Live を操作する

CLI

AWS CLI

テーブルの Time to Live 設定を更新するには

次の update-time-to-live 例は、指定したテーブルの Time to Live を有効にします。

```
aws dynamodb update-time-to-live \
    --table-name MusicCollection \
    --time-to-live-specification Enabled=true,AttributeName=ttl
```

出力:

```
{
    "TimeToLiveSpecification": {
        "Enabled": true,
        "AttributeName": "ttl"
    }
}
```

詳細については、「Amazon DynamoDB デベロッパーガイド」の「Time to Live (TTL)」を参照してください。

- API の詳細については、「AWS CLI Command Reference」の「UpdateTimeToLive」を参照してください。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して既存の DynamoDB テーブルの TTL を有効にします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.TimeToLiveSpecification;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveRequest;
```

```
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveResponse;

import java.util.logging.Level;
import java.util.logging.Logger;

    public UpdateTimeToLiveResponse enableTTL(final String tableName, final
 String attributeName, final Region region) {
        final TimeToLiveSpecification ttlSpec = TimeToLiveSpecification.builder()
            .attributeName(attributeName)
            .enabled(true)
            .build();

        final UpdateTimeToLiveRequest request = UpdateTimeToLiveRequest.builder()
            .tableName(tableName)
            .timeToLiveSpecification(ttlSpec)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.updateTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して既存の DynamoDB テーブルの TTL を無効にします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.TimeToLiveSpecification;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateTimeToLiveResponse;

import java.util.logging.Level;
import java.util.logging.Logger;
```

```
    public UpdateTimeToLiveResponse disableTTL(
        final String tableName, final String attributeName, final Region region)
 {
        final TimeToLiveSpecification ttlSpec = TimeToLiveSpecification.builder()
            .attributeName(attributeName)
            .enabled(false)
            .build();

        final UpdateTimeToLiveRequest request = UpdateTimeToLiveRequest.builder()
            .tableName(tableName)
            .timeToLiveSpecification(ttlSpec)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            return ddb.updateTimeToLive(request);
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateTimeToLive」を参照してください。

JavaScript

SDK for JavaScript (v3)

既存の DynamoDB テーブルの TTL を有効にします。

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const enableTTL = async (tableName, ttlAttribute, region = 'us-east-1') =>
 {
```

```
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: true,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL enabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to enable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error enabling TTL: ${e}`);
        throw e;
    }
};

// Example usage (commented out for testing)
// enableTTL('ExampleTable', 'exampleTtlAttribute');
```

既存の DynamoDB テーブルの TTL を無効にします。

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-
dynamodb";

export const disableTTL = async (tableName, ttlAttribute, region = 'us-east-1')
 => {

    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
```

```
    });

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: false,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL disabled successfully for table ${tableName}, using
 attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to disable TTL for table ${tableName}, response
 object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error disabling TTL: ${e}`);
        throw e;
    }
};

// Example usage (commented out for testing)
// disableTTL('ExampleTable', 'exampleTtlAttribute');
```

- API の詳細については、「AWS SDK for JavaScript API Reference」の「UpdateTimeToLive」を参照してください。

Python

SDK for Python (Boto3)

既存の DynamoDB テーブルの TTL を有効にします。

```
import boto3


def enable_ttl(table_name, ttl_attribute_name):
    """
```

```
    Enables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table
    :param ttl_attribute_name: The name of the TTL attribute being provided to
 the table.
    """
    try:
        dynamodb = boto3.client("dynamodb")

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={"Enabled": True, "AttributeName":
 ttl_attribute_name},
        )

        # In the returned response, check for a successful status code.
        if response["ResponseMetadata"]["HTTPStatusCode"] == 200:
            print("TTL has been enabled successfully.")
        else:
            print(
                f"Failed to enable TTL, status code {response['ResponseMetadata']
['HTTPStatusCode']}"
            )
        return response
    except Exception as ex:
        print("Couldn't enable TTL in table %s. Here's why: %s" % (table_name,
 ex))
        raise


# your values
enable_ttl("your-table-name", "expireAt")
```

既存の DynamoDB テーブルの TTL を無効にします。

```
import boto3


def disable_ttl(table_name, ttl_attribute_name):
```

```
    """
    Disables TTL on DynamoDB table for a given attribute name
        on success, returns a status code of 200
        on error, throws an exception

    :param table_name: Name of the DynamoDB table being modified
    :param ttl_attribute_name: The name of the TTL attribute being provided to
 the table.
    """
    try:
        dynamodb = boto3.client("dynamodb")

        # Enable TTL on an existing DynamoDB table
        response = dynamodb.update_time_to_live(
            TableName=table_name,
            TimeToLiveSpecification={"Enabled": False, "AttributeName":
 ttl_attribute_name},
        )

        # In the returned response, check for a successful status code.
        if response["ResponseMetadata"]["HTTPStatusCode"] == 200:
            print("TTL has been disabled successfully.")
        else:
            print(
                f"Failed to disable TTL, status code
 {response['ResponseMetadata']['HTTPStatusCode']}"
            )
    except Exception as ex:
        print("Couldn't disable TTL in table %s. Here's why: %s" % (table_name,
 ex))
        raise


# your values
disable_ttl("your-table-name", "expireAt")
```

- API の詳細については、「AWS SDK for Python (Boto3) API Reference」の「UpdateTimeToLive」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した DynamoDB のシナリオ

次のコード例は、AWS SDK を使用して DynamoDB で一般的なシナリオを実装する方法を示しています。これらのシナリオでは、DynamoDB 内で複数の関数を呼び出すか、他の AWS のサービスと組み合わせることで、特定のタスクを達成する方法を示します。各シナリオには、完全なソースコードへのリンクが含まれており、そこからコードの設定方法と実行方法に関する手順を確認できます。

シナリオは、サービスアクションをコンテキストで理解するのに役立つ中級レベルの経験を対象としています。

例

- AWS SDK を使用して DAX で DynamoDB の読み取りを高速化する
- AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスの高度なシナリオを操作する
- DynamoDB テーブルにデータを送信するアプリケーションを構築する
- AWS SDK を使用して、DynamoDB で複数の値を 1 つの属性と比較する
- AWS SDK を使用して TTL を含む DynamoDB 項目を条件付きで更新する
- AWS SDK を使用してローカル DynamoDB インスタンスに接続する
- AWS SDKを使用した DynamoDB のカウント式演算子
- COVID-19 データを追跡する API Gateway REST API を作成する
- ステップ関数でメッセンジャーアプリケーションを作成する
- ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成
- AWS SDK を使用してグローバルセカンダリインデックスを含む DynamoDB テーブルを作成する
- AWS SDK を使用してウォームスループット設定で DynamoDB テーブルを作成する
- DynamoDB データを追跡するウェブアプリケーションを作成する
- API Gateway で WebSocket チャットアプリケーションを作成する
- AWS SDK を使用して TTL を含む DynamoDB 項目を作成する
- AWS SDK を使用して、DynamoDB でマルチリージョンの強力な整合性を備えたグローバルテーブルを作成して管理する
- AWS SDK を使用した MREC を示す DynamoDB グローバルテーブルの作成と管理

- AWS SDK で PartiQL DELETE ステートメントを使用して DynamoDB データを削除する
- AWS SDK を使用して、Amazon Rekognition で画像内の PPE を検出する
- NoSQL データベースの開始方法
- AWS SDK で PartiQL INSERT ステートメントを使用して DynamoDB データを挿入する
- ブラウザからの Lambda 関数の呼び出し
- AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスを管理する
- AWS Command Line Interface v2 を使用して DynamoDB リソースベースのポリシーを管理する
- AWS SDK を使用した Amazon DynamoDB のパフォーマンスのモニタリング
- AWS SDK を使用して高度な DynamoDB クエリオペレーションを実行する
- AWS SDK を使用して DynamoDB でリストオペレーションを実行する
- AWS SDK を使用して DynamoDB でマップオペレーションを実行する
- AWS SDK を使用して DynamoDB でセットオペレーションを実行する
- PartiQL ステートメントのバッチと AWS SDK を使用して DynamoDB テーブルにクエリを実行する
- PartiQL と AWS SDK を使用して DynamoDB テーブルに対してクエリを実行する
- AWS SDK でグローバルセカンダリインデックスを使用して DynamoDB テーブルをクエリする
- AWS SDK で begins_with 条件を使用して DynamoDB テーブルをクエリする
- AWS SDK でソートキーの日付範囲を使用して DynamoDB テーブルをクエリする
- AWS SDK で複雑なフィルター式を使用して DynamoDB テーブルをクエリする
- AWS SDK で動的フィルター式を使用して DynamoDB テーブルをクエリする
- AWS SDK でフィルター式と制限を使用して DynamoDB テーブルをクエリする
- AWS SDK でネストされた属性を使用して DynamoDB テーブルをクエリする
- AWS SDK でページ分割を使用して DynamoDB テーブルをクエリする
- AWS SDK で強力な整合性のある読み込みを使用して DynamoDB テーブルをクエリする
- AWS SDK で PartiQL SELECT ステートメントを使用して DynamoDB データをクエリする
- AWS SDK を使用して、DynamoDB テーブルの TTL 項目に対してクエリを実行する
- AWS SDK で日付と時刻のパターンを使用して DynamoDB テーブルをクエリする
- AWS SDK を使用して、EXIF とその他の画像情報を保存する
- AWS Command Line Interface v2 を使用して DynamoDB の属性ベースのアクセス制御を設定する
- AWS SDK を使用した DynamoDB の式順序の更新を理解する

- AWS SDK を使用してウォームスループットで DynamoDB テーブル設定を更新する
- AWS SDK を使用して TTL を含む DynamoDB 項目を更新する
- AWS SDK で PartiQL UPDATE ステートメントを使用して DynamoDB データを更新する
- API Gateway を使用して Lambda 関数を呼び出す
- Step Functions を使用して Lambda 関数を呼び出す
- AWS SDK を使用して DynamoDB のドキュメントモデルを使用する
- AWS SDK を使用して DynamoDB 用の高レベルのオブジェクト永続性モデルを使用する
- AWS SDK を使用して DynamoDB でアトミックカウンターオペレーションを使用する
- AWS SDK を使用して DynamoDB で条件付きオペレーションを使用する
- AWS SDK を使用して DynamoDB で式の属性名を使用する
- スケジュールされたイベントを使用した Lambda 関数の呼び出し
- AWS Command Line Interface v2 を使用して DynamoDB ローカルセカンダリインデックスを操作する
- AWS Command Line Interface v2 を使用して DynamoDB Streams と Time-to-Live を操作する
- AWS Command Line Interface v2 を使用して、DynamoDB グローバルテーブルと最終的な一貫性を備えたマルチリージョンレプリケーション (MREC) を操作する
- AWS Command Line Interface v2 を使用して DynamoDB リソースのタグ付けを操作する
- AWS Command Line Interface v2 を使用して DynamoDB テーブルの暗号化を操作する

AWS SDK を使用して DAX で DynamoDB の読み取りを高速化する

次のコード例は、以下の操作方法を示しています。

- DAX クライアントと SDK クライアントの両方でデータを作成してテーブルに書き込みます。
- 両方のクライアントでテーブルを取得、クエリ、スキャンし、パフォーマンスを比較します。

詳細については、「DynamoDB Accelerator クライアントで開発する」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

DAX または Boto3 クライアントでテーブルを作成します。

```
import boto3


def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table_name = "TryDaxTable"
    params = {
        "TableName": table_name,
        "KeySchema": [
            {"AttributeName": "partition_key", "KeyType": "HASH"},
            {"AttributeName": "sort_key", "KeyType": "RANGE"},
        ],
        "AttributeDefinitions": [
            {"AttributeName": "partition_key", "AttributeType": "N"},
            {"AttributeName": "sort_key", "AttributeType": "N"},
        ],
        "BillingMode": "PAY_PER_REQUEST",
    }
    table = dyn_resource.create_table(**params)
    print(f"Creating {table_name}...")
    table.wait_until_exists()
    return table
```

```
if __name__ == "__main__":
    dax_table = create_dax_table()
    print(f"Created table.")
```

テーブルにテストデータを書き込みます。

```
import boto3


def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate
 the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    some_data = "X" * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(
                Item={
                    "partition_key": partition_key,
                    "sort_key": sort_key,
                    "some_data": some_data,
                }
            )
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")


if __name__ == "__main__":
    write_key_count = 10
    write_item_size = 1000
    print(
```

```
        f"Writing {write_key_count*write_key_count} items to the table. "
        f"Each item is {write_item_size} characters."
    )
    write_data_to_dax_table(write_key_count, write_item_size)
```

DAX クライアントと Boto3 クライアントの両方について、いくつかのイテレーションの項目を取得し、それぞれに費やした時間を報告します。

```
import argparse
import sys
import time
import amazondax
import boto3


def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each
 iteration.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        for partition_key in range(1, key_count + 1):
            for sort_key in range(1, key_count + 1):
                table.get_item(
                    Key={"partition_key": partition_key, "sort_key": sort_key}
                )
                print(".", end="")
                sys.stdout.flush()
    print()
    end = time.perf_counter()
```

```
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
 used.",
    )
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the DAX client."
        )
        # Use a with statement so the DAX client closes the cluster after
 completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
 as dax:
            test_start, test_end = get_item_test(
                test_key_count, test_iterations, dyn_resource=dax
            )
    else:
        print(
            f"Getting each item from the table {test_iterations} times, "
            f"using the Boto3 client."
        )
        test_start, test_end = get_item_test(test_key_count, test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/ test_iterations}."
    )
```

DAX クライアントと Boto3 クライアントの両方について、いくつかのイテレーションのテーブルに対してクエリを実行し、それぞれに費やした時間を報告します。

```
import argparse
import time
import sys
import amazondax
import boto3
from boto3.dynamodb.conditions import Key


def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param partition_key: The partition key value to use in the query. The query
                          returns items that have partition keys equal to this
 value.
    :param sort_keys: The range of sort key values for the query. The query
 returns
                      items that have sort key values between these two values.
    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    key_condition_expression = Key("partition_key").eq(partition_key) & Key(
        "sort_key"
    ).between(*sort_keys)

    start = time.perf_counter()
    for _ in range(iterations):
        table.query(KeyConditionExpression=key_condition_expression)
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
```

```
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
used.",
    )
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX
client.")
        # Use a with statement so the DAX client closes the cluster after
completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations,
dyn_resource=dax
            )
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3
client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations
        )

    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

DAX クライアントと Boto3 クライアントの両方について、いくつかのイテレーションのテーブルをスキャンし、それぞれに費やした時間を報告します。

```
import argparse
import time
import sys
```

```
import amazondax
import boto3


def scan_test(iterations, dyn_resource=None):
    """
    Scans the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param iterations: The number of iterations to run.
    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The start and end times of the test.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    start = time.perf_counter()
    for _ in range(iterations):
        table.scan()
        print(".", end="")
        sys.stdout.flush()
    print()
    end = time.perf_counter()
    return start, end


if __name__ == "__main__":
    # pylint: disable=not-context-manager
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "endpoint_url",
        nargs="?",
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not
 used.",
    )
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
        print(f"Scanning the table {test_iterations} times, using the DAX
 client.")
```

```
        # Use a with statement so the DAX client closes the cluster after
 completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url)
 as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3
 client.")
        test_start, test_end = scan_test(test_iterations)
    print(
        f"Total time: {test_end - test_start:.4f} sec. Average time: "
        f"{(test_end - test_start)/test_iterations}."
    )
```

テーブルを削除する。

```
import boto3


def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource("dynamodb")

    table = dyn_resource.Table("TryDaxTable")
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()


if __name__ == "__main__":
    delete_dax_table()
    print("Table deleted!")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の以下のトピックを参照してください。

- CreateTable
- DeleteTable
- GetItem
- PutItem
- Query
- Scan

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスの高度なシナリオを操作する

次のコード例は、グローバルセカンダリインデックスの高度な設定の操作方法を示しています。

- 複数の GSI を含むテーブルを作成します。
- オンデマンドキャパシティと GSI を含むテーブルを作成します。
- 複数の GSI を含むテーブルに項目を配置します。
- 異なる条件で複数の GSI をクエリします。

Bash

AWS CLIBash スクリプトを使用した

複数の GSI を含むテーブルを作成します。

```
# Create a table with multiple GSIs
aws dynamodb create-table \
    --table-name MusicLibrary \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S \
        AttributeName=Genre,AttributeType=S \
        AttributeName=Year,AttributeType=N \
    --key-schema \
```

```
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"AlbumIndex\",
                \"KeySchema\": [{\"AttributeName\":\"AlbumTitle\",\"KeyType\":
\"HASH\"}],
                \"Projection\": {\"ProjectionType\":\"ALL\"}
            },
            {
                \"IndexName\": \"GenreYearIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"Genre\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"Year\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {\"ProjectionType\":\"INCLUDE\",
\"NonKeyAttributes\":[\"Artist\",\"SongTitle\"]}
            }
        ]"
```

オンデマンドキャパシティと GSI を含むテーブルを作成します。

```
# Create a table with on-demand capacity and GSI
aws dynamodb create-table \
    --table-name MusicOnDemand \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=Genre,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "[
            {
                \"IndexName\": \"GenreIndex\",
                \"KeySchema\": [{\"AttributeName\":\"Genre\",\"KeyType\":\"HASH
\"}],
                \"Projection\": {\"ProjectionType\":\"ALL\"}
```

```
            }
        ]"
```

複数の GSI を含むテーブルに項目を配置します。

```
# Add items to MusicLibrary table
aws dynamodb put-item \
    --table-name MusicLibrary \
    --item '{
        "Artist": {"S": "The Beatles"},
        "SongTitle": {"S": "Hey Jude"},
        "AlbumTitle": {"S": "Past Masters"},
        "Genre": {"S": "Rock"},
        "Year": {"N": "1968"}
    }'

aws dynamodb put-item \
    --table-name MusicLibrary \
    --item '{
        "Artist": {"S": "Miles Davis"},
        "SongTitle": {"S": "So What"},
        "AlbumTitle": {"S": "Kind of Blue"},
        "Genre": {"S": "Jazz"},
        "Year": {"N": "1959"}
    }'
```

複数の GSI を含むテーブルの項目をクエリします。

```
# Query the AlbumIndex GSI
echo "Querying AlbumIndex GSI:"
aws dynamodb query \
    --table-name MusicLibrary \
    --index-name AlbumIndex \
    --key-condition-expression "AlbumTitle = :album" \
    --expression-attribute-values '{":album":{"S":"Kind of Blue"}}'

# Query the GenreYearIndex GSI with a range condition
echo "Querying GenreYearIndex GSI with range condition:"
aws dynamodb query \
    --table-name MusicLibrary \
    --index-name GenreYearIndex \
```

```
    --key-condition-expression "Genre = :genre AND #yr > :year" \
    --expression-attribute-names '{"#yr": "Year"}' \
    --expression-attribute-values '{":genre":{"S":"Rock"},":year":{"N":"1965"}}'
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - PutItem
  - Query

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB テーブルにデータを送信するアプリケーションを構築する

次のコード例は、Amazon DynamoDB テーブルにデータを送信し、ユーザーがテーブルを更新したときに通知するアプリケーションを構築する方法を示しています。

Java

SDK for Java 2.x

Amazon DynamoDB Java API を使用してデータを送信し、Amazon Simple Notification Service Java API を使用してテキストメッセージを送信する動的ウェブアプリケーションを作成する方法について説明します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

この例では、ユーザーが Amazon DynamoDB テーブルにデータを送信し、Amazon Simple Notification Service (Amazon SNS) を使用して管理者にテキストメッセージを送信できるようにするアプリを構築する方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- DynamoDB
- Amazon SNS

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK で DynamoDB を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して、DynamoDB で複数の値を 1 つの属性と比較する

次のコード例は、DynamoDB で複数の値を 1 つの属性と比較する方法を示しています。

- IN 演算子を使用して、複数の値を 1 つの属性と比較します。
- IN 演算子を複数の OR 条件と比較します。
- IN の使用によるパフォーマンスと式の複雑さのメリットを理解します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して、複数の値を DynamoDB の 1 つの属性と比較します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
```

```
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;

    /**
     * Queries a table using the IN operator to compare multiple values with a
 single attribute.
     *
     * <p>This method demonstrates how to use the IN operator in a filter
 expression
     * to match an attribute against multiple values.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key to query
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The query response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse compareMultipleValues(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String attributeName,
        List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":pkValue", partitionKeyValue);

        // Add values for IN operator
```

```
        for (int i = 0; i < valuesList.size(); i++) {
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the IN clause
        StringBuilder inClause = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                inClause.append(", ");
            }
            inClause.append(":val").append(i);
        }

        // Define the query parameters
        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression("#attrName IN (" + inClause.toString() + ")")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
    }

    /**
     * Queries a table using multiple OR conditions to compare multiple values
with a single attribute.
     *
     * <p>This method demonstrates the alternative approach to using the IN
operator,
     * by using multiple OR conditions.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key to query
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The query response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse compareWithOrConditions(
```

```
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String attributeName,
        List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":pkValue", partitionKeyValue);

        // Add values for OR conditions
        for (int i = 0; i < valuesList.size(); i++) {
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the OR conditions
        StringBuilder orConditions = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                orConditions.append(" OR ");
            }
            orConditions.append("#attrName = :val").append(i);
        }

        // Define the query parameters
        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression(orConditions.toString())
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
    }

    /**
```

```
    * Compares the performance of using the IN operator versus multiple OR
conditions.
    *
    * <p>This method demonstrates the performance difference between using the
IN operator
    * and using multiple OR conditions.
    *
    * @param dynamoDbClient The DynamoDB client
    * @param tableName The name of the DynamoDB table
    * @param partitionKeyName The name of the partition key attribute
    * @param partitionKeyValue The value of the partition key to query
    * @param attributeName The name of the attribute to compare
    * @param valuesList List of values to compare against
    * @return Map containing the performance comparison results
    */
   public static Map<String, Object> comparePerformance(
       DynamoDbClient dynamoDbClient,
       String tableName,
       String partitionKeyName,
       AttributeValue partitionKeyValue,
       String attributeName,
       List<AttributeValue> valuesList) {

       Map<String, Object> results = new HashMap<>();

       try {
           // Measure performance of IN operator
           long inStartTime = System.nanoTime();
           QueryResponse inResponse = compareMultipleValues(
               dynamoDbClient, tableName, partitionKeyName, partitionKeyValue,
attributeName, valuesList);
           long inEndTime = System.nanoTime();
           long inDuration = inEndTime - inStartTime;

           // Measure performance of OR conditions
           long orStartTime = System.nanoTime();
           QueryResponse orResponse = compareWithOrConditions(
               dynamoDbClient, tableName, partitionKeyName, partitionKeyValue,
attributeName, valuesList);
           long orEndTime = System.nanoTime();
           long orDuration = orEndTime - orStartTime;

           // Record results
           results.put("inOperatorDuration", inDuration);
```

```
            results.put("orConditionsDuration", orDuration);
            results.put("inOperatorItems", inResponse.count());
            results.put("orConditionsItems", orResponse.count());
            results.put("inOperatorExpression", "IN operator with " +
valuesList.size() + " values");
            results.put("orConditionsExpression", valuesList.size() + " OR
conditions");
            results.put("success", true);

        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }

        return results;
    }

    /**
     * Scans a table using the IN operator with a large number of values.
     *
     * <p>This method demonstrates how to use the IN operator with a large number
of values,
     * which can help stay within the 300 operator limit.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param attributeName The name of the attribute to compare
     * @param valuesList List of values to compare against
     * @return The scan response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static ScanResponse scanWithLargeInClause(
        DynamoDbClient dynamoDbClient, String tableName, String attributeName,
List<AttributeValue> valuesList) {

        // Create expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#attrName", attributeName);

        // Create expression attribute values
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();

        // Add values for IN operator
        for (int i = 0; i < valuesList.size(); i++) {
```

```
            expressionAttributeValues.put(":val" + i, valuesList.get(i));
        }

        // Build the IN clause
        StringBuilder inClause = new StringBuilder();
        for (int i = 0; i < valuesList.size(); i++) {
            if (i > 0) {
                inClause.append(", ");
            }
            inClause.append(":val").append(i);
        }

        // Define the scan parameters
        ScanRequest request = ScanRequest.builder()
            .tableName(tableName)
            .filterExpression("#attrName IN (" + inClause.toString() + ")")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the scan operation
        return dynamoDbClient.scan(request);
    }

    /**
     * Generates a list of sample values for testing.
     *
     * <p>Helper method to generate a list of sample values for testing.
     *
     * @param valueType The type of values to generate (string, number, or
boolean)
     * @param count The number of values to generate
     * @return List of generated attribute values
     */
    public static List<AttributeValue> generateSampleValues(String valueType, int
count) {
        List<AttributeValue> values = new ArrayList<>();

        for (int i = 0; i < count; i++) {
            AttributeValue value;

            switch (valueType.toLowerCase(Locale.ROOT)) {
                case "string":
                    value = AttributeValue.builder().s("Value" + i).build();
```

```
                    break;
                case "number":
                    value =
AttributeValue.builder().n(String.valueOf(i)).build();
                    break;
                case "boolean":
                    value = AttributeValue.builder().bool(i % 2 == 0).build();
                    break;
                default:
                    throw new IllegalArgumentException("Unsupported value type: "
+ valueType);
            }

            values.add(value);
        }

        return values;
    }
```

AWS SDK for Java 2.x を使用した複数の値の比較の使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        System.out.println("Demonstrating how to compare multiple values with a
single attribute in DynamoDB");

        try {
            // Example 1: Using the IN operator
            System.out.println("\nExample 1: Using the IN operator");
            List<AttributeValue> categories = List.of(
                AttributeValue.builder().s("Electronics").build(),
                AttributeValue.builder().s("Computers").build(),
                AttributeValue.builder().s("Accessories").build());

            QueryResponse inResponse = compareMultipleValues(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
                "Category",
                categories);
```

```
            System.out.println("Found " + inResponse.count() + " items using IN
operator");
            System.out.println("Items: " + inResponse.items());

            // Example 2: Using multiple OR conditions
            System.out.println("\nExample 2: Using multiple OR conditions");
            QueryResponse orResponse = compareWithOrConditions(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
                "Category",
                categories);

            System.out.println("Found " + orResponse.count() + " items using OR
conditions");
            System.out.println("Items: " + orResponse.items());

            // Example 3: Performance comparison
            System.out.println("\nExample 3: Performance comparison");
            Map<String, Object> perfComparison = comparePerformance(
                dynamoDbClient,
                tableName,
                "Department",
                AttributeValue.builder().s("Retail").build(),
                "Category",
                categories);

            if ((boolean) perfComparison.get("success")) {
                System.out.println("IN operator duration: " +
perfComparison.get("inOperatorDuration") + " ns");
                System.out.println("OR conditions duration: " +
perfComparison.get("orConditionsDuration") + " ns");
                System.out.println("IN operator found " +
perfComparison.get("inOperatorItems") + " items");
                System.out.println("OR conditions found " +
perfComparison.get("orConditionsItems") + " items");
                System.out.println("Expression complexity comparison:");
                System.out.println("  IN operator: " +
perfComparison.get("inOperatorExpression"));
                System.out.println("  OR conditions: " +
perfComparison.get("orConditionsExpression"));
            } else {
```

```
                System.out.println("Performance comparison failed: " +
perfComparison.get("error"));
            }

            // Example 4: Using IN with a large number of values
            System.out.println("\nExample 4: Using IN with a large number of
values");
            List<AttributeValue> productIds = generateSampleValues("string", 20);

            ScanResponse largeInResponse = scanWithLargeInClause(dynamoDbClient,
tableName, "ProductId", productIds);

            System.out.println(
                "Found " + largeInResponse.count() + " items using IN with " +
productIds.size() + " values");

            // Explain the benefits of using IN
            System.out.println("\nKey points about using the IN operator in
DynamoDB:");
            System.out.println("1. The IN operator allows comparing a single
attribute against multiple values");
            System.out.println("2. IN is more concise than using multiple OR
conditions");
            System.out.println("3. IN counts as only 1 operator regardless of the
number of values");
            System.out.println("4. Multiple OR conditions count as 1 operator per
condition plus 1 per OR");
            System.out.println("5. Using IN helps stay within the 300 operator
limit for complex expressions");
            System.out.println("6. IN can be used in filter expressions and
condition expressions");
            System.out.println("7. The IN operator supports up to 100 comparison
values");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。

- Query
- Scan

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して複数の値を単一の属性と比較する。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  ScanCommand,
  QueryCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Query or scan a DynamoDB table to find items where an attribute matches any
 value from a list.
 *
 * This function demonstrates the use of the IN operator to compare a single
 attribute
 * against multiple possible values, which is more efficient than using multiple
 OR conditions.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} attributeName - The name of the attribute to compare against
 the values list
 * @param {Array} valuesList - List of values to compare the attribute against
 * @param {string} [partitionKeyName] - Optional name of the partition key
 attribute for query operations
 * @param {string} [partitionKeyValue] - Optional value of the partition key to
 query
 * @returns {Promise<Object>} - The response from DynamoDB containing the
 matching items
 */
async function compareMultipleValues(
  config,
  tableName,
  attributeName,
  valuesList,
```

```
  partitionKeyName,
  partitionKeyValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the filter expression using the IN operator
  const filterExpression = `${attributeName} IN (${valuesList.map((_, index) =>
 `:val${index}`).join(', ')})`;

  // Create expression attribute values for the values list
  const expressionAttributeValues = valuesList.reduce((acc, val, index) => {
    acc[`:val${index}`] = val;
    return acc;
  }, {});

  // If partition key is provided, perform a query operation
  if (partitionKeyName && partitionKeyValue) {
    const keyCondition = `${partitionKeyName} = :partitionKey`;
    expressionAttributeValues[':partitionKey'] = partitionKeyValue;

    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        KeyConditionExpression: keyCondition,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new QueryCommand(params));

      // Add the items from this page to our collection
```

```
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);

    // Return the complete result
    return {
      Items: allItems,
      Count: allItems.length
    };
  } else {
    // Otherwise, perform a scan operation
    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous scan
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new ScanCommand(params));

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }

      // Get the key for the next page of results
      lastEvaluatedKey = response.LastEvaluatedKey;
    } while (lastEvaluatedKey);

    // Return the complete result
```

```
    return {
      Items: allItems,
      Count: allItems.length
    };
  }
}

/**
 * Alternative implementation using multiple OR conditions instead of the IN
 operator.
 *
 * This function is provided for comparison to show why using the IN operator is
 preferable.
 * With many values, this approach becomes verbose and less efficient.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} attributeName - The name of the attribute to compare against
 the values list
 * @param {Array} valuesList - List of values to compare the attribute against
 * @param {string} [partitionKeyName] - Optional name of the partition key
 attribute for query operations
 * @param {string} [partitionKeyValue] - Optional value of the partition key to
 query
 * @returns {Promise<Object>} - The response from DynamoDB containing the
 matching items
 */
async function compareWithOrConditions(
  config,
  tableName,
  attributeName,
  valuesList,
  partitionKeyName,
  partitionKeyValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // If no values provided, return empty result
  if (!valuesList || valuesList.length === 0) {
    return {
      Items: [],
      Count: 0
```

```
    };
  }

  // Create the filter expression using multiple OR conditions
  const filterConditions = valuesList.map((_, index) => `${attributeName} = :val
${index}`);
  const filterExpression = filterConditions.join(' OR ');

  // Create expression attribute values for the values list
  const expressionAttributeValues = valuesList.reduce((acc, val, index) => {
    acc[`:val${index}`] = val;
    return acc;
  }, {});

  // If partition key is provided, perform a query operation
  if (partitionKeyName && partitionKeyValue) {
    const keyCondition = `${partitionKeyName} = :partitionKey`;
    expressionAttributeValues[':partitionKey'] = partitionKeyValue;

    // Initialize array to collect all items
    let allItems = [];
    let lastEvaluatedKey;

    // Use pagination to get all results
    do {
      const params = {
        TableName: tableName,
        KeyConditionExpression: keyCondition,
        FilterExpression: filterExpression,
        ExpressionAttributeValues: expressionAttributeValues
      };

      // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        params.ExclusiveStartKey = lastEvaluatedKey;
      }

      const response = await docClient.send(new QueryCommand(params));

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems = [...allItems, ...response.Items];
      }
```

```
    // Get the key for the next page of results
    lastEvaluatedKey = response.LastEvaluatedKey;
  } while (lastEvaluatedKey);

  // Return the complete result
  return {
    Items: allItems,
    Count: allItems.length
  };
} else {
  // Otherwise, perform a scan operation
  // Initialize array to collect all items
  let allItems = [];
  let lastEvaluatedKey;

  // Use pagination to get all results
  do {
    const params = {
      TableName: tableName,
      FilterExpression: filterExpression,
      ExpressionAttributeValues: expressionAttributeValues
    };

    // Add ExclusiveStartKey if we have a lastEvaluatedKey from a previous scan
    if (lastEvaluatedKey) {
      params.ExclusiveStartKey = lastEvaluatedKey;
    }

    const response = await docClient.send(new ScanCommand(params));

    // Add the items from this page to our collection
    if (response.Items && response.Items.length > 0) {
      allItems = [...allItems, ...response.Items];
    }

    // Get the key for the next page of results
    lastEvaluatedKey = response.LastEvaluatedKey;
  } while (lastEvaluatedKey);

  // Return the complete result
  return {
    Items: allItems,
    Count: allItems.length
```

```
    };
  }
}
```

AWS SDK for JavaScript を使用した複数の値の比較の使用例。

```
/**
 * Example of how to use the compareMultipleValues function.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const attributeName = "Category";
  const valuesList = ["Electronics", "Computers", "Accessories"];

  console.log(`Searching for products in any of these categories:
 ${valuesList.join(', ')}`);

  try {
    // Using the IN operator (recommended approach)
    console.log("\nApproach 1: Using the IN operator");
    const response = await compareMultipleValues(
      config,
      tableName,
      attributeName,
      valuesList
    );

    console.log(`Found ${response.Count} products in the specified categories`);

    // Using multiple OR conditions (alternative approach)
    console.log("\nApproach 2: Using multiple OR conditions");
    const response2 = await compareWithOrConditions(
      config,
      tableName,
      attributeName,
      valuesList
    );

    console.log(`Found ${response2.Count} products in the specified categories`);
```

```
    // Example with a query operation
    console.log("\nQuerying a specific manufacturer's products in multiple
 categories");
    const partitionKeyName = "Manufacturer";
    const partitionKeyValue = "Acme";

    const response3 = await compareMultipleValues(
      config,
      tableName,
      attributeName,
      valuesList,
      partitionKeyName,
      partitionKeyValue
    );

    console.log(`Found ${response3.Count} Acme products in the specified
 categories`);

    // Explain the benefits of using the IN operator
    console.log("\nBenefits of using the IN operator:");
    console.log("1. More concise expression compared to multiple OR conditions");
    console.log("2. Better readability and maintainability");
    console.log("3. Potentially better performance with large value lists");
    console.log("4. Simpler code that's less prone to errors");
    console.log("5. Easier to modify when adding or removing values");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - Query
  - Scan

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して複数の値を単一の属性と比較する。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key
from typing import Any, Dict, List, Optional


def compare_multiple_values(
    table_name: str,
    attribute_name: str,
    values_list: List[Any],
    partition_key_name: Optional[str] = None,
    partition_key_value: Optional[str] = None,
) -> Dict[str, Any]:
    """
    Query or scan a DynamoDB table to find items where an attribute matches any
 value from a list.

    This function demonstrates the use of the IN operator to compare a single
 attribute
    against multiple possible values, which is more efficient than using multiple
 OR conditions.

    Args:
        table_name (str): The name of the DynamoDB table.
        attribute_name (str): The name of the attribute to compare against the
 values list.
        values_list (List[Any]): List of values to compare the attribute against.
        partition_key_name (Optional[str]): The name of the partition key
 attribute for query operations.
        partition_key_value (Optional[str]): The value of the partition key to
 query.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the matching items.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create the filter expression using the is_in method
    filter_expression = Attr(attribute_name).is_in(values_list)

    # If partition key is provided, perform a query operation
    if partition_key_name and partition_key_value:
```

```
        key_condition = Key(partition_key_name).eq(partition_key_value)
        response = table.query(
            KeyConditionExpression=key_condition,
 FilterExpression=filter_expression
        )
    else:
        # Otherwise, perform a scan operation
        response = table.scan(FilterExpression=filter_expression)

    # Handle pagination if there are more results
    items = response.get("Items", [])
    while "LastEvaluatedKey" in response:
        if partition_key_name and partition_key_value:
            response = table.query(
                KeyConditionExpression=key_condition,
                FilterExpression=filter_expression,
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
        else:
            response = table.scan(
                FilterExpression=filter_expression,
 ExclusiveStartKey=response["LastEvaluatedKey"]
            )
        items.extend(response.get("Items", []))

    # Return the complete result
    return {"Items": items, "Count": len(items)}


def compare_with_or_conditions(
    table_name: str,
    attribute_name: str,
    values_list: List[Any],
    partition_key_name: Optional[str] = None,
    partition_key_value: Optional[str] = None,
) -> Dict[str, Any]:
    """
    Alternative implementation using multiple OR conditions instead of the IN
 operator.

    This function is provided for comparison to show why using the IN operator is
 preferable.
    With many values, this approach becomes verbose and less efficient.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        attribute_name (str): The name of the attribute to compare against the
values list.
        values_list (List[Any]): List of values to compare the attribute against.
        partition_key_name (Optional[str]): The name of the partition key
attribute for query operations.
        partition_key_value (Optional[str]): The value of the partition key to
query.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the matching items.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create a filter expression with multiple OR conditions
    filter_expression = None
    for value in values_list:
        condition = Attr(attribute_name).eq(value)
        if filter_expression is None:
            filter_expression = condition
        else:
            filter_expression = filter_expression | condition

    # If partition key is provided, perform a query operation
    if partition_key_name and partition_key_value and filter_expression:
        key_condition = Key(partition_key_name).eq(partition_key_value)
        response = table.query(
            KeyConditionExpression=key_condition,
FilterExpression=filter_expression
        )
    elif filter_expression:
        # Otherwise, perform a scan operation
        response = table.scan(FilterExpression=filter_expression)
    else:
        # Return empty response if no values provided
        return {"Items": [], "Count": 0}

    # Handle pagination if there are more results
    items = response.get("Items", [])
    while "LastEvaluatedKey" in response:
        if partition_key_name and partition_key_value:
```

```
            response = table.query(
                KeyConditionExpression=key_condition,
                FilterExpression=filter_expression,
                ExclusiveStartKey=response["LastEvaluatedKey"],
            )
        else:
            response = table.scan(
                FilterExpression=filter_expression,
 ExclusiveStartKey=response["LastEvaluatedKey"]
            )
        items.extend(response.get("Items", []))

    # Return the complete result
    return {"Items": items, "Count": len(items)}
```

AWS SDK for Python (Boto3) を使用した複数の値の比較の使用例。

```
def example_usage():
    """Example of how to use the compare_multiple_values function."""
    # Example parameters
    table_name = "Products"
    attribute_name = "Category"
    values_list = ["Electronics", "Computers", "Accessories"]

    print(f"Searching for products in any of these categories: {values_list}")

    # Using the IN operator (recommended approach)
    print("\nApproach 1: Using the IN operator")
    response = compare_multiple_values(
        table_name=table_name, attribute_name=attribute_name,
 values_list=values_list
    )

    print(f"Found {response['Count']} products in the specified categories")

    # Using multiple OR conditions (alternative approach)
    print("\nApproach 2: Using multiple OR conditions")
    response2 = compare_with_or_conditions(
        table_name=table_name, attribute_name=attribute_name,
 values_list=values_list
```

```
    )

    print(f"Found {response2['Count']} products in the specified categories")

    # Example with a query operation
    print("\nQuerying a specific manufacturer's products in multiple categories")
    partition_key_name = "Manufacturer"
    partition_key_value = "Acme"

    response3 = compare_multiple_values(
        table_name=table_name,
        attribute_name=attribute_name,
        values_list=values_list,
        partition_key_name=partition_key_name,
        partition_key_value=partition_key_value,
    )

    print(f"Found {response3['Count']} Acme products in the specified
categories")

    # Explain the benefits of using the IN operator
    print("\nBenefits of using the IN operator:")
    print("1. More concise expression compared to multiple OR conditions")
    print("2. Better readability and maintainability")
    print("3. Potentially better performance with large value lists")
    print("4. Simpler code that's less prone to errors")
    print("5. Easier to modify when adding or removing values")
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の以下のトピックを参照してください。
  - Query
  - Scan

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して TTL を含む DynamoDB 項目を条件付きで更新する

以下のコード例は、項目の TTL を条件付きで更新する方法を示しています。

Java

SDK for Java 2.x

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Map;
import java.util.Optional;

/**
 * Updates an item in a DynamoDB table with TTL attributes using a conditional
 expression.
 * This class demonstrates how to conditionally update TTL expiration timestamps.
 */
public class UpdateTTLConditional {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
```

```
                sortKey - The name of the sort key. Also known as the range
attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
is located in. (Default: us-east-1)
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String UPDATED_AT_ATTR = "updatedAt";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String UPDATE_EXPRESSION = "SET " + UPDATED_AT_ATTR +
"=:c, " + EXPIRE_AT_ATTR + "=:e";
    private static final String CONDITION_EXPRESSION = EXPIRE_AT_ATTR + " > :c";
    private static final String SUCCESS_MESSAGE = "%s UpdateItem operation with
TTL successful.";
    private static final String CONDITION_FAILED_MESSAGE = "Condition check
failed. The item may have already expired.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs an UpdateTTLConditional with a default DynamoDB client.
     */
    public UpdateTTLConditional() {
        this.dynamoDbClient = null;
    }

    /**
     * Constructs an UpdateTTLConditional with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public UpdateTTLConditional(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Main method to demonstrate conditionally updating an item with TTL.
     *
     * @param args Command line arguments
     */
```

```
    public static void main(final String[] args) {
        try {
            int result = new UpdateTTLConditional().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Process command line arguments and conditionally update an item with TTL.
     *
     * @param args Command line arguments
     * @return 0 if successful, non-zero otherwise
     * @throws ResourceNotFoundException If the table doesn't exist
     * @throws DynamoDbException If an error occurs during the operation
     * @throws IllegalArgumentException If arguments are invalid
     */
    public int processArgs(final String[] args) {
        // Argument validation (remove or replace this line when reusing this
code)
        CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

        final String tableName = args[0];
        final String primaryKey = args[1];
        final String sortKey = args[2];
        final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
            .map(Region::of)
            .orElse(Region.US_EAST_1);

        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = Map.of(
            PRIMARY_KEY_ATTR, AttributeValue.builder().s(primaryKey).build(),
            SORT_KEY_ATTR, AttributeValue.builder().s(sortKey).build());

        // Create the expression attribute values
```

```
        final Map<String, AttributeValue> expressionAttributeValues = Map.of(
            ":c",
 AttributeValue.builder().n(String.valueOf(currentTime)).build(),
            ":e",
 AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .conditionExpression(CONDITION_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final UpdateItemResponse response = ddb.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return 0;
        } catch (ConditionalCheckFailedException e) {
            System.err.println(CONDITION_FAILED_MESSAGE);
            throw e;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItemConditional = async (tableName, partitionKey, sortKey,
 region = 'us-east-1', newAttribute = 'default-value') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            artist: partitionKey,
            album: sortKey
        }),
        UpdateExpression: "SET newAttribute = :newAttribute",
        ConditionExpression: "expireAt > :expiration",
        ExpressionAttributeValues: marshall({
            ':newAttribute': newAttribute,
            ':expiration': currentTime
        }),
        ReturnValues: "ALL_NEW"
    };

    try {
        const response = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(response.Attributes);
        console.log("Item updated successfully: ", responseData);
        return responseData;
    } catch (error) {
        if (error.name === "ConditionalCheckFailedException") {
            console.log("Condition check failed: Item's 'expireAt' is expired.");
        } else {
            console.error("Error updating item: ", error);
        }
        throw error;
    }
};

// Example usage (commented out for testing)
```

```
// updateItemConditional('your-table-name', 'your-partition-key-value', 'your-
sort-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

条件を指定して、テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
from datetime import datetime, timedelta

import boto3
from botocore.exceptions import ClientError


def update_dynamodb_item_ttl(table_name, region, primary_key, sort_key,
 ttl_attribute):
    """
    Updates an existing record in a DynamoDB table with a new or updated TTL
 attribute.

    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :param ttl_attribute: name of the TTL attribute in the target DynamoDB table
    :return:
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Generate updated TTL in epoch second format
        updated_expiration_time = int((datetime.now() +
 timedelta(days=90)).timestamp())

        # Define the update expression for adding/updating a new attribute
        update_expression = "SET newAttribute = :val1"
```

```
        # Define the condition expression for checking if 'expireAt' is not
 expired
        condition_expression = "expireAt > :val2"

        # Define the expression attribute values
        expression_attribute_values = {":val1": ttl_attribute, ":val2":
 updated_expiration_time}

        response = table.update_item(
            Key={"primaryKey": primary_key, "sortKey": sort_key},
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeValues=expression_attribute_values,
        )

        print("Item updated successfully.")
        return response["ResponseMetadata"]["HTTPStatusCode"]  # Ideally a 200 OK
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            print("Condition check failed: Item's 'expireAt' is expired.")
        else:
            print(f"Error updating item: {e}")
    except Exception as e:
        print(f"Error updating item: {e}")


# replace with your values
update_dynamodb_item_ttl(
    "your-table-name",
    "us-east-1",
    "your-partition-key-value",
    "your-sort-key-value",
    "your-ttl-attribute-value",
)
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用してローカル DynamoDB インスタンスに接続する

次のコード例は、エンドポイント URL を上書きして DynamoDB と AWS SDK のローカル開発デプロイに接続する方法を示しています。

詳細については、「DynamoDB Local」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// Lists your tables from a local DynamoDB instance by setting the SDK Config's
/// endpoint_url and test_credentials.
#[tokio::main]
async fn main() {
    tracing_subscriber::fmt::init();

    let config = aws_config::defaults(aws_config::BehaviorVersion::latest())
        .test_credentials()
        // DynamoDB run locally uses port 8000 by default.
        .endpoint_url("http://localhost:8000")
        .load()
        .await;
    let dynamodb_local_config =
 aws_sdk_dynamodb::config::Builder::from(&config).build();

    let client = aws_sdk_dynamodb::Client::from_conf(dynamodb_local_config);

    let list_resp = client.list_tables().send().await;
    match list_resp {
        Ok(resp) => {
            println!("Found {} tables", resp.table_names().len());
            for name in resp.table_names() {
                println!("  {}", name);
            }
```

```
        }
        Err(err) => eprintln!("Failed to list local dynamodb tables: {err:?}"),
    }
}
```

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDKを使用した DynamoDB のカウント式演算子

次のコード例は、DynamoDB で式演算子をカウントする方法を示しています。

- DynamoDB の演算子の制限数が 300 であることを理解します。
- 複雑な式での演算子をカウントする。
- 式を最適化して制限内に収まるようにします。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して式演算子のカウントをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

    /**
     * Creates a complex filter expression with a specified number of conditions.
     *
     * <p>This method demonstrates how to generate a complex expression with
     * a specific number of operators to test the 300 operator limit.
```

```
    *
    * @param conditionsCount Number of conditions to include
    * @param useAnd Whether to use AND (true) or OR (false) between conditions
    * @return Map containing the filter expression, attribute values, and
operator count
    */
   public static Map<String, Object> createComplexFilterExpression(int
conditionsCount, boolean useAnd) {
       // Initialize the expression parts and attribute values
       StringBuilder filterExpression = new StringBuilder();
       Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();

       // Generate the specified number of conditions
       for (int i = 0; i < conditionsCount; i++) {
           // Add the operator between conditions (except for the first one)
           if (i > 0) {
               filterExpression.append(useAnd ? " AND " : " OR ");
           }

           // Alternate between different comparison operators for variety
           String valueKey = ":val" + i;

           switch (i % 5) {
               case 0:
                   filterExpression.append("attribute").append(i).append(" =
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey, AttributeValue.builder().s("value" +
i).build());
                   break;
               case 1:
                   filterExpression.append("attribute").append(i).append(" >
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey,
AttributeValue.builder().n(String.valueOf(i)).build());
                   break;
               case 2:
                   filterExpression.append("attribute").append(i).append(" <
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey,
                       AttributeValue.builder().n(String.valueOf(i *
10)).build());
```

```
                    break;
                case 3:
                    filterExpression
                        .append("contains(attribute")
                        .append(i)
                        .append(", ")
                        .append(valueKey)
                        .append(")");
                    expressionAttributeValues.put(
                        valueKey, AttributeValue.builder().s("substring" +
i).build());
                    break;
                case 4:
                    filterExpression
                        .append("attribute_exists(attribute")
                        .append(i)
                        .append(")");
                    break;
                default:
                    // This case will never be reached, but added to satisfy
checkstyle
                    break;
            }
        }

        // Calculate the operator count
        // Each condition has 1 operator (=, >, <, contains, attribute_exists)
        // Each AND or OR between conditions is 1 operator
        int operatorCount = conditionsCount + (conditionsCount > 0 ?
conditionsCount - 1 : 0);

        // Create the result map
        Map<String, Object> result = new HashMap<>();
        result.put("filterExpression", filterExpression.toString());
        result.put("expressionAttributeValues", expressionAttributeValues);
        result.put("operatorCount", operatorCount);

        return result;
    }

    /**
     * Creates a complex update expression with a specified number of operations.
     *
```

```
    * <p>This method demonstrates how to generate a complex update expression
with
    * a specific number of operators to test the 300 operator limit.
    *
    * @param operationsCount Number of operations to include
    * @return Map containing the update expression, attribute values, and
operator count
    */
   public static Map<String, Object> createComplexUpdateExpression(int
operationsCount) {
       // Initialize the expression parts and attribute values
       StringBuilder updateExpression = new StringBuilder("SET ");
       Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();

       // Generate the specified number of SET operations
       for (int i = 0; i < operationsCount; i++) {
           // Add comma between operations (except for the first one)
           if (i > 0) {
               updateExpression.append(", ");
           }

           // Alternate between different types of SET operations
           String valueKey = ":val" + i;

           switch (i % 3) {
               case 0:
                   // Simple assignment (1 operator: =)
                   updateExpression.append("attribute").append(i).append(" =
").append(valueKey);
                   expressionAttributeValues.put(
                       valueKey, AttributeValue.builder().s("value" +
i).build());
                   break;
               case 1:
                   // Addition (2 operators: = and +)
                   updateExpression
                       .append("attribute")
                       .append(i)
                       .append(" = attribute")
                       .append(i)
                       .append(" + ")
                       .append(valueKey);
                   expressionAttributeValues.put(
```

```
                        valueKey,
AttributeValue.builder().n(String.valueOf(i)).build());
                    break;
                case 2:
                    // Conditional assignment with if_not_exists (2 operators: =
and if_not_exists)
                    updateExpression
                        .append("attribute")
                        .append(i)
                        .append(" = if_not_exists(attribute")
                        .append(i)
                        .append(", ")
                        .append(valueKey)
                        .append(")");
                    expressionAttributeValues.put(
                        valueKey,
                        AttributeValue.builder().n(String.valueOf(i *
10)).build());
                    break;
                default:
                    // This case will never be reached, but added to satisfy
checkstyle
                    break;
            }
        }

        // Calculate the operator count
        // Each operation has 1-2 operators as noted above
        int operatorCount = 0;
        for (int i = 0; i < operationsCount; i++) {
            operatorCount += (i % 3 == 0) ? 1 : 2;
        }

        // Create the result map
        Map<String, Object> result = new HashMap<>();
        result.put("updateExpression", updateExpression.toString());
        result.put("expressionAttributeValues", expressionAttributeValues);
        result.put("operatorCount", operatorCount);

        return result;
    }

    /**
```

```
    * Test the operator limit by attempting an operation with a complex
expression.
    *
    * <p>This method demonstrates what happens when an expression approaches or
    * exceeds the 300 operator limit.
    *
    * @param dynamoDbClient The DynamoDB client
    * @param tableName The name of the DynamoDB table
    * @param key The key of the item to update
    * @param operatorCount Target number of operators to include
    * @return Map containing the result of the operation attempt
    */
   public static Map<String, Object> testOperatorLimit(
       DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, int operatorCount) {

       // Create a complex update expression with the specified operator count
       Map<String, Object> expressionData =
           createComplexUpdateExpression((int) Math.ceil(operatorCount /
1.5)); // Adjust to get close to target count

       String updateExpression = (String)
expressionData.get("updateExpression");
       @SuppressWarnings("unchecked")
       Map<String, AttributeValue> expressionAttributeValues =
           (Map<String, AttributeValue>)
expressionData.get("expressionAttributeValues");
       int actualCount = (int) expressionData.get("operatorCount");

       System.out.println("Generated update expression with approximately " +
actualCount + " operators");

       // Define the update parameters
       UpdateItemRequest request = UpdateItemRequest.builder()
           .tableName(tableName)
           .key(key)
           .updateExpression(updateExpression)
           .expressionAttributeValues(expressionAttributeValues)
           .returnValues("UPDATED_NEW")
           .build();

       try {
           // Attempt the update operation
           UpdateItemResponse response = dynamoDbClient.updateItem(request);
```

```
            Map<String, Object> result = new HashMap<>();
            result.put("success", true);
            result.put("message", "Operation succeeded with " + actualCount + "
operators");
            result.put("data", response);
            return result;

        } catch (DynamoDbException e) {
            // Check if the error is due to exceeding the operator limit
            if (e.getMessage().contains("too many operators")) {
                Map<String, Object> result = new HashMap<>();
                result.put("success", false);
                result.put("message", "Operation failed: " + e.getMessage());
                result.put("operatorCount", actualCount);
                return result;
            }

            // Return other errors
            Map<String, Object> result = new HashMap<>();
            result.put("success", false);
            result.put("message", "Operation failed: " + e.getMessage());
            result.put("error", e);
            return result;
        }
    }

    /**
     * Break down a complex expression into multiple simpler operations.
     *
     * <p>This method demonstrates how to handle expressions that would exceed
     * the 300 operator limit by breaking them into multiple operations.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param totalOperations Total number of operations to perform
     * @return Map containing the results of the operations
     */
    public static Map<String, Object> breakDownComplexExpression(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, int totalOperations) {

        // Calculate how many operations we can safely include in each batch
```

```
        // Using 150 as a conservative limit (well below 300)
        final int operationsPerBatch = 100;
        final int batchCount = (int) Math.ceil((double) totalOperations /
operationsPerBatch);

        System.out.println("Breaking down " + totalOperations + " operations into
" + batchCount + " batches");

        Map<String, Object> results = new HashMap<>();
        results.put("totalBatches", batchCount);

        Map<Integer, Map<String, Object>> batchResults = new HashMap<>();

        // Process each batch
        for (int batch = 0; batch < batchCount; batch++) {
            // Calculate the operations for this batch
            int batchStart = batch * operationsPerBatch;
            int batchEnd = Math.min(batchStart + operationsPerBatch,
totalOperations);
            int batchSize = batchEnd - batchStart;

            System.out.println(
                "Processing batch " + (batch + 1) + "/" + batchCount + " with " +
batchSize + " operations");

            // Create an update expression for this batch
            Map<String, Object> expressionData =
createComplexUpdateExpression(batchSize);

            String updateExpression = (String)
expressionData.get("updateExpression");
            @SuppressWarnings("unchecked")
            Map<String, AttributeValue> expressionAttributeValues =
                (Map<String, AttributeValue>)
expressionData.get("expressionAttributeValues");
            int operatorCount = (int) expressionData.get("operatorCount");

            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression(updateExpression)
                .expressionAttributeValues(expressionAttributeValues)
                .returnValues("UPDATED_NEW")
```

```
                .build();

            try {
                // Perform the update operation for this batch
                UpdateItemResponse response = dynamoDbClient.updateItem(request);

                Map<String, Object> batchResult = new HashMap<>();
                batchResult.put("batch", batch + 1);
                batchResult.put("success", true);
                batchResult.put("operatorCount", operatorCount);
                batchResult.put("attributes", response.attributes());

                batchResults.put(batch, batchResult);

            } catch (DynamoDbException e) {
                Map<String, Object> batchResult = new HashMap<>();
                batchResult.put("batch", batch + 1);
                batchResult.put("success", false);
                batchResult.put("operatorCount", operatorCount);
                batchResult.put("error", e.getMessage());

                batchResults.put(batch, batchResult);

                // Continue with next batch instead of breaking
                continue;
            }
        }

        results.put("results", batchResults);
        return results;
    }

    /**
     * Count operators in a DynamoDB expression based on the rules in the
documentation.
     *
     * <p>This method demonstrates how operators are counted according to the
     * DynamoDB documentation.
     *
     * @param expression The DynamoDB expression to analyze
     * @return Map containing the breakdown of operator counts
     */
    public static Map<String, Integer> countOperatorsInExpression(String
expression) {
```

```
        // Initialize counters for different operator types
        Map<String, Integer> counts = new HashMap<>();
        counts.put("comparisonOperators", 0);
        counts.put("logicalOperators", 0);
        counts.put("functions", 0);
        counts.put("arithmeticOperators", 0);
        counts.put("specialOperators", 0);
        counts.put("total", 0);

        // Count comparison operators (=, <>, <, <=, >, >=)
        // This is a simplified approach and may not catch all cases
        int comparisonCount = 0;
        Pattern comparisonPattern = Pattern.compile("(=|<>|<=|>=|<|>)");
        Matcher comparisonMatcher = comparisonPattern.matcher(expression);
        while (comparisonMatcher.find()) {
            comparisonCount++;
        }
        counts.put("comparisonOperators", comparisonCount);

        // Count logical operators (AND, OR, NOT)
        int andCount = countOccurrences(expression, "\\bAND\\b");
        int orCount = countOccurrences(expression, "\\bOR\\b");
        int notCount = countOccurrences(expression, "\\bNOT\\b");
        counts.put("logicalOperators", andCount + orCount + notCount);

        // Count functions (attribute_exists, attribute_not_exists,
 attribute_type, begins_with, contains, size)
        int functionCount = countOccurrences(
            expression,
            "\\b(attribute_exists|attribute_not_exists|attribute_type|
begins_with|contains|size|if_not_exists)\\(");
        counts.put("functions", functionCount);

        // Count arithmetic operators (+ and -)
        // This is a simplified approach and may not catch all cases
        int arithmeticCount = 0;
        Pattern arithmeticPattern = Pattern.compile("[a-zA-Z0-9_)\\]]\\s*[\\+\
\-]\\s*[a-zA-Z0-9_:(]");
        Matcher arithmeticMatcher = arithmeticPattern.matcher(expression);
        while (arithmeticMatcher.find()) {
            arithmeticCount++;
        }
        counts.put("arithmeticOperators", arithmeticCount);
```

```
        // Count special operators (BETWEEN, IN)
        int betweenCount = countOccurrences(expression, "\\bBETWEEN\\b");
        int inCount = countOccurrences(expression, "\\bIN\\b");
        counts.put("specialOperators", betweenCount + inCount);

        // Add extra operators for BETWEEN (each BETWEEN includes an AND)
        int currentLogicalOps = counts.getOrDefault("logicalOperators", 0);
        counts.put("logicalOperators", currentLogicalOps + betweenCount);

        // Calculate total
        int total = counts.getOrDefault("comparisonOperators", 0)
            + counts.getOrDefault("logicalOperators", 0)
            + counts.getOrDefault("functions", 0)
            + counts.getOrDefault("arithmeticOperators", 0)
            + counts.getOrDefault("specialOperators", 0);
        counts.put("total", total);

        return counts;
    }

    /**
     * Helper method to count occurrences of a pattern in a string.
     *
     * @param text The text to search in
     * @param regex The regular expression pattern to search for
     * @return The number of occurrences
     */
    private static int countOccurrences(String text, String regex) {
        final Pattern pattern = Pattern.compile(regex);
        final Matcher matcher = pattern.matcher(text);
        int count = 0;
        while (matcher.find()) {
            count++;
        }
        return count;
    }
```

AWS SDK for Java 2.x を使用した式演算子のカウントの使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
```

```
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating DynamoDB expression operator counting
and the 300 operator limit");

        try {
            // Example 1: Analyze a simple expression
            System.out.println("\nExample 1: Analyzing a simple expression");
            String simpleExpression = "Price = :price AND Rating > :rating AND
Category IN (:cat1, :cat2, :cat3)";
            Map<String, Integer> simpleCount =
countOperatorsInExpression(simpleExpression);

            System.out.println("Expression: " + simpleExpression);
            System.out.println("Operator count breakdown:");
            System.out.println("- Comparison operators: " +
simpleCount.get("comparisonOperators"));
            System.out.println("- Logical operators: " +
simpleCount.get("logicalOperators"));
            System.out.println("- Functions: " + simpleCount.get("functions"));
            System.out.println("- Arithmetic operators: " +
simpleCount.get("arithmeticOperators"));
            System.out.println("- Special operators: " +
simpleCount.get("specialOperators"));
            System.out.println("- Total operators: " + simpleCount.get("total"));

            // Example 2: Analyze a complex expression
            System.out.println("\nExample 2: Analyzing a complex expression");
            String complexExpression = "(attribute_exists(Category) AND Size
BETWEEN :min AND :max) OR "
                + "(Price > :price AND contains(Description, :keyword) AND "
                + "(Rating >= :minRating OR Reviews > :minReviews))";
            Map<String, Integer> complexCount =
countOperatorsInExpression(complexExpression);

            System.out.println("Expression: " + complexExpression);
            System.out.println("Operator count breakdown:");
            System.out.println("- Comparison operators: " +
complexCount.get("comparisonOperators"));
            System.out.println("- Logical operators: " +
complexCount.get("logicalOperators"));
            System.out.println("- Functions: " + complexCount.get("functions"));
```

```
            System.out.println("- Arithmetic operators: " +
complexCount.get("arithmeticOperators"));
            System.out.println("- Special operators: " +
complexCount.get("specialOperators"));
            System.out.println("- Total operators: " +
complexCount.get("total"));

            // Example 3: Test approaching the operator limit
            System.out.println("\nExample 3: Testing an expression approaching
the operator limit");
            Map<String, Object> approachingLimit =
testOperatorLimit(dynamoDbClient, tableName, key, 290);
            System.out.println(approachingLimit.get("message"));

            // Example 4: Test exceeding the operator limit
            System.out.println("\nExample 4: Testing an expression exceeding the
operator limit");
            Map<String, Object> exceedingLimit =
testOperatorLimit(dynamoDbClient, tableName, key, 310);
            System.out.println(exceedingLimit.get("message"));

            // Example 5: Breaking down a complex expression
            System.out.println("\nExample 5: Breaking down a complex expression
into multiple operations");
            Map<String, Object> breakdownResult =
breakDownComplexExpression(dynamoDbClient, tableName, key, 500);
            @SuppressWarnings("unchecked")
            Map<Integer, Map<String, Object>> results =
                (Map<Integer, Map<String, Object>>)
breakdownResult.get("results");
            System.out.println(
                "Processed " + results.size() + " of " +
breakdownResult.get("totalBatches") + " batches");

            // Explain the operator counting rules
            System.out.println("\nKey points about DynamoDB expression operator
counting:");
            System.out.println("1. The maximum number of operators in any
expression is 300");
            System.out.println("2. Each comparison operator (=, <>, <, <=, >, >=)
counts as 1 operator");
            System.out.println("3. Each logical operator (AND, OR, NOT) counts as
1 operator");
```

```
            System.out.println("4. Each function call (attribute_exists,
 contains, etc.) counts as 1 operator");
            System.out.println("5. Each arithmetic operator (+ or -) counts as 1
 operator");
            System.out.println("6. BETWEEN counts as 2 operators (BETWEEN itself
 and the AND within it)");
            System.out.println("7. IN counts as 1 operator regardless of the
 number of values");
            System.out.println("8. Parentheses for grouping and attribute paths
 don't count as operators");
            System.out.println("9. When you exceed the limit, the error always
 reports '301 operators'");
            System.out.println("10. For complex operations, break them into
 multiple smaller operations");

        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して式演算子のカウントをデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  QueryCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Create a complex filter expression with a specified number of conditions.
 *
 * This function demonstrates how to generate a complex expression with
 * a specific number of operators to test the 300 operator limit.
```

```
 *
 * @param {number} conditionsCount - Number of conditions to include
 * @param {boolean} useAnd - Whether to use AND (true) or OR (false) between
 conditions
 * @returns {Object} - Object containing the filter expression and attribute
 values
 */
function createComplexFilterExpression(conditionsCount, useAnd = true) {
  // Initialize the expression parts and attribute values
  const conditions = [];
  const expressionAttributeValues = {};

  // Generate the specified number of conditions
  for (let i = 0; i < conditionsCount; i++) {
    // Alternate between different comparison operators for variety
    let condition;
    const valueKey = `:val${i}`;

    switch (i % 5) {
      case 0:
        condition = `attribute${i} = ${valueKey}`;
        expressionAttributeValues[valueKey] = `value${i}`;
        break;
      case 1:
        condition = `attribute${i} > ${valueKey}`;
        expressionAttributeValues[valueKey] = i;
        break;
      case 2:
        condition = `attribute${i} < ${valueKey}`;
        expressionAttributeValues[valueKey] = i * 10;
        break;
      case 3:
        condition = `contains(attribute${i}, ${valueKey})`;
        expressionAttributeValues[valueKey] = `substring${i}`;
        break;
      case 4:
        condition = `attribute_exists(attribute${i})`;
        break;
    }

    conditions.push(condition);
  }

  // Join the conditions with AND or OR
```

```
  const operator = useAnd ? " AND " : " OR ";
  const filterExpression = conditions.join(operator);

  // Calculate the operator count
  // Each condition has 1 operator (=, >, <, contains, attribute_exists)
  // Each AND or OR between conditions is 1 operator
  const operatorCount = conditionsCount + (conditionsCount > 0 ? conditionsCount
 - 1 : 0);

  return {
    filterExpression,
    expressionAttributeValues,
    operatorCount
  };
}

/**
 * Create a complex update expression with a specified number of operations.
 *
 * This function demonstrates how to generate a complex update expression with
 * a specific number of operators to test the 300 operator limit.
 *
 * @param {number} operationsCount - Number of operations to include
 * @returns {Object} - Object containing the update expression and attribute
 values
 */
function createComplexUpdateExpression(operationsCount) {
  // Initialize the expression parts and attribute values
  const setOperations = [];
  const expressionAttributeValues = {};

  // Generate the specified number of SET operations
  for (let i = 0; i < operationsCount; i++) {
    // Alternate between different types of SET operations
    let operation;
    const valueKey = `:val${i}`;

    switch (i % 3) {
      case 0:
        // Simple assignment (1 operator: =)
        operation = `attribute${i} = ${valueKey}`;
        expressionAttributeValues[valueKey] = `value${i}`;
        break;
      case 1:
```

```
        // Addition (2 operators: = and +)
        operation = `attribute${i} = attribute${i} + ${valueKey}`;
        expressionAttributeValues[valueKey] = i;
        break;
      case 2:
        // Conditional assignment with if_not_exists (2 operators: = and
 if_not_exists)
        operation = `attribute${i} = if_not_exists(attribute${i}, ${valueKey})`;
        expressionAttributeValues[valueKey] = i * 10;
        break;
    }

    setOperations.push(operation);
  }

  // Create the update expression
  const updateExpression = `SET ${setOperations.join(", ")}`;

  // Calculate the operator count
  // Each operation has 1-2 operators as noted above
  let operatorCount = 0;
  for (let i = 0; i < operationsCount; i++) {
    operatorCount += (i % 3 === 0) ? 1 : 2;
  }

  return {
    updateExpression,
    expressionAttributeValues,
    operatorCount
  };
}

/**
 * Test the operator limit by attempting an operation with a complex expression.
 *
 * This function demonstrates what happens when an expression approaches or
 * exceeds the 300 operator limit.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {number} operatorCount - Target number of operators to include
 * @returns {Promise<Object>} - Result of the operation attempt
 */
```

```
async function testOperatorLimit(
  config,
  tableName,
  key,
  operatorCount
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create a complex update expression with the specified operator count
  const { updateExpression, expressionAttributeValues, operatorCount:
 actualCount } =
    createComplexUpdateExpression(Math.ceil(operatorCount / 1.5)); // Adjust to
 get close to target count

  console.log(`Generated update expression with approximately ${actualCount}
 operators`);

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Attempt the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return {
      success: true,
      message: `Operation succeeded with ${actualCount} operators`,
      data: response
    };
  } catch (error) {
    // Check if the error is due to exceeding the operator limit
    if (error.name === "ValidationException" &&
        error.message.includes("too many operators")) {
      return {
        success: false,
        message: `Operation failed: ${error.message}`,
        operatorCount: actualCount
```

```
      };
    }

    // Return other errors
    return {
      success: false,
      message: `Operation failed: ${error.message}`,
      error
    };
  }
}

/**
 * Break down a complex expression into multiple simpler operations.
 *
 * This function demonstrates how to handle expressions that would exceed
 * the 300 operator limit by breaking them into multiple operations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {number} totalOperations - Total number of operations to perform
 * @returns {Promise<Object>} - Result of the operations
 */
async function breakDownComplexExpression(
  config,
  tableName,
  key,
  totalOperations
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Calculate how many operations we can safely include in each batch
  // Using 150 as a conservative limit (well below 300)
  const operationsPerBatch = 100;
  const batchCount = Math.ceil(totalOperations / operationsPerBatch);

  console.log(`Breaking down ${totalOperations} operations into ${batchCount}
 batches`);

  const results = [];
```

```
 // Process each batch
 for (let batch = 0; batch < batchCount; batch++) {
   // Calculate the operations for this batch
   const batchStart = batch * operationsPerBatch;
   const batchEnd = Math.min(batchStart + operationsPerBatch, totalOperations);
   const batchSize = batchEnd - batchStart;

   console.log(`Processing batch ${batch + 1}/${batchCount} with ${batchSize}
operations`);

   // Create an update expression for this batch
   const { updateExpression, expressionAttributeValues, operatorCount } =
     createComplexUpdateExpression(batchSize);

   // Define the update parameters
   const params = {
     TableName: tableName,
     Key: key,
     UpdateExpression: updateExpression,
     ExpressionAttributeValues: expressionAttributeValues,
     ReturnValues: "UPDATED_NEW"
   };

   try {
     // Perform the update operation for this batch
     const response = await docClient.send(new UpdateCommand(params));

     results.push({
       batch: batch + 1,
       success: true,
       operatorCount,
       attributes: response.Attributes
     });
   } catch (error) {
     results.push({
       batch: batch + 1,
       success: false,
       operatorCount,
       error: error.message
     });

     // Stop processing if an error occurs
     break;
   }
```

```
  }

  return {
    totalBatches: batchCount,
    results
  };
}

/**
 * Count operators in a DynamoDB expression based on the rules in the
 documentation.
 *
 * This function demonstrates how operators are counted according to the
 * DynamoDB documentation.
 *
 * @param {string} expression - The DynamoDB expression to analyze
 * @returns {Object} - Breakdown of operator counts
 */
function countOperatorsInExpression(expression) {
  // Initialize counters for different operator types
  const counts = {
    comparisonOperators: 0,
    logicalOperators: 0,
    functions: 0,
    arithmeticOperators: 0,
    specialOperators: 0,
    total: 0
  };

  // Count comparison operators (=, <>, <, <=, >, >=)
  const comparisonRegex = /[^<>]=[^=]|<>|<=|>=|[^<]>[^=]|[^>]<[^=]/g;
  const comparisonMatches = expression.match(comparisonRegex) || [];
  counts.comparisonOperators = comparisonMatches.length;

  // Count logical operators (AND, OR, NOT)
  const andMatches = expression.match(/\bAND\b/g) || [];
  const orMatches = expression.match(/\bOR\b/g) || [];
  const notMatches = expression.match(/\bNOT\b/g) || [];
  counts.logicalOperators = andMatches.length + orMatches.length +
 notMatches.length;

  // Count functions (attribute_exists, attribute_not_exists, attribute_type,
 begins_with, contains, size)
```

```
  const functionRegex = /\b(attribute_exists|attribute_not_exists|attribute_type|
begins_with|contains|size|if_not_exists)\(/g;
  const functionMatches = expression.match(functionRegex) || [];
  counts.functions = functionMatches.length;

  // Count arithmetic operators (+ and -)
  const arithmeticMatches = expression.match(/[a-zA-Z0-9_)\]]\s*[\+\-]\s*[a-zA-
Z0-9_(:]/g) || [];
  counts.arithmeticOperators = arithmeticMatches.length;

  // Count special operators (BETWEEN, IN)
  const betweenMatches = expression.match(/\bBETWEEN\b/g) || [];
  const inMatches = expression.match(/\bIN\b/g) || [];
  counts.specialOperators = betweenMatches.length + inMatches.length;

  // Add extra operators for BETWEEN (each BETWEEN includes an AND)
  counts.logicalOperators += betweenMatches.length;

  // Calculate total
  counts.total = counts.comparisonOperators +
                 counts.logicalOperators +
                 counts.functions +
                 counts.arithmeticOperators +
                 counts.specialOperators;

  return counts;
}
```

AWS SDK for JavaScript を使用した式演算子のカウントの使用例。

```
/**
 * Example of how to work with expression operator counting.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating DynamoDB expression operator counting and the 300
 operator limit");
```

```
 try {
   // Example 1: Analyze a simple expression
   console.log("\nExample 1: Analyzing a simple expression");
   const simpleExpression = "Price = :price AND Rating > :rating AND Category IN
(:cat1, :cat2, :cat3)";
   const simpleCount = countOperatorsInExpression(simpleExpression);

   console.log(`Expression: ${simpleExpression}`);
   console.log("Operator count breakdown:");
   console.log(`- Comparison operators: ${simpleCount.comparisonOperators}`);
   console.log(`- Logical operators: ${simpleCount.logicalOperators}`);
   console.log(`- Functions: ${simpleCount.functions}`);
   console.log(`- Arithmetic operators: ${simpleCount.arithmeticOperators}`);
   console.log(`- Special operators: ${simpleCount.specialOperators}`);
   console.log(`- Total operators: ${simpleCount.total}`);

   // Example 2: Analyze a complex expression
   console.log("\nExample 2: Analyzing a complex expression");
   const complexExpression =
     "(attribute_exists(Category) AND Size BETWEEN :min AND :max) OR " +
     "(Price > :price AND contains(Description, :keyword) AND " +
     "(Rating >= :minRating OR Reviews > :minReviews))";
   const complexCount = countOperatorsInExpression(complexExpression);

   console.log(`Expression: ${complexExpression}`);
   console.log("Operator count breakdown:");
   console.log(`- Comparison operators: ${complexCount.comparisonOperators}`);
   console.log(`- Logical operators: ${complexCount.logicalOperators}`);
   console.log(`- Functions: ${complexCount.functions}`);
   console.log(`- Arithmetic operators: ${complexCount.arithmeticOperators}`);
   console.log(`- Special operators: ${complexCount.specialOperators}`);
   console.log(`- Total operators: ${complexCount.total}`);

   // Example 3: Test approaching the operator limit
   console.log("\nExample 3: Testing an expression approaching the operator
limit");
   const approachingLimit = await testOperatorLimit(config, tableName, key,
290);
   console.log(approachingLimit.message);

   // Example 4: Test exceeding the operator limit
   console.log("\nExample 4: Testing an expression exceeding the operator
limit");
   const exceedingLimit = await testOperatorLimit(config, tableName, key, 310);
```

```
    console.log(exceedingLimit.message);

    // Example 5: Breaking down a complex expression
    console.log("\nExample 5: Breaking down a complex expression into multiple
 operations");
    const breakdownResult = await breakDownComplexExpression(config, tableName,
 key, 500);
    console.log(`Processed ${breakdownResult.results.length} of
 ${breakdownResult.totalBatches} batches`);

    // Explain the operator counting rules
    console.log("\nKey points about DynamoDB expression operator counting:");
    console.log("1. The maximum number of operators in any expression is 300");
    console.log("2. Each comparison operator (=, <>, <, <=, >, >=) counts as 1
 operator");
    console.log("3. Each logical operator (AND, OR, NOT) counts as 1 operator");
    console.log("4. Each function call (attribute_exists, contains, etc.) counts
 as 1 operator");
    console.log("5. Each arithmetic operator (+ or -) counts as 1 operator");
    console.log("6. BETWEEN counts as 2 operators (BETWEEN itself and the AND
 within it)");
    console.log("7. IN counts as 1 operator regardless of the number of values");
    console.log("8. Parentheses for grouping and attribute paths don't count as
 operators");
    console.log("9. When you exceed the limit, the error always reports '301
 operators'");
    console.log("10. For complex operations, break them into multiple smaller
 operations");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して式演算子のカウントをデモンストレーションします。

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, List, Optional, Tuple



def create_complex_filter_expression(
    attribute_name: str, values: List[Any], use_or: bool = True
) -> Tuple[str, Dict[str, Any], Dict[str, str], int]:
    """
    Create a complex filter expression with multiple conditions.

    This function demonstrates how to build a complex filter expression
    and count the number of operators used.

    Args:
        attribute_name (str): The name of the attribute to filter on.
        values (List[Any]): List of values to compare against.
        use_or (bool, optional): Whether to use OR between conditions. Defaults
 to True.

    Returns:
        Tuple[str, Dict[str, Any], Dict[str, str], int]: A tuple containing:
            - The filter expression string
            - Expression attribute values
            - Expression attribute names
            - The number of operators used
    """
    if not values:
        return "", {}, {}, 0

    # Initialize expression components
    filter_expression = ""
    expression_attribute_values = {}
    expression_attribute_names = {"#attr": attribute_name}
    operator_count = 0
```

```
    # Build the filter expression
    for i, value in enumerate(values):
        value_placeholder = f":val{i}"
        expression_attribute_values[value_placeholder] = value

        if i > 0:
            # Add OR or AND operator between conditions
            filter_expression += " OR " if use_or else " AND "
            operator_count += 1  # Count the OR/AND operator

        # Add the condition
        filter_expression += f"#attr = {value_placeholder}"
        operator_count += 1  # Count the = operator

    return (
        filter_expression,
        expression_attribute_values,
        expression_attribute_names,
        operator_count,
    )


def create_nested_filter_expression(
    depth: int, conditions_per_level: int
) -> Tuple[str, Dict[str, Any], Dict[str, str], int]:
    """
    Create a deeply nested filter expression with multiple conditions.

    This function demonstrates how to build a complex nested filter expression
    and count the number of operators used.

    Args:
        depth (int): The depth of nesting.
        conditions_per_level (int): Number of conditions at each level.

    Returns:
        Tuple[str, Dict[str, Any], Dict[str, str], int]: A tuple containing:
            - The filter expression string
            - Expression attribute values
            - Expression attribute names
            - The number of operators used
    """
    if depth <= 0 or conditions_per_level <= 0:
        return "", {}, {}, 0
```

```
    # Initialize expression components
    expression_attribute_values = {}
    expression_attribute_names = {}
    operator_count = 0

    def build_nested_expression(current_depth: int, prefix: str) -> str:
        nonlocal operator_count

        if current_depth <= 0:
            return ""

        # Build conditions at this level
        conditions = []
        for i in range(conditions_per_level):
            attr_name = f"attr{prefix}_{i}"
            attr_placeholder = f"#attr{prefix}_{i}"
            val_placeholder = f":val{prefix}_{i}"

            expression_attribute_names[attr_placeholder] = attr_name
            expression_attribute_values[val_placeholder] = i

            conditions.append(f"{attr_placeholder} = {val_placeholder}")
            operator_count += 1  # Count the = operator

        # Join conditions with AND
        level_expression = " AND ".join(conditions)
        operator_count += max(0, len(conditions) - 1)  # Count the AND operators

        # If not at the deepest level, add nested expressions
        if current_depth > 1:
            nested_expr = build_nested_expression(current_depth - 1,
f"{prefix}_{current_depth}")
            if nested_expr:
                level_expression = f"({level_expression}) OR ({nested_expr})"
                operator_count += 1  # Count the OR operator

        return level_expression

    # Build the expression starting from the top level
    filter_expression = build_nested_expression(depth, "1")

    return (
        filter_expression,
```

```
        expression_attribute_values,
        expression_attribute_names,
        operator_count,
    )


def count_operators_in_update_expression(update_expression: str) -> int:
    """
    Count the number of operators in an update expression.

    This function demonstrates how to count operators in an update expression
    based on DynamoDB's rules.

    Args:
        update_expression (str): The update expression to analyze.

    Returns:
        int: The number of operators in the expression.
    """
    operator_count = 0

    # Count SET operations
    if "SET" in update_expression:
        set_section = (
            update_expression.split("SET")[1].split("REMOVE")[0].split("ADD")
[0].split("DELETE")[0]
        )

        # Count assignment operators (=)
        operator_count += set_section.count("=")

        # Count arithmetic operators (+, -)
        operator_count += set_section.count("+")
        operator_count += set_section.count("-")

        # Count list_append function calls (each counts as 1 operator)
        operator_count += set_section.lower().count("list_append")

        # Count if_not_exists function calls (each counts as 1 operator)
        operator_count += set_section.lower().count("if_not_exists")

    # Count REMOVE operations (no additional operators)

    # Count ADD operations (each ADD counts as 1 operator)
```

```
    if "ADD" in update_expression:
        add_section = (
            update_expression.split("ADD")[1].split("DELETE")[0].split("SET")
[0].split("REMOVE")[0]
        )
        operator_count += add_section.count(",") + 1

    # Count DELETE operations (each DELETE counts as 1 operator)
    if "DELETE" in update_expression:
        delete_section = (
            update_expression.split("DELETE")[1].split("SET")[0].split("ADD")
[0].split("REMOVE")[0]
        )
        operator_count += delete_section.count(",") + 1

    return operator_count


def count_operators_in_condition_expression(condition_expression: str) -> int:
    """
    Count the number of operators in a condition expression.

    This function demonstrates how to count operators in a condition expression
    based on DynamoDB's rules.

    Args:
        condition_expression (str): The condition expression to analyze.

    Returns:
        int: The number of operators in the expression.
    """
    operator_count = 0

    # Count comparison operators
    comparison_operators = ["=", "<>", "<", "<=", ">", ">="]
    for op in comparison_operators:
        operator_count += condition_expression.count(op)

    # Count logical operators
    operator_count += condition_expression.upper().count(" AND ")
    operator_count += condition_expression.upper().count(" OR ")
    operator_count += condition_expression.upper().count("NOT ")

    # Count BETWEEN operator (counts as 2: BETWEEN + AND)
```

```
    between_count = condition_expression.upper().count(" BETWEEN ")
    operator_count += between_count * 2

    # Count IN operator (counts as 1 regardless of number of values)
    operator_count += condition_expression.upper().count(" IN ")

    # Count functions (each counts as 1 operator)
    functions = [
        "attribute_exists",
        "attribute_not_exists",
        "attribute_type",
        "begins_with",
        "contains",
        "size",
    ]
    for func in functions:
        operator_count += condition_expression.lower().count(func)

    return operator_count


# Note: This function is for demonstration purposes only and should be called
 from example_usage()
# It's not meant to be used directly as a test function
def _test_expression_limit(
    table_name: str, key: Dict[str, Any], operator_count: int, attribute_name:
 str = "TestAttribute"
) -> Tuple[bool, Optional[str]]:
    """
    Test if an expression with a specific number of operators exceeds the limit.

    This function demonstrates how to test the 300 operator limit by creating
    an expression with a specified number of operators.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        operator_count (int): The number of operators to include in the
 expression.
        attribute_name (str, optional): The name of the attribute to update.
 Defaults to "TestAttribute".

    Returns:
        Tuple[bool, Optional[str]]: A tuple containing:
```

```
        - A boolean indicating if the operation succeeded
        - The error message if it failed, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Create an update expression with the specified number of operators
    update_expression = f"SET #{attribute_name} = :val0"
    expression_attribute_names = {f"#{attribute_name}": attribute_name}
    expression_attribute_values = {":val0": 0}

    # Add additional SET operations to reach the desired operator count
    # Each assignment adds 1 operator
    for i in range(1, operator_count):
        attr_name = f"{attribute_name}{i}"
        attr_placeholder = f"#attr{i}"
        val_placeholder = f":val{i}"

        update_expression += f", {attr_placeholder} = {val_placeholder}"
        expression_attribute_names[attr_placeholder] = attr_name
        expression_attribute_values[val_placeholder] = i

    try:
        # Attempt the update operation
        table.update_item(
            Key=key,
            UpdateExpression=update_expression,
            ExpressionAttributeNames=expression_attribute_names,
            ExpressionAttributeValues=expression_attribute_values,
        )
        return True, None
    except ClientError as e:
        error_message = e.response["Error"]["Message"]

        if "expression contains too many operators" in error_message.lower():
            return False, error_message
        else:
            # Other error occurred
            raise
```

AWS SDK for Python (Boto3) を使用した式演算子のカウントの使用例。

```
def example_usage():
    """Example of how to use the expression operator counting functions."""

    print("Example 1: Creating a complex filter expression with multiple
 conditions")
    attribute_name = "Status"
    values = ["Active", "Pending", "Processing", "Shipped", "Delivered"]

    filter_expr, expr_attr_vals, expr_attr_names, op_count =
 create_complex_filter_expression(
        attribute_name=attribute_name, values=values, use_or=True
    )

    print(f"Filter Expression: {filter_expr}")
    print(f"Expression Attribute Values: {expr_attr_vals}")
    print(f"Expression Attribute Names: {expr_attr_names}")
    print(f"Operator Count: {op_count}")

    print("\nExample 2: Creating a nested filter expression")
    nested_expr, nested_vals, nested_names, nested_count =
 create_nested_filter_expression(
        depth=3, conditions_per_level=2
    )

    print(f"Nested Filter Expression: {nested_expr}")
    print(f"Operator Count: {nested_count}")

    print("\nExample 3: Counting operators in an update expression")
    update_expression = "SET #name = :name, #age = :age + :increment,
 #address.#city = :city, #status = if_not_exists(#status, :default_status) REMOVE
 #old_field ADD #counter :value DELETE #set_attr :set_val"
    update_op_count = count_operators_in_update_expression(update_expression)

    print(f"Update Expression: {update_expression}")
    print(f"Operator Count: {update_op_count}")

    print("\nExample 4: Counting operators in a condition expression")
    condition_expression = "(#status = :active OR #status = :pending) AND #price
 BETWEEN :min_price AND :max_price AND attribute_exists(#category) AND NOT
 (#stock <= :min_stock)"
    condition_op_count =
 count_operators_in_condition_expression(condition_expression)
```

```
    print(f"Condition Expression: {condition_expression}")
    print(f"Operator Count: {condition_op_count}")

    print("\nExample 5: Testing the 300 operator limit")

    # This is just for demonstration - in a real application, you would use your
actual table
    # Note: This function is renamed to _test_expression_limit to avoid pytest
trying to run it
    print("In a real application, you would test with _test_expression_limit
function")
    print("Expression with 250 operators would be under the limit")
    print("Expression with 350 operators would exceed the 300 operator limit")

    print("\nOperator Counting Rules in DynamoDB:")
    print("1. Comparison Operators (=, <>, <, <=, >, >=): 1 operator each")
    print("2. Logical Operators (AND, OR, NOT): 1 operator each")
    print("3. BETWEEN: 2 operators (BETWEEN + AND)")
    print("4. IN: 1 operator (regardless of number of values)")
    print("5. Functions (attribute_exists, begins_with, etc.): 1 operator each")
    print("6. Arithmetic Operators (+, -): 1 operator each")
    print("7. SET assignments (=): 1 operator each")
    print("8. ADD and DELETE operations: 1 operator each")

    print("\nStrategies for Working Within the 300 Operator Limit:")
    print("1. Break operations into multiple requests")
    print("2. Use DynamoDB Transactions for complex operations")
    print("3. Optimize data model to reduce query complexity")
    print("4. Use application-side filtering for less critical filters")
    print("5. Consider using IN operator instead of multiple OR conditions")
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

COVID-19 データを追跡する API Gateway REST API を作成する

次のコード例は、架空のデータを使用して、米国の COVID-19 の日常的なケースを追跡するシステムをシミュレートする REST API を作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で AWS Chalice を使用して Amazon API Gateway、AWS Lambda、Amazon DynamoDB を使用するサーバーレス REST API を作成する方法を示しています。REST API は、架空のデータを使用して、米国の COVID-19 の日常的なケースを追跡するシステムをシミュレートします。以下ではその方法を説明しています。

- API Gateway を介して送信される REST リクエストを処理するために呼び出される Lambda 関数内のルートを、AWS Chalice を使って定義します。
- Lambda 関数を使用して、DynamoDB テーブルにデータを取得して保存し、REST リクエストを処理します。
- AWS CloudFormation テンプレート内のテーブル構造とセキュリティロールのリソースを定義します。
- AWS Chalice と CloudFormation を使用して、必要なすべてのリソースをパッケージ化してデプロイします。
- CloudFormation を使用して、作成されたすべてのリソースをクリーンアップします。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- CloudFormation
- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ステップ関数でメッセンジャーアプリケーションを作成する

次のコード例は、データベーステーブルからメッセージレコードを取得する AWS Step Functions メッセンジャーアプリケーションを作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を AWS Step Functions と使用して、Amazon DynamoDB テーブルからメッセージレコードを取得し、Amazon Simple Queue Service (Amazon SQS) で送信するメッセンジャーアプリケーションを作成する方法を紹介します。ステートマシンは AWS Lambda 関数を使用して、データベースで未送信メッセージをスキャンします。

- Amazon DynamoDB テーブルからメッセージレコードを取得および更新するステートマシンを作成します。
- ステートマシンの定義を更新して、Amazon Simple Queue Service (Amazon SQS) にもメッセージを送信します。
- ステートマシンの実行を開始および停止します。
- サービス統合を使用して、ステートマシンから Lambda、DynamoDB、および Amazon SQS に接続します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda
- Amazon SQS
- ステップ関数

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成

以下のコード例は、ユーザーがラベルを使用して写真を管理できるサーバーレスアプリケーションの作成方法を示しています。

.NET

SDK for .NET

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK で DynamoDB を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用してグローバルセカンダリインデックスを含む DynamoDB テーブルを作成する

次のコード例は、グローバルセカンダリインデックスを含むテーブルの作成方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してグローバルセカンダリインデックスを含む DynamoDB テーブルを作成します。

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableRequest;
import software.amazon.awssdk.services.dynamodb.model.DescribeTableResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.ProjectionType;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.waiters.DynamoDbWaiter;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public void createTable() {
        try {
            // Attribute definitions
```

```
            final List<AttributeDefinition> attributeDefinitions = new
ArrayList<>();
            attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(ISSUE_ID_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
            attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(TITLE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
            attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(CREATE_DATE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());
            attributeDefinitions.add(AttributeDefinition.builder()
                .attributeName(DUE_DATE_ATTR)
                .attributeType(ScalarAttributeType.S)
                .build());

            // Key schema for table
            final List<KeySchemaElement> tableKeySchema = new ArrayList<>();
            tableKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
                .keyType(KeyType.HASH)
                .build()); // Partition key
            tableKeySchema.add(KeySchemaElement.builder()
                .attributeName(TITLE_ATTR)
                .keyType(KeyType.RANGE)
                .build()); // Sort key

            // Initial provisioned throughput settings for the indexes
            final ProvisionedThroughput ptIndex = ProvisionedThroughput.builder()
                .readCapacityUnits(1L)
                .writeCapacityUnits(1L)
                .build();

            // CreateDateIndex
            final List<KeySchemaElement> createDateKeySchema = new ArrayList<>();
            createDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(CREATE_DATE_ATTR)
                .keyType(KeyType.HASH)
                .build());
            createDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
```

```
                .keyType(KeyType.RANGE)
                .build());

            final Projection createDateProjection = Projection.builder()
                .projectionType(ProjectionType.INCLUDE)
                .nonKeyAttributes(DESCRIPTION_ATTR, STATUS_ATTR)
                .build();

            final GlobalSecondaryIndex createDateIndex =
GlobalSecondaryIndex.builder()
                .indexName(CREATE_DATE_INDEX)
                .keySchema(createDateKeySchema)
                .projection(createDateProjection)
                .provisionedThroughput(ptIndex)
                .build();

            // TitleIndex
            final List<KeySchemaElement> titleKeySchema = new ArrayList<>();
            titleKeySchema.add(KeySchemaElement.builder()
                .attributeName(TITLE_ATTR)
                .keyType(KeyType.HASH)
                .build());
            titleKeySchema.add(KeySchemaElement.builder()
                .attributeName(ISSUE_ID_ATTR)
                .keyType(KeyType.RANGE)
                .build());

            final Projection titleProjection =

Projection.builder().projectionType(ProjectionType.KEYS_ONLY).build();

            final GlobalSecondaryIndex titleIndex =
GlobalSecondaryIndex.builder()
                .indexName(TITLE_INDEX)
                .keySchema(titleKeySchema)
                .projection(titleProjection)
                .provisionedThroughput(ptIndex)
                .build();

            // DueDateIndex
            final List<KeySchemaElement> dueDateKeySchema = new ArrayList<>();
            dueDateKeySchema.add(KeySchemaElement.builder()
                .attributeName(DUE_DATE_ATTR)
                .keyType(KeyType.HASH)
```

```
                .build());

            final Projection dueDateProjection =
                Projection.builder().projectionType(ProjectionType.ALL).build();

            final GlobalSecondaryIndex dueDateIndex =
GlobalSecondaryIndex.builder()
                .indexName(DUE_DATE_INDEX)
                .keySchema(dueDateKeySchema)
                .projection(dueDateProjection)
                .provisionedThroughput(ptIndex)
                .build();

            final CreateTableRequest createTableRequest =
CreateTableRequest.builder()
                .tableName(TABLE_NAME)
                .keySchema(tableKeySchema)
                .attributeDefinitions(attributeDefinitions)
                .globalSecondaryIndexes(createDateIndex, titleIndex,
dueDateIndex)
                .provisionedThroughput(ProvisionedThroughput.builder()
                    .readCapacityUnits(1L)
                    .writeCapacityUnits(1L)
                    .build())
                .build();

            System.out.println("Creating table " + TABLE_NAME + "...");
            dynamoDbClient.createTable(createTableRequest);

            // Wait for table to become active
            System.out.println("Waiting for " + TABLE_NAME + " to become
ACTIVE...");
            final DynamoDbWaiter waiter = dynamoDbClient.waiter();
            final DescribeTableRequest describeTableRequest =
                DescribeTableRequest.builder().tableName(TABLE_NAME).build();

            final WaiterResponse<DescribeTableResponse> waiterResponse =
                waiter.waitUntilTableExists(describeTableRequest);
            waiterResponse.matched().response().ifPresent(response ->
System.out.println("Table is now ready for use"));

        } catch (DynamoDbException e) {
            System.err.println("Error creating table: " + e.getMessage());
            e.printStackTrace();
```

```
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「CreateTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用してウォームスループット設定で DynamoDB テーブルを作成する

次のコード例は、ウォームスループットを有効にしたテーブルを作成する方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeDefinition;
import software.amazon.awssdk.services.dynamodb.model.CreateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.CreateTableResponse;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndex;
import software.amazon.awssdk.services.dynamodb.model.KeySchemaElement;
import software.amazon.awssdk.services.dynamodb.model.KeyType;
import software.amazon.awssdk.services.dynamodb.model.Projection;
import software.amazon.awssdk.services.dynamodb.model.ProvisionedThroughput;
import software.amazon.awssdk.services.dynamodb.model.ScalarAttributeType;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
```

```
    }

    /**
     * Builds a ProvisionedThroughput object with the specified read and write
capacity units.
     *
     * @param readCapacityUnits The read capacity units
     * @param writeCapacityUnits The write capacity units
     * @return A configured ProvisionedThroughput object
     */
    public static ProvisionedThroughput buildProvisionedThroughput(
        final Long readCapacityUnits, final Long writeCapacityUnits) {
        return ProvisionedThroughput.builder()
            .readCapacityUnits(readCapacityUnits)
            .writeCapacityUnits(writeCapacityUnits)
            .build();
    }

    /**
     * Builds an AttributeDefinition with the specified name and type.
     *
     * @param attributeName The attribute name
     * @param scalarAttributeType The attribute type
     * @return A configured AttributeDefinition
     */
    private static AttributeDefinition buildAttributeDefinition(
        final String attributeName, final ScalarAttributeType
scalarAttributeType) {
        return AttributeDefinition.builder()
            .attributeName(attributeName)
            .attributeType(scalarAttributeType)
            .build();
    }

    /**
     * Builds a KeySchemaElement with the specified name and key type.
     *
     * @param attributeName The attribute name
     * @param keyType The key type (HASH or RANGE)
     * @return A configured KeySchemaElement
     */
    private static KeySchemaElement buildKeySchemaElement(final String
attributeName, final KeyType keyType) {
        return KeySchemaElement.builder()
```

```
            .attributeName(attributeName)
            .keyType(keyType)
            .build();
    }

    /**
     * Creates a DynamoDB table with the specified configuration including warm
throughput settings.
     *
     * @param ddb The DynamoDB client
     * @param tableName The name of the table to create
     * @param partitionKey The partition key attribute name
     * @param sortKey The sort key attribute name
     * @param miscellaneousKeyAttribute Additional key attribute name for GSI
     * @param nonKeyAttribute Non-key attribute to include in GSI projection
     * @param tableReadCapacityUnits Read capacity units for the table
     * @param tableWriteCapacityUnits Write capacity units for the table
     * @param tableWarmReadUnitsPerSecond Warm read units per second for the
table
     * @param tableWarmWriteUnitsPerSecond Warm write units per second for the
table
     * @param globalSecondaryIndexName The name of the GSI to create
     * @param globalSecondaryIndexReadCapacityUnits Read capacity units for the
GSI
     * @param globalSecondaryIndexWriteCapacityUnits Write capacity units for the
GSI
     * @param globalSecondaryIndexWarmReadUnitsPerSecond Warm read units per
second for the GSI
     * @param globalSecondaryIndexWarmWriteUnitsPerSecond Warm write units per
second for the GSI
     */
    public static void createDynamoDBTable(
        final DynamoDbClient ddb,
        final String tableName,
        final String partitionKey,
        final String sortKey,
        final String miscellaneousKeyAttribute,
        final String nonKeyAttribute,
        final Long tableReadCapacityUnits,
        final Long tableWriteCapacityUnits,
        final Long tableWarmReadUnitsPerSecond,
        final Long tableWarmWriteUnitsPerSecond,
        final String globalSecondaryIndexName,
        final Long globalSecondaryIndexReadCapacityUnits,
```

```
        final Long globalSecondaryIndexWriteCapacityUnits,
        final Long globalSecondaryIndexWarmReadUnitsPerSecond,
        final Long globalSecondaryIndexWarmWriteUnitsPerSecond) {

        // Define the table attributes
        final AttributeDefinition partitionKeyAttribute =
buildAttributeDefinition(partitionKey, ScalarAttributeType.S);
        final AttributeDefinition sortKeyAttribute =
buildAttributeDefinition(sortKey, ScalarAttributeType.S);
        final AttributeDefinition miscellaneousKeyAttributeDefinition =
            buildAttributeDefinition(miscellaneousKeyAttribute,
ScalarAttributeType.N);
        final AttributeDefinition[] attributeDefinitions = {
            partitionKeyAttribute, sortKeyAttribute,
miscellaneousKeyAttributeDefinition
        };

        // Define the table key schema
        final KeySchemaElement partitionKeyElement =
buildKeySchemaElement(partitionKey, KeyType.HASH);
        final KeySchemaElement sortKeyElement = buildKeySchemaElement(sortKey,
KeyType.RANGE);
        final KeySchemaElement[] keySchema = {partitionKeyElement,
sortKeyElement};

        // Define the provisioned throughput for the table
        final ProvisionedThroughput provisionedThroughput =
            buildProvisionedThroughput(tableReadCapacityUnits,
tableWriteCapacityUnits);

        // Define the Global Secondary Index (GSI)
        final KeySchemaElement globalSecondaryIndexPartitionKeyElement =
buildKeySchemaElement(sortKey, KeyType.HASH);
        final KeySchemaElement globalSecondaryIndexSortKeyElement =
            buildKeySchemaElement(miscellaneousKeyAttribute, KeyType.RANGE);
        final KeySchemaElement[] gsiKeySchema = {
            globalSecondaryIndexPartitionKeyElement,
globalSecondaryIndexSortKeyElement
        };

        final Projection gsiProjection = Projection.builder()
            .projectionType(PROJECTION_TYPE_INCLUDE)
            .nonKeyAttributes(nonKeyAttribute)
            .build();
```

```
        final ProvisionedThroughput gsiProvisionedThroughput =
            buildProvisionedThroughput(globalSecondaryIndexReadCapacityUnits,
globalSecondaryIndexWriteCapacityUnits);

        // Define the warm throughput for the Global Secondary Index (GSI)
        final WarmThroughput gsiWarmThroughput = buildWarmThroughput(
            globalSecondaryIndexWarmReadUnitsPerSecond,
globalSecondaryIndexWarmWriteUnitsPerSecond);

        final GlobalSecondaryIndex globalSecondaryIndex =
GlobalSecondaryIndex.builder()
            .indexName(globalSecondaryIndexName)
            .keySchema(gsiKeySchema)
            .projection(gsiProjection)
            .provisionedThroughput(gsiProvisionedThroughput)
            .warmThroughput(gsiWarmThroughput)
            .build();

        // Define the warm throughput for the table
        final WarmThroughput tableWarmThroughput =
            buildWarmThroughput(tableWarmReadUnitsPerSecond,
tableWarmWriteUnitsPerSecond);

        final CreateTableRequest request = CreateTableRequest.builder()
            .tableName(tableName)
            .attributeDefinitions(attributeDefinitions)
            .keySchema(keySchema)
            .provisionedThroughput(provisionedThroughput)
            .globalSecondaryIndexes(globalSecondaryIndex)
            .warmThroughput(tableWarmThroughput)
            .build();

        final CreateTableResponse response = ddb.createTable(request);
        System.out.println(response);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「CreateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
import { DynamoDBClient, CreateTableCommand } from "@aws-sdk/client-dynamodb";

export async function createDynamoDBTableWithWarmThroughput(
  tableName,
  partitionKey,
  sortKey,
  miscKeyAttr,
  nonKeyAttr,
  tableProvisionedReadUnits,
  tableProvisionedWriteUnits,
  tableWarmReads,
  tableWarmWrites,
  indexName,
  indexProvisionedReadUnits,
  indexProvisionedWriteUnits,
  indexWarmReads,
  indexWarmWrites,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });
    const command = new CreateTableCommand({
      TableName: tableName,
      AttributeDefinitions: [
          { AttributeName: partitionKey, AttributeType: "S" },
          { AttributeName: sortKey, AttributeType: "S" },
          { AttributeName: miscKeyAttr, AttributeType: "N" },
      ],
      KeySchema: [
          { AttributeName: partitionKey, KeyType: "HASH" },
          { AttributeName: sortKey, KeyType: "RANGE" },
      ],
      ProvisionedThroughput: {
          ReadCapacityUnits: tableProvisionedReadUnits,
          WriteCapacityUnits: tableProvisionedWriteUnits,
      },
      WarmThroughput: {
```

```
            ReadUnitsPerSecond: tableWarmReads,
            WriteUnitsPerSecond: tableWarmWrites,
        },
        GlobalSecondaryIndexes: [
            {
              IndexName: indexName,
              KeySchema: [
                  { AttributeName: sortKey, KeyType: "HASH" },
                  { AttributeName: miscKeyAttr, KeyType: "RANGE" },
              ],
              Projection: {
                  ProjectionType: "INCLUDE",
                  NonKeyAttributes: [nonKeyAttr],
              },
              ProvisionedThroughput: {
                  ReadCapacityUnits: indexProvisionedReadUnits,
                  WriteCapacityUnits: indexProvisionedWriteUnits,
              },
              WarmThroughput: {
                  ReadUnitsPerSecond: indexWarmReads,
                  WriteUnitsPerSecond: indexWarmWrites,
              },
            },
        ],
      });
      const response = await ddbClient.send(command);
      console.log(response);
      return response;
    } catch (error) {
      console.error(`Error creating table: ${error}`);
      throw error;
    }
}

// Example usage (commented out for testing)
/*
createDynamoDBTableWithWarmThroughput(
  'example-table',
  'pk',
  'sk',
  'gsiKey',
  'data',
  10, 10, 5, 5,
  'example-index',
```

```
  5, 5, 2, 2
);
*/
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「CreateTable」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してウォームスループット設定を含む DynamoDB テーブルを作成します。

```
from boto3 import client
from botocore.exceptions import ClientError


def create_dynamodb_table_warm_throughput(
    table_name,
    partition_key,
    sort_key,
    misc_key_attr,
    non_key_attr,
    table_provisioned_read_units,
    table_provisioned_write_units,
    table_warm_reads,
    table_warm_writes,
    gsi_name,
    gsi_provisioned_read_units,
    gsi_provisioned_write_units,
    gsi_warm_reads,
    gsi_warm_writes,
    region_name="us-east-1",
):
    """
    Creates a DynamoDB table with a warm throughput setting configured.

    :param table_name: The name of the table to be created.
    :param partition_key: The partition key for the table being created.
```

```
    :param sort_key: The sort key for the table being created.
    :param misc_key_attr: A miscellaneous key attribute for the table being
created.
    :param non_key_attr: A non-key attribute for the table being created.
    :param table_provisioned_read_units: The newly created table's provisioned
read capacity units.
    :param table_provisioned_write_units: The newly created table's provisioned
write capacity units.
    :param table_warm_reads: The read units per second setting for the table's
warm throughput.
    :param table_warm_writes: The write units per second setting for the table's
warm throughput.
    :param gsi_name: The name of the Global Secondary Index (GSI) to be created
on the table.
    :param gsi_provisioned_read_units: The configured Global Secondary Index
(GSI) provisioned read capacity units.
    :param gsi_provisioned_write_units: The configured Global Secondary Index
(GSI) provisioned write capacity units.
    :param gsi_warm_reads: The read units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
    :param gsi_warm_writes: The write units per second setting for the Global
Secondary Index (GSI)'s warm throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    """
    try:
        ddb = client("dynamodb", region_name=region_name)

        # Define the table attributes
        attribute_definitions = [
            {"AttributeName": partition_key, "AttributeType": "S"},
            {"AttributeName": sort_key, "AttributeType": "S"},
            {"AttributeName": misc_key_attr, "AttributeType": "N"},
        ]

        # Define the table key schema
        key_schema = [
            {"AttributeName": partition_key, "KeyType": "HASH"},
            {"AttributeName": sort_key, "KeyType": "RANGE"},
        ]

        # Define the provisioned throughput for the table
        provisioned_throughput = {
            "ReadCapacityUnits": table_provisioned_read_units,
            "WriteCapacityUnits": table_provisioned_write_units,
```

```
        }

        # Define the global secondary index
        gsi_key_schema = [
            {"AttributeName": sort_key, "KeyType": "HASH"},
            {"AttributeName": misc_key_attr, "KeyType": "RANGE"},
        ]
        gsi_projection = {"ProjectionType": "INCLUDE", "NonKeyAttributes":
[non_key_attr]}
        gsi_provisioned_throughput = {
            "ReadCapacityUnits": gsi_provisioned_read_units,
            "WriteCapacityUnits": gsi_provisioned_write_units,
        }
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_warm_reads,
            "WriteUnitsPerSecond": gsi_warm_writes,
        }
        global_secondary_indexes = [
            {
                "IndexName": gsi_name,
                "KeySchema": gsi_key_schema,
                "Projection": gsi_projection,
                "ProvisionedThroughput": gsi_provisioned_throughput,
                "WarmThroughput": gsi_warm_throughput,
            }
        ]

        # Define the warm throughput for the table
        warm_throughput = {
            "ReadUnitsPerSecond": table_warm_reads,
            "WriteUnitsPerSecond": table_warm_writes,
        }

        # Create the DynamoDB client and create the table
        response = ddb.create_table(
            TableName=table_name,
            AttributeDefinitions=attribute_definitions,
            KeySchema=key_schema,
            ProvisionedThroughput=provisioned_throughput,
            GlobalSecondaryIndexes=global_secondary_indexes,
            WarmThroughput=warm_throughput,
        )

        print(response)
```

```
        return response
    except ClientError as e:
        print(f"Error creating table: {e}")
        raise e
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の「CreateTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB データを追跡するウェブアプリケーションを作成する

次のコードは、Amazon DynamoDB の作業項目を追跡し、Amazon Simple Email Service (Amazon SES) を使用してレポートを送信するウェブアプリケーションを作成する方法を示します。

.NET

SDK for .NET

Amazon DynamoDB .NET API を使用して、DynamoDB 作業データを追跡する動的ウェブアプリケーションを作成する方法を示しています。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon SES

Java

SDK for Java 2.x

Amazon DynamoDB API を使用して、DynamoDB 作業データを追跡する動的ウェブアプリケーションを作成する方法を示しています。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon SES

Kotlin

SDK for Kotlin

Amazon DynamoDB API を使用して、DynamoDB 作業データを追跡する動的ウェブアプリケーションを作成する方法を示しています。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon SES

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して Amazon DynamoDB の作業項目を追跡し、Amazon Simple Email Service (Amazon SES) を使用してレポートを E メールで送信する REST サービスを作成する方法を示します。この例では、Flask ウェブフレームワークを使用して HTTP ルーティングを処理し、React ウェブページと統合して完全に機能するウェブアプリケーションを提供しています。

- AWS のサービス と統合できる Flask REST サービスを構築します。
- DynamoDB テーブルに保存されている作業項目の読み取り、書き込み、更新を行います。
- Amazon SES を使用して作業項目のレポートを E メールで送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub の「AWS コード例のレポジトリ」で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon SES

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

API Gateway で WebSocket チャットアプリケーションを作成する

次のコード例は、Amazon API Gateway 上に構築された WebSocket API によって提供されるチャットアプリケーションを作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) と Amazon API Gateway V2 を使用して、AWS Lambda や Amazon DynamoDB と統合する WebSocket API を作成する方法を示します。

- API Gateway で提供される WebSocket API を作成します。
- DynamoDB に接続を保存し、他のチャット参加者にメッセージを投稿する Lambda ハンドラを定義します。
- WebSocket チャットアプリケーションに接続し、WebSockets パッケージを使用してメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して TTL を含む DynamoDB 項目を作成する

次のコード例は、TTL を含む項目の作成方法を示しています。

Java

SDK for Java 2.x

```
package com.amazon.samplelib.ttl;

import com.amazon.samplelib.CodeSampleUtils;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

/**
 * Creates an item in a DynamoDB table with TTL attributes.
 * This class demonstrates how to add TTL expiration timestamps to DynamoDB
 items.
 */
public class CreateTTL {

    private static final String USAGE =
        """
            Usage:
                <tableName> <primaryKey> <sortKey> <region>
            Where:
                tableName - The Amazon DynamoDB table being queried.
                primaryKey - The name of the primary key. Also known as the hash
 or partition key.
                sortKey - The name of the sort key. Also known as the range
 attribute.
                region (optional) - The AWS region that the Amazon DynamoDB table
 is located in. (Default: us-east-1)
```

```
            """;
    private static final int DAYS_TO_EXPIRE = 90;
    private static final int SECONDS_PER_DAY = 24 * 60 * 60;
    private static final String PRIMARY_KEY_ATTR = "primaryKey";
    private static final String SORT_KEY_ATTR = "sortKey";
    private static final String CREATION_DATE_ATTR = "creationDate";
    private static final String EXPIRE_AT_ATTR = "expireAt";
    private static final String SUCCESS_MESSAGE = "%s PutItem operation with TTL
successful.";
    private static final String TABLE_NOT_FOUND_ERROR = "Error: The Amazon
DynamoDB table \"%s\" can't be found.";

    private final DynamoDbClient dynamoDbClient;

    /**
     * Constructs a CreateTTL instance with the specified DynamoDB client.
     *
     * @param dynamoDbClient The DynamoDB client to use
     */
    public CreateTTL(final DynamoDbClient dynamoDbClient) {
        this.dynamoDbClient = dynamoDbClient;
    }

    /**
     * Constructs a CreateTTL with a default DynamoDB client.
     */
    public CreateTTL() {
        this.dynamoDbClient = null;
    }

    /**
     * Main method to demonstrate creating an item with TTL.
     *
     * @param args Command line arguments
     */
    public static void main(final String[] args) {
        try {
            int result = new CreateTTL().processArgs(args);
            System.exit(result);
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

```
   /**
    * Process command line arguments and create an item with TTL.
    *
    * @param args Command line arguments
    * @return 0 if successful, non-zero otherwise
    * @throws ResourceNotFoundException If the table doesn't exist
    * @throws DynamoDbException If an error occurs during the operation
    * @throws IllegalArgumentException If arguments are invalid
    */
   public int processArgs(final String[] args) {
       // Argument validation (remove or replace this line when reusing this
code)
       CodeSampleUtils.validateArgs(args, new int[] {3, 4}, USAGE);

       final String tableName = args[0];
       final String primaryKey = args[1];
       final String sortKey = args[2];
       final Region region = Optional.ofNullable(args.length > 3 ? args[3] :
null)
           .map(Region::of)
           .orElse(Region.US_EAST_1);

       try (DynamoDbClient ddb = dynamoDbClient != null
           ? dynamoDbClient
           : DynamoDbClient.builder().region(region).build()) {
           final CreateTTL createTTL = new CreateTTL(ddb);
           createTTL.createItemWithTTL(tableName, primaryKey, sortKey);
           return 0;
       } catch (Exception e) {
           throw e;
       }
   }

   /**
    * Creates an item in the specified table with TTL attributes.
    *
    * @param tableName The name of the table
    * @param primaryKeyValue The value for the primary key
    * @param sortKeyValue The value for the sort key
    * @return The response from the PutItem operation
    * @throws ResourceNotFoundException If the table doesn't exist
    * @throws DynamoDbException If an error occurs during the operation
    */
```

```
    public PutItemResponse createItemWithTTL(
        final String tableName, final String primaryKeyValue, final String
 sortKeyValue) {
        // Get current time in epoch second format
        final long createDate = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = createDate + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        final Map<String, AttributeValue> itemMap = new HashMap<>();
        itemMap.put(
            PRIMARY_KEY_ATTR,
 AttributeValue.builder().s(primaryKeyValue).build());
        itemMap.put(SORT_KEY_ATTR,
 AttributeValue.builder().s(sortKeyValue).build());
        itemMap.put(
            CREATION_DATE_ATTR,
            AttributeValue.builder().n(String.valueOf(createDate)).build());
        itemMap.put(
            EXPIRE_AT_ATTR,
            AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final PutItemRequest request =
            PutItemRequest.builder().tableName(tableName).item(itemMap).build();

        try {
            final PutItemResponse response = dynamoDbClient.putItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「PutItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

export function createDynamoDBItem(table_name, region, partition_key, sort_key) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    // Get the current time in epoch second format
    const current_time = Math.floor(new Date().getTime() / 1000);

    // Calculate the expireAt time (90 days from now) in epoch second format
    const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 *
 1000) / 1000);

    // Create DynamoDB item
    const item = {
        'partitionKey': {'S': partition_key},
        'sortKey': {'S': sort_key},
        'createdAt': {'N': current_time.toString()},
        'expireAt': {'N': expire_at.toString()}
    };

    const putItemCommand = new PutItemCommand({
        TableName: table_name,
        Item: item,
        ProvisionedThroughput: {
            ReadCapacityUnits: 1,
            WriteCapacityUnits: 1,
        },
    });

    client.send(putItemCommand, function(err, data) {
        if (err) {
            console.log("Exception encountered when creating item %s, here's what
 happened: ", data, err);
            throw err;
        } else {
            console.log("Item created successfully: %s.", data);
            return data;
```

```
        }
    });
}

// Example usage (commented out for testing)
// createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
 'your-sort-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「PutItem」を参照してください。

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def create_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Creates a DynamoDB item with an attached expiry attribute.

    :param table_name: Table name for the boto3 resource to target when creating
 an item
    :param region: string representing the AWS region. Example: `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expiration time (90 days from now) in epoch second format
        expiration_time = int((datetime.now() + timedelta(days=90)).timestamp())

        item = {
```

```
            "primaryKey": primary_key,
            "sortKey": sort_key,
            "creationDate": current_time,
            "expireAt": expiration_time,
        }
        response = table.put_item(Item=item)

        print("Item created successfully.")
        return response
    except Exception as e:
        print(f"Error creating item: {e}")
        raise e


# Use your own values
create_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「PutItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して、DynamoDB でマルチリージョンの強力な整合性を備えたグローバルテーブルを作成して管理する

次のコード例は、マルチリージョンの強力な整合性 (MRSC) を備えた DynamoDB グローバルテーブルを作成および管理する方法を示しています。

- マルチリージョンの強力な整合性を備えたテーブルを作成します。
- MRSC 設定とレプリカのステータスを確認します。
- 即時読み取りでリージョン間の強力な整合性をテストします。
- MRSC 保証を使用して条件付き書き込みを実行します。
- MRSC グローバルテーブルリソースをクリーンアップします。

Bash

AWS CLIBash スクリプトを使用した

マルチリージョンの強力な整合性を備えたテーブルを作成します。

```
# Step 1: Create a new table in us-east-2 (primary region for MRSC)
# Note: Table must be empty when enabling MRSC
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --region us-east-2

# Wait for table to become active
aws dynamodb wait table-exists --table-name MusicTable --region us-east-2

# Step 2: Add replica and witness with Multi-Region Strong Consistency
# MRSC requires exactly three replicas in supported regions
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Create": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Create": {"RegionName": "us-west-2"}}]' \
    --multi-region-consistency STRONG \
    --region us-east-2
```

MRSC 設定とレプリカのステータスを確認します。

```
# Verify the global table configuration and MRSC setting
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

リージョン間の即時読み取りの強力な整合性をテストします。

```
# Write an item to the primary region
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"},"Album":
 {"S":"The Beatles 1967-1970"},"Year": {"N":"1968"}}' \
    --region us-east-2

# Read the item from replica region to verify strong consistency (cannot read or
 write to witness)
# No wait time needed - MRSC provides immediate consistency
echo "Reading from us-east-1 (immediate consistency):"
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --consistent-read \
    --region us-east-1
```

MRSC 保証を使用して条件付き書き込みを実行します。

```
# Perform a conditional update from a different region
# This demonstrates that conditions work consistently across all regions
aws dynamodb update-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"The Beatles"},"SongTitle": {"S":"Hey Jude"}}' \
    --update-expression "SET #rating = :rating" \
    --condition-expression "attribute_exists(Artist)" \
    --expression-attribute-names '{"#rating": "Rating"}' \
    --expression-attribute-values '{":rating": {"N":"5"}}' \
    --region us-east-1
```

MRSC グローバルテーブルリソースをクリーンアップします。

```
# Remove replica tables (must be done before deleting the primary table)
aws dynamodb update-table \
    --table-name MusicTable \
    --replica-updates '[{"Delete": {"RegionName": "us-east-1"}}]' \
    --global-table-witness-updates '[{"Delete": {"RegionName": "us-west-2"}}]' \
    --region us-east-2
```

```
# Wait for replicas to be deleted
echo "Waiting for replicas to be deleted..."
sleep 30

# Delete the primary table
aws dynamodb delete-table \
    --table-name MusicTable \
    --region us-east-2
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateItem
  - UpdateTable

Java

SDK for Java 2.x

MRSC 変換の準備ができたリージョンテーブルを AWS SDK for Java 2.x を使用して作成します。

```
   public static CreateTableResponse createRegionalTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
```

```
        try {
            LOGGER.info("Creating regional table: " + tableName + " (must be
empty for MRSC)");

            CreateTableRequest createTableRequest = CreateTableRequest.builder()
                .tableName(tableName)
                .attributeDefinitions(
                    AttributeDefinition.builder()
                        .attributeName("Artist")
                        .attributeType(ScalarAttributeType.S)
                        .build(),
                    AttributeDefinition.builder()
                        .attributeName("SongTitle")
                        .attributeType(ScalarAttributeType.S)
                        .build())
                .keySchema(
                    KeySchemaElement.builder()
                        .attributeName("Artist")
                        .keyType(KeyType.HASH)
                        .build(),
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Regional table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create regional table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to create regional table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、リージョンテーブルをレプリカと監視を備えた MRSC に変換します。

```
    public static UpdateTableResponse convertToMRSCWithWitness(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Converting table to MRSC with replica in " +
replicaRegion.id() + " and witness in "
                + witnessRegion.id());

            // Create replica update using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .create(CreateReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Create witness update
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .create(CreateGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
```

```
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .multiRegionConsistency(MultiRegionConsistency.STRONG)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC conversion initiated. Status: "
                + response.tableDescription().tableStatus());
            LOGGER.info("UpdateTableResponse full object: " + response);
            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to convert table to MRSC: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to convert table to MRSC: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC のグローバルテーブル設定を記述します。

```
    public static DescribeTableResponse describeMRSCTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing MRSC global table: " + tableName);
```

```
            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            LOGGER.info("Multi-region consistency: " +
response.table().multiRegionConsistency());

            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            if (response.table().globalTableWitnesses() != null
                && !response.table().globalTableWitnesses().isEmpty()) {
                LOGGER.info("Number of witnesses: "
                    + response.table().globalTableWitnesses().size());
                response.table()
                    .globalTableWitnesses()
                    .forEach(witness -> LOGGER.info(
                        "Witness region: " + witness.regionName() + ", Status: "
+ witness.witnessStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Table not found: " + tableName)
                .cause(e)
                .build();
        } catch (DynamoDbException e) {
```

```
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to describe table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC の強力な整合性を検証するためのテスト項目を追加します。

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final String artist,
        final String songTitle,
        final String album,
        final String year) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to MRSC global table: " + tableName);

            Map<String, AttributeValue> item = new HashMap<>();
            item.put("Artist", AttributeValue.builder().s(artist).build());
            item.put("SongTitle", AttributeValue.builder().s(songTitle).build());
```

```
            if (album != null && !album.trim().isEmpty()) {
                item.put("Album", AttributeValue.builder().s(album).build());
            }
            if (year != null && !year.trim().isEmpty()) {
                item.put("Year", AttributeValue.builder().n(year).build());
            }

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully with strong consistency");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to add test item to table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカから一貫した読み取りを持つ項目を読み取ります。

```
    public static GetItemResponse getItemWithConsistentRead(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
```

```
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Reading item from MRSC global table with consistent
read: " + tableName);

            Map<String, AttributeValue> key = new HashMap<>();
            key.put("Artist", AttributeValue.builder().s(artist).build());
            key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

            GetItemRequest getItemRequest = GetItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .consistentRead(true)
                .build();

            GetItemResponse response = dynamoDbClient.getItem(getItemRequest);

            if (response.hasItem()) {
                LOGGER.info("Item found with strong consistency - no wait time
needed");
            } else {
                LOGGER.info("Item not found");
            }

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to read item from table: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to read item from table: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC 保証を使用する条件付きの更新を実行します。

```
   public static UpdateItemResponse performConditionalUpdate(
       final DynamoDbClient dynamoDbClient,
       final String tableName,
       final String artist,
       final String songTitle,
       final String rating) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (artist == null || artist.trim().isEmpty()) {
           throw new IllegalArgumentException("Artist cannot be null or empty");
       }
       if (songTitle == null || songTitle.trim().isEmpty()) {
           throw new IllegalArgumentException("Song title cannot be null or
empty");
       }
       if (rating == null || rating.trim().isEmpty()) {
           throw new IllegalArgumentException("Rating cannot be null or empty");
       }

       try {
           LOGGER.info("Performing conditional update on MRSC global table: " +
tableName);

           Map<String, AttributeValue> key = new HashMap<>();
           key.put("Artist", AttributeValue.builder().s(artist).build());
           key.put("SongTitle", AttributeValue.builder().s(songTitle).build());

           Map<String, String> expressionAttributeNames = new HashMap<>();
           expressionAttributeNames.put("#rating", "Rating");

           Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
           expressionAttributeValues.put(
               ":rating", AttributeValue.builder().n(rating).build());

           UpdateItemRequest updateItemRequest = UpdateItemRequest.builder()
               .tableName(tableName)
```

```
                .key(key)
                .updateExpression("SET #rating = :rating")
                .conditionExpression("attribute_exists(Artist)")
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .build();

            UpdateItemResponse response =
dynamoDbClient.updateItem(updateItemRequest);
            LOGGER.info("Conditional update successful - demonstrates strong
consistency");

            return response;

        } catch (ConditionalCheckFailedException e) {
            LOGGER.warning("Conditional check failed: " + e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to perform conditional update: " + tableName +
" - " + e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to perform conditional update: " + tableName)
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカと監視がアクティブになるまで待ちます。

```
    public static void waitForMRSCReplicasActive(
        final DynamoDbClient dynamoDbClient, final String tableName, final int
maxWaitTimeSeconds)
        throws InterruptedException {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
```

```
        if (maxWaitTimeSeconds <= 0) {
            throw new IllegalArgumentException("Max wait time must be positive");
        }

        try {
            LOGGER.info("Waiting for MRSC replicas and witnesses to become
active: " + tableName);

            final long startTime = System.currentTimeMillis();
            final long maxWaitTimeMillis = maxWaitTimeSeconds * 1000L;
            int backoffSeconds = 5; // Start with 5 second intervals
            final int maxBackoffSeconds = 30; // Cap at 30 seconds

            while (System.currentTimeMillis() - startTime < maxWaitTimeMillis) {
                DescribeTableResponse response =
describeMRSCTable(dynamoDbClient, tableName);

                boolean allActive = true;
                StringBuilder statusReport = new StringBuilder();

                if (response.table().multiRegionConsistency() == null
                    || !MultiRegionConsistency.STRONG
                        .toString()

.equals(response.table().multiRegionConsistency().toString())) {
                    allActive = false;
                    statusReport
                        .append("MultiRegionConsistency: ")
                        .append(response.table().multiRegionConsistency())
                        .append(" ");
                }
                if (response.table().replicas() == null
                    || response.table().replicas().isEmpty()) {
                    allActive = false;
                    statusReport.append("No replicas found. ");
                }
                if (response.table().globalTableWitnesses() == null
                    || response.table().globalTableWitnesses().isEmpty()) {
                    allActive = false;
                    statusReport.append("No witnesses found. ");
                }

                // Check table status
```

```
                if (!"ACTIVE".equals(response.table().tableStatus().toString()))
{
                    allActive = false;
                    statusReport
                        .append("Table: ")
                        .append(response.table().tableStatus())
                        .append(" ");
                }

                // Check replica status
                if (response.table().replicas() != null) {
                    for (var replica : response.table().replicas()) {
                        if (!"ACTIVE".equals(replica.replicaStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Replica(")
                                .append(replica.regionName())
                                .append("): ")
                                .append(replica.replicaStatus())
                                .append(" ");
                        }
                    }
                }

                // Check witness status
                if (response.table().globalTableWitnesses() != null) {
                    for (var witness : response.table().globalTableWitnesses()) {
                        if (!"ACTIVE".equals(witness.witnessStatus().toString()))
{
                            allActive = false;
                            statusReport
                                .append("Witness(")
                                .append(witness.regionName())
                                .append("): ")
                                .append(witness.witnessStatus())
                                .append(" ");
                        }
                    }
                }

                if (allActive) {
                    LOGGER.info("All MRSC replicas and witnesses are now active:
" + tableName);
```

```
                    return;
                }

                LOGGER.info("Waiting for MRSC components to become active.
Status: " + statusReport.toString());
                LOGGER.info("Next check in " + backoffSeconds + " seconds...");

                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, maxBackoffSeconds);
            }

            throw DynamoDbException.builder()
                .message("Timeout waiting for MRSC replicas to become active
after " + maxWaitTimeSeconds + " seconds")
                .build();

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("Failed to wait for MRSC replicas to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC レプリカと監視をクリーンアップします。

```
    public static UpdateTableResponse cleanupMRSCReplicas(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
```

```
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("Cleaning up MRSC replicas and witnesses for table: " +
tableName);

            // Remove replica using ReplicationGroupUpdate
            ReplicationGroupUpdate replicaUpdate =
ReplicationGroupUpdate.builder()
                .delete(DeleteReplicationGroupMemberAction.builder()
                    .regionName(replicaRegion.id())
                    .build())
                .build();

            // Remove witness
            GlobalTableWitnessGroupUpdate witnessUpdate =
GlobalTableWitnessGroupUpdate.builder()
                .delete(DeleteGlobalTableWitnessGroupMemberAction.builder()
                    .regionName(witnessRegion.id())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(List.of(replicaUpdate))
                .globalTableWitnessUpdates(List.of(witnessUpdate))
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("MRSC cleanup initiated - removing replica and witness.
Response: " + response);

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to cleanup MRSC replicas: " + tableName + " - "
+ e.getMessage());
            throw DynamoDbException.builder()
                .message("Failed to cleanup MRSC replicas: " + tableName)
                .cause(e)
```

```
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用して、MRSC ワークフローのデモンストレーションを完了します。

```
    public static void demonstrateCompleteMRSCWorkflow(
        final DynamoDbClient primaryClient,
        final DynamoDbClient replicaClient,
        final String tableName,
        final Region replicaRegion,
        final Region witnessRegion)
        throws InterruptedException {

        if (primaryClient == null) {
            throw new IllegalArgumentException("Primary DynamoDB client cannot be
null");
        }
        if (replicaClient == null) {
            throw new IllegalArgumentException("Replica DynamoDB client cannot be
null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (witnessRegion == null) {
            throw new IllegalArgumentException("Witness region cannot be null");
        }

        try {
            LOGGER.info("=== Starting Complete MRSC Workflow Demonstration ===");

            // Step 1: Create an empty single-Region table
            LOGGER.info("Step 1: Creating empty single-Region table");
            createRegionalTable(primaryClient, tableName);
```

```
            // Use the existing GlobalTableOperations method for basic table
waiting
            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 2: Convert to MRSC with replica and witness
            LOGGER.info("Step 2: Converting to MRSC with replica and witness");
            convertToMRSCWithWitness(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for MRSC conversion to complete using MRSC-specific waiter
            LOGGER.info("Waiting for MRSC conversion to complete...");
            waitForMRSCReplicasActive(primaryClient, tableName);

            LOGGER.info("Intermediate step: Waiting for table [" + tableName + "]
to become active before continuing");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // Step 3: Verify MRSC configuration
            LOGGER.info("Step 3: Verifying MRSC configuration");
            describeMRSCTable(primaryClient, tableName);

            // Step 4: Test strong consistency with data operations
            LOGGER.info("Step 4: Testing strong consistency with data
operations");

            // Add test item to primary region
            putTestItem(primaryClient, tableName, "The Beatles", "Hey Jude", "The
Beatles 1967-1970", "1968");

            // Immediately read from replica region (no wait needed with MRSC)
            LOGGER.info("Reading from replica region immediately (strong
consistency):");
            GetItemResponse getResponse =
                getItemWithConsistentRead(replicaClient, tableName, "The
Beatles", "Hey Jude");

            if (getResponse.hasItem()) {
                LOGGER.info("# Strong consistency verified - item immediately
available in replica region");
            } else {
                LOGGER.warning("# Item not found in replica region");
            }
```

```
            // Test conditional update from replica region
            LOGGER.info("Testing conditional update from replica region:");
            performConditionalUpdate(replicaClient, tableName, "The Beatles",
"Hey Jude", "5");
            LOGGER.info("# Conditional update successful - demonstrates strong
consistency");

            // Step 5: Cleanup
            LOGGER.info("Step 5: Cleaning up resources");
            cleanupMRSCReplicas(primaryClient, tableName, replicaRegion,
witnessRegion);

            // Wait for cleanup to complete using basic table waiter
            LOGGER.info("Waiting for replica cleanup to complete...");
            GlobalTableOperations.waitForTableActive(primaryClient, tableName);

            // "Halt" until replica/witness cleanup is complete
            DescribeTableResponse cleanupVerification =
describeMRSCTable(primaryClient, tableName);
            int backoffSeconds = 5; // Start with 5 second intervals
            while (cleanupVerification.table().multiRegionConsistency() != null)
{
                LOGGER.info("Waiting additional time (" + backoffSeconds + "
seconds) for MRSC cleanup to complete...");
                tempWait(backoffSeconds);

                // Exponential backoff with cap
                backoffSeconds = Math.min(backoffSeconds * 2, 30);
                cleanupVerification = describeMRSCTable(primaryClient,
tableName);
            }

            // Delete the primary table
            deleteTable(primaryClient, tableName);

            LOGGER.info("=== MRSC Workflow Demonstration Complete ===");
            LOGGER.info("");
            LOGGER.info("Key benefits of Multi-Region Strong Consistency
(MRSC):");
            LOGGER.info("- Immediate consistency across all regions (no eventual
consistency delays)");
            LOGGER.info("- Simplified application logic (no need to handle
eventual consistency)");
```

```
            LOGGER.info("- Support for conditional writes and transactions across
regions");
            LOGGER.info("- Consistent read operations from any region without
waiting");

        } catch (DynamoDbException | InterruptedException e) {
            LOGGER.severe("MRSC workflow failed: " + e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateItem
  - UpdateTable

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した MREC を示す DynamoDB グローバルテーブルの作成と管理

次のコード例は、複数のリージョンにまたがるレプリカを含む DynamoDB グローバルテーブルを作成して管理する方法を示しています。

- グローバルセカンダリインデックスおよび DynamoDB Streams を含むテーブルを作成します。
- 異なるリージョンにレプリカを追加して、グローバルテーブルを作成します。
- グローバルテーブルからレプリカを削除します。
- テスト項目を追加して、リージョン間のレプリケーションを検証します。

- グローバルテーブル設定とレプリカのステータスについて説明します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して、グローバルセカンダリインデックスおよび DynamoDB Streams を含むテーブルを作成します。

```
   public static CreateTableResponse createTableWithGSI(
       final DynamoDbClient dynamoDbClient, final String tableName, final String
indexName) {

       if (dynamoDbClient == null) {
           throw new IllegalArgumentException("DynamoDB client cannot be null");
       }
       if (tableName == null || tableName.trim().isEmpty()) {
           throw new IllegalArgumentException("Table name cannot be null or
empty");
       }
       if (indexName == null || indexName.trim().isEmpty()) {
           throw new IllegalArgumentException("Index name cannot be null or
empty");
       }

       try {
           LOGGER.info("Creating table: " + tableName + " with GSI: " +
indexName);

           CreateTableRequest createTableRequest = CreateTableRequest.builder()
               .tableName(tableName)
               .attributeDefinitions(
                   AttributeDefinition.builder()
                       .attributeName("Artist")
                       .attributeType(ScalarAttributeType.S)
                       .build(),
                   AttributeDefinition.builder()
                       .attributeName("SongTitle")
                       .attributeType(ScalarAttributeType.S)
                       .build())
               .keySchema(
                   KeySchemaElement.builder()
                       .attributeName("Artist")
```

```
                        .keyType(KeyType.HASH)
                        .build(),
                    KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.RANGE)
                        .build())
                .billingMode(BillingMode.PAY_PER_REQUEST)
                .globalSecondaryIndexes(GlobalSecondaryIndex.builder()
                    .indexName(indexName)
                    .keySchema(KeySchemaElement.builder()
                        .attributeName("SongTitle")
                        .keyType(KeyType.HASH)
                        .build())
                    .projection(

Projection.builder().projectionType(ProjectionType.ALL).build())
                    .build())
                .streamSpecification(StreamSpecification.builder()
                    .streamEnabled(true)
                    .streamViewType(StreamViewType.NEW_AND_OLD_IMAGES)
                    .build())
                .build();

            CreateTableResponse response =
dynamoDbClient.createTable(createTableRequest);
            LOGGER.info("Table creation initiated. Status: "
                + response.tableDescription().tableStatus());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to create table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、テーブルがアクティブになるまで待ちます。

```
    public static void waitForTableActive(final DynamoDbClient dynamoDbClient,
final String tableName) {
```

```
        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Waiting for table to become active: " + tableName);

            try (DynamoDbWaiter waiter =
                DynamoDbWaiter.builder().client(dynamoDbClient).build()) {
                DescribeTableRequest request =
                    DescribeTableRequest.builder().tableName(tableName).build();

                waiter.waitUntilTableExists(request);
                LOGGER.info("Table is now active: " + tableName);
            }

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to wait for table to become active: " +
tableName + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、レプリカを追加して、グローバルテーブルを作成または拡張します。

```
    public static UpdateTableResponse addReplica(
        final DynamoDbClient dynamoDbClient,
        final String tableName,
        final Region replicaRegion,
        final String indexName,
        final Long readCapacity) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
```

```
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }
        if (indexName == null || indexName.trim().isEmpty()) {
            throw new IllegalArgumentException("Index name cannot be null or
empty");
        }
        if (readCapacity == null || readCapacity <= 0) {
            throw new IllegalArgumentException("Read capacity must be a positive
number");
        }

        try {
            LOGGER.info("Adding replica in region: " + replicaRegion.id() + " for
table: " + tableName);

            // Create a ReplicationGroupUpdate for adding a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .create(builder -> builder.regionName(replicaRegion.id())
                    .globalSecondaryIndexes(ReplicaGlobalSecondaryIndex.builder()
                        .indexName(indexName)

.provisionedThroughputOverride(ProvisionedThroughputOverride.builder()
                            .readCapacityUnits(readCapacity)
                            .build())
                        .build())
                    .build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();

            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica addition initiated in region: " +
replicaRegion.id());

            return response;
```

```
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用してグローバルテーブルからレプリカを削除します。

```
    public static UpdateTableResponse removeReplica(
        final DynamoDbClient dynamoDbClient, final String tableName, final Region
replicaRegion) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (replicaRegion == null) {
            throw new IllegalArgumentException("Replica region cannot be null");
        }

        try {
            LOGGER.info("Removing replica in region: " + replicaRegion.id() + "
for table: " + tableName);

            // Create a ReplicationGroupUpdate for removing a replica
            ReplicationGroupUpdate replicationGroupUpdate =
ReplicationGroupUpdate.builder()
                .delete(builder ->
builder.regionName(replicaRegion.id()).build())
                .build();

            UpdateTableRequest updateTableRequest = UpdateTableRequest.builder()
                .tableName(tableName)
                .replicaUpdates(replicationGroupUpdate)
                .build();
```

```
            UpdateTableResponse response =
dynamoDbClient.updateTable(updateTableRequest);
            LOGGER.info("Replica removal initiated in region: " +
replicaRegion.id());

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to remove replica in region: " +
replicaRegion.id() + " - " + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用してテスト項目を追加して、レプリケーションを検証します。

```
    public static PutItemResponse putTestItem(
        final DynamoDbClient dynamoDbClient, final String tableName, final String
artist, final String songTitle) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }
        if (artist == null || artist.trim().isEmpty()) {
            throw new IllegalArgumentException("Artist cannot be null or empty");
        }
        if (songTitle == null || songTitle.trim().isEmpty()) {
            throw new IllegalArgumentException("Song title cannot be null or
empty");
        }

        try {
            LOGGER.info("Adding test item to table: " + tableName);

            Map<String,
software.amazon.awssdk.services.dynamodb.model.AttributeValue> item = new
HashMap<>();
            item.put(
```

```
                "Artist",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(artist)
                    .build());
            item.put(
                "SongTitle",

software.amazon.awssdk.services.dynamodb.model.AttributeValue.builder()
                    .s(songTitle)
                    .build());

            PutItemRequest putItemRequest =
                PutItemRequest.builder().tableName(tableName).item(item).build();

            PutItemResponse response = dynamoDbClient.putItem(putItemRequest);
            LOGGER.info("Test item added successfully");

            return response;

        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to add test item to table: " + tableName + " -
" + e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用して、グローバルテーブル設定とレプリカを記述します。

```
    public static DescribeTableResponse describeTable(final DynamoDbClient
dynamoDbClient, final String tableName) {

        if (dynamoDbClient == null) {
            throw new IllegalArgumentException("DynamoDB client cannot be null");
        }
        if (tableName == null || tableName.trim().isEmpty()) {
            throw new IllegalArgumentException("Table name cannot be null or
empty");
        }

        try {
            LOGGER.info("Describing table: " + tableName);
```

```
            DescribeTableRequest request =
                DescribeTableRequest.builder().tableName(tableName).build();

            DescribeTableResponse response =
dynamoDbClient.describeTable(request);

            LOGGER.info("Table status: " + response.table().tableStatus());
            if (response.table().replicas() != null
                && !response.table().replicas().isEmpty()) {
                LOGGER.info("Number of replicas: " +
response.table().replicas().size());
                response.table()
                    .replicas()
                    .forEach(replica -> LOGGER.info(
                        "Replica region: " + replica.regionName() + ", Status: "
+ replica.replicaStatus()));
            }

            return response;

        } catch (ResourceNotFoundException e) {
            LOGGER.severe("Table not found: " + tableName + " - " +
e.getMessage());
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.severe("Failed to describe table: " + tableName + " - " +
e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x を使用したグローバルテーブル操作の完全な例。

```
    public static void exampleUsage(final Region sourceRegion, final Region
replicaRegion) {

        String tableName = "Music";
        String indexName = "SongTitleIndex";
        Long readCapacity = 15L;

        // Create DynamoDB client for the source region
```

```
        try (DynamoDbClient dynamoDbClient =
            DynamoDbClient.builder().region(sourceRegion).build()) {

            try {
                // Step 1: Create the initial table with GSI and streams
                LOGGER.info("Step 1: Creating table in source region: " +
sourceRegion.id());
                createTableWithGSI(dynamoDbClient, tableName, indexName);

                // Step 2: Wait for table to become active
                LOGGER.info("Step 2: Waiting for table to become active");
                waitForTableActive(dynamoDbClient, tableName);

                // Step 3: Add replica in destination region
                LOGGER.info("Step 3: Adding replica in region: " +
replicaRegion.id());
                addReplica(dynamoDbClient, tableName, replicaRegion, indexName,
readCapacity);

                // Step 4: Wait a moment for replica creation to start
                Thread.sleep(5000);

                // Step 5: Describe table to view replica information
                LOGGER.info("Step 5: Describing table to view replicas");
                describeTable(dynamoDbClient, tableName);

                // Step 6: Add a test item to verify replication
                LOGGER.info("Step 6: Adding test item to verify replication");
                putTestItem(dynamoDbClient, tableName, "TestArtist", "TestSong");

                LOGGER.info("Global table setup completed successfully!");
                LOGGER.info("You can verify replication by checking the item in
region: " + replicaRegion.id());

                // Step 7: Remove replica and clean up table
                LOGGER.info("Step 7: Removing replica from region: " +
replicaRegion.id());
                removeReplica(dynamoDbClient, tableName, replicaRegion);
                DeleteTableResponse deleteTableResponse =
dynamoDbClient.deleteTable(
                    DeleteTableRequest.builder().tableName(tableName).build());
                LOGGER.info("MREC global table demonstration completed
successfully!");
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Thread was interrupted", e);
            } catch (DynamoDbException e) {
                LOGGER.severe("DynamoDB operation failed: " + e.getMessage());
                throw e;
            }
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - CreateTable
  - DescribeTable
  - PutItem
  - UpdateTable

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で PartiQL DELETE ステートメントを使用して DynamoDB データを削除する

次のコード例は、PartiQL DELETE ステートメントを使用してデータを削除する方法を示しています。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で PartiQL DELETE ステートメントを使用して DynamoDB テーブルから項目を削除します。

```
/**
 * This example demonstrates how to delete items from a DynamoDB table using
 PartiQL.
```

```
 * It shows different ways to delete documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Delete a single item by its partition key using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemByPartitionKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ?`,
    Parameters: [partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted successfully");
    return data;
  } catch (err) {
    console.error("Error deleting item:", err);
    throw err;
  }
};

/**
 * Delete an item by its composite key (partition key + sort key) using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
```

```
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemByCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${sortKeyName} = ?`,
    Parameters: [partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted successfully");
    return data;
  } catch (err) {
    console.error("Error deleting item:", err);
    throw err;
  }
};

/**
 * Delete an item with a condition to ensure the delete only happens if a
 condition is met.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param conditionAttribute - The attribute to check in the condition
 * @param conditionValue - The value to compare against in the condition
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemWithCondition = async (
```

```
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  conditionAttribute: string,
  conditionValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${conditionAttribute} = ?`,
    Parameters: [partitionKeyValue, conditionValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item deleted with condition successfully");
    return data;
  } catch (err) {
    console.error("Error deleting item with condition:", err);
    throw err;
  }
};

/**
 * Batch delete multiple items using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param keys - Array of objects containing key information
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchDeleteItems = async (
  tableName: string,
  keys: Array<{
    partitionKeyName: string;
    partitionKeyValue: string | number;
    sortKeyName?: string;
    sortKeyValue?: string | number;
  }>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);
```

```
  // Create statements for each delete
  const statements = keys.map((key) => {
    if (key.sortKeyName && key.sortKeyValue !== undefined) {
      return {
        Statement: `DELETE FROM "${tableName}" WHERE ${key.partitionKeyName} = ?
 AND ${key.sortKeyName} = ?`,
        Parameters: [key.partitionKeyValue, key.sortKeyValue],
      };
    } else {
      return {
        Statement: `DELETE FROM "${tableName}" WHERE ${key.partitionKeyName} = ?
`,
        Parameters: [key.partitionKeyValue],
      };
    }
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items batch deleted successfully");
    return data;
  } catch (err) {
    console.error("Error batch deleting items:", err);
    throw err;
  }
};

/**
 * Delete multiple items that match a filter condition.
 * Note: This performs a scan operation which can be expensive on large tables.
 *
 * @param tableName - The name of the DynamoDB table
 * @param filterAttribute - The attribute to filter on
 * @param filterValue - The value to filter by
 * @returns The response from the ExecuteStatementCommand
 */
export const deleteItemsByFilter = async (
  tableName: string,
  filterAttribute: string,
  filterValue: any
```

```
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `DELETE FROM "${tableName}" WHERE ${filterAttribute} = ?`,
    Parameters: [filterValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items deleted by filter successfully");
    return data;
  } catch (err) {
    console.error("Error deleting items by filter:", err);
    throw err;
  }
};

/**
 * Example usage showing how to delete items with different index types
 */
export const deleteExamples = async () => {
  // Delete an item by partition key (simple primary key)
  await deleteItemByPartitionKey("UsersTable", "userId", "user123");

  // Delete an item by composite key (partition key + sort key)
  await deleteItemByCompositeKey(
    "OrdersTable",
    "orderId",
    "order456",
    "productId",
    "prod789"
  );

  // Delete with a condition
  await deleteItemWithCondition(
    "UsersTable",
    "userId",
    "user789",
    "userStatus",
    "inactive"
  );
```

```
  // Batch delete multiple items
  await batchDeleteItems("UsersTable", [
    { partitionKeyName: "userId", partitionKeyValue: "user234" },
    { partitionKeyName: "userId", partitionKeyValue: "user345" },
  ]);

  // Batch delete items with composite keys
  await batchDeleteItems("OrdersTable", [
    {
      partitionKeyName: "orderId",
      partitionKeyValue: "order567",
      sortKeyName: "productId",
      sortKeyValue: "prod123",
    },
    {
      partitionKeyName: "orderId",
      partitionKeyValue: "order678",
      sortKeyName: "productId",
      sortKeyValue: "prod456",
    },
  ]);

  // Delete items by filter (use with caution)
  await deleteItemsByFilter("UsersTable", "userStatus", "deleted");
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - BatchExecuteStatement
  - ExecuteStatement

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して、Amazon Rekognition で画像内の PPE を検出する

次のコード例は、Amazon Rekognition を使用してイメージ内の個人用防護具 (PPE) を検出するアプリケーションを構築する方法を示しています。

Java

SDK for Java 2.x

AWS Lambda 個人用保護具のイメージを検出する関数の作成方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

NoSQL データベースの開始方法

次のコード例は、以下の操作方法を示しています。

- DynamoDB でテーブルを作成する
- DynamoDB テーブルのデータを更新する
- DynamoDB テーブル内のデータにクエリを実行する
- DynamoDB テーブルを削除する

Bash

AWS CLIBash スクリプトを使用した

> **Note**
>
> GitHub には、その他のリソースもあります。デベロッパーチュートリアルのサンプルレポジトリで用例一覧を検索し、設定と実行の方法を確認してください。

```
#!/bin/bash

# DynamoDB Getting Started Tutorial Script
# This script demonstrates basic operations with Amazon DynamoDB:
# - Creating a table
# - Writing data to the table
# - Reading data from the table
# - Updating data in the table
# - Querying data in the table
# - Deleting the table (cleanup)

set -uo pipefail

# Set up logging with secure permissions
LOG_DIR="${XDG_STATE_HOME:-.}/dynamodb-tutorial-logs"
mkdir -p "$LOG_DIR"
LOG_FILE="$LOG_DIR/dynamodb-tutorial-$(date +%Y%m%d-%H%M%S).log"
chmod 700 "$LOG_DIR"
touch "$LOG_FILE"
chmod 600 "$LOG_FILE"
exec > >(tee -a "$LOG_FILE") 2>&1

echo "Starting DynamoDB Getting Started Tutorial at $(date)"
echo "Logging to $LOG_FILE"

# Validate AWS CLI is configured
if ! command -v aws &> /dev/null; then
    echo "ERROR: AWS CLI is not installed or not in PATH"
    exit 1
fi

# Check AWS credentials are available
if ! aws sts get-caller-identity &> /dev/null; then
    echo "ERROR: AWS credentials not configured or invalid"
    exit 1
fi

# Function to check for errors in command output
check_error() {
    local output=$1
    local cmd_name=$2

    if echo "$output" | grep -qi "error\|failed"; then
```

```
        echo "ERROR detected in $cmd_name command:" >&2
        echo "$output" >&2
        return 1
    fi
    return 0
}

# Function to wait for table to be in ACTIVE state
wait_for_table_active() {
    local table_name=$1
    local max_attempts=60
    local attempt=0
    local status=""

    echo "Waiting for table $table_name to become ACTIVE..."

    while [[ "$status" != "ACTIVE" && $attempt -lt $max_attempts ]]; do
        sleep 5
        status=$(aws dynamodb describe-table --table-name "$table_name" --query
 "Table.TableStatus" --output text 2>/dev/null || echo "UNKNOWN")
        echo "Current status: $status"
        ((attempt++))
    done

    if [[ "$status" != "ACTIVE" ]]; then
        echo "ERROR: Table $table_name did not become ACTIVE within timeout
 period" >&2
        return 1
    fi

    echo "Table $table_name is now ACTIVE"
    return 0
}

# Track created resources for cleanup
declare -a RESOURCES=()

# Cleanup function
cleanup() {
    local exit_code=$?

    if [[ $exit_code -ne 0 ]]; then
        echo "Script encountered an error (exit code: $exit_code)" >&2
    fi
```

```
    echo ""
    echo "============================================="
    echo "CLEANUP"
    echo "============================================="
    echo "Resources to clean up:"
    for resource in "${RESOURCES[@]+"${RESOURCES[@]}"}"; do
        echo "- $resource"
    done
    echo ""

    if [[ ${#RESOURCES[@]} -gt 0 ]]; then
        echo "Proceeding with cleanup of all created resources..."

        for resource in "${RESOURCES[@]+"${RESOURCES[@]}"}"; do
            if [[ "$resource" == Table:* ]]; then
                local table_name="${resource#Table:}"
                echo "Deleting table: $table_name"
                if aws dynamodb delete-table --table-name "$table_name" 2>/dev/
null; then
                    echo "Waiting for table deletion to complete..."
                    aws dynamodb wait table-not-exists --table-name "$table_name"
 2>/dev/null || true
                else
                    echo "Warning: Failed to delete table $table_name" >&2
                fi
            fi
        done

        echo "Cleanup completed."
    fi

    return $exit_code
}

trap cleanup EXIT

# Validate table name
validate_table_name() {
    local name=$1
    if [[ ! $name =~ ^[a-zA-Z0-9._-]+$ ]] || [[ ${#name} -gt 255 ]]; then
        echo "ERROR: Invalid table name: $name" >&2
        return 1
    fi
```

```
    return 0
}

# Step 1: Create a table in DynamoDB
echo "Step 1: Creating Music table in DynamoDB..."

TABLE_NAME="Music"
validate_table_name "$TABLE_NAME"

CREATE_TABLE_OUTPUT=$(aws dynamodb create-table \
    --table-name "$TABLE_NAME" \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --table-class STANDARD \
    --tags Key=project,Value=doc-smith Key=tutorial,Value=amazon-dynamodb-gs
 2>&1) || {
    echo "ERROR: Failed to create table" >&2
    exit 1
}

check_error "$CREATE_TABLE_OUTPUT" "create-table"
echo "$CREATE_TABLE_OUTPUT"

# Add table to resources list
RESOURCES+=("Table:$TABLE_NAME")

# Wait for table to be active
wait_for_table_active "$TABLE_NAME"

# Enable point-in-time recovery (best practice)
echo "Enabling point-in-time recovery for the $TABLE_NAME table..."

PITR_OUTPUT=$(aws dynamodb update-continuous-backups \
    --table-name "$TABLE_NAME" \
    --point-in-time-recovery-specification PointInTimeRecoveryEnabled=true 2>&1)
 || {
    echo "ERROR: Failed to enable PITR" >&2
    exit 1
}
```

```
check_error "$PITR_OUTPUT" "update-continuous-backups"
echo "$PITR_OUTPUT"

# Step 2: Write data to the DynamoDB table
echo "Step 2: Writing data to the $TABLE_NAME table..."

# Use a temporary file for item data
ITEMS_TEMP=$(mktemp)
trap "rm -f '$ITEMS_TEMP'" EXIT

cat > "$ITEMS_TEMP" << 'EOF'
{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Call Me Today"},
 "AlbumTitle": {"S": "Somewhat Famous"}, "Awards": {"N": "1"}}
{"Artist": {"S": "No One You Know"}, "SongTitle": {"S": "Howdy"}, "AlbumTitle":
 {"S": "Somewhat Famous"}, "Awards": {"N": "2"}}
{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}, "AlbumTitle":
 {"S": "Songs About Life"}, "Awards": {"N": "10"}}
{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "PartiQL Rocks"}, "AlbumTitle":
 {"S": "Another Album Title"}, "Awards": {"N": "8"}}
EOF

declare -i item_num=0
while IFS= read -r item_data; do
    ((item_num++))
    ITEM_OUTPUT=$(aws dynamodb put-item \
        --table-name "$TABLE_NAME" \
        --item "$item_data" 2>&1) || {
        echo "ERROR: Failed to put item $item_num" >&2
        exit 1
    }
    check_error "$ITEM_OUTPUT" "put-item (item $item_num)"
    echo "Item $item_num added successfully"
done < "$ITEMS_TEMP"

# Step 3: Read data from the DynamoDB table
echo "Step 3: Reading data from the $TABLE_NAME table..."

# Get a specific item
GET_ITEM_OUTPUT=$(aws dynamodb get-item --consistent-read \
    --table-name "$TABLE_NAME" \
    --key '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}' 2>&1)
 || {
    echo "ERROR: Failed to get item" >&2
    exit 1
```

```
}

check_error "$GET_ITEM_OUTPUT" "get-item"
echo "Retrieved item:"
echo "$GET_ITEM_OUTPUT"

# Step 4: Update data in the DynamoDB table
echo "Step 4: Updating data in the $TABLE_NAME table..."

# Update an item
UPDATE_ITEM_OUTPUT=$(aws dynamodb update-item \
    --table-name "$TABLE_NAME" \
    --key '{"Artist": {"S": "Acme Band"}, "SongTitle": {"S": "Happy Day"}}' \
    --update-expression "SET AlbumTitle = :newval" \
    --expression-attribute-values '{":newval": {"S": "Updated Album Title"}}' \
    --return-values ALL_NEW 2>&1) || {
    echo "ERROR: Failed to update item" >&2
    exit 1
}

check_error "$UPDATE_ITEM_OUTPUT" "update-item"
echo "Updated item:"
echo "$UPDATE_ITEM_OUTPUT"

# Step 5: Query data in the DynamoDB table
echo "Step 5: Querying data in the $TABLE_NAME table..."

# Query items by Artist
QUERY_OUTPUT=$(aws dynamodb query \
    --table-name "$TABLE_NAME" \
    --key-condition-expression "Artist = :name" \
    --expression-attribute-values '{":name": {"S": "Acme Band"}}' 2>&1) || {
    echo "ERROR: Failed to query table" >&2
    exit 1
}

check_error "$QUERY_OUTPUT" "query"
echo "Query results:"
echo "$QUERY_OUTPUT"

echo "DynamoDB Getting Started Tutorial completed successfully at $(date)"
echo "Log file: $LOG_FILE"
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - Query
  - UpdateContinuousBackups
  - UpdateItem
  - 待ちます。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で PartiQL INSERT ステートメントを使用して DynamoDB データを挿入する

次のコード例は、PartiQL INSERT ステートメントを使用してデータを挿入する方法を示しています。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で PartiQL INSERT ステートメントを使用して DynamoDB テーブルに項目を挿入します。

```
/**
 * This example demonstrates how to insert items into a DynamoDB table using
 PartiQL.
 * It shows different ways to insert documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
```

```
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Insert a single item into a DynamoDB table using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param item - The item to insert
 * @returns The response from the ExecuteStatementCommand
 */
export const insertItem = async (tableName: string, item: Record<string, any>) =>
 {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Convert the item to a string representation for PartiQL
  const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');

  const params = {
    Statement: `INSERT INTO "${tableName}" VALUE ${itemString}`,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item inserted successfully");
    return data;
  } catch (err) {
    console.error("Error inserting item:", err);
    throw err;
  }
};

/**
 * Insert multiple items into a DynamoDB table using PartiQL batch operation.
 * This is more efficient than inserting items one by one.
 *
 * @param tableName - The name of the DynamoDB table
 * @param items - Array of items to insert
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchInsertItems = async (tableName: string, items: Record<string,
 any>[]) => {
```

```
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create statements for each item
  const statements = items.map((item) => {
    const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');
    return {
      Statement: `INSERT INTO "${tableName}" VALUE ${itemString}`,
    };
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items inserted successfully");
    return data;
  } catch (err) {
    console.error("Error batch inserting items:", err);
    throw err;
  }
};

/**
 * Insert an item with a condition to prevent overwriting existing items.
 * This is useful for ensuring you don't accidentally overwrite data.
 *
 * @param tableName - The name of the DynamoDB table
 * @param item - The item to insert
 * @param partitionKeyName - The name of the partition key attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const insertItemWithCondition = async (
  tableName: string,
  item: Record<string, any>,
  partitionKeyName: string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const itemString = JSON.stringify(item).replace(/"([^"]+)":/g, '$1:');
  const partitionKeyValue = JSON.stringify(item[partitionKeyName]);
```

```
  const params = {
    Statement: `INSERT INTO "${tableName}" VALUE ${itemString} WHERE
 attribute_not_exists(${partitionKeyName})`,
    Parameters: [{ S: partitionKeyValue }],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item inserted with condition successfully");
    return data;
  } catch (err) {
    console.error("Error inserting item with condition:", err);
    throw err;
  }
};

/**
 * Example usage showing how to insert items with different index types
 */
export const insertExamples = async () => {
  // Example table with a simple primary key (just partition key)
  const simpleKeyItem = {
    userId: "user123",
    name: "John Doe",
    email: "john@example.com",
  };
  await insertItem("UsersTable", simpleKeyItem);

  // Example table with composite key (partition key + sort key)
  const compositeKeyItem = {
    orderId: "order456",
    productId: "prod789",
    quantity: 2,
    price: 29.99,
  };
  await insertItem("OrdersTable", compositeKeyItem);

  // Example with Global Secondary Index (GSI)
  // The GSI might be on the email attribute
  const gsiItem = {
    userId: "user789",
    email: "jane@example.com",
    name: "Jane Smith",
```

```
    userType: "premium",  // This could be part of a GSI
  };
  await insertItem("UsersTable", gsiItem);

  // Example with Local Secondary Index (LSI)
  // LSI uses the same partition key but different sort key
  const lsiItem = {
    orderId: "order567",  // Partition key
    productId: "prod123", // Sort key for the table
    orderDate: "2023-11-15", // Potential sort key for an LSI
    quantity: 1,
    price: 19.99,
  };
  await insertItem("OrdersTable", lsiItem);

  // Batch insert example with multiple items
  const batchItems = [
    {
      userId: "user234",
      name: "Alice Johnson",
      email: "alice@example.com",
    },
    {
      userId: "user345",
      name: "Bob Williams",
      email: "bob@example.com",
    },
  ];
  await batchInsertItems("UsersTable", batchItems);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - BatchExecuteStatement
  - ExecuteStatement

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ブラウザからの Lambda 関数の呼び出し

次のコード例は、ブラウザから AWS Lambda 関数を呼び出す方法を示しています。

JavaScript

SDK for JavaScript (v2)

Amazon DynamoDB のテーブルをユーザーの選択内容で更新する AWS Lambda 関数を使用した、ブラウザベースのアプリケーションを作成することができます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda

SDK for JavaScript (v3)

Amazon DynamoDB のテーブルをユーザーの選択内容で更新する AWS Lambda 関数を使用した、ブラウザベースのアプリケーションを作成することができます。このアプリは AWS SDK for JavaScript v3 を使用します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB グローバルセカンダリインデックスを管理する

次のコード例は、グローバルセカンダリインデックスのライフサイクル全体を管理する方法を示しています。

- グローバルセカンダリインデックスを含むテーブルを作成します。
- 新しい GSI を既存のテーブルに追加します。
- GSI ウォームスループットを更新 (増加) します。
- GSI を使用してデータをクエリします。
- GSI を削除します。

Bash

AWS CLIBash スクリプトを使用した

グローバルセカンダリインデックスを含むテーブルを作成します。

```
# Create a table with a GSI
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
        AttributeName=AlbumTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --global-secondary-indexes \
        "IndexName=AlbumIndex,\
        KeySchema=[{AttributeName=AlbumTitle,KeyType=HASH}],\
        Projection={ProjectionType=ALL}"
```

新しい (オンデマンド) GSI を既存のテーブルに追加します。

```
# Add a new GSI to an existing table
aws dynamodb update-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Genre,AttributeType=S \
    --global-secondary-index-updates \
        "[{\"Create\":{\"IndexName\":\"GenreIndex\",\
        \"KeySchema\":[{\"AttributeName\":\"Genre\",\"KeyType\":\"HASH\"}],\
        \"Projection\":{\"ProjectionType\":\"ALL\"}}}]"
```

GSI ウォームスループットを更新 (増加) します。

```
# Increase the warm throughput of a GSI (default values are 12k reads, 4k writes)
aws dynamodb update-table \
    --table-name MusicCollection \
    --global-secondary-index-updates \
        "[{\"Update\":{\"IndexName\":\"AlbumIndex\",\
        \"WarmThroughput\":{\"ReadUnitsPerSecond\":15000,\"WriteUnitsPerSecond
\":6000}}}]"
```

GSI を使用してデータをクエリします。

```
# Query the AlbumIndex GSI
aws dynamodb query \
    --table-name MusicCollection \
    --index-name AlbumIndex \
    --key-condition-expression "AlbumTitle = :album" \
    --expression-attribute-values '{":album":{"S":"Let It Be"}}'

# Query the GenreIndex GSI
aws dynamodb query \
    --table-name MusicCollection \
    --index-name GenreIndex \
    --key-condition-expression "Genre = :genre" \
    --expression-attribute-values '{":genre":{"S":"Jazz"}}'
```

GSI を削除します。

```
# Delete a GSI from a table
aws dynamodb update-table \
    --table-name MusicCollection \
    --global-secondary-index-updates \
        "[{\"Delete\":{\"IndexName\":\"GenreIndex\"}}]"
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。

- CreateTable
- DeleteTable
- クエリ:
- UpdateTable

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB リソースベースのポリシーを管理する

次のコード例は、DynamoDB テーブルのリソースベースのポリシーのライフサイクル全体を管理する方法を示しています。

- リソースポリシーを含むテーブルを作成します。
- リソースポリシーを取得します。
- リソースポリシーを更新します。
- リソースポリシーを削除します。

Bash

AWS CLIBash スクリプトを使用した

リソースポリシーを含むテーブルを作成します。

```
# Step 1: Create a DynamoDB table
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

```
# Step 2: Create a resource-based policy document
cat > policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/DynamoDBReadOnly"
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    }
  ]
}
EOF

# Step 3: Attach the resource-based policy to the table
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
 \
    --policy file://policy.json
```

リソースポリシーを取得します。

```
# Get the resource-based policy attached to a table
aws dynamodb get-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

リソースポリシーを更新します。

```
# Step 1: Create an updated policy document
cat > updated-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::123456789012:role/DynamoDBReadOnly",
          "arn:aws:iam::123456789012:role/DynamoDBAnalytics"
        ]
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
    }
  ]
}
EOF

# Step 2: Update the resource-based policy on the table
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
 \
    --policy file://updated-policy.json
```

リソースポリシーを削除します。

```
# Delete the resource-based policy from a table
aws dynamodb delete-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteResourcePolicy
  - GetResourcePolicy
  - PutResourcePolicy

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した Amazon DynamoDB のパフォーマンスのモニタリング

次のコード例は、DynamoDB のパフォーマンスをモニタリングするように、アプリケーションを設定する方法を示しています。

Java

SDK for Java 2.x

この例では、DynamoDB のパフォーマンスをモニタリングするように Java アプリケーションを設定する方法を示します。アプリケーションからメトリクスデータを CloudWatch に送信してパフォーマンスをモニタリングできます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- CloudWatch
- DynamoDB

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して高度な DynamoDB クエリオペレーションを実行する

以下のコード例は、DynamoDB で高度なクエリオペレーションを実行する方法を示しています。

- さまざまなフィルタリングと条件の手法を使用してテーブルをクエリします。
- 大きな結果セットのページ分割を実装します。
- 代替アクセスパターンにはグローバルセカンダリインデックスを使用します。
- アプリケーション要件に基づいて整合性コントロールを適用します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x で強力な整合性のある読み込みを使用してクエリを実行します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithConsistentReads(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final boolean useConsistentRead) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
```

```
            .expressionAttributeValues(expressionAttributeValues)
            .consistentRead(useConsistentRead)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query successful. Found {0} items",
 response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with consistent reads", e);
            throw e;
        }
    }
```

AWS SDK for Java 2.x でグローバルセカンダリインデックスを使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryTable(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
```

```
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on base table successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw new DynamoDbQueryException("Table not found: " + tableName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying base table: " + e.getMessage());
            throw new DynamoDbQueryException("Failed to execute query on base
table", e);
        }
    }

    /**
     * Queries a DynamoDB Global Secondary Index (GSI) by partition key.
     *
     * @param tableName         The name of the DynamoDB table
     * @param indexName         The name of the GSI
     * @param partitionKeyName  The name of the GSI partition key attribute
     * @param partitionKeyValue The value of the GSI partition key to query
     * @return The query response from DynamoDB
     * @throws ResourceNotFoundException if the table or index doesn't exist
     * @throws DynamoDbException if the query fails
```

```
     */
    public QueryResponse queryGlobalSecondaryIndex(
        final String tableName, final String indexName, final String
partitionKeyName, final String partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Index name", indexName);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_IK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_IK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .indexName(indexName)
            .keyConditionExpression(GSI_KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on GSI successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(
                "Error: The Amazon DynamoDB table \"%s\" or index \"%s\" can't be
found.\n", tableName, indexName);
            throw new DynamoDbQueryException("Table or index not found: " +
tableName + "/" + indexName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying GSI: " + e.getMessage());
```

```
            throw new DynamoDbQueryException("Failed to execute query on GSI",
 e);
        }
    }
```

AWS SDK for Java 2.x でページ分割を使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public List<Map<String, AttributeValue>> queryWithPagination(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue, final int pageSize) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validatePositiveInteger("Page size", pageSize);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
```

```
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(pageSize);

        // List to store all items from all pages
        final List<Map<String, AttributeValue>> allItems = new ArrayList<>();

        // Map to store the last evaluated key for pagination
        Map<String, AttributeValue> lastEvaluatedKey = null;
        int pageNumber = 1;

        try {
            do {
                // If we have a last evaluated key, use it for the next page
                if (lastEvaluatedKey != null) {
                    queryRequestBuilder.exclusiveStartKey(lastEvaluatedKey);
                }

                // Execute the query
                final QueryResponse response =
dynamoDbClient.query(queryRequestBuilder.build());

                // Process the current page of results
                final List<Map<String, AttributeValue>> pageItems =
response.items();
                allItems.addAll(pageItems);

                // Get the last evaluated key for the next page
                lastEvaluatedKey = response.lastEvaluatedKey();
                if (lastEvaluatedKey != null && lastEvaluatedKey.isEmpty()) {
                    lastEvaluatedKey = null;
                }

                System.out.println("Page " + pageNumber + ": Retrieved " +
pageItems.size() + " items (Running total: "
                    + allItems.size() + ")");

                pageNumber++;

            } while (lastEvaluatedKey != null);
```

```
            System.out.println("Query with pagination complete. Retrieved a total
 of " + allItems.size()
                + " items across " + (pageNumber - 1) + " pages");

            return allItems;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
 found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with pagination: " +
 e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x で複雑なフィルターを使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithComplexFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String statusAttrName,
        final String activeStatus,
        final String pendingStatus,
        final String priceAttrName,
        final double minPrice,
        final double maxPrice,
        final String categoryAttrName) {
```

```
        // Validate parameters
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Status attribute name",
statusAttrName);
        CodeSampleUtils.validateStringParameter("Active status", activeStatus);
        CodeSampleUtils.validateStringParameter("Pending status", pendingStatus);
        CodeSampleUtils.validateStringParameter("Price attribute name",
priceAttrName);
        CodeSampleUtils.validateStringParameter("Category attribute name",
categoryAttrName);
        CodeSampleUtils.validateNumericRange("Minimum price", minPrice, 0.0,
Double.MAX_VALUE);
        CodeSampleUtils.validateNumericRange("Maximum price", maxPrice, minPrice,
Double.MAX_VALUE);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pk", partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_STATUS,
statusAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PRICE,
priceAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_CATEGORY,
categoryAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":pkValue", AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_ACTIVE,
            AttributeValue.builder().s(activeStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PENDING,
            AttributeValue.builder().s(pendingStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MIN_PRICE,
            AttributeValue.builder().n(String.valueOf(minPrice)).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MAX_PRICE,
            AttributeValue.builder().n(String.valueOf(maxPrice)).build());
```

```
        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        return dynamoDbClient.query(queryRequest);
    }
```

AWS SDK for Java 2.x で動的に構築されたフィルター式を使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public static QueryResponse queryWithDynamicFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final Map<String, Object> filterCriteria,
        final Region region,
        final DynamoDbClient dynamoDbClient) {

        validateParameters(tableName, partitionKeyName, partitionKeyValue,
 filterCriteria);

        DynamoDbClient ddbClient = dynamoDbClient;
        boolean shouldClose = false;

        try {
            if (ddbClient == null) {
```

```
                ddbClient = createClient(region);
                shouldClose = true;
            }

            final QueryWithDynamicFilter queryHelper = new
QueryWithDynamicFilter(ddbClient);
            return queryHelper.queryWithDynamicFilter(tableName,
partitionKeyName, partitionKeyValue, filterCriteria);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Failed to execute dynamic filter query: " +
e.getMessage());
            throw e;
        } catch (Exception e) {
            System.err.println("Unexpected error during query: " +
e.getMessage());
            throw e;
        } finally {
            if (shouldClose && ddbClient != null) {
                ddbClient.close();
            }
        }
    }

    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue>
<filterAttrName> <filterAttrValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    filterAttrName - The name of the attribute to filter on.
                    filterAttrValue - The value to filter by.
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 5) {
            System.out.println(usage);
```

```
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final String filterAttrName = args[3];
        final String filterAttrValue = args[4];
        final Region region = args.length > 5 ? Region.of(args[5]) :
Region.US_EAST_1;

        System.out.println("Querying items with dynamic filter: " +
filterAttrName + " = " + filterAttrValue);

        try {
            // Using the builder pattern to create and execute the query
            final QueryResponse response = new DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriterion(filterAttrName, filterAttrValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> System.out.println(item));

            // Demonstrate multiple filter criteria
            System.out.println("\nNow querying with multiple filter criteria:");

            Map<String, Object> multipleFilters = new HashMap<>();
            multipleFilters.put(filterAttrName, filterAttrValue);
            multipleFilters.put("status", "active");

            final QueryResponse multiFilterResponse = new
DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriteria(multipleFilters)
                .withRegion(region)
                .execute();
```

```
            System.out.println("Found " + multiFilterResponse.count() + " items
 with multiple filters:");
            multiFilterResponse.items().forEach(item ->
 System.out.println(item));

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

AWS SDK for Java 2.x でフィルター式と制限を使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithFilterAndLimit(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String filterAttrName,
        final String filterAttrValue,
        final int limit) {
```

```
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Filter attribute name",
filterAttrName);
        CodeSampleUtils.validateStringParameter("Filter attribute value",
filterAttrValue);
        CodeSampleUtils.validatePositiveInteger("Limit", limit);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_FILTER,
filterAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_FILTER,
            AttributeValue.builder().s(filterAttrValue).build());

        // Create the filter expression
        final String filterExpression = "#filterAttr = :filterValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(limit)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with filter and limit successful. Found
{0} items", response.count());
            LOGGER.log(
```

```
                Level.INFO, "ScannedCount: {0} (total items evaluated before
 filtering)", response.scannedCount());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with filter and limit: {0}",
 e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で強力な整合性のある読み込みを使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with configurable read consistency
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {boolean} useConsistentRead - Whether to use strongly consistent reads
 * @returns {Promise<Object>} - The query response
 */
async function queryWithConsistentRead(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  useConsistentRead = false
) {
  try {
```

```
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue }
      },
      ConsistentRead: useConsistentRead
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with consistent read: ${error}`);
    throw error;
  }
}
```

AWS SDK for JavaScript でグローバルセカンダリインデックスを使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table using the primary key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} userId - The user ID to query by (partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryTable(
  config,
  tableName,
  userId
```

```
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the base table
    const input = {
      TableName: tableName,
      KeyConditionExpression: "user_id = :userId",
      ExpressionAttributeValues: {
        ":userId": { S: userId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying table: ${error}`);
    throw error;
  }
}

/**
 * Queries a DynamoDB Global Secondary Index (GSI)
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} indexName - The name of the GSI to query
 * @param {string} gameId - The game ID to query by (GSI partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryGSI(
  config,
  tableName,
  indexName,
  gameId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the GSI
    const input = {
```

```
      TableName: tableName,
      IndexName: indexName,
      KeyConditionExpression: "game_id = :gameId",
      ExpressionAttributeValues: {
        ":gameId": { S: gameId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying GSI: ${error}`);
    throw error;
  }
}
```

AWS SDK for JavaScript でページ分割を使用してクエリを実行します。

```
/**
 * Example demonstrating how to handle large query result sets in DynamoDB using
 pagination
 *
 * This example shows:
 * - How to use pagination to handle large result sets
 * - How to use LastEvaluatedKey to retrieve the next page of results
 * - How to construct subsequent query requests using ExclusiveStartKey
 */
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with pagination to handle large result sets
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number} pageSize - Number of items per page
 * @returns {Promise<Array>} - All items from the query
 */
async function queryWithPagination(
```

```
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  pageSize = 25
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize variables for pagination
    let lastEvaluatedKey = undefined;
    const allItems = [];
    let pageCount = 0;

    // Loop until all pages are retrieved
    do {
      // Construct the query input
      const input = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pkValue",
        Limit: pageSize,
        ExpressionAttributeNames: {
          "#pk": partitionKeyName
        },
        ExpressionAttributeValues: {
          ":pkValue": { S: partitionKeyValue }
        }
      };

      // Add ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        input.ExclusiveStartKey = lastEvaluatedKey;
      }

      // Execute the query
      const command = new QueryCommand(input);
      const response = await client.send(command);

      // Process the current page of results
      pageCount++;
      console.log(`Processing page ${pageCount} with ${response.Items.length}
 items`);
```

```
      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems.push(...response.Items);
      }

      // Get the LastEvaluatedKey for the next page
      lastEvaluatedKey = response.LastEvaluatedKey;

    } while (lastEvaluatedKey); // Continue until there are no more pages

    console.log(`Query complete. Retrieved ${allItems.length} items in
 ${pageCount} pages.`);
    return allItems;
  } catch (error) {
    console.error(`Error querying with pagination: ${error}`);
    throw error;
  }
}

/**
 * Example usage:
 *
 * // Query all items in the "AWS DynamoDB" forum with pagination
 * const allItems = await queryWithPagination(
 *   { region: "us-west-2" },
 *   "ForumThreads",
 *   "ForumName",
 *   "AWS DynamoDB",
 *   25 // 25 items per page
 * );
 *
 * console.log(`Total items retrieved: ${allItems.length}`);
 *
 * // Notes on pagination:
 * // - LastEvaluatedKey contains the primary key of the last evaluated item
 * // - When LastEvaluatedKey is undefined/null, there are no more items to
 retrieve
 * // - ExclusiveStartKey tells DynamoDB where to start the next page
 * // - Pagination helps manage memory usage for large result sets
 * // - Each page requires a separate network request to DynamoDB
 */

module.exports = { queryWithPagination };
```

AWS SDK for JavaScript で複雑なフィルターを使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with a complex filter expression
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number|string} minViews - Minimum number of views for filtering
 * @param {number|string} minReplies - Minimum number of replies for filtering
 * @param {string} requiredTag - Tag that must be present in the item's tags set
 * @returns {Promise<Object>} - The query response
 */
async function queryWithComplexFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  minViews,
  minReplies,
  requiredTag
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      FilterExpression: "views >= :minViews AND replies >= :minReplies AND
 contains(tags, :tag)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":minViews": { N: minViews.toString() },
```

```
        ":minReplies": { N: minReplies.toString() },
        ":tag": { S: requiredTag }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with complex filter: ${error}`);
    throw error;
  }
}
```

AWS SDK for JavaScript で動的に構築されたフィルター式を使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

async function queryWithDynamicFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  sortKeyValue,
  filterParams = {}
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize filter expression components
    let filterExpressions = [];
    const expressionAttributeValues = {
      ":pkValue": { S: partitionKeyValue },
      ":skValue": { S: sortKeyValue }
    };
    const expressionAttributeNames = {
      "#pk": partitionKeyName,
      "#sk": sortKeyName
    };
```

```
    // Add status filter if provided
    if (filterParams.status) {
      filterExpressions.push("status = :status");
      expressionAttributeValues[":status"] = { S: filterParams.status };
    }

    // Add minimum views filter if provided
    if (filterParams.minViews !== undefined) {
      filterExpressions.push("views >= :minViews");
      expressionAttributeValues[":minViews"] = { N:
 filterParams.minViews.toString() };
    }

    // Add author filter if provided
    if (filterParams.author) {
      filterExpressions.push("author = :author");
      expressionAttributeValues[":author"] = { S: filterParams.author };
    }

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue AND #sk = :skValue"
    };

    // Add filter expression if any filters were provided
    if (filterExpressions.length > 0) {
      input.FilterExpression = filterExpressions.join(" AND ");
    }

    // Add expression attribute names and values
    input.ExpressionAttributeNames = expressionAttributeNames;
    input.ExpressionAttributeValues = expressionAttributeValues;

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with dynamic filter: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で強力な整合性のある読み込みを使用してクエリを実行します。

```
import time

import boto3
from boto3.dynamodb.conditions import Key


def query_with_consistent_read(
    table_name,
    partition_key_name,
    partition_key_value,
    sort_key_name=None,
    sort_key_value=None,
    consistent_read=True,
):
    """
    Query a DynamoDB table with the option for strongly consistent reads.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str, optional): The name of the sort key attribute.
        sort_key_value (str, optional): The value of the sort key to query.
        consistent_read (bool, optional): Whether to use strongly consistent
 reads. Defaults to True.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)
```

```
    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    if sort_key_name and sort_key_value:
        key_condition = key_condition & Key(sort_key_name).eq(sort_key_value)

    # Perform the query with the consistent read option
    response = table.query(KeyConditionExpression=key_condition,
 ConsistentRead=consistent_read)

    return response
```

AWS SDK for Python (Boto3) でグローバルセカンダリインデックスを使用してクエリを実行します。

```
import boto3
from boto3.dynamodb.conditions import Key


def query_table(table_name, partition_key_name, partition_key_value):
    """
    Query a DynamoDB table using its primary key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the table's primary key
    response =
 table.query(KeyConditionExpression=Key(partition_key_name).eq(partition_key_value))

    return response
```

```
def query_gsi(table_name, index_name, partition_key_name, partition_key_value):
    """
    Query a Global Secondary Index (GSI) on a DynamoDB table.

    Args:
        table_name (str): The name of the DynamoDB table.
        index_name (str): The name of the Global Secondary Index.
        partition_key_name (str): The name of the GSI's partition key attribute.
        partition_key_value (str): The value of the GSI's partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the GSI
    response = table.query(
        IndexName=index_name,
 KeyConditionExpression=Key(partition_key_name).eq(partition_key_value)
    )

    return response
```

AWS SDK for Python (Boto3) でページ分割を使用してクエリを実行します。

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_pagination(
    table_name, partition_key_name, partition_key_value, page_size=25,
 max_pages=None
):
    """
    Query a DynamoDB table with pagination to handle large result sets.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
Defaults to 25.
        max_pages (int, optional): The maximum number of pages to retrieve. If
None, retrieves all pages.

    Returns:
        list: All items retrieved from the query across all pages.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_count = 0
    all_items = []

    # Paginate through the results
    while True:
        # Check if we've reached the maximum number of pages
        if max_pages is not None and page_count >= max_pages:
            break

        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Process the current page of results
        items = response.get("Items", [])
```

```
        all_items.extend(items)

        # Update pagination tracking
        page_count += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if not last_evaluated_key:
            break

    return all_items


def query_with_pagination_generator(
    table_name, partition_key_name, partition_key_value, page_size=25
):
    """
    Query a DynamoDB table with pagination using a generator to handle large
 result sets.
    This approach is memory-efficient as it yields one page at a time.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.

    Yields:
        tuple: A tuple containing (items, page_number, last_page) where:
            - items is a list of items for the current page
            - page_number is the current page number (starting from 1)
            - last_page is a boolean indicating if this is the last page
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_number = 0
```

```
    # Paginate through the results
    while True:
        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Get the current page of results
        items = response.get("Items", [])
        page_number += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # Determine if this is the last page
        is_last_page = last_evaluated_key is None

        # Yield the current page of results
        yield (items, page_number, is_last_page)

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if is_last_page:
            break
```

AWS SDK for Python (Boto3) で複雑なフィルターを使用してクエリを実行します。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key
```

```
def query_with_complex_filter(
    table_name,
    partition_key_name,
    partition_key_value,
    min_rating=None,
    status_list=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        min_rating (float, optional): Minimum rating value for filtering.
        status_list (list, optional): List of status values to include.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize the filter expression and expression attribute values
    filter_expression = None
    expression_attribute_values = {}

    # Build the filter expression based on provided parameters
    if min_rating is not None:
        filter_expression = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if status_list and len(status_list) > 0:
        status_condition = None
        for i, status in enumerate(status_list):
            status_value_name = f":status{i}"
            expression_attribute_values[status_value_name] = status
```

```
            if status_condition is None:
                status_condition = Attr("status").eq(status)
            else:
                status_condition = status_condition | Attr("status").eq(status)

        if filter_expression is None:
            filter_expression = status_condition
        else:
            filter_expression = filter_expression & status_condition

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if filter_expression is None:
            filter_expression = price_condition
        else:
            filter_expression = filter_expression & price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
 expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response


def query_with_complex_filter_and_or(
    table_name,
    partition_key_name,
    partition_key_value,
    category=None,
    min_rating=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression using AND and OR
 operators.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        category (str, optional): Category value for filtering.
        min_rating (float, optional): Minimum rating value for filtering.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build a complex filter expression with AND and OR operators
    filter_expression = None
    expression_attribute_values = {}

    # Build the category condition
    if category:
        filter_expression = Attr("category").eq(category)
        expression_attribute_values[":category"] = category

    # Build the rating and price condition (rating >= min_rating OR price <=
max_price)
    rating_price_condition = None

    if min_rating is not None:
        rating_price_condition = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if rating_price_condition is None:
            rating_price_condition = price_condition
        else:
            rating_price_condition = rating_price_condition | price_condition
```

```
    # Combine the conditions
    if rating_price_condition:
        if filter_expression is None:
            filter_expression = rating_price_condition
        else:
            filter_expression = filter_expression & rating_price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
 expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response
```

AWS SDK for Python (Boto3) で動的に構築されたフィルター式を使用してクエリを実行します。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_dynamic_filter(
    table_name, partition_key_name, partition_key_value, filter_conditions=None
):
    """
    Query a DynamoDB table with a dynamically constructed filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_conditions (dict, optional): A dictionary of filter conditions
 where
```

```
            keys are attribute names and values are dictionaries with 'operator'
and 'value'.
            Example: {'rating': {'operator': '>=', 'value': 4}, 'status':
{'operator': '=', 'value': 'active'}}

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize variables for the filter expression and attribute values
    filter_expression = None
    expression_attribute_values = {":pk_val": partition_key_value}

    # Dynamically build the filter expression if filter conditions are provided
    if filter_conditions:
        for attr_name, condition in filter_conditions.items():
            operator = condition.get("operator")
            value = condition.get("value")
            attr_value_name = f":{attr_name}"
            expression_attribute_values[attr_value_name] = value

            # Create the appropriate filter expression based on the operator
            current_condition = None
            if operator == "=":
                current_condition = Attr(attr_name).eq(value)
            elif operator == "!=":
                current_condition = Attr(attr_name).ne(value)
            elif operator == ">":
                current_condition = Attr(attr_name).gt(value)
            elif operator == ">=":
                current_condition = Attr(attr_name).gte(value)
            elif operator == "<":
                current_condition = Attr(attr_name).lt(value)
            elif operator == "<=":
                current_condition = Attr(attr_name).lte(value)
            elif operator == "contains":
                current_condition = Attr(attr_name).contains(value)
            elif operator == "begins_with":
```

```
                current_condition = Attr(attr_name).begins_with(value)

            # Combine with existing filter expression using AND
            if current_condition:
                if filter_expression is None:
                    filter_expression = current_condition
                else:
                    filter_expression = filter_expression & current_condition

    # Perform the query with the dynamically built filter expression
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression

    response = table.query(**query_params)
    return response
```

AWS SDK for Python (Boto3) でフィルター式と制限を使用してクエリを実行します。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_filter_and_limit(
    table_name,
    partition_key_name,
    partition_key_value,
    filter_attribute=None,
    filter_value=None,
    limit=10,
):
    """
    Query a DynamoDB table with a filter expression and limit the number of
 results.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
```

```
        filter_attribute (str, optional): The attribute name to filter on.
        filter_value (any, optional): The value to compare against in the filter.
        limit (int, optional): The maximum number of items to evaluate. Defaults
to 10.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition, "Limit": limit}

    # Add the filter expression if filter attributes are provided
    if filter_attribute and filter_value is not None:
        query_params["FilterExpression"] =
Attr(filter_attribute).gt(filter_value)
        query_params["ExpressionAttributeValues"] = {":filter_value":
filter_value}

    # Execute the query
    response = table.query(**query_params)
    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB でリストオペレーションを実行する

以下のコード例は、DynamoDB でリストオペレーションを実行する方法を示しています。

- リスト属性に要素を追加します。
- リスト属性から要素を削除します。
- インデックスでリスト内の特定の要素を更新します。
- リスト追加関数およびリストインデックス関数を使用します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してリストオペレーションをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Appends items to a list attribute.
     *
     * <p>This method demonstrates how to use the list_append function to add
     * items to the end of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param itemsToAppend The items to append to the list
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse appendToList(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
```

```
        String listAttributeName,
        List<AttributeValue> itemsToAppend) {

        // Create a list value from the items to append
        AttributeValue listValue =
AttributeValue.builder().l(itemsToAppend).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName =
list_append(if_not_exists(#attrName, :emptyList), :newItems)")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(
                ":newItems",
                listValue,
                ":emptyList",
                AttributeValue.builder().l(new
ArrayList<AttributeValue>()).build()))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Prepends items to a list attribute.
     *
     * <p>This method demonstrates how to use the list_append function to add
     * items to the beginning of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param itemsToPrepend The items to prepend to the list
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse prependToList(
        DynamoDbClient dynamoDbClient,
        String tableName,
```

```
        Map<String, AttributeValue> key,
        String listAttributeName,
        List<AttributeValue> itemsToPrepend) {

        // Create a list value from the items to prepend
        AttributeValue listValue =
AttributeValue.builder().l(itemsToPrepend).build();

        // Define the update parameters
        // Note: To prepend, we put the new items first in the list_append
function
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName = list_append(:newItems,
if_not_exists(#attrName, :emptyList))")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(
                ":newItems",
                listValue,
                ":emptyList",
                AttributeValue.builder().l(new
ArrayList<AttributeValue>()).build()))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Updates a specific element in a list attribute.
     *
     * <p>This method demonstrates how to update a specific element in a list
     * by its index.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param index The index of the element to update
     * @param newValue The new value for the element
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
```

```
     */
    public static UpdateItemResponse updateListElement(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        int index,
        AttributeValue newValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attrName[" + index + "] = :newValue")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .expressionAttributeValues(Map.of(":newValue", newValue))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes a specific element from a list attribute.
     *
     * <p>This method demonstrates how to remove a specific element from a list
     * by its index.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param listAttributeName The name of the list attribute
     * @param index The index of the element to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeListElement(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String listAttributeName,
        int index) {
```

```
        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #attrName[" + index + "]")
            .expressionAttributeNames(Map.of("#attrName", listAttributeName))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a list attribute.
     *
     * <p>Helper method to retrieve the current value of a list attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param listAttributeName The name of the list attribute
     * @return The list attribute value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static List<AttributeValue> getListAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String listAttributeName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(listAttributeName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the list attribute if it exists, otherwise null
            if (response.item() != null &&
response.item().containsKey(listAttributeName)) {
                return response.item().get(listAttributeName).l();
```

```
            }

            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get list attribute: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用したリストオペレーションの使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating list operations in DynamoDB");

        try {
            // Example 1: Append items to a list
            System.out.println("\nExample 1: Appending items to a list");
            List<AttributeValue> tagsToAppend = List.of(
                AttributeValue.builder().s("Electronics").build(),
                AttributeValue.builder().s("Gadget").build());

            UpdateItemResponse appendResponse = appendToList(dynamoDbClient,
tableName, key, "Tags", tagsToAppend);

            System.out.println("Updated list attribute: " +
appendResponse.attributes());

            // Example 2: Prepend items to a list
            System.out.println("\nExample 2: Prepending items to a list");
            List<AttributeValue> tagsToPrepend = List.of(
                AttributeValue.builder().s("Featured").build(),
                AttributeValue.builder().s("New").build());

            UpdateItemResponse prependResponse = prependToList(dynamoDbClient,
tableName, key, "Tags", tagsToPrepend);
```

```
            System.out.println("Updated list attribute: " +
prependResponse.attributes());

            // Example 3: Update a specific element in a list
            System.out.println("\nExample 3: Updating a specific element in a
list");
            UpdateItemResponse updateResponse = updateListElement(
                dynamoDbClient,
                tableName,
                key,
                "Tags",
                0,
                AttributeValue.builder().s("BestSeller").build());

            System.out.println("Updated list attribute: " +
updateResponse.attributes());

            // Example 4: Remove a specific element from a list
            System.out.println("\nExample 4: Removing a specific element from a
list");
            UpdateItemResponse removeResponse = removeListElement(dynamoDbClient,
tableName, key, "Tags", 1);

            System.out.println("Updated list attribute: " +
removeResponse.attributes());

            // Example 5: Get the current value of a list attribute
            System.out.println("\nExample 5: Getting the current value of a list
attribute");
            List<AttributeValue> currentList = getListAttribute(dynamoDbClient,
tableName, key, "Tags");

            if (currentList != null) {
                System.out.println("Current list attribute:");
                for (int i = 0; i < currentList.size(); i++) {
                    System.out.println("  [" + i + "]: " +
currentList.get(i).s());
                }
            } else {
                System.out.println("List attribute not found");
            }

            // Explain list operations
```

```
            System.out.println("\nKey points about DynamoDB list operations:");
            System.out.println("1. Lists are ordered collections of attributes");
            System.out.println("2. Use list_append to add items to a list");
            System.out.println("3. To append items, use list_append(existingList,
 newItems)");
            System.out.println("4. To prepend items, use list_append(newItems,
 existingList)");
            System.out.println("5. Use index notation (list[0]) to access or
 update specific elements");
            System.out.println("6. Use REMOVE to delete elements from a list");
            System.out.println("7. List indices are zero-based");
            System.out.println("8. Use if_not_exists to handle the case where the
 list doesn't exist yet");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してリストオペレーションをデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Append elements to a list attribute.
 *
 * This function demonstrates how to use the list_append function to add elements
 * to the end of a list.
```

```
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} values - The values to append to the list
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function appendToList(
  config,
  tableName,
  key,
  listName,
  values
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} =
 list_append(if_not_exists(${listName}, :empty_list), :values)`,
    ExpressionAttributeValues: {
      ":empty_list": [],
      ":values": values
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Prepend elements to a list attribute.
 *
 * This function demonstrates how to use the list_append function to add elements
 * to the beginning of a list.
 *
```

```
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} values - The values to prepend to the list
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function prependToList(
  config,
  tableName,
  key,
  listName,
  values
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  // Note: To prepend, we put the new values first in the list_append function
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} = list_append(:values,
 if_not_exists(${listName}, :empty_list))`,
    ExpressionAttributeValues: {
      ":empty_list": [],
      ":values": values
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update a specific element in a list by index.
 *
 * This function demonstrates how to update a specific element in a list
 * using the index notation.
 *
```

```
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} index - The index of the element to update
 * @param {any} value - The new value for the element
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateListElement(
  config,
  tableName,
  key,
  listName,
  index,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using index notation
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName}[${index}] = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Remove an element from a list by index.
 *
 * This function demonstrates how to remove a specific element from a list
 * using the REMOVE action with index notation.
 *
 * @param {Object} config - AWS configuration object
```

```
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} index - The index of the element to remove
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function removeListElement(
  config,
  tableName,
  key,
  listName,
  index
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using REMOVE with index notation
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `REMOVE ${listName}[${index}]`,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Concatenate two lists.
 *
 * This function demonstrates how to concatenate two lists using the list_append
 function.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName1 - The name of the first list attribute
 * @param {string} listName2 - The name of the second list attribute
 * @param {string} resultListName - The name of the attribute to store the
 concatenated list
```

```
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function concatenateLists(
  config,
  tableName,
  key,
  listName1,
  listName2,
  resultListName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using list_append
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${resultListName} =
 list_append(if_not_exists(${listName1}, :empty_list),
 if_not_exists(${listName2}, :empty_list))`,
    ExpressionAttributeValues: {
      ":empty_list": []
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Create a nested list structure.
 *
 * This function demonstrates how to create and work with nested lists.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {Array} nestedLists - An array of arrays to create a nested list
 structure
```

```
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function createNestedList(
  config,
  tableName,
  key,
  listName,
  nestedLists
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters to create a nested list
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName} = :nested_lists`,
    ExpressionAttributeValues: {
      ":nested_lists": nestedLists
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update an element in a nested list.
 *
 * This function demonstrates how to update an element in a nested list
 * using multiple index notations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} listName - The name of the list attribute
 * @param {number} outerIndex - The index in the outer list
 * @param {number} innerIndex - The index in the inner list
 * @param {any} value - The new value for the element
 * @returns {Promise<Object>} - The response from DynamoDB
```

```
 */
async function updateNestedListElement(
  config,
  tableName,
  key,
  listName,
  outerIndex,
  innerIndex,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using multiple index notations
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${listName}[${outerIndex}][${innerIndex}] = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
```

```
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

AWS SDK for JavaScript を使用したリストオペレーションの使用例。

```
/**
 * Example of how to work with lists in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "UserProfiles";
  const key = { UserId: "U12345" };

  console.log("Demonstrating list operations in DynamoDB");

  try {
    // Example 1: Append elements to a list
    console.log("\nExample 1: Appending elements to a list");
    const response1 = await appendToList(
      config,
      tableName,
      key,
      "RecentSearches",
      ["laptop", "headphones", "monitor"]
    );
```

```
    console.log("Appended to list:", response1.Attributes);

    // Example 2: Prepend elements to a list
    console.log("\nExample 2: Prepending elements to a list");
    const response2 = await prependToList(
      config,
      tableName,
      key,
      "RecentSearches",
      ["keyboard", "mouse"]
    );

    console.log("Prepended to list:", response2.Attributes);

    // Get the current state of the item
    let currentItem = await getItem(config, tableName, key);
    console.log("\nCurrent state of RecentSearches:",
currentItem?.RecentSearches);

    // Example 3: Update a specific element in a list
    console.log("\nExample 3: Updating a specific element in a list");
    const response3 = await updateListElement(
      config,
      tableName,
      key,
      "RecentSearches",
      0, // Update the first element
      "mechanical keyboard" // New value
    );

    console.log("Updated list element:", response3.Attributes);

    // Example 4: Remove an element from a list
    console.log("\nExample 4: Removing an element from a list");
    const response4 = await removeListElement(
      config,
      tableName,
      key,
      "RecentSearches",
      2 // Remove the third element
    );

    console.log("List after removing element:", response4.Attributes);
```

```
// Example 5: Create and concatenate lists
console.log("\nExample 5: Creating and concatenating lists");

// First, create two separate lists
await updateWithMultipleActions(
  config,
  tableName,
  key,
  "SET WishList = :wishlist, SavedItems = :saveditems",
  null,
  {
    ":wishlist": ["gaming laptop", "wireless earbuds"],
    ":saveditems": ["smartphone", "tablet"]
  }
);

// Then, concatenate them
const response5 = await concatenateLists(
  config,
  tableName,
  key,
  "WishList",
  "SavedItems",
  "AllItems"
);

console.log("Concatenated lists:", response5.Attributes);

// Example 6: Create a nested list structure
console.log("\nExample 6: Creating a nested list structure");
const response6 = await createNestedList(
  config,
  tableName,
  key,
  "Categories",
  [
    ["Electronics", "Computers", "Accessories"],
    ["Books", "Magazines", "E-books"],
    ["Clothing", "Shoes", "Watches"]
  ]
);

console.log("Created nested list:", response6.Attributes);
```

```
    // Example 7: Update an element in a nested list
    console.log("\nExample 7: Updating an element in a nested list");
    const response7 = await updateNestedListElement(
      config,
      tableName,
      key,
      "Categories",
      0, // First inner list
      1, // Second element in that list
      "Laptops" // New value
    );

    console.log("Updated nested list element:", response7.Attributes);

    // Get the final state of the item
    currentItem = await getItem(config, tableName, key);
    console.log("\nFinal state of the item:", JSON.stringify(currentItem, null,
 2));

    // Explain list operations
    console.log("\nKey points about list operations in DynamoDB:");
    console.log("1. Use list_append to add elements to a list");
    console.log("2. To append elements, use list_append(existingList,
 newElements)");
    console.log("3. To prepend elements, use list_append(newElements,
 existingList)");
    console.log("4. Use if_not_exists to handle cases where the list might not
 exist yet");
    console.log("5. Use index notation (list[0]) to access or update specific
 elements");
    console.log("6. Use REMOVE with index notation to remove elements from a
 list");
    console.log("7. Lists can contain elements of different types");
    console.log("8. Lists can be nested (lists of lists)");
    console.log("9. Use multiple index notations (list[0][1]) to access nested
 list elements");

  } catch (error) {
    console.error("Error:", error);
  }
}

/**
```

```
 * Helper function for the examples.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} updateExpression - The update expression
 * @param {Object} expressionAttributeNames - Expression attribute name
 placeholders
 * @param {Object} expressionAttributeValues - Expression attribute value
 placeholders
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleActions(
  config,
  tableName,
  key,
  updateExpression,
  expressionAttributeNames,
  expressionAttributeValues
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Prepare the update parameters
  const updateParams = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ReturnValues: "UPDATED_NEW"
  };

  // Add expression attribute names if provided
  if (expressionAttributeNames) {
    updateParams.ExpressionAttributeNames = expressionAttributeNames;
  }

  // Add expression attribute values if provided
  if (expressionAttributeValues) {
    updateParams.ExpressionAttributeValues = expressionAttributeValues;
  }

  // Execute the update
  const response = await docClient.send(new UpdateCommand(updateParams));
```

```
  return response;
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してリストオペレーションをデモンストレーションします。

```
import boto3
import json
from typing import Any, Dict, List, Optional, Union


def create_list_attribute(
    table_name: str, key: Dict[str, Any], list_name: str, list_values: List[Any]
) -> Dict[str, Any]:
    """
    Create a new list attribute or replace an existing one.

    This function demonstrates how to create a new list attribute or replace
    an existing list with new values.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        list_values (List[Any]): The values to set in the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)
```

```
    # Use the SET operation to create or replace the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = :list_values",
        ExpressionAttributeValues={":list_values": list_values},
        ReturnValues="UPDATED_NEW",
    )

    return response


def append_to_list(
    table_name: str, key: Dict[str, Any], list_name: str, values_to_append:
 List[Any]
) -> Dict[str, Any]:
    """
    Append values to the end of a list attribute.

    This function demonstrates how to use the list_append function to add
 elements
    to the end of a list attribute.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_append (List[Any]): The values to append to the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use list_append to add values to the end of the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = list_append({list_name}, :values)",
        ExpressionAttributeValues={":values": values_to_append},
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def prepend_to_list(
    table_name: str, key: Dict[str, Any], list_name: str, values_to_prepend:
 List[Any]
) -> Dict[str, Any]:
    """
    Prepend values to the beginning of a list attribute.

    This function demonstrates how to use the list_append function to add
 elements
    to the beginning of a list attribute.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_prepend (List[Any]): The values to prepend to the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use list_append with reversed order to add values to the beginning of the
 list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} = list_append(:values, {list_name})",
        ExpressionAttributeValues={":values": values_to_prepend},
        ReturnValues="UPDATED_NEW",
    )

    return response


def update_list_element(
    table_name: str, key: Dict[str, Any], list_name: str, index: int, new_value:
 Any
```

```
) -> Dict[str, Any]:
    """
    Update a specific element in a list attribute.

    This function demonstrates how to update a specific element in a list
 attribute
    using the index notation.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        index (int): The zero-based index of the element to update.
        new_value (Any): The new value for the element.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the index notation to update a specific element
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name}[{index}] = :value",
        ExpressionAttributeValues={":value": new_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def remove_list_element(
    table_name: str, key: Dict[str, Any], list_name: str, index: int
) -> Dict[str, Any]:
    """
    Remove a specific element from a list attribute.

    This function demonstrates how to remove a specific element from a list
 attribute
    using the REMOVE action with index notation.
```

```
    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        index (int): The zero-based index of the element to remove.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the REMOVE action with index notation to remove a specific element
    response = table.update_item(
        Key=key, UpdateExpression=f"REMOVE {list_name}[{index}]",
 ReturnValues="UPDATED_NEW"
    )

    return response


def update_nested_list_element(
    table_name: str, key: Dict[str, Any], path: str, new_value: Any
) -> Dict[str, Any]:
    """
    Update an element in a nested list structure.

    This function demonstrates how to update an element in a nested list
 structure
    using expression attribute names for the path components.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path (str): The path to the nested element (e.g., "parent[0].child[1]").
        new_value (Any): The new value for the element.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Define a type for path parts
```

```
    path_part = Dict[str, Union[str, int]]
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Parse the path to extract attribute names and indices
    path_parts: List[path_part] = []
    current_part = ""
    in_bracket = False

    for char in path:
        if char == "[":
            if current_part:
                path_parts.append({"type": "attribute", "value": current_part})
                current_part = ""
            in_bracket = True
        elif char == "]":
            if current_part:
                # Fix for mypy: Use a properly typed dictionary with Union type
                path_parts.append({"type": "index", "value": int(current_part)})
                current_part = ""
            in_bracket = False
        elif char == "." and not in_bracket:
            if current_part:
                path_parts.append({"type": "attribute", "value": current_part})
                current_part = ""
        else:
            current_part += char

    if current_part:
        path_parts.append({"type": "attribute", "value": current_part})

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}

    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        if part["type"] == "attribute":
            name_placeholder = f"#attr{i}"
            expression_attribute_names[name_placeholder] = part["value"]

            if path_expression:
```

```
                path_expression += "."
            path_expression += name_placeholder
        elif part["type"] == "index":
            path_expression += f"[{part['value']}]"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues={":value": new_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def create_list_if_not_exists(
    table_name: str, key: Dict[str, Any], list_name: str, default_values:
 List[Any]
) -> Dict[str, Any]:
    """
    Create a list attribute if it doesn't exist.

    This function demonstrates how to use if_not_exists to create a list
 attribute
    with default values if it doesn't already exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        default_values (List[Any]): The default values for the list.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)
```

```
    # Use if_not_exists to create the list if it doesn't exist
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} =
 if_not_exists({list_name}, :default)",
        ExpressionAttributeValues={":default": default_values},
        ReturnValues="UPDATED_NEW",
    )

    return response


def append_to_list_safely(
    table_name: str,
    key: Dict[str, Any],
    list_name: str,
    values_to_append: List[Any],
    default_values: Optional[List[Any]] = None,
) -> Dict[str, Any]:
    """
    Append values to a list, creating it if it doesn't exist.

    This function demonstrates how to safely append values to a list attribute,
    creating the list with default values if it doesn't exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        list_name (str): The name of the list attribute.
        values_to_append (List[Any]): The values to append to the list.
        default_values (Optional[List[Any]]): The default values if the list
 doesn't exist.
            If not provided, values_to_append will be used as the default.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # If default_values is not provided, use values_to_append
```

```
    if default_values is None:
        default_values = values_to_append

    # Use if_not_exists with list_append to safely append to the list
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {list_name} =
 list_append(if_not_exists({list_name}, :default), :values)",
        ExpressionAttributeValues={
            ":default": default_values if default_values else [],
            ":values": values_to_append,
        },
        ReturnValues="UPDATED_NEW",
    )

    return response
```

AWS SDK for Python (Boto3) を使用したリストオペレーションの使用例。

```
def example_usage():
    """Example of how to use list operations in DynamoDB."""
    # Example parameters
    table_name = "UserData"
    key = {"UserId": "user123"}

    print("Example 1: Creating a list attribute")
    try:
        response = create_list_attribute(
            table_name=table_name,
            key=key,
            list_name="Interests",
            list_values=["Reading", "Hiking", "Photography"],
        )
        print(
            f"List attribute created successfully:
 {json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error creating list attribute: {e}")
```

```
    print("\nExample 2: Appending values to a list")
    try:
        response = append_to_list(
            table_name=table_name,
            key=key,
            list_name="Interests",
            values_to_append=["Cooking", "Gardening"],
        )
        print(
            f"Values appended to list successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error appending to list: {e}")

    print("\nExample 3: Prepending values to a list")
    try:
        response = prepend_to_list(
            table_name=table_name,
            key=key,
            list_name="Interests",
            values_to_prepend=["Travel", "Music"],
        )
        print(
            f"Values prepended to list successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error prepending to list: {e}")

    print("\nExample 4: Updating a specific list element")
    try:
        response = update_list_element(
            table_name=table_name,
            key=key,
            list_name="Interests",
            index=2,
            new_value="Mountain Hiking",
        )
        print(
            f"List element updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
```

```
        print(f"Error updating list element: {e}")

    print("\nExample 5: Removing a list element")
    try:
        response = remove_list_element(
            table_name=table_name, key=key, list_name="Interests", index=0
        )
        print(
            f"List element removed successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error removing list element: {e}")

    print("\nExample 6: Working with nested lists")
    try:
        # First, create an item with a nested structure
        dynamodb = boto3.resource("dynamodb")
        table = dynamodb.Table(table_name)

        table.update_item(
            Key={"UserId": "user456"},
            UpdateExpression="SET #skills = :skills",
            ExpressionAttributeNames={"#skills": "Skills"},
            ExpressionAttributeValues={
                ":skills": [
                    {"Category": "Programming", "Languages": ["Python", "Java",
"JavaScript"]},
                    {"Category": "Database", "Systems": ["DynamoDB", "MongoDB",
"PostgreSQL"]},
                ]
            },
        )

        # Now update a nested element
        response = update_nested_list_element(
            table_name=table_name,
            key={"UserId": "user456"},
            path="Skills[0].Languages[1]",
            new_value="TypeScript",
        )
        print(
            f"Nested list element updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
```

```
        )
    except Exception as e:
        print(f"Error working with nested lists: {e}")

    print("\nExample 7: Creating a list if it doesn't exist")
    try:
        response = create_list_if_not_exists(
            table_name=table_name,
            key={"UserId": "user789"},
            list_name="Preferences",
            default_values=["Default1", "Default2", "Default3"],
        )
        print(
            f"List created with default values:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error creating list with default values: {e}")

    print("\nExample 8: Safely appending to a list")
    try:
        response = append_to_list_safely(
            table_name=table_name,
            key={"UserId": "user789"},
            list_name="Notifications",
            values_to_append=["New message received"],
            default_values=[],
        )
        print(f"Safely appended to list: {json.dumps(response.get('Attributes',
{}), default=str)}")
    except Exception as e:
        print(f"Error safely appending to list: {e}")

    print("\nKey Points About Working with Lists in DynamoDB:")
    print("1. Lists are ordered collections of elements that can be of different
types")
    print("2. Use the SET operation with direct assignment to create or replace a
list")
    print("3. Use list_append() to add elements to a list without replacing the
entire list")
    print("4. To append to the end: list_append(list_name, :values)")
    print("5. To prepend to the beginning: list_append(:values, list_name)")
    print("6. Use index notation list_name[index] to access or update specific
elements")
```

```
    print("7. Use the REMOVE action with index notation to remove specific
elements")
    print("8. Lists can contain nested structures like maps and other lists")
    print("9. Use if_not_exists() to create a list with default values if it
doesn't exist")
    print("10. List indices are zero-based (the first element is at index 0)")
    print("11. Attempting to access an index beyond the list bounds will result
in an error")
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB でマップオペレーションを実行する

以下のコード例は、DynamoDB でマップオペレーションを実行する方法を示しています。

- マップ構造にネストされた属性を追加および更新します。
- マップから特定のフィールドを削除します。
- 深くネストされたマップ属性を操作します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してマップオペレーションをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;
```

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Updates a map attribute that may not exist.
     *
     * <p>This method demonstrates how to safely update a map attribute
     * by using if_not_exists to handle the case where the map doesn't exist yet.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param mapKey The key within the map to update
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateMapAttributeSafe(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String mapName,
        String mapKey,
        AttributeValue value) {

        // A single UpdateExpression can't reference both a map and a path inside
        // that same map (for example "#mapName" and "#mapName.#mapKey"):
DynamoDB
        // rejects overlapping document paths with a ValidationException.
Instead,
        // perform the update in two steps.

        // Step 1: create the map with an empty value if it doesn't already
exist.
        dynamoDbClient.updateItem(UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #mapName =
if_not_exists(#mapName, :emptyMap)")
            .expressionAttributeNames(Map.of("#mapName", mapName))
            .expressionAttributeValues(Map.of(
```

```
                ":emptyMap", AttributeValue.builder().m(new
HashMap<>()).build()))
            .build());

        // Step 2: set the key within the now-guaranteed-to-exist map.
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #mapName.#mapKey = :value")
            .expressionAttributeNames(Map.of(
                "#mapName", mapName,
                "#mapKey", mapKey))
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Adds an attribute to a nested map.
     *
     * <p>This method demonstrates how to update a nested attribute without
     * overwriting the entire map.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param path The path to the nested attribute as a list
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse addToNestedMap(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        List<String> path,
        AttributeValue value) {

        // Create expression attribute names for each part of the path
        Map<String, String> expressionAttributeNames = new HashMap<>();
        for (int i = 0; i < path.size(); i++) {
```

```
            expressionAttributeNames.put("#attr" + i, path.get(i));
        }

        // Build the attribute path using the expression attribute names
        StringBuilder attributePathExpression = new StringBuilder();
        for (int i = 0; i < path.size(); i++) {
            if (i > 0) {
                attributePathExpression.append(".");
            }
            attributePathExpression.append("#attr").append(i);
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET " + attributePathExpression.toString() + "
= :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes an attribute from a map.
     *
     * <p>This method demonstrates how to remove a specific attribute from a map.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param mapKey The key within the map to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeMapAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
```

```
        String mapName,
        String mapKey) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #mapName.#mapKey")
            .expressionAttributeNames(Map.of(
                "#mapName", mapName,
                "#mapKey", mapKey))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Creates a map with multiple attributes in a single operation.
     *
     * <p>This method demonstrates how to create a map with multiple attributes
     * in a single update operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param mapName The name of the map attribute
     * @param attributes The attributes to set in the map
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse createMapWithAttributes(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String mapName,
        Map<String, AttributeValue> attributes) {

        // Create a map value from the attributes
        AttributeValue mapValue = AttributeValue.builder().m(attributes).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
```

```
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #mapName = :mapValue")
            .expressionAttributeNames(Map.of("#mapName", mapName))
            .expressionAttributeValues(Map.of(":mapValue", mapValue))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a map attribute.
     *
     * <p>Helper method to retrieve the current value of a map attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param mapName The name of the map attribute
     * @return The map attribute value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, AttributeValue> getMapAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String mapName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(mapName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the map attribute if it exists, otherwise null
            if (response.item() != null && response.item().containsKey(mapName))
{
                return response.item().get(mapName).m();
            }
```

```
            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get map attribute: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用したマップオペレーションの使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating map operations in DynamoDB");

        try {
            // Example 1: Create a map with multiple attributes
            System.out.println("\nExample 1: Creating a map with multiple
attributes");
            Map<String, AttributeValue> productDetails = new HashMap<>();
            productDetails.put("Color",
AttributeValue.builder().s("Red").build());
            productDetails.put("Weight",
AttributeValue.builder().n("2.5").build());
            productDetails.put(
                "Dimensions", AttributeValue.builder().s("10x20x5").build());

            UpdateItemResponse createResponse =
                createMapWithAttributes(dynamoDbClient, tableName, key,
"Details", productDetails);

            System.out.println("Created map attribute: " +
createResponse.attributes());

            // Example 2: Update a specific attribute in a map
            System.out.println("\nExample 2: Updating a specific attribute in a
map");
```

```
            UpdateItemResponse updateResponse = updateMapAttributeSafe(
                dynamoDbClient,
                tableName,
                key,
                "Details",
                "Color",
                AttributeValue.builder().s("Blue").build());

            System.out.println("Updated map attribute: " +
updateResponse.attributes());

            // Example 3: Add an attribute to a nested map
            System.out.println("\nExample 3: Adding an attribute to a nested
map");
            UpdateItemResponse nestedResponse = addToNestedMap(
                dynamoDbClient,
                tableName,
                key,
                List.of("Specifications", "Technical", "Resolution"),
                AttributeValue.builder().s("1920x1080").build());

            System.out.println("Added to nested map: " +
nestedResponse.attributes());

            // Example 4: Remove an attribute from a map
            System.out.println("\nExample 4: Removing an attribute from a map");
            UpdateItemResponse removeResponse =
                removeMapAttribute(dynamoDbClient, tableName, key, "Details",
"Dimensions");

            System.out.println("Updated map after removal: " +
removeResponse.attributes());

            // Example 5: Get the current value of a map attribute
            System.out.println("\nExample 5: Getting the current value of a map
attribute");
            Map<String, AttributeValue> currentMap =
getMapAttribute(dynamoDbClient, tableName, key, "Details");

            if (currentMap != null) {
                System.out.println("Current map attribute:");
                for (Map.Entry<String, AttributeValue> entry :
currentMap.entrySet()) {
```

```
                    System.out.println("  " + entry.getKey() + ": " +
entry.getValue());
                }
            } else {
                System.out.println("Map attribute not found");
            }

            // Explain map operations
            System.out.println("\nKey points about DynamoDB map operations:");
            System.out.println("1. Maps are unordered collections of name-value
pairs");
            System.out.println("2. Use dot notation (map.key) to access or update
specific attributes");
            System.out.println("3. You can update individual attributes without
overwriting the entire map");
            System.out.println("4. Maps can be nested to create complex data
structures");
            System.out.println("5. Use REMOVE to delete attributes from a map");
            System.out.println("6. You can create a map with multiple attributes
in a single operation");
            System.out.println("7. Map keys are case-sensitive");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してマップオペレーションをデモンストレーションします。

```
/**
 * Example of updating map attributes in DynamoDB.
 *
 * This module demonstrates how to update map attributes that may not exist,
```

```
 * how to update nested attributes, and how to handle various map update
 scenarios.
 */

const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update a map attribute safely, handling the case where the map might not
 exist.
 *
 * This function demonstrates using the if_not_exists function to safely update
 * a map attribute that might not exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
 * @param {string} mapKey - The key within the map to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMapAttributeSafe(
  config,
  tableName,
  key,
  mapName,
  mapKey,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with if_not_exists
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${mapName}.${mapKey} = :value`,
    ExpressionAttributeValues: {
```

```
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return response;
  } catch (error) {
    // If the error is because the map doesn't exist, create it
    if (error.name === "ValidationException" &&
        error.message.includes("The document path provided in the update
 expression is invalid")) {

      // Create the map with the specified key-value pair
      const createParams = {
        TableName: tableName,
        Key: key,
        UpdateExpression: `SET ${mapName} = :map`,
        ExpressionAttributeValues: {
          ":map": { [mapKey]: value }
        },
        ReturnValues: "UPDATED_NEW"
      };

      return await docClient.send(new UpdateCommand(createParams));
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Update a map attribute using the if_not_exists function.
 *
 * This function demonstrates a more elegant approach using if_not_exists
 * to handle the case where the map doesn't exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
```

```
 * @param {string} mapKey - The key within the map to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMapAttributeWithIfNotExists(
  config,
  tableName,
  key,
  mapName,
  mapKey,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // A single UpdateExpression can't reference both a map and a path inside that
  // same map (for example `${mapName}` and `${mapName}.${mapKey}`): DynamoDB
  // rejects overlapping document paths with a ValidationException. Instead,
  // perform the update in two steps.

  // Step 1: create the map with an empty value if it doesn't already exist.
  await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${mapName} = if_not_exists(${mapName}, :emptyMap)`,
    ExpressionAttributeValues: {
      ":emptyMap": {}
    }
  }));

  // Step 2: set the key within the now-guaranteed-to-exist map.
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${mapName}.${mapKey} = :value`,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));
```

```
  return response;
}

/**
 * Add a value to a deeply nested map, creating parent maps if they don't exist.
 *
 * This function demonstrates how to update a deeply nested attribute,
 * creating any parent maps that don't exist along the way.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string[]} path - The path to the nested attribute as an array of keys
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function addToNestedMap(
  config,
  tableName,
  key,
  path,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression and expression attribute values
  let updateExpression = "SET";
  const expressionAttributeValues = {};

  // For each level in the path, create a map if it doesn't exist
  for (let i = 0; i < path.length; i++) {
    const currentPath = path.slice(0, i + 1).join(".");
    const parentPath = i > 0 ? path.slice(0, i).join(".") : null;

    if (parentPath) {
      updateExpression += ` ${parentPath} =
 if_not_exists(${parentPath}, :emptyMap${i}),`;
      expressionAttributeValues[`:emptyMap${i}`] = {};
    }
  }
```

```
  // Set the final value
  const fullPath = path.join(".");
  updateExpression += ` ${fullPath} = :value`;
  expressionAttributeValues[":value"] = value;

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update multiple fields in a map attribute in a single operation.
 *
 * This function demonstrates how to update multiple fields in a map
 * in a single DynamoDB operation.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} mapName - The name of the map attribute
 * @param {Object} updates - Object containing key-value pairs to update
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateMultipleMapFields(
  config,
  tableName,
  key,
  mapName,
  updates
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);
```

```
  // Build the update expression and expression attribute values
  let updateExpression = `SET ${mapName} = if_not_exists(${mapName}, :emptyMap)`;
  const expressionAttributeValues = {
    ":emptyMap": {}
  };

  // Add each update to the expression
  Object.entries(updates).forEach(([field, value], index) => {
    updateExpression += `, ${mapName}.${field} = :val${index}`;
    expressionAttributeValues[`:val${index}`] = value;
  });

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
```

```
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}

/**
 * Example of how to use the map attribute update functions.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Users";
  const key = { UserId: "U12345" };

  console.log("Demonstrating different approaches to update map attributes in
 DynamoDB");

  try {
    // Example 1: Update a map attribute that might not exist (two-step approach)
    console.log("\nExample 1: Updating a map attribute that might not exist (two-
step approach)");
    const response1 = await updateMapAttributeSafe(
      config,
      tableName,
      key,
      "Preferences",
      "Theme",
      "Dark"
    );

    console.log("Updated preferences:", response1.Attributes);

    // Example 2: Update a map attribute using if_not_exists (elegant approach)
```

```
    console.log("\nExample 2: Updating a map attribute using if_not_exists
(elegant approach)");
    const response2 = await updateMapAttributeWithIfNotExists(
      config,
      tableName,
      key,
      "Settings",
      "NotificationsEnabled",
      true
    );

    console.log("Updated settings:", response2.Attributes);

    // Example 3: Update a deeply nested attribute
    console.log("\nExample 3: Updating a deeply nested attribute");
    const response3 = await addToNestedMap(
      config,
      tableName,
      key,
      ["Profile", "Address", "City"],
      "Seattle"
    );

    console.log("Updated nested attribute:", response3.Attributes);

    // Example 4: Update multiple fields in a map
    console.log("\nExample 4: Updating multiple fields in a map");
    const response4 = await updateMultipleMapFields(
      config,
      tableName,
      key,
      "ContactInfo",
      {
        Email: "user@example.com",
        Phone: "555-123-4567",
        PreferredContact: "Email"
      }
    );

    console.log("Updated multiple fields:", response4.Attributes);

    // Get the final state of the item
    console.log("\nFinal state of the item:");
    const item = await getItem(config, tableName, key);
```

```
    console.log(JSON.stringify(item, null, 2));

    // Explain the benefits of different approaches
    console.log("\nKey points about updating map attributes:");
    console.log("1. Use if_not_exists to handle maps that might not exist");
    console.log("2. Multiple updates can be combined in a single operation");
    console.log("3. Deeply nested attributes require creating parent maps");
    console.log("4. DynamoDB expressions are atomic - the entire update succeeds
 or fails");
    console.log("5. Using a single operation is more efficient than multiple
 separate updates");

  } catch (error) {
    console.error("Error:", error);
  }
}

// Export the functions
module.exports = {
  updateMapAttributeSafe,
  updateMapAttributeWithIfNotExists,
  addToNestedMap,
  updateMultipleMapFields,
  getItem,
  exampleUsage
};

// Run the example if this file is executed directly
if (require.main === module) {
  exampleUsage();
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してマップオペレーションをデモンストレーションします。

```
"""
Example of updating map attributes in DynamoDB.

This module demonstrates how to update map attributes in DynamoDB, including
handling cases where the map attribute might not exist yet.
"""


import boto3
from typing import Any, Dict, Optional



def update_map_attribute_safe(
    table_name: str, key: Dict[str, Any], map_name: str, map_key: str, value: Any
) -> Dict[str, Any]:
    """
    Update a specific key in a map attribute, creating the map if it doesn't
 exist.

    This function demonstrates how to safely update a key within a map attribute,
    even if the map doesn't exist yet in the item.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        map_name (str): The name of the map attribute.
        map_key (str): The key within the map to update.
        value (Any): The value to set for the map key.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use SET with attribute_not_exists to safely update the map
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #map.#key = :value",
```

```
        ExpressionAttributeNames={"#map": map_name, "#key": map_key},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def add_to_nested_map(
    table_name: str, key: Dict[str, Any], path: str, value: Any
) -> Dict[str, Any]:
    """
    Add or update a value in a deeply nested map structure.

    This function demonstrates how to update a value at a specific path in a
    nested map structure, creating any intermediate maps as needed.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path (str): The path to the nested attribute (e.g.,
 "user.preferences.theme").
        value (Any): The value to set at the specified path.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Split the path into components
    path_parts = path.split(".")

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}

    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        name_placeholder = f"#attr{i}"
        expression_attribute_names[name_placeholder] = part
```

```
        if i == 0:
            path_expression = name_placeholder
        else:
            path_expression += f".{name_placeholder}"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def update_map_with_if_not_exists(
    table_name: str,
    key: Dict[str, Any],
    map_name: str,
    map_key: str,
    value: Any,
    default_map: Optional[Dict[str, Any]] = None,
) -> Dict[str, Any]:
    """
    Update a key in a map, creating the map with default values if it doesn't
 exist.

    This function demonstrates how to use if_not_exists to initialize a map with
    default values if it doesn't exist yet, and then update a specific key.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        map_name (str): The name of the map attribute.
        map_key (str): The key within the map to update.
        value (Any): The value to set for the map key.
        default_map (Optional[Dict[str, Any]]): Default map values if the map
 doesn't exist.
```

```
    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Set default map if not provided
    if default_map is None:
        default_map = {}

    # Create a map with the new key-value pair
    updated_map = default_map.copy()
    updated_map[map_key] = value

    # Use if_not_exists to initialize the map if it doesn't exist
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #map = if_not_exists(#map, :default_map)",
        ExpressionAttributeNames={"#map": map_name},
        ExpressionAttributeValues={":default_map": updated_map},
        ReturnValues="UPDATED_NEW",
    )

    return response


def merge_into_map(
    table_name: str, key: Dict[str, Any], map_name: str, values_to_merge:
 Dict[str, Any]
) -> Dict[str, Any]:
    """
    Merge multiple key-value pairs into a map attribute.

    This function demonstrates how to update multiple keys in a map attribute
    in a single operation, without overwriting the entire map.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        map_name (str): The name of the map attribute.
        values_to_merge (Dict[str, Any]): Key-value pairs to merge into the map.
```

```
    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the update expression for each key-value pair
    update_expression = "SET "
    expression_attribute_names = {"#map": map_name}
    expression_attribute_values = {}

    # Add each key-value pair to the update expression
    for i, (k, v) in enumerate(values_to_merge.items()):
        key_placeholder = f"#key{i}"
        value_placeholder = f":value{i}"

        expression_attribute_names[key_placeholder] = k
        expression_attribute_values[value_placeholder] = v

        if i > 0:
            update_expression += ", "
        update_expression += f"#map.{key_placeholder} = {value_placeholder}"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues=expression_attribute_values,
        ReturnValues="UPDATED_NEW",
    )

    return response




def example_usage():
    """Example of how to use the map attribute update functions."""
    # Example parameters
    table_name = "UserProfiles"
```

```
    key = {"UserId": "user123"}

    print("Example 1: Updating a specific key in a map attribute")
    try:
        response = update_map_attribute_safe(
            table_name=table_name, key=key, map_name="Preferences",
map_key="Theme", value="Dark"
        )
        print(f"Map attribute updated successfully: {response.get('Attributes',
{})}")
    except Exception as e:
        print(f"Error updating map attribute: {e}")

    print("\nExample 2: Adding a value to a deeply nested map")
    try:
        response = add_to_nested_map(
            table_name=table_name, key=key, path="Settings.Notifications.Email",
value=True
        )
        print(f"Nested map updated successfully: {response.get('Attributes',
{})}")
    except Exception as e:
        print(f"Error updating nested map: {e}")

    print("\nExample 3: Initializing a map with default values if it doesn't
exist")
    try:
        default_map = {"Language": "English", "Currency": "USD", "TimeZone":
"UTC"}

        response = update_map_with_if_not_exists(
            table_name=table_name,
            key={"UserId": "newuser456"},
            map_name="Preferences",
            map_key="Theme",
            value="Light",
            default_map=default_map,
        )
        print(f"Map initialized with defaults: {response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error initializing map: {e}")

    print("\nExample 4: Merging multiple values into a map")
    try:
```

```
        values_to_merge = {
            "NotificationsEnabled": True,
            "EmailFrequency": "Daily",
            "PushNotifications": False,
        }

        response = merge_into_map(
            table_name=table_name,
            key=key,
            map_name="NotificationSettings",
            values_to_merge=values_to_merge,
        )
        print(f"Multiple values merged into map: {response.get('Attributes',
{})}")
    except Exception as e:
        print(f"Error merging values into map: {e}")

    print("\nBest practices for working with map attributes in DynamoDB:")
    print("1. Use dot notation to access and update nested attributes")
    print("2. Use ExpressionAttributeNames to handle reserved words and special
characters")
    print("3. Use if_not_exists() to handle cases where attributes might not
exist")
    print("4. Update specific map keys rather than overwriting the entire map")
    print("5. Use a single update operation to modify multiple map keys for
better performance")
    print("6. Consider your data model carefully to minimize the need for deeply
nested attributes")



if __name__ == "__main__":
    example_usage()
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB でセットオペレーションを実行する

以下のコード例は、DynamoDB でセットオペレーションを実行する方法を示しています。

- セット属性に要素を追加します。
- セット属性から要素を削除します。
- セットで ADD および DELETE オペレーションを使用します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してセットオペレーションをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

    /**
     * Adds values to a string set attribute.
     *
     * <p>This method demonstrates how to use the ADD operation to add values
     * to a string set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param valuesToAdd The values to add to the set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
```

```
    public static UpdateItemResponse addToStringSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<String> valuesToAdd) {

        // Create a string set value from the values to add
        AttributeValue setValue =
AttributeValue.builder().ss(valuesToAdd).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #setAttr :valuesToAdd")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToAdd", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Adds values to a number set attribute.
     *
     * <p>This method demonstrates how to use the ADD operation to add values
     * to a number set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param valuesToAdd The values to add to the set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse addToNumberSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
```

```
        Set<Number> valuesToAdd) {

        // Convert numbers to strings for DynamoDB
        Set<String> stringValues = new HashSet<>();
        for (Number value : valuesToAdd) {
            stringValues.add(value.toString());
        }

        // Create a number set value from the values to add
        AttributeValue setValue =
AttributeValue.builder().ns(stringValues).build();

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #setAttr :valuesToAdd")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToAdd", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Removes values from a set attribute.
     *
     * <p>This method demonstrates how to use the DELETE operation to remove
values
     * from a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param valuesToRemove The values to remove from the set
     * @param isNumberSet Whether the set is a number set (true) or string set
(false)
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse removeFromSet(
```

```
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<?> valuesToRemove,
        boolean isNumberSet) {

        AttributeValue setValue;

        if (isNumberSet) {
            // Convert numbers to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : valuesToRemove) {
                if (value instanceof Number) {
                    stringValues.add(value.toString());
                } else {
                    throw new IllegalArgumentException("Values must be numbers
for a number set");
                }
            }

            setValue = AttributeValue.builder().ns(stringValues).build();
        } else {
            // Convert objects to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : valuesToRemove) {
                stringValues.add(value.toString());
            }

            setValue = AttributeValue.builder().ss(stringValues).build();
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("DELETE #setAttr :valuesToRemove")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
            .expressionAttributeValues(Map.of(":valuesToRemove", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
```

```
    }

    /**
     * Checks if a value exists in a set attribute.
     *
     * <p>This method demonstrates how to use the contains function to check
     * if a value exists in a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to check
     * @param setAttributeName The name of the set attribute
     * @param valueToCheck The value to check for
     * @return Map containing the result of the check
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, Object> checkIfValueInSet(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        String valueToCheck) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the update parameters with a condition expression
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #tempAttr = :tempVal")
                .conditionExpression("contains(#setAttr, :valueToCheck)")
                .expressionAttributeNames(Map.of("#setAttr", setAttributeName,
"#tempAttr", "TempAttribute"))
                .expressionAttributeValues(Map.of(
                    ":valueToCheck",
AttributeValue.builder().s(valueToCheck).build(),
                    ":tempVal", AttributeValue.builder().s("TempValue").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            // Attempt the update operation
            dynamoDbClient.updateItem(request);
```

```
            // If we get here, the condition was met
            result.put("exists", true);
            result.put("message", "Value '" + valueToCheck + "' exists in the
set");

            // Clean up the temporary attribute
            UpdateItemRequest cleanupRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("REMOVE #tempAttr")
                .expressionAttributeNames(Map.of("#tempAttr", "TempAttribute"))
                .build();

            dynamoDbClient.updateItem(cleanupRequest);

        } catch (DynamoDbException e) {
            if (e.getMessage().contains("ConditionalCheckFailed")) {
                // The condition was not met
                result.put("exists", false);
                result.put("message", "Value '" + valueToCheck + "' does not
exist in the set");
            } else {
                // Some other error occurred
                result.put("exists", false);
                result.put("message", "Error checking set: " + e.getMessage());
                result.put("error", e.getClass().getSimpleName());
            }
        }

        return result;
    }

    /**
     * Creates a set with multiple values in a single operation.
     *
     * <p>This method demonstrates how to create a set with multiple values
     * in a single update operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param setAttributeName The name of the set attribute
     * @param setValues The values to include in the set
```

```
     * @param isNumberSet Whether to create a number set (true) or string set
(false)
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse createSetWithValues(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String setAttributeName,
        Set<?> setValues,
        boolean isNumberSet) {

        AttributeValue setValue;

        if (isNumberSet) {
            // Convert numbers to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : setValues) {
                if (value instanceof Number) {
                    stringValues.add(value.toString());
                } else {
                    throw new IllegalArgumentException("Values must be numbers
for a number set");
                }
            }

            setValue = AttributeValue.builder().ns(stringValues).build();
        } else {
            // Convert objects to strings for DynamoDB
            Set<String> stringValues = new HashSet<>();
            for (Object value : setValues) {
                stringValues.add(value.toString());
            }

            setValue = AttributeValue.builder().ss(stringValues).build();
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #setAttr = :setValue")
            .expressionAttributeNames(Map.of("#setAttr", setAttributeName))
```

```
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Gets the current value of a set attribute.
     *
     * <p>Helper method to retrieve the current value of a set attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param setAttributeName The name of the set attribute
     * @return The set attribute value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static AttributeValue getSetAttribute(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String setAttributeName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(setAttributeName)
            .build();

        try {
            // Perform the get operation
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the set attribute if it exists, otherwise null
            if (response.item() != null &&
response.item().containsKey(setAttributeName)) {
                return response.item().get(setAttributeName);
            }

            return null;
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
```

```
                .message("Failed to get set attribute: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用したセットオペレーションの使用例。

```
   public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
       // Example key
       Map<String, AttributeValue> key = new HashMap<>();
       key.put("ProductId", AttributeValue.builder().s("P12345").build());

       System.out.println("Demonstrating set operations in DynamoDB");

       try {
           // Example 1: Create a string set with multiple values
           System.out.println("\nExample 1: Creating a string set with multiple
values");
           Set<String> tags = new HashSet<>();
           tags.add("Electronics");
           tags.add("Gadget");
           tags.add("Smartphone");

           UpdateItemResponse createResponse = createSetWithValues(
               dynamoDbClient, tableName, key, "Tags", tags, false // Not a
number set
               );

           System.out.println("Created set attribute: " +
createResponse.attributes());

           // Example 2: Add values to a string set
           System.out.println("\nExample 2: Adding values to a string set");
           Set<String> additionalTags = new HashSet<>();
           additionalTags.add("Mobile");
           additionalTags.add("Wireless");

           UpdateItemResponse addResponse = addToStringSet(dynamoDbClient,
tableName, key, "Tags", additionalTags);
```

```
            System.out.println("Updated set attribute: " +
addResponse.attributes());

            // Example 3: Create a number set with multiple values
            System.out.println("\nExample 3: Creating a number set with multiple
values");
            Set<Number> ratings = new HashSet<>();
            ratings.add(4);
            ratings.add(5);
            ratings.add(4.5);

            UpdateItemResponse createNumberSetResponse = createSetWithValues(
                dynamoDbClient, tableName, key, "Ratings", ratings, true // Is a
number set
                );

            System.out.println("Created number set attribute: " +
createNumberSetResponse.attributes());

            // Example 4: Add values to a number set
            System.out.println("\nExample 4: Adding values to a number set");
            Set<Number> additionalRatings = new HashSet<>();
            additionalRatings.add(3.5);
            additionalRatings.add(4.2);

            UpdateItemResponse addNumberResponse =
                addToNumberSet(dynamoDbClient, tableName, key, "Ratings",
additionalRatings);

            System.out.println("Updated number set attribute: " +
addNumberResponse.attributes());

            // Example 5: Remove values from a set
            System.out.println("\nExample 5: Removing values from a set");
            Set<String> tagsToRemove = new HashSet<>();
            tagsToRemove.add("Gadget");

            UpdateItemResponse removeResponse = removeFromSet(
                dynamoDbClient, tableName, key, "Tags", tagsToRemove, false //
Not a number set
                );

            System.out.println("Updated set after removal: " +
removeResponse.attributes());
```

```
            // Example 6: Check if a value exists in a set
            System.out.println("\nExample 6: Checking if a value exists in a
set");
            Map<String, Object> checkResult = checkIfValueInSet(dynamoDbClient,
tableName, key, "Tags", "Electronics");

            System.out.println("Check result: " + checkResult.get("message"));

            // Example 7: Get the current value of a set attribute
            System.out.println("\nExample 7: Getting the current value of a set
attribute");
            AttributeValue currentStringSet = getSetAttribute(dynamoDbClient,
tableName, key, "Tags");

            if (currentStringSet != null && currentStringSet.ss() != null) {
                System.out.println("Current string set values: " +
currentStringSet.ss());
            } else {
                System.out.println("String set attribute not found");
            }

            AttributeValue currentNumberSet = getSetAttribute(dynamoDbClient,
tableName, key, "Ratings");

            if (currentNumberSet != null && currentNumberSet.ns() != null) {
                System.out.println("Current number set values: " +
currentNumberSet.ns());
            } else {
                System.out.println("Number set attribute not found");
            }

            // Explain set operations
            System.out.println("\nKey points about DynamoDB set operations:");
            System.out.println(
                "1. DynamoDB supports three set types: string sets (SS), number
sets (NS), and binary sets (BS)");
            System.out.println("2. Sets can only contain elements of the same
type");
            System.out.println("3. Use ADD to add elements to a set");
            System.out.println("4. Use DELETE to remove elements from a set");
            System.out.println("5. Sets automatically remove duplicate values");
            System.out.println("6. Sets are unordered collections");
```

```
            System.out.println("7. Use the contains function to check if a value
 exists in a set");
            System.out.println("8. You can create a set with multiple values in a
 single operation");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスのUpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してセットオペレーションをデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Add elements to a set attribute.
 *
 * This function demonstrates using the ADD operation to add elements to a set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The values to add to the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function addToSet(
  config,
```

```
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
  }

  // Define the update parameters using ADD
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `ADD ${setName} :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Remove elements from a set attribute.
 *
 * This function demonstrates using the DELETE operation to remove elements from
 a set.
```

```
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The values to remove from the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function removeFromSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
  }

  // Define the update parameters using DELETE
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `DELETE ${setName} :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
  };
```

```
  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Create a new set attribute with initial values.
 *
 * This function demonstrates using the SET operation to create a new set
 attribute.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The initial values for the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function createSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create the appropriate set type
  let setValues;
  if (setType === 'string') {
    setValues = new Set(values.map(String));
  } else if (setType === 'number') {
    setValues = new Set(values.map(Number));
  } else if (setType === 'binary') {
    setValues = new Set(values);
  } else {
    throw new Error(`Unsupported set type: ${setType}`);
  }
```

```
  // Define the update parameters using SET
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${setName} = :values`,
    ExpressionAttributeValues: {
      ":values": setValues
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Replace an entire set attribute with a new set of values.
 *
 * This function demonstrates using the SET operation to replace an entire set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @param {Array} values - The new values for the set
 * @param {string} setType - The type of set ('string', 'number', or 'binary')
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function replaceSet(
  config,
  tableName,
  key,
  setName,
  values,
  setType = 'string'
) {
  // This is the same as createSet, but included for clarity of intent
  return await createSet(config, tableName, key, setName, values, setType);
}

/**
 * Remove the last element from a set and handle the empty set case.
```

```
 *
 * This function demonstrates what happens when you delete the last element of a
 set.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} setName - The name of the set attribute
 * @returns {Promise<Object>} - The result of the operation
 */
async function removeLastElementFromSet(
  config,
  tableName,
  key,
  setName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // First, get the current item to check the set
  const currentItem = await getItem(config, tableName, key);

  // Check if the set exists and has elements
  if (!currentItem || !currentItem[setName] || currentItem[setName].size === 0) {
    return {
      success: false,
      message: "Set doesn't exist or is already empty",
      item: currentItem
    };
  }

  // Get the set values
  const setValues = Array.from(currentItem[setName]);

  // If there's only one element left, remove the attribute entirely
  if (setValues.length === 1) {
    // Define the update parameters to remove the attribute
    const params = {
      TableName: tableName,
      Key: key,
      UpdateExpression: `REMOVE ${setName}`,
      ReturnValues: "UPDATED_NEW"
    };
```

```
    // Perform the update operation
    await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Last element removed, attribute has been deleted",
      removedValue: setValues[0]
    };
  } else {
    // Otherwise, remove just the last element
    // Create a set with just the last element
    const lastElement = setValues[setValues.length - 1];
    const setType = typeof lastElement === 'number' ? 'number' : 'string';

    // Remove the last element
    const response = await removeFromSet(
      config,
      tableName,
      key,
      setName,
      [lastElement],
      setType
    );

    return {
      success: true,
      message: "Last element removed, set still contains elements",
      removedValue: lastElement,
      remainingSet: response.Attributes[setName]
    };
  }
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
```

```
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

AWS SDK for JavaScript を使用したセットオペレーションの使用例。

```
/**
 * Example of how to work with sets in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Users";
  const key = { UserId: "U12345" };

  console.log("Demonstrating set operations in DynamoDB");

  try {
    // Example 1: Create a string set
    console.log("\nExample 1: Creating a string set");
    const response1 = await createSet(
      config,
      tableName,
      key,
```

```
      "Interests",
      ["Reading", "Hiking", "Cooking"],
      "string"
    );

    console.log("Created set:", response1.Attributes);

    // Example 2: Add elements to a set
    console.log("\nExample 2: Adding elements to a set");
    const response2 = await addToSet(
      config,
      tableName,
      key,
      "Interests",
      ["Photography", "Travel"],
      "string"
    );

    console.log("Updated set after adding elements:", response2.Attributes);

    // Example 3: Remove elements from a set
    console.log("\nExample 3: Removing elements from a set");
    const response3 = await removeFromSet(
      config,
      tableName,
      key,
      "Interests",
      ["Cooking"],
      "string"
    );

    console.log("Updated set after removing elements:", response3.Attributes);

    // Example 4: Create a number set
    console.log("\nExample 4: Creating a number set");
    const response4 = await createSet(
      config,
      tableName,
      key,
      "FavoriteNumbers",
      [7, 42, 99],
      "number"
    );
```

```
    console.log("Created number set:", response4.Attributes);

    // Example 5: Replace an entire set
    console.log("\nExample 5: Replacing an entire set");
    const response5 = await replaceSet(
      config,
      tableName,
      key,
      "Interests",
      ["Gaming", "Movies", "Music"],
      "string"
    );

    console.log("Replaced set:", response5.Attributes);

    // Example 6: Remove the last element from a set
    console.log("\nExample 6: Removing the last element from a set");

    // First, create a set with just one element
    await createSet(
      config,
      tableName,
      { UserId: "U67890" },
      "Tags",
      ["LastTag"],
      "string"
    );

    // Then, remove the last element
    const response6 = await removeLastElementFromSet(
      config,
      tableName,
      { UserId: "U67890" },
      "Tags"
    );

    console.log(response6.message);
    console.log("Removed value:", response6.removedValue);

    // Get the final state of the items
    console.log("\nFinal state of the items:");
    const item1 = await getItem(config, tableName, key);
    console.log("User U12345:", JSON.stringify(item1, null, 2));
```

```
    const item2 = await getItem(config, tableName, { UserId: "U67890" });
    console.log("User U67890:", JSON.stringify(item2, null, 2));

    // Explain set operations
    console.log("\nKey points about set operations in DynamoDB:");
    console.log("1. Use ADD to add elements to a set (duplicates are
 automatically removed)");
    console.log("2. Use DELETE to remove elements from a set");
    console.log("3. Use SET to create a new set or replace an existing one");
    console.log("4. DynamoDB supports three types of sets: string sets, number
 sets, and binary sets");
    console.log("5. When you delete the last element from a set, the attribute
 remains as an empty set");
    console.log("6. To remove an empty set, use the REMOVE operation");
    console.log("7. Sets automatically maintain unique values (no duplicates)");
    console.log("8. You cannot mix data types within a set");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスのUpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してセットオペレーションをデモンストレーションします。

```
import boto3
from typing import Any, Dict, List


def create_set_attribute(
    table_name: str,
    key: Dict[str, Any],
    set_name: str,
    set_values: List[Any],
```

```
    set_type: str = "string",
) -> Dict[str, Any]:
    """
    Create a new set attribute or add elements to an existing set.

    This function demonstrates how to use the ADD operation to create a new set
    or add elements to an existing set.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        set_values (List[Any]): The values to add to the set.
        set_type (str, optional): The type of set to create: "string", "number",
or "binary".
            Defaults to "string".

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a DynamoDB set based on the specified type
    if set_type == "string":
        dynamo_set = set(str(value) for value in set_values)
    elif set_type == "number":
        # We need to use actual float values for the DynamoDB API
        # but mypy expects strings in sets, so we need to use type: ignore
        dynamo_set = set(float(value) for value in set_values)  # type: ignore
    else:  # binary set is not directly supported in high-level API, handled
differently
        raise ValueError("Binary sets are not supported in this example")

    # Use the ADD operation to create or update the set
    response = table.update_item(
        Key=key,
        UpdateExpression="ADD #set_attr :set_values",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":set_values": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def add_to_set(
    table_name: str, key: Dict[str, Any], set_name: str, values_to_add: List[Any]
) -> Dict[str, Any]:
    """
    Add elements to an existing set attribute.

    This function demonstrates how to use the ADD operation to add elements to an
 existing set.
    If the set doesn't exist, it will be created.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        values_to_add (List[Any]): The values to add to the set.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a set (assuming string set for simplicity)
    dynamo_set = set(str(value) for value in values_to_add)

    # Use the ADD operation to add values to the set
    response = table.update_item(
        Key=key,
        UpdateExpression="ADD #set_attr :values_to_add",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":values_to_add": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )

    return response


def remove_from_set(
```

```
    table_name: str, key: Dict[str, Any], set_name: str, values_to_remove:
 List[Any]
) -> Dict[str, Any]:
    """
    Remove elements from a set attribute.

    This function demonstrates how to use the DELETE operation to remove elements
 from a set.
    If the last element is removed, the attribute will be deleted entirely.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        values_to_remove (List[Any]): The values to remove from the set.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Convert the list to a set (assuming string set for simplicity)
    dynamo_set = set(str(value) for value in values_to_remove)

    # Use the DELETE operation to remove values from the set
    response = table.update_item(
        Key=key,
        UpdateExpression="DELETE #set_attr :values_to_remove",
        ExpressionAttributeNames={"#set_attr": set_name},
        ExpressionAttributeValues={":values_to_remove": dynamo_set},
        ReturnValues="UPDATED_NEW",
    )

    return response


def check_if_set_exists(table_name: str, key: Dict[str, Any], set_name: str) ->
 bool:
    """
    Check if a set attribute exists in an item.
```

```
    This function demonstrates how to check if a set attribute exists after
    potentially removing all elements from it.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to check.
        set_name (str): The name of the set attribute.

    Returns:
        bool: True if the set attribute exists, False otherwise.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Get the item
    response = table.get_item(
        Key=key, ProjectionExpression="#set_attr",
 ExpressionAttributeNames={"#set_attr": set_name}
    )

    # Check if the item exists and has the set attribute
    return "Item" in response and set_name in response["Item"]


def demonstrate_last_element_removal(
    table_name: str, key: Dict[str, Any], set_name: str
) -> Dict[str, Any]:
    """
    Demonstrate what happens when you remove the last element from a set.

    This function creates a set with a single element, then removes that element,
    showing that the attribute is completely removed when the last element is
 deleted.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Step 1: Create a set with a single element
```

```
    create_response = create_set_attribute(
        table_name=table_name,
        key=key,
        set_name=set_name,
        set_values=["last_element"],
        set_type="string",
    )

    # Step 2: Check that the set exists
    exists_before = check_if_set_exists(table_name, key, set_name)

    # Step 3: Remove the last element
    delete_response = remove_from_set(
        table_name=table_name, key=key, set_name=set_name,
 values_to_remove=["last_element"]
    )

    # Step 4: Check if the set still exists
    exists_after = check_if_set_exists(table_name, key, set_name)

    # Return the results
    return {
        "create_response": create_response,
        "exists_before": exists_before,
        "delete_response": delete_response,
        "exists_after": exists_after,
    }


def work_with_number_set(
    table_name: str,
    key: Dict[str, Any],
    set_name: str,
    initial_values: List[float],
    values_to_add: List[float],
    values_to_remove: List[float],
) -> Dict[str, Any]:
    """
    Demonstrate working with a number set in DynamoDB.

    This function shows how to create and manipulate a set of numbers.

    Args:
        table_name (str): The name of the DynamoDB table.
```

```
        key (Dict[str, Any]): The primary key of the item to update.
        set_name (str): The name of the set attribute.
        initial_values (List[float]): The initial values for the set.
        values_to_add (List[float]): Values to add to the set.
        values_to_remove (List[float]): Values to remove from the set.

    Returns:
        Dict[str, Any]: A dictionary containing the responses from each
operation.
    """
    # Step 1: Create the number set
    create_response = create_set_attribute(
        table_name=table_name,
        key=key,
        set_name=set_name,
        set_values=initial_values,
        set_type="number",
    )

    # Step 2: Add more numbers to the set
    add_response = add_to_set(
        table_name=table_name, key=key, set_name=set_name,
values_to_add=values_to_add
    )

    # Step 3: Remove some numbers from the set
    remove_response = remove_from_set(
        table_name=table_name, key=key, set_name=set_name,
values_to_remove=values_to_remove
    )

    # Step 4: Get the final state
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    get_response = table.get_item(
        Key=key,
        ProjectionExpression=f"#{set_name}",
        ExpressionAttributeNames={f"#{set_name}": set_name},
    )

    # Return all responses
    return {
        "create_response": create_response,
```

```
        "add_response": add_response,
        "remove_response": remove_response,
        "final_state": get_response.get("Item", {}),
    }
```

AWS SDK for Python (Boto3) を使用したセットオペレーションの使用例。

```
def example_usage():
    """Example of how to use the set operations functions."""
    # Example parameters
    table_name = "UserPreferences"
    key = {"UserId": "user123"}

    print("Example 1: Creating a string set attribute")
    try:
        response = create_set_attribute(
            table_name=table_name,
            key=key,
            set_name="FavoriteTags",
            set_values=["AWS", "DynamoDB", "NoSQL"],
            set_type="string",
        )
        print(f"Set attribute created successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error creating set attribute: {e}")

    print("\nExample 2: Adding elements to an existing set")
    try:
        response = add_to_set(
            table_name=table_name,
            key=key,
            set_name="FavoriteTags",
            values_to_add=["Database", "Serverless"],
        )
        print(f"Elements added to set successfully: {response.get('Attributes',
 {})}")
    except Exception as e:
        print(f"Error adding to set: {e}")
```

```
    print("\nExample 3: Removing elements from a set")
    try:
        response = remove_from_set(
            table_name=table_name, key=key, set_name="FavoriteTags",
values_to_remove=["NoSQL"]
        )
        print(f"Elements removed from set successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error removing from set: {e}")

    print("\nExample 4: Demonstrating what happens when you remove the last
element from a set")
    try:
        results = demonstrate_last_element_removal(
            table_name=table_name, key={"UserId": "tempUser"},
set_name="SingleElementSet"
        )

        print(f"Set exists before removal: {results['exists_before']}")
        print(f"Set exists after removal: {results['exists_after']}")

        if not results["exists_after"]:
            print("The set attribute was completely removed when the last element
was deleted.")
        else:
            print("The set attribute still exists after removing the last
element.")
    except Exception as e:
        print(f"Error in last element removal demonstration: {e}")

    print("\nExample 5: Working with a number set")
    try:
        results = work_with_number_set(
            table_name=table_name,
            key={"UserId": "user456"},
            set_name="LuckyNumbers",
            initial_values=[7, 13, 42],
            values_to_add=[99, 100],
            values_to_remove=[13],
        )

        print(f"Initial number set: {results['create_response'].get('Attributes',
{})}")
```

```
        print(f"After adding numbers: {results['add_response'].get('Attributes',
{})}")
        print(f"After removing numbers:
{results['remove_response'].get('Attributes', {})}")
        print(f"Final state: {results['final_state']}")
    except Exception as e:
        print(f"Error working with number set: {e}")

    print("\nKey Points About DynamoDB Sets:")
    print("1. Sets can only contain elements of the same type (string, number, or
binary)")
    print("2. Sets automatically eliminate duplicate values")
    print("3. The ADD operation creates a set if it doesn't exist")
    print("4. The DELETE operation removes specified elements from a set")
    print("5. When the last element is removed from a set, the entire attribute
is deleted")
    print("6. Empty sets are not allowed in DynamoDB")
    print("7. Sets are unordered collections")
    print("8. The ADD operation is atomic for sets")
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

PartiQL ステートメントのバッチと AWS SDK を使用して DynamoDB テーブルにクエリを実行する

次のコード例は、以下を実行する方法を示しています。

- 複数の SELECT ステートメントを実行して、項目のバッチを取得する。
- 複数の INSERT ステートメントを実行して、項目のバッチを追加する。
- 複数の UPDATE ステートメントを実行して、項目のバッチを更新する。
- 複数の DELETE ステートメントを実行して、項目のバッチを削除する。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// Before you run this example, download 'movies.json' from
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
GettingStarted.Js.02.html,
// and put it in the same folder as the example.

// Separator for the console display.
var SepBar = new string('-', 80);
const string tableName = "movie_table";
const string movieFileName = @"..\..\..\..\..\..\..\..\resources\sample_files
\movies.json";

DisplayInstructions();

// Create the table and wait for it to be active.
Console.WriteLine($"Creating the movie table: {tableName}");

var success = await DynamoDBMethods.CreateMovieTableAsync(tableName);
if (success)
{
    Console.WriteLine($"Successfully created table: {tableName}.");
}

WaitForEnter();

// Add movie information to the table from moviedata.json. See the
// instructions at the top of this file to download the JSON file.
Console.WriteLine($"Inserting movies into the new table. Please wait...");
success = await PartiQLBatchMethods.InsertMovies(tableName, movieFileName);
if (success)
{
    Console.WriteLine("Movies successfully added to the table.");
```

```
}
else
{
    Console.WriteLine("Movies could not be added to the table.");
}

WaitForEnter();

// Update multiple movies by using the BatchExecute statement.
var title1 = "Star Wars";
var year1 = 1977;
var title2 = "Wizard of Oz";
var year2 = 1939;

Console.WriteLine($"Updating two movies with producer information: {title1} and
 {title2}.");
success = await PartiQLBatchMethods.GetBatch(tableName, title1, title2, year1,
 year2);
if (success)
{
    Console.WriteLine($"Successfully retrieved {title1} and {title2}.");
}
else
{
    Console.WriteLine("Select statement failed.");
}

WaitForEnter();

// Update multiple movies by using the BatchExecute statement.
var producer1 = "LucasFilm";
var producer2 = "MGM";

Console.WriteLine($"Updating two movies with producer information: {title1} and
 {title2}.");
success = await PartiQLBatchMethods.UpdateBatch(tableName, producer1, title1,
 year1, producer2, title2, year2);
if (success)
{
    Console.WriteLine($"Successfully updated {title1} and {title2}.");
}
else
{
    Console.WriteLine("Update failed.");
```

```
}

WaitForEnter();

// Delete multiple movies by using the BatchExecute statement.
Console.WriteLine($"Now we will delete {title1} and {title2} from the table.");
success = await PartiQLBatchMethods.DeleteBatch(tableName, title1, year1, title2,
 year2);

if (success)
{
    Console.WriteLine($"Deleted {title1} and {title2}");
}
else
{
    Console.WriteLine($"could not delete {title1} or {title2}");
}

WaitForEnter();

// DNow that the PartiQL Batch scenario is complete, delete the movie table.
success = await DynamoDBMethods.DeleteTableAsync(tableName);

if (success)
{
    Console.WriteLine($"Successfully deleted {tableName}");
}
else
{
    Console.WriteLine($"Could not delete {tableName}");
}

/// <summary>
/// Displays the description of the application on the console.
/// </summary>
void DisplayInstructions()
{
    Console.Clear();
    Console.WriteLine();
    Console.Write(new string(' ', 24));
    Console.WriteLine("DynamoDB PartiQL Basics Example");
    Console.WriteLine(SepBar);
    Console.WriteLine("This demo application shows the basics of using Amazon
 DynamoDB with the AWS SDK for");
```

```
    Console.WriteLine(".NET version 3.7 and .NET 6.");
    Console.WriteLine(SepBar);
    Console.WriteLine("Creates a table by using the CreateTable method.");
    Console.WriteLine("Gets multiple movies by using a PartiQL SELECT
 statement.");
    Console.WriteLine("Updates multiple movies by using the ExecuteBatch
 method.");
    Console.WriteLine("Deletes multiple movies by using a PartiQL DELETE
 statement.");
    Console.WriteLine("Cleans up the resources created for the demo by deleting
 the table.");
    Console.WriteLine(SepBar);

    WaitForEnter();
}

/// <summary>
/// Simple method to wait for the <Enter> key to be pressed.
/// </summary>
void WaitForEnter()
{
    Console.WriteLine("\nPress <Enter> to continue.");
    Console.Write(SepBar);
    _ = Console.ReadLine();
}
```

```
        /// <summary>
        /// Gets movies from the movie table by
        /// using an Amazon DynamoDB PartiQL SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year1">The year of the first movie.</param>
        /// <param name="year2">The year of the second movie.</param>
        /// <returns>True if successful.</returns>
        public static async Task<bool> GetBatch(
            string tableName,
            string title1,
            string title2,
            int year1,
            int year2)
        {
```

```
            var getBatch = $"SELECT * FROM {tableName} WHERE title = ? AND year
= ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = getBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            if (response.Responses.Count > 0)
            {
                response.Responses.ForEach(r =>
                {
                    if (r.Item.Any())
                    {

Console.WriteLine($"{r.Item["title"]}\t{r.Item["year"]}");
                    }
                });
                return true;
            }
            else
```

```
            {
                Console.WriteLine($"Couldn't find either {title1} or {title2}.");
                return false;
            }

        }

        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table into which the movie
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
```

```
                                Parameters = new List<AttributeValue>
                                {
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
```

```
            {
                return null!;
            }

            using var sr = new StreamReader(movieFileName);
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
            }
            else
            {
                return null!;
            }
        }

        /// <summary>
        /// Updates information for multiple movies.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// movies to be updated.</param>
        /// <param name="producer1">The producer name for the first movie
        /// to update.</param>
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year that the first movie was released.</
param>
        /// <param name="producer2">The producer name for the second
        /// movie to update.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year that the second movie was released.</
param>
        /// <returns>A Boolean value that indicates the success of the update.</
returns>
        public static async Task<bool> UpdateBatch(
            string tableName,
            string producer1,
            string title1,
            int year1,
            string producer2,
            string title2,
            int year2)
```

```
        {

            string updateBatch = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer1 },
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = producer2 },
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
BatchExecuteStatementRequest
            {
                Statements = statements,
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }

        /// <summary>
        /// Deletes multiple movies using a PartiQL BatchExecuteAsync
        /// statement.
        /// </summary>
        /// <param name="tableName">The name of the table containing the
        /// moves that will be deleted.</param>
```

```
        /// <param name="title1">The title of the first movie.</param>
        /// <param name="year1">The year the first movie was released.</param>
        /// <param name="title2">The title of the second movie.</param>
        /// <param name="year2">The year the second movie was released.</param>
        /// <returns>A Boolean value indicating the success of the operation.</
returns>
        public static async Task<bool> DeleteBatch(
            string tableName,
            string title1,
            int year1,
            string title2,
            int year2)
        {

            string updateBatch = $"DELETE FROM {tableName} WHERE title = ? AND
 year = ?";
            var statements = new List<BatchStatementRequest>
            {
                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title1 },
                        new AttributeValue { N = year1.ToString() },
                    },
                },

                new BatchStatementRequest
                {
                    Statement = updateBatch,
                    Parameters = new List<AttributeValue>
                    {
                        new AttributeValue { S = title2 },
                        new AttributeValue { N = year2.ToString() },
                    },
                }
            };

            var response = await Client.BatchExecuteStatementAsync(new
 BatchExecuteStatementRequest
            {
                Statements = statements,
            });
```

```
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「BatchExecuteStatement」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        //  1. Create a table. (CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::partiqlBatchExecuteScenario(clientConfig);

            // 7. Delete the table. (DeleteTable)
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
        }

//! Scenario to modify and query a DynamoDB table using PartiQL batch statements.
/*!
  \sa partiqlBatchExecuteScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool AwsDoc::DynamoDB::partiqlBatchExecuteScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {

    // 2. Add multiple movies using "Insert" statements. (BatchExecuteStatement)
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    std::vector<Aws::String> titles;
```

```
    std::vector<float> ratings;
    std::vector<int> years;
    std::vector<Aws::String> plots;
    Aws::String doAgain = "n";
    do {
        Aws::String aTitle = askQuestion(
                "Enter the title of a movie you want to add to the table: ");
        titles.push_back(aTitle);
        int aYear = askQuestionForInt("What year was it released? ");
        years.push_back(aYear);
        float aRating = askQuestionForFloatRange(
                "On a scale of 1 - 10, how do you rate it? ",
                1, 10);
        ratings.push_back(aRating);
        Aws::String aPlot = askQuestion("Summarize the plot for me: ");
        plots.push_back(aPlot);

        doAgain = askQuestion(Aws::String("Would you like to add more movies? (y/
n) "));
    } while (doAgain == "y");

    std::cout << "Adding " << titles.size()
              << (titles.size() == 1 ? " movie " : " movies ")
              << "to the table using a batch \"INSERT\" statement." << std::endl;

    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());

        std::stringstream sqlStream;
        sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                  << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                  << INFO_KEY << "': ?}";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);

            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

 attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
```

```
            // Create attribute for the info map.
            Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute
= Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            ratingAttribute->SetN(ratings[i]);
            infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

            std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
                    ALLOCATION_TAG.c_str());
            plotAttribute->SetS(plots[i]);
            infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
            attributes.push_back(infoMapAttribute);
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add the movies: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "Retrieving the movie data with a batch \"SELECT\" statement."
              << std::endl;

    // 3. Get the data for multiple movies using "Select" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
```

```
                << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (outcome.IsSuccess()) {
            const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

            for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

                printMovieInfo(item);
            }
        }
        else {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }
```

```
   // 4. Update the data for multiple movies using "Update" statements.
(BatchExecuteStatement)

   for (size_t i = 0; i < titles.size(); ++i) {
       ratings[i] = askQuestionForFloatRange(
               Aws::String("\nLet's update your the movie, \"") + titles[i] +
               ".\nYou rated it  " + std::to_string(ratings[i])
               + ", what new rating would you give it? ", 1, 10);
   }

   std::cout << "Updating the movie with a batch \"UPDATE\" statement." <<
std::endl;

   {
       Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
               titles.size());

       std::stringstream sqlStream;
       sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                 << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                 << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";


       std::string sql(sqlStream.str());

       for (size_t i = 0; i < statements.size(); ++i) {
           statements[i].SetStatement(sql);

           Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
           attributes.push_back(
                   Aws::DynamoDB::Model::AttributeValue().SetN(ratings[i]));
           attributes.push_back(
                   Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
           statements[i].SetParameters(attributes);
       }

       Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

       request.SetStatements(statements);
       Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
               request);
```

```
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    std::cout << "Retrieving the updated movie data with a batch \"SELECT\"
statement."
              << std::endl;

    // 5. Get the updated data for multiple movies using "Select" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;

        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
dynamoClient.BatchExecuteStatement(
                request);
        if (outcome.IsSuccess()) {
            const Aws::DynamoDB::Model::BatchExecuteStatementResult &result =
outcome.GetResult();
```

```
            const Aws::Vector<Aws::DynamoDB::Model::BatchStatementResponse>
&responses = result.GetResponses();

            for (const Aws::DynamoDB::Model::BatchStatementResponse &response:
responses) {
                const Aws::Map<Aws::String, Aws::DynamoDB::Model::AttributeValue>
&item = response.GetItem();

                printMovieInfo(item);
            }
        }
        else {
            std::cerr << "Failed to retrieve the movies information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    std::cout << "Deleting the movie data with a batch \"DELETE\" statement."
              << std::endl;

    // 6. Delete multiple movies using "Delete" statements.
(BatchExecuteStatement)
    {
        Aws::Vector<Aws::DynamoDB::Model::BatchStatementRequest> statements(
                titles.size());
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        std::string sql(sqlStream.str());

        for (size_t i = 0; i < statements.size(); ++i) {
            statements[i].SetStatement(sql);
            Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
            attributes.push_back(
                    Aws::DynamoDB::Model::AttributeValue().SetS(titles[i]));

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(years[i]));
            statements[i].SetParameters(attributes);
        }

        Aws::DynamoDB::Model::BatchExecuteStatementRequest request;
```

```
        request.SetStatements(statements);

        Aws::DynamoDB::Model::BatchExecuteStatementOutcome outcome =
 dynamoClient.BatchExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movies: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    return true;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;

    {
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
```

```
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
                std::cerr << "Failed to create table: "
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
```

```
                  << std::endl;
    }

    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
```

```
                                        const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
        count++;
    }
    return false;
}
```

- API の詳細についてはAWS SDK for C++ API リファレンスの「BatchExecuteStatement」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成し、PartiQL クエリのバッチを実行するシナリオを実行します。

```
import (
 "context"
 "fmt"
 "log"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunPartiQLBatchScenario shows you how to use the AWS SDK for Go
// to run batches of PartiQL statements to query a table that stores data about
 movies.
//
//   - Use batches of PartiQL statements to add, get, update, and delete data for
//     individual movies.
//
// This example creates an Amazon DynamoDB service client from the specified
 sdkConfig so that
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLBatchScenario(ctx context.Context, sdkConfig aws.Config, tableName
 string) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
```

```
}()

log.Println(strings.Repeat("-", 88))
log.Println("Welcome to the Amazon DynamoDB PartiQL batch demo.")
log.Println(strings.Repeat("-", 88))

tableBasics := actions.TableBasics{
 DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
 TableName:      tableName,
}
runner := actions.PartiQLRunner{
 DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
 TableName:      tableName,
}

exists, err := tableBasics.TableExists(ctx)
if err != nil {
 panic(err)
}
if !exists {
 log.Printf("Creating table %v...\n", tableName)
 _, err = tableBasics.CreateMovieTable(ctx)
 if err != nil {
  panic(err)
 } else {
  log.Printf("Created table %v.\n", tableName)
 }
} else {
 log.Printf("Table %v already exists.\n", tableName)
}
log.Println(strings.Repeat("-", 88))

currentYear, _, _ := time.Now().Date()
customMovies := []actions.Movie{{
 Title: "House PartiQL",
 Year:  currentYear - 5,
 Info: map[string]interface{}{
  "plot":   "Wacky high jinks result from querying a mysterious database.",
  "rating": 8.5}}, {
 Title: "House PartiQL 2",
 Year:  currentYear - 3,
 Info: map[string]interface{}{
  "plot":   "Moderate high jinks result from querying another mysterious
database.",
```

```
  "rating": 6.5}}, {
 Title: "House PartiQL 3",
 Year:  currentYear - 1,
 Info: map[string]interface{}{
  "plot":   "Tepid high jinks result from querying yet another mysterious
database.",
  "rating": 2.5},
},
}

log.Printf("Inserting a batch of movies into table '%v'.\n", tableName)
err = runner.AddMovieBatch(ctx, customMovies)
if err == nil {
 log.Printf("Added %v movies to the table.\n", len(customMovies))
}
log.Println(strings.Repeat("-", 88))

log.Println("Getting data for a batch of movies.")
movies, err := runner.GetMovieBatch(ctx, customMovies)
if err == nil {
 for _, movie := range movies {
  log.Println(movie)
 }
}
log.Println(strings.Repeat("-", 88))

newRatings := []float64{7.7, 4.4, 1.1}
log.Println("Updating a batch of movies with new ratings.")
err = runner.UpdateMovieBatch(ctx, customMovies, newRatings)
if err == nil {
 log.Printf("Updated %v movies with new ratings.\n", len(customMovies))
}
log.Println(strings.Repeat("-", 88))

log.Println("Getting projected data from the table to verify our update.")
log.Println("Using a page size of 2 to demonstrate paging.")
projections, err := runner.GetAllMovies(ctx, 2)
if err == nil {
 log.Println("All movies:")
 for _, projection := range projections {
  log.Println(projection)
 }
}
log.Println(strings.Repeat("-", 88))
```

```
 log.Println("Deleting a batch of movies.")
 err = runner.DeleteMovieBatch(ctx, customMovies)
 if err == nil {
  log.Printf("Deleted %v movies.\n", len(customMovies))
 }

 err = tableBasics.DeleteTable(ctx)
 if err == nil {
  log.Printf("Deleted table %v.\n", tableBasics.TableName)
 }

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                `dynamodbav:"title"`
 Year  int                   `dynamodbav:"year"`
```

```
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

PartiQL ステートメントを実行する struct およびメソッドを作成します。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
```

```
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}



// AddMovieBatch runs a batch of PartiQL INSERT statements to add multiple movies
 to the
// DynamoDB table.
func (runner PartiQLRunner) AddMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year, movie.Info})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(fmt.Sprintf(
    "INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't insert a batch of items with PartiQL. Here's why: %v\n",
 err)
 }
 return err
}



// GetMovieBatch runs a batch of PartiQL SELECT statements to get multiple movies
 from
// the DynamoDB table by title and year.
```

```
func (runner PartiQLRunner) GetMovieBatch(ctx context.Context, movies []Movie)
 ([]Movie, error) {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 output, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 var outMovies []Movie
 if err != nil {
  log.Printf("Couldn't get a batch of items with PartiQL. Here's why: %v\n", err)
 } else {
  for _, response := range output.Responses {
   var movie Movie
   err = attributevalue.UnmarshalMap(response.Item, &movie)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    outMovies = append(outMovies, movie)
   }
  }
 }
 return outMovies, err
}



// GetAllMovies runs a PartiQL SELECT statement to get all movies from the
 DynamoDB table.
// pageSize is not typically required and is used to show how to paginate the
 results.
```

```
// The results are projected to return only the title and rating of each movie.
func (runner PartiQLRunner) GetAllMovies(ctx context.Context, pageSize int32)
 ([]map[string]interface{}, error) {
 var output []map[string]interface{}
 var response *dynamodb.ExecuteStatementOutput
 var err error
 var nextToken *string
 for moreData := true; moreData; {
  response, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
   Statement: aws.String(
    fmt.Sprintf("SELECT title, info.rating FROM \"%v\"", runner.TableName)),
   Limit:     aws.Int32(pageSize),
   NextToken: nextToken,
  })
  if err != nil {
   log.Printf("Couldn't get movies. Here's why: %v\n", err)
   moreData = false
  } else {
   var pageOutput []map[string]interface{}
   err = attributevalue.UnmarshalListOfMaps(response.Items, &pageOutput)
   if err != nil {
    log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
   } else {
    log.Printf("Got a page of length %v.\n", len(response.Items))
    output = append(output, pageOutput...)
   }
   nextToken = response.NextToken
   moreData = nextToken != nil
  }
 }
 return output, err
}



// UpdateMovieBatch runs a batch of PartiQL UPDATE statements to update the
 rating of
// multiple movies that already exist in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovieBatch(ctx context.Context, movies []Movie,
 ratings []float64) error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
```

```
  params, err := attributevalue.MarshalList([]interface{}{ratings[index],
 movie.Title, movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }

 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't update the batch of movies. Here's why: %v\n", err)
 }
 return err
}



// DeleteMovieBatch runs a batch of PartiQL DELETE statements to remove multiple
 movies
// from the DynamoDB table.
func (runner PartiQLRunner) DeleteMovieBatch(ctx context.Context, movies []Movie)
 error {
 statementRequests := make([]types.BatchStatementRequest, len(movies))
 for index, movie := range movies {
  params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
  if err != nil {
   panic(err)
  }
  statementRequests[index] = types.BatchStatementRequest{
   Statement: aws.String(
    fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
 runner.TableName)),
   Parameters: params,
  }
 }
```

```
 _, err := runner.DynamoDbClient.BatchExecuteStatement(ctx,
 &dynamodb.BatchExecuteStatementInput{
  Statements: statementRequests,
 })
 if err != nil {
  log.Printf("Couldn't delete the batch of movies. Here's why: %v\n", err)
 }
 return err
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「BatchExecuteStatement」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
public class ScenarioPartiQLBatch {
    public static void main(String[] args) throws IOException {
        String tableName = "MoviesPartiQBatch";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println("Creating an Amazon DynamoDB table named " + tableName
            + " with a key named year and a sort key named title.");
        createTable(ddb, tableName);

        System.out.println("Adding multiple records into the " + tableName
            + " table using a batch command.");
        putRecordBatch(ddb);
```

```
        // Update multiple movies by using the BatchExecute statement.
        String title1 = "Star Wars";
        int year1 = 1977;
        String title2 = "Wizard of Oz";
        int year2 = 1939;

        System.out.println("Query two movies.");
        getBatch(ddb, tableName, title1, title2, year1, year2);

        System.out.println("Updating multiple records using a batch command.");
        updateTableItemBatch(ddb);

        System.out.println("Deleting multiple records using a batch command.");
        deleteItemBatch(ddb);

        System.out.println("Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static boolean getBatch(DynamoDbClient ddb, String tableName, String
title1, String title2, int year1, int year2) {
        String getBatch = "SELECT * FROM " + tableName + " WHERE title = ? AND
year = ?";

        List<BatchStatementRequest> statements = new ArrayList<>();
        statements.add(BatchStatementRequest.builder()
            .statement(getBatch)
            .parameters(AttributeValue.builder().s(title1).build(),
                AttributeValue.builder().n(String.valueOf(year1)).build())
            .build());
        statements.add(BatchStatementRequest.builder()
            .statement(getBatch)
            .parameters(AttributeValue.builder().s(title2).build(),
                AttributeValue.builder().n(String.valueOf(year2)).build())
            .build());

        BatchExecuteStatementRequest batchExecuteStatementRequest =
BatchExecuteStatementRequest.builder()
            .statements(statements)
            .build();

        try {
```

```
            BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchExecuteStatementRequest);
            if (!response.responses().isEmpty()) {
                response.responses().forEach(r -> {
                    System.out.println(r.item().get("title") + "\\t" +
r.item().get("year"));
                });
                return true;
            } else {
                System.out.println("Couldn't find either " + title1 + " or " +
title2 + ".");
                return false;
            }
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            return false;
        }
    }

    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());

        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
            .attributeName("title")
            .keyType(KeyType.RANGE) // Sort
            .build();
```

```
        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //  DynamoDB automatically
scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse = dbWaiter
                .waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void putRecordBatch(DynamoDbClient ddb) {
        String sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE {'year':?,
'title' : ?, 'info' : ?}";
        try {
            // Create three movies to add to the Amazon DynamoDB table.
            // Set data for Movie 1.
            List<AttributeValue> parameters = new ArrayList<>();

            AttributeValue att1 = AttributeValue.builder()
                .n("1977")
                .build();
```

```
            AttributeValue att2 = AttributeValue.builder()
                .s("Star Wars")
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s("No Information")
                .build();

            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);

            BatchStatementRequest statementRequestMovie1 =
BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parameters)
                .build();

            // Set data for Movie 2.
            List<AttributeValue> parametersMovie2 = new ArrayList<>();
            AttributeValue attMovie2 = AttributeValue.builder()
                .n("1939")
                .build();

            AttributeValue attMovie2A = AttributeValue.builder()
                .s("Wizard of Oz")
                .build();

            AttributeValue attMovie2B = AttributeValue.builder()
                .s("No Information")
                .build();

            parametersMovie2.add(attMovie2);
            parametersMovie2.add(attMovie2A);
            parametersMovie2.add(attMovie2B);

            BatchStatementRequest statementRequestMovie2 =
BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parametersMovie2)
                .build();

            // Set data for Movie 3.
            List<AttributeValue> parametersMovie3 = new ArrayList<>();
```

```
            AttributeValue attMovie3 = AttributeValue.builder()
                .n(String.valueOf("2022"))
                .build();

            AttributeValue attMovie3A = AttributeValue.builder()
                .s("My Movie 3")
                .build();

            AttributeValue attMovie3B = AttributeValue.builder()
                .s("No Information")
                .build();

            parametersMovie3.add(attMovie3);
            parametersMovie3.add(attMovie3A);
            parametersMovie3.add(attMovie3B);

            BatchStatementRequest statementRequestMovie3 =
BatchStatementRequest.builder()
                .statement(sqlStatement)
                .parameters(parametersMovie3)
                .build();

            // Add all three movies to the list.
            List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
            myBatchStatementList.add(statementRequestMovie1);
            myBatchStatementList.add(statementRequestMovie2);
            myBatchStatementList.add(statementRequestMovie3);

            BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
                .statements(myBatchStatementList)
                .build();

            BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchRequest);
            System.out.println("ExecuteStatement successful: " +
response.toString());
            System.out.println("Added new movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

```
    public static void updateTableItemBatch(DynamoDbClient ddb) {
        String sqlStatement = "UPDATE MoviesPartiQBatch SET info = 'directors\":
[\"Merian C. Cooper\",\"Ernest B. Schoedsack' where year=? and title=?";
        List<AttributeValue> parametersRec1 = new ArrayList<>();

        // Update three records.
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("My Movie 1")
            .build();

        parametersRec1.add(att1);
        parametersRec1.add(att2);

        BatchStatementRequest statementRequestRec1 =
 BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec1)
            .build();

        // Update record 2.
        List<AttributeValue> parametersRec2 = new ArrayList<>();
        AttributeValue attRec2 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec2a = AttributeValue.builder()
            .s("My Movie 2")
            .build();

        parametersRec2.add(attRec2);
        parametersRec2.add(attRec2a);
        BatchStatementRequest statementRequestRec2 =
 BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec2)
            .build();

        // Update record 3.
        List<AttributeValue> parametersRec3 = new ArrayList<>();
```

```
        AttributeValue attRec3 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec3a = AttributeValue.builder()
            .s("My Movie 3")
            .build();

        parametersRec3.add(attRec3);
        parametersRec3.add(attRec3a);
        BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec3)
            .build();

        // Add all three movies to the list.
        List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
        myBatchStatementList.add(statementRequestRec1);
        myBatchStatementList.add(statementRequestRec2);
        myBatchStatementList.add(statementRequestRec3);

        BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
            .statements(myBatchStatementList)
            .build();

        try {
            BatchExecuteStatementResponse response =
ddb.batchExecuteStatement(batchRequest);
            System.out.println("ExecuteStatement successful: " +
response.toString());
            System.out.println("Updated three movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Item was updated!");
    }

    public static void deleteItemBatch(DynamoDbClient ddb) {
        String sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year = ? and
title=?";
```

```
        List<AttributeValue> parametersRec1 = new ArrayList<>();

        // Specify three records to delete.
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("My Movie 1")
            .build();

        parametersRec1.add(att1);
        parametersRec1.add(att2);

        BatchStatementRequest statementRequestRec1 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec1)
            .build();

        // Specify record 2.
        List<AttributeValue> parametersRec2 = new ArrayList<>();
        AttributeValue attRec2 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();

        AttributeValue attRec2a = AttributeValue.builder()
            .s("My Movie 2")
            .build();

        parametersRec2.add(attRec2);
        parametersRec2.add(attRec2a);
        BatchStatementRequest statementRequestRec2 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec2)
            .build();

        // Specify record 3.
        List<AttributeValue> parametersRec3 = new ArrayList<>();
        AttributeValue attRec3 = AttributeValue.builder()
            .n(String.valueOf("2022"))
            .build();
```

```
        AttributeValue attRec3a = AttributeValue.builder()
            .s("My Movie 3")
            .build();

        parametersRec3.add(attRec3);
        parametersRec3.add(attRec3a);

        BatchStatementRequest statementRequestRec3 =
BatchStatementRequest.builder()
            .statement(sqlStatement)
            .parameters(parametersRec3)
            .build();

        // Add all three movies to the list.
        List<BatchStatementRequest> myBatchStatementList = new ArrayList<>();
        myBatchStatementList.add(statementRequestRec1);
        myBatchStatementList.add(statementRequestRec2);
        myBatchStatementList.add(statementRequestRec3);

        BatchExecuteStatementRequest batchRequest =
BatchExecuteStatementRequest.builder()
            .statements(myBatchStatementList)
            .build();

        try {
            ddb.batchExecuteStatement(batchRequest);
            System.out.println("Deleted three movies using a batch command.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{
        DeleteTableRequest request = DeleteTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
```

```
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }

    private static ExecuteStatementResponse
 executeStatementRequest(DynamoDbClient ddb, String statement,

 List<AttributeValue> parameters) {
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
            .statement(statement)
            .parameters(parameters)
            .build();

        return ddb.executeStatement(request);
    }
}
```

- API の詳細についてはAWS SDK for Java 2.x API リファレンスの「BatchExecuteStatement」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL ステートメントのバッチを実行します。

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
```

```
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "Cities";

export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
      "deleteTable",
      `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
      { type: "confirm", confirmAll },
    );
    const deleteTable = await input.handle({}, { confirmAll });
    if (deleteTable) {
      await client.send(new DeleteTableCommand({ tableName }));
    } else {
      console.warn(
        "Scenario could not run. Either delete ${tableName} or provide a unique
 table name.",
      );
      return;
    }
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ResourceNotFoundException"
    ) {
      // Do nothing. This means the table is not there.
    } else {
      throw caught;
```

```
    }
  }

  /**
   * Create a table.
   */

  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "name",
        // 'S' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "S",
      },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "name", KeyType: "HASH" }],
  });
  await client.send(createTableCommand);
  log(`Table created: ${tableName}.`);

  /**
   * Wait until the table is active.
   */

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");
```

```
  /**
   * Insert items.
   */

  log("Inserting cities into the table.");
  const addItemsStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
    Statements: [
      {
        Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
        Parameters: ["Alachua", 10712],
      },
      {
        Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
        Parameters: ["High Springs", 6415],
      },
    ],
  });
  await docClient.send(addItemsStatementCommand);
  log("Cities inserted.");

  /**
   * Select items.
   */

  log("Selecting cities from the table.");
  const selectItemsStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
    Statements: [
      {
        Statement: `SELECT * FROM ${tableName} WHERE name=?`,
        Parameters: ["Alachua"],
      },
      {
        Statement: `SELECT * FROM ${tableName} WHERE name=?`,
        Parameters: ["High Springs"],
      },
    ],
  });
  const selectItemResponse = await docClient.send(selectItemsStatementCommand);
  log(
```

```
    `Got cities: ${selectItemResponse.Responses.map(
      (r) => `${r.Item.name} (${r.Item.population})`,
    ).join(", ")}`,
  );

  /**
   * Update items.
   */

  log("Modifying the populations.");
  const updateItemStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
    Statements: [
      {
        Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
        Parameters: [10, "Alachua"],
      },
      {
        Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
        Parameters: [5, "High Springs"],
      },
    ],
  });
  await docClient.send(updateItemStatementCommand);
  log("Updated cities.");

  /**
   * Delete the items.
   */

  log("Deleting the cities.");
  const deleteItemStatementCommand = new BatchExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
    Statements: [
      {
        Statement: `DELETE FROM ${tableName} WHERE name=?`,
        Parameters: ["Alachua"],
      },
      {
        Statement: `DELETE FROM ${tableName} WHERE name=?`,
        Parameters: ["High Springs"],
      },
```

```
    ],
  });
  await docClient.send(deleteItemStatementCommand);
  log("Cities deleted.");

  /**
   * Delete the table.
   */

  log("Deleting the table.");
  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  await client.send(deleteTableCommand);
  log("Table deleted.");
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「BatchExecuteStatement」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun main() {
    val ddb = DynamoDbClient.fromEnvironment { region = "us-east-1" }
    val tableName = "MoviesPartiQBatch"
    println("Creating an Amazon DynamoDB table named $tableName with a key named
 id and a sort key named title.")
    createTablePartiQLBatch(ddb, tableName, "year")
    putRecordBatch(ddb)
    updateTableItemBatchBatch(ddb)
    deleteItemsBatch(ddb)
    deleteTablePartiQLBatch(tableName)
}
```

```
suspend fun createTablePartiQLBatch(
    ddb: DynamoDbClient,
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }

    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    val response = ddb.createTable(request)
    ddb.waitUntilTableExists {
        // suspend call
        tableName = tableNameVal
    }
    println("The table was successfully created
 ${response.tableDescription?.tableArn}")
```

```
}

suspend fun putRecordBatch(ddb: DynamoDbClient) {
    val sqlStatement = "INSERT INTO MoviesPartiQBatch VALUE {'year':?,
 'title' : ?, 'info' : ?}"

    // Create three movies to add to the Amazon DynamoDB table.
    val parametersMovie1 = mutableListOf<AttributeValue>()
    parametersMovie1.add(AttributeValue.N("2022"))
    parametersMovie1.add(AttributeValue.S("My Movie 1"))
    parametersMovie1.add(AttributeValue.S("No Information"))

    val statementRequestMovie1 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersMovie1
        }

    // Set data for Movie 2.
    val parametersMovie2 = mutableListOf<AttributeValue>()
    parametersMovie2.add(AttributeValue.N("2022"))
    parametersMovie2.add(AttributeValue.S("My Movie 2"))
    parametersMovie2.add(AttributeValue.S("No Information"))

    val statementRequestMovie2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersMovie2
        }

    // Set data for Movie 3.
    val parametersMovie3 = mutableListOf<AttributeValue>()
    parametersMovie3.add(AttributeValue.N("2022"))
    parametersMovie3.add(AttributeValue.S("My Movie 3"))
    parametersMovie3.add(AttributeValue.S("No Information"))

    val statementRequestMovie3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersMovie3
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
```

```
    myBatchStatementList.add(statementRequestMovie1)
    myBatchStatementList.add(statementRequestMovie2)
    myBatchStatementList.add(statementRequestMovie3)

    val batchRequest =
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }
    val response = ddb.batchExecuteStatement(batchRequest)
    println("ExecuteStatement successful: " + response.toString())
    println("Added new movies using a batch command.")
}

suspend fun updateTableItemBatchBatch(ddb: DynamoDbClient) {
    val sqlStatement =
        "UPDATE MoviesPartiQBatch SET info = 'directors\":[\"Merian C. Cooper\",
\"Ernest B. Schoedsack' where year=? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))
    val statementRequestRec1 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec1
        }

    // Update record 2.
    val parametersRec2 = mutableListOf<AttributeValue>()
    parametersRec2.add(AttributeValue.N("2022"))
    parametersRec2.add(AttributeValue.S("My Movie 2"))
    val statementRequestRec2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec2
        }

    // Update record 3.
    val parametersRec3 = mutableListOf<AttributeValue>()
    parametersRec3.add(AttributeValue.N("2022"))
    parametersRec3.add(AttributeValue.S("My Movie 3"))
    val statementRequestRec3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec3
```

```
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
    myBatchStatementList.add(statementRequestRec1)
    myBatchStatementList.add(statementRequestRec2)
    myBatchStatementList.add(statementRequestRec3)

    val batchRequest =
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }

    val response = ddb.batchExecuteStatement(batchRequest)
    println("ExecuteStatement successful: $response")
    println("Updated three movies using a batch command.")
    println("Items were updated!")
}

suspend fun deleteItemsBatch(ddb: DynamoDbClient) {
    // Specify three records to delete.
    val sqlStatement = "DELETE FROM MoviesPartiQBatch WHERE year = ? and title=?"
    val parametersRec1 = mutableListOf<AttributeValue>()
    parametersRec1.add(AttributeValue.N("2022"))
    parametersRec1.add(AttributeValue.S("My Movie 1"))

    val statementRequestRec1 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec1
        }

    // Specify record 2.
    val parametersRec2 = mutableListOf<AttributeValue>()
    parametersRec2.add(AttributeValue.N("2022"))
    parametersRec2.add(AttributeValue.S("My Movie 2"))
    val statementRequestRec2 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec2
        }

    // Specify record 3.
    val parametersRec3 = mutableListOf<AttributeValue>()
```

```
    parametersRec3.add(AttributeValue.N("2022"))
    parametersRec3.add(AttributeValue.S("My Movie 3"))
    val statementRequestRec3 =
        BatchStatementRequest {
            statement = sqlStatement
            parameters = parametersRec3
        }

    // Add all three movies to the list.
    val myBatchStatementList = mutableListOf<BatchStatementRequest>()
    myBatchStatementList.add(statementRequestRec1)
    myBatchStatementList.add(statementRequestRec2)
    myBatchStatementList.add(statementRequestRec3)

    val batchRequest =
        BatchExecuteStatementRequest {
            statements = myBatchStatementList
        }

    ddb.batchExecuteStatement(batchRequest)
    println("Deleted three movies using a batch command.")
}

suspend fun deleteTablePartiQLBatch(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}
```

- API の詳細については、AWS SDK for Kotlin API リファレンスの「BatchExecuteStatement」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\loadMovieData;
use function AwsUtilities\testable_readline;

class GettingStartedWithPartiQLBatch
{
    public function run()
    {
        echo("\n");
        echo("--------------------------------------\n");
        print("Welcome to the Amazon DynamoDB - PartiQL getting started demo
 using PHP!\n");
        echo("--------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDb\DynamoDBService();

        $tableName = "partiql_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
```

```
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
 $tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
            $movieYear = testable_readline("Year released: ");
        }
        $key = [
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
        $service->insertItemByPartiQLBatch($statement, $parameters);

        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
        ];

        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
```

```
        $service->updateItemByPartiQLBatch($statement, $parameters);
        echo "Movie added and updated.\n";

        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);

        $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
        echo "\nThe movie {$movie['Responses'][0]['Item']['title']['S']}
        was released in {$movie['Responses'][0]['Item']['year']['N']}.\n";
        echo "What rating would you like to give {$movie['Responses'][0]['Item']
['title']['S']}?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQLBatch($statement, $parameters);

        $movie = $service->getItemByPartiQLBatch($tableName, [$key]);
        echo "Okay, you have rated {$movie['Responses'][0]['Item']['title']
['S']}
        as a {$movie['Responses'][0]['Item']['rating']['N']}\n";

        $service->deleteItemByPartiQLBatch($statement, $parameters);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
 lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
```

```
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
 born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
 of).\n";
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}

    public function insertItemByPartiQLBatch(string $statement, array
 $parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
```

```
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function getItemByPartiQLBatch(string $tableName, array $keys): Result
    {
        $statements = [];
        foreach ($keys as $key) {
            list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);
            $statements[] = [
                'Statement' => "$statement",
                'Parameters' => $parameters,
            ];
        }

        return $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => $statements,
        ]);
    }

    public function updateItemByPartiQLBatch(string $statement, array
 $parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }

    public function deleteItemByPartiQLBatch(string $statement, array
 $parameters)
    {
        $this->dynamoDbClient->batchExecuteStatement([
            'Statements' => [
                [
```

```
                    'Statement' => "$statement",
                    'Parameters' => $parameters,
                ],
            ],
        ]);
    }
```

- API の詳細についてはAWS SDK for PHP API リファレンスの「BatchExecuteStatement」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL ステートメントのバッチを実行できるクラスを作成します。

```
from datetime import datetime
from decimal import Decimal
import logging
from pprint import pprint

import boto3
from botocore.exceptions import ClientError

from scaffold import Scaffold

logger = logging.getLogger(__name__)

class PartiQLBatchWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
```

```
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statements, param_list):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
the
        resource transforms input and output from plain old Python objects
(POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statements: The batch of PartiQL statements.
        :param param_list: The batch of PartiQL parameters that are associated
with
                           each statement. This list must be in the same order as
the
                           statements.
        :return: The responses returned from running the statements, if any.
        """
        try:
            output = self.dyn_resource.meta.client.batch_execute_statement(
                Statements=[
                    {"Statement": statement, "Parameters": params}
                    for statement, params in zip(statements, param_list)
                ]
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute batch of PartiQL statements because the
table "
                    "does not exist."
                )
            else:
                logger.error(
                    "Couldn't execute batch of PartiQL statements. Here's why:
%s: %s",
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
```

```
            raise
        else:
            return output
```

テーブルを作成し、PartiQL クエリをバッチで実行するシナリオを実行します。

```
def run_scenario(scaffold, wrapper, table_name):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB PartiQL batch statement demo.")
    print("-" * 88)

    print(f"Creating table '{table_name}' for the demo...")
    scaffold.create_table(table_name)
    print("-" * 88)

    movie_data = [
        {
            "title": f"House PartiQL",
            "year": datetime.now().year - 5,
            "info": {
                "plot": "Wacky high jinks result from querying a mysterious
 database.",
                "rating": Decimal("8.5"),
            },
        },
        {
            "title": f"House PartiQL 2",
            "year": datetime.now().year - 3,
            "info": {
                "plot": "Moderate high jinks result from querying another
 mysterious database.",
                "rating": Decimal("6.5"),
            },
        },
        {
            "title": f"House PartiQL 3",
            "year": datetime.now().year - 1,
```

```
            "info": {
                "plot": "Tepid high jinks result from querying yet another
mysterious database.",
                "rating": Decimal("2.5"),
            },
        },
    ]

    print(f"Inserting a batch of movies into table '{table_name}.")
    statements = [
        f'INSERT INTO "{table_name}" ' f"VALUE {{'title': ?, 'year': ?,
'info': ?}}"
    ] * len(movie_data)
    params = [list(movie.values()) for movie in movie_data]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Getting data for a batch of movies.")
    statements = [f'SELECT * FROM "{table_name}" WHERE title=? AND year=?'] *
len(
        movie_data
    )
    params = [[movie["title"], movie["year"]] for movie in movie_data]
    output = wrapper.run_partiql(statements, params)
    for item in output["Responses"]:
        print(f"\n{item['Item']['title']}, {item['Item']['year']}")
        pprint(item["Item"])
    print("-" * 88)

    ratings = [Decimal("7.7"), Decimal("5.5"), Decimal("1.3")]
    print(f"Updating a batch of movies with new ratings.")
    statements = [
        f'UPDATE "{table_name}" SET info.rating=? ' f"WHERE title=? AND year=?"
    ] * len(movie_data)
    params = [
        [rating, movie["title"], movie["year"]]
        for rating, movie in zip(ratings, movie_data)
    ]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Getting projected data from the table to verify our update.")
```

```
    output = wrapper.dyn_resource.meta.client.execute_statement(
        Statement=f'SELECT title, info.rating FROM "{table_name}"'
    )
    pprint(output["Items"])
    print("-" * 88)

    print(f"Deleting a batch of movies from the table.")
    statements = [f'DELETE FROM "{table_name}" WHERE title=? AND year=?'] * len(
        movie_data
    )
    params = [[movie["title"], movie["year"]] for movie in movie_data]
    wrapper.run_partiql(statements, params)
    print("Success!")
    print("-" * 88)

    print(f"Deleting table '{table_name}'...")
    scaffold.delete_table()
    print("-" * 88)

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLBatchWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「BatchExecuteStatement」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成し、PartiQL クエリのバッチを実行するシナリオを実行します。

```
 table_name = "doc-example-table-movies-partiql-#{rand(10**4)}"
 scaffold = Scaffold.new(table_name)
 sdk = DynamoDBPartiQLBatch.new(table_name)

 new_step(1, 'Create a new DynamoDB table if none already exists.')
 unless scaffold.exists?(table_name)
   puts("\nNo such table: #{table_name}. Creating it...")
   scaffold.create_table(table_name)
   print "Done!\n".green
 end

 new_step(2, 'Populate DynamoDB table with movie data.')
 download_file = 'moviedata.json'
 puts("Downloading movie database to #{download_file}...")
 movie_data = scaffold.fetch_movie_data(download_file)
 puts("Writing movie data from #{download_file} into your table...")
 scaffold.write_batch(movie_data)
 puts("Records added: #{movie_data.length}.")
 print "Done!\n".green

 new_step(3, 'Select a batch of items from the movies table.')
 puts "Let's select some popular movies for side-by-side comparison."
 response = sdk.batch_execute_select([['Mean Girls', 2004], ['Goodfellas',
1977], ['The Prancing of the Lambs', 2005]])
 puts("Items selected: #{response['responses'].length}\n")
 print "\nDone!\n".green

 new_step(4, 'Delete a batch of items from the movies table.')
 sdk.batch_execute_write([['Mean Girls', 2004], ['Goodfellas', 1977], ['The
Prancing of the Lambs', 2005]])
 print "\nDone!\n".green
```

```
  new_step(5, 'Delete the table.')
  return unless scaffold.exists?(table_name)

  scaffold.delete_table
end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「BatchExecuteStatement」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

PartiQL と AWS SDK を使用して DynamoDB テーブルに対してクエリを実行する

次のコード例は、以下を実行する方法を示しています。

- SELECT ステートメントを実行して項目を取得する。
- INSERT 文を実行して項目を追加する。
- UPDATE ステートメントを使用して項目を更新する。
- DELETE ステートメントを実行して項目を削除する。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
namespace PartiQL_Basics_Scenario
{
    public class PartiQLMethods
```

```
    {
        private static readonly AmazonDynamoDBClient Client = new
AmazonDynamoDBClient();


        /// <summary>
        /// Inserts movies imported from a JSON file into the movie table by
        /// using an Amazon DynamoDB PartiQL INSERT statement.
        /// </summary>
        /// <param name="tableName">The name of the table where the movie
        /// information will be inserted.</param>
        /// <param name="movieFileName">The name of the JSON file that contains
        /// movie information.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the insert operation.</returns>
        public static async Task<bool> InsertMovies(string tableName, string
movieFileName)
        {
            // Get the list of movies from the JSON file.
            var movies = ImportMovies(movieFileName);

            var success = false;

            if (movies is not null)
            {
                // Insert the movies in a batch using PartiQL. Because the
                // batch can contain a maximum of 25 items, insert 25 movies
                // at a time.
                string insertBatch = $"INSERT INTO {tableName} VALUE
{{'title': ?, 'year': ?}}";
                var statements = new List<BatchStatementRequest>();

                try
                {
                    for (var indexOffset = 0; indexOffset < 250; indexOffset +=
25)
                    {
                        for (var i = indexOffset; i < indexOffset + 25; i++)
                        {
                            statements.Add(new BatchStatementRequest
                            {
                                Statement = insertBatch,
                                Parameters = new List<AttributeValue>
                                {
```

```
                                    new AttributeValue { S = movies[i].Title },
                                    new AttributeValue { N =
movies[i].Year.ToString() },
                                },
                            });
                        }

                        var response = await
Client.BatchExecuteStatementAsync(new BatchExecuteStatementRequest
                        {
                            Statements = statements,
                        });

                        // Wait between batches for movies to be successfully
added.
                        System.Threading.Thread.Sleep(3000);

                        success = response.HttpStatusCode ==
System.Net.HttpStatusCode.OK;

                        // Clear the list of statements for the next batch.
                        statements.Clear();
                    }
                }
                catch (AmazonDynamoDBException ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }

            return success;
        }

        /// <summary>
        /// Loads the contents of a JSON file into a list of movies to be
        /// added to the DynamoDB table.
        /// </summary>
        /// <param name="movieFileName">The full path to the JSON file.</param>
        /// <returns>A generic list of movie objects.</returns>
        public static List<Movie> ImportMovies(string movieFileName)
        {
            if (!File.Exists(movieFileName))
            {
                return null!;
```

```
            }

            using var sr = new StreamReader(movieFileName);
            string json = sr.ReadToEnd();
            var allMovies = JsonConvert.DeserializeObject<List<Movie>>(json);

            if (allMovies is not null)
            {
                // Return the first 250 entries.
                return allMovies.GetRange(0, 250);
            }
            else
            {
                return null!;
            }
        }



        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetSingleMovie(string tableName, string movieTitle)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
```

```
        }



        /// <summary>
        /// Retrieve multiple movies by year using a SELECT statement.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="year">The year the movies were released.</param>
        /// <returns></returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
GetMovies(string tableName, int year)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE year = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { N = year.ToString() },
            };

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }



        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";
```

```
            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Updates a single movie in the table, adding information for the
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });
```

```
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Displays the list of movies returned from a database query.
        /// </summary>
        /// <param name="items">The list of movie information to display.</param>
        private static void DisplayMovies(List<Dictionary<string,
AttributeValue>> items)
        {
            if (items.Count > 0)
            {
                Console.WriteLine($"Found {items.Count} movies.");
```

```
                items.ForEach(item =>
 Console.WriteLine($"{item["year"].N}\t{item["title"].S}"));
            }
            else
            {
                Console.WriteLine($"Didn't find a movie that matched the supplied
 criteria.");
            }
        }


    }
}



        /// <summary>
        /// Uses a PartiQL SELECT statement to retrieve a single movie from the
        /// movie database.
        /// </summary>
        /// <param name="tableName">The name of the movie table.</param>
        /// <param name="movieTitle">The title of the movie to retrieve.</param>
        /// <returns>A list of movie data. If no movie matches the supplied
        /// title, the list is empty.</returns>
        public static async Task<List<Dictionary<string, AttributeValue>>>
 GetSingleMovie(string tableName, string movieTitle)
        {
            string selectSingle = $"SELECT * FROM {tableName} WHERE title = ?";
            var parameters = new List<AttributeValue>
            {
                new AttributeValue { S = movieTitle },
            };

            var response = await Client.ExecuteStatementAsync(new
 ExecuteStatementRequest
            {
                Statement = selectSingle,
                Parameters = parameters,
            });

            return response.Items;
        }
```

```
        /// <summary>
        /// Inserts a single movie into the movies table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to insert.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success or failure of
        /// the INSERT operation.</returns>
        public static async Task<bool> InsertSingleMovie(string tableName, string
movieTitle, int year)
        {
            string insertBatch = $"INSERT INTO {tableName} VALUE {{'title': ?,
'year': ?}}";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertBatch,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Updates a single movie in the table, adding information for the
        /// producer.
        /// </summary>
        /// <param name="tableName">the name of the table.</param>
        /// <param name="producer">The name of the producer.</param>
        /// <param name="movieTitle">The movie title.</param>
        /// <param name="year">The year the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// UPDATE operation.</returns>
        public static async Task<bool> UpdateSingleMovie(string tableName, string
producer, string movieTitle, int year)
        {
```

```
            string insertSingle = $"UPDATE {tableName} SET Producer=? WHERE title
= ? AND year = ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = insertSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = producer },
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }



        /// <summary>
        /// Deletes a single movie from the table.
        /// </summary>
        /// <param name="tableName">The name of the table.</param>
        /// <param name="movieTitle">The title of the movie to delete.</param>
        /// <param name="year">The year that the movie was released.</param>
        /// <returns>A Boolean value that indicates the success of the
        /// DELETE operation.</returns>
        public static async Task<bool> DeleteSingleMovie(string tableName, string
movieTitle, int year)
        {
            var deleteSingle = $"DELETE FROM {tableName} WHERE title = ? AND year
= ?";

            var response = await Client.ExecuteStatementAsync(new
ExecuteStatementRequest
            {
                Statement = deleteSingle,
                Parameters = new List<AttributeValue>
                {
                    new AttributeValue { S = movieTitle },
                    new AttributeValue { N = year.ToString() },
                },
            });
```

```
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }
```

- API の詳細については、AWS SDK for .NET API リファレンスの「ExecuteStatement」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        //  1. Create a table. (CreateTable)
        if (AwsDoc::DynamoDB::createMoviesDynamoDBTable(clientConfig)) {

            AwsDoc::DynamoDB::partiqlExecuteScenario(clientConfig);

            // 7. Delete the table. (DeleteTable)
            AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(clientConfig);
        }

//! Scenario to modify and query a DynamoDB table using single PartiQL
 statements.
/*!
  \sa partiqlExecuteScenario()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
 */
bool
AwsDoc::DynamoDB::partiqlExecuteScenario(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    // 2. Add a new movie using an "Insert" statement. (ExecuteStatement)
```

```
   Aws::String title;
   float rating;
   int year;
   Aws::String plot;
   {
       title = askQuestion(
               "Enter the title of a movie you want to add to the table: ");
       year = askQuestionForInt("What year was it released? ");
       rating = askQuestionForFloatRange("On a scale of 1 - 10, how do you rate
it? ",
                                         1, 10);
       plot = askQuestion("Summarize the plot for me: ");

       Aws::DynamoDB::Model::ExecuteStatementRequest request;
       std::stringstream sqlStream;
       sqlStream << "INSERT INTO \"" << MOVIE_TABLE_NAME << "\" VALUE {'"
                 << TITLE_KEY << "': ?, '" << YEAR_KEY << "': ?, '"
                 << INFO_KEY << "': ?}";

       request.SetStatement(sqlStream.str());

       // Create the parameter attributes.
       Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
       attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

       Aws::DynamoDB::Model::AttributeValue infoMapAttribute;

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> ratingAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       ratingAttribute->SetN(rating);
       infoMapAttribute.AddMEntry(RATING_KEY, ratingAttribute);

       std::shared_ptr<Aws::DynamoDB::Model::AttributeValue> plotAttribute =
Aws::MakeShared<Aws::DynamoDB::Model::AttributeValue>(
               ALLOCATION_TAG.c_str());
       plotAttribute->SetS(plot);
       infoMapAttribute.AddMEntry(PLOT_KEY, plotAttribute);
       attributes.push_back(infoMapAttribute);
       request.SetParameters(attributes);

       Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
```

```
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to add a movie: " <<
outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    std::cout << "\nAdded '" << title << "' to '" << MOVIE_TABLE_NAME << "'."
              << std::endl;

    //  3. Get the data for the movie using a "Select" statement.
(ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to retrieve movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
        else {
            // Print the retrieved movie information.
            const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();
```

```
            if (items.size() == 1) {
                printMovieInfo(items[0]);
            }
            else {
                std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                          << " There should be only one movie." << std::endl;
            }
        }
    }

    //  4. Update the data for the movie using an "Update" statement.
(ExecuteStatement)
    {
        rating = askQuestionForFloatRange(
                Aws::String("\nLet's update your movie.\nYou rated it  ") +
                std::to_string(rating)
                + ", what new rating would you give it? ", 1, 10);

        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "UPDATE \"" << MOVIE_TABLE_NAME << "\" SET "
                  << INFO_KEY << "." << RATING_KEY << "=? WHERE "
                  << TITLE_KEY << "=? AND " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;

attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(rating));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));

        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);

        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to update a movie: "
                      << outcome.GetError().GetMessage();
            return false;
```

```
        }
    }

    std::cout << "\nUpdated '" << title << "' with new attributes:" << std::endl;

    //  5. Get the updated data for the movie using a "Select" statement.
(ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "SELECT * FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to retrieve the movie information: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
        else {
            const Aws::DynamoDB::Model::ExecuteStatementResult &result =
outcome.GetResult();

            const Aws::Vector<Aws::Map<Aws::String,
Aws::DynamoDB::Model::AttributeValue>> &items = result.GetItems();

            if (items.size() == 1) {
                printMovieInfo(items[0]);
            }
            else {
                std::cerr << "Error: " << items.size() << " movies were
retrieved. "
                          << " There should be only one movie." << std::endl;
            }
        }
```

```
    }

    std::cout << "Deleting the movie" << std::endl;

    // 6. Delete the movie using a "Delete" statement. (ExecuteStatement)
    {
        Aws::DynamoDB::Model::ExecuteStatementRequest request;
        std::stringstream sqlStream;
        sqlStream << "DELETE FROM  \"" << MOVIE_TABLE_NAME << "\" WHERE "
                  << TITLE_KEY << "=? and " << YEAR_KEY << "=?";

        request.SetStatement(sqlStream.str());

        Aws::Vector<Aws::DynamoDB::Model::AttributeValue> attributes;
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetS(title));
        attributes.push_back(Aws::DynamoDB::Model::AttributeValue().SetN(year));
        request.SetParameters(attributes);

        Aws::DynamoDB::Model::ExecuteStatementOutcome outcome =
 dynamoClient.ExecuteStatement(
                request);
        if (!outcome.IsSuccess()) {
            std::cerr << "Failed to delete the movie: "
                      << outcome.GetError().GetMessage() << std::endl;
            return false;
        }
    }

    std::cout << "Movie successfully deleted." << std::endl;
    return true;
}

//! Create a DynamoDB table to be used in sample code scenarios.
/*!
  \sa createMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::createMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    bool movieTableAlreadyExisted = false;
```

```
    {
        Aws::DynamoDB::Model::CreateTableRequest request;

        Aws::DynamoDB::Model::AttributeDefinition yearAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(YEAR_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::N);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::AttributeDefinition titleAttributeDefinition;
        yearAttributeDefinition.SetAttributeName(TITLE_KEY);
        yearAttributeDefinition.SetAttributeType(
                Aws::DynamoDB::Model::ScalarAttributeType::S);
        request.AddAttributeDefinitions(yearAttributeDefinition);

        Aws::DynamoDB::Model::KeySchemaElement yearKeySchema;
        yearKeySchema.WithAttributeName(YEAR_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::HASH);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::KeySchemaElement titleKeySchema;
        yearKeySchema.WithAttributeName(TITLE_KEY).WithKeyType(
                Aws::DynamoDB::Model::KeyType::RANGE);
        request.AddKeySchema(yearKeySchema);

        Aws::DynamoDB::Model::ProvisionedThroughput throughput;
        throughput.WithReadCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS).WithWriteCapacityUnits(
                PROVISIONED_THROUGHPUT_UNITS);
        request.SetProvisionedThroughput(throughput);
        request.SetTableName(MOVIE_TABLE_NAME);

        std::cout << "Creating table '" << MOVIE_TABLE_NAME << "'..." <<
std::endl;
        const Aws::DynamoDB::Model::CreateTableOutcome &result =
dynamoClient.CreateTable(
                request);
        if (!result.IsSuccess()) {
            if (result.GetError().GetErrorType() ==
                Aws::DynamoDB::DynamoDBErrors::RESOURCE_IN_USE) {
                std::cout << "Table already exists." << std::endl;
                movieTableAlreadyExisted = true;
            }
            else {
```

```
                std::cerr << "Failed to create table: "
                          << result.GetError().GetMessage();
                return false;
            }
        }
    }

    // Wait for table to become active.
    if (!movieTableAlreadyExisted) {
        std::cout << "Waiting for table '" << MOVIE_TABLE_NAME
                  << "' to become active...." << std::endl;
        if (!AwsDoc::DynamoDB::waitTableActive(MOVIE_TABLE_NAME,
 clientConfiguration)) {
            return false;
        }
        std::cout << "Table '" << MOVIE_TABLE_NAME << "' created and active."
                  << std::endl;
    }

    return true;
}

//! Delete the DynamoDB table used for sample code scenarios.
/*!
  \sa deleteMoviesDynamoDBTable()
  \param clientConfiguration: AWS client configuration.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::deleteMoviesDynamoDBTable(
        const Aws::Client::ClientConfiguration &clientConfiguration) {
    Aws::DynamoDB::DynamoDBClient dynamoClient(clientConfiguration);

    Aws::DynamoDB::Model::DeleteTableRequest request;
    request.SetTableName(MOVIE_TABLE_NAME);

    const Aws::DynamoDB::Model::DeleteTableOutcome &result =
 dynamoClient.DeleteTable(
            request);
    if (result.IsSuccess()) {
        std::cout << "Your table \""
                  << result.GetResult().GetTableDescription().GetTableName()
                  << " was deleted.\n";
    }
    else {
```

```
        std::cerr << "Failed to delete table: " << result.GetError().GetMessage()
                  << std::endl;
    }

    return result.IsSuccess();
}

//! Query a newly created DynamoDB table until it is active.
/*!
  \sa waitTableActive()
  \param waitTableActive: The DynamoDB table's name.
  \param dynamoClient: A DynamoDB client.
  \return bool: Function succeeded.
*/
bool AwsDoc::DynamoDB::waitTableActive(const Aws::String &tableName,
                                       const Aws::DynamoDB::DynamoDBClient
 &dynamoClient) {

    // Repeatedly call DescribeTable until table is ACTIVE.
    const int MAX_QUERIES = 20;
    Aws::DynamoDB::Model::DescribeTableRequest request;
    request.SetTableName(tableName);

    int count = 0;
    while (count < MAX_QUERIES) {
        const Aws::DynamoDB::Model::DescribeTableOutcome &result =
 dynamoClient.DescribeTable(
                request);
        if (result.IsSuccess()) {
            Aws::DynamoDB::Model::TableStatus status =
 result.GetResult().GetTable().GetTableStatus();

            if (Aws::DynamoDB::Model::TableStatus::ACTIVE != status) {
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
            else {
                return true;
            }
        }
        else {
            std::cerr << "Error DynamoDB::waitTableActive "
                      << result.GetError().GetMessage() << std::endl;
            return false;
        }
```

```
        count++;
    }
    return false;
}
```

- API の詳細については、AWS SDK for C++ API リファレンスの「ExecuteStatement」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成し、PartiQL クエリを実行するシナリオを実行します。

```
import (
 "context"
 "fmt"
 "log"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/dynamodb/actions"
)

// RunPartiQLSingleScenario shows you how to use the AWS SDK for Go
// to use PartiQL to query a table that stores data about movies.
//
// * Use PartiQL statements to add, get, update, and delete data for individual
 movies.
//
// This example creates an Amazon DynamoDB service client from the specified
 sdkConfig so that
```

```
// you can replace it with a mocked or stubbed config for unit testing.
//
// This example creates and deletes a DynamoDB table to use during the scenario.
func RunPartiQLSingleScenario(ctx context.Context, sdkConfig aws.Config,
 tableName string) {
 defer func() {
  if r := recover(); r != nil {
   fmt.Printf("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon DynamoDB PartiQL single action demo.")
 log.Println(strings.Repeat("-", 88))

 tableBasics := actions.TableBasics{
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
  TableName:      tableName,
 }
 runner := actions.PartiQLRunner{
  DynamoDbClient: dynamodb.NewFromConfig(sdkConfig),
  TableName:      tableName,
 }

 exists, err := tableBasics.TableExists(ctx)
 if err != nil {
  panic(err)
 }
 if !exists {
  log.Printf("Creating table %v...\n", tableName)
  _, err = tableBasics.CreateMovieTable(ctx)
  if err != nil {
   panic(err)
  } else {
   log.Printf("Created table %v.\n", tableName)
  }
 } else {
  log.Printf("Table %v already exists.\n", tableName)
 }
 log.Println(strings.Repeat("-", 88))

 currentYear, _, _ := time.Now().Date()
 customMovie := actions.Movie{
  Title: "24 Hour PartiQL People",
```

```
 Year:  currentYear,
 Info: map[string]interface{}{
  "plot":   "A group of data developers discover a new query language they can't
stop using.",
  "rating": 9.9,
 },
}

log.Printf("Inserting movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
err = runner.AddMovie(ctx, customMovie)
if err == nil {
 log.Printf("Added %v to the movie table.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting data for movie '%v' released in %v.", customMovie.Title,
customMovie.Year)
movie, err := runner.GetMovie(ctx, customMovie.Title, customMovie.Year)
if err == nil {
 log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

newRating := 6.6
log.Printf("Updating movie '%v' with a rating of %v.", customMovie.Title,
newRating)
err = runner.UpdateMovie(ctx, customMovie, newRating)
if err == nil {
 log.Printf("Updated %v with a new rating.\n", customMovie.Title)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Getting data again to verify the update.")
movie, err = runner.GetMovie(ctx, customMovie.Title, customMovie.Year)
if err == nil {
 log.Println(movie)
}
log.Println(strings.Repeat("-", 88))

log.Printf("Deleting movie '%v'.\n", customMovie.Title)
err = runner.DeleteMovie(ctx, customMovie)
if err == nil {
 log.Printf("Deleted %v.\n", customMovie.Title)
```

```
 }

 err = tableBasics.DeleteTable(ctx)
 if err == nil {
  log.Printf("Deleted table %v.\n", tableBasics.TableName)
 }

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

この例で使用している Movie struct を定義します。

```
import (
 "archive/zip"
 "bytes"
 "encoding/json"
 "fmt"
 "io"
 "log"
 "net/http"

 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// Movie encapsulates data about a movie. Title and Year are the composite
 primary key
// of the movie in Amazon DynamoDB. Title is the sort key, Year is the partition
 key,
// and Info is additional data.
type Movie struct {
 Title string                 `dynamodbav:"title"`
 Year  int                    `dynamodbav:"year"`
 Info  map[string]interface{} `dynamodbav:"info"`
}

// GetKey returns the composite primary key of the movie in a format that can be
// sent to DynamoDB.
```

```
func (movie Movie) GetKey() map[string]types.AttributeValue {
 title, err := attributevalue.Marshal(movie.Title)
 if err != nil {
  panic(err)
 }
 year, err := attributevalue.Marshal(movie.Year)
 if err != nil {
  panic(err)
 }
 return map[string]types.AttributeValue{"title": title, "year": year}
}

// String returns the title, year, rating, and plot of a movie, formatted for the
 example.
func (movie Movie) String() string {
 return fmt.Sprintf("%v\n\tReleased: %v\n\tRating: %v\n\tPlot: %v\n",
  movie.Title, movie.Year, movie.Info["rating"], movie.Info["plot"])
}
```

PartiQL ステートメントを実行する struct およびメソッドを作成します。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// PartiQLRunner encapsulates the Amazon DynamoDB service actions used in the
// PartiQL examples. It contains a DynamoDB service client that is used to act on
 the
// specified table.
type PartiQLRunner struct {
 DynamoDbClient *dynamodb.Client
 TableName      string
}
```

```
// AddMovie runs a PartiQL INSERT statement to add a movie to the DynamoDB table.
func (runner PartiQLRunner) AddMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title, movie.Year,
 movie.Info})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("INSERT INTO \"%v\" VALUE {'title': ?, 'year': ?, 'info': ?}",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't insert an item with PartiQL. Here's why: %v\n", err)
 }
 return err
}



// GetMovie runs a PartiQL SELECT statement to get a movie from the DynamoDB
 table by
// title and year.
func (runner PartiQLRunner) GetMovie(ctx context.Context, title string, year int)
 (Movie, error) {
 var movie Movie
 params, err := attributevalue.MarshalList([]interface{}{title, year})
 if err != nil {
  panic(err)
 }
 response, err := runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("SELECT * FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't get info about %v. Here's why: %v\n", title, err)
```

```
 } else {
  err = attributevalue.UnmarshalMap(response.Items[0], &movie)
  if err != nil {
   log.Printf("Couldn't unmarshal response. Here's why: %v\n", err)
  }
 }
 return movie, err
}



// UpdateMovie runs a PartiQL UPDATE statement to update the rating of a movie
 that
// already exists in the DynamoDB table.
func (runner PartiQLRunner) UpdateMovie(ctx context.Context, movie Movie, rating
 float64) error {
 params, err := attributevalue.MarshalList([]interface{}{rating, movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("UPDATE \"%v\" SET info.rating=? WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't update movie %v. Here's why: %v\n", movie.Title, err)
 }
 return err
}



// DeleteMovie runs a PartiQL DELETE statement to remove a movie from the
 DynamoDB table.
func (runner PartiQLRunner) DeleteMovie(ctx context.Context, movie Movie) error {
 params, err := attributevalue.MarshalList([]interface{}{movie.Title,
 movie.Year})
 if err != nil {
  panic(err)
 }
```

```
 _, err = runner.DynamoDbClient.ExecuteStatement(ctx,
 &dynamodb.ExecuteStatementInput{
  Statement: aws.String(
   fmt.Sprintf("DELETE FROM \"%v\" WHERE title=? AND year=?",
    runner.TableName)),
  Parameters: params,
 })
 if err != nil {
  log.Printf("Couldn't delete %v from the table. Here's why: %v\n", movie.Title,
 err)
 }
 return err
}
```

- API の詳細については、AWS SDK for Go API リファレンスの「ExecuteStatement」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
public class ScenarioPartiQ {
    public static void main(String[] args) throws IOException {
        String fileName = "../../../resources/sample_files/movies.json";
        String tableName = "MoviesPartiQ";
        Region region = Region.US_EAST_1;
        DynamoDbClient ddb = DynamoDbClient.builder()
            .region(region)
            .build();

        System.out.println(
            "******* Creating an Amazon DynamoDB table named MoviesPartiQ with a
 key named year and a sort key named title.");
```

```
        createTable(ddb, tableName);

        System.out.println("Loading data into the MoviesPartiQ table.");
        loadData(ddb, fileName);

        System.out.println("Getting data from the MoviesPartiQ table.");
        getItem(ddb);

        System.out.println("Putting a record into the MoviesPartiQ table.");
        putRecord(ddb);

        System.out.println("Updating a record.");
        updateTableItem(ddb);

        System.out.println("Querying the movies released in 2013.");
        queryTable(ddb);

        System.out.println("Deleting the Amazon DynamoDB table.");
        deleteDynamoDBTable(ddb, tableName);
        ddb.close();
    }

    public static void createTable(DynamoDbClient ddb, String tableName) {
        DynamoDbWaiter dbWaiter = ddb.waiter();
        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>();

        // Define attributes.
        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("year")
            .attributeType("N")
            .build());

        attributeDefinitions.add(AttributeDefinition.builder()
            .attributeName("title")
            .attributeType("S")
            .build());

        ArrayList<KeySchemaElement> tableKey = new ArrayList<>();
        KeySchemaElement key = KeySchemaElement.builder()
            .attributeName("year")
            .keyType(KeyType.HASH)
            .build();

        KeySchemaElement key2 = KeySchemaElement.builder()
```

```
            .attributeName("title")
            .keyType(KeyType.RANGE) // Sort
            .build();

        // Add KeySchemaElement objects to the list.
        tableKey.add(key);
        tableKey.add(key2);

        CreateTableRequest request = CreateTableRequest.builder()
            .keySchema(tableKey)
            .billingMode(BillingMode.PAY_PER_REQUEST) //Scales based on traffic.
            .attributeDefinitions(attributeDefinitions)
            .tableName(tableName)
            .build();

        try {
            CreateTableResponse response = ddb.createTable(request);
            DescribeTableRequest tableRequest = DescribeTableRequest.builder()
                .tableName(tableName)
                .build();

            // Wait until the Amazon DynamoDB table is created.
            WaiterResponse<DescribeTableResponse> waiterResponse =
dbWaiter.waitUntilTableExists(tableRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            String newTable = response.tableDescription().tableName();
            System.out.println("The " + newTable + " was successfully created.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    // Load data into the table.
    public static void loadData(DynamoDbClient ddb, String fileName) throws
IOException {

        String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?,
'title' : ?, 'info' : ?}";
        JsonParser parser = new JsonFactory().createParser(new File(fileName));
        com.fasterxml.jackson.databind.JsonNode rootNode = new
ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();
```

```
        ObjectNode currentNode;
        int t = 0;
        List<AttributeValue> parameters = new ArrayList<>();
        while (iter.hasNext()) {

            // Add 200 movies to the table.
            if (t == 200)
                break;
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();
            String info = currentNode.path("info").toString();

            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf(year))
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s(title)
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s(info)
                .build();

            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);

            // Insert the movie into the Amazon DynamoDB table.
            executeStatementRequest(ddb, sqlStatement, parameters);
            System.out.println("Added Movie " + title);

            parameters.remove(att1);
            parameters.remove(att2);
            parameters.remove(att3);
            t++;
        }
    }

    public static void getItem(DynamoDbClient ddb) {
```

```
        String sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and
title=?";
        List<AttributeValue> parameters = new ArrayList<>();
        AttributeValue att1 = AttributeValue.builder()
            .n("2012")
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("The Perks of Being a Wallflower")
            .build();

        parameters.add(att1);
        parameters.add(att2);

        try {
            ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
            System.out.println("ExecuteStatement successful: " +
response.toString());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void putRecord(DynamoDbClient ddb) {

        String sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?,
'title' : ?, 'info' : ?}";
        try {
            List<AttributeValue> parameters = new ArrayList<>();

            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf("2020"))
                .build();

            AttributeValue att2 = AttributeValue.builder()
                .s("My Movie")
                .build();

            AttributeValue att3 = AttributeValue.builder()
                .s("No Information")
                .build();
```

```
            parameters.add(att1);
            parameters.add(att2);
            parameters.add(att3);

            executeStatementRequest(ddb, sqlStatement, parameters);
            System.out.println("Added new movie.");

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateTableItem(DynamoDbClient ddb) {

        String sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\":
[\"Merian C. Cooper\",\"Ernest B. Schoedsack' where year=? and title=?";
        List<AttributeValue> parameters = new ArrayList<>();
        AttributeValue att1 = AttributeValue.builder()
            .n(String.valueOf("2013"))
            .build();

        AttributeValue att2 = AttributeValue.builder()
            .s("The East")
            .build();

        parameters.add(att1);
        parameters.add(att2);

        try {
            executeStatementRequest(ddb, sqlStatement, parameters);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("Item was updated!");
    }

    // Query the table where the year is 2013.
    public static void queryTable(DynamoDbClient ddb) {
        String sqlStatement = "SELECT * FROM MoviesPartiQ where year = ? ORDER BY
 year";
```

```
        try {

            List<AttributeValue> parameters = new ArrayList<>();
            AttributeValue att1 = AttributeValue.builder()
                .n(String.valueOf("2013"))
                .build();
            parameters.add(att1);

            // Get items in the table and write out the ID value.
            ExecuteStatementResponse response = executeStatementRequest(ddb,
sqlStatement, parameters);
            System.out.println("ExecuteStatement successful: " +
response.toString());

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteDynamoDBTable(DynamoDbClient ddb, String tableName)
{

        DeleteTableRequest request = DeleteTableRequest.builder()
            .tableName(tableName)
            .build();

        try {
            ddb.deleteTable(request);

        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        System.out.println(tableName + " was successfully deleted!");
    }

    private static ExecuteStatementResponse
executeStatementRequest(DynamoDbClient ddb, String statement,

List<AttributeValue> parameters) {
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
            .statement(statement)
            .parameters(parameters)
```

```
            .build();

        return ddb.executeStatement(request);
    }

    private static void processResults(ExecuteStatementResponse
 executeStatementResult) {
        System.out.println("ExecuteStatement successful: " +
 executeStatementResult.toString());
    }
}
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「ExecuteStatement」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

単一の PartiQL ステートメントを実行します。

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "SingleOriginCoffees";

export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
      "deleteTable",
      `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
      { type: "confirm", confirmAll },
    );
    const deleteTable = await input.handle({});
    if (deleteTable) {
      await client.send(new DeleteTableCommand({ tableName }));
    } else {
      console.warn(
        "Scenario could not run. Either delete ${tableName} or provide a unique
 table name.",
      );
      return;
    }
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ResourceNotFoundException"
    ) {
      // Do nothing. This means the table is not there.
    } else {
      throw caught;
    }
  }

  /**
   * Create a table.
   */
```

```
  log("Creating a table.");
  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "varietal",
        // 'S' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
        AttributeType: "S",
      },
    ],
    // The KeySchema defines the primary key. The primary key can be
    // a partition key, or a combination of a partition key and a sort key.
    // Key schema design is important. For more info, see
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "varietal", KeyType: "HASH" }],
  });
  await client.send(createTableCommand);
  log(`Table created: ${tableName}.`);

  /**
   * Wait until the table is active.
   */

  // This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
  // You can't write to a table before it's active.
  log("Waiting for the table to be active.");
  await waitUntilTableExists({ client }, { TableName: tableName });
  log("Table active.");

  /**
   * Insert an item.
   */

  log("Inserting a coffee into the table.");
  const addItemStatementCommand = new ExecuteStatementCommand({
```

```
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
    Statement: `INSERT INTO ${tableName} value {'varietal':?, 'profile':?}`,
    Parameters: ["arabica", ["chocolate", "floral"]],
  });
  await client.send(addItemStatementCommand);
  log("Coffee inserted.");

  /**
   * Select an item.
   */

  log("Selecting the coffee from the table.");
  const selectItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
    Statement: `SELECT * FROM ${tableName} WHERE varietal=?`,
    Parameters: ["arabica"],
  });
  const selectItemResponse = await docClient.send(selectItemStatementCommand);
  log(`Got coffee: ${JSON.stringify(selectItemResponse.Items[0])}`);

  /**
   * Update the item.
   */

  log("Add a flavor profile to the coffee.");
  const updateItemStatementCommand = new ExecuteStatementCommand({
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
    Statement: `UPDATE ${tableName} SET profile=list_append(profile, ?) WHERE
 varietal=?`,
    Parameters: [["fruity"], "arabica"],
  });
  await client.send(updateItemStatementCommand);
  log("Updated coffee");

  /**
   * Delete the item.
   */

  log("Deleting the coffee.");
  const deleteItemStatementCommand = new ExecuteStatementCommand({
```

```
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
    Statement: `DELETE FROM ${tableName} WHERE varietal=?`,
    Parameters: ["arabica"],
  });
  await docClient.send(deleteItemStatementCommand);
  log("Coffee deleted.");

  /**
   * Delete the table.
   */

  log("Deleting the table.");
  const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
  await client.send(deleteTableCommand);
  log("Table deleted.");
};
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「ExecuteStatement」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun main() {
    val ddb = DynamoDbClient.fromEnvironment { region = "us-east-1" }
    val tableName = "MoviesPartiQ"
    val fileName = "../../../resources/sample_files/movies.json"
    println("Creating an Amazon DynamoDB table named MoviesPartiQ with a key
 named id and a sort key named title.")
    createTablePartiQL(ddb, tableName, "year")
    loadDataPartiQL(ddb, fileName)
```

```
    println("******* Getting data from the MoviesPartiQ table.")
    getMoviePartiQL(ddb)

    println("******* Putting a record into the MoviesPartiQ table.")
    putRecordPartiQL(ddb)

    println("******* Updating a record.")
    updateTableItemPartiQL(ddb)

    println("******* Querying the movies released in 2013.")
    queryTablePartiQL(ddb)

    println("******* Deleting the MoviesPartiQ table.")
    deleteTablePartiQL(tableName)
}

suspend fun createTablePartiQL(
    ddb: DynamoDbClient,
    tableNameVal: String,
    key: String,
) {
    val attDef =
        AttributeDefinition {
            attributeName = key
            attributeType = ScalarAttributeType.N
        }

    val attDef1 =
        AttributeDefinition {
            attributeName = "title"
            attributeType = ScalarAttributeType.S
        }

    val keySchemaVal =
        KeySchemaElement {
            attributeName = key
            keyType = KeyType.Hash
        }

    val keySchemaVal1 =
        KeySchemaElement {
            attributeName = "title"
            keyType = KeyType.Range
        }
```

```
    val request =
        CreateTableRequest {
            attributeDefinitions = listOf(attDef, attDef1)
            keySchema = listOf(keySchemaVal, keySchemaVal1)
            billingMode = BillingMode.PayPerRequest
            tableName = tableNameVal
        }

    val response = ddb.createTable(request)
    ddb.waitUntilTableExists {
        // suspend call
        tableName = tableNameVal
    }
    println("The table was successfully created
 ${response.tableDescription?.tableArn}")
}

suspend fun loadDataPartiQL(
    ddb: DynamoDbClient,
    fileName: String,
) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?,
 'info' : ?}"
    val parser = JsonFactory().createParser(File(fileName))
    val rootNode = ObjectMapper().readTree<JsonNode>(parser)
    val iter: Iterator<JsonNode> = rootNode.iterator()
    var currentNode: ObjectNode
    var t = 0

    while (iter.hasNext()) {
        if (t == 200) {
            break
        }

        currentNode = iter.next() as ObjectNode
        val year = currentNode.path("year").asInt()
        val title = currentNode.path("title").asText()
        val info = currentNode.path("info").toString()

        val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
        parameters.add(AttributeValue.N(year.toString()))
        parameters.add(AttributeValue.S(title))
        parameters.add(AttributeValue.S(info))
```

```
        executeStatementPartiQL(ddb, sqlStatement, parameters)
        println("Added Movie $title")
        parameters.clear()
        t++
    }
}

suspend fun getMoviePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year=? and title=?"
    val parameters: MutableList<AttributeValue> = ArrayList<AttributeValue>()
    parameters.add(AttributeValue.N("2012"))
    parameters.add(AttributeValue.S("The Perks of Being a Wallflower"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("ExecuteStatement successful: $response")
}

suspend fun putRecordPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "INSERT INTO MoviesPartiQ VALUE {'year':?, 'title' : ?,
 'info' : ?}"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2020"))
    parameters.add(AttributeValue.S("My Movie"))
    parameters.add(AttributeValue.S("No Info"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Added new movie.")
}

suspend fun updateTableItemPartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "UPDATE MoviesPartiQ SET info = 'directors\":[\"Merian C.
 Cooper\",\"Ernest B. Schoedsack\' where year=? and title=?"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    parameters.add(AttributeValue.S("The East"))
    executeStatementPartiQL(ddb, sqlStatement, parameters)
    println("Item was updated!")
}

// Query the table where the year is 2013.
suspend fun queryTablePartiQL(ddb: DynamoDbClient) {
    val sqlStatement = "SELECT * FROM MoviesPartiQ where year = ?"
    val parameters: MutableList<AttributeValue> = java.util.ArrayList()
    parameters.add(AttributeValue.N("2013"))
    val response = executeStatementPartiQL(ddb, sqlStatement, parameters)
```

```
    println("ExecuteStatement successful: $response")
}

suspend fun deleteTablePartiQL(tableNameVal: String) {
    val request =
        DeleteTableRequest {
            tableName = tableNameVal
        }

    DynamoDbClient { region = "us-east-1" }.use { ddb ->
        ddb.deleteTable(request)
        println("$tableNameVal was deleted")
    }
}

suspend fun executeStatementPartiQL(
    ddb: DynamoDbClient,
    statementVal: String,
    parametersVal: List<AttributeValue>,
): ExecuteStatementResponse {
    val request =
        ExecuteStatementRequest {
            statement = statementVal
            parameters = parametersVal
        }

    return ddb.executeStatement(request)
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「ExecuteStatement」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
namespace DynamoDb\PartiQL_Basics;

use Aws\DynamoDb\Marshaler;
use DynamoDb;
use DynamoDb\DynamoDBAttribute;

use function AwsUtilities\testable_readline;
use function AwsUtilities\loadMovieData;

class GettingStartedWithPartiQL
{
    public function run()
    {
        echo("\n");
        echo("--------------------------------------\n");
        print("Welcome to the Amazon DynamoDB - PartiQL getting started demo
 using PHP!\n");
        echo("--------------------------------------\n");

        $uuid = uniqid();
        $service = new DynamoDb\DynamoDBService();

        $tableName = "partiql_demo_table_$uuid";
        $service->createTable(
            $tableName,
            [
                new DynamoDBAttribute('year', 'N', 'HASH'),
                new DynamoDBAttribute('title', 'S', 'RANGE')
            ]
        );

        echo "Waiting for table...";
        $service->dynamoDbClient->waitUntil("TableExists", ['TableName' =>
 $tableName]);
        echo "table $tableName found!\n";

        echo "What's the name of the last movie you watched?\n";
        while (empty($movieName)) {
            $movieName = testable_readline("Movie name: ");
        }
        echo "And what year was it released?\n";
        $movieYear = "year";
        while (!is_numeric($movieYear) || intval($movieYear) != $movieYear) {
```

```
            $movieYear = testable_readline("Year released: ");
        }
        $key = [
            'Item' => [
                'year' => [
                    'N' => "$movieYear",
                ],
                'title' => [
                    'S' => $movieName,
                ],
            ],
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("INSERT", $tableName, $key);
        $service->insertItemByPartiQL($statement, $parameters);

        echo "How would you rate the movie from 1-10?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        echo "What was the movie about?\n";
        while (empty($plot)) {
            $plot = testable_readline("Plot summary: ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot),
        ];

        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQL($statement, $parameters);
        echo "Movie added and updated.\n";



        $batch = json_decode(loadMovieData());

        $service->writeBatch($tableName, $batch);

        $movie = $service->getItemByPartiQL($tableName, $key);
```

```
        echo "\nThe movie {$movie['Items'][0]['title']['S']} was released in
 {$movie['Items'][0]['year']['N']}.\n";
        echo "What rating would you like to give {$movie['Items'][0]['title']
['S']}?\n";
        $rating = 0;
        while (!is_numeric($rating) || intval($rating) != $rating || $rating < 1
 || $rating > 10) {
            $rating = testable_readline("Rating (1-10): ");
        }
        $attributes = [
            new DynamoDBAttribute('rating', 'N', 'HASH', $rating),
            new DynamoDBAttribute('plot', 'S', 'RANGE', $plot)
        ];
        list($statement, $parameters) = $service-
>buildStatementAndParameters("UPDATE", $tableName, $key, $attributes);
        $service->updateItemByPartiQL($statement, $parameters);

        $movie = $service->getItemByPartiQL($tableName, $key);
        echo "Okay, you have rated {$movie['Items'][0]['title']['S']} as a
 {$movie['Items'][0]['rating']['N']}\n";

        $service->deleteItemByPartiQL($statement, $parameters);
        echo "But, bad news, this was a trap. That movie has now been deleted
 because of your rating...harsh.\n";

        echo "That's okay though. The book was better. Now, for something
 lighter, in what year were you born?\n";
        $birthYear = "not a number";
        while (!is_numeric($birthYear) || $birthYear >= date("Y")) {
            $birthYear = testable_readline("Birth year: ");
        }
        $birthKey = [
            'Key' => [
                'year' => [
                    'N' => "$birthYear",
                ],
            ],
        ];
        $result = $service->query($tableName, $birthKey);
        $marshal = new Marshaler();
        echo "Here are the movies in our collection released the year you were
 born:\n";
        $oops = "Oops! There were no movies released in that year (that we know
 of).\n";
```

```
        $display = "";
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            $display .= $movie['title'] . "\n";
        }
        echo ($display) ?: $oops;

        $yearsKey = [
            'Key' => [
                'year' => [
                    'N' => [
                        'minRange' => 1990,
                        'maxRange' => 1999,
                    ],
                ],
            ],
        ];
        $filter = "year between 1990 and 1999";
        echo "\nHere's a list of all the movies released in the 90s:\n";
        $result = $service->scan($tableName, $yearsKey, $filter);
        foreach ($result['Items'] as $movie) {
            $movie = $marshal->unmarshalItem($movie);
            echo $movie['title'] . "\n";
        }

        echo "\nCleaning up this demo by deleting table $tableName...\n";
        $service->deleteTable($tableName);
    }
}

    public function insertItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => "$statement",
            'Parameters' => $parameters,
        ]);
    }

    public function getItemByPartiQL(string $tableName, array $key): Result
    {
        list($statement, $parameters) = $this-
>buildStatementAndParameters("SELECT", $tableName, $key['Item']);

        return $this->dynamoDbClient->executeStatement([
```

```
            'Parameters' => $parameters,
            'Statement' => $statement,
        ]);
    }

    public function updateItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }

    public function deleteItemByPartiQL(string $statement, array $parameters)
    {
        $this->dynamoDbClient->executeStatement([
            'Statement' => $statement,
            'Parameters' => $parameters,
        ]);
    }
```

- API の詳細については、AWS SDK for PHP API リファレンスの「ExecuteStatement」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

PartiQL ステートメントを実行できるクラスを作成します。

```
from datetime import datetime
from decimal import Decimal
import logging
from pprint import pprint
```

```
import boto3
from botocore.exceptions import ClientError

from scaffold import Scaffold

logger = logging.getLogger(__name__)

class PartiQLWrapper:
    """
    Encapsulates a DynamoDB resource to run PartiQL statements.
    """

    def __init__(self, dyn_resource):
        """
        :param dyn_resource: A Boto3 DynamoDB resource.
        """
        self.dyn_resource = dyn_resource


    def run_partiql(self, statement, params):
        """
        Runs a PartiQL statement. A Boto3 resource is used even though
        `execute_statement` is called on the underlying `client` object because
 the
        resource transforms input and output from plain old Python objects
 (POPOs) to
        the DynamoDB format. If you create the client directly, you must do these
        transforms yourself.

        :param statement: The PartiQL statement.
        :param params: The list of PartiQL parameters. These are applied to the
                       statement in the order they are listed.
        :return: The items returned from the statement, if any.
        """
        try:
            output = self.dyn_resource.meta.client.execute_statement(
                Statement=statement, Parameters=params
            )
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.error(
                    "Couldn't execute PartiQL '%s' because the table does not
 exist.",
                    statement,
```

```
                )
            else:
                logger.error(
                    "Couldn't execute PartiQL '%s'. Here's why: %s: %s",
                    statement,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
            raise
        else:
            return output
```

テーブルを作成し、PartiQL クエリを実行するシナリオを実行します。

```
def run_scenario(scaffold, wrapper, table_name):
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the Amazon DynamoDB PartiQL single statement demo.")
    print("-" * 88)

    print(f"Creating table '{table_name}' for the demo...")
    scaffold.create_table(table_name)
    print("-" * 88)

    title = "24 Hour PartiQL People"
    year = datetime.now().year
    plot = "A group of data developers discover a new query language they can't
 stop using."
    rating = Decimal("9.9")

    print(f"Inserting movie '{title}' released in {year}.")
    wrapper.run_partiql(
        f"INSERT INTO \"{table_name}\" VALUE {{'title': ?, 'year': ?,
 'info': ?}}",
        [title, year, {"plot": plot, "rating": rating}],
    )
    print("Success!")
    print("-" * 88)
```

```
    print(f"Getting data for movie '{title}' released in {year}.")
    output = wrapper.run_partiql(
        f'SELECT * FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    for item in output["Items"]:
        print(f"\n{item['title']}, {item['year']}")
        pprint(output["Items"])
    print("-" * 88)

    rating = Decimal("2.4")
    print(f"Updating movie '{title}' with a rating of {float(rating)}.")
    wrapper.run_partiql(
        f'UPDATE "{table_name}" SET info.rating=? WHERE title=? AND year=?',
        [rating, title, year],
    )
    print("Success!")
    print("-" * 88)

    print(f"Getting data again to verify our update.")
    output = wrapper.run_partiql(
        f'SELECT * FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    for item in output["Items"]:
        print(f"\n{item['title']}, {item['year']}")
        pprint(output["Items"])
    print("-" * 88)

    print(f"Deleting movie '{title}' released in {year}.")
    wrapper.run_partiql(
        f'DELETE FROM "{table_name}" WHERE title=? AND year=?', [title, year]
    )
    print("Success!")
    print("-" * 88)

    print(f"Deleting table '{table_name}'...")
    scaffold.delete_table()
    print("-" * 88)

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
```

```
        dyn_res = boto3.resource("dynamodb")
        scaffold = Scaffold(dyn_res)
        movies = PartiQLWrapper(dyn_res)
        run_scenario(scaffold, movies, "doc-example-table-partiql-movies")
    except Exception as e:
        print(f"Something went wrong with the demo! Here's what: {e}")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「ExecuteStatement」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

テーブルを作成し、PartiQL クエリを実行するシナリオを実行します。

```
table_name = "doc-example-table-movies-partiql-#{rand(10**8)}"
scaffold = Scaffold.new(table_name)
sdk = DynamoDBPartiQLSingle.new(table_name)

new_step(1, 'Create a new DynamoDB table if none already exists.')
unless scaffold.exists?(table_name)
  puts("\nNo such table: #{table_name}. Creating it...")
  scaffold.create_table(table_name)
  print "Done!\n".green
end

new_step(2, 'Populate DynamoDB table with movie data.')
download_file = 'moviedata.json'
puts("Downloading movie database to #{download_file}...")
movie_data = scaffold.fetch_movie_data(download_file)
puts("Writing movie data from #{download_file} into your table...")
scaffold.write_batch(movie_data)
puts("Records added: #{movie_data.length}.")
print "Done!\n".green
```

```
  new_step(3, 'Select a single item from the movies table.')
  response = sdk.select_item_by_title('Star Wars')
  puts("Items selected for title 'Star Wars': #{response.items.length}\n")
  print response.items.first.to_s.yellow
  print "\n\nDone!\n".green

  new_step(4, 'Update a single item from the movies table.')
  puts "Let's correct the rating on The Big Lebowski to 10.0."
  sdk.update_rating_by_title('The Big Lebowski', 1998, 10.0)
  print "\nDone!\n".green

  new_step(5, 'Delete a single item from the movies table.')
  puts "Let's delete The Silence of the Lambs because it's just too scary."
  sdk.delete_item_by_title('The Silence of the Lambs', 1991)
  print "\nDone!\n".green

  new_step(6, 'Insert a new item into the movies table.')
  puts "Let's create a less-scary movie called The Prancing of the Lambs."
  sdk.insert_item('The Prancing of the Lambs', 2005, 'A movie about happy
 livestock.', 5.0)
  print "\nDone!\n".green

  new_step(7, 'Delete the table.')
  return unless scaffold.exists?(table_name)

  scaffold.delete_table
end
```

- API の詳細については、AWS SDK for Ruby API リファレンスの「ExecuteStatement」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
async fn make_table(
    client: &Client,
    table: &str,
    key: &str,
) -> Result<(), SdkError<CreateTableError>> {
    let ad = AttributeDefinition::builder()
        .attribute_name(key)
        .attribute_type(ScalarAttributeType::S)
        .build()
        .expect("creating AttributeDefinition");

    let ks = KeySchemaElement::builder()
        .attribute_name(key)
        .key_type(KeyType::Hash)
        .build()
        .expect("creating KeySchemaElement");

    match client
        .create_table()
        .table_name(table)
        .key_schema(ks)
        .attribute_definitions(ad)
        .billing_mode(BillingMode::PayPerRequest)
        .send()
        .await
    {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

async fn add_item(client: &Client, item: Item) -> Result<(),
 SdkError<ExecuteStatementError>> {
    match client
        .execute_statement()
        .statement(format!(
            r#"INSERT INTO "{}" VALUE {{
                "{}": ?,
                "acount_type": ?,
                "age": ?,
                "first_name": ?,
                "last_name": ?
        }} "#,
```

```
            item.table, item.key
        ))
        .set_parameters(Some(vec![
            AttributeValue::S(item.utype),
            AttributeValue::S(item.age),
            AttributeValue::S(item.first_name),
            AttributeValue::S(item.last_name),
        ]))
        .send()
        .await
    {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

async fn query_item(client: &Client, item: Item) -> bool {
    match client
        .execute_statement()
        .statement(format!(
            r#"SELECT * FROM "{}" WHERE "{}" = ?"#,
            item.table, item.key
        ))
        .set_parameters(Some(vec![AttributeValue::S(item.value)]))
        .send()
        .await
    {
        Ok(resp) => {
            if !resp.items().is_empty() {
                println!("Found a matching entry in the table:");
                println!("{:?}", resp.items.unwrap_or_default().pop());
                true
            } else {
                println!("Did not find a match.");
                false
            }
        }
        Err(e) => {
            println!("Got an error querying table:");
            println!("{}", e);
            process::exit(1);
        }
    }
}
```

```
async fn remove_item(client: &Client, table: &str, key: &str, value: String) ->
 Result<(), Error> {
    client
        .execute_statement()
        .statement(format!(r#"DELETE FROM "{table}" WHERE "{key}" = ?"#))
        .set_parameters(Some(vec![AttributeValue::S(value)]))
        .send()
        .await?;

    println!("Deleted item.");

    Ok(())
}

async fn remove_table(client: &Client, table: &str) -> Result<(), Error> {
    client.delete_table().table_name(table).send().await?;

    Ok(())
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「ExecuteStatement」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK でグローバルセカンダリインデックスを使用して DynamoDB テーブルをクエリする

以下のコード例は、グローバルセカンダリインデックスを使用してテーブルをクエリする方法を示しています。

- プライマリキーを使用して DynamoDB テーブルをクエリします。
- 代替アクセスパターンのグローバルセカンダリインデックス (GSI) をクエリします。
- テーブルクエリと GSI クエリを比較します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でプライマリキーとグローバルセカンダリインデックス (GSI) を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryTable(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();
```

```
       try {
           final QueryResponse response = dynamoDbClient.query(queryRequest);
           System.out.println("Query on base table successful. Found " +
response.count() + " items");
           return response;
       } catch (ResourceNotFoundException e) {
           System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
           throw new DynamoDbQueryException("Table not found: " + tableName, e);
       } catch (DynamoDbException e) {
           System.err.println("Error querying base table: " + e.getMessage());
           throw new DynamoDbQueryException("Failed to execute query on base
table", e);
       }
   }

   /**
    * Queries a DynamoDB Global Secondary Index (GSI) by partition key.
    *
    * @param tableName         The name of the DynamoDB table
    * @param indexName         The name of the GSI
    * @param partitionKeyName  The name of the GSI partition key attribute
    * @param partitionKeyValue The value of the GSI partition key to query
    * @return The query response from DynamoDB
    * @throws ResourceNotFoundException if the table or index doesn't exist
    * @throws DynamoDbException if the query fails
    */
   public QueryResponse queryGlobalSecondaryIndex(
       final String tableName, final String indexName, final String
partitionKeyName, final String partitionKeyValue) {

       CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
       CodeSampleUtils.validateStringParameter("Index name", indexName);

       // Create expression attribute names for the column names
       final Map<String, String> expressionAttributeNames = new HashMap<>();
       expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_IK,
partitionKeyName);

       // Create expression attribute values for the column values
       final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
```

```
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_IK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .indexName(indexName)
            .keyConditionExpression(GSI_KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query on GSI successful. Found " +
response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(
                "Error: The Amazon DynamoDB table \"%s\" or index \"%s\" can't be
found.\n", tableName, indexName);
            throw new DynamoDbQueryException("Table or index not found: " +
tableName + "/" + indexName, e);
        } catch (DynamoDbException e) {
            System.err.println("Error querying GSI: " + e.getMessage());
            throw new DynamoDbQueryException("Failed to execute query on GSI",
e);
        }
    }
```

テーブルの直接クエリと AWS SDK for Java 2.x を使用した GSI のクエリを比較します。

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <basePartitionKeyName> <basePartitionKeyValue>
<gsiName> <gsiPartitionKeyName> <gsiPartitionKeyValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
```

```
                    basePartitionKeyName - The name of the base table partition
key attribute.
                    basePartitionKeyValue - The value of the base table partition
key to query.
                    gsiName - The name of the Global Secondary Index.
                    gsiPartitionKeyName - The name of the GSI partition key
attribute.
                    gsiPartitionKeyValue - The value of the GSI partition key to
query.
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """";

        if (args.length < 6) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String basePartitionKeyName = args[1];
        final String basePartitionKeyValue = args[2];
        final String gsiName = args[3];
        final String gsiPartitionKeyName = args[4];
        final String gsiPartitionKeyValue = args[5];
        final Region region = args.length > 6 ? Region.of(args[6]) :
Region.US_EAST_1;

        try (DynamoDbClient ddb =
DynamoDbClient.builder().region(region).build()) {
            final QueryTableAndGSI queryHelper = new QueryTableAndGSI(ddb);

            // Query the base table
            System.out.println("Querying base table where " +
basePartitionKeyName + " = " + basePartitionKeyValue);
            final QueryResponse tableResponse =
                queryHelper.queryTable(tableName, basePartitionKeyName,
basePartitionKeyValue);

            System.out.println("Found " + tableResponse.count() + " items in base
table:");
            tableResponse.items().forEach(item -> System.out.println(item));

            // Query the GSI
            System.out.println(
```

```
                "\nQuerying GSI '" + gsiName + "' where " + gsiPartitionKeyName +
" = " + gsiPartitionKeyValue);
            final QueryResponse gsiResponse =
                queryHelper.queryGlobalSecondaryIndex(tableName, gsiName,
gsiPartitionKeyName, gsiPartitionKeyValue);

            System.out.println("Found " + gsiResponse.count() + " items in
GSI:");
            gsiResponse.items().forEach(item -> System.out.println(item));

            // Explain the differences between querying a table and a GSI
            System.out.println("\nKey differences between querying a table and a
GSI:");
            System.out.println("1. When querying a GSI, you must specify the
indexName parameter");
            System.out.println("2. GSIs may not contain all attributes from the
base table (projection)");
            System.out.println("3. GSIs consume read capacity units from the
GSI's capacity, not the base table's");
            System.out.println("4. GSIs may have eventually consistent data
(cannot use ConsistentRead=true)");

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table or index not found: " + e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript でプライマリキーを使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table using the primary key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} userId - The user ID to query by (partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryTable(
  config,
  tableName,
  userId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the base table
    const input = {
      TableName: tableName,
      KeyConditionExpression: "user_id = :userId",
      ExpressionAttributeValues: {
        ":userId": { S: userId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying table: ${error}`);
    throw error;
  }
}
```

AWS SDK for JavaScript を使用して DynamoDB グローバルセカンダリインデックス (GSI) をクエリします。

```
/**
 * Queries a DynamoDB Global Secondary Index (GSI)
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} indexName - The name of the GSI to query
 * @param {string} gameId - The game ID to query by (GSI partition key)
 * @returns {Promise<Object>} - The query response
 */
async function queryGSI(
  config,
  tableName,
  indexName,
  gameId
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input for the GSI
    const input = {
      TableName: tableName,
      IndexName: indexName,
      KeyConditionExpression: "game_id = :gameId",
      ExpressionAttributeValues: {
        ":gameId": { S: gameId }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying GSI: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でプライマリキーとグローバルセカンダリインデックス (GSI) を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Key


def query_table(table_name, partition_key_name, partition_key_value):
    """
    Query a DynamoDB table using its primary key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the table's primary key
    response =
 table.query(KeyConditionExpression=Key(partition_key_name).eq(partition_key_value))

    return response


def query_gsi(table_name, index_name, partition_key_name, partition_key_value):
    """
    Query a Global Secondary Index (GSI) on a DynamoDB table.

    Args:
        table_name (str): The name of the DynamoDB table.
```

```
        index_name (str): The name of the Global Secondary Index.
        partition_key_name (str): The name of the GSI's partition key attribute.
        partition_key_value (str): The value of the GSI's partition key to query.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query on the GSI
    response = table.query(
        IndexName=index_name,
KeyConditionExpression=Key(partition_key_name).eq(partition_key_value)
    )

    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で begins_with 条件を使用して DynamoDB テーブルをクエリする

以下のコード例は、begins_with 条件を使用してテーブルをクエリする方法を示しています。

- キー条件式で begins_with 関数を使用します。
- ソートキーのプレフィックスパターンに基づいて項目をフィルタリングします。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でソートキーの begins_with 条件を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithBeginsWithCondition(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String sortKeyName,
        final String sortKeyPrefix) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Sort key name", sortKeyName);
        CodeSampleUtils.validateStringParameter("Sort key prefix",
 sortKeyPrefix);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, sortKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
```

```
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_SK_PREFIX,
            AttributeValue.builder().s(sortKeyPrefix).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with begins_with condition successful.
Found {0} items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with begins_with condition",
e);
            throw e;
        }
    }
```

AWS SDK for Java 2.x でプレフィックス長が異なる begins_with の使用を示します。

```
    public static void main(String[] args) {
        try {
            CodeSampleUtils.BeginsWithQueryConfig config =
CodeSampleUtils.BeginsWithQueryConfig.fromArgs(args);
            LOGGER.log(Level.INFO, "Querying items where {0} = {1} and {2} begins
with ''{3}''", new Object[] {
                config.getPartitionKeyName(),
                config.getPartitionKeyValue(),
                config.getSortKeyName(),
                config.getSortKeyPrefix()
            });

            // Using the builder pattern to create and execute the query
```

```
            final QueryResponse response = new BeginsWithQueryBuilder()
                .withTableName(config.getTableName())
                .withPartitionKeyName(config.getPartitionKeyName())
                .withPartitionKeyValue(config.getPartitionKeyValue())
                .withSortKeyName(config.getSortKeyName())
                .withSortKeyPrefix(config.getSortKeyPrefix())
                .withRegion(config.getRegion())
                .execute();

            // Process the results
            LOGGER.log(Level.INFO, "Found {0} items:", response.count());
            response.items().forEach(item -> LOGGER.info(item.toString()));

            // Demonstrate with a different prefix
            if (!config.getSortKeyPrefix().isEmpty()) {
                String shorterPrefix = config.getSortKeyPrefix()
                    .substring(0, Math.max(1,
config.getSortKeyPrefix().length() / 2));
                LOGGER.log(Level.INFO, "\nNow querying with a shorter prefix:
''{0}''", shorterPrefix);

                final QueryResponse response2 = new BeginsWithQueryBuilder()
                    .withTableName(config.getTableName())
                    .withPartitionKeyName(config.getPartitionKeyName())
                    .withPartitionKeyValue(config.getPartitionKeyValue())
                    .withSortKeyName(config.getSortKeyName())
                    .withSortKeyPrefix(shorterPrefix)
                    .withRegion(config.getRegion())
                    .execute();

                LOGGER.log(Level.INFO, "Found {0} items with shorter prefix:",
response2.count());
                response2.items().forEach(item -> LOGGER.info(item.toString()));
            }
        } catch (IllegalArgumentException e) {
            LOGGER.log(Level.SEVERE, "Invalid input: {0}", e.getMessage());
            printUsage();
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found", e);
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "DynamoDB error", e);
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Unexpected error", e);
        }
```

```
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript でソートキーの begins_with 条件を使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items where the sort key begins with a specific
 prefix
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} sortKeyName - The name of the sort key
 * @param {string} prefix - The prefix to match at the beginning of the sort key
 * @returns {Promise<Object>} - The query response
 */
async function queryWithBeginsWith(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  prefix
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
```

```
      KeyConditionExpression: "#pk = :pkValue AND begins_with(#sk, :prefix)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":prefix": { S: prefix }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with begins_with: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でソートキーの begins_with 条件を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_begins_with(
    table_name, partition_key_name, partition_key_value, sort_key_name, prefix
):
    """
    Query a DynamoDB table with a begins_with condition on the sort key.

    Args:
```

```
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute.
        prefix (str): The prefix to match at the beginning of the sort key.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Perform the query with a begins_with condition on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) & Key(
        sort_key_name
    ).begins_with(prefix)
    response = table.query(KeyConditionExpression=key_condition)

    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK でソートキーの日付範囲を使用して DynamoDB テーブルをクエリする

以下のコード例は、ソートキーの日付範囲を使用してテーブルをクエリする方法を示しています。

- 特定の日付範囲内の項目をクエリします。
- 日付形式のソートキーで比較演算子を使用します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して、日付範囲内の項目について DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithDateRange(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDate startDate,
        final LocalDate endDate) {

        // Focus on query logic, assuming parameters are valid
        if (startDate == null || endDate == null) {
            throw new IllegalArgumentException("Start date and end date cannot be
 null");
        }

        if (endDate.isBefore(startDate)) {
            throw new IllegalArgumentException("End date must be after start
 date");
        }

        // Format dates as ISO strings for DynamoDB (using just the date part)
        final String formattedStartDate = startDate.toString();
        final String formattedEndDate = endDate.toString();
```

```
        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_START_DATE,
            AttributeValue.builder().s(formattedStartDate).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_END_DATE,
            AttributeValue.builder().s(formattedEndDate).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query by date range successful. Found {0}
items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying by date range: {0}",
e.getMessage());
            throw e;
        }
    }
```

日付範囲フィルタリングを使用して DynamoDB テーブルをクエリする方法を示します。

```
   public static void main(String[] args) {
       final String usage =
           """
               Usage:
                   <tableName> <partitionKeyName> <partitionKeyValue>
<dateKeyName> <startDate> <endDate> [region]
               Where:
                   tableName - The Amazon DynamoDB table to query.
                   partitionKeyName - The name of the partition key attribute.
                   partitionKeyValue - The value of the partition key to query.
                   dateKeyName - The name of the date attribute to filter on.
                   startDate - The start date for the range query (YYYY-MM-DD).
                   endDate - The end date for the range query (YYYY-MM-DD).
                   region (optional) - The AWS region where the table exists.
(Default: us-east-1)
               """;

       if (args.length < 6) {
           System.out.println(usage);
           System.exit(1);
       }

       try {
           // Parse command line arguments into a config object
           CodeSampleUtils.DateRangeQueryConfig config =
CodeSampleUtils.DateRangeQueryConfig.fromArgs(args);

           LOGGER.log(
               Level.INFO, "Querying items from {0} to {1}", new Object[]
{config.getStartDate(), config.getEndDate()
               });

           // Using the builder pattern to create and execute the query
           final QueryResponse response = new DateRangeQueryBuilder()
               .withTableName(config.getTableName())
               .withPartitionKeyName(config.getPartitionKeyName())
               .withPartitionKeyValue(config.getPartitionKeyValue())
               .withDateKeyName(config.getDateKeyName())
               .withStartDate(config.getStartDate())
               .withEndDate(config.getEndDate())
               .withRegion(config.getRegion())
               .execute();
```

```
            // Process the results
            LOGGER.log(Level.INFO, "Found {0} items:", response.count());
            response.items().forEach(item -> {
                LOGGER.info(item.toString());

                // Extract and display the date attribute for clarity
                if (item.containsKey(config.getDateKeyName())) {
                    LOGGER.log(
                        Level.INFO,
                        "  Date attribute: {0}",
                        item.get(config.getDateKeyName()).s());
                }
            });

            // Demonstrate with a different date range
            LocalDate narrowerStartDate = config.getStartDate().plusDays(1);
            LocalDate narrowerEndDate = config.getEndDate().minusDays(1);

            if (!narrowerStartDate.isAfter(narrowerEndDate)) {
                LOGGER.log(Level.INFO, "\nNow querying with a narrower date
range: {0} to {1}", new Object[] {
                    narrowerStartDate, narrowerEndDate
                });

                final QueryResponse response2 = new DateRangeQueryBuilder()
                    .withTableName(config.getTableName())
                    .withPartitionKeyName(config.getPartitionKeyName())
                    .withPartitionKeyValue(config.getPartitionKeyValue())
                    .withDateKeyName(config.getDateKeyName())
                    .withStartDate(narrowerStartDate)
                    .withEndDate(narrowerEndDate)
                    .withRegion(config.getRegion())
                    .execute();

                LOGGER.log(Level.INFO, "Found {0} items with narrower date
range:", response2.count());
                response2.items().forEach(item -> LOGGER.info(item.toString()));
            }

            LOGGER.info("\nNote: When storing dates in DynamoDB:");
            LOGGER.info("1. Use ISO format (YYYY-MM-DD) for lexicographical
ordering");
```

```
            LOGGER.info("2. Use the BETWEEN operator for inclusive date range
 queries");
            LOGGER.info("3. Consider using ISO-8601 format for timestamps with
 time components");

        } catch (IllegalArgumentException e) {
            LOGGER.log(Level.SEVERE, "Invalid input: {0}", e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", e.getMessage());
            System.exit(1);
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "DynamoDB error: {0}", e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            LOGGER.log(Level.SEVERE, "Unexpected error: {0}", e.getMessage());
            System.exit(1);
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して、日付範囲内の項目について DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range on the sort
 key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
```

```
 * @param {string} sortKeyName - The name of the sort key (must be a date/time
 attribute)
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
async function queryByDateRangeOnSortKey(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: '#pk = :pkValue AND #sk BETWEEN :startDate
 AND :endDate',
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range on sort key: ${error}`);
```

```
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して、日付範囲内の項目について DynamoDB テーブルをクエリします。

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_date_range(
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        start_date (datetime): The start date for the query range.
        end_date (datetime): The end date for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
```

```
    table = dynamodb.Table(table_name)

    # Format the date values as ISO 8601 strings
    # DynamoDB works well with ISO format for date values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key using BETWEEN operator
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response


def query_with_date_range_by_month(
    table_name, partition_key_name, partition_key_value, sort_key_name, year,
 month
):
    """
    Query a DynamoDB table for a specific month's data.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        year (int): The year to query.
        month (int): The month to query (1-12).

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
```

```
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Calculate the start and end dates for the specified month
    if month == 12:
        next_year = year + 1
        next_month = 1
    else:
        next_year = year
        next_month = month + 1

    start_date = datetime(year, month, 1)
    end_date = datetime(next_year, next_month, 1) - timedelta(microseconds=1)

    # Format the date values as ISO 8601 strings
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(KeyConditionExpression=key_condition)

    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で複雑なフィルター式を使用して DynamoDB テーブルをクエリする

以下のコード例は、複雑なフィルター式を使用してテーブルをクエリする方法を示しています。

- 複雑なフィルター式をクエリ結果に適用します。
- 論理演算子を使用して複数の条件を組み合わせます。
- キー以外の属性に基づいて項目をフィルタリングします。

Java

SDK for Java 2.x

AWS SDK for Java 2.x で複雑なフィルター式を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithComplexFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String statusAttrName,
        final String activeStatus,
        final String pendingStatus,
        final String priceAttrName,
        final double minPrice,
        final double maxPrice,
        final String categoryAttrName) {
```

```
        // Validate parameters
        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Status attribute name",
statusAttrName);
        CodeSampleUtils.validateStringParameter("Active status", activeStatus);
        CodeSampleUtils.validateStringParameter("Pending status", pendingStatus);
        CodeSampleUtils.validateStringParameter("Price attribute name",
priceAttrName);
        CodeSampleUtils.validateStringParameter("Category attribute name",
categoryAttrName);
        CodeSampleUtils.validateNumericRange("Minimum price", minPrice, 0.0,
Double.MAX_VALUE);
        CodeSampleUtils.validateNumericRange("Maximum price", maxPrice, minPrice,
Double.MAX_VALUE);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pk", partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_STATUS,
statusAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PRICE,
priceAttrName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_CATEGORY,
categoryAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":pkValue", AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_ACTIVE,
            AttributeValue.builder().s(activeStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PENDING,
            AttributeValue.builder().s(pendingStatus).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MIN_PRICE,
            AttributeValue.builder().n(String.valueOf(minPrice)).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_MAX_PRICE,
            AttributeValue.builder().n(String.valueOf(maxPrice)).build());
```

```
        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        return dynamoDbClient.query(queryRequest);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で複雑なフィルター式を使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with a complex filter expression
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number|string} minViews - Minimum number of views for filtering
 * @param {number|string} minReplies - Minimum number of replies for filtering
 * @param {string} requiredTag - Tag that must be present in the item's tags set
 * @returns {Promise<Object>} - The query response
 */
async function queryWithComplexFilter(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
```

```
  minViews,
  minReplies,
  requiredTag
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      FilterExpression: "views >= :minViews AND replies >= :minReplies AND
 contains(tags, :tag)",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":minViews": { N: minViews.toString() },
        ":minReplies": { N: minReplies.toString() },
        ":tag": { S: requiredTag }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with complex filter: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で複雑なフィルター式を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_complex_filter(
    table_name,
    partition_key_name,
    partition_key_value,
    min_rating=None,
    status_list=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        min_rating (float, optional): Minimum rating value for filtering.
        status_list (list, optional): List of status values to include.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize the filter expression and expression attribute values
    filter_expression = None
    expression_attribute_values = {}
```

```
    # Build the filter expression based on provided parameters
    if min_rating is not None:
        filter_expression = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if status_list and len(status_list) > 0:
        status_condition = None
        for i, status in enumerate(status_list):
            status_value_name = f":status{i}"
            expression_attribute_values[status_value_name] = status

            if status_condition is None:
                status_condition = Attr("status").eq(status)
            else:
                status_condition = status_condition | Attr("status").eq(status)

        if filter_expression is None:
            filter_expression = status_condition
        else:
            filter_expression = filter_expression & status_condition

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if filter_expression is None:
            filter_expression = price_condition
        else:
            filter_expression = filter_expression & price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response
```

```
def query_with_complex_filter_and_or(
    table_name,
    partition_key_name,
    partition_key_value,
    category=None,
    min_rating=None,
    max_price=None,
):
    """
    Query a DynamoDB table with a complex filter expression using AND and OR
 operators.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        category (str, optional): Category value for filtering.
        min_rating (float, optional): Minimum rating value for filtering.
        max_price (float, optional): Maximum price value for filtering.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build a complex filter expression with AND and OR operators
    filter_expression = None
    expression_attribute_values = {}

    # Build the category condition
    if category:
        filter_expression = Attr("category").eq(category)
        expression_attribute_values[":category"] = category

    # Build the rating and price condition (rating >= min_rating OR price <=
 max_price)
    rating_price_condition = None

    if min_rating is not None:
```

```
        rating_price_condition = Attr("rating").gte(min_rating)
        expression_attribute_values[":min_rating"] = min_rating

    if max_price is not None:
        price_condition = Attr("price").lte(max_price)
        expression_attribute_values[":max_price"] = max_price

        if rating_price_condition is None:
            rating_price_condition = price_condition
        else:
            rating_price_condition = rating_price_condition | price_condition

    # Combine the conditions
    if rating_price_condition:
        if filter_expression is None:
            filter_expression = rating_price_condition
        else:
            filter_expression = filter_expression & rating_price_condition

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression
        if expression_attribute_values:
            query_params["ExpressionAttributeValues"] =
expression_attribute_values

    # Execute the query
    response = table.query(**query_params)
    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で動的フィルター式を使用して DynamoDB テーブルをクエリする

以下のコード例は、動的フィルター式を使用してテーブルをクエリする方法を示しています。

- ランタイムにフィルター式を動的に構築します。
- ユーザー入力またはアプリケーション状態に基づいてフィルター条件を構築します。
- 条件付きでフィルター条件を追加または削除します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x で動的に構築されたフィルター式を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public static QueryResponse queryWithDynamicFilter(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final Map<String, Object> filterCriteria,
        final Region region,
        final DynamoDbClient dynamoDbClient) {

        validateParameters(tableName, partitionKeyName, partitionKeyValue,
 filterCriteria);

        DynamoDbClient ddbClient = dynamoDbClient;
        boolean shouldClose = false;
```

```
        try {
            if (ddbClient == null) {
                ddbClient = createClient(region);
                shouldClose = true;
            }

            final QueryWithDynamicFilter queryHelper = new
QueryWithDynamicFilter(ddbClient);
            return queryHelper.queryWithDynamicFilter(tableName,
partitionKeyName, partitionKeyValue, filterCriteria);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Failed to execute dynamic filter query: " +
e.getMessage());
            throw e;
        } catch (Exception e) {
            System.err.println("Unexpected error during query: " +
e.getMessage());
            throw e;
        } finally {
            if (shouldClose && ddbClient != null) {
                ddbClient.close();
            }
        }
    }
```

AWS SDK for Java 2.x で動的フィルター式を使用する方法を示します。

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue>
<filterAttrName> <filterAttrValue> [region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    filterAttrName - The name of the attribute to filter on.
                    filterAttrValue - The value to filter by.
```

```
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 5) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final String filterAttrName = args[3];
        final String filterAttrValue = args[4];
        final Region region = args.length > 5 ? Region.of(args[5]) :
Region.US_EAST_1;

        System.out.println("Querying items with dynamic filter: " +
filterAttrName + " = " + filterAttrValue);

        try {
            // Using the builder pattern to create and execute the query
            final QueryResponse response = new DynamicFilterQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriterion(filterAttrName, filterAttrValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> System.out.println(item));

            // Demonstrate multiple filter criteria
            System.out.println("\nNow querying with multiple filter criteria:");

            Map<String, Object> multipleFilters = new HashMap<>();
            multipleFilters.put(filterAttrName, filterAttrValue);
            multipleFilters.put("status", "active");

            final QueryResponse multiFilterResponse = new
DynamicFilterQueryBuilder()
                .withTableName(tableName)
```

```
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withFilterCriteria(multipleFilters)
                .withRegion(region)
                .execute();

            System.out.println("Found " + multiFilterResponse.count() + " items
 with multiple filters:");
            multiFilterResponse.items().forEach(item ->
 System.out.println(item));

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で動的に構築されたフィルター式を使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

async function queryWithDynamicFilter(
  config,
  tableName,
```

```
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  sortKeyValue,
  filterParams = {}
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize filter expression components
    let filterExpressions = [];
    const expressionAttributeValues = {
      ":pkValue": { S: partitionKeyValue },
      ":skValue": { S: sortKeyValue }
    };
    const expressionAttributeNames = {
      "#pk": partitionKeyName,
      "#sk": sortKeyName
    };

    // Add status filter if provided
    if (filterParams.status) {
      filterExpressions.push("status = :status");
      expressionAttributeValues[":status"] = { S: filterParams.status };
    }

    // Add minimum views filter if provided
    if (filterParams.minViews !== undefined) {
      filterExpressions.push("views >= :minViews");
      expressionAttributeValues[":minViews"] = { N:
 filterParams.minViews.toString() };
    }

    // Add author filter if provided
    if (filterParams.author) {
      filterExpressions.push("author = :author");
      expressionAttributeValues[":author"] = { S: filterParams.author };
    }

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue AND #sk = :skValue"
```

```
    };

    // Add filter expression if any filters were provided
    if (filterExpressions.length > 0) {
      input.FilterExpression = filterExpressions.join(" AND ");
    }

    // Add expression attribute names and values
    input.ExpressionAttributeNames = expressionAttributeNames;
    input.ExpressionAttributeValues = expressionAttributeValues;

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with dynamic filter: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で動的に構築されたフィルター式を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_dynamic_filter(
    table_name, partition_key_name, partition_key_value, filter_conditions=None
):
    """
    Query a DynamoDB table with a dynamically constructed filter expression.

    Args:
```

```
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_conditions (dict, optional): A dictionary of filter conditions
where
            keys are attribute names and values are dictionaries with 'operator'
and 'value'.
            Example: {'rating': {'operator': '>=', 'value': 4}, 'status':
{'operator': '=', 'value': 'active'}}

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Start with the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Initialize variables for the filter expression and attribute values
    filter_expression = None
    expression_attribute_values = {":pk_val": partition_key_value}

    # Dynamically build the filter expression if filter conditions are provided
    if filter_conditions:
        for attr_name, condition in filter_conditions.items():
            operator = condition.get("operator")
            value = condition.get("value")
            attr_value_name = f":{attr_name}"
            expression_attribute_values[attr_value_name] = value

            # Create the appropriate filter expression based on the operator
            current_condition = None
            if operator == "=":
                current_condition = Attr(attr_name).eq(value)
            elif operator == "!=":
                current_condition = Attr(attr_name).ne(value)
            elif operator == ">":
                current_condition = Attr(attr_name).gt(value)
            elif operator == ">=":
                current_condition = Attr(attr_name).gte(value)
            elif operator == "<":
                current_condition = Attr(attr_name).lt(value)
```

```
            elif operator == "<=":
                current_condition = Attr(attr_name).lte(value)
            elif operator == "contains":
                current_condition = Attr(attr_name).contains(value)
            elif operator == "begins_with":
                current_condition = Attr(attr_name).begins_with(value)

            # Combine with existing filter expression using AND
            if current_condition:
                if filter_expression is None:
                    filter_expression = current_condition
                else:
                    filter_expression = filter_expression & current_condition

    # Perform the query with the dynamically built filter expression
    query_params = {"KeyConditionExpression": key_condition}

    if filter_expression:
        query_params["FilterExpression"] = filter_expression

    response = table.query(**query_params)
    return response
```

AWS SDK for Python (Boto3) で動的フィルター式を使用する方法を示します。

```
def example_usage():
    """Example of how to use the query_with_dynamic_filter function."""
    # Example parameters
    table_name = "Products"
    partition_key_name = "Category"
    partition_key_value = "Electronics"

    # Define dynamic filter conditions based on user input or runtime conditions
    user_min_rating = 4  # This could come from user input
    user_status_filter = "active"  # This could come from user input

    filter_conditions = {}

    # Only add conditions that are actually specified
    if user_min_rating is not None:
```

```
        filter_conditions["rating"] = {"operator": ">=", "value":
user_min_rating}

    if user_status_filter:
        filter_conditions["status"] = {"operator": "=", "value":
user_status_filter}

    print(
        f"Querying products in category '{partition_key_value}' with filter
conditions: {filter_conditions}"
    )

    # Execute the query with dynamic filter
    response = query_with_dynamic_filter(
        table_name, partition_key_name, partition_key_value, filter_conditions
    )

    # Process the results
    items = response.get("Items", [])
    print(f"Found {len(items)} items")

    for item in items:
        print(f"Product: {item}")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK でフィルター式と制限を使用して DynamoDB テーブルをクエリする

以下のコード例は、フィルター式と制限を使用してテーブルをクエリする方法を示しています。

- 評価対象項目を制限してクエリ結果にフィルター式を適用します。
- フィルタリングされたクエリ結果に制限がどのように影響するかを理解します。

- クエリで処理される項目の最大数を制御します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でフィルター式と制限を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithFilterAndLimit(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String filterAttrName,
        final String filterAttrValue,
        final int limit) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Filter attribute name",
 filterAttrName);
        CodeSampleUtils.validateStringParameter("Filter attribute value",
 filterAttrValue);
        CodeSampleUtils.validatePositiveInteger("Limit", limit);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
```

```
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_FILTER,
filterAttrName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_FILTER,
            AttributeValue.builder().s(filterAttrValue).build());

        // Create the filter expression
        final String filterExpression = "#filterAttr = :filterValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(limit)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query with filter and limit successful. Found
{0} items", response.count());
            LOGGER.log(
                Level.INFO, "ScannedCount: {0} (total items evaluated before
filtering)", response.scannedCount());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with filter and limit: {0}",
e.getMessage());
            throw e;
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でフィルター式と制限を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_filter_and_limit(
    table_name,
    partition_key_name,
    partition_key_value,
    filter_attribute=None,
    filter_value=None,
    limit=10,
):
    """
    Query a DynamoDB table with a filter expression and limit the number of
 results.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_attribute (str, optional): The attribute name to filter on.
        filter_value (any, optional): The value to compare against in the filter.
        limit (int, optional): The maximum number of items to evaluate. Defaults
 to 10.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
```

```
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Prepare the query parameters
    query_params = {"KeyConditionExpression": key_condition, "Limit": limit}

    # Add the filter expression if filter attributes are provided
    if filter_attribute and filter_value is not None:
        query_params["FilterExpression"] =
 Attr(filter_attribute).gt(filter_value)
        query_params["ExpressionAttributeValues"] = {":filter_value":
 filter_value}

    # Execute the query
    response = table.query(**query_params)
    return response
```

AWS SDK for Python (Boto3) でフィルター式と制限を使用する方法を示します。

```
def example_usage():
    """Example of how to use the query_with_filter_and_limit function."""
    # Example parameters
    table_name = "ProductReviews"
    partition_key_name = "ProductId"
    partition_key_value = "P123456"
    filter_attribute = "Rating"
    filter_value = 3  # Filter for ratings > 3
    limit = 5

    print(f"Querying reviews for product '{partition_key_value}' with rating >
 {filter_value}")
    print(f"Limiting to {limit} evaluated items")

    # Execute the query with filter and limit
    response = query_with_filter_and_limit(
        table_name, partition_key_name, partition_key_value, filter_attribute,
 filter_value, limit
    )
```

```
    # Process the results
    items = response.get("Items", [])
    print(f"\nReturned {len(items)} items that passed the filter")

    for item in items:
        print(f"Review: {item}")

    # Explain the difference between Limit and actual results
    explain_limit_vs_results(response)

    # Check if there are more results
    if "LastEvaluatedKey" in response:
        print("\nThere are more results available. Use the LastEvaluatedKey for
pagination.")
    else:
        print("\nAll matching results have been retrieved.")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK でネストされた属性を使用して DynamoDB テーブルをクエリする

以下のコード例は、ネストされた属性を使用してテーブルをクエリする方法を示しています。

- DynamoDB 項目のネストされた属性を使用してアクセスおよびフィルタリングします。
- ネストされた要素を参照するには、ドキュメントパス式を使用します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でネストされた属性を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithNestedAttributes(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String nestedPath,
        final String nestedAttr,
        final String nestedValue) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Nested path", nestedPath);
        CodeSampleUtils.validateStringParameter("Nested attribute", nestedAttr);
        CodeSampleUtils.validateStringParameter("Nested value", nestedValue);

        // Split the nested path into components
        final String[] pathComponents = nestedPath.split("\\.");

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Build the nested attribute reference using document path notation
        final StringBuilder nestedAttributeRef = new StringBuilder();
        for (int i = 0; i < pathComponents.length; i++) {
            final String aliasName = "#n" + i;
            expressionAttributeNames.put(aliasName, pathComponents[i]);
```

```
            if (i > 0) {
                nestedAttributeRef.append(".");
            }
            nestedAttributeRef.append(aliasName);
        }

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_NESTED,
            AttributeValue.builder().s(nestedValue).build());

        // Create the filter expression using the nested attribute reference
        final String filterExpression = nestedAttributeRef + " = :nestedValue";

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(filterExpression)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query with nested attribute filter successful.
Found " + response.count() + " items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with nested attribute filter: " +
e.getMessage());
            throw e;
        }
    }
```

ネストされた属性を使用して DynamoDB テーブルをクエリする方法を示します。

```
   public static void main(String[] args) {
       final String usage =
           """
               Usage:
                   <tableName> <partitionKeyName> <partitionKeyValue>
<nestedPath> <nestedAttr> <nestedValue> [region]
               Where:
                   tableName - The Amazon DynamoDB table to query.
                   partitionKeyName - The name of the partition key attribute.
                   partitionKeyValue - The value of the partition key to query.
                   nestedPath - The path to the nested map attribute (e.g.,
"address").
                   nestedAttr - The name of the nested attribute (e.g., "city").
                   nestedValue - The value to filter by (e.g., "Seattle").
                   region (optional) - The AWS region where the table exists.
(Default: us-east-1)
               """;

       if (args.length < 6) {
           System.out.println(usage);
           System.exit(1);
       }

       final String tableName = args[0];
       final String partitionKeyName = args[1];
       final String partitionKeyValue = args[2];
       final String nestedPath = args[3];
       final String nestedAttr = args[4];
       final String nestedValue = args[5];
       final Region region = args.length > 6 ? Region.of(args[6]) :
Region.US_EAST_1;

       System.out.println("Querying items where " + partitionKeyName + " = " +
partitionKeyValue + " and " + nestedPath
           + "." + nestedAttr + " = " + nestedValue);

       try {
           // Using the builder pattern to create and execute the query
           final QueryResponse response = new NestedAttributeQueryBuilder()
```

```
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withNestedPath(nestedPath)
                .withNestedAttribute(nestedAttr)
                .withNestedValue(nestedValue)
                .withRegion(region)
                .execute();

            // Process the results
            System.out.println("Found " + response.count() + " items:");
            response.items().forEach(item -> {
                System.out.println(item);

                // Extract and display the nested attribute for clarity
                if (item.containsKey(nestedPath) && item.get(nestedPath).hasM())
{
                    Map<String, AttributeValue> nestedMap =
item.get(nestedPath).m();
                    if (nestedMap.containsKey(nestedAttr)) {
                        System.out.println("  Nested attribute " + nestedPath +
"." + nestedAttr + ": "
                                + formatAttributeValue(nestedMap.get(nestedAttr)));
                    }
                }
            });

            System.out.println("\nNote: When working with nested attributes in
DynamoDB:");
            System.out.println("1. Use dot notation in filter expressions to
access nested attributes");
            System.out.println("2. Use expression attribute names for each
component of the path");
            System.out.println("3. Check if the nested attribute exists before
accessing it");

        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
```

```
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript でネストされた属性を使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table filtering on a nested attribute
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} productId - The product ID to query by (partition key)
 * @param {string} category - The category to filter by (nested attribute)
 * @returns {Promise<Object>} - The query response
 */
async function queryWithNestedAttribute(
  config,
  tableName,
  productId,
  category
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
```

```
      KeyConditionExpression: "product_id = :productId",
      FilterExpression: "details.category = :category",
      ExpressionAttributeValues: {
        ":productId": { S: productId },
        ":category": { S: category }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with nested attribute: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でネストされた属性を使用して DynamoDB テーブルをクエリします。

```
from typing import Any, Dict, List

import boto3
from boto3.dynamodb.conditions import Attr, Key


def query_with_nested_attributes(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
    nested_path: str,
    comparison_operator: str,
    comparison_value: Any,
) -> Dict[str, Any]:
```

```
    """
    Query a DynamoDB table and filter by nested attributes.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        nested_path (str): The path to the nested attribute (e.g.,
'specs.weight').
        comparison_operator (str): The comparison operator to use ('=', '!=',
'<', '<=', '>', '>=').
        comparison_value (any): The value to compare against.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build the filter expression based on the nested attribute path and
comparison operator
    filter_expression = None
    if comparison_operator == "=":
        filter_expression = Attr(nested_path).eq(comparison_value)
    elif comparison_operator == "!=":
        filter_expression = Attr(nested_path).ne(comparison_value)
    elif comparison_operator == "<":
        filter_expression = Attr(nested_path).lt(comparison_value)
    elif comparison_operator == "<=":
        filter_expression = Attr(nested_path).lte(comparison_value)
    elif comparison_operator == ">":
        filter_expression = Attr(nested_path).gt(comparison_value)
    elif comparison_operator == ">=":
        filter_expression = Attr(nested_path).gte(comparison_value)
    elif comparison_operator == "contains":
        filter_expression = Attr(nested_path).contains(comparison_value)
    elif comparison_operator == "begins_with":
        filter_expression = Attr(nested_path).begins_with(comparison_value)

    # Execute the query with the filter expression
```

```
    response = table.query(KeyConditionExpression=key_condition,
 FilterExpression=filter_expression)

    return response


def query_with_multiple_nested_attributes(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
    nested_conditions: List[Dict[str, Any]],
) -> Dict[str, Any]:
    """
    Query a DynamoDB table and filter by multiple nested attributes.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        nested_conditions (list): A list of dictionaries, each containing:
            - path (str): The path to the nested attribute
            - operator (str): The comparison operator
            - value (any): The value to compare against

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    # Build the combined filter expression for all nested attributes
    combined_filter = None

    for condition in nested_conditions:
        if not isinstance(condition, dict):
            continue
        path = condition.get("path", "")
        operator = condition.get("operator", "")
        value = condition.get("value")
```

```
        if not path or not operator:
            continue

        # Build the individual filter expression
        current_filter = None
        if operator == "=":
            current_filter = Attr(path).eq(value)
        elif operator == "!=":
            current_filter = Attr(path).ne(value)
        elif operator == "<":
            current_filter = Attr(path).lt(value)
        elif operator == "<=":
            current_filter = Attr(path).lte(value)
        elif operator == ">":
            current_filter = Attr(path).gt(value)
        elif operator == ">=":
            current_filter = Attr(path).gte(value)
        elif operator == "contains":
            current_filter = Attr(path).contains(value)
        elif operator == "begins_with":
            current_filter = Attr(path).begins_with(value)

        # Combine with the existing filter using AND
        if current_filter:
            if combined_filter is None:
                combined_filter = current_filter
            else:
                combined_filter = combined_filter & current_filter

    # Execute the query with the combined filter expression
    response = table.query(KeyConditionExpression=key_condition,
FilterExpression=combined_filter)

    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK でページ分割を使用して DynamoDB テーブルをクエリする

以下のコード例は、ページ分割を使用してテーブルをクエリする方法を示しています。

- DynamoDB クエリ結果のページ分割を実装します。
- LastEvaluatedKey を使用して後続のページを取得します。
- Limit パラメータを使用して、ページあたりの項目数を制御します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でページ分割を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    public List<Map<String, AttributeValue>> queryWithPagination(
        final String tableName, final String partitionKeyName, final String
 partitionKeyValue, final int pageSize) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validatePositiveInteger("Page size", pageSize);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
```

```
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
        QueryRequest.Builder queryRequestBuilder = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .limit(pageSize);

        // List to store all items from all pages
        final List<Map<String, AttributeValue>> allItems = new ArrayList<>();

        // Map to store the last evaluated key for pagination
        Map<String, AttributeValue> lastEvaluatedKey = null;
        int pageNumber = 1;

        try {
            do {
                // If we have a last evaluated key, use it for the next page
                if (lastEvaluatedKey != null) {
                    queryRequestBuilder.exclusiveStartKey(lastEvaluatedKey);
                }

                // Execute the query
                final QueryResponse response =
dynamoDbClient.query(queryRequestBuilder.build());

                // Process the current page of results
                final List<Map<String, AttributeValue>> pageItems =
response.items();
                allItems.addAll(pageItems);

                // Get the last evaluated key for the next page
                lastEvaluatedKey = response.lastEvaluatedKey();
                if (lastEvaluatedKey != null && lastEvaluatedKey.isEmpty()) {
```

```
                    lastEvaluatedKey = null;
                }

                System.out.println("Page " + pageNumber + ": Retrieved " +
pageItems.size() + " items (Running total: "
                    + allItems.size() + ")");

                pageNumber++;

            } while (lastEvaluatedKey != null);

            System.out.println("Query with pagination complete. Retrieved a total
of " + allItems.size()
                + " items across " + (pageNumber - 1) + " pages");

            return allItems;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with pagination: " +
e.getMessage());
            throw e;
        }
    }
```

ページ分割を使用して DynamoDB テーブルをクエリする方法を示します。

```
    public static void main(String[] args) {
        final String usage =
            """
                Usage:
                    <tableName> <partitionKeyName> <partitionKeyValue> [pageSize]
[region]
                Where:
                    tableName - The Amazon DynamoDB table to query.
                    partitionKeyName - The name of the partition key attribute.
                    partitionKeyValue - The value of the partition key to query.
                    pageSize (optional) - The maximum number of items to return
per page. (Default: 10)
```

```
                    region (optional) - The AWS region where the table exists.
(Default: us-east-1)
                """;

        if (args.length < 3) {
            System.out.println(usage);
            System.exit(1);
        }

        final String tableName = args[0];
        final String partitionKeyName = args[1];
        final String partitionKeyValue = args[2];
        final int pageSize = args.length > 3 ? Integer.parseInt(args[3]) : 10;
        final Region region = args.length > 4 ? Region.of(args[4]) :
Region.US_EAST_1;

        System.out.println("Querying items with pagination (page size: " +
pageSize + ")");

        try {
            // Using the builder pattern to create and execute the query
            final List<Map<String, AttributeValue>> allItems = new
PaginationQueryBuilder()
                .withTableName(tableName)
                .withPartitionKeyName(partitionKeyName)
                .withPartitionKeyValue(partitionKeyValue)
                .withPageSize(pageSize)
                .withRegion(region)
                .executeWithPagination();

            // Process the results
            System.out.println("\nSummary: Retrieved a total of " +
allItems.size() + " items");

            // Display the first few items as a sample
            final int sampleSize = Math.min(5, allItems.size());
            if (sampleSize > 0) {
                System.out.println("\nSample of retrieved items (first " +
sampleSize + "):");
                for (int i = 0; i < sampleSize; i++) {
                    System.out.println(allItems.get(i));
                }

                if (allItems.size() > sampleSize) {
```

```
                    System.out.println("... and " + (allItems.size() -
 sampleSize) + " more items");
                }
            }
        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input: " + e.getMessage());
            System.exit(1);
        } catch (ResourceNotFoundException e) {
            System.err.println("Table not found: " + tableName);
            System.exit(1);
        } catch (DynamoDbException e) {
            System.err.println("DynamoDB error: " + e.getMessage());
            System.exit(1);
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
            System.exit(1);
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript でページ分割を使用して DynamoDB テーブルをクエリします。

```
/**
 * Example demonstrating how to handle large query result sets in DynamoDB using
 pagination
 *
 * This example shows:
 * - How to use pagination to handle large result sets
 * - How to use LastEvaluatedKey to retrieve the next page of results
 * - How to construct subsequent query requests using ExclusiveStartKey
 */
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with pagination to handle large result sets
```

```
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {number} pageSize - Number of items per page
 * @returns {Promise<Array>} - All items from the query
 */
async function queryWithPagination(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  pageSize = 25
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Initialize variables for pagination
    let lastEvaluatedKey = undefined;
    const allItems = [];
    let pageCount = 0;

    // Loop until all pages are retrieved
    do {
      // Construct the query input
      const input = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pkValue",
        Limit: pageSize,
        ExpressionAttributeNames: {
          "#pk": partitionKeyName
        },
        ExpressionAttributeValues: {
          ":pkValue": { S: partitionKeyValue }
        }
      };

      // Add ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
      if (lastEvaluatedKey) {
        input.ExclusiveStartKey = lastEvaluatedKey;
      }
```

```
      // Execute the query
      const command = new QueryCommand(input);
      const response = await client.send(command);

      // Process the current page of results
      pageCount++;
      console.log(`Processing page ${pageCount} with ${response.Items.length}
 items`);

      // Add the items from this page to our collection
      if (response.Items && response.Items.length > 0) {
        allItems.push(...response.Items);
      }

      // Get the LastEvaluatedKey for the next page
      lastEvaluatedKey = response.LastEvaluatedKey;

    } while (lastEvaluatedKey); // Continue until there are no more pages

    console.log(`Query complete. Retrieved ${allItems.length} items in
 ${pageCount} pages.`);
    return allItems;
  } catch (error) {
    console.error(`Error querying with pagination: ${error}`);
    throw error;
  }
}

/**
 * Example usage:
 *
 * // Query all items in the "AWS DynamoDB" forum with pagination
 * const allItems = await queryWithPagination(
 *   { region: "us-west-2" },
 *   "ForumThreads",
 *   "ForumName",
 *   "AWS DynamoDB",
 *   25 // 25 items per page
 * );
 *
 * console.log(`Total items retrieved: ${allItems.length}`);
 *
 * // Notes on pagination:
```

```
 * // - LastEvaluatedKey contains the primary key of the last evaluated item
 * // - When LastEvaluatedKey is undefined/null, there are no more items to
 retrieve
 * // - ExclusiveStartKey tells DynamoDB where to start the next page
 * // - Pagination helps manage memory usage for large result sets
 * // - Each page requires a separate network request to DynamoDB
 */

module.exports = { queryWithPagination };
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でページ分割を使用して DynamoDB テーブルをクエリします。

```
import boto3
from boto3.dynamodb.conditions import Key


def query_with_pagination(
    table_name, partition_key_name, partition_key_value, page_size=25,
 max_pages=None
):
    """
    Query a DynamoDB table with pagination to handle large result sets.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.
        max_pages (int, optional): The maximum number of pages to retrieve. If
 None, retrieves all pages.

    Returns:
        list: All items retrieved from the query across all pages.
    """
```

```
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_count = 0
    all_items = []

    # Paginate through the results
    while True:
        # Check if we've reached the maximum number of pages
        if max_pages is not None and page_count >= max_pages:
            break

        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
query
        if last_evaluated_key:
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Process the current page of results
        items = response.get("Items", [])
        all_items.extend(items)

        # Update pagination tracking
        page_count += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if not last_evaluated_key:
            break
```

```
    return all_items


def query_with_pagination_generator(
    table_name, partition_key_name, partition_key_value, page_size=25
):
    """
    Query a DynamoDB table with pagination using a generator to handle large
 result sets.
    This approach is memory-efficient as it yields one page at a time.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        page_size (int, optional): The number of items to return per page.
 Defaults to 25.

    Yields:
        tuple: A tuple containing (items, page_number, last_page) where:
            - items is a list of items for the current page
            - page_number is the current page number (starting from 1)
            - last_page is a boolean indicating if this is the last page
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Initialize variables for pagination
    last_evaluated_key = None
    page_number = 0

    # Paginate through the results
    while True:
        # Prepare the query parameters
        query_params = {
            "KeyConditionExpression":
 Key(partition_key_name).eq(partition_key_value),
            "Limit": page_size,
        }

        # Add the ExclusiveStartKey if we have a LastEvaluatedKey from a previous
 query
        if last_evaluated_key:
```

```
            query_params["ExclusiveStartKey"] = last_evaluated_key

        # Execute the query
        response = table.query(**query_params)

        # Get the current page of results
        items = response.get("Items", [])
        page_number += 1

        # Get the LastEvaluatedKey for the next page, if any
        last_evaluated_key = response.get("LastEvaluatedKey")

        # Determine if this is the last page
        is_last_page = last_evaluated_key is None

        # Yield the current page of results
        yield (items, page_number, is_last_page)

        # If there's no LastEvaluatedKey, we've reached the end of the results
        if is_last_page:
            break
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で強力な整合性のある読み込みを使用して DynamoDB テーブルをクエリする

以下のコード例は、強力な整合性のある読み込みを使用してテーブルをクエリする方法を示しています。

- DynamoDB クエリの整合性レベルを設定します。
- 強力な整合性のある読み込みを使用して最新のデータを取得します。

- 結果整合性と強力な整合性のトレードオフを理解します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x で設定可能な読み取り整合性を使用して DynamoDB テーブルをクエリします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

    public QueryResponse queryWithConsistentReads(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final boolean useConsistentRead) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());

        // Create the query request
```

```
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .consistentRead(useConsistentRead)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query successful. Found {0} items",
 response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying with consistent reads", e);
            throw e;
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で設定可能な読み取り整合性を使用して DynamoDB テーブルをクエリします。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table with configurable read consistency
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
```

```
 * @param {boolean} useConsistentRead - Whether to use strongly consistent reads
 * @returns {Promise<Object>} - The query response
 */
async function queryWithConsistentRead(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  useConsistentRead = false
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: "#pk = :pkValue",
      ExpressionAttributeNames: {
        "#pk": partitionKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue }
      },
      ConsistentRead: useConsistentRead
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying with consistent read: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で強力な整合性のある読み込みのオプションを使用して DynamoDB テーブルをクエリします。

```
import time

import boto3
from boto3.dynamodb.conditions import Key


def query_with_consistent_read(
    table_name,
    partition_key_name,
    partition_key_value,
    sort_key_name=None,
    sort_key_value=None,
    consistent_read=True,
):
    """
    Query a DynamoDB table with the option for strongly consistent reads.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str, optional): The name of the sort key attribute.
        sort_key_value (str, optional): The value of the sort key to query.
        consistent_read (bool, optional): Whether to use strongly consistent
 reads. Defaults to True.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the key condition expression
    key_condition = Key(partition_key_name).eq(partition_key_value)

    if sort_key_name and sort_key_value:
```

```
        key_condition = key_condition & Key(sort_key_name).eq(sort_key_value)

    # Perform the query with the consistent read option
    response = table.query(KeyConditionExpression=key_condition,
ConsistentRead=consistent_read)

    return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で PartiQL SELECT ステートメントを使用して DynamoDB データをクエリする

次のコード例は、PartiQL SELECT ステートメントを使用してデータをクエリする方法を示しています。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で PartiQL SELECT ステートメントを使用して DynamoDB テーブルから項目をクエリします。

```
/**
 * This example demonstrates how to query items from a DynamoDB table using
 PartiQL.
 * It shows different ways to select data with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
```

```
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Select all items from a DynamoDB table using PartiQL.
 * Note: This should be used with caution on large tables.
 *
 * @param tableName - The name of the DynamoDB table
 * @returns The response from the ExecuteStatementCommand
 */
export const selectAllItems = async (tableName: string) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}"`,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select an item by its primary key using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemByPartitionKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);
```

```
  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${partitionKeyName} = ?`,
    Parameters: [partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving item:", err);
    throw err;
  }
};

/**
 * Select an item by its composite key (partition key + sort key) using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemByCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${partitionKeyName} = ? AND
 ${sortKeyName} = ?`,
    Parameters: [partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item retrieved successfully");
```

```
    return data;
  } catch (err) {
    console.error("Error retrieving item:", err);
    throw err;
  }
};

/**
 * Select items using a filter condition with PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param filterAttribute - The attribute to filter on
 * @param filterValue - The value to filter by
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemsWithFilter = async (
  tableName: string,
  filterAttribute: string,
  filterValue: string | number
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${filterAttribute} = ?`,
    Parameters: [filterValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select items using a begins_with function for prefix matching.
 * This is useful for querying hierarchical data.
 *
 * @param tableName - The name of the DynamoDB table
 * @param attributeName - The attribute to check for prefix
```

```
 * @param prefix - The prefix to match
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemsByPrefix = async (
  tableName: string,
  attributeName: string,
  prefix: string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE
 begins_with(${attributeName}, ?)`,
    Parameters: [prefix],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Select items using a between condition for range queries.
 *
 * @param tableName - The name of the DynamoDB table
 * @param attributeName - The attribute to check for range
 * @param startValue - The start value of the range
 * @param endValue - The end value of the range
 * @returns The response from the ExecuteStatementCommand
 */
export const selectItemsByRange = async (
  tableName: string,
  attributeName: string,
  startValue: number | string,
  endValue: number | string
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);
```

```
  const params = {
    Statement: `SELECT * FROM "${tableName}" WHERE ${attributeName} BETWEEN ?
 AND ?`,
    Parameters: [startValue, endValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Items retrieved successfully");
    return data;
  } catch (err) {
    console.error("Error retrieving items:", err);
    throw err;
  }
};

/**
 * Example usage showing how to select items with different index types
 */
export const selectExamples = async () => {
  // Select all items from a table (use with caution on large tables)
  await selectAllItems("UsersTable");

  // Select by partition key (simple primary key)
  await selectItemByPartitionKey("UsersTable", "userId", "user123");

  // Select by composite key (partition key + sort key)
  await selectItemByCompositeKey("OrdersTable", "orderId", "order456",
 "productId", "prod789");

  // Select with a filter condition (can use any attribute)
  await selectItemsWithFilter("UsersTable", "userType", "premium");

  // Select items with a prefix (useful for hierarchical data)
  await selectItemsByPrefix("ProductsTable", "category", "electronics");

  // Select items within a range (useful for numeric or date ranges)
  await selectItemsByRange("OrdersTable", "orderDate", "2023-01-01",
 "2023-12-31");
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - BatchExecuteStatement
  - ExecuteStatement

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して、DynamoDB テーブルの TTL 項目に対してクエリを実行する

以下のコード例は、TTL 項目のクエリの開始方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.util.Map;
import java.util.Optional;

        final QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .filterExpression(FILTER_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();
```

```
        try (DynamoDbClient ddb = dynamoDbClient != null
            ? dynamoDbClient
            : DynamoDbClient.builder().region(region).build()) {
            final QueryResponse response = ddb.query(request);
            System.out.println("Query successful. Found " + response.count() + "
 items that have not expired yet.");

            // Print each item
            response.items().forEach(item -> {
                System.out.println("Item: " + item);
            });

            return 0;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const queryFiltered = async (tableName, primaryKey, region = 'us-east-1')
 => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });
```

```
    const currentTime = Math.floor(Date.now() / 1000);

    const params = {
        TableName: tableName,
        KeyConditionExpression: "#pk = :pk",
        FilterExpression: "#ea > :ea",
        ExpressionAttributeNames: {
            "#pk": "primaryKey",
            "#ea": "expireAt"
        },
        ExpressionAttributeValues: marshall({
            ":pk": primaryKey,
            ":ea": currentTime
        })
    };

    try {
        const { Items } = await client.send(new QueryCommand(params));
        Items.forEach(item => {
            console.log(unmarshall(item))
        });
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

// Example usage (commented out for testing)
// queryFiltered('your-table-name', 'your-partition-key-value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用しフィルタリングされた式をクエリして、DynamoDB テーブル内の TTL 項目を収集します。

```
from datetime import datetime

import boto3


def query_dynamodb_items(table_name, partition_key):
    """

    :param table_name: Name of the DynamoDB table
    :param partition_key:
    :return:
    """
    try:
        # Initialize a DynamoDB resource
        dynamodb = boto3.resource("dynamodb", region_name="us-east-1")

        # Specify your table
        table = dynamodb.Table(table_name)

        # Get the current time in epoch format
        current_time = int(datetime.now().timestamp())

        # Perform the query operation with a filter expression to exclude expired
 items
        # response = table.query(
        #
 KeyConditionExpression=boto3.dynamodb.conditions.Key('partitionKey').eq(partition_key),
        #
 FilterExpression=boto3.dynamodb.conditions.Attr('expireAt').gt(current_time)
        # )
        response = table.query(

 KeyConditionExpression=dynamodb.conditions.Key("partitionKey").eq(partition_key),

 FilterExpression=dynamodb.conditions.Attr("expireAt").gt(current_time),
        )

        # Print the items that are not expired
        for item in response["Items"]:
            print(item)

    except Exception as e:
        print(f"Error querying items: {e}")
```

```
# Call the function with your values
query_dynamodb_items("Music", "your-partition-key-value")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で日付と時刻のパターンを使用して DynamoDB テーブルをクエリする

以下のコード例は、日付と時刻のパターンを使用してテーブルをクエリする方法を示しています。

- DynamoDB に日付/時刻値を保存してクエリします。
- ソートキーを使用して日付範囲クエリを実装します。
- 有効なクエリを行うために日付文字列をフォーマットします。

Java

SDK for Java 2.x

AWS SDK for Java 2.x でソートキーの日付範囲を使用してクエリを実行します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;

    public QueryResponse queryWithDateRange(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDate startDate,
        final LocalDate endDate) {

        // Focus on query logic, assuming parameters are valid
        if (startDate == null || endDate == null) {
            throw new IllegalArgumentException("Start date and end date cannot be
 null");
        }

        if (endDate.isBefore(startDate)) {
            throw new IllegalArgumentException("End date must be after start
 date");
        }

        // Format dates as ISO strings for DynamoDB (using just the date part)
        final String formattedStartDate = startDate.toString();
        final String formattedEndDate = endDate.toString();

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_SK, dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
 HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_START_DATE,
            AttributeValue.builder().s(formattedStartDate).build());
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_END_DATE,
            AttributeValue.builder().s(formattedEndDate).build());
```

```
        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            LOGGER.log(Level.INFO, "Query by date range successful. Found {0}
 items", response.count());
            return response;
        } catch (ResourceNotFoundException e) {
            LOGGER.log(Level.SEVERE, "Table not found: {0}", tableName);
            throw e;
        } catch (DynamoDbException e) {
            LOGGER.log(Level.SEVERE, "Error querying by date range: {0}",
 e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x で日時変数を使用してクエリを実行します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTime(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
```

```
        final String dateKeyName,
        final String startDate,
        final String endDate) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
partitionKeyValue);
        CodeSampleUtils.validateDateRangeParameters(dateKeyName, startDate,
endDate);
        CodeSampleUtils.validateDateFormat("Start date", startDate);
        CodeSampleUtils.validateDateFormat("End date", endDate);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
partitionKeyName);
        expressionAttributeNames.put("#dateKey", dateKeyName);

        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            ":startDate", AttributeValue.builder().s(startDate).build());
        expressionAttributeValues.put(
            ":endDate", AttributeValue.builder().s(endDate).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query successful. Found " + response.count() + "
items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
```

```
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with date range: " +
 e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x で Unix エポックタイムスタンプの日付範囲内でクエリを実行します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTimeEpoch(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final long startEpoch,
        final long endEpoch) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Date key name", dateKeyName);
        CodeSampleUtils.validateEpochTimestamp("Start epoch", startEpoch);
        CodeSampleUtils.validateEpochTimestamp("End epoch", endEpoch);

        // Create expression attribute names for the column names
        final Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put(EXPRESSION_ATTRIBUTE_NAME_PK,
 partitionKeyName);
        expressionAttributeNames.put("#dateKey", dateKeyName);
```

```
        // Create expression attribute values for the column values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            EXPRESSION_ATTRIBUTE_VALUE_PK,
            AttributeValue.builder().s(partitionKeyValue).build());
        expressionAttributeValues.put(
            ":startDate",
AttributeValue.builder().n(String.valueOf(startEpoch)).build());
        expressionAttributeValues.put(
            ":endDate",
AttributeValue.builder().n(String.valueOf(endEpoch)).build());

        // Create the query request
        final QueryRequest queryRequest = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression(KEY_CONDITION_EXPRESSION)
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final QueryResponse response = dynamoDbClient.query(queryRequest);
            System.out.println("Query successful. Found " + response.count() + "
items");
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format("Error: The Amazon DynamoDB table \"%s\" can't be
found.\n", tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println("Error querying with epoch timestamps: " +
e.getMessage());
            throw e;
        }
    }
```

AWS SDK for Java 2.x で LocalDateTime オブジェクトを使用して日付範囲内でクエリを実行します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

```
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.HashMap;
import java.util.Map;

    public QueryResponse queryWithDateTimeLocalDateTime(
        final String tableName,
        final String partitionKeyName,
        final String partitionKeyValue,
        final String dateKeyName,
        final LocalDateTime startDateTime,
        final LocalDateTime endDateTime) {

        CodeSampleUtils.validateTableParameters(tableName, partitionKeyName,
 partitionKeyValue);
        CodeSampleUtils.validateStringParameter("Date key name", dateKeyName);
        if (startDateTime == null || endDateTime == null) {
            throw new IllegalArgumentException("Start and end LocalDateTime must
 not be null");
        }

        // Convert LocalDateTime to ISO-8601 strings in UTC with the correct
 format
        final String startDate =
 startDateTime.atZone(ZoneOffset.UTC).format(DATE_TIME_FORMATTER);
        final String endDate =
 endDateTime.atZone(ZoneOffset.UTC).format(DATE_TIME_FORMATTER);

        return queryWithDateTime(tableName, partitionKeyName, partitionKeyValue,
 dateKeyName, startDate, endDate);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「Query」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript でソートキーの日付範囲を使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range on the sort
 key
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} sortKeyName - The name of the sort key (must be a date/time
 attribute)
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
async function queryByDateRangeOnSortKey(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  sortKeyName,
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: '#pk = :pkValue AND #sk BETWEEN :startDate
 AND :endDate',
```

```
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#sk": sortKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range on sort key: ${error}`);
    throw error;
  }
}
```

AWS SDK for JavaScript で日時変数を使用してクエリを実行します。

```
const { DynamoDBClient, QueryCommand } = require("@aws-sdk/client-dynamodb");

/**
 * Queries a DynamoDB table for items within a specific date range
 *
 * @param {Object} config - AWS SDK configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key
 * @param {string} partitionKeyValue - The value of the partition key
 * @param {string} dateKeyName - The name of the date attribute to filter on
 * @param {Date} startDate - The start date for the range query
 * @param {Date} endDate - The end date for the range query
 * @returns {Promise<Object>} - The query response
 */
async function queryByDateRange(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  dateKeyName,
```

```
  startDate,
  endDate
) {
  try {
    // Create DynamoDB client
    const client = new DynamoDBClient(config);

    // Format dates as ISO strings for DynamoDB
    const formattedStartDate = startDate.toISOString();
    const formattedEndDate = endDate.toISOString();

    // Construct the query input
    const input = {
      TableName: tableName,
      KeyConditionExpression: `#pk = :pkValue AND #dateAttr BETWEEN :startDate
 AND :endDate`,
      ExpressionAttributeNames: {
        "#pk": partitionKeyName,
        "#dateAttr": dateKeyName
      },
      ExpressionAttributeValues: {
        ":pkValue": { S: partitionKeyValue },
        ":startDate": { S: formattedStartDate },
        ":endDate": { S: formattedEndDate }
      }
    };

    // Execute the query
    const command = new QueryCommand(input);
    return await client.send(command);
  } catch (error) {
    console.error(`Error querying by date range: ${error}`);
    throw error;
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「Query」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) でソートキーの日付範囲を使用してクエリを実行します。

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_date_range(
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        start_date (datetime): The start date for the query range.
        end_date (datetime): The end date for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Format the date values as ISO 8601 strings
    # DynamoDB works well with ISO format for date values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key using BETWEEN operator
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
```

```
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response


def query_with_date_range_by_month(
    table_name, partition_key_name, partition_key_value, sort_key_name, year,
 month
):
    """
    Query a DynamoDB table for a specific month's data.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date
 values).
        year (int): The year to query.
        month (int): The month to query (1-12).

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Calculate the start and end dates for the specified month
    if month == 12:
        next_year = year + 1
        next_month = 1
    else:
        next_year = year
        next_month = month + 1
```

```
    start_date = datetime(year, month, 1)
    end_date = datetime(next_year, next_month, 1) - timedelta(microseconds=1)

    # Format the date values as ISO 8601 strings
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(KeyConditionExpression=key_condition)

    return response
```

AWS SDK for Python (Boto3) で日時変数を使用してクエリを実行します。

```
from datetime import datetime, timedelta

import boto3
from boto3.dynamodb.conditions import Key


def query_with_datetime(
    table_name, partition_key_name, partition_key_value, sort_key_name,
 start_date, end_date
):
    """
    Query a DynamoDB table with a date range filter on the sort key.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        sort_key_name (str): The name of the sort key attribute (containing date/
time values).
        start_date (datetime): The start date/time for the query range.
```

```
        end_date (datetime): The end date/time for the query range.

    Returns:
        dict: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Format the date/time values as ISO 8601 strings
    # DynamoDB works well with ISO format for date/time values
    start_date_str = start_date.isoformat()
    end_date_str = end_date.isoformat()

    # Perform the query with a date range on the sort key
    key_condition = Key(partition_key_name).eq(partition_key_value) &
 Key(sort_key_name).between(
        start_date_str, end_date_str
    )

    response = table.query(
        KeyConditionExpression=key_condition,
        ExpressionAttributeValues={
            ":pk_val": partition_key_value,
            ":start_date": start_date_str,
            ":end_date": end_date_str,
        },
    )

    return response



def example_usage():
    """Example of how to use the query_with_datetime function."""
    # Example parameters
    table_name = "Events"
    partition_key_name = "EventType"
    partition_key_value = "UserLogin"
    sort_key_name = "Timestamp"

    # Create date/time variables for the query
    end_date = datetime.now()
```

```
    start_date = end_date - timedelta(days=7)  # Query events from the last 7
days

    print(f"Querying events from {start_date.isoformat()} to
{end_date.isoformat()}")

    # Execute the query
    response = query_with_datetime(
        table_name, partition_key_name, partition_key_value, sort_key_name,
start_date, end_date
    )

    # Process the results
    items = response.get("Items", [])
    print(f"Found {len(items)} items")

    for item in items:
        print(f"Event: {item}")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Query」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して、EXIF とその他の画像情報を保存する

次のコード例は、以下の操作方法を示しています。

- JPG、JPEG、または PNG ファイルから EXIF 情報を取得します。
- Amazon S3 バケットにイメージファイルをアップロードします。
- Amazon Rekognition を使用して、ファイル内の 3 つの上位属性 (ラベル) を特定します。
- EXIF およびラベル情報を、リージョンの Amazon DynamoDB テーブルに追加します。

Rust

SDK for Rust

JPG、JPEG、または PNG ファイルから EXIF 情報を取得し、イメージファイルを Amazon S3 バケットにアップロードし、Amazon Rekognition を使用してファイル内の 3 つの上位属性 (Amazon Rekognition のラベル) を特定し、リージョンの Amazon DynamoDB テーブルに EXIF およびラベル情報を追加します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Amazon Rekognition
- Amazon S3

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB の属性ベースのアクセス制御を設定する

次のコード例は、DynamoDB の属性ベースのアクセス制御 (ABAC) を実装する方法を示しています。

- ABAC の IAM ポリシーを作成します。
- さまざまな部門のタグを含むテーブルを作成します。
- タグに基づいてテーブルを一覧表示およびフィルタリングします。

Bash

AWS CLIBash スクリプトを使用した

ABAC の IAM ポリシーを作成します。

```
# Step 1: Create a policy document for ABAC
```

```
cat > abac-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Department": "${aws:PrincipalTag/Department}",
          "aws:ResourceTag/Environment": "Development"
        }
      }
    }
  ]
}
EOF

# Step 2: Create the IAM policy
aws iam create-policy \
    --policy-name DynamoDBDepartmentBasedAccess \
    --policy-document file://abac-policy.json
```

さまざまな部門のタグを含むテーブルを作成します。

```
# Create a DynamoDB table with tags for ABAC
aws dynamodb create-table \
    --table-name FinanceData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Department,Value=Finance \
        Key=Environment,Value=Development

# Create another table with different tags
aws dynamodb create-table \
    --table-name MarketingData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Department,Value=Marketing \
        Key=Environment,Value=Production
```

タグに基づいてテーブルを一覧表示およびフィルタリングします。

```
# List all DynamoDB tables
echo "Listing all tables:"
aws dynamodb list-tables

# Get ARNs for all tables
echo -e "\nGetting ARNs for all tables:"
TABLE_ARNS=$(aws dynamodb list-tables --query "TableNames[*]" --output text |
 xargs -I {} aws dynamodb describe-table --table-name {} --query "Table.TableArn"
 --output text)

# For each table ARN, list its tags
echo -e "\nListing tags for each table:"
```

```
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    echo -e "\nTags for table: $TABLE_NAME"
    aws dynamodb list-tags-of-resource --resource-arn $ARN
done

# Example: Find tables with a specific tag
echo -e "\nFinding tables with Environment=Production tag:"
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    TAGS=$(aws dynamodb list-tags-of-resource --resource-arn $ARN --query "Tags[?
Key=='Environment' && Value=='Production']" --output text)
    if [ ! -z "$TAGS" ]; then
        echo "Table with Production tag: $TABLE_NAME"
    fi
done
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreatePolicy
  - CreateTable
  - ListTables

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した DynamoDB の式順序の更新を理解する

次のコード例は、式順序の更新を理解する方法を示しています。

- DynamoDB が更新式を処理する方法について説明します。
- 更新式のオペレーションの順序を理解します。
- 式の評価を理解することで、予期しない結果を回避できます。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して式順序の更新をデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Demonstrates the effect of update expression order.
     *
     * <p>This method shows how the order of operations in an update expression
     * affects the result of the update.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return Map containing the results of different update orders
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, Object> demonstrateUpdateOrder(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
 AttributeValue> key) {

        Map<String, Object> results = new HashMap<>();

        try {
            // Initialize the item with a counter
            UpdateItemRequest initRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :zero, OldCounter = :zero")
                .expressionAttributeValues(
                    Map.of(":zero", AttributeValue.builder().n("0").build()))
```

```
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            dynamoDbClient.updateItem(initRequest);

            // Example 1: SET first, then ADD
            UpdateItemRequest setFirstRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :value ADD
OldCounter :increment")
                .expressionAttributeValues(Map.of(
                    ":value", AttributeValue.builder().n("10").build(),
                    ":increment", AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse setFirstResponse =
dynamoDbClient.updateItem(setFirstRequest);
            results.put("setFirstResponse", setFirstResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 2: ADD first, then SET
            UpdateItemRequest addFirstRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("ADD Counter :increment SET OldCounter
= :value")
                .expressionAttributeValues(Map.of(
                    ":value", AttributeValue.builder().n("10").build(),
                    ":increment", AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse addFirstResponse =
dynamoDbClient.updateItem(addFirstRequest);
            results.put("addFirstResponse", addFirstResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 3: SET with multiple attributes
```

```
            UpdateItemRequest multiSetRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = :value, OldCounter = Counter")
                .expressionAttributeValues(
                    Map.of(":value", AttributeValue.builder().n("10").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse multiSetResponse =
dynamoDbClient.updateItem(multiSetRequest);
            results.put("multiSetResponse", multiSetResponse);

            // Reset the item
            dynamoDbClient.updateItem(initRequest);

            // Example 4: SET with expression using the same attribute
            UpdateItemRequest selfReferenceRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET Counter = Counter + :increment, OldCounter
= Counter")
                .expressionAttributeValues(
                    Map.of(":increment",
AttributeValue.builder().n("5").build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse selfReferenceResponse =
dynamoDbClient.updateItem(selfReferenceRequest);
            results.put("selfReferenceResponse", selfReferenceResponse);

            results.put("success", true);

        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }

        return results;
    }

    /**
     * Updates an item with SET first, then REMOVE.
```

```
     *
     * <p>This method demonstrates updating an item with SET operation first,
     * followed by a REMOVE operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributeToSet The attribute to set
     * @param setValue The value to set
     * @param attributeToRemove The attribute to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithSetFirst(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String attributeToSet,
        AttributeValue setValue,
        String attributeToRemove) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #setAttr = :setValue REMOVE #removeAttr")
            .expressionAttributeNames(Map.of(
                "#setAttr", attributeToSet,
                "#removeAttr", attributeToRemove))
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with SET first: " +
e.getMessage())
                .cause(e)
                .build();
        }
    }
```

```
    /**
     * Updates an item with REMOVE first, then SET.
     *
     * <p>This method demonstrates updating an item with REMOVE operation first,
     * followed by a SET operation.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributeToSet The attribute to set
     * @param setValue The value to set
     * @param attributeToRemove The attribute to remove
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithRemoveFirst(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String attributeToSet,
        AttributeValue setValue,
        String attributeToRemove) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("REMOVE #removeAttr SET #setAttr = :setValue")
            .expressionAttributeNames(Map.of(
                "#setAttr", attributeToSet,
                "#removeAttr", attributeToRemove))
            .expressionAttributeValues(Map.of(":setValue", setValue))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with REMOVE first: " +
e.getMessage())
                .cause(e)
```

```
                .build();
        }
    }

    /**
     * Updates an item with all operation types in a specific order.
     *
     * <p>This method demonstrates using all operation types (SET, REMOVE, ADD,
DELETE)
     * in a specific order in a single update expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateWithAllOperationTypes(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #stringAttr = :stringVal, #mapAttr.#nestedAttr
= :nestedVal " + "REMOVE #oldAttr "
                + "ADD #counterAttr :increment "
                + "DELETE #stringSetAttr :stringSetVal")
            .expressionAttributeNames(Map.of(
                "#stringAttr", "StringAttribute",
                "#mapAttr", "MapAttribute",
                "#nestedAttr", "NestedAttribute",
                "#oldAttr", "OldAttribute",
                "#counterAttr", "CounterAttribute",
                "#stringSetAttr", "StringSetAttribute"))
            .expressionAttributeValues(Map.of(
                ":stringVal", AttributeValue.builder().s("New Value").build(),
                ":nestedVal", AttributeValue.builder().s("Nested Value").build(),
                ":increment", AttributeValue.builder().n("1").build(),
                ":stringSetVal", AttributeValue.builder().ss("Value1").build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();
```

```
        // Perform the update operation
        try {
            return dynamoDbClient.updateItem(request);
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to update item with all operation types: " +
e.getMessage())
                .cause(e)
                .build();
        }
    }

    /**
     * Gets the current state of an item.
     *
     * <p>Helper method to retrieve the current state of an item.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @return The item or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Map<String, AttributeValue> getItem(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        // Define the get parameters
        GetItemRequest request =
            GetItemRequest.builder().tableName(tableName).key(key).build();

        // Perform the get operation
        try {
            GetItemResponse response = dynamoDbClient.getItem(request);

            // Return the item if it exists, otherwise null
            return response.item();
        } catch (DynamoDbException e) {
            throw DynamoDbException.builder()
                .message("Failed to get item: " + e.getMessage())
                .cause(e)
                .build();
        }
    }
```

AWS SDK for Java 2.x を使用した式順序の更新の使用例。

```
   public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
       // Example key
       Map<String, AttributeValue> key = new HashMap<>();
       key.put("ProductId", AttributeValue.builder().s("P12345").build());

       System.out.println("Demonstrating update expression order in DynamoDB");

       try {
           // Example 1: Demonstrate update order effects
           System.out.println("\nExample 1: Demonstrating update order
effects");
           Map<String, Object> orderResults =
demonstrateUpdateOrder(dynamoDbClient, tableName, key);

           if ((boolean) orderResults.get("success")) {
               System.out.println("SET first, then ADD:");
               System.out.println("  " + orderResults.get("setFirstResponse"));

               System.out.println("ADD first, then SET:");
               System.out.println("  " + orderResults.get("addFirstResponse"));

               System.out.println("SET with multiple attributes:");
               System.out.println("  " + orderResults.get("multiSetResponse"));

               System.out.println("SET with self-reference:");
               System.out.println("  " +
orderResults.get("selfReferenceResponse"));
           } else {
               System.out.println("Error: " + orderResults.get("error"));
           }

           // Example 2: Update with SET first, then REMOVE
           System.out.println("\nExample 2: Update with SET first, then
REMOVE");
           UpdateItemResponse setFirstResponse = updateWithSetFirst(
               dynamoDbClient,
               tableName,
               key,
```

```
                "Status",
                AttributeValue.builder().s("Active").build(),
                "OldStatus");

            System.out.println("Updated attributes: " +
setFirstResponse.attributes());

            // Example 3: Update with REMOVE first, then SET
            System.out.println("\nExample 3: Update with REMOVE first, then
SET");
            UpdateItemResponse removeFirstResponse = updateWithRemoveFirst(
                dynamoDbClient,
                tableName,
                key,
                "Status",
                AttributeValue.builder().s("Inactive").build(),
                "OldStatus");

            System.out.println("Updated attributes: " +
removeFirstResponse.attributes());

            // Example 4: Update with all operation types
            System.out.println("\nExample 4: Update with all operation types");
            UpdateItemResponse allOpsResponse =
updateWithAllOperationTypes(dynamoDbClient, tableName, key);

            System.out.println("Updated attributes: " +
allOpsResponse.attributes());

            // Example 5: Get the current state of the item
            System.out.println("\nExample 5: Current state of the item");
            Map<String, AttributeValue> item = getItem(dynamoDbClient, tableName,
key);

            if (item != null) {
                System.out.println("Item: " + item);
            } else {
                System.out.println("Item not found");
            }

            // Explain update expression order
            System.out.println("\nKey points about update expression order in
DynamoDB:");
```

```
            System.out.println("1. Update expressions are processed in this
 order: SET, REMOVE, ADD, DELETE");
            System.out.println("2. Within each clause, operations are processed
 from left to right");
            System.out.println("3. SET operations use the item state before any
 updates in the expression");
            System.out.println("4. When an attribute is referenced multiple
 times, the first operation wins");
            System.out.println("5. To reference a new value, split the update
 into multiple operations");
            System.out.println("6. The order of clauses in the expression doesn't
 change the evaluation order");
            System.out.println("7. For complex updates, consider using multiple
 separate update operations");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して式順序の更新をデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update an item with multiple actions in a single update expression.
 *
```

```
 * This function demonstrates how to use multiple actions in a single update
 expression
 * and how DynamoDB processes these actions.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
 * @param {string} updateExpression - The update expression with multiple actions
 * @param {Object} [expressionAttributeNames] - Expression attribute name
 placeholders
 * @param {Object} [expressionAttributeValues] - Expression attribute value
 placeholders
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleActions(
  config,
  tableName,
  key,
  updateExpression,
  expressionAttributeNames,
  expressionAttributeValues
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Prepare the update parameters
  const updateParams = {
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ReturnValues: "UPDATED_NEW"
  };

  // Add expression attribute names if provided
  if (expressionAttributeNames) {
    updateParams.ExpressionAttributeNames = expressionAttributeNames;
  }

  // Add expression attribute values if provided
  if (expressionAttributeValues) {
    updateParams.ExpressionAttributeValues = expressionAttributeValues;
  }
```

```
  // Execute the update
  const response = await docClient.send(new UpdateCommand(updateParams));

  return response;
}

/**
 * Demonstrate that variables hold copies of existing values before
 modifications.
 *
 * This function creates an item with initial values, then updates it with an
 expression
 * that uses the values of attributes before they are modified in the same
 expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
async function demonstrateValueCopying(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Step 1: Create an item with initial values
  const initialItem = { ...key, a: 1, b: 2, c: 3 };

  await docClient.send(new PutCommand({
    TableName: tableName,
    Item: initialItem
  }));

  // Step 2: Get the item to verify initial state
  const responseBefore = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));
```

```
  const itemBefore = responseBefore.Item || {};

  // Step 3: Update the item with an expression that uses values before they are
 modified
  // This expression removes 'a', then sets 'b' to the value of 'a', and 'c' to
 the value of 'b'
  const updateResponse = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: "REMOVE a SET b = a, c = b",
    ReturnValues: "UPDATED_NEW"
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}

/**
 * Demonstrate the order in which different action types are processed.
 *
 * This function creates an item with initial values, then updates it with an
 expression
 * that includes multiple action types (SET, REMOVE, ADD, DELETE) to show the
 order
 * in which they are processed.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
```

```
async function demonstrateActionOrder(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Step 1: Create an item with initial values
  const initialItem = {
    ...key,
    counter: 10,
    set_attr: new Set(["A", "B", "C"]),
    to_remove: "This will be removed",
    to_modify: "Original value"
  };

  await docClient.send(new PutCommand({
    TableName: tableName,
    Item: initialItem
  }));

  // Step 2: Get the item to verify initial state
  const responseBefore = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemBefore = responseBefore.Item || {};

  // Step 3: Update the item with multiple action types
  // The actions will be processed in this order: REMOVE, SET, ADD, DELETE
  const updateResponse = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: "REMOVE to_remove SET to_modify = :new_value ADD
 counter :increment DELETE set_attr :elements",
    ExpressionAttributeValues: {
      ":new_value": "Updated value",
      ":increment": 5,
      ":elements": new Set(["B"])
    },
    ReturnValues: "UPDATED_NEW"
```

```
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}

/**
 * Update multiple attributes with a single SET action.
 *
 * This function demonstrates how to update multiple attributes in a single SET
 action,
 * which is more efficient than using multiple separate update operations.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
 * @param {Object} attributes - The attributes to update and their new values
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithMultipleSetActions(
  config,
  tableName,
  key,
  attributes
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression and expression attribute values
  let updateExpression = "SET ";
  const expressionAttributeValues = {};
```

```
  // Add each attribute to the update expression
  Object.entries(attributes).forEach(([attrName, attrValue], index) => {
    const valuePlaceholder = `:val${index}`;

    if (index > 0) {
      updateExpression += ", ";
    }
    updateExpression += `${attrName} = ${valuePlaceholder}`;

    expressionAttributeValues[valuePlaceholder] = attrValue;
  });

  // Execute the update
  const response = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  }));

  return response;
}

/**
 * Update an attribute with a value from another attribute or a default value.
 *
 * This function demonstrates how to use if_not_exists to conditionally copy a
 value
 * from one attribute to another, or use a default value if the source doesn't
 exist.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to update
 * @param {string} sourceAttribute - The attribute to copy the value from
 * @param {string} targetAttribute - The attribute to update
 * @param {any} defaultValue - The default value to use if the source attribute
 doesn't exist
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateWithConditionalValueCopying(
  config,
```

```
  tableName,
  key,
  sourceAttribute,
  targetAttribute,
  defaultValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Use if_not_exists to conditionally copy the value
  const response = await docClient.send(new UpdateCommand({
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${targetAttribute} =
 if_not_exists(${sourceAttribute}, :default)`,
    ExpressionAttributeValues: {
      ":default": defaultValue
    },
    ReturnValues: "UPDATED_NEW"
  }));

  return response;
}

/**
 * Demonstrate complex update expressions with multiple operations on the same
 attribute.
 *
 * This function shows how DynamoDB processes multiple operations on the same
 attribute
 * in a single update expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The primary key of the item to create and update
 * @returns {Promise<Object>} - A dictionary containing the results of the
 demonstration
 */
async function demonstrateMultipleOperationsOnSameAttribute(
  config,
  tableName,
  key
) {
```

```
// Initialize the DynamoDB client
const client = new DynamoDBClient(config);
const docClient = DynamoDBDocumentClient.from(client);

// Step 1: Create an item with initial values
const initialItem = {
  ...key,
  counter: 10,
  list_attr: [1, 2, 3],
  map_attr: {
    nested1: "value1",
    nested2: "value2"
  }
};

await docClient.send(new PutCommand({
  TableName: tableName,
  Item: initialItem
}));

// Step 2: Get the item to verify initial state
const responseBefore = await docClient.send(new GetCommand({
  TableName: tableName,
  Key: key
}));

const itemBefore = responseBefore.Item || {};

// Step 3: Update the item with multiple operations on the same attributes
const updateResponse = await docClient.send(new UpdateCommand({
  TableName: tableName,
  Key: key,
  UpdateExpression: `
    SET counter = counter + :inc1,
        counter = counter + :inc2,
        map_attr.nested1 = :new_val1,
        map_attr.nested3 = :new_val3,
        list_attr[0] = list_attr[1],
        list_attr[1] = list_attr[2]
  `,
  ExpressionAttributeValues: {
    ":inc1": 5,
    ":inc2": 3,
    ":new_val1": "updated_value1",
```

```
      ":new_val3": "new_value3"
    },
    ReturnValues: "UPDATED_NEW"
  }));

  // Step 4: Get the item to verify final state
  const responseAfter = await docClient.send(new GetCommand({
    TableName: tableName,
    Key: key
  }));

  const itemAfter = responseAfter.Item || {};

  // Return the results
  return {
    initialState: itemBefore,
    updateResponse: updateResponse,
    finalState: itemAfter
  };
}
```

AWS SDK for JavaScript を使用した式順序の更新の使用例。

```
/**
 * Example of how to use update expression order of operations in DynamoDB.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "OrderProcessing";

  console.log("Demonstrating update expression order of operations in DynamoDB");

  try {
    // Example 1: Demonstrating value copying in update expressions
    console.log("\nExample 1: Demonstrating value copying in update
 expressions");
    const results1 = await demonstrateValueCopying(
      config,
      tableName,
      { OrderId: "order123" }
    );
```

```
   console.log("Initial state:", JSON.stringify(results1.initialState, null,
2));
   console.log("Update response:", JSON.stringify(results1.updateResponse, null,
2));
   console.log("Final state:", JSON.stringify(results1.finalState, null, 2));

   console.log("\nExplanation:");
   console.log("1. The initial state had a=1, b=2, c=3");
   console.log("2. The update expression 'REMOVE a SET b = a, c = b' did the
following:");
   console.log("   - Copied the value of 'a' (which was 1) to be used for 'b'");
   console.log("   - Copied the value of 'b' (which was 2) to be used for 'c'");
   console.log("   - Removed the attribute 'a'");
   console.log("3. The final state has b=1, c=2, and 'a' is removed");
   console.log("4. This demonstrates that DynamoDB uses the values of attributes
as they were BEFORE any modifications");

   // Example 2: Demonstrating the order of different action types
   console.log("\nExample 2: Demonstrating the order of different action
types");
   const results2 = await demonstrateActionOrder(
     config,
     tableName,
     { OrderId: "order456" }
   );

   console.log("Initial state:", JSON.stringify(results2.initialState, null,
2));
   console.log("Update response:", JSON.stringify(results2.updateResponse, null,
2));
   console.log("Final state:", JSON.stringify(results2.finalState, null, 2));

   console.log("\nExplanation:");
   console.log("1. The update expression contained multiple action types:
REMOVE, SET, ADD, DELETE");
   console.log("2. DynamoDB processes these actions in this order: REMOVE, SET,
ADD, DELETE");
   console.log("3. First, 'to_remove' was removed");
   console.log("4. Then, 'to_modify' was set to a new value");
   console.log("5. Next, 'counter' was incremented by 5");
   console.log("6. Finally, 'B' was removed from the set attribute");

   // Example 3: Updating multiple attributes in a single SET action
```

```
    console.log("\nExample 3: Updating multiple attributes in a single SET
action");
    const response3 = await updateWithMultipleSetActions(
      config,
      tableName,
      { OrderId: "order789" },
      {
        Status: "Shipped",
        ShippingDate: "2025-05-28",
        TrackingNumber: "1Z999AA10123456784"
      }
    );

    console.log("Multiple attributes updated successfully:",
JSON.stringify(response3.Attributes, null, 2));

    // Example 4: Conditional value copying with if_not_exists
    console.log("\nExample 4: Conditional value copying with if_not_exists");
    const response4 = await updateWithConditionalValueCopying(
      config,
      tableName,
      { OrderId: "order101" },
      "PreferredShippingMethod",
      "ShippingMethod",
      "Standard"
    );

    console.log("Conditional value copying result:",
JSON.stringify(response4.Attributes, null, 2));

    // Example 5: Multiple operations on the same attribute
    console.log("\nExample 5: Multiple operations on the same attribute");
    const results5 = await demonstrateMultipleOperationsOnSameAttribute(
      config,
      tableName,
      { OrderId: "order202" }
    );

    console.log("Initial state:", JSON.stringify(results5.initialState, null,
2));
    console.log("Update response:", JSON.stringify(results5.updateResponse, null,
2));
    console.log("Final state:", JSON.stringify(results5.finalState, null, 2));
```

```
    console.log("\nExplanation:");
    console.log("1. The counter was incremented twice (first by 5, then by 3) for
 a total of +8");
    console.log("2. The map attribute had one value updated and a new nested
 attribute added");
    console.log("3. The list attribute had values shifted (value at index 1 moved
 to index 0, value at index 2 moved to index 1)");
    console.log("4. All operations within the SET action are processed from left
 to right");

    // Key points about update expression order of operations
    console.log("\nKey Points About Update Expression Order of Operations:");
    console.log("1. Variables in expressions hold copies of attribute values as
 they existed BEFORE any modifications");
    console.log("2. Multiple actions in an update expression are processed in
 this order: REMOVE, SET, ADD, DELETE");
    console.log("3. Within each action type, operations are processed from left
 to right");
    console.log("4. You can reference the same attribute multiple times in an
 expression");
    console.log("5. You can use if_not_exists() to conditionally set values based
 on attribute existence");
    console.log("6. Using a single update expression with multiple actions is
 more efficient than multiple separate updates");
    console.log("7. The update expression is atomic - either all actions succeed
 or none do");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して式順序の更新をデモンストレーションします。

```
import boto3
```

```
import json
from typing import Any, Dict, Optional


def update_with_multiple_actions(
    table_name: str,
    key: Dict[str, Any],
    update_expression: str,
    expression_attribute_names: Optional[Dict[str, str]] = None,
    expression_attribute_values: Optional[Dict[str, Any]] = None,
) -> Dict[str, Any]:
    """
    Update an item with multiple actions in a single update expression.

    This function demonstrates how to use multiple actions in a single update
 expression
    and how DynamoDB processes these actions.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        update_expression (str): The update expression with multiple actions.
        expression_attribute_names (Optional[Dict[str, str]]): Expression
 attribute name placeholders.
        expression_attribute_values (Optional[Dict[str, Any]]): Expression
 attribute value placeholders.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Prepare the update parameters
    update_params = {
        "Key": key,
        "UpdateExpression": update_expression,
        "ReturnValues": "UPDATED_NEW",
    }

    # Add expression attribute names if provided
    if expression_attribute_names:
```

```
        update_params["ExpressionAttributeNames"] = expression_attribute_names

    # Add expression attribute values if provided
    if expression_attribute_values:
        update_params["ExpressionAttributeValues"] = expression_attribute_values

    # Execute the update
    response = table.update_item(**update_params)

    return response


def demonstrate_value_copying(table_name: str, key: Dict[str, Any]) -> Dict[str,
 Any]:
    """
    Demonstrate that variables hold copies of existing values before
 modifications.

    This function creates an item with initial values, then updates it with an
 expression
    that uses the values of attributes before they are modified in the same
 expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to create and update.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Step 1: Create an item with initial values
    initial_item = key.copy()
    initial_item.update({"a": 1, "b": 2, "c": 3})

    table.put_item(Item=initial_item)

    # Step 2: Get the item to verify initial state
    response_before = table.get_item(Key=key)
    item_before = response_before.get("Item", {})
```

```
    # Step 3: Update the item with an expression that uses values before they are
 modified
    # This expression removes 'a', then sets 'b' to the value of 'a', and 'c' to
 the value of 'b'
    update_response = table.update_item(
        Key=key, UpdateExpression="REMOVE a SET b = a, c = b",
 ReturnValues="UPDATED_NEW"
    )

    # Step 4: Get the item to verify final state
    response_after = table.get_item(Key=key)
    item_after = response_after.get("Item", {})

    # Return the results
    return {
        "initial_state": item_before,
        "update_response": update_response,
        "final_state": item_after,
    }


def demonstrate_action_order(table_name: str, key: Dict[str, Any]) -> Dict[str,
 Any]:
    """
    Demonstrate the order in which different action types are processed.

    This function creates an item with initial values, then updates it with an
 expression
    that includes multiple action types (SET, REMOVE, ADD, DELETE) to show the
 order
    in which they are processed.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to create and update.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Step 1: Create an item with initial values
```

```
    initial_item = key.copy()
    initial_item.update(
        {
            "counter": 10,
            "set_attr": set(["A", "B", "C"]),
            "to_remove": "This will be removed",
            "to_modify": "Original value",
        }
    )

    table.put_item(Item=initial_item)

    # Step 2: Get the item to verify initial state
    response_before = table.get_item(Key=key)
    item_before = response_before.get("Item", {})

    # Step 3: Update the item with multiple action types
    # The actions will be processed in this order: REMOVE, SET, ADD, DELETE
    update_response = table.update_item(
        Key=key,
        UpdateExpression="REMOVE to_remove SET to_modify = :new_value ADD
 counter :increment DELETE set_attr :elements",
        ExpressionAttributeValues={
            ":new_value": "Updated value",
            ":increment": 5,
            ":elements": set(["B"]),
        },
        ReturnValues="UPDATED_NEW",
    )

    # Step 4: Get the item to verify final state
    response_after = table.get_item(Key=key)
    item_after = response_after.get("Item", {})

    # Return the results
    return {
        "initial_state": item_before,
        "update_response": update_response,
        "final_state": item_after,
    }


def update_with_multiple_set_actions(
    table_name: str, key: Dict[str, Any], attributes: Dict[str, Any]
```

```
) -> Dict[str, Any]:
    """
    Update multiple attributes with a single SET action.

    This function demonstrates how to update multiple attributes in a single SET
 action,
    which is more efficient than using multiple separate update operations.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        attributes (Dict[str, Any]): The attributes to update and their new
 values.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Build the update expression and expression attribute values
    update_expression = "SET "
    expression_attribute_values = {}

    # Add each attribute to the update expression
    for i, (attr_name, attr_value) in enumerate(attributes.items()):
        value_placeholder = f":val{i}"

        if i > 0:
            update_expression += ", "
        update_expression += f"{attr_name} = {value_placeholder}"

        expression_attribute_values[value_placeholder] = attr_value

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeValues=expression_attribute_values,
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def update_with_conditional_value_copying(
    table_name: str,
    key: Dict[str, Any],
    source_attribute: str,
    target_attribute: str,
    default_value: Any,
) -> Dict[str, Any]:
    """
    Update an attribute with a value from another attribute or a default value.

    This function demonstrates how to use if_not_exists to conditionally copy a
 value
    from one attribute to another, or use a default value if the source doesn't
 exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        source_attribute (str): The attribute to copy the value from.
        target_attribute (str): The attribute to update.
        default_value (Any): The default value to use if the source attribute
 doesn't exist.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use if_not_exists to conditionally copy the value
    response = table.update_item(
        Key=key,
        UpdateExpression=f"SET {target_attribute} =
 if_not_exists({source_attribute}, :default)",
        ExpressionAttributeValues={":default": default_value},
        ReturnValues="UPDATED_NEW",
    )

    return response
```

AWS SDK for Python (Boto3) を使用した式順序の更新の使用例。

```
def example_usage():
    """Example of how to use update expression order of operations in
 DynamoDB."""
    # Example parameters
    table_name = "OrderProcessing"
    key = {"OrderId": "order123"}

    print("Example 1: Demonstrating value copying in update expressions")
    try:
        results = demonstrate_value_copying(table_name=table_name, key=key)

        print(f"Initial state: {json.dumps(results['initial_state'],
 default=str)}")
        print(f"Update response: {json.dumps(results['update_response'],
 default=str)}")
        print(f"Final state: {json.dumps(results['final_state'], default=str)}")

        print("\nExplanation:")
        print("1. The initial state had a=1, b=2, c=3")
        print("2. The update expression 'REMOVE a SET b = a, c = b' did the
 following:")
        print("   - Copied the value of 'a' (which was 1) to be used for 'b'")
        print("   - Copied the value of 'b' (which was 2) to be used for 'c'")
        print("   - Removed the attribute 'a'")
        print("3. The final state has b=1, c=2, and 'a' is removed")
        print(
            "4. This demonstrates that DynamoDB uses the values of attributes as
 they were BEFORE any modifications"
        )
    except Exception as e:
        print(f"Error demonstrating value copying: {e}")

    print("\nExample 2: Demonstrating the order of different action types")
    try:
        results = demonstrate_action_order(table_name=table_name, key={"OrderId":
 "order456"})
```

```
        print(f"Initial state: {json.dumps(results['initial_state'],
default=str)}")
        print(f"Update response: {json.dumps(results['update_response'],
default=str)}")
        print(f"Final state: {json.dumps(results['final_state'], default=str)}")

        print("\nExplanation:")
        print("1. The update expression contained multiple action types: REMOVE,
SET, ADD, DELETE")
        print("2. DynamoDB processes these actions in this order: REMOVE, SET,
ADD, DELETE")
        print("3. First, 'to_remove' was removed")
        print("4. Then, 'to_modify' was set to a new value")
        print("5. Next, 'counter' was incremented by 5")
        print("6. Finally, 'B' was removed from the set attribute")
    except Exception as e:
        print(f"Error demonstrating action order: {e}")

    print("\nExample 3: Updating multiple attributes in a single SET action")
    try:
        response = update_with_multiple_set_actions(
            table_name=table_name,
            key={"OrderId": "order789"},
            attributes={
                "Status": "Shipped",
                "ShippingDate": "2025-05-14",
                "TrackingNumber": "1Z999AA10123456784",
            },
        )

        print(
            f"Multiple attributes updated successfully:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error updating multiple attributes: {e}")

    print("\nExample 4: Conditional value copying with if_not_exists")
    try:
        response = update_with_conditional_value_copying(
            table_name=table_name,
            key={"OrderId": "order101"},
            source_attribute="PreferredShippingMethod",
            target_attribute="ShippingMethod",
```

```
            default_value="Standard",
        )

        print(
            f"Conditional value copying result:
{json.dumps(response.get('Attributes', {}), default=str)}"
        )
    except Exception as e:
        print(f"Error with conditional value copying: {e}")

    print("\nKey Points About Update Expression Order of Operations:")
    print(
        "1. Variables in expressions hold copies of attribute values as they
existed BEFORE any modifications"
    )
    print(
        "2. Multiple actions in an update expression are processed in this order:
REMOVE, SET, ADD, DELETE"
    )
    print("3. Within each action type, operations are processed from left to
right")
    print("4. You can reference the same attribute multiple times in an
expression")
    print("5. You can use if_not_exists() to conditionally set values based on
attribute existence")
    print(
        "6. Using a single update expression with multiple actions is more
efficient than multiple separate updates"
    )
    print("7. The update expression is atomic - either all actions succeed or
none do")
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用してウォームスループットで DynamoDB テーブル設定を更新する

次のコード例は、テーブルのウォームスループット設定を更新する方法を示しています。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GlobalSecondaryIndexUpdate;
import
 software.amazon.awssdk.services.dynamodb.model.UpdateGlobalSecondaryIndexAction;
import software.amazon.awssdk.services.dynamodb.model.UpdateTableRequest;
import software.amazon.awssdk.services.dynamodb.model.WarmThroughput;

    public static WarmThroughput buildWarmThroughput(final Long
 readUnitsPerSecond, final Long writeUnitsPerSecond) {
        return WarmThroughput.builder()
            .readUnitsPerSecond(readUnitsPerSecond)
            .writeUnitsPerSecond(writeUnitsPerSecond)
            .build();
    }

    /**
     * Updates a DynamoDB table with warm throughput settings for both the table
 and a global secondary index.
     *
     * @param ddb The DynamoDB client
     * @param tableName The name of the table to update
     * @param tableReadUnitsPerSecond Read units per second for the table
     * @param tableWriteUnitsPerSecond Write units per second for the table
     * @param globalSecondaryIndexName The name of the global secondary index to
 update
     * @param globalSecondaryIndexReadUnitsPerSecond Read units per second for
 the GSI
     * @param globalSecondaryIndexWriteUnitsPerSecond Write units per second for
 the GSI
     */
```

```
    public static void updateDynamoDBTable(
        final DynamoDbClient ddb,
        final String tableName,
        final Long tableReadUnitsPerSecond,
        final Long tableWriteUnitsPerSecond,
        final String globalSecondaryIndexName,
        final Long globalSecondaryIndexReadUnitsPerSecond,
        final Long globalSecondaryIndexWriteUnitsPerSecond) {

        final WarmThroughput tableWarmThroughput =
            buildWarmThroughput(tableReadUnitsPerSecond,
tableWriteUnitsPerSecond);
        final WarmThroughput gsiWarmThroughput =
            buildWarmThroughput(globalSecondaryIndexReadUnitsPerSecond,
globalSecondaryIndexWriteUnitsPerSecond);

        final GlobalSecondaryIndexUpdate globalSecondaryIndexUpdate =
GlobalSecondaryIndexUpdate.builder()
            .update(UpdateGlobalSecondaryIndexAction.builder()
                .indexName(globalSecondaryIndexName)
                .warmThroughput(gsiWarmThroughput)
                .build())
            .build();

        final UpdateTableRequest request = UpdateTableRequest.builder()
            .tableName(tableName)
            .globalSecondaryIndexUpdates(globalSecondaryIndexUpdate)
            .warmThroughput(tableWarmThroughput)
            .build();

        try {
            ddb.updateTable(request);
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }

        System.out.println(SUCCESS_MESSAGE);
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateTable」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
import { DynamoDBClient, UpdateTableCommand } from "@aws-sdk/client-dynamodb";

export async function updateDynamoDBTableWarmThroughput(
  tableName,
  tableReadUnits,
  tableWriteUnits,
  gsiName,
  gsiReadUnits,
  gsiWriteUnits,
  region = "us-east-1"
) {
  try {
    const ddbClient = new DynamoDBClient({ region: region });

    // Construct the update table request
    const updateTableRequest = {
      TableName: tableName,
      GlobalSecondaryIndexUpdates: [
        {
            Update: {
                IndexName: gsiName,
                WarmThroughput: {
                    ReadUnitsPerSecond: gsiReadUnits,
                    WriteUnitsPerSecond: gsiWriteUnits,
                },
            },
        },
      ],
      WarmThroughput: {
          ReadUnitsPerSecond: tableReadUnits,
          WriteUnitsPerSecond: tableWriteUnits,
      },
    };

    const command = new UpdateTableCommand(updateTableRequest);
    const response = await ddbClient.send(command);
```

```
    console.log(`Table updated successfully! Response:
 ${JSON.stringify(response)}`);
    return response;
  } catch (error) {
    console.error(`Error updating table: ${error}`);
    throw error;
  }
}

// Example usage (commented out for testing)
/*
updateDynamoDBTableWarmThroughput(
  'example-table',
  5, 5,
  'example-index',
  2, 2
);
*/
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateTable」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して既存の DynamoDB テーブルのウォームスループット設定を更新します。

```
from boto3 import client
from botocore.exceptions import ClientError


def update_dynamodb_table_warm_throughput(
    table_name,
    table_read_units,
    table_write_units,
    gsi_name,
    gsi_read_units,
    gsi_write_units,
```

```
    region_name="us-east-1",
):
    """
    Updates the warm throughput of a DynamoDB table and a global secondary index.

    :param table_name: The name of the table to update.
    :param table_read_units: The new read units per second for the table's warm
 throughput.
    :param table_write_units: The new write units per second for the table's warm
 throughput.
    :param gsi_name: The name of the global secondary index to update.
    :param gsi_read_units: The new read units per second for the GSI's warm
 throughput.
    :param gsi_write_units: The new write units per second for the GSI's warm
 throughput.
    :param region_name: The AWS Region name to target. defaults to us-east-1
    :return: The response from the update_table operation
    """
    try:
        ddb = client("dynamodb", region_name=region_name)

        # Update the table's warm throughput
        table_warm_throughput = {
            "ReadUnitsPerSecond": table_read_units,
            "WriteUnitsPerSecond": table_write_units,
        }

        # Update the global secondary index's warm throughput
        gsi_warm_throughput = {
            "ReadUnitsPerSecond": gsi_read_units,
            "WriteUnitsPerSecond": gsi_write_units,
        }

        # Construct the global secondary index update
        global_secondary_index_update = [
            {"Update": {"IndexName": gsi_name, "WarmThroughput":
 gsi_warm_throughput}}
        ]

        # Construct the update table request
        update_table_request = {
            "TableName": table_name,
            "GlobalSecondaryIndexUpdates": global_secondary_index_update,
            "WarmThroughput": table_warm_throughput,
```

```
        }

        # Update the table
        response = ddb.update_table(**update_table_request)
        print("Table updated successfully!")
        return response  # Make sure to return the response
    except ClientError as e:
        print(f"Error updating table: {e}")
        raise e
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の「UpdateTable」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して TTL を含む DynamoDB 項目を更新する

以下のコード例は、項目の TTL の更新方法を示しています。

Java

SDK for Java 2.x

テーブル内の既存の DynamoDB 項目の TTL を更新します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.ResourceNotFoundException;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

    public UpdateItemResponse updateItemWithTTL(
```

```
        final String tableName, final String primaryKeyValue, final String
sortKeyValue) {
        // Get current time in epoch second format
        final long currentTime = System.currentTimeMillis() / 1000;

        // Calculate expiration time 90 days from now in epoch second format
        final long expireDate = currentTime + (DAYS_TO_EXPIRE * SECONDS_PER_DAY);

        // Create the key map for the item to update
        final Map<String, AttributeValue> keyMap = new HashMap<>();
        keyMap.put(PRIMARY_KEY_ATTR,
AttributeValue.builder().s(primaryKeyValue).build());
        keyMap.put(SORT_KEY_ATTR,
AttributeValue.builder().s(sortKeyValue).build());

        // Create the expression attribute values
        final Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();
        expressionAttributeValues.put(
            ":c",
AttributeValue.builder().n(String.valueOf(currentTime)).build());
        expressionAttributeValues.put(
            ":e",
AttributeValue.builder().n(String.valueOf(expireDate)).build());

        final UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(keyMap)
            .updateExpression(UPDATE_EXPRESSION)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        try {
            final UpdateItemResponse response =
dynamoDbClient.updateItem(request);
            System.out.println(String.format(SUCCESS_MESSAGE, tableName));
            return response;
        } catch (ResourceNotFoundException e) {
            System.err.format(TABLE_NOT_FOUND_ERROR, tableName);
            throw e;
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            throw e;
        }
```

```
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

export const updateItem = async (tableName, partitionKey, sortKey, region = 'us-
east-1') => {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };

    try {
        const data = await client.send(new UpdateItemCommand(params));
        const responseData = unmarshall(data.Attributes);
        console.log("Item updated successfully: %s", responseData);
        return responseData;
    } catch (err) {
```

```
        console.error("Error updating item:", err);
        throw err;
    }
}

// Example usage (commented out for testing)
// updateItem('your-table-name', 'your-partition-key-value', 'your-sort-key-
value');
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

```
from datetime import datetime, timedelta

import boto3


def update_dynamodb_item(table_name, region, primary_key, sort_key):
    """
    Update an existing DynamoDB item with a TTL.
    :param table_name: Name of the DynamoDB table
    :param region: AWS Region of the table - example `us-east-1`
    :param primary_key: one attribute known as the partition key.
    :param sort_key: Also known as a range attribute.
    :return: Void (nothing)
    """
    try:
        # Create the DynamoDB resource.
        dynamodb = boto3.resource("dynamodb", region_name=region)
        table = dynamodb.Table(table_name)

        # Get the current time in epoch second format
        current_time = int(datetime.now().timestamp())

        # Calculate the expireAt time (90 days from now) in epoch second format
        expire_at = int((datetime.now() + timedelta(days=90)).timestamp())
```

```
        table.update_item(
            Key={"partitionKey": primary_key, "sortKey": sort_key},
            UpdateExpression="set updatedAt=:c, expireAt=:e",
            ExpressionAttributeValues={":c": current_time, ":e": expire_at},
        )

        print("Item updated successfully.")
    except Exception as e:
        print(f"Error updating item: {e}")


# Replace with your own values
update_dynamodb_item(
    "your-table-name", "us-west-2", "your-partition-key-value", "your-sort-key-
value"
)
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK で PartiQL UPDATE ステートメントを使用して DynamoDB データを更新する

次のコード例は、PartiQL UPDATE ステートメントを使用してデータを更新する方法を示しています。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript で PartiQL UPDATE ステートメントを使用して DynamoDB テーブルの項目を更新します。

```
/**
```

```
 * This example demonstrates how to update items in a DynamoDB table using
 PartiQL.
 * It shows different ways to update documents with various index types.
 */
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

/**
 * Update a single attribute of an item using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const updateSingleAttribute = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeName: string,
  attributeValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ?`,
    Parameters: [attributeValue, partitionKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
```

```
  }
};

/**
 * Update multiple attributes of an item using PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param attributeUpdates - Object containing attribute names and their new
 values
 * @returns The response from the ExecuteStatementCommand
 */
export const updateMultipleAttributes = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeUpdates: Record<string, any>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create SET clause for each attribute
  const setClause = Object.keys(attributeUpdates)
    .map((attr, index) => `${attr} = ?`)
    .join(", ");

  // Create parameters array with attribute values followed by the partition key
 value
  const parameters = [...Object.values(attributeUpdates), partitionKeyValue];

  const params = {
    Statement: `UPDATE "${tableName}" SET ${setClause} WHERE ${partitionKeyName}
 = ?`,
    Parameters: parameters,
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
```

```
  }
};

/**
 * Update an item identified by a composite key (partition key + sort key) using
 PartiQL.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param sortKeyName - The name of the sort key attribute
 * @param sortKeyValue - The value of the sort key
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @returns The response from the ExecuteStatementCommand
 */
export const updateItemWithCompositeKey = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  sortKeyName: string,
  sortKeyValue: string | number,
  attributeName: string,
  attributeValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ? AND ${sortKeyName} = ?`,
    Parameters: [attributeValue, partitionKeyValue, sortKeyValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated successfully");
    return data;
  } catch (err) {
    console.error("Error updating item:", err);
    throw err;
  }
};
```

```
/**
 * Update an item with a condition to ensure the update only happens if a
 condition is met.
 *
 * @param tableName - The name of the DynamoDB table
 * @param partitionKeyName - The name of the partition key attribute
 * @param partitionKeyValue - The value of the partition key
 * @param attributeName - The name of the attribute to update
 * @param attributeValue - The new value for the attribute
 * @param conditionAttribute - The attribute to check in the condition
 * @param conditionValue - The value to compare against in the condition
 * @returns The response from the ExecuteStatementCommand
 */
export const updateItemWithCondition = async (
  tableName: string,
  partitionKeyName: string,
  partitionKeyValue: string | number,
  attributeName: string,
  attributeValue: any,
  conditionAttribute: string,
  conditionValue: any
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  const params = {
    Statement: `UPDATE "${tableName}" SET ${attributeName} = ? WHERE
 ${partitionKeyName} = ? AND ${conditionAttribute} = ?`,
    Parameters: [attributeValue, partitionKeyValue, conditionValue],
  };

  try {
    const data = await docClient.send(new ExecuteStatementCommand(params));
    console.log("Item updated with condition successfully");
    return data;
  } catch (err) {
    console.error("Error updating item with condition:", err);
    throw err;
  }
};

/**
 * Batch update multiple items using PartiQL.
 *
```

```
 * @param tableName - The name of the DynamoDB table
 * @param updates - Array of objects containing key and update information
 * @returns The response from the BatchExecuteStatementCommand
 */
export const batchUpdateItems = async (
  tableName: string,
  updates: Array<{
    partitionKeyName: string;
    partitionKeyValue: string | number;
    attributeName: string;
    attributeValue: any;
  }>
) => {
  const client = new DynamoDBClient({});
  const docClient = DynamoDBDocumentClient.from(client);

  // Create statements for each update
  const statements = updates.map((update) => {
    return {
      Statement: `UPDATE "${tableName}" SET ${update.attributeName} = ? WHERE
 ${update.partitionKeyName} = ?`,
      Parameters: [update.attributeValue, update.partitionKeyValue],
    };
  });

  const params = {
    Statements: statements,
  };

  try {
    const data = await docClient.send(new BatchExecuteStatementCommand(params));
    console.log("Items batch updated successfully");
    return data;
  } catch (err) {
    console.error("Error batch updating items:", err);
    throw err;
  }
};

/**
 * Example usage showing how to update items with different index types
 */
export const updateExamples = async () => {
  // Update a single attribute using a simple primary key
```

```
 await updateSingleAttribute("UsersTable", "userId", "user123", "email",
"newemail@example.com");

 // Update multiple attributes at once
 await updateMultipleAttributes("UsersTable", "userId", "user123", {
   email: "newemail@example.com",
   name: "John Smith",
   lastLogin: new Date().toISOString(),
 });

 // Update an item with a composite key (partition key + sort key)
 await updateItemWithCompositeKey(
   "OrdersTable",
   "orderId",
   "order456",
   "productId",
   "prod789",
   "quantity",
   5
 );

 // Update with a condition
 await updateItemWithCondition(
   "UsersTable",
   "userId",
   "user123",
   "userStatus",
   "active",
   "userType",
   "premium"
 );

 // Batch update multiple items
 await batchUpdateItems("UsersTable", [
   {
     partitionKeyName: "userId",
     partitionKeyValue: "user123",
     attributeName: "lastLogin",
     attributeValue: new Date().toISOString(),
   },
   {
     partitionKeyName: "userId",
     partitionKeyValue: "user456",
     attributeName: "lastLogin",
```

```
      attributeValue: new Date().toISOString(),
    },
  ]);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - BatchExecuteStatement
  - ExecuteStatement

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

API Gateway を使用して Lambda 関数を呼び出す

次のコード例は、Amazon API Gateway によって呼び出される AWS Lambda 関数の作成方法を示しています。

Java

SDK for Java 2.x

Lambda Java ランタイム API を使用して AWS Lambda 関数を作成する方法を示します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、Amazon API Gateway によって呼び出される Lambda 関数を作成する方法を示します。この関数は、Amazon DynamoDB テーブルをスキャンして、Amazon Simple Notification Service (Amazon SNS) を使用して、従業員に年間の記念日を祝福するテキストメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Lambda JavaScript ランタイム API を使用して AWS Lambda 関数を作成する方法を示します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、Amazon API Gateway によって呼び出される Lambda 関数を作成する方法を示します。この関数は、Amazon DynamoDB テーブルをスキャンして、Amazon Simple Notification Service (Amazon SNS) を使用して、従業員に年間の記念日を祝福するテキストメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

この例は、AWS Lambda 関数を対象とする Amazon API Gateway REST API を作成して使用する方法を示しています。Lambda ハンドラーは、HTTP メソッドに基づいてルーティングする方法を示します。クエリ文字列、ヘッダー、および本文からデータを取得する方法。そして、JSON 応答を返す方法。

- Lambda 関数をデプロイします。
- API ゲートウェイ REST API を作成します。
- Lambda 関数をターゲットとする REST リソースを作成します。
- API Gateway に Lambda 関数を呼び出す権限を付与します。

- リクエストパッケージを使用して、REST API にリクエストを送信します。
- デモ中に作成されたすべてのリソースをクリーンアップします。

この例は GitHub で最もよく確認できます。完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda
- Amazon SNS

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK で DynamoDB を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Step Functions を使用して Lambda 関数を呼び出す

次のコード例は、AWS Lambda の関数を順次呼び出す AWS Step Functions ステートマシンを作成する方法を示しています。

Java

SDK for Java 2.x

AWS Step Functions と AWS SDK for Java 2.x を使用して AWS サーバーレスワークフローを作成する方法を示します。各ワークフローステップは、AWS Lambda 関数を使用して実装されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda
- Amazon SES
- ステップ関数

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB のドキュメントモデルを使用する

次のコード例は、DynamoDB のドキュメントモデルと AWS SDK を使用して、作成、読み込み、更新、削除 (CRUD) オペレーションとバッチオペレーションを実行する方法を示しています。

詳細については、「ドキュメントモデル」を参照してください。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

ドキュメントモデルを使用して CRUD オペレーションを実行します。

```
/// <summary>
/// Performs CRUD operations on an Amazon DynamoDB table.
/// </summary>
public class MidlevelItemCRUD
{
    public static async Task Main()
    {
        var tableName = "ProductCatalog";
        var sampleBookId = 555;

        var client = new AmazonDynamoDBClient();
        var productCatalog = LoadTable(client, tableName);

        await CreateBookItem(productCatalog, sampleBookId);
        RetrieveBook(productCatalog, sampleBookId);

        // Couple of sample updates.
        UpdateMultipleAttributes(productCatalog, sampleBookId);
```

```
            UpdateBookPriceConditionally(productCatalog, sampleBookId);

            // Delete.
            await DeleteBook(productCatalog, sampleBookId);
        }

        /// <summary>
        /// Loads the contents of a DynamoDB table.
        /// </summary>
        /// <param name="client">An initialized DynamoDB client object.</param>
        /// <param name="tableName">The name of the table to load.</param>
        /// <returns>A DynamoDB table object.</returns>
        public static Table LoadTable(IAmazonDynamoDB client, string tableName)
        {
            Table productCatalog = Table.LoadTable(client, tableName);
            return productCatalog;
        }

        /// <summary>
        /// Creates an example book item and adds it to the DynamoDB table
        /// ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async Task CreateBookItem(Table productCatalog, int
sampleBookId)
        {
            Console.WriteLine("\n*** Executing CreateBookItem() ***");
            var book = new Document
            {
                ["Id"] = sampleBookId,
                ["Title"] = "Book " + sampleBookId,
                ["Price"] = 19.99,
                ["ISBN"] = "111-1111111111",
                ["Authors"] = new List<string> { "Author 1", "Author 2", "Author
3" },
                ["PageCount"] = 500,
                ["Dimensions"] = "8.5x11x.5",
                ["InPublication"] = new DynamoDBBool(true),
                ["InStock"] = new DynamoDBBool(false),
                ["QuantityOnHand"] = 0,
            };
```

```
            // Adds the book to the ProductCatalog table.
            await productCatalog.PutItemAsync(book);
        }

        /// <summary>
        /// Retrieves an item, a book, from the DynamoDB ProductCatalog table.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void RetrieveBook(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing RetrieveBook() ***");

            // Optional configuration.
            var config = new GetItemOperationConfig
            {
                AttributesToGet = new List<string> { "Id", "ISBN", "Title",
"Authors", "Price" },
                ConsistentRead = true,
            };

            Document document = await productCatalog.GetItemAsync(sampleBookId,
config);
            Console.WriteLine("RetrieveBook: Printing book retrieved...");
            PrintDocument(document);
        }

        /// <summary>
        /// Updates multiple attributes for a book and writes the changes to the
        /// DynamoDB table ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void UpdateMultipleAttributes(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\nUpdating multiple attributes....");
            int partitionKey = sampleBookId;
```

```
            var book = new Document
            {
                ["Id"] = partitionKey,

                // List of attribute updates.
                // The following replaces the existing authors list.
                ["Authors"] = new List<string> { "Author x", "Author y" },
                ["newAttribute"] = "New Value",
                ["ISBN"] = null, // Remove it.
            };

            // Optional parameters.
            var config = new UpdateItemOperationConfig
            {
                // Gets updated item in response.
                ReturnValues = ReturnValues.AllNewAttributes,
            };

            Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
            Console.WriteLine("UpdateMultipleAttributes: Printing item after
updates ...");
            PrintDocument(updatedBook);
        }

        /// <summary>
        /// Updates a book item if it meets the specified criteria.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async void UpdateBookPriceConditionally(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing UpdateBookPriceConditionally()
***");

            int partitionKey = sampleBookId;

            var book = new Document
            {
                ["Id"] = partitionKey,
                ["Price"] = 29.99,
```

```
            };

            // For conditional price update, creating a condition expression.
            var expr = new Expression
            {
                ExpressionStatement = "Price = :val",
            };
            expr.ExpressionAttributeValues[":val"] = 19.00;

            // Optional parameters.
            var config = new UpdateItemOperationConfig
            {
                ConditionalExpression = expr,
                ReturnValues = ReturnValues.AllNewAttributes,
            };

            Document updatedBook = await productCatalog.UpdateItemAsync(book,
config);
            Console.WriteLine("UpdateBookPriceConditionally: Printing item whose
price was conditionally updated");
            PrintDocument(updatedBook);
        }

        /// <summary>
        /// Deletes the book with the supplied Id value from the DynamoDB table
        /// ProductCatalog.
        /// </summary>
        /// <param name="productCatalog">A DynamoDB table object.</param>
        /// <param name="sampleBookId">An integer value representing the book's
ID.</param>
        public static async Task DeleteBook(
          Table productCatalog,
          int sampleBookId)
        {
            Console.WriteLine("\n*** Executing DeleteBook() ***");

            // Optional configuration.
            var config = new DeleteItemOperationConfig
            {
                // Returns the deleted item.
                ReturnValues = ReturnValues.AllOldAttributes,
            };
            Document document = await
productCatalog.DeleteItemAsync(sampleBookId, config);
```

```
            Console.WriteLine("DeleteBook: Printing deleted just deleted...");

            PrintDocument(document);
        }

        /// <summary>
        /// Prints the information for the supplied DynamoDB document.
        /// </summary>
        /// <param name="updatedDocument">A DynamoDB document object.</param>
        public static void PrintDocument(Document updatedDocument)
        {
            if (updatedDocument is null)
            {
                return;
            }

            foreach (var attribute in updatedDocument.GetAttributeNames())
            {
                string stringValue = null;
                var value = updatedDocument[attribute];

                if (value is null)
                {
                    continue;
                }

                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}", attribute,
stringValue);
            }
        }
    }
```

ドキュメントモデルを使用してバッチ書き込みオペレーションを実行します。

```
    /// <summary>
    /// Shows how to use mid-level Amazon DynamoDB API calls to perform batch
    /// operations.
    /// </summary>
    public class MidLevelBatchWriteItem
    {
        public static async Task Main()
        {
            IAmazonDynamoDB client = new AmazonDynamoDBClient();

            await SingleTableBatchWrite(client);
            await MultiTableBatchWrite(client);
        }

        /// <summary>
        /// Perform a batch operation on a single DynamoDB table.
        /// </summary>
        /// <param name="client">An initialized DynamoDB object.</param>
        public static async Task SingleTableBatchWrite(IAmazonDynamoDB client)
        {
            Table productCatalog = Table.LoadTable(client, "ProductCatalog");
            var batchWrite = productCatalog.CreateBatchWrite();

            var book1 = new Document
            {
                ["Id"] = 902,
                ["Title"] = "My book1 in batch write using .NET helper classes",
                ["ISBN"] = "902-11-11-1111",
                ["Price"] = 10,
                ["ProductCategory"] = "Book",
                ["Authors"] = new List<string> { "Author 1", "Author 2", "Author
3" },
                ["Dimensions"] = "8.5x11x.5",
                ["InStock"] = new DynamoDBBool(true),
                ["QuantityOnHand"] = new DynamoDBNull(), // Quantity is unknown
at this time.
            };
```

```
            batchWrite.AddDocumentToPut(book1);

            // Specify delete item using overload that takes PK.
            batchWrite.AddKeyToDelete(12345);
            Console.WriteLine("Performing batch write in
SingleTableBatchWrite()");
            await batchWrite.ExecuteAsync();
        }

        /// <summary>
        /// Perform a batch operation involving multiple DynamoDB tables.
        /// </summary>
        /// <param name="client">An initialized DynamoDB client object.</param>
        public static async Task MultiTableBatchWrite(IAmazonDynamoDB client)
        {
            // Specify item to add in the Forum table.
            Table forum = Table.LoadTable(client, "Forum");
            var forumBatchWrite = forum.CreateBatchWrite();

            var forum1 = new Document
            {
                ["Name"] = "Test BatchWrite Forum",
                ["Threads"] = 0,
            };
            forumBatchWrite.AddDocumentToPut(forum1);

            // Specify item to add in the Thread table.
            Table thread = Table.LoadTable(client, "Thread");
            var threadBatchWrite = thread.CreateBatchWrite();

            var thread1 = new Document
            {
                ["ForumName"] = "S3 forum",
                ["Subject"] = "My sample question",
                ["Message"] = "Message text",
                ["KeywordTags"] = new List<string> { "S3", "Bucket" },
            };
            threadBatchWrite.AddDocumentToPut(thread1);

            // Specify item to delete from the Thread table.
            threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

            // Create multi-table batch.
            var superBatch = new MultiTableDocumentBatchWrite();
```

```
            superBatch.AddBatch(forumBatchWrite);
            superBatch.AddBatch(threadBatchWrite);
            Console.WriteLine("Performing batch write in
MultiTableBatchWrite()");

            // Execute the batch.
            await superBatch.ExecuteAsync();
        }
    }
```

ドキュメントモデルを使用してテーブルをスキャンします。

```
    /// <summary>
    /// Shows how to use mid-level Amazon DynamoDB API calls to scan a DynamoDB
    /// table for values.
    /// </summary>
    public class MidLevelScanOnly
    {
        public static async Task Main()
        {
            IAmazonDynamoDB client = new AmazonDynamoDBClient();

            Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");

            await FindProductsWithNegativePrice(productCatalogTable);
            await FindProductsWithNegativePriceWithConfig(productCatalogTable);
        }

        /// <summary>
        /// Retrieves any products that have a negative price in a DynamoDB
table.
        /// </summary>
        /// <param name="productCatalogTable">A DynamoDB table object.</param>
        public static async Task FindProductsWithNegativePrice(
          Table productCatalogTable)
        {
            // Assume there is a price error. So we scan to find items priced <
0.
            var scanFilter = new ScanFilter();
```

```
            scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

            Search search = productCatalogTable.Scan(scanFilter);

            do
            {
                var documentList = await search.GetNextSetAsync();
                Console.WriteLine("\nFindProductsWithNegativePrice:
printing .....");

                foreach (var document in documentList)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Finds any items in the ProductCatalog table using a DynamoDB
        /// configuration object.
        /// </summary>
        /// <param name="productCatalogTable">A DynamoDB table object.</param>
        public static async Task FindProductsWithNegativePriceWithConfig(
          Table productCatalogTable)
        {
            // Assume there is a price error. So we scan to find items priced <
0.
            var scanFilter = new ScanFilter();
            scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

            var config = new ScanOperationConfig()
            {
                Filter = scanFilter,
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string> { "Title", "Id" },
            };

            Search search = productCatalogTable.Scan(config);

            do
            {
                var documentList = await search.GetNextSetAsync();
```

```
                Console.WriteLine("\nFindProductsWithNegativePriceWithConfig:
printing .....");

                foreach (var document in documentList)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Displays the details of the passed DynamoDB document object on the
        /// console.
        /// </summary>
        /// <param name="document">A DynamoDB document object.</param>
        public static void PrintDocument(Document document)
        {
            Console.WriteLine();
            foreach (var attribute in document.GetAttributeNames())
            {
                string stringValue = null;
                var value = document[attribute];
                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}");
            }
        }
    }
```

ドキュメントモデルを使用して、テーブルをクエリおよびスキャンする。

```
    /// <summary>
    /// Shows how to perform mid-level query procedures on an Amazon DynamoDB
    /// table.
    /// </summary>
    public class MidLevelQueryAndScan
    {
        public static async Task Main()
        {
            IAmazonDynamoDB client = new AmazonDynamoDBClient();

            // Query examples.
            Table replyTable = Table.LoadTable(client, "Reply");
            string forumName = "Amazon DynamoDB";
            string threadSubject = "DynamoDB Thread 2";

            await FindRepliesInLast15Days(replyTable);
            await FindRepliesInLast15DaysWithConfig(replyTable, forumName,
threadSubject);
            await FindRepliesPostedWithinTimePeriod(replyTable, forumName,
threadSubject);

            // Get Example.
            Table productCatalogTable = Table.LoadTable(client,
"ProductCatalog");
            int productId = 101;

            await GetProduct(productCatalogTable, productId);
        }

        /// <summary>
        /// Retrieves information about a product from the DynamoDB table
        /// ProductCatalog based on the product ID and displays the information
        /// on the console.
        /// </summary>
        /// <param name="tableName">The name of the table from which to retrieve
        /// product information.</param>
        /// <param name="productId">The ID of the product to retrieve.</param>
        public static async Task GetProduct(Table tableName, int productId)
        {
            Console.WriteLine("*** Executing GetProduct() ***");
            Document productDocument = await tableName.GetItemAsync(productId);
            if (productDocument != null)
```

```
            {
                PrintDocument(productDocument);
            }
            else
            {
                Console.WriteLine("Error: product " + productId + " does not
exist");
            }
        }

        /// <summary>
        /// Retrieves replies from the passed DynamoDB table object.
        /// </summary>
        /// <param name="table">The table we want to query.</param>
        public static async Task FindRepliesInLast15Days(
          Table table)
        {
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
            var filter = new QueryFilter("Id", QueryOperator.Equal, "Id");
            filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

            // Use Query overloads that take the minimum required query
parameters.
            Search search = table.Query(filter);

            do
            {
                var documentSet = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesInLast15Days:
printing .............");

                foreach (var document in documentSet)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Retrieve replies made during a specific time period.
        /// </summary>
        /// <param name="table">The table we want to query.</param>
```

```
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">The subject of the thread, which we are
        /// searching for replies.</param>
        public static async Task FindRepliesPostedWithinTimePeriod(
          Table table,
          string forumName,
          string threadSubject)
        {
            DateTime startDate = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0,
0));
            DateTime endDate = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0,
0));

            var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadSubject);
            filter.AddCondition("ReplyDateTime", QueryOperator.Between,
startDate, endDate);

            var config = new QueryOperationConfig()
            {
                Limit = 2, // 2 items/page.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
        {
          "Message",
          "ReplyDateTime",
          "PostedBy",
        },
                ConsistentRead = true,
                Filter = filter,
            };

            Search search = table.Query(config);

            do
            {
                var documentList = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: printing
replies posted within dates: {0} and {1} .....", startDate, endDate);

                foreach (var document in documentList)
                {
                    PrintDocument(document);
```

```
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Perform a query for replies made in the last 15 days using a DynamoDB
        /// QueryOperationConfig object.
        /// </summary>
        /// <param name="table">The table we want to query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadName">The bane of the thread that we are searching
        /// for replies.</param>
        public static async Task FindRepliesInLast15DaysWithConfig(
          Table table,
          string forumName,
          string threadName)
        {
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
            var filter = new QueryFilter("Id", QueryOperator.Equal, forumName +
"#" + threadName);
            filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

            var config = new QueryOperationConfig()
            {
                Filter = filter,

                // Optional parameters.
                Select = SelectValues.SpecificAttributes,
                AttributesToGet = new List<string>
                {
                  "Message",
                  "ReplyDateTime",
                  "PostedBy",
                },
                ConsistentRead = true,
            };

            Search search = table.Query(config);

            do
            {
```

```
                var documentSet = await search.GetNextSetAsync();
                Console.WriteLine("\nFindRepliesInLast15DaysWithConfig:
printing ............");

                foreach (var document in documentSet)
                {
                    PrintDocument(document);
                }
            }
            while (!search.IsDone);
        }

        /// <summary>
        /// Displays the contents of the passed DynamoDB document on the console.
        /// </summary>
        /// <param name="document">A DynamoDB document to display.</param>
        public static void PrintDocument(Document document)
        {
            Console.WriteLine();
            foreach (var attribute in document.GetAttributeNames())
            {
                string stringValue = null;
                var value = document[attribute];

                if (value is Primitive)
                {
                    stringValue = value.AsPrimitive().Value.ToString();
                }
                else if (value is PrimitiveList)
                {
                    stringValue = string.Join(",", (from primitive
                      in value.AsPrimitiveList().Entries
                                                    select
primitive.Value).ToArray());
                }

                Console.WriteLine($"{attribute} - {stringValue}");
            }
        }
    }
```

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB 用の高レベルのオブジェクト永続性モデルを使用する

次のコード例は、DynamoDB と AWS SDK のオブジェクト永続性モデルを使用して、作成、読み取り、更新、削除 (CRUD) オペレーションとバッチオペレーションを実行する方法を示しています。

詳細については、「オブジェクト永続性モデル」を参照してください。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

高レベルのオブジェクト永続性モデルを使用して CRUD オペレーションを実行します。

```
/// <summary>
/// Shows how to perform high-level CRUD operations on an Amazon DynamoDB
/// table.
/// </summary>
public class HighLevelItemCrud
{
    public static async Task Main()
    {
        var client = new AmazonDynamoDBClient();
        DynamoDBContext context = new DynamoDBContext(client);
        await PerformCRUDOperations(context);
    }

    public static async Task PerformCRUDOperations(IDynamoDBContext context)
    {
        int bookId = 1001; // Some unique value.
        Book myBook = new Book
```

```
            {
                Id = bookId,
                Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
                Isbn = "111-1111111001",
                BookAuthors = new List<string> { "Author 1", "Author 2" },
            };

            // Save the book to the ProductCatalog table.
            await context.SaveAsync(myBook);

            // Retrieve the book from the ProductCatalog table.
            Book bookRetrieved = await context.LoadAsync<Book>(bookId);

            // Update some properties.
            bookRetrieved.Isbn = "222-2222221001";

            // Update existing authors list with the following values.
            bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author
x" };
            await context.SaveAsync(bookRetrieved);

            // Retrieve the updated book. This time, add the optional
            // ConsistentRead parameter using DynamoDBContextConfig object.
            await context.LoadAsync<Book>(bookId, new DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            // Delete the book.
            await context.DeleteAsync<Book>(bookId);

            // Try to retrieve deleted book. It should return null.
            Book deletedBook = await context.LoadAsync<Book>(bookId, new
DynamoDBContextConfig
            {
                ConsistentRead = true,
            });

            if (deletedBook == null)
            {
                Console.WriteLine("Book is deleted");
            }
        }
    }
```

高レベルのオブジェクト永続性モデルを使用してバッチ書き込みオペレーションを実行します。

```
/// <summary>
/// Performs high-level batch write operations to an Amazon DynamoDB table.
/// This example was written using the AWS SDK for .NET version 3.7 and .NET
/// Core 5.0.
/// </summary>
public class HighLevelBatchWriteItem
{
    public static async Task SingleTableBatchWrite(IDynamoDBContext context)
    {
        Book book1 = new Book
        {
            Id = 902,
            InPublication = true,
            Isbn = "902-11-11-1111",
            PageCount = "100",
            Price = 10,
            ProductCategory = "Book",
            Title = "My book3 in batch write",
        };

        Book book2 = new Book
        {
            Id = 903,
            InPublication = true,
            Isbn = "903-11-11-1111",
            PageCount = "200",
            Price = 10,
            ProductCategory = "Book",
            Title = "My book4 in batch write",
        };

        var bookBatch = context.CreateBatchWrite<Book>();
        bookBatch.AddPutItems(new List<Book> { book1, book2 });

        Console.WriteLine("Adding two books to ProductCatalog table.");
        await bookBatch.ExecuteAsync();
```

```
        }

        public static async Task MultiTableBatchWrite(IDynamoDBContext context)
        {
            // New Forum item.
            Forum newForum = new Forum
            {
                Name = "Test BatchWrite Forum",
                Threads = 0,
            };
            var forumBatch = context.CreateBatchWrite<Forum>();
            forumBatch.AddPutItem(newForum);

            // New Thread item.
            Thread newThread = new Thread
            {
                ForumName = "S3 forum",
                Subject = "My sample question",
                KeywordTags = new List<string> { "S3", "Bucket" },
                Message = "Message text",
            };

            DynamoDBOperationConfig config = new DynamoDBOperationConfig();
            config.SkipVersionCheck = true;
            var threadBatch = context.CreateBatchWrite<Thread>(config);
            threadBatch.AddPutItem(newThread);
            threadBatch.AddDeleteKey("some partition key value", "some sort key
value");

            var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);

            Console.WriteLine("Performing batch write in
MultiTableBatchWrite().");
            await superBatch.ExecuteAsync();
        }

        public static async Task Main()
        {
            AmazonDynamoDBClient client = new AmazonDynamoDBClient();
            DynamoDBContext context = new DynamoDBContext(client);

            await SingleTableBatchWrite(context);
            await MultiTableBatchWrite(context);
        }
```

```
    }
```

オブジェクト永続性モデルを使用して、任意のデータをテーブルにマッピングします。

```
    /// <summary>
    /// Shows how to map arbitrary data to an Amazon DynamoDB table.
    /// </summary>
    public class HighLevelMappingArbitraryData
    {
        /// <summary>
        /// Creates a book, adds it to the DynamoDB ProductCatalog table,
retrieves
        /// the new book from the table, updates the dimensions and writes the
        /// changed item back to the table.
        /// </summary>
        /// <param name="context">The DynamoDB context object used to write and
        /// read data from the table.</param>
        public static async Task AddRetrieveUpdateBook(IDynamoDBContext context)
        {
            // Create a book.
            DimensionType myBookDimensions = new DimensionType()
            {
                Length = 8M,
                Height = 11M,
                Thickness = 0.5M,
            };

            Book myBook = new Book
            {
                Id = 501,
                Title = "AWS SDK for .NET Object Persistence Model Handling
Arbitrary Data",
                Isbn = "999-9999999999",
                BookAuthors = new List<string> { "Author 1", "Author 2" },
                Dimensions = myBookDimensions,
            };

            // Add the book to the DynamoDB table ProductCatalog.
            await context.SaveAsync(myBook);
```

```
            // Retrieve the book.
            Book bookRetrieved = await context.LoadAsync<Book>(501);

            // Update the book dimensions property.
            bookRetrieved.Dimensions.Height += 1;
            bookRetrieved.Dimensions.Length += 1;
            bookRetrieved.Dimensions.Thickness += 0.2M;

            // Write the changed item to the table.
            await context.SaveAsync(bookRetrieved);
        }

        public static async Task Main()
        {
            var client = new AmazonDynamoDBClient();
            DynamoDBContext context = new DynamoDBContext(client);
            await AddRetrieveUpdateBook(context);
        }
    }
```

高レベルのオブジェクト永続性モデルを使用してテーブルをクエリおよびスキャンします。

```
    /// <summary>
    /// Shows how to perform high-level query and scan operations to Amazon
    /// DynamoDB tables.
    /// </summary>
    public class HighLevelQueryAndScan
    {
        public static async Task Main()
        {
            var client = new AmazonDynamoDBClient();

            DynamoDBContext context = new DynamoDBContext(client);

            // Get an item.
            await GetBook(context, 101);

            // Sample forum and thread to test queries.
            string forumName = "Amazon DynamoDB";
            string threadSubject = "DynamoDB Thread 1";
```

```
            // Sample queries.
            await FindRepliesInLast15Days(context, forumName, threadSubject);
            await FindRepliesPostedWithinTimePeriod(context, forumName,
threadSubject);

            // Scan table.
            await FindProductsPricedLessThanZero(context);
        }

        public static async Task GetBook(IDynamoDBContext context, int productId)
        {
            Book bookItem = await context.LoadAsync<Book>(productId);

            Console.WriteLine("\nGetBook: Printing result.....");
            Console.WriteLine($"Title: {bookItem.Title} \n ISBN:{bookItem.Isbn}
\n No. of pages: {bookItem.PageCount}");
        }

        /// <summary>
        /// Queries a DynamoDB table to find replies posted within the last 15
days.
        /// </summary>
        /// <param name="context">The DynamoDB context used to perform the
query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">The thread object containing the query
parameters.</param>
        public static async Task FindRepliesInLast15Days(
          IDynamoDBContext context,
          string forumName,
          string threadSubject)
        {
            string replyId = $"{forumName} #{threadSubject}";
            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);

            List<object> times = new List<object>();
            times.Add(twoWeeksAgoDate);

            List<ScanCondition> scs = new List<ScanCondition>();
            var sc = new ScanCondition("PostedBy", ScanOperator.GreaterThan,
times.ToArray());
            scs.Add(sc);
```

```
            var cfg = new DynamoDBOperationConfig
            {
                QueryFilter = scs,
            };

            AsyncSearch<Reply> response = context.QueryAsync<Reply>(replyId,
cfg);
            IEnumerable<Reply> latestReplies = await
response.GetRemainingAsync();

            Console.WriteLine("\nReplies in last 15 days:");

            foreach (Reply r in latestReplies)
            {

Console.WriteLine($"{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
            }
        }

        /// <summary>
        /// Queries for replies posted within a specific time period.
        /// </summary>
        /// <param name="context">The DynamoDB context used to perform the
query.</param>
        /// <param name="forumName">The name of the forum that we're interested
in.</param>
        /// <param name="threadSubject">Information about the subject that we're
        /// interested in.</param>
        public static async Task FindRepliesPostedWithinTimePeriod(
          IDynamoDBContext context,
          string forumName,
          string threadSubject)
        {
            string forumId = forumName + "#" + threadSubject;
            Console.WriteLine("\nReplies posted within time period:");

            DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);
            DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);

            List<object> times = new List<object>();
            times.Add(startDate);
            times.Add(endDate);
```

```
            List<ScanCondition> scs = new List<ScanCondition>();
            var sc = new ScanCondition("LastPostedBy", ScanOperator.Between,
times.ToArray());
            scs.Add(sc);

            var cfg = new DynamoDBOperationConfig
            {
                QueryFilter = scs,
            };

            AsyncSearch<Reply> response = context.QueryAsync<Reply>(forumId,
cfg);
            IEnumerable<Reply> repliesInAPeriod = await
response.GetRemainingAsync();

            foreach (Reply r in repliesInAPeriod)
            {

Console.WriteLine("{r.Id}\t{r.PostedBy}\t{r.Message}\t{r.ReplyDateTime}");
            }
        }

        /// <summary>
        /// Queries the DynamoDB ProductCatalog table for products costing less
        /// than zero.
        /// </summary>
        /// <param name="context">The DynamoDB context object used to perform the
        /// query.</param>
        public static async Task FindProductsPricedLessThanZero(IDynamoDBContext
context)
        {
            int price = 0;

            List<ScanCondition> scs = new List<ScanCondition>();
            var sc1 = new ScanCondition("Price", ScanOperator.LessThan, price);
            var sc2 = new ScanCondition("ProductCategory", ScanOperator.Equal,
"Book");
            scs.Add(sc1);
            scs.Add(sc2);

            AsyncSearch<Book> response = context.ScanAsync<Book>(scs);

            IEnumerable<Book> itemsWithWrongPrice = await
response.GetRemainingAsync();
```

```
            Console.WriteLine("\nFindProductsPricedLessThanZero: Printing
result.....");

            foreach (Book r in itemsWithWrongPrice)
            {
                Console.WriteLine($"{r.Id}\t{r.Title}\t{r.Price}\t{r.Isbn}");
            }
        }
    }
```

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB でアトミックカウンターオペレーションを使用する

次のコード例は、DynamoDB でアトミックカウンタオペレーションを使用する方法を示しています。

- ADD および SET オペレーションを使用してカウンターをアトミックに増分します。
- 存在しない可能性のあるカウンターを安全に増分します。
- カウンターオペレーションに楽観的ロックを実装します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用してアトミックカウンタオペレーションをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
```

```
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Increments a counter using the ADD operation.
     *
     * <p>This method demonstrates how to use the ADD operation to atomically
     * increment a counter attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param incrementValue The value to increment by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse incrementCounterWithAdd(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int incrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("ADD #counterName :increment")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":increment",

 AttributeValue.builder().n(String.valueOf(incrementValue)).build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }
```

```
   /**
    * Increments a counter using the SET operation.
    *
    * <p>This method demonstrates how to use the SET operation with an
expression
    * to increment a counter attribute.
    *
    * @param dynamoDbClient The DynamoDB client
    * @param tableName The name of the DynamoDB table
    * @param key The key of the item to update
    * @param counterName The name of the counter attribute
    * @param incrementValue The value to increment by
    * @return The response from DynamoDB
    * @throws DynamoDbException if an error occurs during the operation
    */
   public static UpdateItemResponse incrementCounterWithSet(
       DynamoDbClient dynamoDbClient,
       String tableName,
       Map<String, AttributeValue> key,
       String counterName,
       int incrementValue) {

       // Define the update parameters
       UpdateItemRequest request = UpdateItemRequest.builder()
           .tableName(tableName)
           .key(key)
           .updateExpression("SET #counterName = #counterName + :increment")
           .expressionAttributeNames(Map.of("#counterName", counterName))
           .expressionAttributeValues(Map.of(
               ":increment",

AttributeValue.builder().n(String.valueOf(incrementValue)).build()))
           .returnValues(ReturnValue.UPDATED_NEW)
           .build();

       // Perform the update operation
       return dynamoDbClient.updateItem(request);
   }

   /**
    * Increments a counter safely, handling the case where the counter doesn't
exist yet.
    *
```

```
     * <p>This method demonstrates how to use if_not_exists to safely increment a
counter
     * that may not exist yet.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param incrementValue The value to increment by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse incrementCounterSafely(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int incrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #counterName =
if_not_exists(#counterName, :zero) + :increment")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":increment",

AttributeValue.builder().n(String.valueOf(incrementValue)).build(),
                ":zero", AttributeValue.builder().n("0").build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Decrements a counter safely, ensuring it doesn't go below zero.
     *
     * <p>This method demonstrates how to use a condition expression to safely
     * decrement a counter without going below zero.
     *
```

```
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param counterName The name of the counter attribute
     * @param decrementValue The value to decrement by
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation or if
the counter would go below zero
     */
    public static UpdateItemResponse decrementCounterSafely(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String counterName,
        int decrementValue) {

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #counterName = #counterName - :decrement")
            .conditionExpression("#counterName >= :decrement")
            .expressionAttributeNames(Map.of("#counterName", counterName))
            .expressionAttributeValues(Map.of(
                ":decrement",

AttributeValue.builder().n(String.valueOf(decrementValue)).build()))
            .returnValues(ReturnValue.UPDATED_NEW)
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Compares the ADD and SET approaches for incrementing counters.
     *
     * <p>This method demonstrates the differences between using ADD and SET
     * for incrementing counters in DynamoDB.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @return Map containing the comparison results
```

```
     */
    public static Map<String, Object> compareAddVsSet(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key) {

        Map<String, Object> results = new HashMap<>();

        try {
            // Reset counters to ensure a fair comparison
            UpdateItemRequest resetRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET AddCounter = :zero, SetCounter = :zero")
                .expressionAttributeValues(
                    Map.of(":zero", AttributeValue.builder().n("0").build()))
                .build();

            dynamoDbClient.updateItem(resetRequest);

            // Increment with ADD
            long addStartTime = System.nanoTime();
            UpdateItemResponse addResponse =
incrementCounterWithAdd(dynamoDbClient, tableName, key, "AddCounter", 1);
            long addEndTime = System.nanoTime();
            long addDuration = addEndTime - addStartTime;

            // Increment with SET
            long setStartTime = System.nanoTime();
            UpdateItemResponse setResponse =
incrementCounterWithSet(dynamoDbClient, tableName, key, "SetCounter", 1);
            long setEndTime = System.nanoTime();
            long setDuration = setEndTime - setStartTime;

            // Record results
            results.put("addResponse", addResponse);
            results.put("setResponse", setResponse);
            results.put("addDuration", addDuration);
            results.put("setDuration", setDuration);
            results.put("success", true);

        } catch (DynamoDbException e) {
            results.put("success", false);
            results.put("error", e.getMessage());
        }
```

```
        return results;
    }

    /**
     * Gets the current value of a counter attribute.
     *
     * <p>Helper method to retrieve the current value of a counter attribute.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to get
     * @param counterName The name of the counter attribute
     * @return The counter value or null if not found
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static Integer getCounterValue(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String counterName) {

        // Define the get parameters
        GetItemRequest request = GetItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .projectionExpression(counterName)
            .build();

        // Perform the get operation
        GetItemResponse response = dynamoDbClient.getItem(request);

        // Return the counter value if it exists, otherwise null
        if (response.item() != null && response.item().containsKey(counterName))
{
            return Integer.parseInt(response.item().get(counterName).n());
        }

        return null;
    }
```

AWS SDK for Java 2.x を使用したアトミックカウンタオペレーションの使用例。

```
   public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
       // Example key
       Map<String, AttributeValue> key = new HashMap<>();
       key.put("ProductId", AttributeValue.builder().s("P12345").build());

       System.out.println("Demonstrating atomic counter operations in
DynamoDB");

       try {
           // Example 1: Increment a counter using ADD
           System.out.println("\nExample 1: Incrementing a counter using ADD");
           UpdateItemResponse addResponse =
incrementCounterWithAdd(dynamoDbClient, tableName, key, "ViewCount", 1);

           System.out.println("Updated counter: " + addResponse.attributes());

           // Example 2: Increment a counter using SET
           System.out.println("\nExample 2: Incrementing a counter using SET");
           UpdateItemResponse setResponse =
incrementCounterWithSet(dynamoDbClient, tableName, key, "LikeCount", 1);

           System.out.println("Updated counter: " + setResponse.attributes());

           // Example 3: Increment a counter safely
           System.out.println("\nExample 3: Incrementing a counter safely");
           UpdateItemResponse safeResponse =
incrementCounterSafely(dynamoDbClient, tableName, key, "ShareCount", 1);

           System.out.println("Updated counter: " + safeResponse.attributes());

           // Example 4: Decrement a counter safely
           System.out.println("\nExample 4: Decrementing a counter safely");
           try {
               UpdateItemResponse decrementResponse =
                   decrementCounterSafely(dynamoDbClient, tableName, key,
"InventoryCount", 1);

               System.out.println("Updated counter: " +
decrementResponse.attributes());
           } catch (DynamoDbException e) {
               if (e.getMessage().contains("ConditionalCheckFailed")) {
                   System.out.println("Cannot decrement counter below zero");
```

```
                } else {
                    throw e;
                }
            }

            // Example 5: Compare ADD vs SET
            System.out.println("\nExample 5: Comparing ADD vs SET");
            Map<String, Object> comparison = compareAddVsSet(dynamoDbClient,
tableName, key);

            if ((boolean) comparison.get("success")) {
                System.out.println("ADD duration: " +
comparison.get("addDuration") + " ns");
                System.out.println("SET duration: " +
comparison.get("setDuration") + " ns");
                System.out.println("ADD response: " +
comparison.get("addResponse"));
                System.out.println("SET response: " +
comparison.get("setResponse"));
            } else {
                System.out.println("Comparison failed: " +
comparison.get("error"));
            }

            // Explain atomic counter operations
            System.out.println("\nKey points about DynamoDB atomic counter
operations:");
            System.out.println("1. Both ADD and SET can be used for atomic
counters");
            System.out.println("2. ADD is more concise for simple increments");
            System.out.println("3. SET with an expression is more flexible for
complex operations");
            System.out.println("4. Use if_not_exists to handle the case where the
counter doesn't exist yet");
            System.out.println("5. Use condition expressions to prevent counters
from going below zero");
            System.out.println("6. Atomic operations are guaranteed to be
isolated from other writes");
            System.out.println("7. ADD can only be used with number and set data
types");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
```

```
        }
    }
```

- API の詳細については、AWS SDK for Java 2.x API リファレンスの「UpdateItem」を参照してください。

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用してアトミックカウンタオペレーションをデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Increment a counter using the ADD operation.
 *
 * This function demonstrates using the ADD operation for atomic increments.
 * The ADD operation is atomic and is the recommended way to increment counters.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithAdd(
  config,
  tableName,
  key,
  counterName,
  incrementValue
) {
  // Initialize the DynamoDB client
```

```
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using ADD
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `ADD ${counterName} :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter using the SET operation with an expression.
 *
 * This function demonstrates using the SET operation with an expression for
 increments.
 * While this approach works, it's less idiomatic for simple increments than
 using ADD.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithSet(
  config,
  tableName,
  key,
  counterName,
  incrementValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
```

```
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with an expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} = ${counterName} + :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter safely, handling the case where the counter might not
 exist.
 *
 * This function demonstrates using the if_not_exists function with SET to safely
 * increment a counter that might not exist yet.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @param {number} defaultValue - The default value if the counter doesn't exist
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterSafely(
  config,
  tableName,
  key,
  counterName,
  incrementValue,
  defaultValue = 0
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
```

```
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using SET with if_not_exists
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} =
 if_not_exists(${counterName}, :default) + :increment`,
    ExpressionAttributeValues: {
      ":increment": incrementValue,
      ":default": defaultValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Increment a counter with optimistic locking to prevent race conditions.
 *
 * This function demonstrates using a condition expression to implement
 optimistic
 * locking, which prevents race conditions when multiple processes try to update
 * the same counter.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} counterName - The name of the counter attribute
 * @param {number} incrementValue - The value to increment by
 * @param {number} expectedValue - The expected current value of the counter
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function incrementCounterWithLocking(
  config,
  tableName,
  key,
  counterName,
  incrementValue,
  expectedValue
```

```
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${counterName} = ${counterName} + :increment`,
    ConditionExpression: `${counterName} = :expected`,
    ExpressionAttributeValues: {
      ":increment": incrementValue,
      ":expected": expectedValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));
    return {
      success: true,
      data: response
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        error: "Optimistic locking failed: the counter value has changed"
      };
    }
    // Re-throw other errors
    throw error;
  }
}

/**
 * Get the current value of a counter.
 *
 * Helper function to retrieve the current value of a counter attribute.
 *
 * @param {Object} config - AWS configuration object
```

```
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @param {string} counterName - The name of the counter attribute
 * @returns {Promise<number|null>} - The current counter value or null if not
 found
 */
async function getCounterValue(
  config,
  tableName,
  key,
  counterName
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the counter value if it exists, otherwise null
  return response.Item && counterName in response.Item
    ? response.Item[counterName]
    : null;
}
```

AWS SDK for JavaScript を使用したアトミックカウンタオペレーションの使用例。

```
/**
 * Example of how to use the atomic counter operations.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };
  const counterName = "ViewCount";
```

```
 const incrementValue = 1;

 console.log("Demonstrating different approaches to increment counters in
DynamoDB");

 try {
   // Example 1: Using ADD operation (recommended for simple increments)
   console.log("\nExample 1: Incrementing counter with ADD operation");
   const response1 = await incrementCounterWithAdd(
     config,
     tableName,
     key,
     counterName,
     incrementValue
   );

   console.log(`Counter incremented to: ${response1.Attributes[counterName]}`);

   // Example 2: Using SET operation with an expression
   console.log("\nExample 2: Incrementing counter with SET operation");
   const response2 = await incrementCounterWithSet(
     config,
     tableName,
     key,
     counterName,
     incrementValue
   );

   console.log(`Counter incremented to: ${response2.Attributes[counterName]}`);

   // Example 3: Safely incrementing a counter that might not exist
   console.log("\nExample 3: Safely incrementing counter that might not exist");
   const newKey = { ProductId: "P67890" };
   const response3 = await incrementCounterSafely(
     config,
     tableName,
     newKey,
     counterName,
     incrementValue,
     0
   );

   console.log(`Counter initialized and incremented to:
${response3.Attributes[counterName]}`);
```

```
    // Example 4: Incrementing with optimistic locking
    console.log("\nExample 4: Incrementing with optimistic locking");

    // First, get the current counter value
    const currentValue = await getCounterValue(config, tableName, key,
 counterName);
    console.log(`Current counter value: ${currentValue}`);

    // Then, try to increment with optimistic locking
    const response4 = await incrementCounterWithLocking(
      config,
      tableName,
      key,
      counterName,
      incrementValue,
      currentValue
    );

    if (response4.success) {
      console.log(`Counter successfully incremented to:
 ${response4.data.Attributes[counterName]}`);
    } else {
      console.log(response4.error);
    }

    // Explain the differences between ADD and SET
    console.log("\nKey differences between ADD and SET for counter operations:");
    console.log("1. ADD is more concise and idiomatic for simple increments");
    console.log("2. SET with expressions is more flexible for complex
 operations");
    console.log("3. Both operations are atomic and safe for concurrent updates");
    console.log("4. SET with if_not_exists is required when the attribute might
 not exist");
    console.log("5. Optimistic locking can be added to either approach for
 additional safety");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、AWS SDK for JavaScript API リファレンスの「UpdateItem」を参照してください。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用してアトミックカウンタオペレーションをデモンストレーションします。

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, Union


def increment_counter_with_add(
    table_name: str, key: Dict[str, Any], counter_name: str, increment_value: int
 = 1
) -> Dict[str, Any]:
    """
    Increment a counter attribute using the ADD operation.

    This function demonstrates the atomic ADD operation, which is ideal for
    incrementing counters without the risk of race conditions.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the ADD operation to atomically increment the counter
    response = table.update_item(
        Key=key,
```

```
        UpdateExpression="ADD #counter :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def increment_counter_with_set(
    table_name: str, key: Dict[str, Any], counter_name: str, increment_value: int
 = 1
) -> Dict[str, Any]:
    """
    Increment a counter attribute using the SET operation with an expression.

    This function demonstrates using SET with an expression to increment a
 counter.
    While this works, it's generally recommended to use ADD for simple
 increments.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use the SET operation with an expression to increment the counter
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #counter = #counter + :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value},
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def increment_counter_safely(
    table_name: str,
    key: Dict[str, Any],
    counter_name: str,
    increment_value: int = 1,
    initial_value: int = 0,
) -> Dict[str, Any]:
    """
    Increment a counter attribute safely, handling the case where it might not
 exist.

    This function demonstrates a best practice for incrementing counters by using
    the if_not_exists function to handle the case where the counter doesn't exist
 yet.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        increment_value (int, optional): The value to increment by. Defaults to
 1.
        initial_value (int, optional): The initial value if the counter doesn't
 exist. Defaults to 0.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use SET with if_not_exists to safely increment the counter
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #counter = if_not_exists(#counter, :initial)
 + :increment",
        ExpressionAttributeNames={"#counter": counter_name},
        ExpressionAttributeValues={":increment": increment_value, ":initial":
 initial_value},
```

```
        ReturnValues="UPDATED_NEW",
    )

    return response


def atomic_conditional_increment(
    table_name: str,
    key: Dict[str, Any],
    counter_name: str,
    condition_attribute: str,
    condition_value: Any,
    increment_value: int = 1,
) -> Union[Dict[str, Any], None]:
    """
    Atomically increment a counter only if a condition is met.

    This function demonstrates combining atomic counter operations with
    conditional expressions for more complex update scenarios.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        counter_name (str): The name of the counter attribute.
        condition_attribute (str): The attribute to check in the condition.
        condition_value (Any): The value to compare against.
        increment_value (int, optional): The value to increment by. Defaults to
 1.

    Returns:
        Optional[Dict[str, Any]]: The response from DynamoDB if successful, None
 if condition failed.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        # Use ADD with a condition expression
        response = table.update_item(
            Key=key,
            UpdateExpression="ADD #counter :increment",
            ConditionExpression="#condition = :value",
```

```
            ExpressionAttributeNames={"#counter": counter_name, "#condition":
 condition_attribute},
            ExpressionAttributeValues={":increment": increment_value, ":value":
 condition_value},
            ReturnValues="UPDATED_NEW",
        )
        return response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return None
        else:
            # Other error occurred
            raise
```

AWS SDK for Python (Boto3) を使用したアトミックカウンタオペレーションの使用例。

```
def example_usage():
    """Example of how to use the atomic counter operations functions."""
    # Example parameters
    table_name = "GameScores"
    key = {"UserId": "user123", "GameId": "game456"}
    counter_name = "Score"

    print("Example 1: Incrementing a counter with ADD operation")
    try:
        response = increment_counter_with_add(
            table_name=table_name, key=key, counter_name=counter_name,
 increment_value=10
        )
        print(
            f"Counter incremented successfully. New value:
 {response.get('Attributes', {}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error incrementing counter with ADD: {e}")

    print("\nExample 2: Incrementing a counter with SET operation")
    try:
        response = increment_counter_with_set(
```

```
            table_name=table_name, key=key, counter_name=counter_name,
increment_value=5
        )
        print(
            f"Counter incremented successfully. New value:
{response.get('Attributes', {}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error incrementing counter with SET: {e}")

    print("\nExample 3: Safely incrementing a counter that might not exist")
    try:
        new_key = {"UserId": "newuser789", "GameId": "game456"}
        response = increment_counter_safely(
            table_name=table_name,
            key=new_key,
            counter_name=counter_name,
            increment_value=15,
            initial_value=100,
        )
        print(
            f"Counter safely incremented. New value: {response.get('Attributes',
{}).get(counter_name)}"
        )
    except Exception as e:
        print(f"Error safely incrementing counter: {e}")

    print("\nExample 4: Conditional counter increment")
    try:
        # Fix for mypy: Handle the case where response might be None
        result = atomic_conditional_increment(
            table_name=table_name,
            key=key,
            counter_name="Achievements",
            condition_attribute="Level",
            condition_value=5,
            increment_value=1,
        )

        if result is not None:
            print(
                f"Conditional increment succeeded. New value:
{result.get('Attributes', {}).get('Achievements')}"
            )
```

```
        else:
            print("Conditional increment failed because condition was not met.")
        if response:
            print(
                f"Conditional increment succeeded. New value:
{response.get('Attributes', {}).get('Achievements')}"
            )
        else:
            print("Conditional increment failed because condition was not met.")
    except Exception as e:
        print(f"Error with conditional increment: {e}")

    print("\nComparison of ADD vs SET for counter operations:")
    print("1. ADD is specifically designed for atomic numeric increments and set
operations")
    print("2. SET with an expression can be used for more complex calculations")
    print("3. Both operations are atomic, preventing race conditions")
    print("4. ADD is more concise for simple increments")
    print("5. SET with if_not_exists() is recommended when the attribute might
not exist")
    print("6. For counters, ADD is generally preferred for clarity and
simplicity")
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateItem」を参照してください。

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB で条件付きオペレーションを使用する

次のコード例は、DynamoDB で条件付きオペレーションを使用する方法を示しています。

- データの上書きを防ぐため、条件付き書き込みを実装します。
- 条件式を使用してビジネスルールを適用します。
- 条件付きチェックの失敗を適切に処理します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して条件付きオペレーションをデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import
 software.amazon.awssdk.services.dynamodb.model.ConditionalCheckFailedException;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemRequest;
import software.amazon.awssdk.services.dynamodb.model.DeleteItemResponse;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.GetItemResponse;
import software.amazon.awssdk.services.dynamodb.model.ReturnValue;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.HashMap;
import java.util.Map;

    /**
     * Performs a conditional update on an item.
     *
     * <p>This method demonstrates how to use a condition expression to update an
 item
     * only if a specific condition is met.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param conditionAttribute The attribute to check in the condition
     * @param conditionValue The value to compare against
     * @param updateAttribute The attribute to update
     * @param updateValue The new value to set
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalUpdate(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String conditionAttribute,
        AttributeValue conditionValue,
```

```
        String updateAttribute,
        AttributeValue updateValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #updateAttr = :updateVal")
                .conditionExpression("#condAttr = :condVal")
                .expressionAttributeNames(Map.of(
                    "#condAttr", conditionAttribute,
                    "#updateAttr", updateAttribute))
                .expressionAttributeValues(Map.of(
                    ":condVal", conditionValue,
                    ":updateVal", updateValue))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            // Perform the update operation
            UpdateItemResponse response = dynamoDbClient.updateItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "Condition was met and update was performed");
            result.put("attributes", response.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "Condition was not met, update was not
performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
```

```
    }

    /**
     * Performs a conditional delete on an item.
     *
     * <p>This method demonstrates how to use a condition expression to delete an
item
     * only if a specific condition is met.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to delete
     * @param conditionAttribute The attribute to check in the condition
     * @param conditionValue The value to compare against
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalDelete(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String conditionAttribute,
        AttributeValue conditionValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Define the delete parameters
            DeleteItemRequest request = DeleteItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .conditionExpression("#condAttr = :condVal")
                .expressionAttributeNames(Map.of("#condAttr",
conditionAttribute))
                .expressionAttributeValues(Map.of(":condVal", conditionValue))
                .returnValues(ReturnValue.ALL_OLD)
                .build();

            // Perform the delete operation
            DeleteItemResponse response = dynamoDbClient.deleteItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "Condition was met and delete was performed");
            result.put("attributes", response.attributes());
```

```
        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "Condition was not met, delete was not
performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }

    /**
     * Demonstrates optimistic locking using a version attribute.
     *
     * <p>This method shows how to implement optimistic locking by using a
version
     * attribute that is incremented with each update.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param versionAttribute The name of the version attribute
     * @return Map containing the operation result
     */
    public static Map<String, Object> optimisticLockingExample(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> key, String versionAttribute) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Get the current version of the item
            GetItemRequest getRequest = GetItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .projectionExpression(versionAttribute)
                .build();
```

```
            GetItemResponse getResponse = dynamoDbClient.getItem(getRequest);

            // Check if the item exists
            if (getResponse.item() == null || !
getResponse.item().containsKey(versionAttribute)) {
                // Item doesn't exist or doesn't have a version attribute
                // Initialize with version 1
                UpdateItemRequest initRequest = UpdateItemRequest.builder()
                    .tableName(tableName)
                    .key(key)
                    .updateExpression("SET #verAttr = :newVer, #dataAttr
 = :data")
                    .expressionAttributeNames(Map.of("#verAttr",
 versionAttribute, "#dataAttr", "Data"))
                    .expressionAttributeValues(Map.of(
                        ":newVer", AttributeValue.builder().n("1").build(),
                        ":data", AttributeValue.builder().s("Initial
 data").build()))
                    .returnValues(ReturnValue.UPDATED_NEW)
                    .build();

                UpdateItemResponse initResponse =
 dynamoDbClient.updateItem(initRequest);

                result.put("operation", "initialize");
                result.put("success", true);
                result.put("attributes", initResponse.attributes());

                return result;
            }

            // Get the current version number
            int currentVersion =
                Integer.parseInt(getResponse.item().get(versionAttribute).n());
            int newVersion = currentVersion + 1;

            // Update the item with a condition on the version
            UpdateItemRequest updateRequest = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #verAttr = :newVer, #dataAttr = :newData")
                .conditionExpression("#verAttr = :curVer")
```

```
                .expressionAttributeNames(Map.of("#verAttr", versionAttribute,
"#dataAttr", "Data"))
                .expressionAttributeValues(Map.of(
                    ":curVer",
                        AttributeValue.builder()
                            .n(String.valueOf(currentVersion))
                            .build(),
                    ":newVer",

AttributeValue.builder().n(String.valueOf(newVersion)).build(),
                    ":newData",
                        AttributeValue.builder()
                            .s("Updated data at version " + newVersion)
                            .build()))
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            UpdateItemResponse updateResponse =
dynamoDbClient.updateItem(updateRequest);

            // Record success result
            result.put("operation", "update");
            result.put("success", true);
            result.put("oldVersion", currentVersion);
            result.put("newVersion", newVersion);
            result.put("attributes", updateResponse.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to version mismatch
            result.put("operation", "update");
            result.put("success", false);
            result.put("message", "Version mismatch, another process may have
updated the item");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
            // Record failure due to other errors
            result.put("operation", "update");
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
```

```
    }

    /**
     * Performs a conditional update with multiple conditions.
     *
     * <p>This method demonstrates how to use multiple conditions in a condition
expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param conditions Map of attribute names to values for conditions
     * @param updateAttribute The attribute to update
     * @param updateValue The new value to set
     * @return Map containing the operation result and status
     */
    public static Map<String, Object> conditionalUpdateWithMultipleConditions(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        Map<String, AttributeValue> conditions,
        String updateAttribute,
        AttributeValue updateValue) {

        Map<String, Object> result = new HashMap<>();

        try {
            // Build the condition expression and attribute names/values
            StringBuilder conditionExpression = new StringBuilder();
            Map<String, String> expressionAttributeNames = new HashMap<>();
            Map<String, AttributeValue> expressionAttributeValues = new
HashMap<>();

            // Add update attribute
            expressionAttributeNames.put("#updateAttr", updateAttribute);
            expressionAttributeValues.put(":updateVal", updateValue);

            // Add conditions
            int i = 0;
            for (Map.Entry<String, AttributeValue> condition :
conditions.entrySet()) {
                String attrName = condition.getKey();
                AttributeValue attrValue = condition.getValue();
```

```
                String nameKey = "#cond" + i;
                String valueKey = ":val" + i;

                expressionAttributeNames.put(nameKey, attrName);
                expressionAttributeValues.put(valueKey, attrValue);

                // Add AND between conditions (except for the first one)
                if (i > 0) {
                    conditionExpression.append(" AND ");
                }

                conditionExpression.append(nameKey).append(" =
").append(valueKey);
                i++;
            }

            // Define the update parameters
            UpdateItemRequest request = UpdateItemRequest.builder()
                .tableName(tableName)
                .key(key)
                .updateExpression("SET #updateAttr = :updateVal")
                .conditionExpression(conditionExpression.toString())
                .expressionAttributeNames(expressionAttributeNames)
                .expressionAttributeValues(expressionAttributeValues)
                .returnValues(ReturnValue.UPDATED_NEW)
                .build();

            // Perform the update operation
            UpdateItemResponse response = dynamoDbClient.updateItem(request);

            // Record success result
            result.put("success", true);
            result.put("message", "All conditions were met and update was
performed");
            result.put("attributes", response.attributes());

        } catch (ConditionalCheckFailedException e) {
            // Record failure due to condition not being met
            result.put("success", false);
            result.put("message", "One or more conditions were not met, update
was not performed");
            result.put("error", "ConditionalCheckFailedException");

        } catch (DynamoDbException e) {
```

```
            // Record failure due to other errors
            result.put("success", false);
            result.put("message", "Error occurred: " + e.getMessage());
            result.put("error", e.getClass().getSimpleName());
        }

        return result;
    }
```

AWS SDK for Java 2.x を使用した条件付きオペレーションの使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating conditional operations in DynamoDB");

        try {
            // Example 1: Conditional update
            System.out.println("\nExample 1: Conditional update");
            Map<String, Object> updateResult = conditionalUpdate(
                dynamoDbClient,
                tableName,
                key,
                "InStock",
                AttributeValue.builder().bool(true).build(),
                "Status",
                AttributeValue.builder().s("Available").build());

            System.out.println("Update result: " + updateResult.get("message"));
            if ((boolean) updateResult.get("success")) {
                System.out.println("Updated attributes: " +
updateResult.get("attributes"));
            }

            // Example 2: Conditional delete
            System.out.println("\nExample 2: Conditional delete");
            Map<String, Object> deleteResult = conditionalDelete(
                dynamoDbClient,
                tableName,
```

```
                key,
                "Status",
                AttributeValue.builder().s("Discontinued").build());

            System.out.println("Delete result: " + deleteResult.get("message"));
            if ((boolean) deleteResult.get("success")) {
                System.out.println("Deleted item: " +
deleteResult.get("attributes"));
            }

            // Example 3: Optimistic locking
            System.out.println("\nExample 3: Optimistic locking");
            Map<String, Object> lockingResult =
optimisticLockingExample(dynamoDbClient, tableName, key, "Version");

            System.out.println("Optimistic locking result:");
            System.out.println("  Operation: " + lockingResult.get("operation"));
            System.out.println("  Success: " + lockingResult.get("success"));
            if (lockingResult.get("operation").equals("update") && (boolean)
lockingResult.get("success")) {
                System.out.println("  Old version: " +
lockingResult.get("oldVersion"));
                System.out.println("  New version: " +
lockingResult.get("newVersion"));
            }
            System.out.println("  Attributes: " +
lockingResult.get("attributes"));

            // Example 4: Multiple conditions
            System.out.println("\nExample 4: Multiple conditions");
            Map<String, AttributeValue> conditions = new HashMap<>();
            conditions.put("Price",
AttributeValue.builder().n("199.99").build());
            conditions.put("Category",
AttributeValue.builder().s("Electronics").build());

            Map<String, Object> multiConditionResult =
conditionalUpdateWithMultipleConditions(
                dynamoDbClient,
                tableName,
                key,
                conditions,
                "OnSale",
                AttributeValue.builder().bool(true).build());
```

```
            System.out.println("Multiple conditions result: " +
multiConditionResult.get("message"));
            if ((boolean) multiConditionResult.get("success")) {
                System.out.println("Updated attributes: " +
multiConditionResult.get("attributes"));
            }

            // Explain conditional operations
            System.out.println("\nKey points about DynamoDB conditional
operations:");
            System.out.println("1. Conditional operations only succeed if the
condition is met");
            System.out.println("2. ConditionalCheckFailedException is thrown when
the condition fails");
            System.out.println("3. No changes are made to the item if the
condition fails");
            System.out.println("4. Conditions can be used with update, delete,
and put operations");
            System.out.println("5. Multiple conditions can be combined with AND
and OR");
            System.out.println("6. Optimistic locking can be implemented using a
version attribute");
            System.out.println(
                "7. Conditional operations consume the same amount of write
capacity whether they succeed or fail");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - DeleteItem
  - PutItem
  - UpdateItem

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して条件付きオペレーションをデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  DeleteCommand,
  GetCommand,
  PutCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Perform a conditional update operation.
 *
 * This function demonstrates how to update an item only if a condition is met.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} conditionAttribute - The attribute to check in the condition
 * @param {any} conditionValue - The value to compare against
 * @param {string} updateAttribute - The attribute to update
 * @param {any} updateValue - The new value to set
 * @returns {Promise<Object>} - Result of the operation
 */
async function conditionalUpdate(
  config,
  tableName,
  key,
  conditionAttribute,
  conditionValue,
  updateAttribute,
  updateValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters with a condition expression
```

```
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${updateAttribute} = :value`,
    ConditionExpression: `${conditionAttribute} = :condition`,
    ExpressionAttributeValues: {
      ":value": updateValue,
      ":condition": conditionValue
    },
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Condition was met and update was performed",
      updatedAttributes: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Condition was not met, update was not performed",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Perform a conditional delete operation.
 *
 * This function demonstrates how to delete an item only if a condition is met.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to delete
```

```
 * @param {string} conditionAttribute - The attribute to check in the condition
 * @param {any} conditionValue - The value to compare against
 * @returns {Promise<Object>} - Result of the operation
 */
async function conditionalDelete(
  config,
  tableName,
  key,
  conditionAttribute,
  conditionValue
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the delete parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    ConditionExpression: `${conditionAttribute} = :condition`,
    ExpressionAttributeValues: {
      ":condition": conditionValue
    },
    ReturnValues: "ALL_OLD"
  };

  try {
    // Perform the delete operation
    const response = await docClient.send(new DeleteCommand(params));

    return {
      success: true,
      message: "Condition was met and item was deleted",
      deletedItem: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Condition was not met, item was not deleted",
        error: "ConditionalCheckFailedException"
      };
    }
```

```
    // Re-throw other errors
    throw error;
  }
}

/**
 * Implement optimistic locking with a version number.
 *
 * This function demonstrates how to use a version number for optimistic locking
 * to prevent race conditions when multiple processes update the same item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {Object} updates - The attributes to update
 * @param {number} expectedVersion - The expected current version number
 * @returns {Promise<Object>} - Result of the operation
 */
async function updateWithOptimisticLocking(
  config,
  tableName,
  key,
  updates,
  expectedVersion
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Build the update expression
  const updateExpressions = [];
  const expressionAttributeValues = {
    ":expectedVersion": expectedVersion,
    ":newVersion": expectedVersion + 1
  };

  // Add each update to the expression
  Object.entries(updates).forEach(([attribute, value], index) => {
    updateExpressions.push(`${attribute} = :val${index}`);
    expressionAttributeValues[`:val${index}`] = value;
  });

  // Add the version update
```

```
  updateExpressions.push("version = :newVersion");

  // Define the update parameters with a condition expression
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${updateExpressions.join(", ")}`,
    ConditionExpression: "version = :expectedVersion",
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: "UPDATED_NEW"
  };

  try {
    // Perform the update operation
    const response = await docClient.send(new UpdateCommand(params));

    return {
      success: true,
      message: "Update succeeded with optimistic locking",
      newVersion: expectedVersion + 1,
      updatedAttributes: response.Attributes
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Optimistic locking failed: the item was modified by another
 process",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Implement a conditional write that creates an item only if it doesn't exist.
 *
 * This function demonstrates how to use attribute_not_exists to create an item
 * only if it doesn't already exist (similar to an "INSERT IF NOT EXISTS"
 operation).
```

```
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} item - The item to create
 * @returns {Promise<Object>} - Result of the operation
 */
async function createIfNotExists(
  config,
  tableName,
  item
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Extract the primary key attributes
  const keyAttributes = Object.keys(item).filter(attr =>
    attr === "id" || attr === "ID" || attr === "Id" ||
    attr.endsWith("Id") || attr.endsWith("ID") ||
    attr.endsWith("Key")
  );

  if (keyAttributes.length === 0) {
    throw new Error("Could not determine primary key attributes");
  }

  // Create a condition expression that checks if the item doesn't exist
  const conditionExpression = `attribute_not_exists(${keyAttributes[0]})`;

  // Define the put parameters with a condition expression
  const params = {
    TableName: tableName,
    Item: item,
    ConditionExpression: conditionExpression
  };

  try {
    // Perform the put operation
    await docClient.send(new PutCommand(params));

    return {
      success: true,
      message: "Item was created because it didn't exist",
      item
```

```
    };
  } catch (error) {
    // Check if the error is due to the condition check failing
    if (error.name === "ConditionalCheckFailedException") {
      return {
        success: false,
        message: "Item already exists, creation was skipped",
        error: "ConditionalCheckFailedException"
      };
    }

    // Re-throw other errors
    throw error;
  }
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));
```

```
  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

AWS SDK for JavaScript を使用した条件付きオペレーションの使用例。

```
/**
 * Example of how to use conditional operations.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating conditional operations in DynamoDB");

  try {
    // Example 1: Conditional update based on attribute value
    console.log("\nExample 1: Conditional update based on attribute value");
    const updateResult = await conditionalUpdate(
      config,
      tableName,
      key,
      "Category",
      "Electronics",
      "Price",
      299.99
    );

    console.log(`Result: ${updateResult.message}`);
    if (updateResult.success) {
      console.log("Updated attributes:", updateResult.updatedAttributes);
    }

    // Example 2: Conditional delete based on attribute value
    console.log("\nExample 2: Conditional delete based on attribute value");
    const deleteResult = await conditionalDelete(
      config,
      tableName,
      key,
      "InStock",
```

```
      false
    );

    console.log(`Result: ${deleteResult.message}`);
    if (deleteResult.success) {
      console.log("Deleted item:", deleteResult.deletedItem);
    }

    // Example 3: Optimistic locking with version number
    console.log("\nExample 3: Optimistic locking with version number");

    // First, get the current item to check its version
    const currentItem = await getItem(config, tableName, { ProductId:
"P67890" });
    const currentVersion = currentItem ? (currentItem.version || 0) : 0;

    console.log(`Current version: ${currentVersion}`);

    // Then, update with optimistic locking
    const lockingResult = await updateWithOptimisticLocking(
      config,
      tableName,
      { ProductId: "P67890" },
      {
        Name: "Updated Product Name",
        Description: "This is an updated description"
      },
      currentVersion
    );

    console.log(`Result: ${lockingResult.message}`);
    if (lockingResult.success) {
      console.log(`New version: ${lockingResult.newVersion}`);
      console.log("Updated attributes:", lockingResult.updatedAttributes);
    }

    // Example 4: Create item only if it doesn't exist
    console.log("\nExample 4: Create item only if it doesn't exist");
    const createResult = await createIfNotExists(
      config,
      tableName,
      {
        ProductId: "P99999",
        Name: "New Product",
```

```
        Category: "Accessories",
        Price: 19.99,
        InStock: true
      }
    );

    console.log(`Result: ${createResult.message}`);
    if (createResult.success) {
      console.log("Created item:", createResult.item);
    }

    // Explain conditional operations
    console.log("\nKey points about conditional operations:");
    console.log("1. Conditional operations only succeed if the condition is
 met");
    console.log("2. ConditionalCheckFailedException indicates the condition
 wasn't met");
    console.log("3. Optimistic locking prevents race conditions in concurrent
 updates");
    console.log("4. attribute_exists and attribute_not_exists are useful for
 checking if attributes are present");
    console.log("5. Conditional operations are atomic - they either succeed
 completely or fail completely");
    console.log("6. You can use any valid comparison operators and functions in
 condition expressions");
    console.log("7. Conditional operations don't consume write capacity if the
 condition fails");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - DeleteItem
  - PutItem
  - UpdateItem

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して条件付きオペレーションをデモンストレーションします。

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, Optional, Tuple, Union


def conditional_update(
    table_name: str,
    key: Dict[str, Any],
    condition_attribute: str,
    condition_value: Any,
    update_attribute: str,
    update_value: Any,
) -> Tuple[bool, Optional[Dict[str, Any]]]:
    """
    Update an item only if a condition is met.

    This function demonstrates how to perform a conditional update operation
    and determine if the condition was met.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        condition_attribute (str): The attribute to check in the condition.
        condition_value (Any): The value to compare against.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.

    Returns:
        Tuple[bool, Optional[Dict[str, Any]]]: A tuple containing:
            - A boolean indicating if the update succeeded
            - The response from DynamoDB if successful, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
```

```
        # Perform the conditional update
        response = table.update_item(
            Key=key,
            UpdateExpression="SET #update_attr = :update_val",
            ConditionExpression="#cond_attr = :cond_val",
            ExpressionAttributeNames={
                "#update_attr": update_attribute,
                "#cond_attr": condition_attribute,
            },
            ExpressionAttributeValues={":update_val": update_value, ":cond_val":
 condition_value},
            ReturnValues="UPDATED_NEW",
        )
        # Update succeeded, condition was met
        return True, response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return False, None
        else:
            # Other error occurred
            raise


def conditional_delete(
    table_name: str, key: Dict[str, Any], condition_attribute: str,
 condition_value: Any
) -> bool:
    """
    Delete an item only if a condition is met.

    This function demonstrates how to perform a conditional delete operation
    and determine if the condition was met.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to delete.
        condition_attribute (str): The attribute to check in the condition.
        condition_value (Any): The value to compare against.

    Returns:
        bool: True if the delete succeeded (condition was met), False otherwise.
    """
    # Initialize the DynamoDB resource
```

```
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        # Perform the conditional delete
        table.delete_item(
            Key=key,
            ConditionExpression="#attr = :val",
            ExpressionAttributeNames={"#attr": condition_attribute},
            ExpressionAttributeValues={":val": condition_value},
        )
        # Delete succeeded, condition was met
        return True
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return False
        else:
            # Other error occurred
            raise


def optimistic_locking_update(
    table_name: str,
    key: Dict[str, Any],
    version_attribute: str,
    update_attribute: str,
    update_value: Any,
) -> Tuple[bool, Optional[Dict[str, Any]]]:
    """
    Update an item using optimistic locking with a version attribute.

    This function demonstrates how to implement optimistic locking using
    a version attribute that is incremented with each update.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        version_attribute (str): The name of the version attribute.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.

    Returns:
        Tuple[bool, Optional[Dict[str, Any]]]: A tuple containing:
```

```
            - A boolean indicating if the update succeeded
            - The response from DynamoDB if successful, None otherwise
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # First, get the current version
    try:
        response = table.get_item(
            Key=key,
            ProjectionExpression=f"#{version_attribute}",
            ExpressionAttributeNames={f"#{version_attribute}":
version_attribute},
        )

        item = response.get("Item", {})
        current_version = item.get(version_attribute, 0)

        # Now, try to update with a condition on the version
        try:
            update_response = table.update_item(
                Key=key,
                UpdateExpression=f"SET #{update_attribute} = :update_val,
#{version_attribute} = :new_version",
                ConditionExpression=f"#{version_attribute} = :current_version",
                ExpressionAttributeNames={
                    f"#{update_attribute}": update_attribute,
                    f"#{version_attribute}": version_attribute,
                },
                ExpressionAttributeValues={
                    ":update_val": update_value,
                    ":current_version": current_version,
                    ":new_version": current_version + 1,
                },
                ReturnValues="UPDATED_NEW",
            )
            # Update succeeded
            return True, update_response
        except ClientError as e:
            if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
                # Version has changed, optimistic locking failed
                return False, None
            else:
```

```
                # Other error occurred
                raise
    except ClientError:
        # Error getting the item
        raise


def conditional_check_and_update(
    table_name: str,
    key: Dict[str, Any],
    check_attribute: str,
    check_value: Any,
    update_attribute: str,
    update_value: Any,
    create_if_not_exists: bool = False,
) -> Union[Dict[str, Any], None]:
    """
    Check if an attribute has a specific value and update another attribute if it
 does.

    This function demonstrates a more complex conditional update that can also
    create the item if it doesn't exist.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        check_attribute (str): The attribute to check in the condition.
        check_value (Any): The value to compare against.
        update_attribute (str): The attribute to update.
        update_value (Any): The new value to set.
        create_if_not_exists (bool, optional): Whether to create the item if it
 doesn't exist.

    Returns:
        Union[Dict[str, Any], None]: The response from DynamoDB if successful,
 None otherwise.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    try:
        if create_if_not_exists:
            # Use attribute_not_exists to create the item if it doesn't exist
```

```
            condition_expression = "attribute_not_exists(#pk) OR #check_attr
= :check_val"
            update_expression = "SET #update_attr = :update_val, #check_attr =
if_not_exists(#check_attr, :check_val)"

            # Get the partition key name from the key dictionary
            pk_name = next(iter(key))

            expression_attribute_names = {
                "#pk": pk_name,
                "#check_attr": check_attribute,
                "#update_attr": update_attribute,
            }
        else:
            # Only update if the check attribute has the expected value
            condition_expression = "#check_attr = :check_val"
            update_expression = "SET #update_attr = :update_val"

            expression_attribute_names = {
                "#check_attr": check_attribute,
                "#update_attr": update_attribute,
            }

        # Perform the conditional update
        response = table.update_item(
            Key=key,
            UpdateExpression=update_expression,
            ConditionExpression=condition_expression,
            ExpressionAttributeNames=expression_attribute_names,
            ExpressionAttributeValues={":check_val": check_value, ":update_val":
update_value},
            ReturnValues="UPDATED_NEW",
        )
        return response
    except ClientError as e:
        if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
            # Condition was not met
            return None
        else:
            # Other error occurred
            raise
```

AWS SDK for Python (Boto3) を使用した条件付きオペレーションの使用例。

```
def example_usage():
    """Example of how to use the conditional operations functions."""
    # Example parameters
    table_name = "Products"
    key = {"ProductId": "prod123"}

    print("Example 1: Conditional Update")
    try:
        # Update the price only if the current stock is greater than 10
        success, response = conditional_update(
            table_name=table_name,
            key=key,
            condition_attribute="Stock",
            condition_value=10,
            update_attribute="Price",
            update_value=99.99,
        )

        if success:
            # Fix for mypy: Handle the case where response might be None
            attributes = {} if response is None else response.get("Attributes",
 {})
            print(f"Update succeeded! New values: {attributes}")
        else:
            print("Update failed because the condition was not met.")
    except Exception as e:
        print(f"Error during conditional update: {e}")

    print("\nExample 2: Conditional Delete")
    try:
        # Delete the product only if it's discontinued
        success = conditional_delete(
            table_name=table_name,
            key=key,
            condition_attribute="Status",
            condition_value="Discontinued",
        )

        if success:
```

```
            print("Delete succeeded! The item was deleted.")
        else:
            print("Delete failed because the condition was not met.")
    except Exception as e:
        print(f"Error during conditional delete: {e}")

    print("\nExample 3: Optimistic Locking")
    try:
        # Update with optimistic locking using a version attribute
        success, response = optimistic_locking_update(
            table_name=table_name,
            key=key,
            version_attribute="Version",
            update_attribute="Description",
            update_value="Updated product description",
        )

        if success:
            # Fix for mypy: Handle the case where response might be None
            attributes = {} if response is None else response.get("Attributes",
{})
            print(f"Optimistic locking update succeeded! New values:
{attributes}")
        else:
            print("Optimistic locking update failed because the version has
changed.")
    except Exception as e:
        print(f"Error during optimistic locking update: {e}")

    print("\nExample 4: Conditional Check and Update")
    try:
        # Update the featured status if the product is in stock
        response = conditional_check_and_update(
            table_name=table_name,
            key=key,
            check_attribute="InStock",
            check_value=True,
            update_attribute="Featured",
            update_value=True,
            create_if_not_exists=True,
        )

        if response:
            print(
```

```
                f"Conditional check and update succeeded! New values:
{response.get('Attributes', {})}"
            )
        else:
            print("Conditional check and update failed because the condition was
not met.")
    except Exception as e:
        print(f"Error during conditional check and update: {e}")

    print("\nUnderstanding Conditional Operations in DynamoDB:")
    print("1. Conditional operations help maintain data integrity")
    print("2. They prevent race conditions in concurrent environments")
    print("3. Failed conditions result in ConditionalCheckFailedException")
    print("4. No DynamoDB capacity is consumed when conditions fail")
    print("5. Optimistic locking is a common pattern using version attributes")
    print("6. Conditions can be combined with logical operators (AND, OR, NOT)")
    print("7. Conditions can use comparison operators (=, <>, <, <=, >, >=)")
    print(
        "8. attribute_exists() and attribute_not_exists() are useful for checking
attribute presence"
    )
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の以下のトピックを参照してください。
  - DeleteItem
  - PutItem
  - UpdateItem

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して DynamoDB で式の属性名を使用する

次のコード例は、DynamoDB で式の属性名を使用する方法を示しています。

- DynamoDB 式で予約語を操作します。

- 式の属性名のプレースホルダーを使用します。
- 属性名の特殊文字を処理します。

Java

SDK for Java 2.x

AWS SDK for Java 2.x を使用して式の属性名をデモンストレーションします。

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.QueryRequest;
import software.amazon.awssdk.services.dynamodb.model.QueryResponse;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanResponse;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemRequest;
import software.amazon.awssdk.services.dynamodb.model.UpdateItemResponse;

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

    /**
     * Updates an attribute that is a reserved word in DynamoDB.
     *
     * <p>This method demonstrates how to use expression attribute names to
 update
     * attributes that are reserved words in DynamoDB.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param reservedWordAttribute The reserved word attribute to update
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
```

```
        Map<String, AttributeValue> key,
        String reservedWordAttribute,
        AttributeValue value) {

        // Define the update parameters using expression attribute names
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET #attr = :value")
            .expressionAttributeNames(Map.of("#attr", reservedWordAttribute))
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Updates an attribute that contains special characters.
     *
     * <p>This method demonstrates how to use expression attribute names to
update
     * attributes that contain special characters.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param specialCharAttribute The attribute with special characters to
update
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateSpecialCharacterAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        String specialCharAttribute,
        AttributeValue value) {

        // Define the update parameters using expression attribute names
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
```

```
            .key(key)
            .updateExpression("SET #attr = :value")
            .expressionAttributeNames(Map.of("#attr", specialCharAttribute))
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Queries items using an attribute that is a reserved word.
     *
     * <p>This method demonstrates how to use expression attribute names in a
query
     * when the attribute is a reserved word.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param partitionKeyName The name of the partition key attribute
     * @param partitionKeyValue The value of the partition key
     * @param reservedWordAttribute The reserved word attribute to filter on
     * @param value The value to compare against
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static QueryResponse queryWithReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        String partitionKeyName,
        AttributeValue partitionKeyValue,
        String reservedWordAttribute,
        AttributeValue value) {

        // Define the query parameters using expression attribute names
        Map<String, String> expressionAttributeNames = new HashMap<>();
        expressionAttributeNames.put("#pkName", partitionKeyName);
        expressionAttributeNames.put("#attr", reservedWordAttribute);

        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        expressionAttributeValues.put(":pkValue", partitionKeyValue);
        expressionAttributeValues.put(":value", value);
```

```
        QueryRequest request = QueryRequest.builder()
            .tableName(tableName)
            .keyConditionExpression("#pkName = :pkValue")
            .filterExpression("#attr = :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the query operation
        return dynamoDbClient.query(request);
    }

    /**
     * Updates a nested attribute with a path that contains reserved words.
     *
     * <p>This method demonstrates how to use expression attribute names to
update
     * nested attributes where the path contains reserved words.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param key The key of the item to update
     * @param attributePath The path to the nested attribute as an array
     * @param value The value to set
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static UpdateItemResponse updateNestedReservedWordAttribute(
        DynamoDbClient dynamoDbClient,
        String tableName,
        Map<String, AttributeValue> key,
        List<String> attributePath,
        AttributeValue value) {

        // Create expression attribute names for each part of the path
        Map<String, String> expressionAttributeNames = new HashMap<>();
        for (int i = 0; i < attributePath.size(); i++) {
            expressionAttributeNames.put("#attr" + i, attributePath.get(i));
        }

        // Build the attribute path using the expression attribute names
        StringBuilder attributePathExpression = new StringBuilder();
        for (int i = 0; i < attributePath.size(); i++) {
            if (i > 0) {
```

```
                attributePathExpression.append(".");
            }
            attributePathExpression.append("#attr").append(i);
        }

        // Define the update parameters
        UpdateItemRequest request = UpdateItemRequest.builder()
            .tableName(tableName)
            .key(key)
            .updateExpression("SET " + attributePathExpression.toString() + "
= :value")
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(Map.of(":value", value))
            .returnValues("UPDATED_NEW")
            .build();

        // Perform the update operation
        return dynamoDbClient.updateItem(request);
    }

    /**
     * Scans a table with multiple attribute name placeholders.
     *
     * <p>This method demonstrates how to use multiple expression attribute names
     * in a complex filter expression.
     *
     * @param dynamoDbClient The DynamoDB client
     * @param tableName The name of the DynamoDB table
     * @param filters Object mapping attribute names to filter values
     * @return The response from DynamoDB
     * @throws DynamoDbException if an error occurs during the operation
     */
    public static ScanResponse scanWithMultipleAttributeNames(
        DynamoDbClient dynamoDbClient, String tableName, Map<String,
AttributeValue> filters) {

        // Create expression attribute names and values
        Map<String, String> expressionAttributeNames = new HashMap<>();
        Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
        StringBuilder filterExpression = new StringBuilder();

        // Build the filter expression
        int index = 0;
        for (Map.Entry<String, AttributeValue> entry : filters.entrySet()) {
```

```
            String attrName = entry.getKey();
            AttributeValue attrValue = entry.getValue();

            String nameKey = "#attr" + index;
            String valueKey = ":val" + index;

            expressionAttributeNames.put(nameKey, attrName);
            expressionAttributeValues.put(valueKey, attrValue);

            // Add AND between conditions (except for the first one)
            if (index > 0) {
                filterExpression.append(" AND ");
            }

            filterExpression.append(nameKey).append(" = ").append(valueKey);
            index++;
        }

        // Define the scan parameters
        ScanRequest request = ScanRequest.builder()
            .tableName(tableName)
            .filterExpression(filterExpression.toString())
            .expressionAttributeNames(expressionAttributeNames)
            .expressionAttributeValues(expressionAttributeValues)
            .build();

        // Perform the scan operation
        return dynamoDbClient.scan(request);
    }
```

AWS SDK for Java 2.x を使用した式の属性名の使用例。

```
    public static void exampleUsage(DynamoDbClient dynamoDbClient, String
tableName) {
        // Example key
        Map<String, AttributeValue> key = new HashMap<>();
        key.put("ProductId", AttributeValue.builder().s("P12345").build());

        System.out.println("Demonstrating expression attribute names in
DynamoDB");

        try {
```

```
            // Example 1: Update an attribute that is a reserved word
            System.out.println("\nExample 1: Updating an attribute that is a
reserved word");
            UpdateItemResponse response1 = updateReservedWordAttribute(
                dynamoDbClient,
                tableName,
                key,
                "Size", // "SIZE" is a reserved word in DynamoDB
                AttributeValue.builder().s("Large").build());

            System.out.println("Updated attribute: " + response1.attributes());

            // Example 2: Update an attribute with special characters
            System.out.println("\nExample 2: Updating an attribute with special
characters");
            UpdateItemResponse response2 = updateSpecialCharacterAttribute(
                dynamoDbClient,
                tableName,
                key,
                "Product-Type", // Contains a hyphen, which is a special
character
                AttributeValue.builder().s("Electronics").build());

            System.out.println("Updated attribute: " + response2.attributes());

            // Example 3: Query with a reserved word attribute
            System.out.println("\nExample 3: Querying with a reserved word
attribute");
            QueryResponse response3 = queryWithReservedWordAttribute(
                dynamoDbClient,
                tableName,
                "Category",
                AttributeValue.builder().s("Electronics").build(),
                "Count", // "COUNT" is a reserved word in DynamoDB
                AttributeValue.builder().n("10").build());

            System.out.println("Found " + response3.count() + " items");

            // Example 4: Update a nested attribute with reserved words in the
path
            System.out.println("\nExample 4: Updating a nested attribute with
reserved words in the path");
            UpdateItemResponse response4 = updateNestedReservedWordAttribute(
                dynamoDbClient,
```

```
                tableName,
                key,
                Arrays.asList("Dimensions", "Size", "Height"), // "SIZE" is a
 reserved word
                AttributeValue.builder().n("30").build());

            System.out.println("Updated nested attribute: " +
 response4.attributes());

            // Example 5: Scan with multiple attribute name placeholders
            System.out.println("\nExample 5: Scanning with multiple attribute
 name placeholders");
            Map<String, AttributeValue> filters = new HashMap<>();
            filters.put("Size", AttributeValue.builder().s("Large").build());
            filters.put("Count", AttributeValue.builder().n("10").build());
            filters.put(
                "Product-Type",
 AttributeValue.builder().s("Electronics").build());

            ScanResponse response5 =
 scanWithMultipleAttributeNames(dynamoDbClient, tableName, filters);

            System.out.println("Found " + response5.count() + " items");

            // Show some common reserved words
            System.out.println("\nSome common DynamoDB reserved words:");
            List<String> commonReservedWords = getDynamoDBReservedWords();
            System.out.println(String.join(", ", commonReservedWords));

            // Explain expression attribute names
            System.out.println("\nKey points about expression attribute names:");
            System.out.println("1. Use expression attribute names (#name) for
 reserved words");
            System.out.println("2. Use expression attribute names for attributes
 with special characters");
            System.out.println(
                "3. Special characters include: spaces, hyphens, dots, and other
 non-alphanumeric characters");
            System.out.println("4. Expression attribute names are required for
 nested attributes with reserved words");
            System.out.println("5. You can use multiple expression attribute
 names in a single expression");
            System.out.println("6. Expression attribute names are case-
sensitive");
```

```
            System.out.println("7. Expression attribute names are only used in
 expressions, not in the actual data");

        } catch (DynamoDbException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - クエリ:
  - UpdateItem

JavaScript

SDK for JavaScript (v3)

AWS SDK for JavaScript を使用して式の属性名をデモンストレーションします。

```
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const {
  DynamoDBDocumentClient,
  UpdateCommand,
  GetCommand,
  QueryCommand,
  ScanCommand
} = require("@aws-sdk/lib-dynamodb");

/**
 * Update an attribute that is a reserved word in DynamoDB.
 *
 * This function demonstrates how to use expression attribute names to update
 * attributes that are reserved words in DynamoDB.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} reservedWordAttribute - The reserved word attribute to update
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
```

```
 */
async function updateReservedWordAttribute(
  config,
  tableName,
  key,
  reservedWordAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using expression attribute names
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: "SET #attr = :value",
    ExpressionAttributeNames: {
      "#attr": reservedWordAttribute
    },
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Update an attribute that contains special characters.
 *
 * This function demonstrates how to use expression attribute names to update
 * attributes that contain special characters.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string} specialCharAttribute - The attribute with special characters
 to update
 * @param {any} value - The value to set
```

```
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateSpecialCharacterAttribute(
  config,
  tableName,
  key,
  specialCharAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the update parameters using expression attribute names
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: "SET #attr = :value",
    ExpressionAttributeNames: {
      "#attr": specialCharAttribute
    },
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));

  return response;
}

/**
 * Query items using an attribute that is a reserved word.
 *
 * This function demonstrates how to use expression attribute names in a query
 * when the attribute is a reserved word.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {string} partitionKeyName - The name of the partition key attribute
 * @param {any} partitionKeyValue - The value of the partition key
```

```
 * @param {string} reservedWordAttribute - The reserved word attribute to filter
 on
 * @param {any} value - The value to compare against
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function queryWithReservedWordAttribute(
  config,
  tableName,
  partitionKeyName,
  partitionKeyValue,
  reservedWordAttribute,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the query parameters using expression attribute names
  const params = {
    TableName: tableName,
    KeyConditionExpression: "#pkName = :pkValue",
    FilterExpression: "#attr = :value",
    ExpressionAttributeNames: {
      "#pkName": partitionKeyName,
      "#attr": reservedWordAttribute
    },
    ExpressionAttributeValues: {
      ":pkValue": partitionKeyValue,
      ":value": value
    }
  };

  // Perform the query operation
  const response = await docClient.send(new QueryCommand(params));

  return response;
}

/**
 * Update a nested attribute with a path that contains reserved words.
 *
 * This function demonstrates how to use expression attribute names to update
 * nested attributes where the path contains reserved words.
 *
```

```
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to update
 * @param {string[]} attributePath - The path to the nested attribute as an array
 * @param {any} value - The value to set
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function updateNestedReservedWordAttribute(
  config,
  tableName,
  key,
  attributePath,
  value
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create expression attribute names for each part of the path
  const expressionAttributeNames = {};
  for (let i = 0; i < attributePath.length; i++) {
    expressionAttributeNames[`#attr${i}`] = attributePath[i];
  }

  // Build the attribute path using the expression attribute names
  const attributePathExpression = attributePath
    .map((_, i) => `#attr${i}`)
    .join(".");

  // Define the update parameters
  const params = {
    TableName: tableName,
    Key: key,
    UpdateExpression: `SET ${attributePathExpression} = :value`,
    ExpressionAttributeNames: expressionAttributeNames,
    ExpressionAttributeValues: {
      ":value": value
    },
    ReturnValues: "UPDATED_NEW"
  };

  // Perform the update operation
  const response = await docClient.send(new UpdateCommand(params));
```

```
  return response;
}

/**
 * Scan a table with multiple attribute name placeholders.
 *
 * This function demonstrates how to use multiple expression attribute names
 * in a complex filter expression.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} filters - Object mapping attribute names to filter values
 * @returns {Promise<Object>} - The response from DynamoDB
 */
async function scanWithMultipleAttributeNames(
  config,
  tableName,
  filters
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Create expression attribute names and values
  const expressionAttributeNames = {};
  const expressionAttributeValues = {};
  const filterConditions = [];

  // Build the filter expression
  Object.entries(filters).forEach(([attrName, value], index) => {
    const nameKey = `#attr${index}`;
    const valueKey = `:val${index}`;

    expressionAttributeNames[nameKey] = attrName;
    expressionAttributeValues[valueKey] = value;
    filterConditions.push(`${nameKey} = ${valueKey}`);
  });

  // Join the filter conditions with AND
  const filterExpression = filterConditions.join(" AND ");

  // Define the scan parameters
  const params = {
    TableName: tableName,
```

```
    FilterExpression: filterExpression,
    ExpressionAttributeNames: expressionAttributeNames,
    ExpressionAttributeValues: expressionAttributeValues
  };

  // Perform the scan operation
  const response = await docClient.send(new ScanCommand(params));

  return response;
}

/**
 * Get the current value of an item.
 *
 * Helper function to retrieve the current value of an item.
 *
 * @param {Object} config - AWS configuration object
 * @param {string} tableName - The name of the DynamoDB table
 * @param {Object} key - The key of the item to get
 * @returns {Promise<Object|null>} - The item or null if not found
 */
async function getItem(
  config,
  tableName,
  key
) {
  // Initialize the DynamoDB client
  const client = new DynamoDBClient(config);
  const docClient = DynamoDBDocumentClient.from(client);

  // Define the get parameters
  const params = {
    TableName: tableName,
    Key: key
  };

  // Perform the get operation
  const response = await docClient.send(new GetCommand(params));

  // Return the item if it exists, otherwise null
  return response.Item || null;
}
```

AWS SDK for JavaScript を使用した式の属性名の使用例。

```
/**
 * Example of how to use expression attribute names.
 */
async function exampleUsage() {
  // Example parameters
  const config = { region: "us-west-2" };
  const tableName = "Products";
  const key = { ProductId: "P12345" };

  console.log("Demonstrating expression attribute names in DynamoDB");

  try {
    // Example 1: Update an attribute that is a reserved word
    console.log("\nExample 1: Updating an attribute that is a reserved word");
    const response1 = await updateReservedWordAttribute(
      config,
      tableName,
      key,
      "Size", // "SIZE" is a reserved word in DynamoDB
      "Large"
    );

    console.log("Updated attribute:", response1.Attributes);

    // Example 2: Update an attribute with special characters
    console.log("\nExample 2: Updating an attribute with special characters");
    const response2 = await updateSpecialCharacterAttribute(
      config,
      tableName,
      key,
      "Product-Type", // Contains a hyphen, which is a special character
      "Electronics"
    );

    console.log("Updated attribute:", response2.Attributes);

    // Example 3: Query with a reserved word attribute
    console.log("\nExample 3: Querying with a reserved word attribute");
    const response3 = await queryWithReservedWordAttribute(
      config,
      tableName,
      "Category",
```

```
      "Electronics",
      "Count", // "COUNT" is a reserved word in DynamoDB
      10
    );

    console.log(`Found ${response3.Items.length} items`);

    // Example 4: Update a nested attribute with reserved words in the path
    console.log("\nExample 4: Updating a nested attribute with reserved words in
the path");
    const response4 = await updateNestedReservedWordAttribute(
      config,
      tableName,
      key,
      ["Dimensions", "Size", "Height"], // "SIZE" is a reserved word
      30
    );

    console.log("Updated nested attribute:", response4.Attributes);

    // Example 5: Scan with multiple attribute name placeholders
    console.log("\nExample 5: Scanning with multiple attribute name
placeholders");
    const response5 = await scanWithMultipleAttributeNames(
      config,
      tableName,
      {
        "Size": "Large",
        "Count": 10,
        "Product-Type": "Electronics"
      }
    );

    console.log(`Found ${response5.Items.length} items`);

    // Get the final state of the item
    console.log("\nFinal state of the item:");
    const item = await getItem(config, tableName, key);
    console.log(JSON.stringify(item, null, 2));

    // Show some common reserved words
    console.log("\nSome common DynamoDB reserved words:");
    const commonReservedWords = [
      "ABORT", "ABSOLUTE", "ACTION", "ADD", "ALL", "ALTER", "AND", "ANY", "AS",
```

```
      "ASC", "BETWEEN", "BY", "CASE", "CAST", "COLUMN", "CONNECT", "COUNT",
      "CREATE", "CURRENT", "DATE", "DELETE", "DESC", "DROP", "ELSE", "EXISTS",
      "FOR", "FROM", "GRANT", "GROUP", "HAVING", "IN", "INDEX", "INSERT", "INTO",
      "IS", "JOIN", "KEY", "LEVEL", "LIKE", "LIMIT", "LOCAL", "MAX", "MIN",
 "NAME",
      "NOT", "NULL", "OF", "ON", "OR", "ORDER", "OUTER", "REPLACE", "RETURN",
      "SELECT", "SET", "SIZE", "TABLE", "THEN", "TO", "UPDATE", "USER", "VALUES",
      "VIEW", "WHERE"
    ];
    console.log(commonReservedWords.join(", "));

    // Explain expression attribute names
    console.log("\nKey points about expression attribute names:");
    console.log("1. Use expression attribute names (#name) for reserved words");
    console.log("2. Use expression attribute names for attributes with special
 characters");
    console.log("3. Special characters include: spaces, hyphens, dots, and other
 non-alphanumeric characters");
    console.log("4. Expression attribute names are required for nested attributes
 with reserved words");
    console.log("5. You can use multiple expression attribute names in a single
 expression");
    console.log("6. Expression attribute names are case-sensitive");
    console.log("7. Expression attribute names are only used in expressions, not
 in the actual data");

  } catch (error) {
    console.error("Error:", error);
  }
}
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - クエリ:
  - UpdateItem

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を使用して式の属性名をデモンストレーションします。

```
import boto3
from botocore.exceptions import ClientError
from typing import Any, Dict, List


def use_reserved_word_attribute(
    table_name: str, key: Dict[str, Any], reserved_word: str, value: Any
) -> Dict[str, Any]:
    """
    Update an attribute whose name is a DynamoDB reserved word.

    This function demonstrates how to use expression attribute names to work with
    attributes that have names that are DynamoDB reserved words.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        reserved_word (str): The reserved word to use as an attribute name.
        value (Any): The value to set for the attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names to handle the reserved word
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #reserved_attr = :value",
        ExpressionAttributeNames={"#reserved_attr": reserved_word},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def use_special_character_attribute(
    table_name: str, key: Dict[str, Any], attribute_with_special_chars: str,
 value: Any
```

```
) -> Dict[str, Any]:
    """
    Update an attribute whose name contains special characters.

    This function demonstrates how to use expression attribute names to work with
    attributes that have names containing special characters like spaces, dots,
 or hyphens.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        attribute_with_special_chars (str): The attribute name with special
 characters.
        value (Any): The value to set for the attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
 attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names to handle special characters
    response = table.update_item(
        Key=key,
        UpdateExpression="SET #special_attr = :value",
        ExpressionAttributeNames={"#special_attr": attribute_with_special_chars},
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )

    return response


def query_with_attribute_names(
    table_name: str,
    partition_key_name: str,
    partition_key_value: str,
    filter_attribute_name: str,
    filter_value: Any,
) -> Dict[str, Any]:
    """
```

```
    Query a table using expression attribute names for both key and filter
 attributes.

    This function demonstrates how to use expression attribute names in a query
 operation
    for both the key condition expression and filter expression.

    Args:
        table_name (str): The name of the DynamoDB table.
        partition_key_name (str): The name of the partition key attribute.
        partition_key_value (str): The value of the partition key to query.
        filter_attribute_name (str): The name of the attribute to filter on.
        filter_value (Any): The value to compare against in the filter.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the query results.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Use expression attribute names for both key condition and filter
    response = table.query(
        KeyConditionExpression="#pk = :pk_val",
        FilterExpression="#filter_attr = :filter_val",
        ExpressionAttributeNames={"#pk": partition_key_name, "#filter_attr":
 filter_attribute_name},
        ExpressionAttributeValues={":pk_val": partition_key_value, ":filter_val":
 filter_value},
    )

    return response


def update_nested_attribute_with_dots(
    table_name: str, key: Dict[str, Any], path_with_dots: str, value: Any
) -> Dict[str, Any]:
    """
    Update a nested attribute using a path with dot notation.

    This function demonstrates how to use expression attribute names to work with
    nested attributes specified using dot notation.

    Args:
```

```
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.
        path_with_dots (str): The path to the nested attribute using dot notation
(e.g., "a.b.c").
        value (Any): The value to set for the nested attribute.

    Returns:
        Dict[str, Any]: The response from DynamoDB containing the updated
attribute values.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Split the path into components
    path_parts = path_with_dots.split(".")

    # Build the update expression and attribute names
    update_expression = "SET "
    expression_attribute_names = {}

    # Build the path expression
    path_expression = ""
    for i, part in enumerate(path_parts):
        name_placeholder = f"#attr{i}"
        expression_attribute_names[name_placeholder] = part

        if i == 0:
            path_expression = name_placeholder
        else:
            path_expression += f".{name_placeholder}"

    # Complete the update expression
    update_expression += f"{path_expression} = :value"

    # Execute the update
    response = table.update_item(
        Key=key,
        UpdateExpression=update_expression,
        ExpressionAttributeNames=expression_attribute_names,
        ExpressionAttributeValues={":value": value},
        ReturnValues="UPDATED_NEW",
    )
```

```
    return response


def demonstrate_attribute_name_requirements(table_name: str, key: Dict[str, Any])
 -> Dict[str, Any]:
    """
    Demonstrate the requirements and allowed characters for attribute names.

    This function shows examples of valid and invalid attribute names and how to
    handle them using expression attribute names.

    Args:
        table_name (str): The name of the DynamoDB table.
        key (Dict[str, Any]): The primary key of the item to update.

    Returns:
        Dict[str, Any]: A dictionary containing the results of the demonstration.
    """
    # Initialize the DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    table = dynamodb.Table(table_name)

    # Examples of attribute names with different characteristics
    examples = {
        "valid_standard": "NormalAttribute",  # Standard attribute name (no
 placeholder needed)
        "valid_with_underscore": "Normal_Attribute",  # Underscore is allowed
        "valid_with_number": "Attribute123",  # Numbers are allowed
        "reserved_word": "Timestamp",  # Reserved word (requires placeholder)
        "starts_with_number": "123Attribute",  # Starts with number (valid but
 may need placeholder in some contexts)
        "with_space": "Attribute Name",  # Contains space (requires placeholder)
        "with_dot": "Attribute.Name",  # Contains dot (requires placeholder)
        "with_hyphen": "Attribute-Name",  # Contains hyphen (requires
 placeholder)
        "with_special_chars": "Attribute#$%",  # Contains special characters
 (requires placeholder)
    }

    results = {}

    # Try to update each attribute type
    for example_type, attr_name in examples.items():
        try:
```

```
            # For attributes that don't need placeholders, try direct reference
            if example_type in ["valid_standard", "valid_with_underscore",
"valid_with_number"]:
                try:
                    # Try without expression attribute names first
                    response = table.update_item(
                        Key=key,
                        UpdateExpression=f"SET {attr_name} = :value",
                        ExpressionAttributeValues={":value": f"Value for
{attr_name}"},
                        ReturnValues="UPDATED_NEW",
                    )
                    results[example_type] = {
                        "attribute_name": attr_name,
                        "success": True,
                        "needed_placeholder": False,
                        "response": response,
                    }
                except ClientError:
                    # If direct reference fails, try with placeholder
                    response = table.update_item(
                        Key=key,
                        UpdateExpression="SET #attr = :value",
                        ExpressionAttributeNames={"#attr": attr_name},
                        ExpressionAttributeValues={":value": f"Value for
{attr_name}"},
                        ReturnValues="UPDATED_NEW",
                    )
                    results[example_type] = {
                        "attribute_name": attr_name,
                        "success": True,
                        "needed_placeholder": True,
                        "response": response,
                    }
            else:
                # For attributes that definitely need placeholders
                response = table.update_item(
                    Key=key,
                    UpdateExpression="SET #attr = :value",
                    ExpressionAttributeNames={"#attr": attr_name},
                    ExpressionAttributeValues={":value": f"Value for
{attr_name}"},
                    ReturnValues="UPDATED_NEW",
                )
```

```
                results[example_type] = {
                    "attribute_name": attr_name,
                    "success": True,
                    "needed_placeholder": True,
                    "response": response,
                }
        except ClientError as e:
            results[example_type] = {"attribute_name": attr_name, "success":
 False, "error": str(e)}

    return results
```

AWS SDK for Python (Boto3) を使用した式の属性名の使用例。

```
def example_usage():
    """Example of how to use expression attribute names in DynamoDB."""
    # Example parameters
    table_name = "Products"
    key = {"ProductId": "prod123"}

    print("Example 1: Using a reserved word as an attribute name")
    try:
        response = use_reserved_word_attribute(
            table_name=table_name, key=key, reserved_word="Timestamp",
 value="2025-05-14T12:00:00Z"
        )
        print(f"Reserved word attribute updated successfully:
 {response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating reserved word attribute: {e}")

    print("\nExample 2: Using an attribute name with special characters")
    try:
        response = use_special_character_attribute(
            table_name=table_name,
            key=key,
            attribute_with_special_chars="Product Info",
            value="Special product information",
        )
```

```
        print(f"Special character attribute updated successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating special character attribute: {e}")

    print("\nExample 3: Querying with expression attribute names")
    try:
        response = query_with_attribute_names(
            table_name=table_name,
            partition_key_name="Category",
            partition_key_value="Electronics",
            filter_attribute_name="Price",
            filter_value=500,
        )
        print(
            f"Query with expression attribute names returned
{len(response.get('Items', []))} items"
        )
    except Exception as e:
        print(f"Error querying with expression attribute names: {e}")

    print("\nExample 4: Updating a nested attribute with dot notation")
    try:
        response = update_nested_attribute_with_dots(
            table_name=table_name,
            key=key,
            path_with_dots="Product.Details.Specifications",
            value={"Weight": "2.5 kg", "Dimensions": "30x20x10 cm"},
        )
        print(f"Nested attribute updated successfully:
{response.get('Attributes', {})}")
    except Exception as e:
        print(f"Error updating nested attribute: {e}")

    print("\nExample 5: Demonstrating attribute name requirements")
    try:
        results = demonstrate_attribute_name_requirements(table_name=table_name,
key=key)

        print("Attribute Name Requirements Results:")
        for example_type, result in results.items():
            if result.get("success", False):
                needed_placeholder = result.get("needed_placeholder", True)
                print(
```

```
                    f"  - {example_type}: '{result['attribute_name']}' -
{'Requires' if needed_placeholder else 'Does not require'} placeholder"
                )
            else:
                print(
                    f"  - {example_type}: '{result['attribute_name']}' - Failed:
{result.get('error', 'Unknown error')}"
                )
    except Exception as e:
        print(f"Error demonstrating attribute name requirements: {e}")

    print("\nCommon DynamoDB Reserved Words (sample):")
    reserved_words = get_common_reserved_words()
    print(", ".join(reserved_words[:20]) + "... (and many more)")

    print("\nWhen to Use Expression Attribute Names:")
    print("1. When the attribute name is a DynamoDB reserved word")
    print("2. When the attribute name contains special characters (spaces, dots,
hyphens)")
    print("3. When the attribute name begins with a number")
    print("4. When working with nested attributes using dot notation")
    print("5. When you need to reference the same attribute multiple times in an
expression")

    print("\nExpression Attribute Name Requirements:")
    print("1. Must begin with a pound sign (#)")
    print("2. After the pound sign, must contain at least one character")
    print("3. Can contain alphanumeric characters and underscore (_)")
    print("4. Are case-sensitive")
    print("5. Must be unique within a single expression")

    print("\nAttribute Name Requirements in DynamoDB:")
    print("1. Can begin with a-z, A-Z, or 0-9")
    print("2. Can contain a-z, A-Z, 0-9, underscore (_), dash (-), and dot (.)")
    print("3. Are case-sensitive")
    print("4. No length restrictions, but practical limits apply")
    print("5. Cannot be a DynamoDB reserved word if used directly in
expressions")
```

- API の詳細については、『AWS SDK for Python (Boto3) API リファレンス』の以下のトピックを参照してください。
  - クエリ:
  - UpdateItem

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

スケジュールされたイベントを使用した Lambda 関数の呼び出し

次のコード例は、Amazon EventBridge スケジュールイベントによって呼び出される AWS Lambda 関数を作成する方法を示しています。

Java

SDK for Java 2.x

AWS Lambda 関数を呼び出す Amazon EventBridge スケジュールイベントを作成する方法を示します。cron 式を使用して Lambda 関数が呼び出されるタイミングをスケジュールするように EventBridge を設定します。この例では、Lambda Java ランタイム API を使用して Lambda 関数を作成します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、年間の記念日に従業員を祝福するモバイルテキストメッセージを従業員に送信するアプリを作成する方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

AWS Lambda 関数を呼び出す Amazon EventBridge スケジュールイベントを作成する方法を示します。cron 式を使用して Lambda 関数が呼び出されるタイミングをスケジュールするように EventBridge を設定します。この例では、Lambda JavaScript ランタイム API を使用して Lambda 関数を作成します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、年間の記念日に従業員を祝福するモバイルテキストメッセージを従業員に送信するアプリを作成する方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

以下の例では、スケジュールした Amazon EventBridge イベントのターゲットとして、AWS Lambda 関数を登録する方法を示しています。Lambda ハンドラーは、後で取得するために Amazon CloudWatch Logs にわかりやすいメッセージと完全なイベントデータを書き込みます。

- Lambda 関数をデプロイします。
- EventBridge スケジュールイベントを作成し、Lambda 関数をターゲットにします。
- EventBridge に Lambda 関数を呼び出す許可を付与します
- CloudWatch Logs から最新のデータを出力して、スケジュールされた呼び出しの結果を表示しています。
- デモ中に作成されたすべてのリソースをクリーンアップします。

この例は GitHub で最もよく確認できます。完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK で DynamoDB を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB ローカルセカンダリインデックスを操作する

次のコード例は、ローカルセカンダリインデックスを含むテーブルを作成およびクエリする方法を示しています。

- ローカルセカンダリインデックス (LSI) を含むテーブルを作成します。
- 射影タイプが異なる複数の LSI を含むテーブルを作成します。
- LSI を使用してデータをクエリします。

Bash

AWS CLIBash スクリプトを使用した

ローカルセカンダリインデックスを含むテーブルを作成します。

```
# Create a table with a Local Secondary Index
aws dynamodb create-table \
    --table-name CustomerOrders \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
        AttributeName=OrderID,AttributeType=S \
        AttributeName=OrderDate,AttributeType=S \
```

```
    --key-schema \
        AttributeName=CustomerID,KeyType=HASH \
        AttributeName=OrderID,KeyType=RANGE \
    --local-secondary-indexes \
        "IndexName=OrderDateIndex,\
        KeySchema=[{AttributeName=CustomerID,KeyType=HASH},
{AttributeName=OrderDate,KeyType=RANGE}],\
        Projection={ProjectionType=ALL}" \
    --billing-mode PAY_PER_REQUEST
```

複数の LSI を含むテーブルを作成します。

```
# Create a table with multiple Local Secondary Indexes
aws dynamodb create-table \
    --table-name CustomerDetails \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
        AttributeName=Name,AttributeType=S \
        AttributeName=Email,AttributeType=S \
        AttributeName=RegistrationDate,AttributeType=S \
    --key-schema \
        AttributeName=CustomerID,KeyType=HASH \
        AttributeName=Name,KeyType=RANGE \
    --local-secondary-indexes \
        "[
            {
                \"IndexName\": \"EmailIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"CustomerID\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"Email\",\"KeyType\":\"RANGE\"}
                ],
                \"Projection\": {\"ProjectionType\":\"INCLUDE\",
\"NonKeyAttributes\":[\"Address\",\"Phone\"]}
            },
            {
                \"IndexName\": \"RegistrationIndex\",
                \"KeySchema\": [
                    {\"AttributeName\":\"CustomerID\",\"KeyType\":\"HASH\"},
                    {\"AttributeName\":\"RegistrationDate\",\"KeyType\":\"RANGE
\"}
                ],
                \"Projection\": {\"ProjectionType\":\"KEYS_ONLY\"}
```

```
            }
        ]" \
    --billing-mode PAY_PER_REQUEST
```

LSI を使用してデータをクエリします。

```
# Query the OrderDateIndex LSI
aws dynamodb query \
    --table-name CustomerOrders \
    --index-name OrderDateIndex \
    --key-condition-expression "CustomerID = :custId AND OrderDate BETWEEN :date1
 AND :date2" \
    --expression-attribute-values '{
        ":custId": {"S": "C1"},
        ":date1": {"S": "2023-01-01"},
        ":date2": {"S": "2023-02-01"}
    }'

# Query with a filter expression
aws dynamodb query \
    --table-name CustomerOrders \
    --index-name OrderDateIndex \
    --key-condition-expression "CustomerID = :custId" \
    --filter-expression "Amount > :amount" \
    --expression-attribute-values '{
        ":custId": {"S": "C1"},
        ":amount": {"N": "150"}
    }'
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - クエリ:

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB Streams と Time-to-Live を操作する

次のコード例は、DynamoDB Streams 機能と Time-to-Live 機能を管理する方法を示しています。

- Streams を有効にしてテーブルを作成します。
- Streams について説明します。
- Streams を処理する Lambda 関数を作成します。
- テーブルで TTL を有効にします。
- TTL 属性を持つ項目を追加します。
- TTL 設定について説明します。

Bash

AWS CLIBash スクリプトを使用した

Streams を有効にしてテーブルを作成します。

```
# Create a table with DynamoDB Streams enabled
aws dynamodb create-table \
    --table-name StreamsDemo \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Streams について説明します。

```
# Get information about the stream
aws dynamodb describe-table \
    --table-name StreamsDemo \
    --query "Table.StreamSpecification"

# Get the stream ARN
STREAM_ARN=$(aws dynamodb describe-table \
    --table-name StreamsDemo \
```

```
    --query "Table.LatestStreamArn" \
    --output text)

echo "Stream ARN: $STREAM_ARN"

# Describe the stream
aws dynamodbstreams describe-stream \
    --stream-arn $STREAM_ARN
```

Streams の Lambda 関数を作成します。

```
# Step 1: Create an IAM role for the Lambda function
cat > trust-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF

aws iam create-role \
    --role-name DynamoDBStreamsLambdaRole \
    --assume-role-policy-document file://trust-policy.json

# Step 2: Attach permissions to the role
aws iam attach-role-policy \
    --role-name DynamoDBStreamsLambdaRole \
    --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaDynamoDBExecutionRole

# Step 3: Create a Lambda function (code would be in a separate file)
echo "Lambda function creation would be done separately with appropriate code"

# Step 4: Create an event source mapping
echo "Example command to create event source mapping:"
```

```
echo "aws lambda create-event-source-mapping \\"
echo "    --function-name ProcessDynamoDBRecords \\"
echo "    --event-source $STREAM_ARN \\"
echo "    --batch-size 100 \\"
echo "    --starting-position LATEST"
```

テーブルで TTL を有効にします。

```
# Create a table for TTL demonstration
aws dynamodb create-table \
    --table-name TTLDemo \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST

# Wait for table to become active
aws dynamodb wait table-exists --table-name TTLDemo

# Enable TTL on the table
aws dynamodb update-time-to-live \
    --table-name TTLDemo \
    --time-to-live-specification "Enabled=true, AttributeName=ExpirationTime"
```

TTL 属性を持つ項目を追加します。

```
# Calculate expiration time (current time + 1 day in seconds)
EXPIRATION_TIME=$(date -d "+1 day" +%s)

# Add an item with TTL attribute
aws dynamodb put-item \
    --table-name TTLDemo \
    --item '{
        "ID": {"S": "item1"},
        "Data": {"S": "This item will expire in 1 day"},
        "ExpirationTime": {"N": "'$EXPIRATION_TIME'"}
    }'

# Add an item that expires in 1 hour
EXPIRATION_TIME_HOUR=$(date -d "+1 hour" +%s)
```

```
aws dynamodb put-item \
    --table-name TTLDemo \
    --item '{
        "ID": {"S": "item2"},
        "Data": {"S": "This item will expire in 1 hour"},
        "ExpirationTime": {"N": "'$EXPIRATION_TIME_HOUR'"}
    }'
```

TTL 設定について説明します。

```
# Describe TTL settings for a table
aws dynamodb describe-time-to-live \
    --table-name TTLDemo
```

- API の詳細については、「AWS CLI コマンドリファレンス」の以下のトピックを参照してください。
  - AttachRolePolicy
  - CreateRole
  - CreateTable
  - DescribeTable
  - DescribeTimeToLive
  - PutItem
  - UpdateTimeToLive

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して、DynamoDB グローバルテーブルと最終的な一貫性を備えたマルチリージョンレプリケーション (MREC) を操作する

次のコード例は、最終的な一貫性を備えたマルチリージョンレプリケーション (MREC) を使用して DynamoDB グローバルテーブルを管理する方法を示しています。

- マルチリージョンレプリケーション (MREC)を備えたテーブルを作成します。
- レプリカテーブルの項目を配置および取得します。
- レプリカを一つずつ削除します。
- テーブルを削除してクリーンアップします。

Bash

AWS CLIBash スクリプトを使用した

マルチリージョンレプリケーションを備えたテーブルを作成します。

```
# Step 1: Create a new table (MusicTable) in US East (Ohio), with DynamoDB
 Streams enabled (NEW_AND_OLD_IMAGES)
aws dynamodb create-table \
    --table-name MusicTable \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2

# Step 2: Create an identical MusicTable table in US East (N. Virginia)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2

# Step 3: Create a table in Europe (Ireland)
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
```

```
  "ReplicaUpdates":
  [
    {
      "Create": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2
```

マルチリージョンテーブルについて説明します。

```
# Step 4: View the list of replicas created using describe-table
aws dynamodb describe-table \
    --table-name MusicTable \
    --region us-east-2 \
    --query 'Table.
{TableName:TableName,TableStatus:TableStatus,MultiRegionConsistency:MultiRegionConsistenc
{Region:RegionName,Status:ReplicaStatus}}'
```

レプリカテーブルに項目を配置します。

```
# Step 5: To verify that replication is working, add a new item to the Music
 table in US East (Ohio)
aws dynamodb put-item \
    --table-name MusicTable \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

レプリカテーブルから項目を取得します。

```
# Step 6: Wait for a few seconds, and then check to see whether the item has
 been
# successfully replicated to US East (N. Virginia) and Europe (Ireland)
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1
```

```
aws dynamodb get-item \
    --table-name MusicTable \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

レプリカを削除します。

```
# Step 7: Delete the replica table in Europe (Ireland) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "eu-west-1"
      }
    }
  ]
}' \
--region us-east-2

# Delete the replica table in US East (N. Virginia) Region
aws dynamodb update-table --table-name MusicTable --cli-input-json \
'{
  "ReplicaUpdates":
  [
    {
      "Delete": {
        "RegionName": "us-east-1"
      }
    }
  ]
}' \
--region us-east-2
```

テーブルを削除してクリーンアップします。

```
# Clean up: Delete the primary table
aws dynamodb delete-table --table-name MusicTable --region us-east-2
```

```
echo "Global table demonstration complete."
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - DeleteTable
  - DescribeTable
  - GetItem
  - PutItem
  - UpdateTable

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB リソースのタグ付けを操作する

次のコード例は、DynamoDB リソースのタグを管理する方法を示しています。

- タグを含むテーブルを作成します。
- リソースのタグを一覧表示します。
- リソースにタグを追加します。
- リソースからタグを削除します。
- テーブルをタグでフィルタリングします。

Bash

AWS CLIBash スクリプトを使用した

タグを含むテーブルを作成します。

```
# Create a table with tags
aws dynamodb create-table \
    --table-name TaggedTable \
```

```
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Environment,Value=Production \
        Key=Project,Value=Analytics \
        Key=Owner,Value=DataTeam
```

リソースのタグを一覧表示します。

```
# Get the table ARN
TABLE_ARN=$(aws dynamodb describe-table \
    --table-name TaggedTable \
    --query "Table.TableArn" \
    --output text)

# List tags for the table
aws dynamodb list-tags-of-resource \
    --resource-arn $TABLE_ARN
```

リソースにタグを追加します。

```
# Add tags to an existing table
aws dynamodb tag-resource \
    --resource-arn $TABLE_ARN \
    --tags \
        Key=CostCenter,Value=12345 \
        Key=BackupSchedule,Value=Daily
```

リソースからタグを削除します。

```
# Remove tags from a table
aws dynamodb untag-resource \
    --resource-arn $TABLE_ARN \
    --tag-keys Owner BackupSchedule
```

テーブルをタグでフィルタリングします。

```
# Create another table with different tags
aws dynamodb create-table \
    --table-name AnotherTaggedTable \
    --attribute-definitions \
        AttributeName=ID,AttributeType=S \
    --key-schema \
        AttributeName=ID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --tags \
        Key=Environment,Value=Development \
        Key=Project,Value=Testing

# Wait for table to become active
aws dynamodb wait table-exists --table-name AnotherTaggedTable

# List all tables
echo "All tables:"
aws dynamodb list-tables

# Get ARNs for all tables
echo -e "\nFiltering tables by Environment=Production tag:"
TABLE_ARNS=$(aws dynamodb list-tables --query "TableNames[*]" --output text |
 xargs -I {} aws dynamodb describe-table --table-name {} --query "Table.TableArn"
 --output text)

# Find tables with specific tag
for ARN in $TABLE_ARNS; do
    TABLE_NAME=$(echo $ARN | awk -F/ '{print $2}')
    TAGS=$(aws dynamodb list-tags-of-resource --resource-arn $ARN --query "Tags[?
Key=='Environment' && Value=='Production']" --output text)
    if [ ! -z "$TAGS" ]; then
        echo "Table with Production tag: $TABLE_NAME"
    fi
done
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateTable
  - ListTagsOfResource

- TagResource
- UntagResource

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS Command Line Interface v2 を使用して DynamoDB テーブルの暗号化を操作する

次のコード例は、DynamoDB テーブルの暗号化オプションを管理する方法を示しています。

- デフォルトの暗号化を使用してテーブルを作成します。
- カスタマーマネージド CMK を使用してテーブルを作成します。
- テーブルの暗号化設定を更新します。
- テーブルの暗号化について説明します。

Bash

AWS CLIBash スクリプトを使用した

デフォルトの暗号化を使用してテーブルを作成します。

```
# Create a table with default encryption (AWS owned key)
aws dynamodb create-table \
    --table-name CustomerData \
    --attribute-definitions \
        AttributeName=CustomerID,AttributeType=S \
    --key-schema \
        AttributeName=CustomerID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --sse-specification Enabled=true,SSEType=KMS
```

カスタマーマネージド CMK を使用してテーブルを作成します。

```
# Step 1: Create a customer managed key in KMS
aws kms create-key \
```

```
    --description "Key for DynamoDB table encryption" \
    --key-usage ENCRYPT_DECRYPT \
    --customer-master-key-spec SYMMETRIC_DEFAULT

# Store the key ID for later use
KEY_ID=$(aws kms list-keys --query "Keys[?contains(KeyArn, 'Key for
 DynamoDB')].KeyId" --output text)

# Step 2: Create a table with the customer managed key
aws dynamodb create-table \
    --table-name SensitiveData \
    --attribute-definitions \
        AttributeName=RecordID,AttributeType=S \
    --key-schema \
        AttributeName=RecordID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=$KEY_ID
```

テーブルの暗号化を更新します。

```
# Update a table to use a different KMS key
aws dynamodb update-table \
    --table-name CustomerData \
    --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=$KEY_ID
```

テーブルの暗号化について説明します。

```
# Describe the table to see encryption settings
aws dynamodb describe-table \
    --table-name CustomerData \
    --query "Table.SSEDescription"
```

- API の詳細については、AWS CLI コマンドリファレンスの以下のトピックを参照してください。
  - CreateKey
  - CreateTable
  - DescribeTable
  - UpdateTable

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB 用のサーバーレスサンプル

以下は、AWS SDK で DynamoDB を使用する方法を説明するコード例です。

例

- DynamoDB トリガーから Lambda 関数を呼び出す
- DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする

DynamoDB トリガーから Lambda 関数を呼び出す

次のコードの例では、DynamoDB ストリームからレコードを受信することによってトリガーされるイベントを受け取る Lambda 関数の実装方法が示されています。関数は DynamoDB ペイロードを取得し、レコードの内容をログ記録します。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Go を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Java を使用して Lambda で DynamoDB イベントの消費。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

JavaScript を使用した Lambda での DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

TypeScript を使用した Lambda での DynamoDB イベントの消費。

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB イベントの消費。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data
```

```
            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Python を使用して Lambda で DynamoDB イベントの消費。

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB イベントの消費。

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB イベントの消費。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())
```

```
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする

次のコードの例では、DynamoDB ストリームからイベントを受け取る Lambda 関数の部分的なバッチレスポンスの実装方法が示されています。この関数は、レスポンスとしてバッチアイテムの失敗を報告し、対象のメッセージを後で再試行するよう Lambda に伝えます。

.NET

SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
```

```
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Go を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""
```

```
 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Java を使用した Lambda での DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;
```

```
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

JavaScript を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

TypeScript を使用して Lambda で DynamoDB のバッチアイテム失敗をレポート。

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
```

```
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB バッチ項目失敗のレポート。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
```

```
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
```

```
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Python を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を見つけて、設定と実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
```

```
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWSDynamoDB の コミュニティへの貢献

AWS コミュニティへの貢献は、AWS 間の複数のチームによって作成および維持されている例です。フィードバックを提供するには、リンクされたリポジトリで提供されているメカニズムを使用します。

例

- サーバーレスアプリケーションの構築とテスト

サーバーレスアプリケーションの構築とテスト

次のコード例では、Lambda および DynamoDB を備えた API Gateway を使用してサーバーレスアプリケーションを構築およびテストする方法について示されます

.NET

SDK for .NET

.NET SDK を使用し、Lambda および DynamoDB を備えた API Gateway で構成されるサーバーレスアプリケーションを構築およびテストする方法が示されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

Go

SDK for Go V2

Go SDK を使用し、Lambda および DynamoDB を備えた API Gateway で構成されるサーバーレスアプリケーションを構築およびテストする方法が示されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

Java

SDK for Java 2.x

Java SDK を使用し、Lambda および DynamoDB を備えた API Gateway で構成されるサーバーレスアプリケーションを構築およびテストする方法が示されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

Rust

SDK for Rust

Rust SDK を使用し、Lambda および DynamoDB を備えた API Gateway で構成されるサーバーレスアプリケーションを構築およびテストする方法が示されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API ゲートウェイ
- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、AWS SDK で DynamoDB を使用する を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon DynamoDB のセキュリティとコンプライアンス

AWS ではクラウドセキュリティが最優先事項です。セキュリティを最も重視する組織の要件を満たすために構築された 「AWS」 のデータセンターとネットワークアーキテクチャは、お客様に大きく貢献します。

セキュリティは、「AWS」 と顧客の間の責任共有です。責任共有モデルでは、この責任がクラウドのセキュリティおよびクラウド内のセキュリティとして説明されています。

- クラウドのセキュリティ - AWS は、AWS クラウドで AWS サービスを実行するインフラストラクチャを保護する責任を負います。また、AWS は、使用するサービスを安全に提供します。セキュリティの有効性については、AWS コンプライアンスプログラムの一環として、サードパーティー監査人によるテストと検証が定期的に行われています。DynamoDB に適用するコンプライアンスプログラムの詳細については、コンプライアンスプログラムによる対象範囲内の AWS サービスを参照してください。
- クラウド内のセキュリティ - お客様の責任は、使用する AWS のサービスに応じて異なります。また、お客様は、お客様のデータの機密性、組織の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントでは、DynamoDB を使用するときに、責任共有モデルを適用する方法を理解できます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するために DynamoDB を設定する方法を示します。また、DynamoDB リソースのモニタリングや保護に役立つ他の AWS サービスの用法についても説明します。

トピック

- AWSAmazon DynamoDB の マネージドポリシー
- DynamoDB 用のリソースベースのポリシーを使用する
- DynamoDB での属性ベースのアクセス制御の使用
- DynamoDB におけるデータ保護
- AWS Identity and Access Management (IAM) と DynamoDB
- DynamoDB の業界別のコンプライアンス検証
- Amazon DynamoDB での耐障害性と災害対策
- Amazon DynamoDB のインフラストラクチャセキュリティ
- AWS PrivateLink for DynamoDB

- Amazon DynamoDB での設定と脆弱性の分析
- Amazon DynamoDB のセキュリティベストプラクティス

AWSAmazon DynamoDB の マネージドポリシー

DynamoDB は、AWS マネージドポリシーを使用して、サービスが特定のアクションを実行するために必要な一連のアクセス許可を定義します。DynamoDB は、その AWS マネージドポリシーを維持および更新します。AWS マネージドポリシーのアクセス許可を変更することはできません。AWS マネージドポリシーの詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」を参照してください。

DynamoDB は、新しい機能をサポートするために、AWS マネージドポリシーに新しいアクセス許可を追加する場合があります。この種の更新は、ポリシーがアタッチされているすべてのアイデンティティ (ユーザー、グループ、ロール) に影響を与えます。AWS マネージドポリシーは、新しい機能がリリースされたときや新しいオペレーションが使用可能になったときに、更新される可能性が最も高くなります。DynamoDB は AWS マネージドポリシーからアクセス許可を削除しないため、ポリシーの更新によって既存のアクセス許可が破棄されることはありません。AWS マネージドポリシーの完全なリストについては、「AWS マネージドポリシー」を参照してください。

AWS マネージドポリシー: DynamoDBReplicationServiceRolePolicy

`DynamoDBReplicationServiceRolePolicy` ポリシーを IAM エンティティにアタッチすることはできません。このポリシーは、ユーザーに代わってアクションを実行することを DynamoDB に許可する、サービスにリンクされたロールにアタッチします。詳細については、「グローバルテーブルでの IAM の使用」を参照してください。

このポリシーは、サービスにリンクされたロールに対して、グローバルテーブルのレプリカ間でデータをレプリケートするアクセス許可を付与します。また、ユーザーに代わってグローバルテーブルのレプリカを管理する管理者アクセス許可も付与します。

アクセス許可の詳細

このポリシーは、以下を行うアクセス許可を付与します。

- `dynamodb` — データのレプリケーションを実行し、テーブルのレプリカを管理します。
- `application-autoscaling` — テーブルの AutoScaling 設定を取得および管理します。
- `account` — レプリカのアクセシビリティを評価するためのリージョンのステータスを取得します。

- iam — サービスリンクロールが現時点で存在しない場合、アプリケーションの AutoScaling 用のサービスリンクロールを作成します。

詳細については、「AWS マネージドポリシーリファレンス」の「DynamoDBReplicationServiceRolePolicy」を参照してください。

AWS マネージドポリシー: AmazonDynamoDBFullAccess_v2

スコープダウン AmazonDynamoDBFullAccess_v2 ポリシーは、ユーザーに特定のアクセス権限を付与します。AmazonDynamoDBFullAccess_v2 ポリシーを IAM IDにアタッチできます。このポリシーは、Amazon DynamoDB リソースへの管理アクセスを許可し、DynamoDB が統合されている AWS のサービスへのアクセスを IAM ID (ユーザー、グループ、ロールなど) に許可して、DynamoDB のすべての機能を使用できるようにします。このポリシーを使用すると、AWS マネジメントコンソールで利用可能な DynamoDB のすべての機能にアクセスできます。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- Amazon DynamoDB
- DynamoDB Accelerator
- AWS KMS
- AWS Resource Groups Tagging
- Lambda
- Application Auto Scaling
- CloudWatch
- Amazon Kinesis
- Amazon EC2
- IAM

JSON 形式のポリシーを確認するには、「AmazonDynamoDBFullAccess_v2」を参照してください。

AWS 管理ポリシー: AmazonDynamoDBReadOnlyAccess

AmazonDynamoDBReadOnlyAccess ポリシーを IAM IDにアタッチできます。

このポリシーは Amazon DynamoDB への読み取り専用アクセスを許可します。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- `Amazon DynamoDB` – Amazon DynamoDB への読み取り専用アクセスを提供します。
- `Amazon DynamoDB Accelerator (DAX)` – Amazon DynamoDB Accelerator (DAX) への読み取り専用アクセスを提供します。
- `Application Auto Scaling` — Application Auto Scaling の設定をプリンシパルが表示できるようにします。これは、テーブルにアタッチされているオートスケーリングポリシーをユーザーが表示できる場合に必須です。
- `CloudWatch` - CloudWatch で設定されたメトリクスデータとアラームをプリンシパルが表示できるようにします。これは、テーブルに対して設定された請求対象テーブルサイズと CloudWatch アラームをユーザーが表示できる場合に必須です。
- `AWS Data Pipeline` — プリンシパルが AWS Data Pipeline と関連オブジェクトを表示することを許可します。
- `Amazon EC2` – プリンシパルが Amazon EC2 VPC、サブネット、およびセキュリティグループを表示することを許可します。
- `IAM` — プリンシパルが IAM ロールを表示することを許可します。
- `AWS KMS` — で設定されたキーをプリンシパルが表示できるようにします。AWS KMSこれは、ユーザーが、各自のアカウントで作成して管理している AWS KMS keys を表示するために必要です。
- `Amazon SNS` – Amazon SNS のトピックとトピック別のサブスクリプションを一覧表示することをプリンシパルに許可します。
- `AWS Resource Groups` — プリンシパルがリソースグループとクエリを表示することを許可します。
- `AWS Resource Groups Tagging` — プリンシパルがリージョン内のタグ付けされたリソースまたは以前にタグ付けされたリソースのすべてを一覧表示することを許可します。
- `Kinesis` — プリンシパルが Kinesis データストリーム記述を表示することを許可します。
- `Amazon CloudWatch Contributor Insights` — プリンシパルが Contributor Insights ルールによって収集された時系列データを表示することを許可します。

ポリシーを `JSON` 形式で確認するには、「AmazonDynamoDBReadOnlyAccess」を参照してください。

DynamoDB での AWS マネージドポリシーの更新

この表は、DynamoDB の AWS アクセス管理ポリシーの更新を示しています。

変更	説明	変更日
AmazonDynamoDBFullAccess - 廃止済み	このポリシーは、AmazonDynamoDBFullAccess_v2 という名前のスコープダウンポリシーに置き換えられました。 AmazonDynamoDBFullAccess ポリシーは廃止され、推奨されなくなりました。	2025 年 4 月 28 日
AmazonDynamoDBReadOnlyAccess の既存のポリシーに対する更新	AmazonDynamoDBReadOnlyAccess に dynamodb:GetAbacStatus および dynamodb:UpdateAbacStatus のアクセス許可を追加しました。これらのアクセス許可により、現在のリージョンで AWS アカウントの ABAC ステータスを表示し、ABAC を有効にすることができます。	2024 年 11 月 18 日
AmazonDynamoDBReadOnlyAccess の既存のポリシーに対する更新	AmazonDynamoDBReadOnlyAccess にアクセス許可 を追加しました。dynamodb:GetResourcePolicy このアクセス許可は、DynamoDB リソースにアタッチされたリソースベースのポリシーの読み取りアクセスを提供します。	2024 年 3 月 20 日
DynamoDBReplicationServiceR	DynamoDBReplicationServiceRolePolicy にアクセス許可 を追加しました。dynamodb:GetResour	2023 年 12 月 15 日

変更	説明	変更日
olePolicy の既存のポリシーに対する更新	cePolicy このアクセス許可では、サービスリンクロールは、DynamoDB リソースにアタッチされたリソースベースのポリシーを読み取ることができます。	
DynamoDBReplicationServiceRolePolicy の既存のポリシーに対する更新	DynamoDBReplicationServiceRolePolicy にアクセス許可 account:ListRegions を追加しました。このアクセス許可は、サービスにリンクされたロールに対して、レプリカのアクセシビリティを評価することを許可します。	2023 年 5 月 10 日
DynamoDBReplicationServiceRolePolicy をマネージドポリシーのリストに追加	マネージドポリシー DynamoDBReplicationServiceRolePolicy に関する情報を追加しました。このポリシーは DynamoDB グローバルテーブルのサービスにリンクされたロールで使用されます。	2023 年 5 月 10 日
DynamoDB グローバルテーブルが変更の追跡を開始	DynamoDB グローバルテーブルが AWS マネージドポリシーの変更の追跡を開始しました。	2023 年 5 月 10 日

DynamoDB 用のリソースベースのポリシーを使用する

DynamoDB では、テーブル、インデックス、ストリームに対してリソースベースのポリシーを利用できます。リソースベースのポリシーでは、各リソースにアクセスできるユーザーと、各リソースに対して実行できるアクションを指定することで、アクセス許可を定義できます。

リソースベースのポリシーを DynamoDB リソース (テーブルやストリームなど) にアタッチできます。このポリシーでは、Identity and Access Management (IAM) プリンシパルに対して、該当する DynamoDB リソースで特定のアクションを実行するためのアクセス許可を指定します。例えば、テーブルにアタッチされたポリシーでは、そのテーブルとそのインデックスへのアクセス許可を指定します。そのため、リソースベースのポリシーを使用すれば、リソース単位でアクセス許可を定義することで、DynamoDB のテーブル、インデックス、ストリームへのアクセスを簡単に制御できます。DynamoDB リソースにアタッチできるポリシーの最大サイズは 20 KB です。

リソースベースのポリシーを使用する大きな利点は、さまざまな AWS アカウント で IAM プリンシパルにクロスアカウントアクセスを許可するための、クロスアカウントアクセス制御が簡単になることです。詳細については、「クロスアカウントアクセスのリソースベースのポリシー」を参照してください。

リソースベースのポリシーは、IAM Access Analyzer の外部アクセスアナライザーやブロックパブリックアクセス (BPA) 機能と統合することもできます。IAM Access Analyzer は、リソースベースのポリシーで指定された外部エンティティへのクロスアカウントアクセスについて報告します。また、情報を可視化してくれるので、アクセス許可を調整し、最小特権の原則を守るうえでも役立ちます。BPA では、DynamoDB のテーブル、インデックス、ストリームへのパブリックアクセスを阻止できます。BPA は、リソースベースのポリシーの作成と変更のワークフローで自動的に有効になります。

トピック

- リソースベースのポリシーを指定してテーブルを作成する
- DynamoDB の既存のテーブルにポリシーをアタッチする
- リソースベースのポリシーを DynamoDB ストリームにアタッチする
- リソースベースのポリシーを DynamoDB テーブルから削除する
- DynamoDB でリソースベースのポリシーを使用したクロスアカウントアクセス
- DynamoDB でリソースベースのポリシーでパブリックアクセスをブロックする
- リソースベースのポリシーでサポートされる DynamoDB API オペレーション
- IAM アイデンティティベースのポリシーと DynamoDB リソースベースのポリシーによる認可
- DynamoDB リソースベースのポリシーの例
- DynamoDB リソースベースのポリシーの考慮事項
- DynamoDB リソースベースのポリシーに関するベストプラクティス

リソースベースのポリシーを指定してテーブルを作成する

DynamoDB コンソール、CreateTable API、AWS CLI、AWS SDK、または CloudFormation テンプレートを使用して、テーブルの作成時にリソースベースのポリシーを追加できます。

AWS CLI

次の例では、`create-table` AWS CLI コマンドを使用して、*MusicCollection* という名前のテーブルを作成します。このコマンドでは、そのテーブルにリソースベースのポリシーを追加する `resource-policy` パラメータも指定されています。このポリシーは、ユーザー *John* に対して、該当のテーブルで RestoreTableToPointInTime、GetItem、PutItem の各 API アクションの実行を許可しています。

*######*のテキストを、リソース固有の情報に必ず置き換えてください。

```
aws dynamodb create-table \
    --table-name MusicCollection \
    --attribute-definitions AttributeName=Artist,AttributeType=S
 AttributeName=SongTitle,AttributeType=S \
    --key-schema AttributeName=Artist,KeyType=HASH
 AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --resource-policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::123456789012:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:RestoreTableToPointInTime\",
                        \"dynamodb:GetItem\",
                        \"dynamodb:DescribeTable\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS マネジメントコンソール

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ダッシュボードで、[テーブルの作成] を選択します。
3. [テーブルの詳細] に、テーブル名、パーティションキー、ソートキーの詳細を入力します。
4. [テーブル設定] セクションで [設定をカスタマイズ] を選択します。
5. (オプション) [テーブルクラス]、[キャパシティー計算ツール]、[読み込み/書き込みキャパシティーの設定]、[セカンダリインデックス]、[保管時の暗号化]、[削除保護] のオプションを指定します。
6. [リソースベースのポリシー] で、テーブルとそのインデックスへのアクセス許可を定義するポリシーを追加します。このポリシーでは、これらのリソースにアクセスできるユーザーと、各リソースに対して実行できるアクションを指定します。ポリシーを追加するには、以下のいずれかを実行します。

   - JSON ポリシードキュメントを入力するか貼り付けます。IAM ポリシー言語の詳細については、「IAM ユーザーガイド」の「JSON エディターを使用したポリシーの作成」を参照してください。

     > **Tip**
     >
     > 「Amazon DynamoDB デベロッパーガイド」のリソースベースのポリシーの例を確認するには、[ポリシーの例] を選択してください。

   - [新しいステートメントを追加] を選択して新しいステートメントを追加し、提示されたフィールドに情報を入力します。このステップを、追加するステートメントの数だけ繰り返します。

   > **Important**
   >
   > セキュリティ警告、エラー、提案がある場合は、ポリシーを保存する前に解決してください。

   次の IAM ポリシーの例では、ユーザー *John* に対して、テーブル *MusicCollection* で RestoreTableToPointInTime、GetItem、PutItem の API アクションを実行することを許可しています。

######のテキストを、リソース固有の情報に必ず置き換えてください。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:user/username"
      },
      "Action": [
        "dynamodb:RestoreTableToPointInTime",
        "dynamodb:GetItem",
        "dynamodb:PutItem"
      ],
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection"
    }
  ]
}
```

7. (オプション) 新しいポリシーがリソースへのパブリックアクセスおよびクロスアカウントアクセスにどのように影響するかをプレビューするには、[外部アクセスをプレビュー] を選択します。ポリシーを保存する前に、新しい IAM Access Analyzer の結果が導入されているかどうかや、既存の結果を解決するかどうかを確認できます。アクティブなアナライザーが表示されない場合は、[Access Analyzer へ移動] を選択し、[IAM Access Analyzer] でアカウントアナライザーを作成します。詳細については、「アクセスのプレビュー」を参照してください。
8. [テーブルの作成] を選択します。

AWS CloudFormation テンプレート

Using the AWS::DynamoDB::Table resource

次の CloudFormation テンプレートでは、AWS::DynamoDB::Table リソースを使用して、ストリームを有効にしたテーブルを作成します。このテンプレートでは、該当するテーブルとストリームの両方にアタッチされたリソースベースのポリシーも指定しています。

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "MusicCollectionTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "Artist",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "Artist",
                        "KeyType": "HASH"
                    }
                ],
                "BillingMode": "PROVISIONED",
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                },
                "StreamSpecification": {
                  "StreamViewType": "OLD_IMAGE",
                  "ResourcePolicy": {
                    "PolicyDocument": {
                      "Version": "2012-10-17",
                      "Statement": [
                        {
                            "Principal": {
                                "AWS": "arn:aws:iam::111122223333:user/John"
                            },
                            "Effect": "Allow",
                            "Action": [
                                "dynamodb:GetRecords",
                                "dynamodb:GetShardIterator",
                                "dynamodb:DescribeStream"
                            ],
                            "Resource": "*"
                        }
                      ]
                    }
```

```
                }
            },
            "TableName": "MusicCollection",
            "ResourcePolicy": {
                "PolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Principal": {
                                "AWS": [
                                    "arn:aws:iam::111122223333:user/John"
                                ]
                            },
                            "Effect": "Allow",
                            "Action": "dynamodb:GetItem",
                            "Resource": "*"
                        }
                    ]
                }
            }
        }
    }
}
```

Using the AWS::DynamoDB::GlobalTable resource

次の CloudFormation テンプレートでは、AWS::DynamoDB::GlobalTable リソースを使用してテーブルを作成し、リソースベースのポリシーを該当テーブルとそのストリームにアタッチします。

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "GlobalMusicCollection": {
            "Type": "AWS::DynamoDB::GlobalTable",
            "Properties": {
                "TableName": "MusicCollection",
                "AttributeDefinitions": [{
                    "AttributeName": "Artist",
                    "AttributeType": "S"
                }],
```

```
                "KeySchema": [{
                    "AttributeName": "Artist",
                    "KeyType": "HASH"
                }],
                "BillingMode": "PAY_PER_REQUEST",
                "StreamSpecification": {
                    "StreamViewType": "NEW_AND_OLD_IMAGES"
                },
                "Replicas": [
                    {
                        "Region": "us-east-1",
                        "ResourcePolicy": {
                            "PolicyDocument": {
                                "Version": "2012-10-17",
                                "Statement": [{
                                    "Principal": {
                                        "AWS": [
                                            "arn:aws:iam::111122223333:user/John"
                                        ]
                                    },
                                    "Effect": "Allow",
                                    "Action": "dynamodb:GetItem",
                                    "Resource": "*"
                                }]
                            }
                        },
                        "ReplicaStreamSpecification": {
                            "ResourcePolicy": {
                                "PolicyDocument": {
                                    "Version": "2012-10-17",
                                    "Statement": [{
                                        "Principal": {
                                            "AWS":
"arn:aws:iam::111122223333:user/John"
                                        },
                                        "Effect": "Allow",
                                        "Action": [
                                            "dynamodb:GetRecords",
                                            "dynamodb:GetShardIterator",
                                            "dynamodb:DescribeStream"
                                        ],
                                        "Resource": "*"
                                    }]
                                }
```

```
                            }
                        }
                    }
                ]
            }
        }
    }
}
```

DynamoDB の既存のテーブルにポリシーをアタッチする

DynamoDB コンソール、PutResourcePolicy API、AWS CLI、AWS SDK、または CloudFormation テンプレートを使用して、リソースベースのポリシーを既存のテーブルにアタッチしたり、既存のポリシーを変更したりできます。

AWS CLI の例: 新しいポリシーをアタッチする

次の IAM ポリシーの例では、put-resource-policy AWS CLI コマンドを使用して、リソースベースのポリシーを既存のテーブルにアタッチします。この例では、ユーザー *John* に対して、*MusicCollection* という名前の既存のテーブルで GetItem、PutItem、UpdateItem、UpdateTable の API アクションを実行することを許可しています。

*######*のテキストを、リソース固有の情報に必ず置き換えてください。

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:GetItem\",
                        \"dynamodb:PutItem\",
                        \"dynamodb:UpdateItem\",
                        \"dynamodb:UpdateTable\"
```

```
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS CLI の例: 既存のポリシーを条件に従って更新する

テーブルの既存のリソースベースポリシーを条件に従って更新する場合は、オプションの `expected-revision-id` パラメータを使用できます。次の例では、ポリシーが DynamoDB に存在し、現在のリビジョン ID が `expected-revision-id` パラメータに指定された値と一致する場合にのみ、ポリシーを更新します。

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --expected-revision-id 1709841168699 \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:GetItem\",
                        \"dynamodb:UpdateItem\",
                        \"dynamodb:UpdateTable\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection\"
                }
            ]
        }"
```

AWS マネジメントコンソール

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ダッシュボードから、既存のテーブルを選択します。

3. [アクセス許可] タブに移動し、[テーブルポリシーを作成] を選択します。
4. [リソースベースのポリシー] エディターで、アタッチするポリシーを追加し、[ポリシーを作成] を選択します。

   次の IAM ポリシーの例では、ユーザー *John* に対して、*MusicCollection* という名前の既存のテーブルで GetItem、PutItem、UpdateItem、UpdateTable の API アクションを実行することを許可しています。

   ######のテキストを、リソース固有の情報に必ず置き換えてください。

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:UpdateTable"
      ],
      "Resource": "arn:aws:dynamodb:us-
east-1:123456789012:table/MusicCollection"
    }
  ]
}
```

AWS SDK for Java 2.x

次の IAM ポリシーの例では、putResourcePolicy メソッドを使用して、リソースベースのポリシーを既存のテーブルにアタッチしています。このポリシーは、既存のテーブルで GetItem API アクションを実行することをユーザーに許可します。

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

```
import software.amazon.awssdk.services.dynamodb.model.DynamoDbException;
import software.amazon.awssdk.services.dynamodb.model.PutResourcePolicyRequest;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * Get started with the AWS SDK for Java 2.x
 */
public class PutResourcePolicy {

    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <tableArn> <allowedAWSPrincipal>

                Where:
                    tableArn - The Amazon DynamoDB table ARN to attach the policy to.
 For example, arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection.
                    allowedAWSPrincipal - Allowed AWS principal ARN that the example
 policy will give access to. For example, arn:aws:iam::123456789012:user/John.
                """;

        if (args.length != 2) {
            System.out.println(usage);
            System.exit(1);
        }

        String tableArn = args[0];
        String allowedAWSPrincipal = args[1];
        System.out.println("Attaching a resource-based policy to the Amazon DynamoDB
 table with ARN " +
                tableArn);
        Region region = Region.US_WEST_2;
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(region)
                .build();

        String result = putResourcePolicy(ddb, tableArn, allowedAWSPrincipal);
        System.out.println("Revision ID for the attached policy is " + result);
        ddb.close();
```

```
    }

    public static String putResourcePolicy(DynamoDbClient ddb, String tableArn, String
 allowedAWSPrincipal) {
        String policy = generatePolicy(tableArn, allowedAWSPrincipal);
        PutResourcePolicyRequest request = PutResourcePolicyRequest.builder()
                .policy(policy)
                .resourceArn(tableArn)
                .build();

        try {
            return ddb.putResourcePolicy(request).revisionId();
        } catch (DynamoDbException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        return "";
    }

    private static String generatePolicy(String tableArn, String allowedAWSPrincipal) {
        return "{\n" +
                "    \"Version\": \"2012-10-17\",\n" +,
                "    \"Statement\": [\n" +
                "        {\n" +
                "            \"Effect\": \"Allow\",\n" +
                "            \"Principal\": {\"AWS\":\"" + allowedAWSPrincipal + "\"},
\n" +
                "            \"Action\": [\n" +
                "                \"dynamodb:GetItem\"\n" +
                "            ],\n" +
                "            \"Resource\": \"" + tableArn + "\"\n" +
                "        }\n" +
                "    ]\n" +
                "}";
    }
}
```

リソースベースのポリシーを DynamoDB ストリームにアタッチする

DynamoDB コンソール、PutResourcePolicy API、AWS CLI、AWS SDK、または CloudFormation テンプレートを使用して、リソースベースのポリシーを既存のテーブルのストリームにアタッチしたり、既存のポリシーを変更したりできます。

> **Note**
>
> CreateTable API や UpdateTable API で作成中のストリームには、ポリシーをアタッチできません。ただし、テーブルの削除後にポリシーを変更または削除することはできます。また、無効になっているストリームのポリシーは変更または削除できます。

AWS CLI

次の IAM ポリシーの例では、put-resource-policy AWS CLI コマンドを使用して、*MusicCollection* という名前のテーブルのストリームにリソースベースのポリシーをアタッチしています。この例では、ユーザー *John* に対して、該当のストリームで GetRecords、GetShardIterator、DescribeStream の各 API アクションの実行を許可しています。

*######*のテキストを、リソース固有の情報に必ず置き換えてください。

```
aws dynamodb put-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492 \
    --policy \
        "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
              {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"AWS\": \"arn:aws:iam::111122223333:user/John\"
                    },
                    \"Action\": [
                        \"dynamodb:GetRecords\",
                        \"dynamodb:GetShardIterator\",
                        \"dynamodb:DescribeStream\"
                    ],
                    \"Resource\": \"arn:aws:dynamodb:us-
west-2:123456789012:table/MusicCollection/stream/2024-02-12T18:57:26.492\"
                }
            ]
        }"
```

AWS マネジメントコンソール

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. DynamoDB コンソールのダッシュボードで [テーブル] を選択し、既存のテーブルを選択します。

   ストリームが有効になっているテーブルを選択してください。テーブルのストリームを有効にする方法については、「ストリームの有効化」を参照してください。
3. [アクセス許可] タブを選択します。
4. [アクティブストリームについてのリソースベースのポリシー] で、[ストリームポリシーを作成] を選択します。
5. [リソースベースのポリシー] エディターで、ストリームに対するアクセス許可を定義するポリシーを追加します。このポリシーでは、ストリームにアクセスできるユーザーと、ストリームに対して実行できるアクションを指定します。ポリシーを追加するには、以下のいずれかを実行します。

   - JSON ポリシードキュメントを入力するか貼り付けます。IAM ポリシー言語の詳細については、「IAM ユーザーガイド」の「JSON エディターを使用したポリシーの作成」を参照してください。

     > **Tip**
     >
     > 「Amazon DynamoDB デベロッパーガイド」のリソースベースのポリシーの例を確認するには、[ポリシーの例] を選択してください。

   - [新しいステートメントを追加] を選択して新しいステートメントを追加し、提示されたフィールドに情報を入力します。このステップを、追加するステートメントの数だけ繰り返します。

     > **Important**
     >
     > セキュリティ警告、エラー、提案がある場合は、ポリシーを保存する前に解決してください。
6. (オプション) 新しいポリシーがリソースへのパブリックアクセスおよびクロスアカウントアクセスにどのように影響するかをプレビューするには、[外部アクセスをプレビュー] を選択します。ポリシーを保存する前に、新しい IAM Access Analyzer の結果が導入されているかどうかや、既

存の結果を解決するかどうかを確認できます。アクティブなアナライザーが表示されない場合は、[Access Analyzer へ移動] を選択し、[IAM Access Analyzer] でアカウントアナライザーを作成します。詳細については、「アクセスのプレビュー」を参照してください。

7. [Create policy] (ポリシーの作成) を選択します。

次の IAM ポリシーの例では、*MusicCollection* という名前のテーブルのストリームにリソースベースのポリシーをアタッチします。この例では、ユーザー *John* に対して、該当のストリームで GetRecords、GetShardIterator、DescribeStream の各 API アクションの実行を許可しています。

*######*のテキストを、リソース固有の情報に必ず置き換えてください。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action": [
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator",
        "dynamodb:DescribeStream"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492"
      ]
    }
  ]
}
```

リソースベースのポリシーを DynamoDB テーブルから削除する

DynamoDB コンソール、DeleteResourcePolicy API、AWS CLI、AWS SDK、または CloudFormation テンプレートを使用して、既存のテーブルからリソースベースのポリシーを削除できます。

AWS CLI

次の例では、delete-resource-policy AWS CLI コマンドを使用して、*MusicCollection* という名前のテーブルからリソースベースのポリシーを削除しています。

######のテキストを、リソース固有の情報に必ず置き換えてください。

```
aws dynamodb delete-resource-policy \
    --resource-arn arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection
```

AWS マネジメントコンソール

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. DynamoDB コンソールのダッシュボードで [テーブル] を選択し、既存のテーブルを選択します。
3. [アクセス許可] を選択します。
4. [ポリシーを管理] ドロップダウンから [ポリシーを削除] を選択します。
5. [テーブルについてのリソースベースのポリシーを削除] ダイアログボックスで、「**confirm**」と入力して削除アクションを確定します。
6. [削除] を選択します。

DynamoDB でリソースベースのポリシーを使用したクロスアカウントアクセス

リソースベースのポリシーを使用して、異なる AWS アカウント で利用可能なリソースへのクロスアカウントアクセスを許可できます。リソースベースのポリシーで許可されているクロスアカウントアクセスについては、IAM Access Analyzer が該当リソースと同じ AWS リージョン にある場合は、IAM Access Analyzer で外部アクセスの検出結果としてすべて報告されます。IAM Access Analyzer は、IAM ポリシーの文法とベストプラクティスに照らしてポリシーチェックを行います。これらのチェックにより、機能的でセキュリティのベストプラクティスに準拠したポリシーを作成するのに、役立つ結果と実行可能なレコメンデーションが示されます。IAM Access Analyzer のアクティブな検出結果を DynamoDB コンソールの [アクセス許可] タブで表示できます。

IAM Access Analyzer を使用したポリシーの検証の詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer ポリシーの検証」を参照してください。IAM Access Analyzer によって返さ

れる警告、エラー、および提案のリストを表示するには、「IAM Access Analyzer ポリシーチェックリファレンス」を参照してください。

アカウント A のユーザー A に、アカウント B のテーブル B にアクセスするための GetItem 権限を付与するには、以下の手順を実行します。

1. GetItem アクションの実行権限をユーザー A に付与するリソースベースのポリシーを、テーブル B にアタッチします。
2. テーブル B に対して GetItem アクションを実行する権限をユーザー A に付与するアイデンティティベースのポリシーを、ユーザー A にアタッチします。

DynamoDB コンソールにある [外部アクセスをプレビュー] オプションを使用すると、リソースへのパブリックアクセスとクロスアカウントアクセスに新しいポリシーがどのように影響するかをプレビューできます。ポリシーを保存する前に、新しい IAM Access Analyzer の結果が導入されているかどうかや、既存の結果を解決するかどうかを確認できます。アクティブなアナライザーが表示されない場合は、[Access Analyzer へ移動] を選択し、[IAM Access Analyzer] でアカウントアナライザーを作成します。詳細については、「アクセスのプレビュー」を参照してください。

DynamoDB のデータプレーンとコントロールプレーンの API のテーブル名パラメータには、クロスアカウントのオペレーションをサポートするため、テーブルの完全な Amazon リソースネーム (ARN) を指定できます。完全な ARN ではなくテーブル名パラメータのみを指定した場合、API オペレーションは、リクエスタが属するアカウントの該当テーブルに対して実行されます。クロスアカウントアクセスを使用するポリシーの例については、「クロスアカウントアクセスのリソースベースのポリシー」を参照してください。

リソース所有者のアカウントは、別のアカウントのプリンシパルが所有者のアカウントの DynamoDB テーブルに対して読み書きを実行している場合にも課金されます。テーブルにプロビジョンドスループットが指定されている場合、所有者アカウントと他のアカウントのリクエスタからのすべてのリクエストの合計によって、リクエストをスロットリングするか (自動スケーリングが無効な場合)、スケールアップ/スケールダウンするか (自動スケーリングが有効な場合) が決まります。

リクエストは所有者アカウントとリクエスタアカウントの両方の CloudTrail ログに記録されるため、2 つのアカウントはそれぞれ、どちらのアカウントがどのデータにアクセスしたかを追跡できます。

クロスアカウントの AWS Lambda 関数を使用したアクセスを共有する

アカウント A の Lambda 関数

1. IAM コンソールに移動して、アカウント A のAWS Lambda 関数の Lambda 実行ロールとして使用される IAM ロールを作成します。必要な DynamoDB Streams と Lambda 呼び出しアクセス許可を持つマネージド IAM ポリシー `AWSLambdaDynamoDBExecutionRole` を追加します。このポリシーは、アカウント A でユーザーがアクセスする可能性のあるすべての DynamoDB Streams リソースへのアクセスも付与します。
2. Lambda コンソールで、DynamoDB ストリーム内のレコードを処理する AWS Lambda 関数を作成し、実行ロールのセットアップ中に、前のステップで作成したロールを選択します。
3. Lambda 関数の実行ロールをアカウント B の DynamoDB Streams の所有者に提供して、クロスアカウント読み取りアクセス用のリソースベースのポリシーを設定します。
4. Lambda 関数のセットアップを完了します。

アカウント B の DynamoDB ストリーム

1. Lambda 関数を呼び出すアカウント A からクロスアカウントの Lambda 実行ロールを取得します。
2. アカウント B の Amazon DynamoDB コンソールで、Lambda クロスアカウントトリガーのテーブルを選択します。[エクスポートおよびストリーム] タブで、DynamoDB ストリーム ARN を見つけます。DynamoDB Streams のステータスがオンになっていることを確認し、リソースポリシーに必要なフルストリーム ARN を書き留めます。
3. [アクセス許可] タブで、[ストリームポリシーを作成] を選択してビジュアルポリシーエディタを起動します。[新しいステートメントを追加] を選択するか、既に存在する場合はポリシーを編集します。
4. アカウント A の Lambda 実行ロールをプリンシパルとして指定するポリシーを作成し、必要な DynamoDB Streams アクションを付与します。`dynamodb:DescribeStream`、`dynamodb:GetRecords`、`dynamodb:GetShardIterator` アクションを必ず含めてください。DynamoDB Streams のリソースポリシーの例の詳細については、「DynamoDB リソースベースのポリシーの例」を参照してください。

> **Note**
>
> コントロールプレーン API のクロスアカウントアクセスでは、1 秒あたりのトランザクション数 (TPS) の上限が低く、500 リクエストです。

DynamoDB でリソースベースのポリシーでパブリックアクセスをブロックする

ブロックパブリックアクセス (BPA) は、Amazon Web Services (AWS) アカウント全体で DynamoDB のテーブル、インデックス、ストリームへのパブリックアクセスを許可するリソースベースのポリシーを特定し、アタッチされないように防ぐ機能です。BPA を使用すると、DynamoDB リソースへのパブリックアクセスを阻止できます。BPA はリソースベースのポリシーの作成時や変更時にチェックを行い、DynamoDB のセキュリティ体制の改善に貢献します。BPA は、これらのオペレーション中に自動的に適用されます。BPA を有効にするための個別の設定や構成はありません。

BPA は自動推論を活用して、リソースベースのポリシーで許可されるアクセスを分析し、該当するアクセス許可がリソースベースのポリシーの管理時に見つかった場合は警告します。分析では、リソースベースのポリシーのステートメント、アクション、ポリシーで使用されている条件キーセットをすべてまたいでアクセスを検証します。

> **Important**
>
> BPA は、DynamoDB リソース (テーブル、インデックス、ストリームなど) に直接アタッチされたリソースベースのポリシーによってパブリックアクセスが許可されることがないように防いで、リソースを保護します。BPA を利用したうえでさらに、次のポリシーを注意深く調べて、パブリックアクセスが許可されていないことを確かめてください。
>
> - 関連する AWS プリンシパル (IAM ロールなど) にアタッチされているアイデンティティベースのポリシー
> - 関連する AWS リソース (AWS Key Management Service (KMS) キーなど) にアタッチされているリソースベースのポリシー

プリンシパル に * エントリが含まれていないこと、または指定された条件キーのいずれかがプリンシパルからリソースへのアクセスを制限していることを確認する必要があります。リソースベースのポリシーが AWS アカウント 全体でテーブル、インデックス、またはストリームへのパブリックアクセスを許可している場合、ポリシー内の指定内容が修正され、非パブリックと判断されない限り、そのポリシーの作成または変更は阻止されます。

`Principal` ブロック内に 1 つ以上のプリンシパルを指定することで、ポリシーを非パブリックにすることができます。次のリソースベースのポリシーの例では、2 つのプリンシパルを指定することで、パブリックアクセスをブロックしています。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": [
      "123456789012",
      "111122223333"
    ]
  },
  "Action": "dynamodb:*",
  "Resource": "*"
}
```

特定の条件キーを指定してアクセスを制限するポリシーも、パブリックとは見なされません。リソースベースのポリシーで指定されているプリンシパルを評価するほかに、以下の信頼できる条件キーも、リソースベースのポリシーが付与するアクセスが非パブリックであるという評価の補足に使用できます。

- aws:PrincipalAccount
- aws:PrincipalArn
- aws:PrincipalOrgID
- aws:PrincipalOrgPaths
- aws:SourceAccount
- aws:SourceArn
- aws:SourceVpc
- aws:SourceVpce
- aws:UserId
- aws:PrincipalServiceName
- aws:PrincipalServiceNamesList
- aws:PrincipalIsAWSService
- aws:Ec2InstanceSourceVpc
- aws:SourceOrgID
- aws:SourceOrgPaths

さらに、リソースベースのポリシーが非パブリックとなるには、Amazon リソースネーム (ARN) と文字列キーの値にワイルドカードや変数が含まれていてはいけません。リソースベースのポリシーで

aws:PrincipalIsAWSService キーが使用されている場合は、キー値を true に設定する必要があります。

次のポリシーでは、指定されたアカウントのユーザー John にアクセスが限定されています。この条件により Principal が制限されるため、パブリックとは見なされません。

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "*"
  },
  "Action": "dynamodb:*",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "aws:PrincipalArn": "arn:aws:iam::123456789012:user/John"
    }
  }
}
```

次の例は非パブリックのリソースベースのポリシーです。StringEquals 演算子を使用して sourceVPC を制限しています。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "dynamodb:*",
      "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/
MusicCollection",
      "Condition": {
        "StringEquals": {
          "aws:SourceVpc": [
            "vpc-91237329"
          ]
        }
```

```
      }
    }
  ]
}
```

リソースベースのポリシーでサポートされる DynamoDB API オペレーション

このトピックでは、リソースベースのポリシーでサポートされる API オペレーションを示します。ただし、クロスアカウントアクセスの場合、リソースベースのポリシーで使用できる DynamoDB API は特定のセットに限られます。リソースベースのポリシーをリソースタイプ (バックアップやインポートなど) にアタッチすることはできません。これらのリソースタイプで動作する API に対応する IAM アクションは、リソースベースのポリシーでサポートされる IAM アクションからは除外されます。テーブル管理者は同一アカウント内で内部テーブル設定を構成するため、UpdateTimeToLive や DisableKinesisStreamingDestination などの API は、リソースベースのポリシーによるクロスアカウントアクセスに対応していません。

クロスアカウントアクセスに対応した DynamoDB データプレーン API とコントロールプレーン API では、テーブル名のオーバーロードにも対応しています。そのため、テーブル名の代わりにテーブル ARN を指定することができます。これらの API の `TableName` パラメータにテーブル ARN を指定できます。ただし、これらの API がすべてクロスアカウントアクセスに対応しているわけではありません。

トピック

- データプレーンの API オペレーション
- PartiQL API オペレーション
- コントロールプレーンの API オペレーション
- バージョン 2019.11.21 (現行) のグローバルテーブルの API オペレーション
- バージョン 2017.11.29 (レガシー) のグローバルテーブルの API オペレーション
- タグの API オペレーション
- バックアップと復元の API オペレーション
- 継続的バックアップ/復元 (PITR) の API オペレーション
- 寄稿者のインサイトの API オペレーション
- エクスポートの API オペレーション
- インポートの API オペレーション

- Amazon Kinesis Data Streams の API オペレーション
- リソースベースのポリシーの API オペレーション
- Time-to-Live の API オペレーション
- その他の API オペレーション
- DynamoDB Streams の API オペレーション

データプレーンの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、データプレーンの API オペレーションが提供する API レベルのサポートをまとめています。

データプレーン - テーブル/インデックス API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DeleteItem	あり	はい
GetItem	あり	はい
PutItem	あり	はい
クエリ:	あり	はい
Scan	あり	はい
UpdateItem	あり	はい
TransactGetItems	あり	はい
TransactWriteItems	あり	はい
BatchGetItem	あり	はい
BatchWriteItem	あり	はい

PartiQL API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、PartiQL API オペレーションが提供する API レベルのサポートをまとめています。

PartiQL API	リソースベースのポリシーのサポート	クロスアカウントのサポート
BatchExecuteStatement	はい	いいえ
ExecuteStatement	はい	いいえ
ExecuteTransaction	はい	いいえ

コントロールプレーンの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、コントロールプレーンの API オペレーションが提供する API レベルのサポートをまとめています。

コントロールプレーン - テーブル API	リソースベースのポリシーのサポート	クロスアカウントのサポート
CreateTable	いいえ	いいえ
DeleteTable	あり	はい
DescribeTable	あり	はい
UpdateTable	あり	はい

バージョン 2019.11.21 (現行) のグローバルテーブルの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、バージョン 2019.11.21 (現行) のグローバルテーブルの API オペレーションが提供する API レベルのサポートをまとめています。

バージョン 2019.11.21 (現行) のグローバルテーブル API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeTableReplicaAutoScaling	はい	いいえ

バージョン 2019.11.21 (現行) のグローバルテーブル API	リソースベースのポリシーのサポート	クロスアカウントのサポート
UpdateTableReplicaAutoScaling	はい	いいえ

バージョン 2017.11.29 (レガシー) のグローバルテーブルの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、バージョン 2017.11.29 (レガシー) のグローバルテーブルの API オペレーションが提供する API レベルのサポートをまとめています。

バージョン 2017.11.29 (レガシー) のグローバルテーブル API	リソースベースのポリシーのサポート	クロスアカウントのサポート
CreateGlobalTable	いいえ	いいえ
DescribeGlobalTable	いいえ	いいえ
DescribeGlobalTableSettings	いいえ	いいえ
ListGlobalTables	いいえ	いいえ
UpdateGlobalTable	いいえ	いいえ
UpdateGlobalTableSettings	いいえ	いいえ

タグの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、タグに関連する API オペレーションが提供する API レベルのサポートをまとめています。

タグの API	リソースベースのポリシーのサポート	クロスアカウントのサポート
ListTagsOfResource	あり	はい

タグの API	リソースベースのポリシーのサポート	クロスアカウントのサポート
TagResource	あり	はい
UntagResource	あり	はい

バックアップと復元の API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、バックアップと復元に関連する API オペレーションが提供する API レベルのサポートをまとめています。

バックアップと復元の API	リソースベースのポリシーのサポート	クロスアカウントのサポート
CreateBackup	はい	いいえ
DescribeBackup	いいえ	いいえ
DeleteBackup	いいえ	いいえ
RestoreTableFromBackup	いいえ	いいえ

継続的バックアップ/復元 (PITR) の API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、継続的バックアップ/復元 (PITR) に関連する API オペレーションが提供する API レベルのサポートをまとめています。

継続的バックアップ/復元 (PITR) の API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeContinuousBackups	はい	いいえ
RestoreTableToPointInTime	はい	いいえ
UpdateContinuousBackups	はい	いいえ

寄稿者のインサイトの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、継続的バックアップ/復元 (PITR) に関連する API オペレーションが提供する API レベルのサポートをまとめています。

寄稿者のインサイトの API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeContributorInsights	はい	いいえ
ListContributorInsights	いいえ	いいえ
UpdateContributorInsights	はい	いいえ

エクスポートの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、エクスポートの API オペレーションが提供する API レベルのサポートをまとめています。

エクスポート API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeExport	いいえ	いいえ
ExportTableToPointInTime	はい	いいえ
ListExports	いいえ	いいえ

インポートの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、インポートの API オペレーションが提供する API レベルのサポートをまとめています。

インポート API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeImport	いいえ	いいえ

インポート API	リソースベースのポリシーのサポート	クロスアカウントのサポート
ImportTable	いいえ	いいえ
ListImports	いいえ	いいえ

Amazon Kinesis Data Streams の API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、Kinesis Data Streams の API オペレーションが提供する API レベルのサポートをまとめています。

Kinesis API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeKinesisStreamingDestination	はい	いいえ
DisableKinesisStreamingDestination	はい	いいえ
EnableKinesisStreamingDestination	はい	いいえ
UpdateKinesisStreamingDestination	はい	いいえ

リソースベースのポリシーの API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、リソースベースのポリシーの API オペレーションが提供する API レベルのサポートをまとめています。

リソースベースのポリシーの API	リソースベースのポリシーのサポート	クロスアカウントのサポート
GetResourcePolicy	はい	いいえ

リソースベースのポリシーの API	リソースベースのポリシーのサポート	クロスアカウントのサポート
PutResourcePolicy	はい	いいえ
DeleteResourcePolicy	はい	いいえ

Time-to-Live の API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、Time to Live (TTL) の API オペレーションが提供する API レベルのサポートをまとめています。

TTL API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeTimeToLive	はい	いいえ
UpdateTimeToLive	はい	いいえ

その他の API オペレーション

次の表は、リソースベースのポリシーとクロスアカウントアクセスに対して、その他の API オペレーションが提供する API レベルのサポートをまとめています。

その他の API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeLimits	いいえ	いいえ
DescribeEndpoint	いいえ	いいえ
ListBackups	いいえ	いいえ
ListTables	いいえ	いいえ

DynamoDB Streams の API オペレーション

次の表には、リソースベースのポリシーとクロスアカウントアクセスに対する DynamoDB Streams API の API レベルのサポートをまとめています。

DynamoDB Streams API	リソースベースのポリシーのサポート	クロスアカウントのサポート
DescribeStream	あり	はい
GetRecords	あり	はい
GetShardIterator	あり	はい
ListStreams	いいえ	なし

IAM アイデンティティベースのポリシーと DynamoDB リソースベースのポリシーによる認可

アイデンティティベースのポリシーはアイデンティティ (IAM ユーザー、ユーザーグループ、ロールなど) にアタッチされます。これらは、アイデンティティが実行できるアクション、実行対象となるリソース、実行条件を制御する IAM ポリシードキュメントです。アイデンティティベースのポリシーには、管理ポリシーまたはインラインポリシーがあります。

リソースベースのポリシーは、リソース (DynamoDB テーブルなど) にアタッチする IAM ポリシードキュメントです。これらのポリシーでは、そのリソースに対して特定のアクションを実行するために指定されたプリンシパルのアクセス許可を付与するとともに、このアクセス許可が適用される条件を定義します。例えば、DynamoDB テーブルのリソースベースのポリシーには、そのテーブルに関連付けられたインデックスも含まれます。リソースベースのポリシーはインラインポリシーです。マネージド型のリソースベースのポリシーはありません。

これらのポリシーの詳細については、「IAM ユーザーガイド」の「アイデンティティベースおよびリソースベースのポリシー」を参照してください。

IAM プリンシパルがリソース所有者と同じアカウントに属している場合、そのリソースへのアクセス権限の指定には、リソースベースのポリシーで事足ります。さらに、IAM アイデンティティベースのポリシーをリソースベースのポリシーと併せて用意しておくこともできます。クロスアカウントアクセスの場合は、アイデンティティポリシーとリソースポリシーの両方でアクセスを明示的に許可

する必要があります。詳細については、「DynamoDB でリソースベースのポリシーを使用したクロスアカウントアクセス」に記載されています。両タイプのポリシーを併用する場合、「アカウント内でのリクエストの許可または拒否の決定」の解説通りにポリシーが評価されます。

> **Important**
>
> アイデンティティベースのポリシーが DynamoDB テーブルへの無条件アクセス (条件なしの `dynamodb:GetItem` など) を許可する場合、`dynamodb:Attributes` への条件付きアクセスを許可するリソースベースのポリシーは、そのアクセスを制限しません。アイデンティティベースのポリシーの無条件許可が優先され、リソースベースのポリシーの条件は制限として適用されません。特定の属性へのアクセスを制限するには、リソースベースのポリシーの条件付き `Deny` ステートメントのみに依存するのではなく、明示的な `Allow` ステートメントを使用します。

DynamoDB リソースベースのポリシーの例

リソースベースのポリシーの `Resource` フィールドで ARN を指定した場合は、その ARN がアタッチ先の DynamoDB リソースの ARN と一致する場合にのみ、ポリシーが有効になります。

> **Note**
>
> *######*のテキストを、リソース固有の情報に必ず置き換えてください。

テーブルのリソースベースのポリシー

次のリソースベースのポリシーは、*MusicCollection* という名前の DynamoDB テーブルにアタッチされています。IAM ユーザーの *John* と *Jane* に対して、*MusicCollection* リソースで GetItem アクションと BatchGetItem アクションを実行するアクセス許可を付与します。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
```

```
        "Effect": "Allow",
        "Principal": {
          "AWS": [
            "arn:aws:iam::111122223333:user/username",
            "arn:aws:iam::111122223333:user/Jane"
          ]
        },
        "Action": [
          "dynamodb:GetItem",
          "dynamodb:BatchGetItem"
        ],
        "Resource": [
          "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection"
        ]
    }
  ]
}
```

ストリームのリソースベースのポリシー

次のリソースベースのポリシーは、2024-02-12T18:57:26.492 という名前の DynamoDB ストリームにアタッチされています。IAM ユーザーの *John* と *Jane* に対して、2024-02-12T18:57:26.492 リソースで GetRecords、GetShardIterator、DescribeStream の各 API アクションを実行するアクセス許可を付与します。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111122223333:user/username",
          "arn:aws:iam::111122223333:user/Jane"
        ]
      },
```

```
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
stream/2024-02-12T18:57:26.492"
      ]
    }
  ]
}
```

指定されたリソースに対するあらゆるアクションの実行権限を付与するリソースベースのポリシー

テーブルと、テーブルに関連付けられたすべてのインデックスに対するすべてのアクションの実行をユーザーに許可する場合は、ワイルドカード (*) を使用して、テーブルに関連するアクションとリソースを表すことができます。リソースにワイルドカード文字を使用した場合、該当する DynamoDB テーブルとそのすべての関連インデックス (未作成のものも含む) へのアクセスがユーザーに許可されます。例えば、次のポリシーは、ユーザー *John* に対して、*MusicCollection* テーブルとそのすべてのインデックス (今後作成されるインデックスを含む) に対するあらゆるアクションの実行権限を付与します。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/username"
      },
      "Action": "dynamodb:*",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection",
```

```
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/
index/index-name"
      ]
    }
  ]
}
```

クロスアカウントアクセスのリソースベースのポリシー

クロスアカウントの IAM アイデンティティに対して、DynamoDB リソースへのアクセス権限を指定できます。例えば、信頼できるアカウントのユーザーが、特定の項目とその項目内の特定の属性のみにアクセスするという条件で、テーブルの内容の読み取り権限を取得できるようにする場合が考えられます。次のポリシーは、信頼できる AWS アカウント ID *111111111111* のユーザー *John* に対して、GetItem API を使用してアカウント *123456789012* のテーブルのデータにアクセスすることを許可します。このポリシーでは、プライマリキーが *Jane* の項目のみに該当ユーザーがアクセスして、属性 `Artist` と `SongTitle` のみを取得可能であり、その他の属性は取得できないようになっています。

> **Important**
>
> `SPECIFIC_ATTRIBUTES` 条件を指定しない場合は、返された項目のすべての属性が表示されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CrossAccountTablePolicy",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111111111111:user/John"
            },
            "Action": "dynamodb:GetItem",
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection"
```

```
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": "Jane",
                    "dynamodb:Attributes": [
                        "Artist",
                        "SongTitle"
                    ]
                },
                "StringEquals": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

前述のリソースベースのポリシーに加えて、クロスアカウントアクセスを実現するためには、ユーザー *John* にアタッチしたアイデンティティベースのポリシーでも GetItem API アクションを許可する必要があります。以下は、ユーザー *John* にアタッチする必要があるアイデンティティベースのポリシーの例です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CrossAccountIdentityBasedPolicy",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": "Jane",
                    "dynamodb:Attributes": [
                        "Artist",
```

```
                        "SongTitle"
                    ]
                },
                "StringEquals": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

ユーザー John は、アカウント *123456789012* のテーブル *MusicCollection* にアクセスするため、table-name パラメータにテーブル ARN を指定して GetItem リクエストを行うことができます。

```
aws dynamodb get-item \
    --table-name arn:aws:dynamodb:us-west-2:123456789012:table/MusicCollection \
    --key '{"Artist": {"S": "Jane"}' \
    --projection-expression 'Artist, SongTitle' \
    --return-consumed-capacity TOTAL
```

IP アドレス条件を指定したリソースベースのポリシー

条件を適用して、発信元の IP アドレス、仮想プライベートクラウド (VPC)、VPC エンドポイント (VPCE) を制限できます。リクエストの発信元のアドレスに基づいてアクセス許可を指定できます。例えば、特定の IP ソース (企業の VPN エンドポイントなど) がアクセス元である場合に限り、ユーザーに DynamoDB リソースへのアクセスを許可したい場合が考えられます。これらの IP アドレスを Condition ステートメントに指定します。

次の例では、ユーザー *John* に対して、発信元 IP が 54.240.143.0/24 および 2001:DB8:1234:5678::/64 である場合に任意の DynamoDB リソースへのアクセスを許可します。

JSON

```
{
  "Id":"PolicyId2",
  "Version":"2012-10-17",
  "Statement":[
    {
```

```
      "Sid":"AllowIPmix",
      "Effect":"Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111111111111:user/username"
      },
      "Action":"dynamodb:*",
      "Resource":"*",
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": [
            "54.240.143.0/24",
            "2001:DB8:1234:5678::/64"
          ]
        }
      }
    }
  ]
}
```

また、発信元が特定の VPC エンドポイント (*vpce-1a2b3c4d* など) 以外である場合に、DynamoDB リソースへのアクセスを一切拒否することもできます。

> **Important**
>
> `"Principal": "*"` を使用する Deny ステートメントのみを含むリソースベースのポリシーは、後でポリシーを更新または削除する機能を含め、すべてのプリンシパルへのアクセスを拒否します。この方法でロックアウトするポリシーをアタッチすると、DynamoDB は「新しいリソースポリシーでは、今後リソースポリシーを更新することはできません」というエラーでポリシーを拒否します。別の `Allow` ステートメントを追加してもエラーは解決されません。これは、明示的な Deny は常に `Allow` を上書きするためです。ポリシーを管理する機能を維持しながら、特定の VPC エンドポイント以外のすべてのユーザーへのアクセスを拒否するには、次の例に示すように、`aws:PrincipalArn` に `ArnNotEquals` 条件を追加して、管理プリンシパルを Deny ステートメントから除外します。

> **Important**
>
> IPv6 専用の環境で IP ベースのリソースポリシーを持つ DynamoDB テーブルで DAX を使用する場合は、追加のアクセスルールを設定する必要があります。リソースポリシー

がテーブルの IPv4 アドレス空間 `0.0.0.0/0` へのアクセスを制限する場合は、DAX クラスターに関連付けられた IAM ロールのアクセスを許可する必要があります。ポリシーに `ArnNotEquals` 条件を追加して、DAX が DynamoDB テーブルへのアクセスを維持できるようにします。詳細については、「DAX と IPv6」を参照してください。

JSON

```
{
  "Id":"PolicyId",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AccessToSpecificVPCEOnly",
      "Principal": "*",
      "Action": "dynamodb:*",
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringNotEquals":{
          "aws:sourceVpce":"vpce-1a2b3c4d"
        },
        "ArnNotEquals":{
          "aws:PrincipalArn":"arn:aws:iam::123456789012:role/AdminRole"
        }
      }
    }
  ]
}
```

IAM ロールを使用するリソースベースのポリシー

リソースベースのポリシーで IAM サービスロールを指定することもできます。このロールを引き受ける IAM エンティティは、そのロールに指定されているアクションのみを、リソースベースのポリシーで指定されているリソースセットに限定して実行できます。

次の例では、IAM エンティティが *MusicCollection* と *MusicCollection* の DynamoDB リソースですべての DynamoDB アクションを実行することを許可しています。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "1111",
      "Effect": "Allow",
      "Principal": { "AWS": "arn:aws:iam::111122223333:role/role-name" },
      "Action": "dynamodb:*",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection",
        "arn:aws:dynamodb:us-east-1:123456789012:table/MusicCollection/*"
      ]
    }
  ]
}
```

DynamoDB リソースベースのポリシーの考慮事項

DynamoDB リソースに対してリソースベースのポリシーを定義する際には、以下の点を考慮してください。

一般的な考慮事項

- リソースベースのポリシードキュメントでサポートされる最大サイズは 20 KB です。DynamoDB では、この上限に照らしてポリシーのサイズを計算する際に空白はカウントされません。
- 特定のリソースのポリシーの更新が 2 回目以降の場合、同じリソースに対するそのポリシーの前回の更新が正常に終わってから 15 秒間は更新できません。
- 現時点では、リソースベースのポリシーは既存のストリームにのみアタッチできます。作成中のストリームにポリシーをアタッチすることはできません。

グローバルテーブルに関する考慮事項

- リソースベースのポリシーは、グローバルテーブルバージョン 2017.11.29 (レガシー) のレプリカではサポートされていません。

- リソースベースのポリシー内で、グローバルテーブルのデータをレプリケートするアクションが DynamoDB のサービスにリンクされたロール (SLR) に対して拒否されている場合、レプリカの追加または削除はエラーで失敗します。
- AWS::DynamoDB::GlobalTable リソースでは、スタックの更新を行うリージョン以外のリージョンでは、スタックの更新時にレプリカの作成と、そのレプリカへのリソースベースのポリシーの追加を同時に行うことはできません。

クロスアカウントに関する考慮事項

- リソースベースのポリシーを使用したクロスアカウントアクセスでは、AWS マネージドキーで暗号化したテーブルはサポートされません。AWS KMS のマネージドポリシーへのクロスアカウントアクセスは許可できないからです。

CloudFormation に関する考慮事項

- リソースベースのポリシーはドリフト検出には対応していません。AWS CloudFormation スタックテンプレートの外部でリソースベースのポリシーを更新した場合は、CloudFormation スタックに変更内容を反映させる必要があります。
- リソースベースのポリシーは、アウトオブバンドの変更に対応していません。CloudFormation テンプレートの外部でポリシーを追加、更新、または削除しても、テンプレート内のポリシーに変更がない限り、変更は上書きされません。

  例えば、テンプレートに含まれているリソースベースのポリシーを、後からテンプレートの外部で更新したとします。テンプレート内のポリシーを変更しない限り、DynamoDB 内の更新済みポリシーはテンプレート内のポリシーと同期されません。

  逆に、テンプレートにリソースベースのポリシーは含まれていないが、テンプレートの外部でポリシーを追加したとします。このポリシーは、テンプレートに追加しない限りは、DynamoDB から削除されることはありません。テンプレートにポリシーを追加してスタックを更新すると、DynamoDB の既存のポリシーが、テンプレートで定義されているポリシーと一致するように更新されます。

DynamoDB リソースベースのポリシーに関するベストプラクティス

このトピックでは、DynamoDB リソースに対するアクセス許可と、それらのリソースに対して実行できるアクションを定義する際のベストプラクティスについて説明します。

DynamoDB リソースへのアクセス制御を簡素化する

DynamoDB リソースにアクセスする必要がある AWS Identity and Access Management プリンシパルがリソース所有者と同じ AWS アカウント に属している場合、プリンシパルごとに IAM アイデンティティベースポリシーを用意する必要はありません。該当するリソースにリソースベースのポリシーをアタッチすれば十分です。このように構成すれば、アクセス制御が簡単になります。

リソースベースのポリシーで DynamoDB リソースを保護する

DynamoDB のすべてのテーブルとストリームに対してリソースベースのポリシーを作成し、これらのリソースへのアクセスを制御します。リソースベースのポリシーのおかげで、アクセス許可をリソース単位で一元管理し、DynamoDB のテーブル、インデックス、ストリームへのアクセス制御を簡素化し、管理オーバーヘッドを削減できます。リソースベースのポリシーがテーブルまたはストリームに対して指定されていない場合、そのテーブルやストリームへのアクセスは、IAM プリンシパルに関連付けられたアイデンティティベースのポリシーでアクセスが許可されていない限り、暗黙的に拒否されます。

最小特権アクセス許可を適用する

DynamoDB リソースに対するリソースベースのポリシーでアクセス許可を設定するときは、アクションの実行に必要なアクセス許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。その際、ワークロードやユースケースに必要なアクセス許可を検討しながら、大まかなアクセス許可から始めるとよいでしょう。ユースケースが成熟してきたら、最小特権になるように付与する権限を減らしていくことができます。

最小特権ポリシーの生成のためにクロスアカウントアクセスアクティビティを分析する

IAM Access Analyzer は、リソースベースのポリシーで指定された外部エンティティへのクロスアカウントアクセスを報告します。また、情報が可視化されるため、アクセス許可を調整し、最小特権の原則を守るうえでも役立ちます。ポリシー生成の詳細については、「IAM Access Analyzer ポリシーの生成」を参照してください。

IAM Access Analyzer を使用して最小特権ポリシーを生成する

タスクの実行に必要なアクセス許可のみを付与するには、AWS CloudTrail にログインしているアクセスアクティビティに基づいてポリシーを生成することができます。IAM Access Analyzer は、ポリシーで使用されるサービスとアクションを分析します。

DynamoDB での属性ベースのアクセス制御の使用

属性ベースのアクセス制御 (ABAC) は、アイデンティティベースのポリシーまたは他の AWS ポリシー (リソースベースのポリシーや組織の IAM ポリシーなど) のタグ条件に基づいてアクセス許可を定義する認可戦略です。DynamoDB テーブルにタグをアタッチすると、タグベースの条件に対して評価されます。テーブルに関連付けられたインデックスは、テーブルに追加するタグを継承します。DynamoDB テーブルごとに最大 50 個のタグを追加できます。テーブル内のすべてのタグにサポートされる最大サイズは 10 KB です。DynamoDB リソースのタグ付けとタグ付けの制限の詳細については、「DynamoDB でのタグ付けのリソース」および「the section called “DynamoDB でのタグ付けの制限”」を参照してください。

タグを使用して AWS リソースへのアクセスを制御する方法の詳細については、「IAM ユーザーガイド」の以下のトピックを参照してください。

- の ABAC とはAWS
- タグを使用した AWS リソースへのアクセスの制御

ABAC を使用すると、チームやアプリケーションに異なるアクセスレベルを適用して、より少ないポリシーで DynamoDB テーブルに対してアクションを実行できます。IAM ポリシーの条件要素でタグを指定して、DynamoDB テーブルやインデックスへのアクセスを制御できます。これらの条件により、IAM プリンシパル、ユーザー、またはロールが DynamoDB テーブルとインデックスに対して持つアクセスのレベルが決まります。IAM プリンシパルが DynamoDB へのアクセスリクエストを行うと、リソースと ID のタグは IAM ポリシーのタグ条件に照らして評価されます。その後、タグ条件が満たされた場合にのみポリシーが有効になります。これにより、次のいずれかを効果的に示す IAM ポリシーを作成できます。

- キーが *X* で値は *Y* であるタグを持つリソースのみをユーザーが管理できるようにします。
- キー *X* でタグ付けされたリソースへのすべてのユーザーのアクセスを拒否します。

例えば、タグのキーと値のペアが `"environment": "staging"` である場合にのみ、ユーザーがテーブルを更新できるようにするポリシーを作成できます。aws:ResourceTag 条件キーを使用して、そのテーブルにアタッチされているタグに基づいてテーブルへのアクセスを許可または拒否できます。

属性ベースの条件は、ポリシーの作成時、または後で AWS マネジメントコンソール、AWS API、AWS Command Line Interface (AWS CLI)、AWS SDK、または AWS CloudFormation を使用して含めることができます。

次の例では、名前 environment と値 production を持つタグキーが含まれている場合に、MusicTable という名前の付いたテーブルで UpdateItem アクションを許可します。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/MusicTable",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/environment": "production"
        }
      }
    }
  ]
}
```

トピック

- ABAC を使用する理由
- DynamoDB で ABAC を実装するための条件キー
- DynamoDB で ABAC を使用する際の考慮事項
- DynamoDB テーブルとインデックスでの属性ベースのアクセス制御の使用
- DynamoDB Streams での属性ベースのアクセス制御の使用

ABAC を使用する理由

- ポリシー管理の簡素化: 異なるポリシーを作成して IAM プリンシパルごとにアクセスレベルを定義する必要がないため、使用するポリシーが少なくなります。
- スケーラブルなアクセス制御: ABAC を使用すると、新しい DynamoDB リソースを作成する際のポリシーの更新が必要なくなるため、アクセス制御のスケーリングが容易になります。タグを使用して、リソースのタグと一致するタグを含む IAM プリンシパルへのアクセスを許可できます。新

しい IAM プリンシパルまたは DynamoDB リソースをオンボードし、適切なタグを適用して、ポリシーを変更することなく、必要なアクセス許可を自動的に付与できます。

- きめ細かなアクセス許可管理: ポリシーの作成時には最小特権を付与することがベストプラクティスです。ABAC を使用すると、IAM プリンシパルのタグを作成し、そのタグを使用して IAM プリンシパルのタグと一致する特定のアクションとリソースへのアクセスを付与できます。
- 社内ディレクトリとの整合性: 社内ディレクトリから既存の従業員属性にタグをマッピングして、アクセス制御ポリシーを組織構造に整合させることができます。

DynamoDB で ABAC を実装するための条件キー

AWS ポリシーで次の条件キーを使用して、DynamoDB テーブルとインデックスへのアクセスレベルを制御できます。

- aws:ResourceTag/tag-key: DynamoDB テーブルまたはインデックスのタグのキーと値のペアがポリシーでのタグのキーと値と一致するかどうかに基づいてアクセスを制御します。この条件キーは、既存のテーブルまたはインデックスで動作するすべての API に関連しています。

  `dynamodb:ResourceTag` 条件は、リソースにタグをアタッチしていないかのように評価されます。
- aws:RequestTag/tag-key: リクエストで渡されたタグのキーと値のペアと、ポリシーで指定したタグのペアを比較できます。この条件キーは、リクエストペイロードの一部としてタグを含む API に関連しています。これらの API には、CreateTable と TagResource が含まれます。
- aws:TagKeys: リクエスト内のタグキーをポリシーで指定したキーと比較します。この条件キーは、リクエストペイロードの一部としてタグを含む API に関連しています。これらの API には、`CreateTable`、`TagResource`、`UntagResource` が含まれます。

DynamoDB で ABAC を使用する際の考慮事項

DynamoDB テーブルまたはインデックスで ABAC を使用する場合は、次の考慮事項が適用されます。

- DynamoDB Streams で ABAC を使用する方法の詳細については、「DynamoDB Streams での属性ベースのアクセス制御の使用」を参照してください。
- DynamoDB のバックアップでは、タグ付けと ABAC はサポートされていません。バックアップで ABAC を使用するには、AWS Backup を使用することをお勧めします。

- タグは復元されたテーブルには保持されません。ポリシーでタグベースの条件を使用する前に、復元されたテーブルにタグを追加する必要があります。

DynamoDB テーブルとインデックスでの属性ベースのアクセス制御の使用

以下のトピックでは、DynamoDB テーブルとインデックスの属性ベースのアクセス制御 (ABAC) を有効化、使用、およびトラブルシューティングする方法について説明します。

トピック

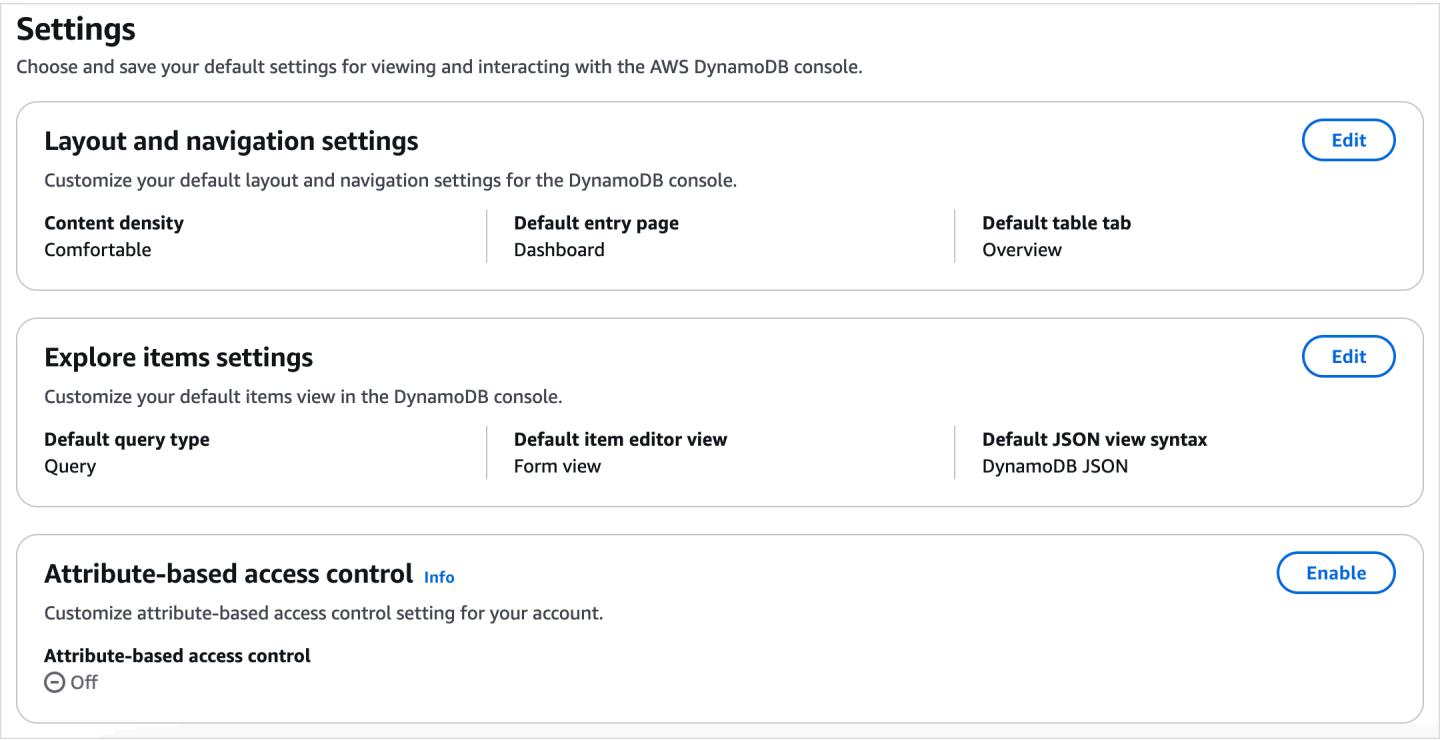
- DynamoDB での ABAC の有効化
- DynamoDB テーブルとインデックスでの ABAC 入門
- DynamoDB テーブルとインデックスで ABAC を使用する例
- DynamoDB テーブルとインデックスに関する一般的な ABAC エラーのトラブルシューティング

DynamoDB での ABAC の有効化

ほとんどの AWS アカウントでは、ABAC はデフォルトで有効になっています。DynamoDB コンソールを使用して、アカウントで ABAC が有効になっているかどうかを確認できます。これを行うには、必ず dynamodb:GetAbacStatus アクセス許可を持つロールで DynamoDB コンソールを開くようにします。次に、DynamoDB コンソールの [設定] ページを開きます。

[属性ベースのアクセス制御] カードが表示されない場合、またはカードのステータスが [オン] と表示される場合は、アカウントで ABAC が有効になっていることを意味します。ただし、次の図に示すように、ステータスが [オフ] の [属性ベースのアクセス制御] カードが表示された場合、アカウントで ABAC は有効になっていません。

属性ベースのアクセス制御 – 有効になっていません



ABAC は、アイデンティティベースのポリシーまたは他のポリシーで指定されたタグベースの条件を引き続き監査する必要がある AWS アカウントに対して有効になっていません。アカウントで ABAC が有効になっていない場合、DynamoDB テーブルまたはインデックスを操作することを目的としたポリシーのタグベースの条件は、リソースまたは API リクエストにタグが存在しないかのように評価されます。アカウントで ABAC が有効になっている場合、アカウントのポリシーのタグベースの条件は、テーブルまたは API リクエストにアタッチされたタグを考慮して評価されます。

アカウントの ABAC を有効にするには、ポリシー監査 セクションの説明に従って、まずポリシーを監査することをお勧めします。次に、IAM ポリシーに ABAC に必要なアクセス許可を含めます。最後に、「コンソールでの ABAC の有効化」で説明されているステップを実行して、現在のリージョンでアカウントの ABAC を有効にします。ABAC を有効にした後、オプトインから 7 暦日以内にオプトアウトできます。

トピック

- ABAC を有効にする前のポリシーの監査
- ABAC を有効にするために必要な IAM アクセス許可
- コンソールでの ABAC の有効化

ABAC を有効にする前のポリシーの監査

アカウントで ABAC を有効にする前に、ポリシーを監査して、アカウント内のポリシーに存在する可能性のあるタグベースの条件が意図したとおりに設定されていることを確認します。ポリシーを監査することで、ABAC が有効になった後に DynamoDB ワークフローで認可が変更されることによる予期しない事態を回避できます。タグを使用した属性ベースの条件の使用例と、ABAC 実装の前後の動作については、「DynamoDB テーブルとインデックスで ABAC を使用する例」を参照してください。

ABAC を有効にするために必要な IAM アクセス許可

現在のリージョンでアカウントの ABAC を有効にするには、`dynamodb:UpdateAbacStatus` アクセス許可が必要です。アカウントで ABAC が有効になっているかどうかを確認するには、`dynamodb:GetAbacStatus` アクセス許可も必要です。このアクセス許可を使用すると、任意のリージョンのアカウントの ABAC ステータスを表示できます。DynamoDB コンソールへのアクセスに必要なアクセス許可に加えて、これらの許可が必要です。

次の IAM ポリシーは、現在のリージョンのアカウントで ABAC を有効にし、そのステータスを表示するアクセス許可を付与します。

```
{
"Version": "2012-10-17",        &TCX5-2025-waiver;
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateAbacStatus",
                "dynamodb:GetAbacStatus"
             ],
            "Resource": "*"
        }
    ]
}
```

コンソールでの ABAC の有効化

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 上部のナビゲーションペインで、ABAC を有効にするリージョンを選択します。
3. 左のナビゲーションペインで、[設定] を選択します。

4. [Settings] (設定) ページで、以下の操作を行います。

   a. [属性ベースのアクセス制御] カードで、[有効化] を選択します。

   b. [属性ベースのアクセス制御設定の確認] ボックスで、[有効化] を選択して選択を確定します。

      これにより、現在のリージョンの ABAC が有効になり、[属性ベースのアクセス制御] カードに [オン] のステータスが表示されます。

      コンソールで ABAC を有効にした後にオプトアウトする場合は、オプトインから 7 暦日以内にオプトアウトできます。オプトアウトするには、[設定] ページの [属性ベースのアクセス制御] カードで [無効化] を選択します。

      > **Note**
      >
      > ABAC のステータスの更新は非同期オペレーションです。ポリシーのタグがすぐに評価されない場合は、しばらく待つ必要がある場合があります。変更の適用は最終的に整合します。

DynamoDB テーブルとインデックスでの ABAC 入門

次の手順は、ABAC を使用してアクセス許可を設定する方法を示しています。このシナリオ例では、DynamoDB テーブルにタグを追加し、タグベースの条件を含むポリシーを使用して IAM ロールを作成します。次に、タグ条件を一致させて、DynamoDB テーブルに対して許可されているアクセス許可をテストします。

トピック

- ステップ 1: DynamoDB テーブルにタグを追加
- ステップ 2: タグベースの条件を含むポリシーを使用して IAM ロールを作成
- ステップ 3: 許可されたアクセス許可をテスト

ステップ 1: DynamoDB テーブルにタグを追加

AWS マネジメントコンソール、AWS API、AWS Command Line Interface (AWS CLI)、AWS SDK、または AWS CloudFormation を使用して、新規または既存の DynamoDB テーブルにタグを追加できます。例えば、次の tag-resource CLI コマンドは、`MusicTable` という名前のテーブルにタグを追加します。

```
aws dynamodb tag-resource –resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/
MusicTable –tags Key=environment,Value=staging
```

ステップ 2: タグベースの条件を含むポリシーを使用して IAM ロールを作成

aws:ResourceTag/tag-key 条件キーを使用して IAM ポリシーを作成し、IAM ポリシーで指定されたタグのキーと値のペアを、テーブルにアタッチされたキーと値のペアと比較します。次のポリシー例では、これらのテーブルにタグのキーと値のペア "environment": "staging" が含まれている場合、ユーザーがテーブルに項目を配置または更新することを許可します。テーブルに指定されたタグのキーと値のペアがない場合、これらのアクションは拒否されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/environment": "staging"
                }
            }
        }
    ]
}
```

ステップ 3: 許可されたアクセス許可をテスト

1. IAM ポリシーを AWS アカウントのテストユーザーまたはロールにアタッチします。使用する IAM プリンシパルが、別のポリシーで DynamoDB テーブルに既にアクセスしていないことを確認します。
2. DynamoDB テーブルに、値が "staging" の "environment" タグキーが含まれていることを確認します。

3. タグ付きテーブルで dynamodb:PutItem および dynamodb:UpdateItem アクションを実行します。"environment": "staging" タグのキーと値のペアが存在する場合、これらのアクションは成功します。

   "environment": "staging" タグのキーと値のペアがないテーブルでこれらのアクションを実行すると、リクエストは AccessDeniedException で失敗します。

次のセクションで説明する他のサンプルユースケースを確認して、ABAC を実装し、さらにテストを実行することもできます。

DynamoDB テーブルとインデックスで ABAC を使用する例

次の例は、タグを使用して属性ベースの条件を実装するいくつかのユースケースを示しています。

トピック

- 例 1: aws:ResourceTag を使用してアクションを許可
- 例 2: aws:RequestTag を使用してアクションを許可
- 例 3: aws:TagKeys を使用してアクションを拒否

例 1: aws:ResourceTag を使用してアクションを許可

aws:ResourceTag/tag-key 条件キーを使用して、IAM ポリシーで指定されたタグのキーと値のペアを、DynamoDB テーブルにアタッチされているキーと値のペアと比較できます。例えば、タグ条件が IAM ポリシーとテーブルで一致する場合、PutItem などの特定のアクションを許可できます。これを作成するには、次のステップを実行します。

Using the AWS CLI

1. テーブルを作成します。次の例では、create-table AWS CLI コマンドを使用して、myMusicTable という名前のテーブルを作成します。

```
aws dynamodb create-table \
  --table-name myMusicTable \
  --attribute-definitions AttributeName=id,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --region us-east-1
```

2. このテーブルにタグを追加します。次の tag-resource AWS CLI コマンドの例では、タグのキーと値のペア Title: ProductManager を myMusicTable に追加します。

```
aws dynamodb tag-resource --region us-east-1 --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/myMusicTable --tags Key=Title,Value=ProductManager
```

3. 次の例に示すように、インラインポリシーを作成し、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーがアタッチされたロールに追加します。

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "dynamodb:PutItem",
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Title": "ProductManager"
        }
      }
    }
  ]
}
```

   このポリシーは、テーブルにアタッチされたタグキーと値がポリシーで指定されたタグと一致する場合に、テーブルに対する PutItem アクションを許可します。

4. ステップ 3 で説明したポリシーを使用してロールを引き受けます。

5. put-item AWS CLI コマンドを使用して、項目を myMusicTable に配置します。

```
aws dynamodb put-item \
    --table-name myMusicTable --region us-east-1 \
    --item '{
        "id": {"S": "2023"},
        "title": {"S": "Happy Day"},
        "info": {"M": {
            "rating": {"N": "9"},
```

```
            "Artists": {"L": [{"S": "Acme Band"}, {"S": "No One You Know"}]},
            "release_date": {"S": "2023-07-21"}
        }}
    }'
```

6. テーブルをスキャンして、項目がテーブルに追加されたかどうかを確認します。

```
aws dynamodb scan --table-name myMusicTable  --region us-east-1
```

Using the AWS SDK for Java 2.x

1. テーブルを作成します。次の例では、CreateTable API を使用して myMusicTable という名前のテーブルを作成します。

```
DynamoDbClient dynamoDB = DynamoDbClient.builder().region(region).build();
CreateTableRequest createTableRequest = CreateTableRequest.builder()
    .attributeDefinitions(
        Arrays.asList(
            AttributeDefinition.builder()
            .attributeName("id")
            .attributeType(ScalarAttributeType.S)
            .build()
        )
    )
    .keySchema(
        Arrays.asList(
            KeySchemaElement.builder()
            .attributeName("id")
            .keyType(KeyType.HASH)
            .build()
        )
    )
    .provisionedThroughput(ProvisionedThroughput.builder()
        .readCapacityUnits(5L)
        .writeCapacityUnits(5L)
        .build()
    )
    .tableName("myMusicTable")
    .build();

CreateTableResponse createTableResponse =
 dynamoDB.createTable(createTableRequest);
```

```
String tableArn = createTableResponse.tableDescription().tableArn();
String tableName = createTableResponse.tableDescription().tableName();
```

2. このテーブルにタグを追加します。次の例の TagResource API は、タグのキーと値のペア `Title: ProductManager` を `myMusicTable` に追加します。

```
TagResourceRequest tagResourceRequest = TagResourceRequest.builder()
    .resourceArn(tableArn)
    .tags(
        Arrays.asList(
            Tag.builder()
            .key("Title")
            .value("ProductManager")
            .build()
        )
    )
    .build();
dynamoDB.tagResource(tagResourceRequest);
```

3. 次の例に示すように、インラインポリシーを作成し、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーがアタッチされたロールに追加します。

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "dynamodb:PutItem",
      "Resource": "arn:aws:dynamodb:*:*:table/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Title": "ProductManager"
        }
      }
    }
  ]
}
```

このポリシーは、テーブルにアタッチされたタグキーと値がポリシーで指定されたタグと一致する場合に、テーブルに対する PutItem アクションを許可します。

4. ステップ 3 で説明したポリシーを使用してロールを引き受けます。
5. PutItem API を使用して、項目を myMusicTable に配置します。

```
HashMap<String, AttributeValue> info = new HashMap<>();
info.put("rating", AttributeValue.builder().s("9").build());
info.put("artists", AttributeValue.builder().ss(List.of("Acme Band","No One You
 Know").build());
info.put("release_date", AttributeValue.builder().s("2023-07-21").build());

HashMap<String, AttributeValue> itemValues = new HashMap<>();
itemValues.put("id", AttributeValue.builder().s("2023").build());
itemValues.put("title", AttributeValue.builder().s("Happy Day").build());
itemValues.put("info", AttributeValue.builder().m(info).build());


PutItemRequest putItemRequest = PutItemRequest.builder()
                .tableName(tableName)
                .item(itemValues)
                .build();
dynamoDB.putItem(putItemRequest);
```

6. テーブルをスキャンして、項目がテーブルに追加されたかどうかを確認します。

```
ScanRequest scanRequest = ScanRequest.builder()
                .tableName(tableName)
                .build();

ScanResponse scanResponse = dynamoDB.scan(scanRequest);
```

Using the AWS SDK for Python (Boto3)

1. テーブルを作成します。次の例では、CreateTable API を使用して myMusicTable という名前のテーブルを作成します。

```
create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
```

```
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 5,
        'WriteCapacityUnits': 5
    },
)

table_arn = create_table_response['TableDescription']['TableArn']
```

2. このテーブルにタグを追加します。次の例の TagResource API は、タグのキーと値のペア `Title: ProductManager` を `myMusicTable` に追加します。

```
tag_resouce_response = ddb_client.tag_resource(
    ResourceArn=table_arn,
    Tags=[
        {
            'Key': 'Title',
            'Value': 'ProductManager'
        },
    ]
)
```

3. 次の例に示すように、インラインポリシーを作成し、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーがアタッチされたロールに追加します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
```

```
        "Action": "dynamodb:PutItem",
        "Resource": "arn:aws:dynamodb:*:*:table/*",
        "Condition": {
            "StringEquals": {
            "aws:ResourceTag/Title": "ProductManager"
            }
        }
        }
    ]
    }
```

このポリシーは、テーブルにアタッチされたタグキーと値がポリシーで指定されたタグと一致する場合に、テーブルに対する PutItem アクションを許可します。

4. ステップ 3 で説明したポリシーを使用してロールを引き受けます。
5. PutItem API を使用して、項目を myMusicTable に配置します。

```
put_item_response = client.put_item(
    TableName = 'myMusicTable'
    Item = {
        'id': '2023',
        'title': 'Happy Day',
        'info': {
            'rating': '9',
            'artists': ['Acme Band','No One You Know'],
            'release_date': '2023-07-21'
        }
    }
)
```

6. テーブルをスキャンして、項目がテーブルに追加されたかどうかを確認します。

```
scan_response = client.scan(
    TableName='myMusicTable'
)
```

ABAC がない場合

AWS アカウント で ABAC が有効になっていない場合、IAM ポリシーと DynamoDB テーブルのタグ条件が一致しません。結果として、AmazonDynamoDBReadOnlyAccess ポリシーの影響により、PutItem アクションは AccessDeniedException を返します。

```
An error occurred (AccessDeniedException) when calling the PutItem operation:
 User: arn:aws:sts::123456789012:assumed-role/DynamoDBReadOnlyAccess/Alice is
 not authorized to perform: dynamodb:PutItem on resource: arn:aws:dynamodb:us-
east-1:123456789012:table/myMusicTable because no identity-based policy allows the
 dynamodb:PutItem action.
```

ABAC がある場合

AWS アカウントで ABAC が有効になっている場合、put-item アクションは正常に完了し、テーブルに新しい項目を追加します。これは、IAM ポリシーとテーブルのタグ条件が一致すると、テーブルのインラインポリシーで PutItem アクションが許可されるためです。

例 2: aws:RequestTag を使用してアクションを許可

aws:RequestTag/tag-key 条件キーを使用すると、リクエストで渡されたタグのキーと値のペアを、IAM ポリシーで指定されたタグペアと比較できます。例えば、タグ条件が一致しない場合は、aws:RequestTag を使用して CreateTable などの特定のアクションを許可できます。これを作成するには、次のステップを実行します。

Using the AWS CLI

1. 次の例に示すように、インラインポリシーを作成し、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーがアタッチされたロールに追加します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
```

```
                "StringEquals": {
                    "aws:RequestTag/Owner": "John"
                }
            }
        }
    ]
}
```

2. "Owner": "John" のタグのキーと値のペアを含むテーブルを作成します。

```
aws dynamodb create-table \
--attribute-definitions AttributeName=ID,AttributeType=S \
--key-schema AttributeName=ID,KeyType=HASH  \
--provisioned-throughput ReadCapacityUnits=1000,WriteCapacityUnits=500 \
--region us-east-1 \
--tags Key=Owner,Value=John \
--table-name myMusicTable
```

Using the AWS SDK for Python (Boto3)

1. 次の例に示すように、インラインポリシーを作成し、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーがアタッチされたロールに追加します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "John"
                }
            }
        }
```

```
    ]
}
```

2. "Owner": "John" のタグのキーと値のペアを含むテーブルを作成します。

```
ddb_client = boto3.client('dynamodb')

create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 1000,
        'WriteCapacityUnits': 500
    },
    Tags=[
        {
            'Key': 'Owner',
            'Value': 'John'
        },
    ],
)
```

ABAC がない場合

AWS アカウントで ABAC が有効になっていない場合、インラインポリシーと DynamoDB テーブルのタグ条件が一致しません。結果として、CreateTable リクエストは失敗し、テーブルは作成されません。

```
An error occurred (AccessDeniedException) when calling the CreateTable operation:
 User: arn:aws:sts::123456789012:assumed-role/Admin/John is not authorized to perform:
```

```
 dynamodb:CreateTable on resource: arn:aws:dynamodb:us-east-1:123456789012:table/
myMusicTable because no identity-based policy allows the dynamodb:CreateTable action.
```

ABAC がある場合

AWS アカウントで ABAC が有効になっている場合、テーブル作成リクエストは正常に完了します。"Owner": "John" のタグのキーと値のペアが CreateTable リクエスト内に存在するため、インラインポリシーはユーザー John が CreateTable アクションを実行することを許可します。

例 3: aws:TagKeys を使用してアクションを拒否

aws:TagKeys 条件キーを使用すると、リクエスト内のタグキーを IAM ポリシーで指定されたキーと比較できます。例えば、リクエストに特定のタグキーが存在しない場合、aws:TagKeys を使用して CreateTable などの特定のアクションを拒否できます。これを作成するには、次のステップを実行します。

Using the AWS CLI

1. 次の例に示すように、AmazonDynamoDBFullAccess AWS 管理ポリシーがアタッチされているロールにカスタマー管理ポリシーを追加します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "Null": {
                    "aws:TagKeys": "false"
                },
                "ForAllValues:StringNotEquals": {
                    "aws:TagKeys": "CostCenter"
                }
            }
        }
```

```
    ]
}
```

2. ポリシーがアタッチされたロールを引き受け、タグキー Title を使用してテーブルを作成します。

```
aws dynamodb create-table \
--attribute-definitions AttributeName=ID,AttributeType=S \
--key-schema AttributeName=ID,KeyType=HASH  \
--provisioned-throughput ReadCapacityUnits=1000,WriteCapacityUnits=500 \
--region us-east-1 \
--tags Key=Title,Value=ProductManager \
--table-name myMusicTable
```

Using the AWS SDK for Python (Boto3)

1. 次の例に示すように、AmazonDynamoDBFullAccess AWS 管理ポリシーがアタッチされているロールにカスタマー管理ポリシーを追加します。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:CreateTable",
                "dynamodb:TagResource"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "Null": {
                    "aws:TagKeys": "false"
                },
                "ForAllValues:StringNotEquals": {
                    "aws:TagKeys": "CostCenter"
                }
            }
        }
    ]
```

```
}
```

2. ポリシーがアタッチされたロールを引き受け、タグキー Title を使用してテーブルを作成します。

```
ddb_client = boto3.client('dynamodb')

create_table_response = ddb_client.create_table(
    AttributeDefinitions=[
        {
            'AttributeName': 'id',
            'AttributeType': 'S'
        },
    ],
    TableName='myMusicTable',
    KeySchema=[
        {
            'AttributeName': 'id',
            'KeyType': 'HASH'
        },
    ],
        ProvisionedThroughput={
        'ReadCapacityUnits': 1000,
        'WriteCapacityUnits': 500
    },
    Tags=[
        {
            'Key': 'Title',
            'Value': 'ProductManager'
        },
    ],
)
```

ABAC がない場合

AWS アカウントで ABAC が有効になっていない場合、DynamoDB は `create-table` コマンドのタグキーを IAM に送信しません。`Null` 条件は、リクエストにタグがない場合に条件が `false` と評価されることを保証します。Deny ポリシーが一致しないため、`create-table` コマンドは正常に完了します。

ABAC がある場合

AWS アカウントで ABAC が有効になっている場合、create-table コマンドで渡されたタグキーは IAM に渡されます。タグキー Title は、Deny ポリシーに存在する条件ベースのタグキー CostCenter に照らして評価されます。StringNotEquals 演算子のため、タグキー Title は Deny ポリシーに存在するタグキーと一致しません。そのため、CreateTable アクションは失敗し、テーブルは作成されません。create-table コマンドを実行すると、AccessDeniedException が返されます。

```
An error occurred (AccessDeniedException) when calling the CreateTable operation:
 User: arn:aws:sts::123456789012:assumed-role/DynamoFullAccessRole/ProductManager is
 not authorized to perform: dynamodb:CreateTable on resource: arn:aws:dynamodb:us-
east-1:123456789012:table/myMusicTable with an explicit deny in an identity-based
 policy.
```

DynamoDB テーブルとインデックスに関する一般的な ABAC エラーのトラブルシューティング

このトピックでは、DynamoDB テーブルまたはインデックスに ABAC を実装する際に発生する可能性がある一般的なエラーや問題のトラブルシューティングに関するアドバイスを提供します。

ポリシー内のサービス固有の条件キーによりエラーが発生する

サービス固有の条件キーは、有効な条件キーとは見なされません。ポリシーでこのようなキーを使用した場合、エラーが発生します。この問題を解決するには、サービス固有の条件キーを DynamoDB に ABAC を実装する適切な条件キーに置き換える必要があります。

例えば、PutItem リクエストを実行するインラインポリシーで dynamodb:ResourceTag 条件キーを使用したとします。リクエストが AccessDeniedException で失敗するとします。次の例は、dynamodb:ResourceTag 条件キーを使用した誤りのあるインラインポリシーを示しています。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "dynamodb:ResourceTag/Owner": "John"
                }
            }
        }
    ]
}
```

この問題を解決するには、次の例に示すように、dynamodb:ResourceTag 条件キーを aws:ResourceTag に置き換えます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/Owner": "John"
                }
            }
        }
    ]
}
```

ABAC をオプトアウトできない

サポート を通じてアカウントで ABAC が有効になっている場合、DynamoDB コンソールから ABAC をオプトアウトすることはできません。オプトアウトするには、サポート に問い合わせてください。

次の条件を満たす場合のみ、ABAC をオプトアウトできます。

- DynamoDB コンソールを通じてセルフサービスでオプトインする方法を使用しました。
- オプトインから 7 暦日以内にオプトアウトします。

DynamoDB Streams での属性ベースのアクセス制御の使用

DynamoDB Streams は、「DynamoDB 用のリソースベースのポリシーを使用する」で説明されているのと同じ認可戦略である属性ベースのアクセス制御 (ABAC) をサポートしています。DynamoDB Streams ABAC では、ストリームにタグをアタッチし、IAM ポリシーでタグベースの条件を使用してアクセスを制御します。

ストリームリソースは親テーブルからタグを継承せず、ストリームのタグを個別に管理できます。DynamoDB ストリームごとに最大 50 個のタグを追加できます。ストリーム上のすべてのタグがサポートする最大サイズは 10 KB です。DynamoDB リソースへのタグ付けと、タグ付けの制限の詳細については、「DynamoDB でのリソースへのタグとラベルの追加」および「DynamoDB でのタグ付けの制限」を参照してください。

aws:ResourceTag 条件キーを使用して、そのストリームにアタッチされているタグに基づいてストリームへのアクセスを許可または拒否できます。例えば、次のポリシーでは、タグキー `environment` に値 `production` を含むすべてのストリームで `GetRecords` アクションを許可します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetRecords"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/environment": "production"
        }
      }
    }
  ]
}
```

トピック

- DynamoDB Streams で ABAC を実装するための条件キー
- DynamoDB Streams で ABAC を使用する際の考慮事項
- DynamoDB Streams での ABAC の有効化
- DynamoDB Streams での ABAC の使用
- DynamoDB Streams で ABAC を使用する例
- DynamoDB Streams の一般的な ABAC エラーのトラブルシューティング

DynamoDB Streams で ABAC を実装するための条件キー

DynamoDB Streams ABAC は、DynamoDB テーブルと同じ条件キー `aws:ResourceTag/tag-key`、`aws:RequestTag/tag-key`、および `aws:TagKeys` を使用します。各条件キーの説明については、DynamoDB で ABAC を実装するための条件キー を参照してください。以下の動作は DynamoDB Streams に固有のものです。

- 条件キーは、TagResource、`UntagResource`、および `GetRecords` などの読み取り API を含む、ストリームで動作する DynamoDB Streams API に適用されます。
- アカウントで DynamoDB Streams ABAC が有効になっていない場合、`aws:ResourceTag` 条件はストリームリソースにタグがアタッチされていないものとして評価されます。

DynamoDB Streams で ABAC を使用する際の考慮事項

DynamoDB Streams で ABAC を使用する場合は、次の考慮事項が適用されます。

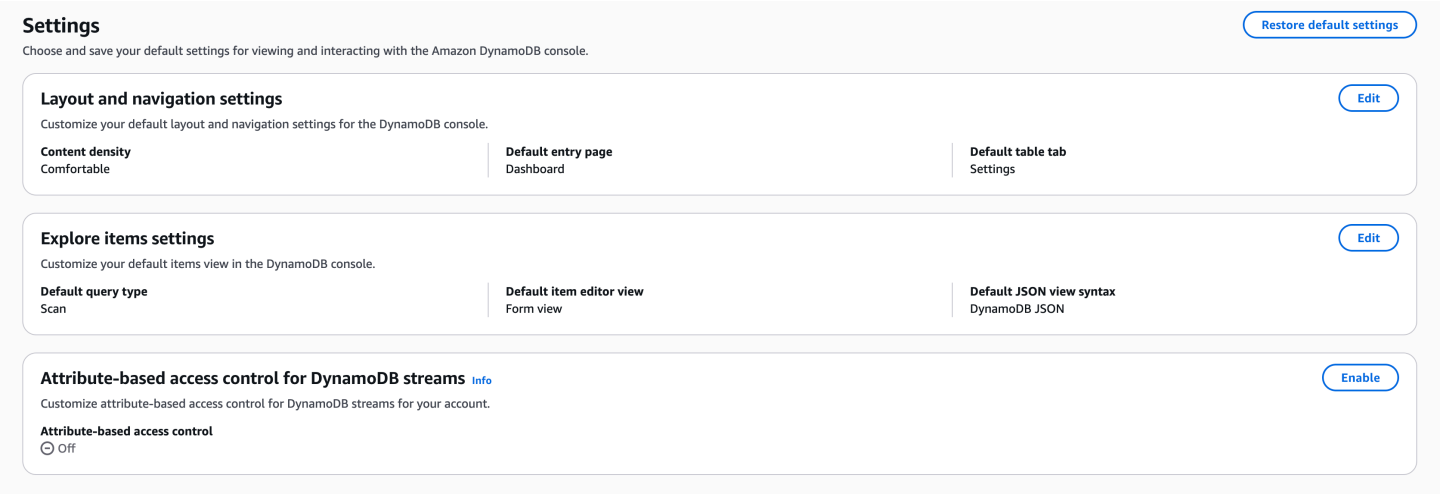
- DynamoDB Streams の ABAC は、DynamoDB テーブルの ABAC とは別のものです。テーブルの ABAC を有効にしても、アカウント内のストリームに対して自動的に有効になるわけではありません。それぞれを個別に有効にする必要があります。
- 親テーブルが削除された後でも、ストリーム ARN に対して CLI または SDK の `tag-resource`、`untag-resource`、および `list-tags-of-resource` コマンドを使用して、ストリームにタグを付けたりタグを解除したりできます。ストリームは、テーブルが削除されてから 24 時間存続し続け、その後削除されます。この機能は、テーブルの削除後にコンソールまたは CloudFormation からは使用できません。
- CloudFormation を使用して DynamoDB リソースを管理する場合は、サービスロールにストリームリソースに対する `dynamodb:TagResource`、`dynamodb:UntagResource`、および `dynamodb:ListTagsOfResource` アクセス許可があることを確認してください。

DynamoDB Streams での ABAC の有効化

DynamoDB Streams ABAC は、ほとんどの AWS アカウントでデフォルトで有効になっています。既存のお客様の場合、Streams ABAC はデフォルトで無効になっており、ポリシー監査後に有効にする必要があります。DynamoDB コンソールを使用して、アカウントで Streams ABAC が有効になっているかどうかを確認できます。`dynamodb:GetAbacStatus` アクセス許可を持つロールを使用して DynamoDB コンソールを開きます。次に、DynamoDB コンソールの [設定] ページを開きます。

[Streams の属性ベースのアクセス制御] カードは、現在の AWS リージョンに DynamoDB テーブルがある場合にのみ、[設定] ページに表示されます。テーブルがある場合、ステータスが [オン] (またはカードなし) の場合は Streams ABAC が有効になっており、[オフ] (次の図を参照) の場合は有効になっていません。リージョンにテーブルがない場合、カードが表示されないことはステータスを示すものではありません。

Streams の属性ベースのアクセス制御 – 有効になっていません



アカウントの Streams ABAC を有効にするには、「Streams ABAC を有効にする前のポリシーの監査」セクションの説明に従って、まずポリシーを監査することをお勧めします。次に、IAM ポリシーに Streams ABAC に必要なアクセス許可を含めます。最後に、「コンソールでの Streams ABAC の有効化」で説明されているステップを実行して、現在の AWS リージョンでアカウントの Streams ABAC を有効にします。Streams ABAC を有効にした後、オプトインから 7 暦日以内にオプトアウトできます。

トピック

- Streams ABAC を有効にする前のポリシーの監査
- Streams ABAC を有効にするために必要な IAM アクセス許可
- コンソールでの Streams ABAC の有効化

Streams ABAC を有効にする前のポリシーの監査

Streams ABAC を有効にする前に、ポリシーを監査してください。アカウントポリシーのタグベースの条件が、ストリームリソースに対して意図したとおりに設定されていることを確認します。ポリシーを監査することで、Streams ABAC を有効にした後に DynamoDB Streams ワークフローで予期しない認可の変更が生じるのを防ぐことができます。

監査時には、以下の条件に該当するポリシーを確認します。

- ストリーム ARN で `aws:ResourceTag` 条件を使用する (形式: `arn:aws:dynamodb:*:*:table/*/stream/*`)
- ストリームリソースで `dynamodb:TagResource` または `dynamodb:UntagResource` アクションに対して `aws:RequestTag` または `aws:TagKeys` 条件を使用する
- 意図せずにストリームに適用される可能性があるタグベースの条件でワイルドカードリソース ARN (`arn:aws:dynamodb:*:*:*`) を使用する

タグを使用した属性ベースの条件の使用例と、ABAC 実装の前後の動作については、「DynamoDB Streams で ABAC を使用する例」を参照してください。

Streams ABAC を有効にするために必要な IAM アクセス許可

現在のリージョンでアカウントの Streams ABAC を有効にするには、`dynamodb:UpdateAbacStatus` アクセス許可が必要です。アカウントで Streams ABAC が有効になっているかどうかを確認するには、`dynamodb:GetAbacStatus` アクセス許可も必要です。このアクセス許可を使用すると、任意のリージョンのアカウントの Streams ABAC ステータスを表示できます。DynamoDB コンソールへのアクセスに必要なアクセス許可に加えて、これらの許可が必要です。

次の IAM ポリシーは、現在のリージョンのアカウントで Streams ABAC を有効にし、そのステータスを表示するアクセス許可を付与します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:UpdateAbacStatus",
        "dynamodb:GetAbacStatus"
      ],
```

```
      "Resource": "*"
    }
  ]
}
```

コンソールでの Streams ABAC の有効化

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. 上部のナビゲーションペインで、Streams ABAC を有効にするリージョンを選択します。
3. 左のナビゲーションペインで、[設定] を選択します。
4. [Settings] (設定) ページで、以下の操作を行います。

   a. [Streams の属性ベースのアクセス制御] カードで、[有効化] を選択します。

   b. [Streams の属性ベースのアクセス制御設定の確認] ボックスで、[有効化] を選択して選択を確定します。

      これにより、現在のリージョンの Streams ABAC が有効になり、[Streams の属性ベースのアクセス制御] カードに [オン] のステータスが表示されます。

      コンソールで Streams ABAC を有効にした後にオプトアウトする場合は、オプトインから 7 暦日以内にオプトアウトできます。オプトアウトするには、[設定] ページの [Streams の属性ベースのアクセス制御] カードで [無効化] を選択します。

      > **Note**
      >
      > Streams ABAC のステータスの更新は非同期オペレーションです。DynamoDB がポリシー内のタグをすぐに評価しない場合は、数分待ってから再試行してください。タグ評価の変更は最終的に整合性が保たれます。

DynamoDB Streams での ABAC の使用

次の手順は、DynamoDB Streams の ABAC を使用してアクセス許可を設定する方法を示しています。このシナリオ例では、DynamoDB ストリームにタグを追加し、タグベースの条件を含むポリシーを使用して IAM ロールを作成します。次に、タグ条件を一致させて、DynamoDB ストリームで許可されているアクセス許可をテストします。

トピック

- ステップ 1: DynamoDB ストリームにタグを追加
- ステップ 2: タグベースの条件を含むポリシーを使用して IAM ロールを作成
- ステップ 3: 許可されたアクセス許可をテスト

ステップ 1: DynamoDB ストリームにタグを追加

AWS CLI または SDK を使用して、DynamoDB Streams にタグを追加できます。次の `tag-resource` CLI コマンドは、ストリームにタグを追加します。

```
aws dynamodb tag-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/MusicTable/
stream/2024-01-01T00:00:00.000 \
  --tags Key=environment,Value=staging
```

> **Note**
>
> ストリーム ARN を検索するには、`describe-table` コマンドを使用してレスポンス内の `LatestStreamArn` フィールドを探します。

ステップ 2: タグベースの条件を含むポリシーを使用して IAM ロールを作成

`aws:ResourceTag/{{tag-key}}` 条件キーを使用して IAM ポリシーを作成し、IAM ポリシーで指定されたタグのキーと値のペアを、ストリームにアタッチされたキーと値のペアと比較します。次のポリシー例では、ストリームにタグのキーと値のペア `"environment": "staging"` が含まれている場合に、ユーザーがレコードを取得し、ストリームを記述することを許可します。ストリームに指定されたタグのキーと値のペアがない場合、これらのアクションは拒否されます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords",
        "dynamodb:GetShardIterator"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
```

```
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/environment": "staging"
        }
      }
    }
  ]
}
```

ステップ 3: 許可されたアクセス許可をテスト

1. IAM ポリシーを AWS アカウントのテストユーザーまたはロールにアタッチします。使用する IAM プリンシパルが、別のポリシーで DynamoDB ストリームに既にアクセスしていないことを確認します。
2. DynamoDB ストリームに、値が "staging" の "environment" タグキーが含まれていることを確認します。
3. タグ付きストリームで dynamodb:DescribeStream および dynamodb:GetRecords アクションを実行します。"environment": "staging" タグのキーと値のペアが存在する場合、これらのアクションは成功します。

   "environment": "staging" タグのキーと値のペアがないストリームでこれらのアクションを実行すると、リクエストは AccessDeniedException で失敗します。

DynamoDB Streams で ABAC を使用する例

以下の例は、DynamoDB Streams でタグを使用して属性ベースの条件を実装するユースケースを示しています。

トピック

- 例 1: aws:ResourceTag を使用してストリームの読み取りアクセスを許可
- 例 2: aws:RequestTag を使用してストリームへのタグ付けを許可
- 例 3: 特定のタグが存在しない場合にストリームへのアクセスを拒否
- 例 4: CloudFormation を使用したストリームのタグ付け

例 1: aws:ResourceTag を使用してストリームの読み取りアクセスを許可

aws:ResourceTag/tag-key 条件キーを使用して、IAM ポリシーで指定されたタグのキーと値のペアを、DynamoDB ストリームにアタッチされているキーと値のペアと比較できます。例えば、タグ条件が一致する場合にストリーム GetRecords オペレーションを許可できます。

AWS CLI の使用

1. ストリームを有効にしたテーブルを作成し、タグを追加します。

```
aws dynamodb create-table \
  --table-name myMusicTable \
  --attribute-definitions AttributeName=id,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
  --region us-east-1
```

2. ストリームにタグを追加します。まず、テーブルの説明からストリーム ARN を取得します。

```
STREAM_ARN=$(aws dynamodb describe-table \
  --table-name myMusicTable \
  --query "Table.LatestStreamArn" \
  --output text \
  --region us-east-1)

aws dynamodb tag-resource \
  --resource-arn $STREAM_ARN \
  --tags Key=Team,Value=Analytics \
  --region us-east-1
```

3. 次の例に示すように、インラインポリシーを作成し、ロールに追加します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
```

```
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Team": "Analytics"
        }
      }
    }
  ]
}
```

このポリシーは、ストリームのタグがポリシーで指定されたキー「Team」と値「Analytics」と一致する場合に、DescribeStream および GetRecords オペレーションを許可します。

4. ステップ 3 で説明したポリシーを使用してロールを引き受けます。
5. describe-stream AWS CLI コマンドをストリームで使用します。

```
aws dynamodb describe-stream \
  --stream-arn $STREAM_ARN \
  --region us-east-1
```

Streams ABAC を使用した場合と使用しない場合の動作

Streams ABAC を使用しない場合

AWS アカウントで Streams ABAC が有効になっていない場合、IAM ポリシーと DynamoDB ストリームのタグ条件が一致しません。タグ条件に一致しない限り、そのアクションを許可するポリシーは存在しないため、DescribeStream アクションは AccessDeniedException を返します。

Streams ABAC を使用する場合

Streams ABAC が AWS アカウントで有効になっている場合、DescribeStream アクションは正常に完了します。これは、IAM ポリシーとストリームのタグ条件が一致すると、インラインポリシーでアクションが許可されるためです。

例 2: aws:RequestTag を使用してストリームへのタグ付けを許可

aws:RequestTag/tag-key 条件キーを使用すると、リクエストで渡されたタグのキーと値のペアを、IAM ポリシーで指定されたタグのペアと比較できます。例えば、リクエストに特定のタグが含まれている場合にのみ、ストリームへのタグ付けを許可できます。

AWS CLI の使用

1. 次の例に示すように、インラインポリシーを作成し、ロールに追加します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:TagResource"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/CostCenter": "12345"
        }
      }
    }
  ]
}
```

2. 必要なタグキーと値のペアをストリームにタグ付けします。

```
aws dynamodb tag-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/myMusicTable/
stream/2024-01-01T00:00:00.000 \
  --tags Key=CostCenter,Value=12345
```

Streams ABAC を使用した場合と使用しない場合の動作

Streams ABAC を使用しない場合

AWS アカウントで Streams ABAC が有効になっていない場合、インラインポリシーとリクエストのタグ条件が一致しません。`TagResource` アクションは `AccessDeniedException` を返します。

Streams ABAC を使用する場合

AWS アカウントで Streams ABAC が有効になっている場合、タグリクエストは正常に完了します。`"CostCenter": "12345"` のタグのキーと値のペアが `TagResource` リクエスト内に存在するため、インラインポリシーはアクションを許可します。

例 3: 特定のタグが存在しない場合にストリームへのアクセスを拒否

aws:TagKeys 条件キーを使用すると、必要なタグキーがリクエストに含まれていない限り、ストリームへのタグ付けを拒否できます。

AWS CLI の使用

1. 次の例に示すように、DynamoDB へのアクセス権を持つロールにカスタマー管理ポリシーを追加します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "dynamodb:TagResource"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
      "Condition": {
        "Null": {
          "aws:TagKeys": "false"
        },
        "ForAllValues:StringNotEquals": {
          "aws:TagKeys": "CostCenter"
        }
      }
    }
  ]
}
```

2. ロールを引き受け、CostCenter ではないタグキーでストリームへのタグ付けを試みます。

```
aws dynamodb tag-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/myMusicTable/
stream/2024-01-01T00:00:00.000 \
  --tags Key=Department,Value=Engineering
```

Streams ABAC を使用した場合と使用しない場合の動作

Streams ABAC を使用しない場合

AWS アカウントで Streams ABAC が有効になっていない場合、DynamoDB はリクエストのタグキーを IAM に送信しません。`Null` 条件により、リクエストにタグキーがない場合に条件は false と評価されます。拒否ポリシーが一致しないため、`tag-resource` コマンドは正常に完了します。

Streams ABAC を使用する場合

AWS アカウントで Streams ABAC が有効になっている場合、タグキー `Department` は、拒否ポリシーに存在する条件ベースのタグキー `CostCenter` に対して評価されます。`StringNotEquals` 演算子のため、タグキー `Department` は拒否ポリシーに存在するタグキーと一致しません。したがって、`TagResource` アクションは失敗し、`AccessDeniedException` を返します。

例 4: CloudFormation を使用したストリームのタグ付け

DynamoDB Streams のタグは、CloudFormation テンプレートで直接指定できます。ストリームタグは、標準テーブルの場合は `StreamSpecification` プロパティ内で定義され、グローバルテーブルの場合は `ReplicaStreamSpecification` プロパティ内で定義されます。

AWS::DynamoDB::Table

```
Resources:
  MyTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: MyTable
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      BillingMode: PAY_PER_REQUEST
      Tags:
        - Key: TableEnv
          Value: Production
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

```
        Tags:
          - Key: StreamEnv
            Value: Production
          - Key: StreamTeam
            Value: Payments
```

AWS::DynamoDB::GlobalTable

```
Resources:
  MyGlobalTable:
    Type: AWS::DynamoDB::GlobalTable
    Properties:
      TableName: MyGlobalTable
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      BillingMode: PAY_PER_REQUEST
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
      Replicas:
        - Region: us-east-1
          Tags:
            - Key: TableEnv
              Value: Production
          ReplicaStreamSpecification:
            Tags:
              - Key: StreamEnv
                Value: Production
              - Key: StreamTeam
                Value: Payments
        - Region: eu-west-1
          Tags:
            - Key: TableEnv
              Value: Production
          ReplicaStreamSpecification:
            Tags:
              - Key: StreamEnv
                Value: Production
              - Key: StreamTeam
                Value: Payments
```

CloudFormation は、明示的なストリームタグに加えて、スタックレベルのタグをストリームに伝播します。スタックにタグが設定されている場合、それらのタグはストリームを含むスタック内のすべてのリソースに自動的に適用されます。

> **Important**
>
> CloudFormation デプロイに使用される IAM ロールには、ストリームにタグを適用するために、ストリームリソース (`arn:aws:dynamodb:*:*:table/*/stream/*`) に対する `dynamodb:TagResource` および `dynamodb:UntagResource` アクセス許可が必要です。これらのアクセス許可がない場合は、「ストリームへのタグ付け時に CloudFormation デプロイが失敗する」を参照してください。

DynamoDB Streams の一般的な ABAC エラーのトラブルシューティング

DynamoDB Streams の ABAC を有効にした後、`AccessDeniedException` を受け取った場合、またはタグベースの条件が期待どおりに動作しない場合は、このトピックを使用して最も一般的な原因を診断し、解決してください。

ポリシー内のサービス固有の条件キーによりエラーが発生する

サービス固有の条件キーは、有効な条件キーとは見なされません。ポリシーでこのようなキーを使用した場合、これらのキーはエラーになります。DynamoDB Streams に ABAC を実装するには、サービス固有の条件キーを適切な条件キーに置き換える必要があります。

例えば、ストリームで `GetRecords` リクエストを実行するポリシーで `dynamodb:ResourceTag` 条件キーを使用したとします。リクエストは `AccessDeniedException` で失敗します。次の例は、`dynamodb:ResourceTag` 条件キーを使用した誤りのあるポリシーを示しています。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetRecords"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
      "Condition": {
        "StringEquals": {
```

```
          "dynamodb:ResourceTag/Team": "Analytics"
        }
      }
    }
  ]
}
```

この問題を解決するには、次の例に示すように、dynamodb:ResourceTag 条件キーを aws:ResourceTag に置き換えます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetRecords"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/*/stream/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Team": "Analytics"
        }
      }
    }
  ]
}
```

Streams ABAC は有効になっているが、タグ条件が評価されない

最近 Streams ABAC を有効にした場合、変更がまだ反映されていない可能性があります。Streams ABAC のステータスの更新は非同期オペレーションです。数分待ってから、リクエストを再試行してください。

親テーブルが削除された後、ストリームにタグを付けることができない

親テーブルが削除された後、ストリームは 24 時間存在し続け、その後削除されます。この 24 時間の間に、ストリーム ARN を使用して CLI または SDK コマンド (tag-resource、untag-resource、list-tags-of-resource) でストリームタグを管理できます。コンソールまたは CloudFormation を使用して、親テーブルが存在しないストリームのタグを変更することはできません。

コンソールでタグ付けしようとしたときにエラーが表示された場合は、代わりに CLI を使用してください。

```
aws dynamodb tag-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/DeletedTable/
stream/2024-01-01T00:00:00.000 \
  --tags Key=environment,Value=production
```

親テーブルが削除されたストリーム上の既存のタグを確認するには:

```
aws dynamodb list-tags-of-resource \
  --resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/DeletedTable/
stream/2024-01-01T00:00:00.000
```

`ResourceNotFoundException` を受け取った場合、24 時間の保持期間は終了し、ストリームは存在しなくなりました。

Streams ABAC をオプトアウトできない

次の条件を満たす場合のみ、Streams ABAC をオプトアウトできます。

- DynamoDB コンソールを通じてセルフサービスでオプトインする方法を使用しました。
- オプトインから 7 暦日以内にオプトアウトします。

Streams ABAC がサポートを通じてアカウントで有効になっている場合、DynamoDB コンソールからオプトアウトすることはできません。オプトアウトするには、サポートに問い合わせてください。

ストリームへのタグ付け時に CloudFormation デプロイが失敗する

ストリームでテーブルを作成または更新する CloudFormation デプロイは、IAM ロールに必要なアクセス許可がない場合に、失敗することがあります。具体的には、ロールにはストリームリソースに対する `dynamodb:TagResource` および `dynamodb:UntagResource` のアクセス許可が必要です。これは `AWS::DynamoDB::Table` と `AWS::DynamoDB::GlobalTable` の両方のリソースタイプに適用され、明示的なストリームタグと継承されたスタックレベルのタグの両方に影響します。

これを解決するには、CloudFormation サービスロールの IAM ポリシーを更新して、ストリームタグ付けのアクセス許可を含めます。

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:TagResource",
        "dynamodb:UntagResource"
      ],
      "Resource": [
        "arn:aws:dynamodb:*:*:table/*/stream/*"
      ]
    }
  ]
}
```

DynamoDB におけるデータ保護

Amazon DynamoDB は、ミッションクリティカルで重要なデータストレージのために設計された、高い耐久性を備えたストレージインフラストラクチャです。冗長性を確保するために、データは、同一の Amazon DynamoDB リージョン内の複数の施設に分散した複数のデバイスに保存されます。

DynamoDB では、ユーザーデータは保管時に保護されるほか、オンプレミスのクライアントと DynamoDB 間の転送中のデータ、DynamoDB と同じ AWS リージョン内のその他の AWS リソース間の転送中のデータも保護されます。

> ⚠ リソース名に機密データを含めない
>
> 顧客のアカウント番号などの機密の識別情報は、DynamoDB テーブル名や DynamoDB Accelerator (DAX) クラスター名を含む、[名前] フィールドなどの自由形式のフィールドに配置しないことを強くお勧めします。このガイダンスは、コンソール、API、AWS CLI、または AWS SDK を使用して DynamoDB または他の AWS サービスを使用する場合に適用されます。診断ログは、自由形式のフィールドに入力した情報をキャプチャする場合があります。転送中に暗号化を使用する DAX クラスターの場合、パブリック証明書の透明性 (CT) ログにもクラスター名が記録されます。詳細については、「DAX のアクセスコントロール」を参照してください。外部サーバーへの URL を指定するときは、そのサーバーへのリクエストを検証するための認証情報を URL に含めないでください。

トピック

- 保管時の DynamoDB 暗号化

- VPC エンドポイントと IAM ポリシーを使用した DynamoDB 接続の保護

保管時の DynamoDB 暗号化

Amazon DynamoDB に保存されているすべてのユーザーデータは、保管時に完全に暗号化されます。DynamoDB の保管時の暗号化は、AWS Key Management Service (AWS KMS) に保存されている暗号化キーを使用してすべての保管中のデータを暗号化することにより高度なセキュリティを提供します。この機能は、機密データの保護における負担と複雑な作業を減らすのに役立ちます。保管時に暗号化することで、セキュリティを重視したアプリケーションを構築して、暗号化のコンプライアンスと規制の厳格な要件を満たすことができます。

DynamoDB の保管時の暗号化は、データを堅牢なメディアに保存されている場合は常に暗号化されたテーブル内にデータを配置して保護することにより、プライマリキー、ローカルおよびグローバルセカンダリインデックス、ストリーム、グローバルテーブル、バックアップ、DynamoDB Accelerator (DAX) クラスターなどに対する追加のデータ保護レイヤーを提供します。組織のポリシー、業界や政府の規制、またはコンプライアンス要件によって、アプリケーションのデータセキュリティを高めるために保管時の暗号化の使用が求められることがあります。データベースアプリケーションの暗号化の詳細については、「What is the AWS Database Encryption SDK?」を参照してください。https://docs.aws.amazon.com/database-encryption-sdk/latest/devguide/what-is-database-encryption-sdk.html

保管時の暗号化には、テーブルの暗号化に使用される暗号化キーを管理するための AWS KMS が統合されます。キーの種類と詳細については、「AWS Key Management Service 開発者ガイド」の「AWS Key Management Service concepts」を参照してください。

新しいテーブルを作成する場合、以下のいずれかの AWS KMS key の種類を選択してテーブルを暗号化できます。キーの種類は、いつでも切り替えることができます。

- AWS 所有のキー – デフォルトの暗号化タイプ。キーは DynamoDB により所有されます (追加料金なし)。
- AWS マネージドキー – キーはアカウントに保存され、AWS KMS によって管理されます (AWS KMS の料金が適用されます)。
- カスタマーマネージド型キー – キーはお客様のアカウントに保存され、ユーザーが作成、所有、管理します。ユーザーには KMS キーに対するフルコントロールの権限があります (AWS KMS の料金が適用されます)。

キーの種類の詳細については、「 Customer keys and AWS keys」を参照してください。

> **Note**
>
> - 保存時の暗号化を有効にして新しい DAX クラスターを作成する場合、AWS マネージドキー を使用して、クラスター内の保管中のデータを暗号化します。
> - テーブルにソートキーが存在する場合、範囲の境界線を示すソートキーの一部が、プレーンテキスト形式でテーブルメタデータに保存されます。

暗号化されたテーブルにアクセスすると、DynamoDB はテーブルデータを透過的に復号化します。暗号化されたテーブルの使用あるいは管理のためにコードやアプリケーションを変更する必要はありません。DynamoDB は、期待される 1 桁ミリ秒のレイテンシーを継続的に提供し、すべての DynamoDB クエリは暗号化されたデータでシームレスに機能します。

新しいテーブルを作成する際に暗号化キーを指定したり、AWS マネジメントコンソール、AWS Command Line Interface (AWS CLI)、Amazon DynamoDB API を使用して既存のテーブルで暗号化キーを切り替えたりすることができます。この方法については、「DynamoDB での暗号化テーブルの管理」を参照してください。

AWS 所有のキー を使用した保管時の暗号化に追加の料金はかかりません。ただし、AWS KMS 料金が AWS マネージドキー およびカスタマーマネージドキーに適用されます。料金の詳細については、「AWS KMS の料金」を参照してください。

DynamoDB の保管時の暗号化は、AWS 中国 (北京)、AWS 中国 (寧夏)、AWS GovCloud (米国) のリージョンを含むすべての AWS リージョンで利用できます。詳細については、「保管時の DynamoDB 暗号化: 仕組み」および「DynamoDB の保管時の暗号化の使用に関する注意事項」を参照してください。

保管時の DynamoDB 暗号化: 仕組み

Amazon DynamoDB の保管時の暗号化では、256 ビットの Advanced Encryption Standard (AES-256) を使用してデータの暗号化が行われるので、基盤となるストレージへの不正アクセスからデータを保護できます。

保管データ暗号化には、テーブルの暗号化に使用される暗号化キーを管理するための AWS Key Management Service (AWS KMS) が統合されます。

> **Note**
>
> 2022 年 5 月、AWS KMS は AWS マネージドキー のローテーションスケジュールを 3 年 (約 1,095 日間隔) ごとから毎年 (約 365 日間隔) に変更しました。
> 新しい AWS マネージドキーは、作成日から 1 年後に自動的にローテーションされ、それ以降はほぼ 1 年ごとにローテーションされます。
> 既存の AWS マネージドキー は、直近のローテーションから 1 年後にローテーションされ、その後毎年ローテーションされます。

AWS 所有のキー

AWS 所有のキー はお客様の AWS アカウントに保存されません。これらは、複数の AWS アカウントで使用するために AWS が所有および管理している KMS キーのコレクションの一部です。AWS のサービスでは、データの保護に AWS 所有のキー を使用できます。DynamoDB で使用する AWS 所有のキーは毎年 (約 365 日ごとに) ローテーションされます。

AWS 所有のキー は表示、管理、使用することはできず、その使用を監視することもできません。ただし、データを暗号化するキーを保護するための作業やプログラムを操作したり変更したりする必要はありません。

AWS 所有のキー のご利用に関しては、月額料金や使用料金は請求されません。また、アカウントの AWS KMS クォータにも影響しません。

AWS マネージドキー

AWS マネージドキー は、お客様のアカウントにある KMS キーであり、AWS KMS と統合されている AWS のサービスがお客様に代わって作成、管理、使用します。アカウントで AWS マネージドキー を表示して、キーポリシーを表示し、AWS CloudTrail ログでその使用を監査できます。ただし、これらの KMS キーを管理したり許可を変更したりすることはできません。

保管時の暗号化は、テーブルの暗号化に使用される DynamoDB (aws/dynamodb) 用の AWS マネージドキー を管理するために AWS KMS と自動的に統合されます。暗号化された DynamoDB テーブルを作成したときにAWS マネージドキー が存在しない場合、AWS KMS は自動的に新しいキーを作成します。このキーは、この先作成するテーブルの暗号化に使用されます。AWS KMS は、安全で可用性の高いハードウェアとソフトウェアを組み合わせて、クラウド向けに拡張されたキー管理システムを提供します。

AWS マネージドキー の許可を管理する方法の詳細については、「AWS Key Management Service デベロッパーガイド」の「AWS マネージドキー の使用の承認」を参照してください。

カスタマーマネージドキー

カスタマーマネージドキーは、お客様が作成、所有、管理する AWS アカウントの KMS キーです。この KMS キーでは、キーポリシー、IAM ポリシー、および許可の確立と管理、有効化と無効化、暗号化対象のローテーション、タグの追加、KMS キーを参照するエイリアスの作成、削除スケジュールの設定などを完全に制御することができます。カスタマーマネージドキーの許可を管理する方法の詳細については、「カスタマーマネージドキー」を参照してください。

カスタマーマネージドキーをテーブルレベルの暗号化キーとして指定すると、DynamoDB テーブル、ローカルおよびグローバルセカンダリインデックス、およびストリームは、同じカスタマーマネージドキーで暗号化されます。オンデマンド Backup は、Backup の作成時に指定されたテーブルレベルの暗号化キーを使用して暗号化されます。テーブルレベルの暗号化キーを更新しても、既存のオンデマンド Backup に関連付けられている暗号化キーは変更されません。

カスタマーマネージドキーの状態を無効に設定するか、削除のスケジュールを設定すると、すべてのユーザーと DynamoDB サービスは、データの暗号化と復号化、およびテーブルに対する読み取り/書き込み操作を実行できなくなります。テーブルに対するアクセスを維持し、データ損失を防止するには、DynamoDB が暗号化キーにアクセスできる必要があります。

カスタマーマネージドキーを無効化したり、削除をスケジュールしたりすると、テーブルステータスはアクセス不能になります。テーブルの操作を続行できるようにするには、指定された暗号化キーへの DynamoDB アクセスを 7 日以内に提供する必要があります。暗号化キーにアクセスできないことが検出されると、DynamoDB から警告メール通知が送信されます。

> **Note**
>
> - DynamoDB サービスがカスタマーマネージドキーに 7 日以上アクセスできない場合、テーブルはアーカイブされてアクセスできなくなります。DynamoDB は、テーブルのオンデマンド Backup を作成し、それに対して課金されます。このオンデマンド Backup を使用して、データを新しいテーブルに復元できます。復元を開始するには、最後に使用したカスタマーマネージドキーをテーブルで有効にし、DynamoDB からのアクセスを確立します。
> - グローバルテーブルレプリカの暗号化に使用したカスタマーマネージドキーにアクセスできない場合、DynamoDB は、このレプリカをレプリケーショングループから除外します。レプリカは削除されず、このリージョンに対するレプリケーションは、カスタマーマネージドキーにアクセス不能と検出されてから 20 時間後に停止します。

詳細については、「キーの有効化」と「キーの削除」を参照してください。

AWS マネージドキー の使用に関する注意事項

AWS KMS アカウントに保存されている KMS キーにアクセスできない場合、Amazon DynamoDB はテーブルデータを読み取れません。DynamoDB は、エンベロープ暗号化とキー階層を使用してデータを暗号化します。AWS KMS 暗号化キーは、このキー階層のルートキーを暗号化するために使用されます。詳細については、「AWS Key Management Service デベロッパーガイド」の「エンベロープ暗号化」を参照してください。

DynamoDB は、DynamoDB オペレーションごとに AWS KMS を呼び出すわけではありません。キーは、アクティブなトラフィックを持つ呼び出しごとに 5 分に 1 回更新されます。

SDK が接続を再利用するように設定されていることを確認してください。そうしないと、DynamoDB は、オペレーションごとに新しい AWS KMS キャッシュエントリを再確立しなければならなくなるのでレイテンシーが発生します。さらに、AWS KMSと CloudTrail のコストが上がってしまう可能性もあります。たとえば、Node.js SDK を使用してこれを行うには、`keepAlive` を有効にした状態で新しい HTTPS エージェントを作成することができます。詳細については、「AWS SDK for JavaScript デベロッパーガイド」の「Node.js での keepAlive の設定」を参照してください。

DynamoDB の保管時の暗号化の使用に関する注意事項

Amazon DynamoDB で保管時の暗号化を使用する場合は、以下の点を考慮してください。

すべてのテーブルデータの暗号化

保管時のサーバー側の暗号化は、すべての DynamoDB テーブルデータで有効になり、無効にできません。テーブル内の項目のサブセットのみを暗号化することはできません。

保管時の暗号化は、永続的ストレージメディアの静的 (保管時) のデータのみを暗号化します。転送中のデータあるいは使用中のデータでデータの安全性が考慮される場合には、追加の対策を実行する必要がある場合があります。

- 転送中のデータ: DynamoDB のすべてのデータは転送時に暗号化されます。デフォルトでは、DynamoDB との通信において、Secure Sockets Layer (SSL)/Transport Layer Security (TLS) 暗号化を使用してネットワークトラフィックを保護する HTTPS プロトコルが使用されます。
- 使用中のデータ: DynamoDB 送信する前のデータを保護するには、クライアント側の暗号化を使用します。詳細については、「Amazon DynamoDB Encryption Client デベロッパーガイド」の「クライアント側とサーバー側の暗号化」を参照してください。

暗号化されたテーブルでストリーミングを使用できます。DynamoDB Streams は、テーブルレベルの暗号化キーを使用して常に暗号化されます。詳細については、「DynamoDB Streams の変更データキャプチャ」を参照してください。

DynamoDB バックアップが暗号化され、バックアップから復元されたテーブルでも暗号化が有効になります。バックアップデータの暗号化に、AWS 所有のキー、AWS マネージドキー、またはカスタマーマネージドキーを使用できます。詳細については、「DynamoDB のバックアップと復元」を参照してください。

ローカルセカンダリインデックスおよびグローバルセカンダリインデックスは、ベーステーブルと同じキーを使用して暗号化されます。

暗号化タイプ

> **Note**
>
> カスタマーマネージドキーは、グローバルテーブルバージョン 2017 ではサポートされていません。DynamoDB グローバルテーブルでカスタマーマネージドキーを使用する場合は、テーブルをグローバルテーブルバージョン 2019 にアップグレードしてから有効にする必要があります。

AWS マネジメントコンソール では、AWS マネージドキー または カスタマーマネージドキーを使用してデータを暗号化すると、暗号化タイプは KMS になります。AWS 所有のキー を使用すると、暗号化タイプは DEFAULT になります。Amazon DynamoDB API では、AWS マネージドキー または カスタマーマネージドキーを使用すると暗号化タイプは KMS になります。暗号化タイプがない場合、データは AWS 所有のキー を使用して暗号化されます。AWS 所有のキー、AWS マネージドキー、カスタマーマネージドキーはいつでも切り替えることができます。コンソール、AWS Command Line Interface (AWS CLI)、または Amazon DynamoDB API を使用して、暗号化キーを切り替えることができます。

カスタマーマネージドキーを使用する場合、次の制限に注意してください。

- DynamoDB アクセラレーター (DAX) クラスターではカスタマーマネージドキーは使用できません。詳細については、「保管時の DAX 暗号化」を参照してください。
- カスタマーマネージドキーを使用して、トランザクションを使用するテーブルを暗号化できます。ただし、トランザクションの伝達の堅牢性を確保するために、トランザクションリクエストのコピーはサービスによって一時的に保存され、AWS 所有のキーを使用して暗号化されます。テーブ

ルとセカンダリインデックスのコミット済みデータは、カスタマーマネージドキーを使用して常に保管時に暗号化されます。

- カスタマーマネージドキーを使用して、Contributor Insights を使用するテーブルを暗号化できます。ただし、Amazon CloudWatch に送信されるデータは、AWS 所有のキー を使用して暗号化されます。
- 新しいカスタマーマネージドキーに移行する場合は、プロセスが完了するまで元のキーを有効にしておいてください。AWS では、新しいキーでデータを暗号化する前に、元のキーを使用してデータを復号化する必要があります。テーブルの SSEDescription ステータスが有効になり、新しいカスタマーマネージドキーの KMSmasterKeyArn が表示されると、プロセスは完了します。この時点で、元のキーを無効にするか、削除のスケジュールを設定できます。
- 新しいカスタマーマネージドキーが表示されると、テーブルと新しいオンデマンドバックアップが新しいキーで暗号化されます。
- 既存のオンデマンドバックアップは、バックアップの作成時に使用されたカスタマー管理のキーで暗号化されたままになります。これらのバックアップを復元するには、同じキーが必要です。DescribeBackup API を使用してバックアップの SSEDescription を表示することで、各バックアップが作成された期間のキーを識別できます。
- カスタマーマネージドキーを無効化した場合、または削除をスケジュールした場合、DynamoDB Streams 内のデータは 24 時間保持されます。作成後 24 時間を超えた未取得のアクティビティデータはすべて、トリミングの対象となります。
- カスタマーマネージドキーを無効化した場合、または削除をスケジュールした場合、有効期限 (TTL) は 30 分間続きます。これらの TTL 削除は引き続き DynamoDB Streams に出力され、標準のトリミングまたは保持間隔が適用されます。

詳細については、「キーの有効化」と「キーの削除」を参照してください。

KMS キーとデータキーを使用する

DynamoDB の保存時の暗号化機能は、AWS KMS key およびデータキーの階層を使用してテーブルのデータを保護します。DynamoDB は、同じキー階層を使用して、DynamoDB Streams、グローバルテーブル、およびバックアップが耐久性のあるメディアに書き込まれるときに保護します。

DynamoDB にテーブルを実装する前に、暗号化戦略を計画することをお勧めします。機密データまたは機密データを DynamoDB に保存する場合は、クライアント側の暗号化をプランに含めることを検討してください。これにより、データをできるだけ送信元に近い状態で暗号化し、ライフサイクル全体にわたってデータを確実に保護できます。詳細については、「DynamoDB 暗号化クライアント」ドキュメントを参照してください。

AWS KMS key

保存時の暗号化機能は、AWS KMS key の DynamoDB テーブルを保護します。デフォルトでは、DynamoDB は AWS 所有のキー (DynamoDB サービスアカウントで作成および管理されるマルチテナントキー) を使用します。ただし、DynamoDB テーブルは、カスタマーマネージドキーまたは AWS アカウント の DynamoDB (aws/dynamodb) 用で暗号化できます。テーブルごとに異なる KMS キーを選択できます。テーブル用に選択した KMS キーも、ローカルおよびグローバルのセカンダリインデックス、ストリーム、バックアップの暗号化に使用されます。

テーブルを作成または更新するときは、テーブル用の KMS キーを選択します。テーブル用 KMS キーは、DynamoDB コンソールで、または UpdateTable オペレーションを使用していつでも変更できます。キーの切り替えプロセスはシームレスであり、ダウンタイムやサービスの低下を必要としません。

> **Note**
>
> キーエイリアスを使用して KMS キーを指定すると、DynamoDB は設定時にエイリアスを基盤となる KMS キー ID に解決し、その KMS キーをテーブルに関連付けます。後で別の KMS キーを指すようにエイリアスを変更しても、テーブルは最初に解決された KMS キーを引き続き使用します。エイリアスを更新しても、テーブルを保護する KMS キーは変更されません。別の KMS キーを使用してテーブルを保護するには、DynamoDB コンソールで、または UpdateTable オペレーションを使用して、テーブルの暗号化キーを直接変更します。

> **Important**
>
> DynamoDB は、対称 KMS キーのみをサポートします。非対称 KMS キーを使用して DynamoDB テーブルを暗号化することはできません。

カスタマーマネージドキーを使用して次の機能を取得します。

- KMS キーを作成および管理します。これには、キーポリシーおよび IAM ポリシーの設定、KMS キーへのアクセスを制御するグラントが含まれます。KMS キーの有効化または無効化、自動キーローテーションの有効化または無効化、使用しなくなった KMS キーの削除を実行できます。
- インポートされたキーマテリアルを持つカスタマーマネージドキー、またはユーザーが所有して管理するカスタムキーストアで、カスタマーマネージドキーを使用できます。

- DynamoDB テーブルの暗号化と復号を監査するには、 AWS CloudTrail ログで AWS KMS への DynamoDB API コールを調べます。

次のいずれかの機能が必要なときは、AWS マネージドキー を使用します。

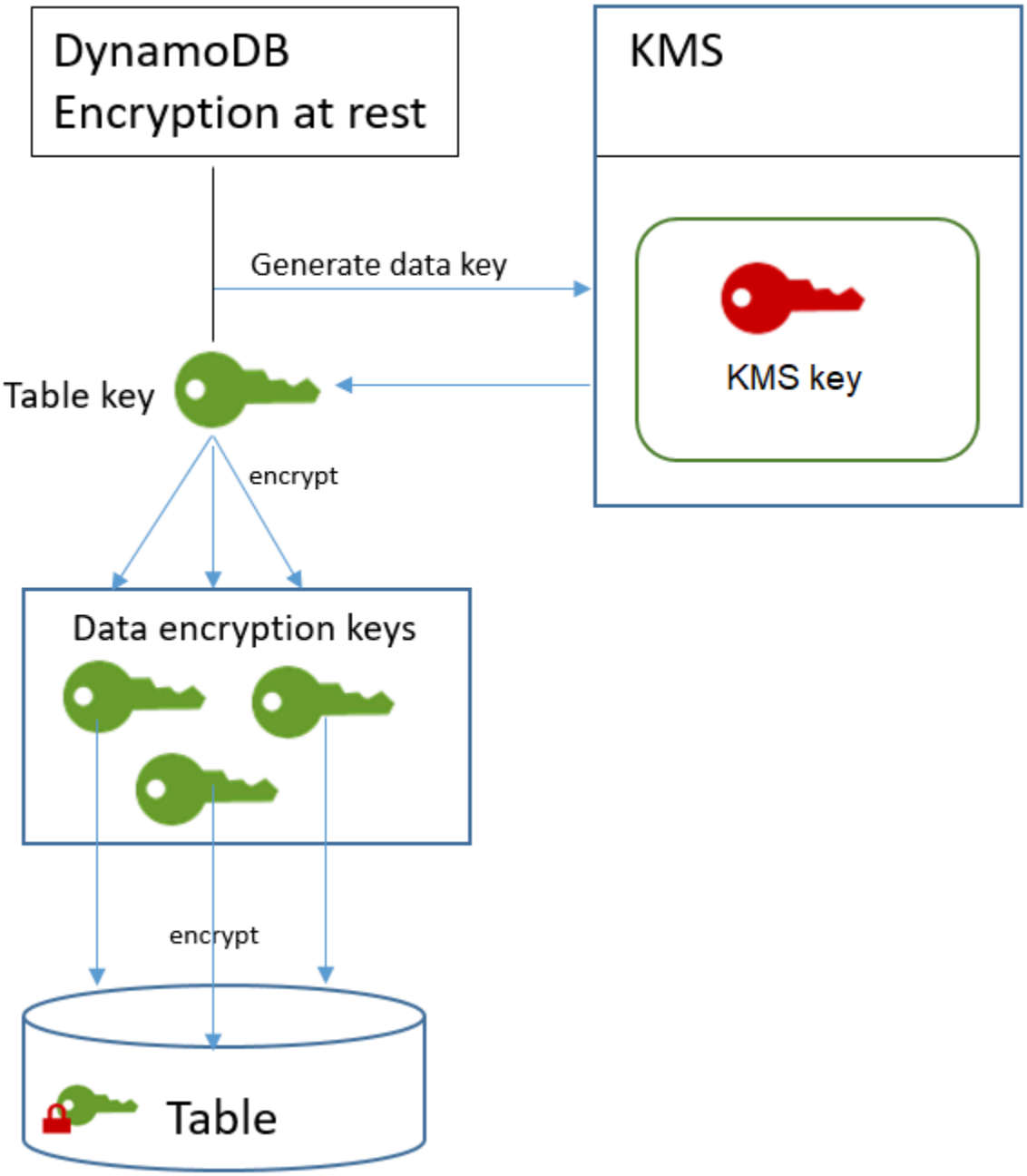
- KMS キーを表示し、そのキーポリシーを表示できます。(キーポリシーの変更はできません)。
- DynamoDB テーブルの暗号化と復号を監査するには、 AWS CloudTrail ログで AWS KMS への DynamoDB API コールを調べます。

ただし、AWS 所有のキー は無料で、その使用は AWS KMS リソースクォータまたはリクエストクォータに対してカウントされません。カスタマーマネージドキーおよび AWS マネージドキーには、API コールごとに料金が発生し、これらの KMS キーには AWS KMS クォータが適用されます。

テーブルキー

DynamoDB は、テーブルの KMS キーを使用して、テーブルキーと呼ばれるテーブルの一意のデータキーを生成して暗号化します。テーブルキーは、暗号化されたテーブルの存続期間中は保持されます。

テーブルキーは、キー暗号化キーとして使用されます。DynamoDB は、このテーブルキーを使用して、テーブルデータの暗号化に使用されるデータ暗号化キーを保護します。DynamoDB は、テーブル内の基礎となる各構造に対して一意のデータ暗号化キーを生成しますが、複数のテーブル項目が同じデータ暗号化キーで保護される場合があります。



暗号化されたテーブルに最初にアクセスすると、DynamoDB は、KMS キーを使用してテーブルキーを復号するリクエストを AWS KMS に送信します。その後、プレーンテキストテーブルキーを使用してデータ暗号化キーを復号化し、プレーンテキストデータ暗号化キーを使用してテーブルデータを復号化します。

DynamoDB は AWS KMS の外部でテーブルキーとデータ暗号化キーを保存して使用します。これによって、Advanced Encryption Standard (AES) 暗号化および 256 ビット暗号化キーのすべて

のキーが保護されます。続いて、暗号化されたキーを暗号化されたデータと一緒に保存します。これらのキーおよびデータは、必要なときにテーブルデータの暗号化に使用できます。

テーブルの KMS キーを変更すると、DynamoDB は新しいテーブルキーを生成します。次に、新しいテーブルキーを使用してデータ暗号化キーの再暗号化が行われます。

テーブルキーのキャッシュ

DynamoDB オペレーションごとに AWS KMS を呼び出さないように、DynamoDB は各呼び出しのプレーンテキストのテーブルキーをメモリにキャッシュします。DynamoDB では、キャッシュしたテーブルキーが 5 分間非アクティブ状態であった後にリクエストを取得すると、AWS KMS に新しいリクエストを送信してテーブルキーを復号します。この呼び出しは、テーブルキーの復号を求める前回のリクエスト以降に AWS KMS または AWS Identity and Access Management (IAM) で KMS キーのアクセスポリシーに加えられた変更をすべてキャプチャします。

KMS キーの使用を認可する

DynamoDB テーブルを保護するために、アカウントでカスタマーマネージドキーまたは AWS マネージドキー を使用する場合は、その KMS キーのポリシーが DynamoDB に、ユーザーに代わってキーを使用する許可を付与する必要があります。DynamoDB の AWS マネージドキー の認可コンテキストには、そのキーポリシーとそれを使用するアクセス許可を委任するグラントが含まれます。

AWS マネージドキー はアカウントにあり、そのポリシーとグラントを表示できるため、カスタマーマネージドキーのポリシーとグラントを完全に制御することができます。ただし、AWS によって管理されているため、ポリシーを変更することはできません。

DynamoDB では、デフォルトの AWS 所有のキー を使用して AWS アカウント の DynamoDB テーブルを保護するための追加の認可は不要です。

トピック

- AWS マネージドキー のキーポリシー
- カスタマーマネージドキーのキーポリシー
- グラントを使用した DynamoDB の承認

AWS マネージドキー のキーポリシー

DynamoDB は、DynamoDB リソースにアクセスしているユーザーの代わりに、暗号化オペレーションで DynamoDB (aws/dynamodb) の AWS マネージドキー を使用します。AWS マネージドキー の

キーポリシーはアカウントのすべてのユーザーに、指定されたオペレーションで AWS マネージドキー を使用する許可を付与します。ただし、アクセス許可が付与されるのは、DynamoDB がユーザーの代わりにリクエストを行う場合のみです。キーポリシーの ViaService 条件では、リクエストが DynamoDB サービスで発生しない限り、ユーザーに AWS マネージドキー の使用を許可しません。

このキーポリシーは、すべての AWS マネージドキー のポリシーと同様に、AWS によって確立されます。このポリシーを変更することはできませんが、いつでも表示できます。詳細については、「Viewing a key policy」を参照してください。

このキーポリシーのポリシーステートメントには次の効果があります

- リクエストがユーザーの代わりに DynamoDB から送信された場合、アカウントのユーザーが DynamoDB の AWS マネージドキー を暗号化オペレーションで使用することを許可します。また、このポリシーはユーザーに KMS キーのグラントの作成も許可します。
- 認可された IAM がアカウントで DynamoDB の AWS マネージドキー のプロパティを表示し、DynamoDB の KMS キー使用を許可するグラントを取り消すことができるようにします。DynamoDB は、継続的なメンテナンス操作にグラントを使用します。
- DynamoDB が読み取り専用オペレーションを実行して、アカウントで DynamoDB の AWS マネージドキー を検索できるようにします。

JSON

```
{
  "Version":"2012-10-17",
  "Id" : "auto-dynamodb-1",
  "Statement" : [ {
    "Sid" : "Allow access through Amazon DynamoDB for all principals in the
 account that are authorized to use Amazon DynamoDB",
    "Effect" : "Allow",
    "Principal" : {
      "AWS" : "*"
    },
    "Action" : [ "kms:Encrypt", "kms:Decrypt", "kms:ReEncrypt*",
 "kms:GenerateDataKey*", "kms:CreateGrant", "kms:DescribeKey" ],
    "Resource" : "*",
    "Condition" : {
      "StringEquals" : {
        "kms:CallerAccount" : "111122223333",
```

```
          "kms:ViaService" : "dynamodb.us-west-2.amazonaws.com"
        }
      }
    }, {
      "Sid" : "Allow direct access to key metadata to the account",
      "Effect" : "Allow",
      "Principal" : {
        "AWS" : "arn:aws:iam::111122223333:root"
      },
      "Action" : [ "kms:Describe*", "kms:Get*", "kms:List*", "kms:RevokeGrant" ],
      "Resource" : "*"
    }, {
      "Sid" : "Allow DynamoDB Service with service principal name
   dynamodb.amazonaws.com to describe the key directly",
      "Effect" : "Allow",
      "Principal" : {
        "Service" : "dynamodb.amazonaws.com"
      },
      "Action" : [ "kms:Describe*", "kms:Get*", "kms:List*" ],
      "Resource" : "*"
    } ]
}
```

カスタマーマネージドキーのキーポリシー

カスタマーマネージドキーを選択して DynamoDB テーブルを保護する際、DynamoDB は、選択を行うプリンシパルの代わりに KMS キーを使用するアクセス許可を取得します。そのプリンシパル、ユーザー、ロールは、DynamoDB に必要な KMS キーに対するアクセス許可を持っている必要があります。これらの許可は、キーポリシー、IAM ポリシー、またはグラントで指定できます。

DynamoDB には、少なくとも、カスタマーマネージドキーに対する次のアクセス許可が必要です。

- kms:Encrypt
- kms:Decrypt
- kms:ReEncrypt* (for kms:ReEncryptFrom および kms:ReEncryptTo 向け)
- kms:GenerateDataKey* (for kms:GenerateDataKey および kms:GenerateDataKeyWithoutPlaintext 向け)
- kms:DescribeKey
- kms:CreateGrant

例えば、次のキーポリシーの例では、必要なアクセス許可のみを提供します。このポリシーには、以下の影響があります。

- DynamoDB が、DynamoDB の使用許可を持つアカウントのプリンシパルの代わりに動作している場合にのみ、DynamoDB が暗号化オペレーションで KMS キーを使用し、グラントを作成することを許可します。ポリシーステートメントで指定されたプリンシパルに DynamoDB を使用する権限がない場合、呼び出しは DynamoDB サービスからのものであっても失敗します。
- KMS: viaService 条件キーでは、ポリシーステートメントにリストされているプリンシパルの代わりに DynamoDB からリクエストが送信された場合にのみアクセス許可が許可されます。これらのプリンシパルは、これらのオペレーションを直接呼び出すことはできません。`kms:ViaService` の値である `dynamodb.*.amazonaws.com` は、リージョンの位置にアスタリスク (*) が付いていることに注意してください。DynamoDB では、クロスリージョン呼び出しを実行して DynamoDB グローバルテーブルをサポートできるように、特定の AWS リージョン から独立したアクセス許可が必要です。
- KMS キー管理者 (`db-team` ロールを引き受けることができるユーザー) に、KMS キーへの読み取り専用アクセス許可、およびグラント (テーブルを保護するために DynamoDB が必要とするグラントを含む) を取り消す許可を付与します。

サンプルキーポリシーを使用する前に、サンプルプリンシパルを AWS アカウント の実際のプリンシパルに置き換えます。

JSON

```
{
  "Id": "key-policy-dynamodb",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid" : "Allow access through Amazon DynamoDB for all principals in the
 account that are authorized to use Amazon DynamoDB",
      "Effect": "Allow",
      "Principal": {"AWS": "arn:aws:iam::111122223333:user/db-lead"},
      "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey",
```

```
        "kms:CreateGrant"
      ],
      "Resource": "*",
      "Condition": {
         "StringLike": {
           "kms:ViaService" : "dynamodb.*.amazonaws.com"
         }
      }
    },
    {
      "Sid":  "Allow administrators to view the KMS key and revoke grants",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/db-team"
       },
      "Action": [
        "kms:Describe*",
        "kms:Get*",
        "kms:List*",
        "kms:RevokeGrant"
      ],
      "Resource": "*"
    }
  ]
}
```

グラントを使用した DynamoDB の承認

キーポリシーに加えて、DynamoDB は DynamoDB (aws/dynamodb) のためにカスタマーマネージドキーまたは AWS マネージドキー の許可を設定するためのグラントを使用します。アカウントの KMS キーのグラントを表示するには、ListGrants 操作を使用します。DynamoDB では、AWS 所有のキー を使用してテーブルを保護するために、グラントや追加のアクセス許可は必要ありません。

DynamoDB は、バックグラウンドシステムメンテナンスと継続的なデータ保護タスクを実行するときに、許可を付与します。また、テーブルキーの生成にグラントを使用します。

各グラントは、テーブルに固有です。アカウントに同じ KMS キーで暗号化された複数のテーブルがある場合、テーブルごとに各タイプのグラントがあります。グラントは、テーブル名と AWS アカウント ID を含む DynamoDB 暗号化コンテキストによって制約を受けます。また、グラントには、それが不要になった場合にグラントを使用停止にする許可が含まれます。

グラントを作成するには、DynamoDB に、暗号化されたテーブルを作成したユーザーに代わって `CreateGrant` を呼び出すための許可が必要です。AWS マネージドキー では、DynamoDB が認可されたユーザーの代わりにリクエストを行う場合にのみ、アカウントのユーザーに KMS キーで CreateGrant の呼び出しを許可するキーポリシーからの `kms:CreateGrant` 許可を DynamoDB が取得します。

キーポリシーは、アカウントが KMS キーのグラントを取り消すことも許可できます。ただし、アクティブな暗号化されたテーブルのグラントを取り消すと、DynamoDB はテーブルを保護および維持できなくなります。

DynamoDB 暗号化コンテキスト

暗号化コンテキストは、一連のキー値のペアおよび任意非シークレットデータを含みます。データを暗号化するリクエストに暗号化コンテキストを組み込むと、AWS KMS は暗号化コンテキストを暗号化されたデータに暗号化してバインドします。データを復号するには、同じ暗号化コンテキストに渡す必要があります。

DynamoDB は、すべての AWS KMS 暗号化オペレーションで同じ暗号化コンテキストを使用します。カスタマーマネージドキーまたは AWS マネージドキー を使用して DynamoDB テーブルを保護する場合、暗号化コンテキストを使用して監査レコードやログにおける KMS キーの使用を特定することができます。これは、AWS CloudTrail や Amazon CloudWatch Logs などの口グにもプレーンテキストで表示されます。

また、暗号化コンテクストはポリシーとグラントの認可用の条件としても使用できます。DynamoDB は暗号化コンテキストを使用して、アカウントとリージョンのカスタマーマネージドキーまたは AWS マネージドキー へのアクセスを許可するグラントを制限します。

DynamoDB は AWS KMS へのリクエストで、2 つのキーバリューペアを持つ暗号化コンテキストを使用します。

```
"encryptionContextSubset": {
    "aws:dynamodb:tableName": "Books"
    "aws:dynamodb:subscriberId": "111122223333"
}
```

- テーブル — 最初のキーと値のペアは、DynamoDB が暗号化しているテーブルを識別します。キーは、`aws:dynamodb:tableName` です。この値は、テーブルの名前です。

  ```
  "aws:dynamodb:tableName": "<table-name>"
  ```

以下はその例です。

```
"aws:dynamodb:tableName": "Books"
```

- アカウント - 2 番目のキーバリューペアは、AWS アカウント を識別します。キーは、aws:dynamodb:subscriberId です。値は、アカウント ID です。

```
"aws:dynamodb:subscriberId": "<account-id>"
```

以下はその例です。

```
"aws:dynamodb:subscriberId": "111122223333"
```

DynamoDB の AWS KMS との対話をモニタリングする

カスタマーマネージドキーまたは AWS マネージドキーを使用して DynamoDB テーブルを保護する場合は、AWS CloudTrail ログを使用して、DynamoDB がユーザーに代わって AWS KMS に送信するリクエストを追跡することができます。

このセクションでは、GenerateDataKey、Decrypt、および CreateGrant リクエストを説明します。さらに、DynamoDB は DescribeKey オペレーションを使用して、選択した KMS キーがアカウントとリージョンに存在するかどうかを判断します。また、 RetireGrant 操作を使用して、テーブルを削除するときにグラントを削除します。

GenerateDataKey

テーブルで保存時の暗号化を有効にすると、DynamoDB は一意のテーブルキーを作成します。また、GenerateDataKey リクエストを、テーブルの KMS キーを指定する AWS KMS に送信します。

GenerateDataKey 演算を記録するイベントは、次のようなサンプルイベントになります。ユーザーは DynamoDB サービスアカウントです。パラメータには、KMS キーの Amazon リソースネーム (ARN)、256 ビットキーを必要とするキー指定子、テーブルと AWS アカウント を識別する暗号化コンテキストが含まれます。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
```

```
        "type": "AWSService",
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T00:15:17Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "aws:dynamodb:tableName": "Services",
            "aws:dynamodb:subscriberId": "111122223333"
        },
        "keySpec": "AES_256",
        "keyId": "1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    "responseElements": null,
    "requestID": "229386c1-111c-11e8-9e21-c11ed5a52190",
    "eventID": "e3c436e9-ebca-494e-9457-8123a1f5e979",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333",
    "sharedEventID": "bf915fa6-6ceb-4659-8912-e36b69846aad"
}
```


Decrypt

暗号化された DynamoDB テーブルにアクセスする場合、DynamoDB はテーブルキーを復号化して、階層内でその下にあるキーを復号化できるようにする必要があります。次に、テーブル内のデータを復号化します。テーブルキーを復号化するには、次が実行されます。DynamoDB は、テーブルの KMS キーを指定する復号リクエストを AWS KMS に送信します。

`Decrypt` 演算を記録するイベントは、次のようなサンプルイベントになります。ユーザーは、テーブルにアクセスしている AWS アカウント のプリンシパルです。パラメータには、暗号化さ

れたテーブルキー (暗号化テキストの blob として)、およびテーブルと AWS アカウント を識別する暗号化コンテキストが含まれます。AWS KMS は、暗号化テキストから KMS キーの ID を取得します。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-02-14T16:42:15Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDT3HGFQZX4RY6RU",
                "arn": "arn:aws:iam::111122223333:role/Admin",
                "accountId": "111122223333",
                "userName": "Admin"
            }
        },
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T16:42:39Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters":
    {
        "encryptionContext":
        {
            "aws:dynamodb:tableName": "Books",
            "aws:dynamodb:subscriberId": "111122223333"
        }
    },
    "responseElements": null,
    "requestID": "11cab293-11a6-11e8-8386-13160d3e5db5",
```

```
    "eventID": "b7d16574-e887-4b5b-a064-bf92f8ec9ad3",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

CreateGrant

カスタマーマネージドキーまたは AWS マネージドキーを使用して DynamoDB テーブルを保護する場合、DynamoDB は許可を使用して、サービスが継続的なデータ保護、メンテナンス、耐久性のタスクを実行できるようにします。これらのグラントは、AWS 所有のキー では不要です。

DynamoDB が作成するグラントは、テーブルに固有です。CreateGrant リクエストのプリンシパルは、テーブルを作成したユーザーです。

`CreateGrant` 演算を記録するイベントは、次のようなサンプルイベントになります。パラメータには、テーブルの KMS キーの Amazon リソースネーム (ARN)、被付与者プリンシパルと使用停止プリンシパル (DynamoDB サービス)、グラントの対象となるオペレーションが含まれます。また、指定された暗号化コンテキストを使用するすべての暗号化オペレーションを必要とする制約も含まれています。

```
{
    "eventVersion": "1.05",
    "userIdentity":
    {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-02-14T00:12:02Z"
            },
```

```
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/Admin",
                "accountId": "111122223333",
                "userName": "Admin"
            }
        },
        "invokedBy": "dynamodb.amazonaws.com"
    },
    "eventTime": "2018-02-14T00:15:15Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "CreateGrant",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "dynamodb.amazonaws.com",
    "userAgent": "dynamodb.amazonaws.com",
    "requestParameters": {
        "keyId": "1234abcd-12ab-34cd-56ef-1234567890ab",
        "retiringPrincipal": "dynamodb.us-west-2.amazonaws.com",
        "constraints": {
            "encryptionContextSubset": {
                "aws:dynamodb:tableName": "Books",
                "aws:dynamodb:subscriberId": "111122223333"
            }
        },
        "granteePrincipal": "dynamodb.us-west-2.amazonaws.com",
        "operations": [
            "DescribeKey",
            "GenerateDataKey",
            "Decrypt",
            "Encrypt",
            "ReEncryptFrom",
            "ReEncryptTo",
            "RetireGrant"
        ]
    },
    "responseElements": {
        "grantId":
"5c5cd4a3d68e65e77795f5ccc2516dff057308172b0cd107c85b5215c6e48bde"
    },
    "requestID": "2192b82a-111c-11e8-a528-f398979205d8",
    "eventID": "a03d65c3-9fee-4111-9816-8bf96b73df01",
    "readOnly": false,
    "resources": [
```

```
        {
            "ARN": "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

DynamoDB での暗号化テーブルの管理

AWS マネジメントコンソール または AWS Command Line Interface (AWS CLI) を使用すると、新しいテーブルで暗号化キーを指定し、Amazon DynamoDB の既存のテーブルで暗号化キーを更新できます。

トピック

- 新しいテーブルの暗号化キーを指定する
- 暗号化キーの更新

新しいテーブルの暗号化キーを指定する

Amazon DynamoDB コンソールまたは AWS CLI を使用して新しいテーブルで暗号化キーを指定するには、以下の手順に従います。

暗号化されたテーブルの作成 (コンソール)

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. [Create Table] (テーブルの作成) を選択します。[Table name] (テーブル名) に「**Music**」と入力します。プライマリキーに「**Artist**」と入力し、ソートキーに「**SongTitle**」をそれぞれ文字列として入力します。
4. [Settings] (設定) では、[Customize settings] (設定のカスタマイズ) が選択されていることを確認してください。

> **Note**
>
> [Use default settings] (デフォルト設定の使用) を選択した場合、テーブルは AWS 所有のキー を使用して保管時に暗号化され、追加コストはかかりません。

5. [Encryption at rest] (保管時の暗号化) で、暗号化タイプ – AWS 所有のキー、AWS マネージドキー、またはカスタマーマネージドキーを選択します。

   - [Owned by Amazon DynamoDB] (Amazon DynamoDB が所有)。AWS が所有するキーで、特に DynamoDB が所有および管理します。このキーは追加料金なしで使用されます。
   - AWS マネージドキー。キーエイリアス: `aws/dynamodb`。キーはアカウントに保存され、AWS Key Management Service (AWS KMS) によって管理されます (AWS KMS の料金が適用されます)。
   - [Stored in your account, and owned and managed by you.] (アカウントに保存され、お客様が所有および管理。) カスタマーマネージドキー。キーはアカウントに保存され、AWS Key Management Service (AWS KMS) によって管理されます (AWS KMS の料金が適用されます)。

   > **Note**
   >
   > 独自のキーを所有および管理することを選択した場合は、KMS キーポリシーが適切に設定されていることを確認してください。例を含む詳細については、「カスタマーマネージドキーのキーポリシー」を参照してください。

6. [Create table] (テーブルの作成) を選択して暗号化テーブルを作成します。暗号化タイプを確認するには、[Overview] (概要) タブのテーブルの詳細を選択し、[Additional details] (その他の詳細) セクションを表示します。

暗号化されたテーブルの作成 (AWS CLI)

AWS CLI により、Amazon DynamoDB のデフォルトの AWS 所有のキー、AWS マネージドキー、またはカスタマーマネージドキーを使用してテーブルを作成します。

デフォルトの AWS 所有のキー で暗号化テーブルを作成するには

- 暗号化された `Music` テーブルを次のとおりに作成します。

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5
```

> **Note**
>
> このテーブルは、DynamoDB サービスアカウントでデフォルトの AWS 所有のキー を使用して暗号化されます。

AWS マネージドキー で DynamoDB 用の暗号化テーブルを作成するには

- 暗号化された `Music` テーブルを次のとおりに作成します。

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --sse-specification Enabled=true,SSEType=KMS
```

テーブルの説明の `SSEDescription` ステータスは、`ENABLED` に設定され、`SSEType` は `KMS` になります。

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
```

```
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

カスタマーマネージドキーで DynamoDB 用の暗号化テーブルを作成するには

- 暗号化された Music テーブルを次のとおりに作成します。

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
      AttributeName=Artist,AttributeType=S \
      AttributeName=SongTitle,AttributeType=S \
  --key-schema \
      AttributeName=Artist,KeyType=HASH \
      AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
      ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-
a123-ab1234a1b234
```

> **Note**
>
> KMSMasterKeyId の場合、キー ID、キー ARN、またはキーエイリアスを使用できます。キーエイリアス (alias/my-key など) を使用する場合、DynamoDB はエイリアスを解決し、基盤となる AWS KMS キーをテーブルに関連付けます。テーブルの説明では、KMSMasterKeyArn はエイリアスではなく、解決されたキーのキー ARN を常に表示します。キー識別子の詳細については、「AWS Key Management Service デベロッパーガイド」の「キー識別子 (KeyId)」を参照してください。

テーブルの説明の SSEDescription ステータスは、ENABLED に設定され、SSEType は KMS になります。

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
```

```
}
```

暗号化キーの更新

DynamoDB コンソールまたは AWS CLI を使用して、AWS 所有のキー、AWS マネージドキー、カスタマーマネージドキーの間でいつでも既存のテーブルの暗号化キーを更新することができます。

暗号化キーの更新 (コンソール)

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. 更新するテーブルを選択します。
4. [Actions] (アクション) ドロップダウンを選択し、[Update settings] (設定の更新) オプションを選択します。
5. [Additional settings] (追加設定) タブに移動します。
6. [Encryption] (暗号化) の下の、[Manage encryption] (暗号化の管理) を選択します。
7. 暗号化タイプを選択します。

   - [Owned by Amazon DynamoDB.] (Amazon DynamoDB によって所有される。) AWS KMS キーは DynamoDB によって所有、管理されます。このキーは追加料金なしで使用されます。
   - AWS マネージドキーのキーエイリアス: `aws/dynamodb`。キーはアカウントに保存され、AWS Key Management Service (AWS KMS) によって管理されます (AWS KMS の料金が適用されます)。
   - [Stored in your account, and owned and managed by you.] (アカウントに保存され、ユーザーが所有、管理される。) キーはアカウントに保存され、AWS Key Management Service (AWS KMS) によって管理されます (AWS KMS の料金が適用されます)。

   > **Note**
   >
   > 独自のキーを所有および管理することを選択した場合は、KMS キーポリシーが適切に設定されていることを確認してください。詳細については、「カスタマーマネージドキーのキーポリシー」を参照してください。

続いて、[Save] (保存) を選択して暗号化されたテーブルを更新します。暗号化タイプを確認するには、[概要] タブでテーブルの詳細を参照します。

暗号化キーの更新 (AWS CLI)

次の例は、AWS CLI を使用して暗号化テーブルを更新する方法を示しています。

デフォルトの AWS 所有のキー で暗号化テーブルを更新するには

- 暗号化された `Music` テーブルを次の例のとおりに更新します。

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=false
```

> **Note**
>
> このテーブルは、DynamoDB サービスアカウントでデフォルトの AWS 所有のキー を使用して暗号化されます。

DynamoDB 用の AWS マネージドキー で暗号化テーブルを更新するには

- 暗号化された `Music` テーブルを次の例のとおりに更新します。

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true
```

テーブルの説明の `SSEDescription` ステータスは、`ENABLED` に設定され、`SSEType` は `KMS` になります。

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
```

```
}
```

DynamoDB のカスタマーマネージドキーで暗号化テーブルを更新するには

- 暗号化された Music テーブルを次の例のとおりに更新します。

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-
a123-ab1234a1b234
```

> **Note**
>
> KMSMasterKeyId の場合、キー ID、キー ARN、またはキーエイリアスを使用できます。キーエイリアス (alias/my-key など) を使用する場合、DynamoDB はエイリアスを解決し、基盤となる AWS KMS キーをテーブルに関連付けます。テーブルの説明では、KMSMasterKeyArn はエイリアスではなく、解決されたキーのキー ARN を常に表示します。

テーブルの説明の SSEDescription ステータスは、ENABLED に設定され、SSEType は KMS になります。

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-
a123-ab1234a1b234",
}
```

VPC エンドポイントと IAM ポリシーを使用した DynamoDB 接続の保護

接続は、Amazon DynamoDB とオンプレミスのアプリケーション間、および同じの AWS リージョン内の DynamoDB と他の AWS リソース間で保護されます。

エンドポイントに必要なポリシー

Amazon DynamoDB には、リージョンのエンドポイント情報を列挙できる DescribeEndpoints API が用意されています。パブリック DynamoDB エンドポイントへのリクエストの場合、API は、IAM または VPC エンドポイントポリシーに明示的または暗黙的な拒否があっても、設定された DynamoDB IAM ポリシーに関係なく応答します。これは、DynamoDB が `DescribeEndpoints` API の承認を意図的にスキップするためです。

VPC エンドポイントからのリクエストの場合、IAM エンドポイントポリシーと仮想プライベートクラウド (VPC) エンドポイントポリシーの両方が、IAM の `dynamodb:DescribeEndpoints` アクションを使用して、リクエスト元の ID およびアクセス管理 (IAM) プリンシパルの `DescribeEndpoints` API コールを承認する必要があります。それ以外の場合、`DescribeEndpoints` API へのアクセスは拒否されます。

以下は、エンドポイントポリシーの例です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "dynamodb:DescribeEndpoints",
            "Resource": "*"
        }
    ]
}
```

サービスとオンプレミスのクライアントおよびアプリケーションとの間のトラフィック

プライベートネットワークと AWS との間には 2 つの接続オプションがあります:

- AWS Site-to-Site VPN 接続。詳細については、AWS Site-to-Site VPN ユーザーガイド の「What is AWS Site-to-Site VPN? (とは ?)」 を参照してください。

- Direct Connect 接続。詳細については、Direct Connect ユーザーガイド の「What is Direct Connect? (とは ?)」 を参照してください。

ネットワーク経由での DynamoDB へのアクセスは、AWS が発行する API を利用して行われます。クライアントは Transport Layer Security (TLS) 1.2 をサポートしている必要があります。TLS 1.3 をお勧めします。クライアントは、Ephemeral Diffie-Hellman (DHE) や Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) などの Perfect Forward Secrecy (PFS) を備えた暗号スイートもサポートする必要があります。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。また、リクエストには、IAM プリンシパルに関連付けられたアクセスキー ID およびシークレットアクセスキーによる署名が必要です。または、リクエストへの署名のために一時的にセキュリティ認証情報を生成する AWS Security Token Service (STS) を使用することもできます。

同じリージョン内の AWS リソース間のトラフィック

Amazon Virtual Private Cloud (Amazon VPC) endpoint for DynamoDB は、DynamoDB への接続のみを許可する VPC 内の論理エンティティです。Amazon VPC はリクエストを DynamoDB にルーティングし、レスポンスを VPC にルーティングします。詳細については、Amazon VPC ユーザーガイドの「VPC エンドポイント」を参照してください。VPC エンドポイントからのアクセスのコントロールに使用できるポリシーの例については、「IAM ポリシーを使用して DynamoDB へのアクセスをコントロールします」を参照してください。

> **Note**
>
> Amazon VPC エンドポイントには、AWS Site-to-Site VPN または Direct Connect を使用してアクセスすることはできません。

AWS Identity and Access Management (IAM) と DynamoDB

AWS Identity and Access Management は、管理者が AWS リソースへのアクセスを安全にコントロールするために役立つ AWS のサービスです。管理者は、誰を認証 (サインイン) し、誰に Amazon DynamoDB および DynamoDB Accelerator リソースの使用を承認する (アクセス許可を付与する) かを制御します。IAM を使用して、Amazon DynamoDB と DynamoDB Accelerator の両方のアクセス許可を管理し、セキュリティポリシーを実装できます。IAM は、AWS のサービスで追加料金は発生しません。

トピック

- Amazon DynamoDB の Identity and Access Management
- 詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用

Amazon DynamoDB の Identity and Access Management

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御するために役立つ AWS のサービス です。IAM 管理者は、誰が認証 (サインイン) され、DynamoDB リソースを使用する認可 を受ける (アクセス許可がある) ことができるかを管理します。IAM は、追加費用なしで使用できる AWS のサービス です。

トピック

- 対象者
- アイデンティティによる認証
- ポリシーを使用したアクセス権の管理
- Amazon DynamoDB で IAM が機能する仕組み
- Amazon DynamoDB のアイデンティティベースのポリシーの例
- Amazon DynamoDB アイデンティティとアクセスのトラブルシューティング
- DynamoDB リザーブドキャパシティの購入を防止する IAM ポリシー

対象者

AWS Identity and Access Management (IAM) の使用方法は、ロールによって異なります。

- サービスユーザー - 機能にアクセスできない場合は、管理者にアクセス許可をリクエストします (「Amazon DynamoDB アイデンティティとアクセスのトラブルシューティング」を参照)。
- サービス管理者 - ユーザーアクセスを決定し、アクセス許可リクエストを送信します (「Amazon DynamoDB で IAM が機能する仕組み」を参照)
- IAM 管理者 - アクセスを管理するためのポリシーを作成します (「Amazon DynamoDB のアイデンティティベースのポリシーの例」を参照)

アイデンティティによる認証

認証とは、アイデンティティ認証情報を使用して AWS にサインインする方法です。ユーザーは、IAM ユーザー の AWS アカウントのルートユーザー として、または IAM ロールを引き受けることによって、認証される 必要があります。

AWS IAM アイデンティティセンター (IAM アイデンティティセンター)、シングルサインオン認証、Google/Facebook 認証情報などの ID ソースからの認証情報を使用して、フェデレーテッドアイデンティティとしてサインインできます。サインインの詳細については、「AWS サインイン ユーザーガイド」の「AWS アカウントにサインインする方法」を参照してください。

プログラムによるアクセスの場合、AWS はリクエストに暗号で署名するための SDK と CLI を提供します。詳細については、「IAM ユーザーガイド」の「API リクエストに対する AWS 署名バージョン 4」を参照してください。

AWS アカウント のルートユーザー

AWS アカウントを作成すると、すべての AWS のサービスとリソースに対する完全なアクセス権を持つ AWS アカウント ルートユーザーと呼ばれる 1 つのサインイン ID を使用して開始します。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザー認証情報を必要とするタスクについては、「IAM ユーザーガイド」の「ルートユーザー認証情報が必要なタスク」を参照してください。

フェデレーテッドアイデンティティ

ベストプラクティスでは、人間のユーザーが一時的な認証情報を使用して AWS のサービス にアクセスする際、アイデンティティプロバイダーとのフェデレーションを使用することが求められます。

フェデレーテッドアイデンティティは、エンタープライズディレクトリ、ウェブ ID プロバイダー、Directory Service のユーザーであり、ID ソースからの認証情報を使用して AWS のサービスにアクセスするユーザーです。フェデレーテッドアイデンティティは、一時的な認証情報を提供するロールを引き受けます。

アクセスを一元管理する場合は、AWS IAM アイデンティティセンター をお勧めします。詳細については、「AWS IAM アイデンティティセンター ユーザーガイド」の「IAM アイデンティティセンターとは」を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、特定の個人やアプリケーションに対する特定のアクセス許可を持つアイデンティティです。長期認証情報を持つ IAM ユーザーの代わりに一時的な認証情報を使用することをお勧め

します。詳細は「IAM ユーザーガイド」の「人間のユーザーが一時的な認証情報を使用して AWS にアクセスするには ID プロバイダーとのフェデレーションの使用が必要です」を参照してください。

IAM グループは、IAM ユーザーの集合を指定し、大量のユーザーに対するアクセス許可の管理を容易にします。詳細については、「IAM ユーザーガイド」の「IAM ユーザーに関するユースケース」を参照してください。

IAM ロール

IAM ロールは、特定のアクセス許可を持つアイデンティであり、一時的な認証情報を提供します。ユーザーから IAM ロール (コンソール) に切り替える、または AWS CLI や AWS API オペレーションを呼び出すことで、ロールを引き受けることができます。詳細については、「IAM ユーザーガイド」の「ロールを引き受けるための各種方法」を参照してください。

IAM ロールは、フェデレーションユーザーアクセス、一時的な IAM ユーザーのアクセス許可、クロスアカウントアクセス、クロスサービスアクセス、および Amazon EC2 で実行するアプリケーションに役立ちます。詳細については、IAM ユーザーガイド の IAM でのクロスアカウントリソースアクセス を参照してください。

ポリシーを使用したアクセス権の管理

AWS でアクセスを制御するには、ポリシーを作成して AWS ID またはリソースにアタッチします。ポリシーは、アイデンティティやリソースとの関連付けに伴うアクセス許可を定義します。AWS は、プリンシパルがリクエストを行うときに、これらのポリシーを評価します。大半のポリシーは JSON ドキュメントとして AWS に保存されます。JSON ポリシードキュメントの詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は、ポリシーを使用して、どのプリンシパルがどのリソースに対して、どのような条件でアクションを実行できるかを定義することで、誰が何にアクセスできるかを指定します。

デフォルトでは、ユーザーやロールにアクセス許可はありません。IAM 管理者は IAM ポリシーを作成してロールに追加し、このロールをユーザーが引き受けられるようにします。IAM ポリシーは、オペレーションの実行方法を問わず、アクセス許可を定義します。

アイデンティティベースのポリシー

アイデンティティベースのポリシーは、アイデンティティ (ユーザー、グループ、またはロール) にアタッチできる JSON アクセス許可ポリシードキュメントです。これらのポリシーは、アイデンティティがどのリソースに対してどのような条件下でどのようなアクションを実行できるかを制御します。アイデンティティベースポリシーの作成方法については、IAM ユーザーガイド の カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する を参照してください。

アイデンティティベースのポリシーは、インラインポリシー (単一の ID に直接埋め込む) または管理ポリシー (複数の ID にアタッチされたスタンドアロンポリシー) にすることができます。管理ポリシーとインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。例としては、IAM ロール信頼ポリシーや Amazon S3 バケットポリシーなどがあります。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーで IAM の AWS マネージドポリシーを使用することはできません。

その他のポリシータイプ

AWS は、より一般的なポリシータイプで付与された最大数のアクセス許可を設定できる、追加のポリシータイプをサポートしています。

- アクセス許可の境界 – アイデンティティベースのポリシーで IAM エンティティに付与することのできるアクセス許可の数の上限を設定します。詳細については、「IAM ユーザーガイド」の「IAM エンティティのアクセス許可境界」を参照してください。
- サービスコントロールポリシー (SCP) - AWS Organizations 内の組織または組織単位の最大のアクセス許可を指定します。詳細については、「AWS Organizations ユーザーガイド」の「サービスコントロールポリシー」を参照してください。
- リソースコントロールポリシー (RCP) – は、アカウント内のリソースで利用できる最大数のアクセス許可を定義します。詳細については、「AWS Organizations ユーザーガイド」の「リソースコントロールポリシー (RCP)」を参照してください。
- セッションポリシー – ロールまたはフェデレーションユーザーの一時セッションを作成する際にパラメータとして渡される高度なポリシーです。詳細については、「IAM ユーザーガイド」の「セッションポリシー」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成されるアクセス許可を理解するのがさらに難しくなります。複数のポリシータイプが関連するとき、リクエストを許可するかどうかを AWS が決定する方法の詳細については、IAM ユーザーガイド の ポリシーの評価ロジック を参照してください。

Amazon DynamoDB で IAM が機能する仕組み

IAM を使用して DynamoDB へのアクセスを管理する前に、DynamoDB で利用できる IAM の機能について学びます。

IAM 機能	DynamoDB のサポート
アイデンティティベースのポリシー	あり
リソースベースのポリシー	はい
ポリシーアクション	あり
ポリシーリソース	あり
ポリシー条件キー	あり
ACL	なし
ABAC (ポリシー内のタグ)	あり
一時的な認証情報	あり
プリンシパルアクセス権限	あり
サービスロール	あり
サービスリンクロール	はい

大部分の IAM 機能が DynamoDB および AWS のその他のサービスでどのように連携するかに関するおおまかな説明については、「IAM ユーザーガイド」の「IAM と連携する AWS のサービス」を参照してください。

DynamoDB のアイデンティティベースのポリシー

アイデンティティベースのポリシーのサポート: あり

アイデンティティベースポリシーは、IAM ユーザー、ユーザーグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティ

ベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。JSON ポリシーで使用できるすべての要素について学ぶには、「IAM ユーザーガイド」の「IAM JSON ポリシーの要素のリファレンス」を参照してください。

DynamoDB のアイデンティティベースのポリシーの例

DynamoDB アイデンティティベースのポリシーの例を確認するには、「Amazon DynamoDB のアイデンティティベースのポリシーの例」を参照してください。

DynamoDB 内のリソースベースのポリシー

リソースベースのポリシーのサポート: あり

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシー や Amazon S3 バケットポリシー があげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスをコントロールできます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーで、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービス を含めることができます。

クロスアカウントアクセスを有効にするには、全体のアカウント、または別のアカウントの IAM エンティティを、リソースベースのポリシーのプリンシパルとして指定します。詳細については、IAM ユーザーガイドのIAM でのクロスアカウントリソースアクセスを参照してください。

DynamoDB のポリシーアクション

ポリシーアクションのサポート: あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのようなリソースにどのような条件でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

DynamoDB アクションのリストを確認するには、「サービス認可リファレンス」の「Amazon DynamoDB で定義されるアクション」を参照してください。

DynamoDB のポリシーアクションは、アクションの前に以下のプレフィックスを使用します。

```
aws
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [
      "aws:action1",
      "aws:action2"
         ]
```

DynamoDB アイデンティティベースのポリシーの例を確認するには、「Amazon DynamoDB のアイデンティティベースのポリシーの例」を参照してください。

DynamoDB のポリシーリソース

ポリシーリソースのサポート: あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

`Resource` JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。リソースレベルのアクセス許可をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

DynamoDB リソースのタイプとその ARN のリストを確認するには、「サービス認可リファレンス」の「Amazon DynamoDB で定義されるリソース」を参照してください。各リソースの ARN を指定できるアクションについては、「Amazon DynamoDB で定義されているアクション」を参照してください。

DynamoDB アイデンティティベースのポリシーの例を確認するには、「Amazon DynamoDB のアイデンティティベースのポリシーの例」を参照してください。

DynamoDB のポリシー条件キー

サービス固有のポリシー条件キーのサポート: あり

管理者は AWS JSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

Condition 要素は、定義された基準に基づいてステートメントが実行される時期を指定します。イコールや未満などの条件演算子を使用して条件式を作成して、ポリシーの条件とリクエスト内の値を一致させることができます。すべての AWS グローバル条件キーを確認するには、IAM ユーザーガイドのAWS グローバル条件コンテキストキーを参照してください。

DynamoDB の条件キーの一覧については、「サービス認可リファレンス」の「Amazon DynamoDB の条件キー」を参照してください。どのアクションおよびリソースで条件キーを使用できるかについては、「Amazon DynamoDB で定義されるアクション」を参照してください。

DynamoDB アイデンティティベースのポリシーの例を確認するには、「Amazon DynamoDB のアイデンティティベースのポリシーの例」を参照してください。

DynamoDB のアクセスコントロールリスト (ACL)

ACL のサポート: なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするためのアクセス許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

DynamoDB での属性ベースのアクセス制御 (ABAC)

ABAC (ポリシー内のタグ) のサポート: あり

属性ベースのアクセス制御 (ABAC) は、タグと呼ばれる属性に基づいてアクセス許可を定義する認可戦略です。IAM エンティティと AWS リソースにタグを付けることで、プリンシパルのタグがリソースタグと一致するときに操作を許可する ABAC ポリシーを設計できます。

タグに基づいてアクセスを管理するには、aws:ResourceTag/*key-name*、aws:RequestTag/*key-name*、または aws:TagKeys の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。

サービスがすべてのリソースタイプに対して 3 つの条件キーすべてをサポートする場合、そのサービスの値はありです。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、「IAM ユーザーガイド」の「ABAC 認可でアクセス許可を定義する」を参照してください。ABAC をセットアップする手順を説明するチュートリアルについては、「IAM ユーザーガイド」の「属性ベースのアクセス制御 (ABAC) を使用する」を参照してください。

DynamoDB での一時的な認証情報の使用

一時的な認証情報のサポート: あり

一時的な認証情報は、AWSリソースへの短期的なアクセスを提供し、フェデレーションの使用時またはロールの切り替え時に自動的に作成されます。AWS では、長期的なアクセスキーを使用する代わりに、一時的な認証情報を動的に生成することをお勧めします。詳細については、「IAM ユーザーガイド」の「IAM の一時的な認証情報」および「AWS のサービス と IAM との連携」を参照してください。

DynamoDB のクロスサービスプリンシパル許可

転送アクセスセッション (FAS) のサポート: あり

転送アクセスセッション (FAS) は、AWS のサービス を呼び出すプリンシパルのアクセス許可を AWS のサービス のリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

DynamoDB のサービスロール

サービスロールのサポート: あり

サービスロールとは、サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、IAM ユーザーガイドの AWS のサービス に許可を委任するロールを作成するを参照してください。

> **Warning**
>
> サービスロールの許可を変更すると、DynamoDB の機能が破損する可能性があります。DynamoDB が指示する場合以外は、サービスロールを編集しないでください。

DynamoDB 用のサービスリンクロール

サービスリンクロールのサポート: あり

サービスにリンクされたロールは、AWS のサービス にリンクされているサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスにリンクされたロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスにリンクされたロールのアクセス許可を表示できますが、編集することはできません。

サービスにリンクされたロールの作成または管理の詳細については、「IAM と提携する AWS のサービス」を参照してください。表の「サービスリンクロール」列に Yes と記載されたサービスを見つけます。サービスにリンクされたロールに関するドキュメントをサービスで表示するには、[はい] リンクを選択します。

DynamoDB でサポートされているサービスにリンクされたロール

DynamoDB では、以下のサービスにリンクされたロールがサポートされています。

- DynamoDB は、サービスにリンクされたロール AWSServiceRoleForDynamoDBReplication を使用して、AWS リージョン全体のグローバルテーブルレプリケーションを行います。AWSServiceRoleForDynamoDBReplication サービスにリンクされたロールの詳細については、「DynamoDB グローバルテーブルのセキュリティ」を参照してください。
- DynamoDB Accelerator (DAX) は、サービスにリンクされたロール AWSServiceRoleForDAX を使用して DAX クラスターを設定および維持します。AWSServiceRoleForDAX サービスにリンクされたロールの詳細については、「the section called “DAX のサービスにリンクされたロールの使用”」を参照してください。

DynamoDB は、これらの DynamoDB サービスにリンクされたロールに加えて、Application Auto Scaling サービスを使用して、プロビジョニングされたキャパシティモードテーブルのスループット設定を自動的に管理します。Application Auto Scaling サービスは、サービスにリンクされたロール AWSServiceRoleForApplicationAutoScaling_DynamoDBTable を使用して、自動スケーリングが有効になっている DynamoDB テーブルのスループット設定を管理します。詳細については、「Service-linked roles for Application Auto Scaling」を参照してください。

Amazon DynamoDB のアイデンティティベースのポリシーの例

デフォルトでは、ユーザーおよびロールには、DynamoDB リソースを作成または変更するアクセス許可はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。

これらのサンプルの JSON ポリシードキュメントを使用して IAM アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーを作成する (コンソール)」を参照してください。

DynamoDB が定義するアクションとリソースタイプ (リソースタイプごとの ARN の形式を含む) の詳細については、「サービス認可リファレンス」の「Amazon DynamoDB のアクション、リソース、および条件キー」を参照してください。

トピック

- ポリシーに関するベストプラクティス
- DynamoDB コンソールの使用
- 自分の権限の表示をユーザーに許可する
- Amazon DynamoDB でのアイデンティティベースポリシーの使用

ポリシーに関するベストプラクティス

ID ベースのポリシーは、ユーザーのアカウントで誰かが DynamoDB リソースを作成、アクセス、または削除できるどうかを決定します。これらのアクションでは、AWS アカウント に費用が発生する場合があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS マネージドポリシーの使用を開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードへのアクセス許可の付与を開始するには、多くの一般的なユースケースのためにアクセス許可を付与する AWS マネージドポリシーを使用します。これらは AWS アカウントで使用できます。ユースケースに固有の AWS カスタマー管理ポリシーを定義して、アクセス許可を絞り込むことをお勧めします。詳細については、IAM ユーザーガイド の AWS マネージドポリシー または ジョブ機能の AWS マネージドポリシー を参照してください。
- 最小特権を適用する – IAM ポリシーでアクセス許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、IAM ユーザーガイド の IAM でのポリシーとアクセス許可 を参照してください。

- IAM ポリシーで条件を使用してアクセスをさらに制限する – ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。たとえば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。また、CloudFormation などの特定の AWS のサービス を介して使用する場合、条件を使用してサービスアクションへのアクセスを許可することもできます。詳細については、IAM ユーザーガイド の IAM JSON ポリシー要素:条件 を参照してください。
- IAM アクセスアナライザー を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する – IAM アクセスアナライザー は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、IAM ユーザーガイド の IAM Access Analyzer でポリシーを検証する を参照してください。
- 多要素認証 (MFA) を要求する – AWS アカウントで IAM ユーザーまたはルートユーザーを要求するシナリオがある場合は、セキュリティを強化するために MFA をオンにします。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、IAM ユーザーガイド の MFA を使用した安全な API アクセス を参照してください。

IAM でのベストプラクティスの詳細については、IAM ユーザーガイド の IAM でのセキュリティのベストプラクティス を参照してください。

DynamoDB コンソールの使用

Amazon DynamoDB コンソールにアクセスするには、許可の最小限のセットが必要です。これらのアクセス許可により、AWS アカウント の DynamoDB リソースの詳細をリストおよび表示できます。最小限必要なアクセス許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) ではコンソールが意図したとおりに機能しません。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソール権限を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスを許可します。

ユーザーとロールが引き続き DynamoDB コンソールを使用できるようにするには、エンティティに DynamoDB の `ConsoleAccess` または `ReadOnly` AWS マネージドポリシーもアタッチします。詳細については、「IAM ユーザーガイド」の「ユーザーへのアクセス許可の追加」を参照してください。

自分の権限の表示をユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI か AWS API を使用してプログラム的に、このアクションを完了するアクセス許可が含まれています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon DynamoDB でのアイデンティティベースポリシーの使用

このトピックでは、Amazon DynamoDB でのアイデンティティベースの AWS Identity and Access Management (IAM) ポリシーの使用についての説明と、例を示します。これらの例は、アカウント管理者が IAM アイデンティティ (ユーザー、グループ、およびロール) に許可ポリシーをアタッチし、それによって Amazon DynamoDB リソースに対するオペレーションを実行するための許可を付与する方法を示しています。

このセクションでは、次のトピックを対象としています。

- Amazon DynamoDB コンソールの使用に必要な IAM 許可
- AWSAmazon DynamoDB の マネージド (事前定義) IAM ポリシー
- カスタマーマネージドポリシーの例

以下は、アクセス権限ポリシーの例です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DescribeQueryScanBooksTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:111122223333:table/Books"
        }
    ]
}
```

上記のポリシーには、us-west-2 AWS リージョンのテーブルで 3 つの DynamoDB アクション (dynamodb:DescribeTable、dynamodb:Query、dynamodb:Scan) を許可する 1 つのステートメントがあります。これは、*account-id* で指定される AWS アカウントで所有されていま

す。Resource 値の Amazon リソースネーム (ARN) では、アクセス許可が適用されるテーブルを指定します。

Amazon DynamoDB コンソールの使用に必要な IAM 許可

ユーザーが DynamoDB コンソールを使用するには、AWS アカウントで DynamoDB リソースの使用を許可する最小限の許可セットが必要です。これらの DynamoDB 許可に加えて、コンソールでは次の許可が必要になります。

- メトリクスとグラフを表示する Amazon CloudWatch 許可。
- AWS Data PipelineDynamoDB データをエクスポートおよびインポートする アクセス許可。
- AWS Identity and Access Managementエクスポートおよびインポートに必要なロールにアクセスする アクセス許可。
- CloudWatch アラームがトリガーされるたびに通知する Amazon Simple Notification Service 許可。
- AWS LambdaDynamoDB Streams レコードを処理するための 許可。

これらの最小限必要なアクセス許可よりも制限された IAM ポリシーを作成している場合、その IAM ポリシーを使用するユーザーに対してコンソールは意図したとおりには機能しません。ユーザーが引き続き DynamoDB コンソールを使用できるようにするには、「AWSAmazon DynamoDB の マネージド (事前定義) IAM ポリシー」で説明されているように、AmazonDynamoDBReadOnlyAccess AWS マネージドポリシーもユーザーにアタッチします。

AWS CLI または Amazon DynamoDB API のみを呼び出すユーザーには、最小限のコンソール許可を付与する必要はありません。

> **Note**
>
> VPC エンドポイントを参照する場合、IAM アクション (dynamodb:DescribeEndpoints) を使用して、リクエストしている IAM プリンシパルの DescribeEndpoints API コールを認可する必要もあります。詳細については、「エンドポイントに必要なポリシー」を参照してください。

AWSAmazon DynamoDB の マネージド (事前定義) IAM ポリシー

AWS は、 によって作成され管理されるスタンドアロンの IAM ポリシーが提供する多くの一般的ユースケースに対応します。AWSこれらの AWS 管理ポリシーは、一般的ユースケースに必要なア

クセス権限を付与することで、どの権限が必要なのかをユーザーが調査する必要をなくすことができます。詳細については、IAM ユーザーガイドの「AWS 管理ポリシー」を参照してください。

アカウントのユーザーにアタッチ可能な以下の AWS マネージドポリシーは、DynamoDB 固有のもので、ユースケースシナリオ別にグループ化されます。

- AmazonDynamoDBReadOnlyAccess – AWS マネジメントコンソール を介して DynamoDB リソース への読み込み専用アクセスを許可します。
- AmazonDynamoDBFullAccess – AWS マネジメントコンソール を介して DynamoDB リソースへのフルアクセスを許可します。

IAM コンソールにサインインし、特定のポリシーを検索することで、これらの AWS マネージド許可ポリシーを確認できます。

> **Important**
>
> ベストプラクティスは、ユーザー、ロール、またはそれらを必要とするグループに最小特権を付与するカスタム IAM ポリシーを作成することです。

カスタマーマネージドポリシーの例

このセクションでは、さまざまな DynamoDB アクションの許可を付与するポリシー例を示しています。これらのポリシーは、AWS SDK または AWS CLI を使用しているときに機能します。コンソールを使用している場合は、コンソールに固有の追加許可を付与する必要があります。詳細については、「Amazon DynamoDB コンソールの使用に必要な IAM 許可」を参照してください。

> **Note**
>
> 以下のすべてのポリシー例では、AWS リージョンの 1 つを使用しており、架空のアカウント ID とテーブル名が含まれています。

例:

- テーブル上のすべての DynamoDB アクションに許可を付与する IAM ポリシー
- DynamoDB テーブルの項目に読み込み専用許可を付与する IAM ポリシー
- 特定の DynamoDB テーブルとそのインデックスへのアクセスを付与する IAM ポリシー

- DynamoDB テーブルに対するアクセスの読み込み、書き込み、更新、削除を行う IAM ポリシー
- 同じ AWS アカウントで DynamoDB 環境を分離する IAM ポリシー
- DynamoDB リザーブドキャパシティの購入を防止する IAM ポリシー
- DynamoDB ストリームのみの読み込みアクセスを付与する IAM ポリシー (テーブルを除く)
- AWS Lambda 関数が DynamoDB ストリームレコードへのアクセスを許可する IAM ポリシー
- DynamoDB アクセラレーター (DAX) クラスターへの読み込みおよび書き込みアクセスに関する IAM ポリシー
- ベクトルインデックスを検索するためのアクセスを許可する IAM ポリシー

「IAM ユーザーガイド」には、追加の DynamoDB 例が 3 つ含まれています。

- Amazon DynamoDB: 特定のテーブルへのアクセスの許可
- Amazon DynamoDB: 特定の列へのアクセスの許可
- Amazon DynamoDB: Amazon Cognito ID に基づいて DynamoDB への行レベルのアクセスを許可する

テーブル上のすべての DynamoDB アクションに許可を付与する IAM ポリシー

以下のポリシーでは、Books というテーブルでのすべての DynamoDB アクションの許可を付与します。Resource で指定されたリソース ARN は、特定の AWS リージョンのテーブルを識別します。Resource ARN のテーブル名 Books をワイルドカード文字 (*) で置き換えると、アカウント内のすべてのテーブルですべての DynamoDB アクションが許可されます。このポリシーまたは IAM ポリシーでワイルドカード文字を使用する前に、考えられるセキュリティへの影響を慎重に検討してください。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
```

```
    ]
}
```

> **Note**
>
> これは、ワイルドカード文字 (*) を使用して、すべてのアクション (管理、データオペレーション、モニタリング、DynamoDB リザーブドキャパシティーの購入など) を許可する例です。代わりに、付与する各アクションと、そのユーザー、ロール、またはグループが必要とするアクションのみを明示的に指定することをお勧めします。

DynamoDB テーブルの項目に読み込み専用許可を付与する IAM ポリシー

次の許可ポリシーは、`GetItem`、`BatchGetItem`、`Scan`、`Query`、および `ConditionCheckItem` DynamoDB アクションのみに許可を付与し、その結果、`Books` テーブルへの読み込み専用アクセスを設定します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

特定の DynamoDB テーブルとそのインデックスへのアクセスを付与する IAM ポリシー

以下のポリシーでは、Books という DynamoDB テーブルと、そのテーブルのすべてのインデックスに対するデータ変更アクションの許可を付与します。インデックスの動作の詳細については、「DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上」を参照してください。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AccessTableAllIndexesOnBooks",
            "Effect": "Allow",
            "Action": [
              "dynamodb:PutItem",
              "dynamodb:UpdateItem",
              "dynamodb:DeleteItem",
              "dynamodb:BatchWriteItem",
              "dynamodb:GetItem",
              "dynamodb:BatchGetItem",
              "dynamodb:Scan",
              "dynamodb:Query",
              "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
            ]
        }
    ]
}
```

DynamoDB テーブルに対するアクセスの読み込み、書き込み、更新、削除を行う IAM ポリシー

アプリケーションが Amazon DynamoDB テーブル、インデックス、およびストリーミングのデータを作成、読み込み、更新、および削除できるようにする必要がある場合は、このポリシーを使用します。必要に応じて、AWS リージョン名、アカウント ID、 テーブル名またはワイルドカード文字 (*) に置き換えます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBIndexAndStreamAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetShardIterator",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/stream/*"
            ]
        },
        {
            "Sid": "DynamoDBTableAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem",
                "dynamodb:PutItem",
                "dynamodb:DescribeTable",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        },
        {
            "Sid": "DynamoDBDescribeLimitsAccess",
            "Effect": "Allow",
            "Action": "dynamodb:DescribeLimits",
            "Resource": [
```

```
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
            ]
        }
    ]
}
```

このポリシーを拡張して、このアカウントのすべての AWS リージョンのすべての DynamoDB テーブルをカバーするには、リージョンとテーブル名にワイルドカード (*) を使用します。以下に例を示します。

```
"Resource":[
                "arn:aws:dynamodb:*:123456789012:table/*",
                "arn:aws:dynamodb:*:123456789012:table/*/index/*"
                ]
```

同じ AWS アカウントで DynamoDB 環境を分離する IAM ポリシー

たとえば、各環境で、ProductCatalog という名前のテーブルを独自のバージョンで維持するとします。2 つの ProductCatalog テーブルを同じ AWS アカウントから作成する場合、許可の設定方法により、片方の環境での動作が他の環境に影響を与える可能性があります。例えば、同時制御プレーンオペレーション (CreateTable など) の数に対するクォータは AWS アカウントレベルで設定されます。

その結果、1 つの環境内の各アクションによって、もう一方の環境で利用可能なオペレーションの数が減少します。また、片方の環境内のコードが他の環境内のテーブルに偶然にアクセスしてしまう危険性もあります。

> **Note**
>
> 本番ワークロードとテストワークロードを分離して、イベントの潜在的な「爆発半径」を制御する場合、ベストプラクティスは、テストワークロードと本番ワークロード用に別々の AWS アカウントを作成することです。詳細については、AWS アカウントの管理および分離を参照してください。

さらに、Amit と Alice という 2 人のデベロッパーが ProductCatalog テーブルをテストしているとします。デベロッパーごとに別々の AWS アカウントを作成する代わりに、デベロッ

パー間で同じテスト AWS アカウントを共有することができます。このテスト用アカウントでは、Alice_ProductCatalog や Amit_ProductCatalog など、対象のデベロッパーごとに、同じテーブルのコピーを作成できます。この場合、テスト環境用に作成した AWS アカウント内で、ユーザーの Alice と Amit を作成できます。次に、所有するテーブルに対して DynamoDB アクションを実行できるように、これらのユーザーに許可を付与することができます。

これらの IAM ユーザーの許可を付与するには、次のいずれかを実行します。

- 各ユーザーに別々のポリシーを作成し、各ポリシーをユーザーごとに個別にアタッチします。たとえば、以下のポリシーをユーザーである Alice にアタッチすることで、Alice_ProductCatalog テーブルに対する DynamoDB アクションへのアクセスをすべて許可することができます。

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnAliceTable",
            "Effect": "Allow",
            "Action": [
              "dynamodb:DeleteItem",
              "dynamodb:DescribeContributorInsights",
              "dynamodb:RestoreTableToPointInTime",
              "dynamodb:ListTagsOfResource",
              "dynamodb:CreateTableReplica",
              "dynamodb:UpdateContributorInsights",
              "dynamodb:CreateBackup",
              "dynamodb:DeleteTable",
              "dynamodb:UpdateTableReplicaAutoScaling",
              "dynamodb:UpdateContinuousBackups",
              "dynamodb:TagResource",
              "dynamodb:DescribeTable",
              "dynamodb:GetItem",
              "dynamodb:DescribeContinuousBackups",
              "dynamodb:BatchGetItem",
              "dynamodb:UpdateTimeToLive",
              "dynamodb:BatchWriteItem",
              "dynamodb:ConditionCheckItem",
              "dynamodb:UntagResource",
              "dynamodb:PutItem",
              "dynamodb:Scan",
              "dynamodb:Query",
```

```
                "dynamodb:UpdateItem",
                "dynamodb:DeleteTableReplica",
                "dynamodb:DescribeTimeToLive",
                "dynamodb:RestoreTableFromBackup",
                "dynamodb:UpdateTable",
                "dynamodb:DescribeTableReplicaAutoScaling",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:DescribeLimits",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
Alice_ProductCatalog/*"
        }
    ]
}
```

次に、ユーザー Amit に対して、同様のポリシーを異なるリソース (Amit_ProductCatalog テーブル) で作成できます。

- 個々のユーザーにポリシーを付与する代わりに、IAM ポリシー変数を使用して単一のポリシーを記述し、グループに付与することができます。この例では、グループを作成し、そのグループにユーザー Alice とユーザー Amit の両者を追加する必要があります。次の例では、${aws:username}_ProductCatalog テーブルのすべての DynamoDB アクションを実行できる許可を付与します。ポリシーが評価されると、ポリシー変数 ${aws:username} はリクエスターのユーザー名に置き換えられます。たとえば、Alice が項目の追加要求を送信する場合、Alice が Alice_ProductCatalog テーブルにアイテムを追加する場合にのみ、そのアクションが許可されます。

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ActionsOnUserSpecificTable",
            "Effect": "Allow",
            "Action": [
              "dynamodb:PutItem",
              "dynamodb:UpdateItem",
              "dynamodb:DeleteItem",
```

```
                "dynamodb:BatchWriteItem",
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
${aws:username}_ProductCatalog"
        },
        {
            "Sid": "AdditionalPrivileges",
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListTables",
                "dynamodb:DescribeTable",
                "dynamodb:DescribeContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/*"
        }
    ]
}
```

> **Note**
>
> IAM ポリシー変数を使用する場合、ポリシーで明示的に IAM ポリシー言語の 2012-10-17 バージョンを指定する必要があります。IAM ポリシー言語のデフォルトバージョン (2008-10-17) では、ポリシー変数をサポートしていません。

通常のように具体的なテーブルをリソースとして特定する代わりに、ワイルドカード文字 (*) を使用して、すべてのテーブルに許可を付与することができます。次の例に示すように、テーブル名の先頭には、リクエストを行っているユーザーの名前が付きます。

```
"Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

DynamoDB リザーブドキャパシティの購入を防止する IAM ポリシー

Amazon DynamoDB リザーブドキャパシティーでは、1 回限りの前払い料金を支払い、期間中、大幅な節約ができる最小使用レベルを支払う契約を結びます。AWS マネジメントコンソール を使用し

て、リザーブドキャパシティーを表示し、購入できます。ただし、組織のすべてのユーザーが、リザーブドキャパシティーを購入できないようにしたい場合があります。リザーブドキャパシティーの詳細については、Amazon DynamoDB の料金を参照してください。

DynamoDB は、リザーブドキャパシティー管理へのアクセスを制御するために、次の API オペレーションを提供します。

- dynamodb:DescribeReservedCapacity – 現在有効なリザーブドキャパシティーの購入を返します。
- dynamodb:DescribeReservedCapacityOfferings – で現在提供されているリザーブドキャパシティープランについての詳細を返します。AWS
- dynamodb:PurchaseReservedCapacityOfferings – リザーブドキャパシティーの実際の注文を実行します。

AWS マネジメントコンソール はこれらの API アクションを使用してリザーブドキャパシティーの情報を表示し、購入を行います。アプリケーションプログラムからこれらのオペレーションを呼び出すことはできません。オペレーションにはコンソールからのみアクセスできるためです。ただし、IAM 許可ポリシーでこれらのオペレーションへのアクセスを許可または拒否することができます。

以下のポリシーでは、ユーザーは AWS マネジメントコンソール を使用して、リザーブドキャパシティの購入およびサービスを表示できますが、新規購入は拒否されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReservedCapacityDescriptions",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeReservedCapacity",
                "dynamodb:DescribeReservedCapacityOfferings"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        },
        {
            "Sid": "DenyReservedCapacityPurchases",
            "Effect": "Deny",
```

```
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        }
    ]
}
```

このポリシーでは、ワイルドカード文字 (*) を使用して、すべてのユーザーに説明許可を付与し、すべての DynamoDB リザーブドキャパシティーの購入を拒否することに注意してください。

DynamoDB ストリームのみの読み込みアクセスを付与する IAM ポリシー (テーブルを除く)

テーブルで DynamoDB Streams を有効にすると、テーブル内の項目に加えられたすべての変更に関する情報をキャプチャします。詳細については、「」を参照してくださいDynamoDB Streams の変更データキャプチャ

場合によっては、アプリケーションが DynamoDB テーブルからデータを読み取らないようにする一方で、そのテーブルのストリーミングへのアクセスは許可したいことがあります。たとえば、項目の更新が検出されたときにストリーミングをポーリングし、Lambda 関数をコールして、追加の処理を実行するよう AWS Lambda を設定できます。

DynamoDB Streams へのアクセスを制御するために、以下のアクションを使用できます。

- `dynamodb:DescribeStream`
- `dynamodb:GetRecords`
- `dynamodb:GetShardIterator`
- `dynamodb:ListStreams`

次のポリシー例では、`GameScores` という名前のテーブルでストリーミングにアクセスする許可をユーザーに付与します。ARN のワイルドカード文字 (*) は、そのテーブルに関連付けられた任意のストリーミング ID に一致します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AccessGameScoresStreamOnly",
```

```
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/*"
        }
    ]
}
```

このポリシーは GameScores テーブルのストリーミングへのアクセスを付与しますが、テーブル自体へのアクセスは付与しないことに注意してください。

AWS Lambda 関数が DynamoDB ストリームレコードへのアクセスを許可する IAM ポリシー

DynamoDB Streams のイベントに基づいて特定のアクションを実行する場合、それらのイベントによってトリガーされる AWS Lambda 関数を書き込むことができます。このような Lambda 関数には、DynamoDB Streams からデータを読み込む許可が必要です。DynamoDB Streams で Lambda を使用する詳細方法については、DynamoDB Streams と AWS Lambda のトリガー を参照してください。

Lambda に許可を付与するには、Lambda 関数の IAM ロール (実行ロールともいう) に関連付けられた許可ポリシーを使用します。Lambda 関数の作成時に、このポリシーを指定します。

たとえば、次の許可ポリシーを実行ロールに関連付けて、リストされた DynamoDB Streams アクションを実行する許可を Lambda に付与できます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "APIAccessForDynamoDBStreams",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetRecords",
```

```
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
GameScores/stream/*"
        }
    ]
}
```

詳細については、AWS Lambda デベロッパーガイドの AWS Lambda 許可を参照してください。

DynamoDB アクセラレーター (DAX) クラスターへの読み込みおよび書き込みアクセスに関する IAM ポリシー

次のポリシーは、DynamoDB アクセラレーター (DAX) クラスターへの読み込み、書き込み、更新、および削除アクセスを許可しますが、関連する DynamoDB テーブルへのアクセスは許可しません。このポリシーを使用するには、AWS リージョン名、アカウント ID、DAX クラスターの名前に置き換えます。

> **Note**
>
> このポリシーは、DAX クラスターへのアクセスを許可しますが、関連する DynamoDB テーブルへのアクセスは許可しません。DAX クラスターに、ユーザーに代わって DynamoDB テーブルでこれらの同じオペレーションを実行するための正しいポリシーがあることを確認してください。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonDynamoDBDAXDataOperations",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:PutItem",
```

```
                "dax:ConditionCheckItem",
                "dax:BatchGetItem",
                "dax:BatchWriteItem",
                "dax:DeleteItem",
                "dax:Query",
                "dax:UpdateItem",
                "dax:Scan"
            ],
            "Resource": "arn:aws:dax:eu-west-1:123456789012:cache/MyDAXCluster"
        }
    ]
}
```

このポリシーを拡張して、アカウントのすべての AWS リージョンの DAX アクセスをカバーするには、リージョン名にワイルドカード文字 (*) を使用します。

```
"Resource": "arn:aws:dax:*:123456789012:cache/MyDAXCluster"
```

ベクトルインデックスを検索するためのアクセスを許可する IAM ポリシー

次のポリシーは、特定のベクトルインデックスに対して SearchVectors オペレーションを実行するアクセス許可を付与します。ベクトルインデックスのリソース ARN は、他の DynamoDB インデックスと同じ形式を使用します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:SearchVectors",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Products/index/
ProductEmbeddingIndex"
        }
    ]
}
```

テーブル上のすべてのインデックスへの SearchVectors アクセスを許可するには、インデックス名にワイルドカード文字 (*) を使用します。

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Products/index/*"
```

追加のアクセス許可は必要ありません。

ベクトルインデックスを作成または削除するために、追加の IAM アクセス許可は必要ありません。既存の `dynamodb:CreateTable` および `dynamodb:UpdateTable` アクセス許可で十分です。

SearchVectors でサポートされていない FGAC 条件キー

`SearchVectors` API では、きめ細かなアクセスコントロール (FGAC) 条件キーはサポートされていません。`dynamodb:LeadingKeys`、`dynamodb:Attributes`、または `dynamodb:Select` 条件キーを使用して `SearchVectors` アクセスを制限することはできません。

Amazon DynamoDB アイデンティティとアクセスのトラブルシューティング

次の情報は、DynamoDB と IAM の使用に伴って発生する可能性がある一般的な問題の診断や解決に役立ちます。

トピック

- DynamoDB でアクションを実行する権限がない
- iam:PassRole を実行する権限がない
- 自分の AWS アカウント 以外のユーザーに DynamoDB リソースへのアクセスを許可したい

DynamoDB でアクションを実行する権限がない

AWS マネジメントコンソール から、アクションを実行することが認可されていないと通知された場合、管理者に問い合わせ、サポートを依頼する必要があります。担当の管理者はお客様のユーザー名とパスワードを発行した人です。

以下のエラー例は、`mateojackson` ユーザーがコンソールを使用して架空の *`my-example-widget`* リソースに関する詳細情報を表示しようとしているが、架空の `aws:GetWidget` 許可がないという場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 aws:GetWidget on resource: my-example-widget
```

この場合、Mateo は、aws:*GetWidget* アクションを使用して *my-example-widget* リソースにアクセスできるように、管理者にポリシーの更新を依頼します。

iam:PassRole を実行する権限がない

iam:PassRole アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して DynamoDB にロールを渡せるようにする必要があります。

一部の AWS のサービス では、新しいサービスロールやサービスリンクロールを作成せずに、既存のロールをサービスに渡すことができます。そのためには、サービスにロールを渡すアクセス許可が必要です。

以下の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して DynamoDB でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与されたアクセス許可が必要です。Mary には、ロールをサービスに渡すアクセス許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに iam:PassRole アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者に問い合わせてください。サインイン認証情報を提供した担当者が管理者です。

自分の AWS アカウント 以外のユーザーに DynamoDB リソースへのアクセスを許可したい

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください。

- DynamoDB でこれらの特徴がサポートされるかどうかを確認するには、「Amazon DynamoDB で IAM が機能する仕組み」を参照してください。

- 所有している AWS アカウント 全体のリソースへのアクセス権を提供する方法については、IAM ユーザーガイド の 所有している別の AWS アカウント へのアクセス権を IAM ユーザーに提供 を参照してください。
- サードパーティの AWS アカウント にリソースへのアクセス権を提供する方法については、「IAM ユーザーガイド」の「サードパーティが所有する AWS アカウント へのアクセス権を付与する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、IAM ユーザーガイド の 外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可 を参照してください。
- クロスアカウントアクセスでのロールとリソースベースのポリシーの使用の違いの詳細については、「IAM ユーザーガイド」の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。

DynamoDB リザーブドキャパシティの購入を防止する IAM ポリシー

Amazon DynamoDB リザーブドキャパシティーでは、1 回限りの前払い料金を支払い、期間中、大幅な節約ができる最小使用レベルを支払う契約を結びます。AWS マネジメントコンソール を使用して、リザーブドキャパシティーを表示し、購入できます。ただし、組織のすべてのユーザーが、リザーブドキャパシティーを購入できないようにしたい場合があります。リザーブドキャパシティーの詳細については、Amazon DynamoDB の料金を参照してください。

DynamoDB は、リザーブドキャパシティー管理へのアクセスを制御するために、次の API オペレーションを提供します。

- `dynamodb:DescribeReservedCapacity` – 現在有効なリザーブドキャパシティーの購入を返します。
- `dynamodb:DescribeReservedCapacityOfferings` – で現在提供されているリザーブドキャパシティープランについての詳細を返します。AWS
- `dynamodb:PurchaseReservedCapacityOfferings` – リザーブドキャパシティーの実際の注文を実行します。

AWS マネジメントコンソール はこれらの API アクションを使用してリザーブドキャパシティーの情報を表示し、購入を行います。アプリケーションプログラムからこれらのオペレーションを呼び出すことはできません。オペレーションにはコンソールからのみアクセスできるためです。ただし、IAM 許可ポリシーでこれらのオペレーションへのアクセスを許可または拒否することができます。

以下のポリシーでは、ユーザーは AWS マネジメントコンソール を使用して、リザーブドキャパシティの購入およびサービスを表示できますが、新規購入は拒否されます。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReservedCapacityDescriptions",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeReservedCapacity",
                "dynamodb:DescribeReservedCapacityOfferings"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        },
        {
            "Sid": "DenyReservedCapacityPurchases",
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:*"
        }
    ]
}
```

このポリシーでは、ワイルドカード文字 (*) を使用して、すべてのユーザーに説明許可を付与し、すべての DynamoDB リザーブドキャパシティーの購入を拒否することに注意してください。

詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用

DynamoDB で許可を付与するときは、許可ポリシーを有効にする方法を決める条件を指定できます。

概要

DynamoDB では、IAM ポリシー (Amazon DynamoDB の Identity and Access Management を参照) を使用して許可を付与するとき、条件を指定することもできます。たとえば、以下のことが可能です。

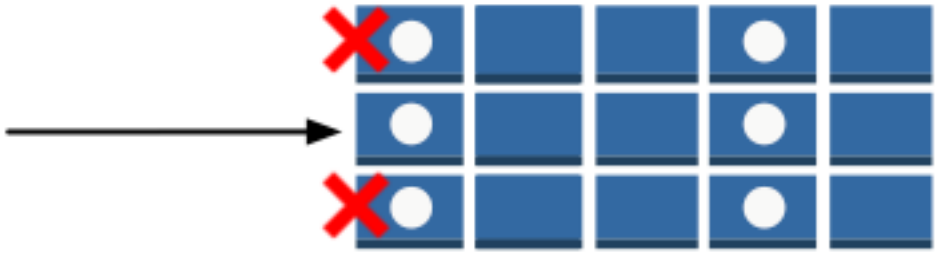
- テーブル内またはセカンダリインデックスの特定の項目と属性に対する読み込み専用アクセスをユーザーに許可する許可を付与します。
- ユーザーのアイデンティティに基づいて、テーブルの特定の属性への書き込み専用アクセスのアクセス権限をユーザーに付与します。

DynamoDB では、次のセクションのユースケースに示すように、条件キーを使用して IAM ポリシーの条件を指定できます。

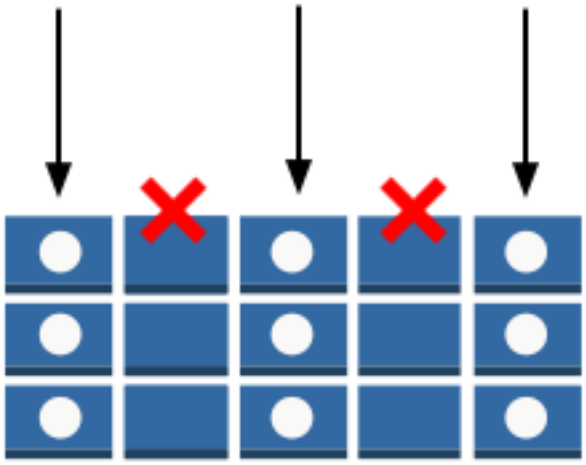
アクセス権限のユースケース

DynamoDB API アクションへのアクセスを制御するだけでなく、個別のデータ項目と属性へのアクセスも制御できます。例えば、次の操作を実行できます。

- テーブルでアクセス権限を付与しますが、特定のプライマリキー値に基づいて、そのテーブル内の特定の項目へのアクセスを制限します。この例として、ゲーム用のソーシャルネットワーキングアプリケーションがあります。次の図に示すように、すべてのユーザーのゲームデータは 1 つのテーブルに保存されますが、いずれのユーザーも、自分が所有していないデータ項目にアクセスすることはできません。



- 属性のサブセットのみがユーザー表示されるように、情報を非表示にします。この例として、ユーザーの位置に基づいて、近くの空港の運航便データを表示するアプリケーションがあります。航空会社名、到着時間、出発時間、および運航便番号がすべて表示されます。ただし、次の図に示すように、パイロット名や乗客数などの属性は非表示になります。



このような種類のきめ細かなアクセス制御を実装するには、セキュリティ認証情報と関連する許可にアクセスするための条件を指定した IAM 許可ポリシーを書き込みます。次に、IAM コンソールを使用して作成する ユーザー、グループ、またはロールにそのポリシーを適用します。IAM ポリシーによって、テーブルの個別項目へのアクセス、その項目の属性へのアクセス、またはその両方へのアクセスを同時に制御できます。

必要に応じて、ウェブアイデンティティフェデレーションを使用して、Login with Amazon、Facebook、または Google によって認証されるユーザーによるアクセスを制御できます。詳細については、「」を参照してくださいウェブ ID フェデレーションの使用

IAM の `Condition` 要素を使用して、詳細なアクセスコントロールポリシーを実装できます。`Condition` 要素を許可ポリシーに追加することで、特定のビジネス要件に基づいて、DynamoDB テーブルとインデックスの項目と属性へのアクセスを許可または拒否できます。

次の動画では、IAM ポリシー条件を使用した DynamoDB でのきめ細かなアクセスコントロールについて説明します。

DynamoDB でのきめ細かなアクセスコントロールを理解する

DynamoDB でのきめ細かなアクセスコントロールにより、複数のレベルで正確なアクセス許可の境界を作成できます。

1. 項目レベルのアクセスコントロール: 通常は、ID またはアクセス許可の範囲と一致する特定のキー値を含む項目のみにユーザーによるアクセスを制限します。

2. 属性レベルのアクセスコントロール: ユーザーが表示または変更できる属性 (列) を制限し、機密情報を保護しながら、同じ項目内の非機密データへのアクセスを許可します。
3. オペレーション固有のコントロール: 実行するオペレーションのタイプに基づいて異なるアクセス許可ルールを適用します。

これらのコントロールは、DynamoDB 固有の条件キーを使用して IAM ポリシーを通じて実装されます。

条件の指定: 条件キーの使用

AWS には、アクセス制御のために IAM をサポートするすべての AWS サービスに合わせて事前定義された一連の条件キー (AWS 全体に対する条件キー) が用意されています。たとえば、`aws:SourceIp` 条件キーを使用して、リクエスタの IP アドレスを確認してから、アクションの実行を許可できます。AWS 全体を対象とするキーの詳細およびリストについては、IAM ユーザーガイドの使用可能な条件キーを参照してください。

以下は、DynamoDB に適用される DynamoDB サービス固有の条件キーを示しています。

`dynamodb:LeadingKeys`

テーブルの最初のキー属性、つまりパーティションキーを表します。キーが 1 つの項目のアクションで使用される場合も、キー名 `LeadingKeys` は複数形になります。また、条件で `ForAllValues` を使用するときには、`LeadingKeys` 修飾子を使用する必要があります。

`dynamodb:Select`

リクエストの `Select` パラメータを表します。`Select` には次のいずれかの値を指定できます。

- `ALL_ATTRIBUTES`
- `ALL_PROJECTED_ATTRIBUTES`
- `SPECIFIC_ATTRIBUTES`
- `COUNT`

> **Note**
>
> 多くの場合、この条件キーはクエリおよびスキャンオペレーションに関連付けられますが、項目属性を返すすべての DynamoDB オペレーションに適用され、すべての API アクションで属性アクセスを制御するために不可欠です。この条件キーに

StringEqualsIfExists または同様の制約を使用すると、この条件キーが適用されるオペレーションに制約が適用され、適用されないオペレーションでは制約が無視されます。

dynamodb:Attributes

リクエストによってアクセスされる最上位属性のリストを表します。最上位属性は、それ自身かその入れ子の属性がリクエストパラメータで指定されている場合、リクエストによってアクセスされます。例えば、"Name, Address.City" の ProjectionExpression を指定する GetItem リクエストの場合、dynamodb:Attributes リストには「名前」と「住所」が含まれます。Attributes パラメータがきめ細かなアクセスコントロールポリシーで列挙されている場合は、GetItem、Query、Scan などの複数の API アクションで指定された属性へのアクセスが制限されるように、ReturnValues パラメータと Selectパラメータも制限することを検討してください。

> **Note**
>
> この条件は、レスポンスの属性ではなく、リクエストで指定された属性 (ProjectionExpression など) に対してのみ評価されます。リクエストに ProjectionExpression が指定されていない場合、ポリシーの属性制限に関係なく、すべての属性が返されます。属性アクセスを適切に保護する方法の詳細については、以下の「属性ベースの制限を確実に適用する」セクションを参照してください。

dynamodb:ReturnValues

リクエストの ReturnValues パラメータを表します。API アクションに応じて、ReturnValues は次のいずれかの値になります。

- ALL_OLD
- UPDATED_OLD
- ALL_NEW
- UPDATED_NEW
- NONE

dynamodb:ReturnConsumedCapacity

リクエストの ReturnConsumedCapacity パラメータを表します。ReturnConsumedCapacity には次のいずれかの値を指定できます。

- TOTAL
- NONE

dynamodb:FirstPartitionKeyValues

テーブルの最初のキー属性、つまり最初のパーティションキーを表します。キーが 1 つの項目のアクションで使用される場合も、キー名 FirstPartitionKeyValues は複数形になります。また、条件で FirstPartitionKeyValues を使用するときには、ForAllValues 修飾子を使用する必要があります。FirstPartitionKeyValues と LeadingKeys の使用は交換可能です。

dynamodb:SecondPartitionKeyValues

dynamodb:FirstPartitionKeyValues と同様です。リソースの 2 番目のパーティションキーを表します。キーが 1 つの項目のアクションで使用される場合も、キー名 SecondPartitionKeyValues は複数形になります。

dynamodb:ThirdPartitionKeyValues

dynamodb:FirstPartitionKeyValues と同様です。リソースの 3 番目のパーティションキーを表します。キーが 1 つの項目のアクションで使用される場合も、キー名 ThirdPartitionKeyValues は複数形になります。

dynamodb:FourthPartitionKeyValues

dynamodb:FirstPartitionKeyValues と同様です。リソースの 4 番目のパーティションキーを表します。キーが 1 つの項目のアクションで使用される場合も、キー名 FourthPartitionKeyValues は複数形になります。

属性ベースの制限を確実に適用する

属性ベースの条件を使用して特定の属性へのアクセスを制限する場合は、これらの条件がどのように評価されるかを理解することが重要です。

- レスポンスの属性ではなく、属性条件は、リクエストで指定された属性でのみ評価されます。
- ProjectionExpression を使用しない読み取りオペレーション (GetItem、Query、Scan など) の場合、ポリシーの属性制限に関係なく、すべての属性が返されます。この機密データの潜在的な漏洩を防ぐには、属性条件 (dynamodb:Attributes) と特定の属性を必要とする条件 (dynamodb:Select) の両方を実装する必要があります。
- 書き込みオペレーション (PutItem、UpdateItem、DeleteItem) の場合、ReturnValues パラメータは完全な項目を返すことができ、書き込みオペレーション自体がポリシーに準拠している場合でも、制限された属性が公開される可能性があります。この露出を防ぐには、ポリシーに属性条件

(dynamodb:Attributes) と ReturnValues (dynamodb:ReturnValues) の制限の両方を実装します。

ユーザーアクセスの制限

多くの IAM アクセス権限ポリシーでは、パーティションキーの値がユーザー識別子と一致するテーブルの項目へのアクセスのみをユーザーに許可します。たとえば、前述のゲームアプリケーションは、この方法でアクセスを制限し、ユーザーは自分のユーザー ID に関連付けらたゲームデータのみにアクセスできるようにします。IAM 置換変数 ${www.amazon.com:user_id}、${graph.facebook.com:id}、および ${accounts.google.com:sub} には、Login with Amazon、Facebook、および Google のユーザー識別子が格納されます。アプリケーションがこのようなアイデンティティプロバイダーのいずれかにログインし、この識別子を取得する方法については、「ウェブ ID フェデレーションの使用」を参照してください。

> **Important**
>
> グローバルテーブルのレプリケーションを制限する場合、きめ細かなアクセスコントロールはサポートされていません。グローバルテーブルのレプリケーションに使用される DynamoDB サービスプリンシパルまたはサービスにリンクされたロールにきめ細かなアクセスコントロールのポリシー条件を適用すると、グローバルテーブル内のレプリケーションが中断される可能性があります。

> **Note**
>
> 以下のセクションの各例では、Effect 句を Allow に設定し、許可されるアクション、リソース、およびパラメータのみを指定します。IAM ポリシーで明示的に指定されたものへのアクセスだけが許可されます。
> 場合によっては、拒否ベースとなるようにこのポリシーを書き直すことができます。つまり、Effect 句を Deny に設定して、ポリシーのすべての口ジックを逆にします。ただし、拒否ベースのポリシーを DynamoDB で使用しないようお勧めします。このようなポリシーは、許可ベースのポリシーと比べて、正しく記述するのが難しいためです。また、DynamoDB API への将来の変更 (または、既存の API 入力への変更) が原因で、拒否ベースのポリシーが機能しなくなる可能性があります。

ポリシー例: きめ細かなアクセスコントロールのための IAM ポリシー条件の使用

このセクションでは、DynamoDB テーブルとインデックスに対してきめ細かなアクセス制御を実装するための一部のポリシーについて説明します。

> **Note**
>
> すべての例で、us-west-2 リージョンを使用し、架空のアカウント ID を含めています。

例 1。属性制限付きの基本的なパーティションキーベースのアクセスコントロール

例として、プレーヤーがさまざまなゲームを選択してプレイできるモバイルゲームアプリケーションを考えてみます。このアプリケーションでは、GameScores という名前の DynamoDB テーブルを使用して、高スコアなどのユーザーデータを記録します。テーブルの各項目は、ユーザー ID とユーザーがプレイしたゲームの名前で一意に識別されます。GameScores テーブルには、パーティションキー (UserId) およびソートキー (GameTitle) で構成されているプライマリキーがあります。ユーザーは、そのユーザー ID に関連付けられたゲームデータだけにアクセスできます。ゲームをプレイするユーザーは、GameRole という名前の IAM ロールに属する必要があります。このロールには、セキュリティポリシーが添付されています。

このアプリケーションのユーザーアクセス権限を管理するには、次のようなアクセス権限ポリシーを記述できます。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAccessToOnlyItemsMatchingUserID",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
```

```
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ],
               "dynamodb:Attributes":[
                  "UserId",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "TopScore",
                  "TopScoreDateTime"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

GameScores テーブル (Resource 要素) の特定の DynamoDB アクション (Action 要素) に対する許可の付与に加えて、Condition 要素は、次のように許可を制限する DynamoDB に固有の次の条件キーを使用します。

- dynamodb:LeadingKeys – この条件キーにより、ユーザーはパーティションのキーバリューがユーザー ID に一致する項目にのみアクセスできます。この ID、${www.amazon.com:user_id} は、置換変数です。置換変数の詳細については、「ウェブ ID フェデレーションの使用」を参照してください。
- dynamodb:Attributes – この条件キーは指定された属性へのアクセスを制限し、許可ポリシーにリストされたアクションのみが、これらの属性の値を返すことができるようにします。さらに、StringEqualsIfExists 句は、アプリケーションが常に対象となる特定の属性のリストを提供し、すべての属性をリクエストできないようにします。

IAM ポリシーが評価されると、結果は必ず true (アクセス許可) または false (アクセス拒否) になります。Condition 要素のいずれかの部分が false の場合、ポリシー全体が false と評価され、アクセスは拒否されます。

> **Important**
>
> dynamodb:Attributes を使用する場合、ポリシーでリストに記載されているテーブルとすべてのセカンダリインデックスについて、すべてのプライマリキー属性とインデックスキー属性の名前を指定する必要があります。指定しなかった場合、DynamoDB は、このキー属性を使用して、リクエストされたアクションを実行できません。

IAM ポリシードキュメントには、次の Unicode 文字のみを含めることができます。水平タブ (U+0009)、ラインフィード (U+000A)、キャリッジリターン (U+000D)、および U+0020 ~ U+00FF の範囲内の文字。

例 2: 特定のパーティションのキー値を持つ項目へのアクセスを制限するアクセス許可の付与

以下の許可ポリシーは、GamesScore テーブルで一連の DynamoDB アクションに対する許可を付与します。このポリシーは dynamodb:LeadingKeys 条件キーを使用して、UserID パーティションキーの値が、このアプリケーション用の Login with Amazon の一意のユーザー ID と一致する項目のみでユーザーアクションを制限します。

> **Important**
>
> Scan 用のアクセス権限は、アクションのリストに含まれていません。これは、Scan が主要なキーにかかわらずすべての項目を返すためです。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"FullAccessToUserItems",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
```

```
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ]
            }
         }
      }
   ]
}
```

> **Note**
>
> ポリシー変数を使用する場合は、ポリシーでバージョン 2012-10-17 を明示的に指定する必要があります。アクセスポリシー言語のデフォルトバージョンは 2008-10-17 です。このバージョンでは、ポリシー変数をサポートしていません。

読み取りアクセスを実装するには、データを変更可能なアクションを削除します。次のポリシーでは、読み込み専用アクセスを提供するアクションだけが条件に追加されています。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ReadOnlyAccessToUserItems",
         "Effect":"Allow",
         "Action":[
```

```
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ]
            }
         }
      }
   ]
}
```

> **Important**
>
> dynamodb:Attributes を使用する場合、ポリシーでリストに記載されているテーブルとあらゆるセカンダリインデックスについて、すべてのプライマリキー属性とインデックスキー属性の名前を指定する必要があります。指定しなかった場合、DynamoDB は、このキー属性を使用して、リクエストされたアクションを実行できません。

例 3: テーブルの特定の属性へのアクセスを制限するアクセス許可の付与

次のアクセス権限ポリシーは、dynamodb:Attributes 条件キーを追加して、テーブルの 2 つの特定の属性のみへアクセスを許可します。この属性に対して、読み込み、書き込み、または条件付き書き込みまたはスキャンフィルタでの評価を実行できます。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"LimitAccessToSpecificAttributes",
         "Effect":"Allow",
```

```
         "Action":[
            "dynamodb:UpdateItem",
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:BatchGetItem",
            "dynamodb:Scan"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:Attributes":[
                  "UserId",
                  "TopScore"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
               "dynamodb:ReturnValues":[
                  "NONE",
                  "UPDATED_OLD",
                  "UPDATED_NEW"
               ]
            }
         }
      }
   ]
}
```

> **Note**
>
> このポリシーでは、許可リスト手法を使用し、列挙された一連の属性へのアクセスを許可します。代わりに、他の属性へのアクセスを拒否する同等のポリシーを記述することもできますが、この拒否リスト手法はお勧めしません。ユーザーは、ウィキペディア (http://en.wikipedia.org/wiki/Principle_of_least_privilege) で説明されている最小権限の原則に従って、これらの拒否属性の名前を判別し、拒否属性を指定する代わりに、許可リストアプローチを使用して、許可されたすべての値を列挙することができます。

このポリシーでは、PutItem、DeleteItem、または BatchWriteItem は許可されません。これらのアクションでは常に前の項目全体が置き換えられ、アクセスが許可されていない属性の以前の値を、ユーザーが削除できる可能性があるためです。

アクセス権限ポリシーの StringEqualsIfExists 句により、以下を実施することができます。

- ユーザーが Select パラメータを指定する場合、その値は SPECIFIC_ATTRIBUTES である必要があります。この要件により、インデックスの射影からなど、API アクションが許可されていない属性を返さないようにします。
- ユーザーが ReturnValues パラメータを指定する場合、その値は、NONE、UPDATED_OLD、または UPDATED_NEW にする必要があります。これが必要なのは、UpdateItem アクションでは、置換する前に項目が存在するかどうかをチェックするために、および、リクエストされた場合に前の属性値を返すことができるように、暗黙的な読み込みオペレーションが実行されるためです。ReturnValues をこのように制限することで、確実にユーザーが許可された属性だけを読み込むまたは書き込むことができるようにします。
- StringEqualsIfExists 節により、許可されたアクションのコンテキストで、リクエストごとに Select または ReturnValues パラメータのいずれか 1 つだけを使用できるようにすることができます。

次に、このポリシーのバリエーションをいくつか示します。

- 読み込みアクションだけを許可するには、許可されるアクションのリストから UpdateItem を削除できます。残りのどのアクションも ReturnValues を受け付けないので、条件から ReturnValues を削除できます。また、StringEqualsIfExists パラメーターには必ず値 (特に指定のない限り、StringEquals) があるので、Select を ALL_ATTRIBUTES に変更できます。
- 書き込みアクションだけを許可するには、許可されるアクションのリストから、UpdateItem を除くすべてのアクションを削除します。UpdateItem では Select パラメーターを使用しないので、条件から Select を削除できます。また、StringEqualsIfExists パラメーターには必ず値 (特に指定のない限り、StringEquals) があるので、ReturnValues を NONE に変更する必要があります。
- 名前がパターンに一致するすべての属性を許可するには、StringLike ではなく StringEquals を使用し、複数文字パターン照合ワイルドカード文字 (*) を使用します。

例 4: 特定の属性で更新を回避するアクセス許可の付与

以下のアクセス権限のポリシーでは、dynamodb:Attributes 条件キーによって識別される特定の属性のみを更新するようユーザーアクセスを制限します。StringNotLike 条件によって、アプリケーションが、dynamodb:Attributes 条件キーを使用して指定された属性を更新できないようになります。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"PreventUpdatesOnCertainAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:UpdateItem"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
         "Condition":{
            "ForAllValues:StringNotLike":{
               "dynamodb:Attributes":[
                  "FreeGamesAvailable",
                  "BossLevelUnlocked"
               ]
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
               "dynamodb:ReturnValues":[
                  "NONE",
                  "UPDATED_OLD",
                  "UPDATED_NEW"
               ]
            }
         }
      }
   ]
}
```

次の点に注意してください。

- UpdateItem アクションは、他の書き込みアクションと同じように、更新前の値と更新後の値を返すことができるように、項目への読み取りアクセスを必要とします。ポリシーでは、dynamodb:ReturnValues 条件キーを指定して、更新が許可される属性のみにアクセスするようアクションを制限します。条件キーは、リクエストの ReturnValues を制限して NONE、UPDATED_OLD、または UPDATED_NEW のみを指定し、ALL_OLD または ALL_NEW は含まれません。
- StringEqualsIfExists 演算子は、リクエストに dynamodb:Select または dynamodb:ReturnValues が存在する場合、指定された値と一致することを保証します。これにより、オペレーションが完全な項目を返すのを防ぐことができます。
- 属性の更新を制限するときは、どのデータを返すことができるかを制御して、保護された属性の情報開示を防ぐ必要があります。
- PutItem および DeleteItem アクションは項目全体を置き換えるため、アプリケーションは任意の属性を変更することができます。したがって、特定の属性のみを更新するようアプリケーションを制限するときは、それらの API 用のアクセス権限を付与しないようにします。

例 5: インデックスで射影された属性のみのクエリを実行するアクセス許可の付与

以下の許可ポリシーでは、dynamodb:Attributes 条件キーを使用して、セカンダリインデックス (TopScoreDateTimeIndex) でクエリを許可します。また、このポリシーでは、インデックスに射影された特定の属性だけをリクエストするようクエリを制限します。

アプリケーションがクエリで属性のリストを指定するよう要求するには、ポリシーで dynamodb:Select 条件キーを指定して、DynamoDB Query アクションの Select パラメータが SPECIFIC_ATTRIBUTES であることを要求します。属性のリストは、dynamodb:Attributes 条件キーを使用して提供される特定のリストに制限されます。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"QueryOnlyProjectedIndexAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:Query"
         ],
         "Resource":[
```

```
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:Attributes":[
                  "TopScoreDateTime",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "Attempts"
               ]
            },
            "StringEquals":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

次のアクセス権限ポリシーは、似ていますが、クエリでは、インデックスに射影されたすべての属性をリクエストする必要があります。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"QueryAllIndexAttributes",
         "Effect":"Allow",
         "Action":[
            "dynamodb:Query"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "StringEquals":{
```

```
                "dynamodb:Select":"ALL_PROJECTED_ATTRIBUTES"
            }
         }
      }
   ]
}
```

例 6: 特定の属性とパーティションキー値へのアクセスを制限するアクセス許可の付与

以下の許可ポリシーでは、特定の DynamoDB アクション (Action 要素で指定) をテーブルとテーブルインデックス (Resource 要素で指定) で許可します。このポリシーは、dynamodb:LeadingKeys 条件キーを使用して、パーティションキーの値がユーザーの Facebook ID に一致する項目のみにアクセス許可を制限します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"LimitAccessToCertainAttributesAndKeyValues",
         "Effect":"Allow",
         "Action":[
            "dynamodb:UpdateItem",
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:BatchGetItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${graph.facebook.com:id}"
               ],
               "dynamodb:Attributes":[
                  "attribute-A",
                  "attribute-B"
```

```
                ]
            },
            "StringEqualsIfExists":{
                "dynamodb:Select":"SPECIFIC_ATTRIBUTES",
                "dynamodb:ReturnValues":[
                    "NONE",
                    "UPDATED_OLD",
                    "UPDATED_NEW"
                ]
            }
        }
    }
  ]
}
```

次の点に注意してください。

- ポリシー (UpdateItem) によって許可された書き込みアクションでは、attribute-A または attribute-B のみを変更できます。
- ポリシーでは UpdateItem が許可されるため、アプリケーションは新しい項目を挿入でき、その非表示の属性は、新しい項目では null として表示されます。この属性が TopScoreDateTimeIndex に射影されている場合、テーブルからのフェッチを引き起こすクエリを防止するという利点がポリシーに追加されます。
- アプリケーションは、dynamodb:Attributes に列挙されている属性以外の属性を読み込むことができません。このポリシーにより、アプリケーションは、読み込みリクエストで Select パラメータを SPECIFIC_ATTRIBUTES に設定する必要があり、許可リストの属性だけをリクエストすることができます。書き込みリクエストの場合、アプリケーションは、ReturnValues を ALL_OLD または ALL_NEW に設定できません。また、他の属性に基づく条件付き書き込みオペレーションを実行できません。

例 7: テーブルの特定の属性へのアクセスを制限するアクセス許可の拒否

次のポリシーは、機密属性へのアクセスを拒否し、射影式を省略してこの制限を回避できないようにします。これにより、CustomerData テーブルへの一般的なアクセスを許可しますが、SSN および CreditCardNumber 属性へのアクセスは明示的に拒否します。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData"
      },
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData",
         "Condition":{
            "ForAnyValue:StringEquals":{
               "dynamodb:Attributes":[
                  "SSN",
                  "CreditCardNumber"
               ]
            }
         }
      },
      {
         "Effect":"Deny",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:Query",
            "dynamodb:Scan"
         ],
         "Resource":"arn:aws:dynamodb:us-west-2:123456789012:table/CustomerData",
         "Condition":{
            "StringNotEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
```

```
        }
      }
   ]
}
```

関連トピック

- Amazon DynamoDB の Identity and Access Management
- DynamoDB API の許可: アクション、リソース、条件リファレンス

ウェブ ID フェデレーションの使用

多数のユーザーを対象とするアプリケーションを記述している場合、必要に応じて、ウェブ ID フェデレーションを使用して、認証と認可を実行できます。ウェブ ID フェデレーションは、個々の ユーザーを作成する必要性を排除します。代わりに、ユーザーは ID プロバイダーにサインインした後、一時的なセキュリティ資格情報を AWS Security Token Service (AWS STS) から取得できます。アプリケーションでは、この認証情報を使用して AWS サービスにアクセスできます。

ウェブアイデンティティフェデレーションでは、次のアイデンティティプロバイダーをサポートしています。

- Login with Amazon
- Facebook
- Google

ウェブアイデンティティフェデレーションに関するその他のリソース

ウェブアイデンティティフェデレーションの詳細を理解するのに、次のリソースが役に立ちます。

- AWS デベロッパーブログの「Web Identity Federation using the AWS SDK for .NET 」の記事では、ウェブ ID フェデレーションを Facebookで使用する方法について説明しています。記事には、ウェブ ID を持つ ロールを想定する方法や、一時的なセキュリティ認証情報を使用して AWS リソースにアクセスする方法を示す C# のコードス二ペットが含まれています。
- AWS Mobile SDK for iOS と AWS Mobile SDK for Android には、サンプルアプリが含まれています。このアプリケーションには、アイデンティティプロバイダーを呼び出す方法、そのプロバイダーからの情報を使用して、一時的なセキュリティ認証情報を取得および使用する方法を示すコードが含まれています。

- 記事「Web Identity Federation with Mobile Applications」では、ウェブ ID フェデレーションについて説明し、ウェブ ID フェデレーションを使用して AWS リソースにアクセスする使用方法の例を挙げています。

ウェブ ID フェデレーションのポリシー例

DynamoDB でウェブ ID フェデレーションを使用する方法を示すには、詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用 で導入された GameScores テーブルに再度アクセスしてください。次に、GameScores のプライマリキーを示します。

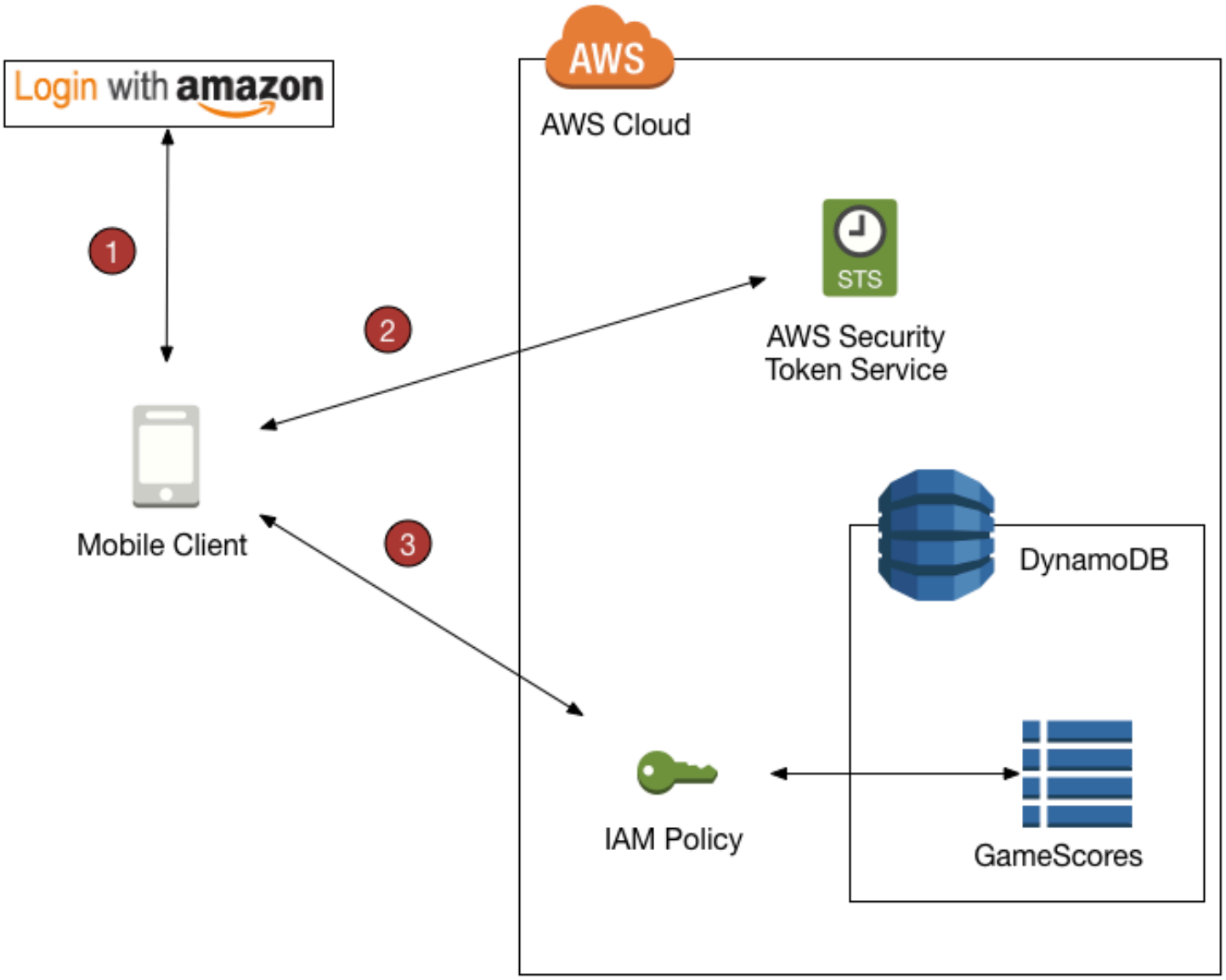
テーブル名	プライマリキーのタイプ	パーティションキーの名前と型	ソートキーの名前と型
GameScores (UserId、GameTitle、...)	複合	属性名: UserId 型: 文字列	属性名: GameTitle 型: 文字列

ここで、あるモバイルゲームアプリケーションがこのテーブルを使用し、そのアプリケーションでは、数千人、数百万人のユーザーをサポートする必要があるとします。この規模では、個別のアプリケーションユーザーを管理し、各ユーザーが GameScores テーブルの自分のデータにだけアクセスできることを確実にするのは非常に難しいです。多くのユーザーは、Facebook、Google、Login with Amazon などのサードパーティーの ID プロバイダーのアカウントを既に持っています。そのため、このようなプロバイダーのいずれかを利用して、認証タスクを行うことができます。

ウェブ ID フェデレーションを使用して認証を行うには、アプリケーションデベロッパーは、そのアプリケーションをアイデンティティプロバイダー (Login with Amazon など) に登録して、一意のアプリケーション ID を取得する必要があります。次に、デベロッパーはロールを作成する必要があります (この例では、このロールには GameRole という名前が付いています)。このロールには、IAM ポリシードキュメントを添付する必要があります。このドキュメントでは、アプリケーションが GameScores テーブルにアクセスできるように条件を指定します。

ユーザーは、ゲームをプレイするとき、ゲームアプリケーション内から Login with Amazon アカウントにサインインします。その後アプリケーションが AWS Security Token Service (AWS STS) を呼び出します。その際には Login with Amazon アプリケーション ID を指定し、GameRole でメンバーシップをリクエストします。AWS STS は一時的な AWS 認証情報をアプリケーションに返し、GameRole ポリシードキュメントに基づいて GameScores テーブルへのアクセスを許可します。

次の図は、これらの要素を組み合わせるとどのようになるかを示しています。



ウェブ ID フェデレーションの概要

1. アプリケーションは、サードパーティのアイデンティティプロバイダーを呼び出して、ユーザーとアプリケーションを認証します。アイデンティティプロバイダーは、ウェブアイデンティティトークンをアプリケーションに返します。
2. アプリは AWS STS を呼び出し、ウェブアイデンティティトークンを入力として渡します。AWS STS は、アプリケーションを認可し、一時的な AWS アクセス認証情報をアプリケーションに渡します。アプリケーションは、IAM ロール (GameRole) を想定し、そのロールのセキュリティポリシーに従って AWS リソースにアクセスすることが許可されます。

3. アプリケーションは、DynamoDB をコールして、GameScores テーブルにアクセスします。アプリケーションは GameRole を想定しているので、そのロールに関連付けられたセキュリティポリシーの影響を受けます。ポリシードキュメントによって、ユーザーに属していないデータにアプリケーションはアクセスできません。

繰り返しになりますが、詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用に示されている GameRole のセキュリティポリシーは次のとおりです。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAccessToOnlyItemsMatchingUserID",
         "Effect":"Allow",
         "Action":[
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem"
         ],
         "Resource":[
            "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
         ],
         "Condition":{
            "ForAllValues:StringEquals":{
               "dynamodb:LeadingKeys":[
                  "${www.amazon.com:user_id}"
               ],
               "dynamodb:Attributes":[
                  "UserId",
                  "GameTitle",
                  "Wins",
                  "Losses",
                  "TopScore",
                  "TopScoreDateTime"
               ]
```

```
            },
            "StringEqualsIfExists":{
               "dynamodb:Select":"SPECIFIC_ATTRIBUTES"
            }
         }
      }
   ]
}
```

Condition 節によって、アプリケーションに表示される GameScores の項目が決まります。Login with Amazon ID と UserId の GameScores パーティションキーの値を比較することで、これを行います。このポリシーに列挙されている DynamoDB アクションのいずれかを使用して処理できるのは、現在のユーザーに属する項目だけです。テーブルのその他の項目にはアクセスできません。さらに、ポリシーに列挙されている特定の属性にだけアクセスできます。

ウェブ ID フェデレーションを使用するための準備

アプリケーションデベロッパーがアプリケーションにウェブ ID フェデレーションを使用する場合、次のステップに従います。

1. デベロッパーとしてサードパーティーの ID プロバイダーにサインアップします。次の外部リンクには、サポートされる ID プロバイダーへのサインアップに関する情報が記載されています。
   - Login with Amazon デベロッパーセンター
   - Facebook サイトの登録
   - Google サイトの OAuth 2.0 を使用して Google API にアクセスする
2. アプリケーションを ID プロバイダーに登録します。登録すると、プロバイダーからアプリケーションに固有の ID が提供されます。複数のアイデンティティプロバイダーで機能するようにアプリケーションを構築する場合は、各プロバイダーからアプリケーション ID を取得する必要があります。
3. 1 つ以上の IAM ロールを作成します。各アプリケーションのアイデンティティプロバイダーごとに 1 つのロールが必要です。たとえば、ユーザーが Login with Amazon を使用してサインインするアプリケーションで想定できるロール、ユーザーが Facebook を使用してサインインする同じアプリケーションの 2 つ目のロール、およびユーザーが Google を使用してサインインするアプリケーションの 3 つ目のロールを作成します。

   ロール作成プロセスの中で、 ポリシーをロールにアタッチする必要があります。ポリシードキュメントでは、アプリケーションに必要な DynamoDB リソース、およびそれらのリソースにアクセスするための許可を定義する必要があります。

詳細については、IAM ユーザーガイドのウェブ ID フェデレーションについてを参照してください。

> **Note**
>
> AWS Security Token Service の代わりに、Amazon Cognito を使用できます。Amazon Cognito は、モバイルアプリケーションの一時的な認証情報を管理するためのおすすめサービスです。詳細については、Amazon Cognito 開発者ガイド」の「認証情報の取得」を参照してください。

DynamoDB コンソールを使用して IAM ポリシーを生成する

DynamoDB コンソールでは、ウェブ ID フェデレーションで使用する IAM ポリシーを作成できます。作成するには、DynamoDB テーブルを選択し、ポリシーに含まれるアイデンティティプロバイダー、アクション、および属性を指定します。DynamoDB コンソールによって、IAM ロールにアタッチすることができるポリシーが生成されます。

1. AWS マネジメントコンソール マネジメントコンソールにサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. ナビゲーションペインで [テーブル] を選択します。
3. テーブルの一覧で、IAM ポリシーを作成するテーブルを選択します。
4. [アクション] ボタン、[アクセス制御ポリシーの作成] の順に選択します。
5. ポリシーの ID プロバイダー、アクション、および属性を選択します。

   すべての設定が正しいことを確認したら、[ポリシーの生成] を選択します。生成されたポリシーが表示されます。
6. [ドキュメンテーションを参照] を選択し、生成されたポリシーを IAM ロールにアタッチするために必要な手順に従います。

ウェブ ID フェデレーションを使用するアプリケーションの記述

ウェブ ID フェデレーションを使用するには、アプリケーションで、作成した IAM ロールを引き受ける必要があります。その時点から、アプリケーションは、そのロールに関連付けられたアクセスポリシーに従います。

実行時、アプリケーションがウェブアイデンティティフェデレーションを使用する場合、次の手順に従う必要があります。

1. サードパーティーの ID プロバイダーと認証を行います。アプリケーションは、ID プロバイダーが提供するインターフェイスを使用してそのプロバイダーを呼び出す必要があります。ユーザーを認証するときの正確な方法は、プロバイダーおよびアプリケーションを実行しているプラットフォームによって異なります。一般的に、ユーザーがまだサインインしていない場合、アイデンティティプロバイダーはそのプロバイダーのサインインページを表示するように対処します。

   アイデンティティプロバイダーはユーザーを認証した後、ウェブアイデンティティトークンをアプリケーションに返します。このトークンの形式は、プロバイダーによって異なりますが、通常は、非常に長い文字列です。

2. 一時的な AWS セキュリティ認証情報を取得します。取得するために、アプリケーションは `AssumeRoleWithWebIdentity` リクエストを AWS Security Token Service (AWS STS) に送信します。このリクエストには次のものが含まれます。

   - 前の手順からのウェブアイデンティティトークン
   - アイデンティティプロバイダーからのアプリケーション ID
   - このアプリケーションのこのアイデンティティプロバイダー用に作成した IAM ロールの Amazon リソースネーム (ARN)

   AWS STS は、一定の時間 (デフォルトでは、3,600 秒) 経過した後に失効する AWS セキュリティ認証情報のセットを返します。

   次に、`AssumeRoleWithWebIdentity` で AWS STS アクションを実行するときのサンプルリクエストとレスポンスを示します。ウェブアイデンティティトークンは、Login with Amazon アイデンティティプロバイダーから取得しました。

```
GET / HTTP/1.1
Host: sts.amazonaws.com
Content-Type: application/json; charset=utf-8
URL: https://sts.amazonaws.com/?ProviderId=www.amazon.com
&DurationSeconds=900&Action=AssumeRoleWithWebIdentity
&Version=2011-06-15&RoleSessionName=web-identity-federation
&RoleArn=arn:aws:iam::123456789012:role/GameRole
&WebIdentityToken=Atza|IQEBLjAsAhQluyKqyBiYZ8-kclvGTYM81e...(remaining characters
 omitted)
```

```
<AssumeRoleWithWebIdentityResponse
  xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <AssumeRoleWithWebIdentityResult>
```

```
    <SubjectFromWebIdentityToken>amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE</
SubjectFromWebIdentityToken>
    <Credentials>
      <SessionToken>AQoDYXdzEMf//////////wEa8AP6nNDwcSLnf+cHupC...(remaining
 characters omitted)</SessionToken>
      <SecretAccessKey>8Jhi60+EWUUbbUShTEsjTxqQtM8UKvsM6XAjdA==</SecretAccessKey>
      <Expiration>2013-10-01T22:14:35Z</Expiration>
      <AccessKeyId>06198791C436IEXAMPLE</AccessKeyId>
    </Credentials>
    <AssumedRoleUser>
      <Arn>arn:aws:sts::123456789012:assumed-role/GameRole/web-identity-federation</
Arn>
      <AssumedRoleId>AROAJU4SA2VW5SZRF2YMG:web-identity-federation</AssumedRoleId>
    </AssumedRoleUser>
  </AssumeRoleWithWebIdentityResult>
  <ResponseMetadata>
    <RequestId>c265ac8e-2ae4-11e3-8775-6969323a932d</RequestId>
  </ResponseMetadata>
</AssumeRoleWithWebIdentityResponse>
```

3. AWSリソースにアクセスする AWS STS からのレスポンスには、DynamoDB リソースにアクセスするためにアプリケーションに必要な情報が格納されています。

   - AccessKeyID、SecretAccessKey、および SessionToken フィールドには、このユーザーとこのアプリケーションにのみ有効なセキュリティ認証情報が入っています。
   - Expiration フィールドは、この認証情報の時間制限を示します。この時間を経過すると、認証情報は無効になります。
   - AssumedRoleId フィールドには、アプリケーションによって想定されたセッション固有の IAM ロールの名前が含まれています。アプリケーションは、このセッションの期間中、IAM ポリシードキュメントのアクセス制御に従います。
   - SubjectFromWebIdentityToken フィールドには、この特定のアイデンティティプロバイダー用の IAM ポリシー変数に表示される一意の ID が含まれています。次に、サポートされるプロバイダーの IAM ポリシー変数と、その値の例を示します。

ポリシー変数	値の例
${www.amazon.com:user_id}	amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE
${graph.facebook.com:id}	123456789

ポリシー変数	値の例
`${accounts.google.com:sub}`	`123456789012345678901`

このポリシー変数が使用される IAM ポリシーの例については、ポリシー例: きめ細かなアクセスコントロールのための IAM ポリシー条件の使用 を参照してください。

AWS STS が一時的なアクセス認証情報を生成する詳しい方法については、IAM ユーザーガイドの一時的なセキュリティ認証情報のリクエストを参照してください。

DynamoDB API の許可: アクション、リソース、条件リファレンス

Amazon DynamoDB の Identity and Access Management を設定し、IAM アイデンティティにアタッチできる許可ポリシー (アイデンティティベースのポリシー) を書き込む場合は、リファレンスとして IAM ユーザーガイドの Amazon DynamoDB のアクション、リソース、および条件キーリストを使用できます。このページは、各 DynamoDB API オペレーション、アクションを実行するための許可を付与できる対応するアクション、および許可を付与できる AWS リソースを示しています。ポリシーの `Action` フィールドでアクションを指定し、ポリシーの `Resource` フィールドでリソースの値を指定します。

DynamoDB ポリシーで AWS 全体の条件キーを使用して、条件を表現することができます。AWS 全体を対象とするすべてのキーのリストについては、IAM ユーザーガイドの IAM JSON ポリシーエレメントリファレンスを参照してください。

AWS 全体を対象とする条件キーに加え、DynamoDB には条件内で使用可能な独自の固有キーもあります。詳細については、「」を参照してください詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用

関連トピック

- Amazon DynamoDB の Identity and Access Management
- 詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用

DynamoDB の業界別のコンプライアンス検証

新しい AWS サインアップエクスペリエンスは、規制対象ワークロード向けには設計されていません。新しい AWS サインアップエクスペリエンスを使用しているものの、規制対象ワークロードに

AWS を使用する場合は、AWS (高度) にサインアップしたり、AWS 環境の高度な機能をアクティブ化したりできます。

AWS のサービスが特定のコンプライアンスプログラムの対象であるかどうかを確認するには、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」を参照して、関心のあるコンプライアンスプログラムを選択してください。一般的な情報については、AWSコンプライアンスプログラム を参照してください。

AWS Artifact を使用して、サードパーティの監査レポートをダウンロードできます。詳細については、「AWS Artifact でレポートをダウンロードする」「」を参照してください。

AWS のサービスを使用する際のお客様のコンプライアンス責任は、お客様のデータの機密性や貴社のコンプライアンス目的、適用可能な法律および規制によって決定されます。AWS のサービス を使用する際のコンプライアンス責任の詳細については、「AWS セキュリティドキュメント」を参照してください。

Amazon DynamoDB での耐障害性と災害対策

AWS のグローバルインフラストラクチャは AWS リージョンとアベイラビリティーゾーンを中心として構築されます。AWSリージョンには、低レイテンシー、高いスループット、そして高度の冗長ネットワークで接続されている複数の物理的に独立し隔離されたアベイラビリティーゾーンがあります。アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS リージョンとアベイラビリティーゾーンの詳細については、「AWS グローバルインフラストラクチャ」を参照してください。

Amazon DynamoDB は、リージョン内の 3 つのアベイラビリティーゾーンにデータを自動的にレプリケートし、組み込みの高い耐久性と 99.99% の可用性 SLA を提供します。さらに、DynamoDB は、データの耐障害性とバックアップのニーズに対応できるように複数の機能を提供しています。

オンデマンドバックアップと復元

> DynamoDB には、オンデマンドバックアップ機能があります。この機能により、長期間の保存とアーカイブのために、テーブルの完全なバックアップを作成できます。詳細については、「DynamoDB のオンデマンドバックアップおよび復元」を参照してください。

ポイントインタイムリカバリ

ポイントインタイムリカバリを使用することで、DynamoDB テーブルを偶発的な書き込みや削除のオペレーションから保護できます。ポイントインタイムリカバリを有効化すれば、オンデマンドバックアップの作成、維持、スケジュールを心配する必要はありません。詳細については、「DynamoDB のポイントインタイムリカバリ」を参照してください。

AWS リージョン間でのグローバルテーブルの同期

DynamoDB では、一貫性のある高速パフォーマンスを維持しながら、スループットとストレージの要件を処理できるように、テーブルのデータとトラフィックが十分な数のサーバーに自動的に分散されます。また、すべてのデータをソリッドステートディスク (SSD) に保存し、AWS リージョン内の複数のアベイラビリティーゾーン間で自動的にレプリケートするため、組み込みの高い可用性とデータ堅牢性が実現します。グローバルテーブルを使用して、DynamoDB テーブルを AWS リージョン間で同期させることができます。

Amazon DynamoDB のインフラストラクチャセキュリティ

マネージドサービスとして、Amazon DynamoDB は、AWS Well-Architected フレームワークにあるインフラストラクチャ保護で説明されている AWS グローバルネットワークセキュリティ手順によって保護されます。

AWS が公開した API コールを使用して、ネットワーク経由で DynamoDB にアクセスします。クライアントは TLS (Transport Layer Security) バージョン 1.2 または 1.3 を使用できます。また、Ephemeral Diffie-Hellman (DHE) や Elliptic Curve Ephemeral Diffie-Hellman (ECDHE) などの Perfect Forward Secrecy (PFS) を使用した暗号スイートもサポートしている必要があります。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。また、リクエストは、アクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、AWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

DynamoDB 用の 仮想プライベートクラウド (VPC) エンドポイントを使用することで、VPC 内の Amazon EC2 インスタンスがパブリックインターネットにさらされることなく、プライベート IP アドレスを使用して DynamoDB にアクセスできるようになります。詳細については、「」を参照してくださいAmazon VPC エンドポイントを使用して DynamoDB にアクセスする

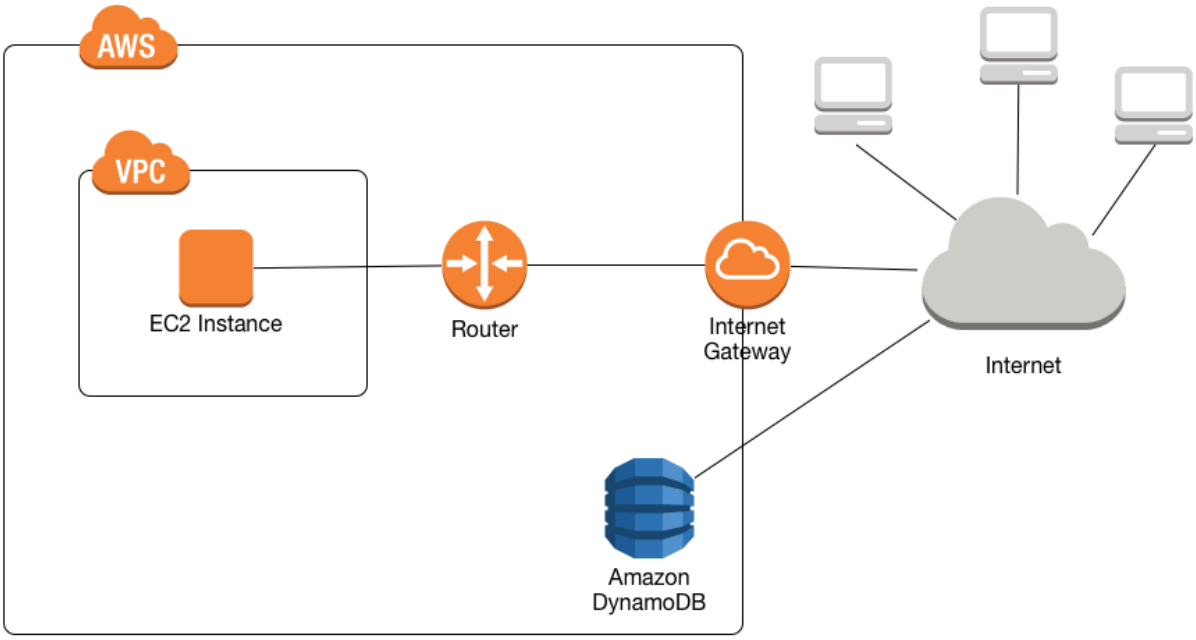
Amazon VPC エンドポイントを使用して DynamoDB にアクセスする

セキュリティ上の理由から、多くの AWS ユーザーがアプリケーションを Amazon Virtual Private Cloud 環境 (Amazon VPC) 内で実行しています。Amazon VPC を使用すると、Amazon EC2 インスタンスを仮想プライベートクラウドで作成できます。そのため、パブリックインターネットなどの他のネットワークから論理的に分離されます。Amazon VPC を使用すると、IP アドレスの範囲、サブネット、ルーティングテーブル、ネットワークゲートウェイ、セキュリティ設定を適切に管理できます。

> **Note**
>
> 2013 年 12 月 4 日以降に AWS アカウントを作成した場合は、各 AWS リージョンにデフォルトの VPC が用意されています。デフォルトの VPC は、使用できる状態になっています。追加で設定手順を実行することなく、すぐに利用開始できます。
> 詳細については、Amazon VPC ユーザーガイドの「デフォルト VPC とデフォルトサブネット」を参照してください。

パブリックインターネットにアクセスするには、VPC にインターネットゲートウェイ (VPC をインターネットに接続する仮想ルーター) が必要です。これにより、VPC 内の Amazon EC2 で実行されているアプリケーションで、Amazon DynamoDB などのインターネットリソースにアクセスすることが可能になります。

デフォルトでは、DynamoDB との通信において、SSL/TLS 暗号化を使用してネットワークトラフィックを保護する HTTPS プロトコルが使用されます。次の図は、VPC 内の Amazon EC2 インスタンスから DynamoDB にアクセスするために、VPC エンドポイントではなくインターネットゲートウェイを使用する例を示しています。



多くのお客様が、パブリックインターネット間のデータ送受信に関して、プライバシーとセキュリティに関する正当な懸念を抱いています。これらの懸念を解決するために、仮想プライベートネットワーク (VPN) を使用して、すべての DynamoDB ネットワークトラフィックをお客様の企業ネットワークのインフラストラクチャ経由でルーティングできます。ただし、このアプローチでは、帯域幅や可用性の課題が生じる場合があります。

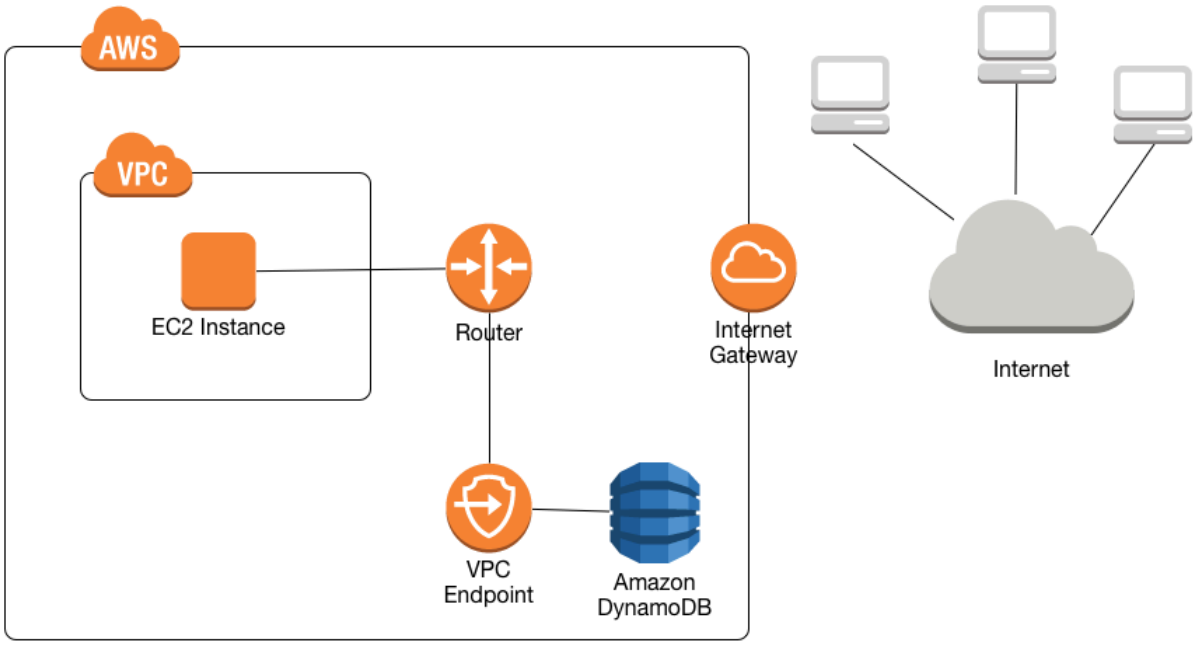
DynamoDB 用の VPC エンドポイントでは、これらの課題は軽減されます。DynamoDB 用の VPC エンドポイントを使用すると、VPC 内の Amazon EC2 インスタンスがパブリックインターネットにさらされることなく、プライベート IP アドレスを使用して DynamoDB にアクセスできるようになります。EC2 インスタンス にパブリック IP アドレスは必要ありません。また、VPC にインターネットゲートウェイ、NAT デバイス、仮想プライベートゲートウェイは不要です。DynamoDB へのアクセスを制御するには、エンドポイントのポリシーを使用します。VPC と AWS サービス間のトラフィックは、Amazon ネットワークを離れません。

> **Note**
>
> パブリック IP アドレスを使用する場合でも、AWS でホストされているインスタンスとサービスの間で発生するすべての VPC 通信は、AWS ネットワーク内でのプライベートな通信となります。AWS ネットワークから発信され、送信先が AWS ネットワーク上のパケット

は、AWS 中国リージョンとの送受信されるトラフィックを除き、AWS グローバルネットワークに残ります。

DynamoDB 用の VPC エンドポイントを作成する際、リージョン内の DynamoDB エンドポイント (例: dynamodb.us-west-2.amazonaws.com) に対するリクエストはすべて、Amazon ネットワーク内のプライベートの DynamoDB エンドポイントにルーティングされます。VPC 内の EC2 インスタンスで実行されているアプリケーションを変更する必要はありません。エンドポイント名は変わりませんが、DynamoDB へのルートは Amazon ネットワーク内に完全にとどまります。パブリックインターネットにアクセスすることはありません。

次の図は、VPC 内の EC2 インスタンスが VPC エンドポイントを使用して DynamoDB にアクセスする様子を示しています。



詳細については、「the section called “チュートリアル: DynamoDB 用の VPC エンドポイントを使用する”」を参照してください。

Amazon VPC エンドポイントと DynamoDB の共有

VPC サブネットのゲートウェイエンドポイントから DynamoDB サービスにアクセスできるようにするには、その VPC サブネットに対する所有者アカウントのアクセス許可が必要です。

VPC サブネットのゲートウェイエンドポイントに DynamoDB へのアクセスを許可すると、そのサブネットへのアクセス権を持つすべての AWS アカウントが DynamoDB を使用できます。つまり、VPC サブネット内のすべてのアカウントユーザーは、アクセス権が付与されている対象のすべての DynamoDB テーブルを使用できます。これには、VPC サブネットとは異なるアカウントに関連付けられた DynamoDB テーブルが含まれます。ただし、VPC サブネットの所有者は、独自の裁量により、サブネット内の特定のユーザーに対して、ゲートウェイエンドポイント経由での DynamoDB サービスの使用を制限できます。

チュートリアル: DynamoDB 用の VPC エンドポイントを使用する

このセクションでは、DynamoDB 用の VPC エンドポイントの設定および使用について説明します。

トピック

- ステップ 1: Amazon EC2 インスタンスを起動する
- ステップ 2: Amazon EC2 インスタンスを設定する
- ステップ 3: DynamoDB 用の VPC エンドポイントを作成する
- ステップ 4: (オプション) クリーンアップする

ステップ 1: Amazon EC2 インスタンスを起動する

このステップでは、デフォルトの Amazon VPC で Amazon EC2 インスタンスを起動します。その後、DynamoDB 用の VPC エンドポイントを作成して使用できます。

1. Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. [Launch Instance] (インスタンスを起動) を選択して、以下を実行します。

   ステップ 1: Amazon マシンイメージ (AMI) の選択

   - AMI のリストの上部で [Amazon Linux AMI] に移動し、[選択] を選びます。

   ステップ 2: インスタンスタイプを選択する

   - インスタンスタイプのリストの上部で、[t2.micro] を選択します。
   - [次のステップ: インスタンスの詳細の設定] を選択します。

   ステップ 3: インスタンスの詳細を設定する

   - [ネットワーク] に移動し、デフォルトの VPC を選択します。

[次の手順: ストレージの追加] を選択します。

ステップ 4: ストレージを追加する

- [Next: Tag Instance] を選択してこのステップをスキップします。

ステップ 5: インスタンスをタグ付けする

- [次のステップ: セキュリティグループの設定] を選択してこのステップをスキップします。

ステップ 6: セキュリティグループを設定する

- [Select an existing security group] (既存のセキュリティグループの選択) を選択します。
- セキュリティグループのリストで、[default (デフォルト)] を選択します。これは VPC のデフォルトのセキュリティグループです。
- [Next: Review and Launch] (次のステップ: 確認と起動) を選択します。

ステップ 7: インスタンス起動の確認

- [Launch] (起動する) を選択します。

3. [既存のキーペアを選択するか、新しいキーペアを作成します。] ウィンドウで、次のいずれかを実行します。

   - Amazon EC2 キーペアがない場合は、[Create a new key pair] (新しいキーペアの作成) を選択して指示に従います。プライベートキーファイル (.pem ファイル) をダウンロードするよう求められます。このファイルは、後で Amazon EC2 インスタンスにログインする際に必要になります。
   - 既存の Amazon EC2 キーペアがすでにある場合は、[Select a key pair] (キーペアの選択) を選択して、リストからキーペアを選択します。Amazon EC2 インスタンスにログインするには、既にプライベートキーファイル (.pem ファイル) が利用可能になっている必要があります。

4. キーペアを設定してある場合は、[Launch Instances] (インスタンスの起動) を選択します。
5. Amazon EC2 コンソールのホームページに戻り、起動したインスタンスを選択します。下のペインの [Description] (説明) タブで、インスタンスの [Public DNS] (パブリック DNS) を見つけます。例: `ec2-00-00-00-00.us-east-1.compute.amazonaws.com`。

このパブリック DNS 名をメモします。パブリック DNS 名は、このチュートリアル (ステップ 2: Amazon EC2 インスタンスを設定する) の次のステップで必要になります。

> **Note**
>
> Amazon EC2 インスタンスが使用できるようになるまで数分かかります。次のステップに行く前に、[インスタンスの状態] が `running` で、その [ステータスチェック] がすべてパスしていることを確認します。

ステップ 2: Amazon EC2 インスタンスを設定する

Amazon EC2 インスタンスが使用できるようになったら、そのインスタンスにログインして、最初に使用できるように準備できます。

> **Note**
>
> 次の手順は、Linux を実行するコンピュータから Amazon EC2 インスタンスに接続していることを想定しています。その他の接続方法については、「Amazon EC2 ユーザーガイド」の「Linux インスタンスへの接続」を参照してください。

1. Amazon EC2 インスタンスへのインバウンド SSH トラフィックを認可する必要があります。これを行うには、新しい EC2 セキュリティグループを作成し、そのセキュリティグループを EC2 インスタンスに割り当てます。

   a. ナビゲーションペインで、[Security Groups] を選択します。

   b. [Create Security Group (セキュリティグループの作成)] を選択します。[セキュリティグループを作成] ウィンドウで以下を行います。

      - [Security group name] (セキュリティグループ名) — セキュリティグループの名前を入力します。例: `my-ssh-access`
      - [Description] (説明) — セキュリティグループの簡単な説明を入力します。
      - VPC — デフォルトの VPC を選択します。
      - [Security group rules] (セキュリティグループのルール) セクションで、[Add Rule] (ルールの追加) を選択して、次の操作を行います

- [Type] (タイプ) — SSH を選択します。
- [Source] (ソース) — My IP を選択します。

すべての設定が正しいことを確認したら、[作成] を選択します。

c. ナビゲーションペインで、[インスタンス] を選択します。

d. ステップ 1: Amazon EC2 インスタンスを起動する で起動した Amazon EC2 インスタンスを選択します。

e. [Actions] (アクション)、[Networking] (ネットワーキング)、[Change Security Groups] (セキュリティグループの変更) の順に選択します。

f. [Change Security Groups] (セキュリティグループの変更) で、この手順で先に作成したセキュリティグループを選択します (例: `my-ssh-access`)。既存の `default` のセキュリティグループも選択する必要があります。すべての設定が正しいことを確認したら、[Assign Security Groups] (セキュリティグループの割り当て) を選択します。

2. 次の例のように、`ssh` コマンドを使用して Amazon EC2 インスタンスにログインします。

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

プライベートキーファイル (.pem ファイル) とインスタンスのパブリック DNS 名を指定する必要があります。(「ステップ 1: Amazon EC2 インスタンスを起動する」を参照してください)。

ログイン ID は `ec2-user` です。パスワードは不要です。

3. 次の例に示すように、AWS 認証情報を設定します。プロンプトが表示されたら、AWS アクセスキー ID、シークレットキー、デフォルトのリージョン名を入力します。

```
aws configure
```

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]:
```

これで、DynamoDB 用の VPC エンドポイントを作成する準備ができました。

ステップ 3: DynamoDB 用の VPC エンドポイントを作成する

このステップでは、DynamoDB 用の VPC エンドポイントを作成し、テストを行い正常に動作することを確認します。

1. 開始する前に、パブリックエンドポイントを使用して DynamoDB と通信できることを確認します。

   ```
   aws dynamodb list-tables
   ```

   出力では、現在所有している DynamoDB テーブルのリストが表示されます。(テーブルがない場合、リストは空になります。)。

2. DynamoDB が、現在の AWS リージョンで VPC エンドポイントを作成するために利用可能なサービスであることを確認します。(コマンドは太字で示され、その後に出力例が続きます。)

   ```
   aws ec2 describe-vpc-endpoint-services
   ```

   ```
   {
       "ServiceNames": [
           "com.amazonaws.us-east-1.s3",
           "com.amazonaws.us-east-1.dynamodb"
       ]
   }
   ```

   出力例では、DynamoDB が利用可能なサービスの 1 つであるため、VPC エンドポイントの作成を続行できます。

3. VPC 識別子を決定します。

   ```
   aws ec2 describe-vpcs
   ```

   ```
   {
       "Vpcs": [
           {
               "VpcId": "vpc-0bbc736e",
               "InstanceTenancy": "default",
               "State": "available",
               "DhcpOptionsId": "dopt-8454b7e1",
               "CidrBlock": "172.31.0.0/16",
               "IsDefault": true
   ```

```
        }
    ]
}
```

出力例では、VPC ID は vpc-0bbc736e です。

4. VPC エンドポイントを作成します。--vpc-id パラメータで、前のステップの VPC ID を指定します。--route-table-ids パラメータを使用して、エンドポイントをルートテーブルに関連付けます。

```
aws ec2 create-vpc-endpoint --vpc-id vpc-0bbc736e --service-name com.amazonaws.us-east-1.dynamodb --route-table-ids rtb-11aa22bb
```

```
{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\",\"Statement\":[{\"Effect\":\"Allow\",\"Principal\":\"*\",\"Action\":\"*\",\"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
        "RouteTableIds": [
            "rtb-11aa22bb"
        ],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

5. VPC エンドポイント経由で DynamoDB にアクセスできることを確認します。

```
aws dynamodb list-tables
```

必要に応じて、DynamoDB 用の他の AWS CLI コマンドを試すことができます。詳細については、AWS CLI コマンドリファレンスを参照してください。

ステップ 4: (オプション) クリーンアップする

このチュートリアルで作成したリソースを削除する場合は、次の手順に従ってください。

DynamoDB 用の VPC エンドポイントを削除するには

1. Amazon EC2 インスタンスにログインします。
2. VPC エンドポイント ID を決定します。

```
aws ec2 describe-vpc-endpoints
```

```
{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\",\"Statement\":[{\"Effect\":
\"Allow\",\"Principal\":\"*\",\"Action\":\"*\",\"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
        "RouteTableIds": [],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

出力例では、VPC エンドポイント ID は `vpce-9b15e2f2` です。

3. VPC エンドポイントを削除します。

```
aws ec2 delete-vpc-endpoints --vpc-endpoint-ids vpce-9b15e2f2
```

```
{
    "Unsuccessful": []
}
```

空の配列 `[]` は成功を示します (失敗したリクエストはありませんでした)。

Amazon EC2 インスタンスを終了するには

1. Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. ナビゲーションペインで、[インスタンス] を選択します。
3. Amazon EC2 インスタンスを選択します。
4. [Actions]、[Instance State]、[Terminate] の順に選択します。

5. 確認ウィンドウで、[Yes, Terminate] (はい、終了します) を選択します。

AWS PrivateLink for DynamoDB

AWS PrivateLink for DynamoDB を使うと、仮想プライベートクラウド (Amazon VPC) でインターフェイス Amazon VPC エンドポイント (インターフェイスエンドポイント) をプロビジョニングできます。これらのエンドポイントには、オンプレミスにあるアプリケーションから VPN および Direct Connect 経由で、または別の AWS リージョンにあるアプリケーションから Amazon VPC ピアリング経由で直接アクセスできます。AWS PrivateLink とインターフェイスエンドポイントを使用することで、アプリケーションから DynamoDB へのプライベートネットワーク接続を簡素化できます。

VPC 内のアプリケーションは、パブリック IP アドレスがなくても、DynamoDB VPC インターフェイスエンドポイントを使用して DynamoDB と通信し、DynamoDB の操作を実行できます。インターフェイスエンドポイントは、Amazon VPC 内のサブネットからプライベート IP アドレスが割り当てられた 1 つ以上の Elastic Network Interface (ENI) で表されます。インターフェイスエンドポイントを介した DynamoDB へのリクエストが Amazon ネットワーク外に出ることはありません。AWS Direct Connect または AWS Virtual Private Network (Site-to-Site VPN) を介して、オンプレミスのアプリケーションから Amazon VPC 内のインターフェイスエンドポイントにアクセスすることもできます。Amazon VPC をオンプレミスネットワークに接続する方法の詳細については、「Direct Connect ユーザーガイド」および「AWS Site-to-Site VPN ユーザーガイド」を参照してください。

インターフェイスエンドポイントの一般的な情報については、「AWS PrivateLink ガイド」の「インターフェイス Amazon VPC エンドポイント (AWS PrivateLink)」を参照してください。AWS PrivateLink は Amazon DynamoDB Streams エンドポイントでもサポートされています。詳細については、「the section called “AWS PrivateLink for DynamoDB Streams”」を参照してください。

トピック

- Amazon DynamoDB で使用される Amazon VPC エンドポイントのタイプ
- AWS PrivateLink for Amazon DynamoDB を使用する場合の考慮事項
- Amazon VPC エンドポイントの作成
- Amazon DynamoDB インターフェイスエンドポイントへのアクセス
- DynamoDB インターフェイスエンドポイントから DynamoDB テーブルおよびコントロール API オペレーションへのアクセス
- オンプレミスの DNS 設定の更新
- DynamoDB 用 Amazon VPC エンドポイントポリシーの作成

- AWS マネジメントコンソール プライベートアクセスでの DynamoDB エンドポイントの使用
- AWS PrivateLink for DynamoDB Streams
- DynamoDB Accelerator (DAX) での AWS PrivateLink の使用

Amazon DynamoDB で使用される Amazon VPC エンドポイントのタイプ

Amazon DynamoDB へのアクセスには、ゲートウェイエンドポイントとインターフェイスエンドポイント (AWS PrivateLink を使用) の 2 つのタイプの Amazon VPC エンドポイントを使用できます。ゲートウェイエンドポイントは、AWS ネットワーク経由で Amazon VPC から DynamoDB にアクセスするために、ルートテーブルで指定するゲートウェイです。インターフェイスエンドポイントは、プライベート IP アドレスを使用して、Amazon VPC 内、オンプレミス、または Amazon VPC ピアリングや AWS Transit Gateway を使用する別の AWS リージョンにある Amazon VPC から DynamoDB にリクエストをルーティングすることにより、ゲートウェイエンドポイントの機能を拡張します。詳細については、「VPC ピア機能とは」および「Transit Gateway と VPC ピアリング」を参照してください。

インターフェイスエンドポイントは、ゲートウェイエンドポイントと互換性があります。Amazon VPC 内に既存のゲートウェイエンドポイントがある場合は、同じ Amazon VPC で両方のタイプのエンドポイントを使用できます。

DynamoDB のゲートウェイエンドポイント	DynamoDB のインターフェイスエンドポイント
いずれの場合も、ネットワークトラフィックは AWS ネットワーク上に残ります。	
Amazon DynamoDB パブリック IP アドレスを使用する	Amazon VPC のプライベート IP アドレスを使用して Amazon DynamoDB にアクセスする
オンプレミスからのアクセスを許可しない	オンプレミスからのアクセスを許可する
別の AWS リージョン からのアクセスを許可しない	Amazon VPC ピアリングまたは AWS Transit Gateway を使用する別の AWS リージョンにある Amazon VPC エンドポイントからのアクセスを許可する
課金されない	請求される

ゲートウェイエンドポイントの詳細については、「AWS PrivateLink ガイド」の「Gateway Amazon VPC endpoints」を参照してください。

AWS PrivateLink for Amazon DynamoDB を使用する場合の考慮事項

Amazon VPC に関する考慮事項が AWS PrivateLink for Amazon DynamoDB に適用されます。詳細については、AWS PrivateLink ガイドの「インターフェイスエンドポイントの考慮事項」と「AWS PrivateLink クォータ」を参照してください。また、以下の制約も適用されます。

AWS PrivateLink for Amazon DynamoDB では、以下はサポートされていません。

- Transport Layer Security (TLS) 1.1
- プライベートおよびハイブリッドドメインネームシステム (DNS) サービス

> **Important**
>
> DynamoDB エンドポイントの DNS 名 (`dynamodb.region.amazonaws.com` や `*.region.amazonaws.com` など) を上書きしてトラフィックをインターフェイスエンドポイントにルーティングするために、プライベートホストゾーンを作成しないでください。DynamoDB の DNS 設定は、時間の経過とともに変更される可能性があります。カスタム DNS オーバーライドは、これらの変更と互換性がないため、リクエストがインターフェイスエンドポイントではなくパブリック IP アドレス経由で予期せずルーティングされる可能性があります。
>
> AWS PrivateLink を介して DynamoDB にアクセスするには、Amazon VPC エンドポイント URL を直接使用するようにクライアントを設定します (例: `https://vpce-1a2b3c4d-5e6f.dynamodb.region.vpce.amazonaws.com`)。

有効にする AWS PrivateLink エンドポイントごとに、1 秒あたり最大 5 万件のリクエストを送信できます。

> **Note**
>
> AWS PrivateLink エンドポイントへのネットワーク接続タイムアウトは DynamoDB エラーレスポンスの範囲ではないため、PrivateLink エンドポイントに接続するアプリケーションで適切に処理する必要があります。

Amazon VPC エンドポイントの作成

Amazon VPC インターフェイスエンドポイントを作成するには、「AWS PrivateLink ガイド」の「Create an Amazon VPC endpoint」を参照してください。

Amazon DynamoDB インターフェイスエンドポイントへのアクセス

インターフェイスエンドポイントを作成すると、DynamoDB はエンドポイント固有の 2 つのタイプの DynamoDB DNS 名 (Regional および Zonal) を生成します。

- Regional DNS 名では、一意の Amazon VPC エンドポイント ID、サービス識別子、AWS リージョン、および vpce.amazonaws.com が名前に含まれます。例えば、Amazon VPC エンドポイント ID が *vpce-1a2b3c4d* の場合、生成される DNS 名は *vpce-1a2b3c4d-5e6f*.dynamodb.us-east-1.vpce.amazonaws.com のようになります。
- Zonal DNS 名には、アベイラビリティーゾーンが含まれます (*vpce-1a2b3c4d-5e6f*-**us-east-1a.**dynamodb.us-east-1.vpce.amazonaws.com など)。このオプションは、アーキテクチャがアベイラビリティーゾーンを分離する場合に使用できます。例えば、障害を隔離し、リージョン間のデータ転送コストを削減するために使用できます。

> **Note**
>
> 最適な信頼性を実現するには、最低 3 つのアベイラビリティーゾーンにサービスをデプロイすることをお勧めします。

DynamoDB インターフェイスエンドポイントから DynamoDB テーブルおよびコントロール API オペレーションへのアクセス

DynamoDB インターフェイスエンドポイントを介して DynamoDB テーブルおよびコントロール API オペレーションにアクセスするには、AWS CLI または AWS SDK を使用します。

AWS CLI の例

AWS CLI コマンドを使って DynamoDB インターフェイスエンドポイントを介して DynamoDB テーブルまたは DynamoDB コントロール API オペレーションにアクセスするには、--region および --endpoint-url パラメータを使用します。

例: VPC エンドポイントの作成

```
aws ec2 create-vpc-endpoint \
--region us-east-1 \
--service-name com.amazonaws.us-east-1.dynamodb \
--vpc-id client-vpc-id \
--subnet-ids client-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id
```

例: VPC エンドポイントの変更

```
aws ec2 modify-vpc-endpoint \
--region us-east-1 \
--vpc-endpoint-id client-vpc-endpoint-id \
--policy-document policy-document \ #example optional parameter
--add-security-group-ids security-group-ids \ #example optional parameter
# any additional parameters needed, see Privatelink documentation for more details
```

例: エンドポイント URL を使ったテーブルの一覧表示

次の例では、リージョン `us-east-1`、VPC エンドポイント ID の DNS 名 `vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` をユーザー自身の情報に置き換えます。

```
aws dynamodb --region us-east-1 --endpoint https://vpce-1a2b3c4d-5e6f.dynamodb.us-
east-1.vpce.amazonaws.com list-tables
```

AWS SDK の例

AWS SDK を使って DynamoDB インターフェイスエンドポイントを介して DynamoDB テーブルまたは DynamoDB コントロール API オペレーションにアクセスするには、SDK を最新バージョンに更新します。次に、エンドポイント URL を使用して DynamoDB インターフェイスエンドポイントを介してテーブルまたは DynamoDB コントロール API オペレーションにアクセスするように、クライアントを設定します。

SDK for Python (Boto3)

例: エンドポイント URL を使用して DynamoDB テーブルにアクセスする

次の例では、リージョン `us-east-1` および VPC エンドポイント ID `https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com` をユーザー自身の情報に置き換えます。

```
ddb_client = session.client(
service_name='dynamodb',
region_name='us-east-1',
endpoint_url='https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com'
)
```

SDK for Java 1.x

例: エンドポイント URL を使用して DynamoDB テーブルにアクセスする

次の例では、リージョン us-east-1 および VPC エンドポイント ID https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
//client build with endpoint config
final AmazonDynamoDB dynamodb =
 AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration(
                "https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com",
                Regions.DEFAULT_REGION.getName()
        )
).build();
```

SDK for Java 2.x

例: エンドポイント URL を使用して DynamoDB テーブルにアクセスする

次の例では、リージョン us-east-1 と VPC エンドポイント ID https://vpce-1a2b3c4d-5e6f.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
Region region = Region.US_EAST_1;
dynamoDbClient = DynamoDbClient.builder().region(region)
.endpointOverride(URI.create("https://vpce-1a2b3c4d-5e6f.dynamodb.us-
east-1.vpce.amazonaws.com"))
.build()
```

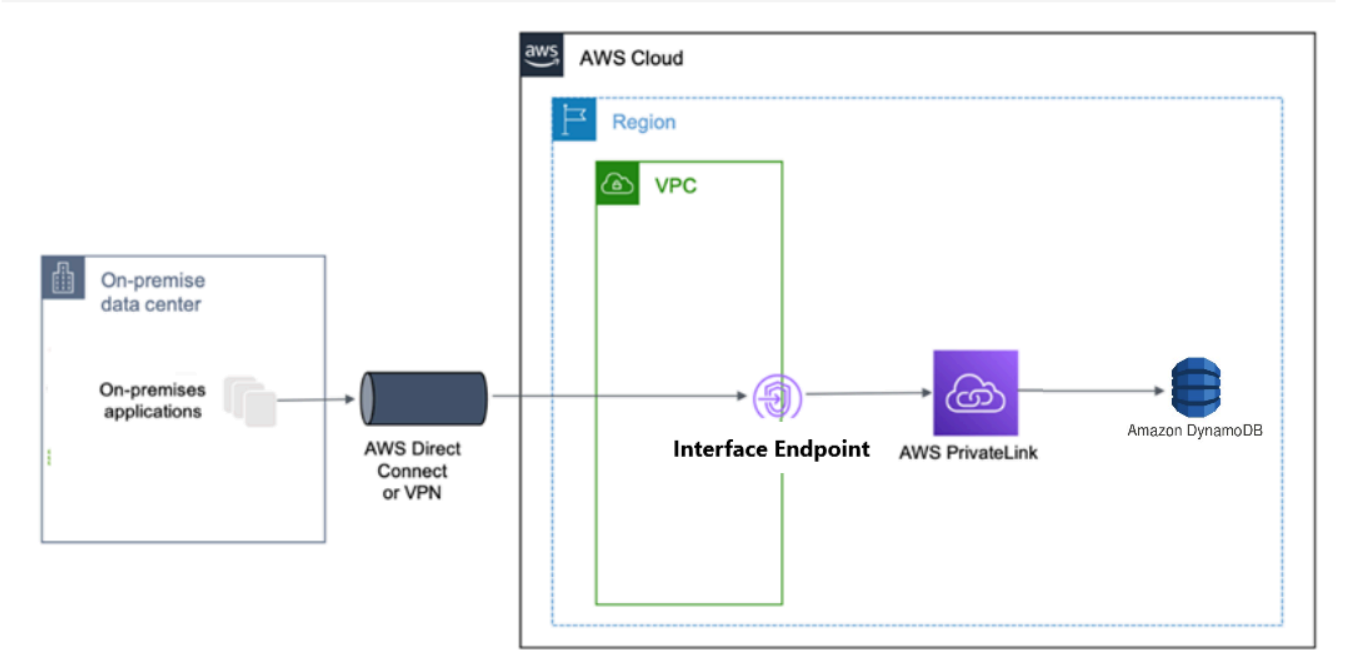
オンプレミスの DNS 設定の更新

エンドポイント固有の DNS 名を使用して DynamoDB のインターフェイスエンドポイントにアクセスする場合、オンプレミス DNS リゾルバーを更新する必要はありません。パブリック DynamoDB

DNS ドメインから、インターフェイスエンドポイントのプライベート IP アドレスを使用してエンドポイント固有の DNS 名を解決できます。

Amazon VPC 内のゲートウェイエンドポイントまたはインターネットゲートウェイは使用せず、インターフェイスエンドポイントを使用して DynamoDB にアクセスする

次の図に示すように、Amazon VPC 内のインターフェイスエンドポイントは、Amazon VPC 内のアプリケーションとオンプレミスアプリケーションの両方を Amazon ネットワーク経由で DynamoDB にルーティングできます。



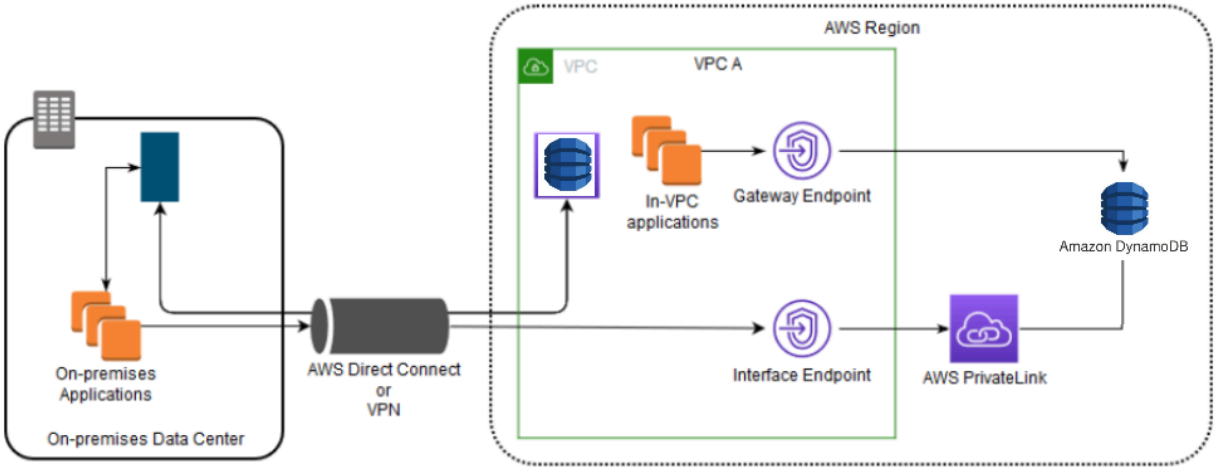
この図表は、以下を示すものです:

- オンプレミスネットワークでは、Direct Connect または Site-to-Site VPN を使用して Amazon VPC A に接続します。
- オンプレミスと Amazon VPC A 内のアプリケーションでは、エンドポイント固有の DNS 名を使用して DynamoDB インターフェイスエンドポイントを介して DynamoDB にアクセスします。
- オンプレミスのアプリケーションは、Direct Connect (または Site-to-Site VPN) を介して Amazon VPC 内のインターフェイスエンドポイントにデータを送信します。AWS PrivateLink は、AWS ネットワークを経由してデータをインターフェイスエンドポイントから DynamoDB に移動します。

- Amazon VPC 内のアプリケーションも、インターフェイスエンドポイントにトラフィックを送信します。AWS PrivateLink は、AWS ネットワークを経由してデータをインターフェイスエンドポイントから DynamoDB に移動します。

ゲートウェイエンドポイントとインターフェイスエンドポイントを同じ Amazon VPC で併用して DynamoDB にアクセスする

次の図に示すように、インターフェイスエンドポイントを作成し、同じ Amazon VPC 内に既存のゲートウェイエンドポイントも保持できます。このアプローチにより、Amazon VPC 内のアプリケーションが引き続きゲートウェイエンドポイントを介して DynamoDB にアクセスすることが許可されます。これについての請求はありません。インターフェイスエンドポイントは、DynamoDB にアクセスするオンプレミスのアプリケーションだけが使用することになります。この方法で DynamoDB にアクセスするには、DynamoDB のエンドポイント固有の DNS 名を使用するようにオンプレミスのアプリケーションを更新する必要があります。



この図表は、以下を示すものです:

- オンプレミスのアプリケーションは、エンドポイント固有の DNS 名を使用して、Direct Connect (または Site-to-Site VPN) を介して Amazon VPC 内のインターフェイスエンドポイントにデータを送信します。AWS PrivateLink は、AWS ネットワークを経由してデータをインターフェイスエンドポイントから DynamoDB に移動します。
- Amazon VPC 内のアプリケーションは、デフォルトのリージョン DynamoDB 名を使用し、AWS ネットワークを経由して DynamoDB に接続するゲートウェイエンドポイントにデータを送信します。

ゲートウェイエンドポイントの詳細については、「Amazon VPC ユーザーガイド」の「Gateway Amazon VPC endpoints」を参照してください。

DynamoDB 用 Amazon VPC エンドポイントポリシーの作成

Amazon VPC エンドポイントに DynamoDB へのアクセスをコントロールするエンドポイントポリシーをアタッチできます。このポリシーでは、以下の情報を指定します。

- アクションを実行できる AWS Identity and Access Management (IAM) プリンシパル
- 実行可能なアクション
- アクションを実行できるリソース

トピック

- 例: Amazon VPC エンドポイントから特定のテーブルへのアクセスの制限

例: Amazon VPC エンドポイントから特定のテーブルへのアクセスの制限

特定の DynamoDB テーブルへのアクセスのみを制限するエンドポイントポリシーを作成できます。このタイプのポリシーは、テーブルを使用する他の AWS のサービスが Amazon VPC にある場合に便利です。次のテーブルポリシーは、*DOC-EXAMPLE-TABLE* へのアクセスのみを制限します。このエンドポイントポリシーを使用するには、*DOC-EXAMPLE-TABLE* をテーブルの名前に置き換えます。

JSON

```
{
"Version":"2012-10-17",
  "Id": "Policy1216114807515",
  "Statement": [
    { "Sid": "Access-to-specific-table-only",
      "Principal": "*",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem"
      ],
      "Effect": "Allow",
      "Resource": ["arn:aws:dynamodb:us-east-1:111122223333:table/DOC-EXAMPLE-
TABLE",
```

```
                "arn:aws:dynamodb:us-east-1:111122223333:table/DOC-EXAMPLE-
TABLE/*"]
    }
  ]
}
```

AWS マネジメントコンソール プライベートアクセスでの DynamoDB エンドポイントの使用

AWS マネジメントコンソール プライベートアクセスの DynamoDB コンソールで VPC エンドポイントを使用する場合は、DynamoDB と DynamoDB Streams の DNS 設定を設定する必要があります。

AWS マネジメントコンソール プライベートアクセスでアクセスできるように DynamoDB を設定するには、次の 2 つの VPC エンドポイントを作成する必要があります。

- `com.amazonaws.<region>.dynamodb`
- `com.amazonaws.<region>.dynamodb-streams`

VPC エンドポイントを作成する場合、Route53 コンソールに移動し、リージョンエンドポイント `dynamodb.us-east-1.amazonaws.com` を使用して DynamoDB のプライベートホストゾーンを作成します。

プライベートホストゾーンに次の 2 つのエイリアスレコードを作成します。

- トラフィックを VPC エンドポイント `com.amazonaws.<region>.dynamodb` にルーティングする `dynamodb.<region>.amazonaws.com`。
- トラフィックを VPC エンドポイント `com.amazonaws.<region>.dynamodb-streams` にルーティングする `streams.dynamodb.<region>.amazonaws.com`。

AWS PrivateLink for DynamoDB Streams

AWS PrivateLink for Amazon DynamoDB Streams を使用すると、仮想プライベートクラウド (Amazon VPC) でインターフェイス Amazon VPC エンドポイント (インターフェイスエンドポイント) をプロビジョニングできます。これらのエンドポイントには、オンプレミスにあるアプリケーションから VPN および Direct Connect 経由で、または別の AWS リージョン にあるアプリケーションから Amazon VPC ピアリング経由で直接アクセスできます。AWS PrivateLink とインターフェイ

スエンドポイントを使用することで、アプリケーションから DynamoDB Streams へのプライベートネットワーク接続を簡素化できます。

Amazon VPC 内のアプリケーションは、パブリック IP アドレスがなくても、Amazon VPC インターフェイスエンドポイントを使用して DynamoDB Streams と通信し、DynamoDB Streams の操作を実行できます。インターフェイスエンドポイントは、Amazon VPC 内のサブネットからプライベート IP アドレスが割り当てられた 1 つ以上の Elastic Network Interface (ENI) で表されます。インターフェイスエンドポイントを介した DynamoDB Streams へのリクエストが Amazon ネットワーク外に出ることはありません。Direct Connect または AWS Virtual Private Network (AWS VPN) を介して、オンプレミスのアプリケーションから Amazon VPC 内のインターフェイスエンドポイントにアクセスすることもできます。AWS Virtual Private Network をオンプレミスネットワークに接続する方法の詳細については、「Direct Connect ユーザーガイド」および「AWS Site-to-Site VPN ユーザーガイド」を参照してください。

インターフェイスエンドポイントの一般的な情報については、「Interface Amazon VPC endpoints (AWS PrivateLink)」を参照してください。

> **Note**
>
> DynamoDB Streams では、インターフェイスエンドポイントのみがサポートされています。ゲートウェイエンドポイントはサポートされていません。

トピック

- AWS PrivateLink for Amazon DynamoDB Streams を使用する場合の考慮事項
- Amazon VPC エンドポイントの作成
- Amazon DynamoDB Streams インターフェイスエンドポイントへのアクセス
- DynamoDB Streams インターフェイスエンドポイントから DynamoDB Streams API オペレーションへのアクセス
- AWS SDK の例
- DynamoDB Streams 用 Amazon VPC エンドポイントポリシーの作成
- AWS マネジメントコンソール プライベートアクセスでの DynamoDB エンドポイントの使用

AWS PrivateLink for Amazon DynamoDB Streams を使用する場合の考慮事項

Amazon VPC に関する考慮事項が AWS PrivateLink for Amazon DynamoDB Streams に適用されます。詳細については、「interface endpoint considerations」および「AWS PrivateLink quotas」を参照してください。以下の制限が適用されます。

AWS PrivateLink for Amazon DynamoDB Streams では、以下はサポートされていません。

- Transport Layer Security (TLS) 1.1
- プライベートおよびハイブリッドドメインネームシステム (DNS) サービス

> **Important**
>
> DynamoDB Streams エンドポイントの DNS 名を上書きして、トラフィックをインターフェイスエンドポイントにルーティングするために、プライベートホストゾーンを作成しないでください。DynamoDB の DNS 設定は時間の経過とともに変更される可能性があり、カスタム DNS オーバーライドにより、リクエストがインターフェイスエンドポイントではなくパブリック IP アドレス経由で予期せずルーティングされる可能性があります。
> AWS PrivateLink を介して DynamoDB Streams にアクセスするには、Amazon VPC エンドポイント URL を直接使用するようにクライアントを設定します (例: `https://vpce-1a2b3c4d-5e6f.streams.dynamodb.region.vpce.amazonaws.com`)。

> **Note**
>
> AWS PrivateLink エンドポイントへのネットワーク接続タイムアウトは DynamoDB Streams エラーレスポンスの範囲ではないため、AWS PrivateLink エンドポイントに接続するアプリケーションで適切に処理する必要があります。

Amazon VPC エンドポイントの作成

Amazon VPC インターフェイスエンドポイントを作成するには、「AWS PrivateLink ガイド」の「Create an Amazon VPC endpoint」を参照してください。

Amazon DynamoDB Streams インターフェイスエンドポイントへのアクセス

インターフェイスエンドポイントを作成すると、DynamoDB はエンドポイント固有の 2 つのタイプの DynamoDB Streams DNS 名 (Regional および Zonal) を生成します。

- Regional DNS 名では、一意の Amazon VPC エンドポイント ID、サービス識別子、AWS リージョン、および vpce.amazonaws.com が名前に含まれます。例えば、Amazon VPC エンドポイント ID が *vpce-1a2b3c4d* の場合、生成される DNS 名は *vpce-1a2b3c4d-5e6f*.streams.dynamodb.us-east-1.vpce.amazonaws.com のようになります。
- Zonal DNS 名には、アベイラビリティーゾーンが含まれます (*vpce-1a2b3c4d-5e6f*-**us-east-1a.**streams.dynamodb.us-east-1.vpce.amazonaws.com など)。このオプションは、アーキテクチャがアベイラビリティーゾーンを分離する場合に使用できます。例えば、障害を隔離し、リージョン間のデータ転送コストを削減するために使用できます。

DynamoDB Streams インターフェイスエンドポイントから DynamoDB Streams API オペレーションへのアクセス

DynamoDB Streams インターフェイスエンドポイントを介して DynamoDB Streams API オペレーションにアクセスするには、AWS CLI SDK または AWS SDK を使用します。

AWS CLI の例

AWS CLI コマンドで DynamoDB Streams インターフェイスエンドポイントを介して DynamoDB Streams または API オペレーションにアクセスするには、--region パラメータおよび --endpoint-url パラメータを使用します。

例: VPC エンドポイントの作成

```
aws ec2 create-vpc-endpoint \
--region us-east-1 \
--service-name com.amazonaws.us-east-1.dynamodb-streams \
--vpc-id client-vpc-id \
--subnet-ids client-subnet-id \
--vpc-endpoint-type Interface \
--security-group-ids client-sg-id
```

例: VPC エンドポイントの変更

```
aws ec2 modify-vpc-endpoint \
```

```
--region us-east-1 \
--vpc-endpoint-id client-vpc-endpoint-id \
--policy-document policy-document \ #example optional parameter
--add-security-group-ids security-group-ids \ #example optional parameter
# any additional parameters needed, see Privatelink documentation for more details
```

例: エンドポイント URL を使ったストリームの一覧表示

次の例では、リージョン us-east-1、VPC エンドポイント ID の DNS 名 vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
aws dynamodbstreams --region us-east-1 –endpoint https://
vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com list-streams
```

AWS SDK の例

AWS SDK を使って DynamoDB Streams インターフェイスエンドポイントを介して Amazon DynamoDB Streams API オペレーションにアクセスするには、SDK を最新バージョンに更新します。次に、エンドポイント URL を使用して DynamoDB Streams インターフェイスエンドポイントを介して DynamoDB Streams API オペレーションにアクセスするように、クライアントを設定します。

SDK for Python (Boto3)

例: エンドポイント URL を使用して DynamoDB stream にアクセスする

次の例では、リージョン us-east-1 および VPC エンドポイント ID https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
ddb_streams_client = session.client(
service_name='dynamodbstreams',
region_name='us-east-1',
endpoint_url='https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com'
)
```

SDK for Java 1.x

例: エンドポイント URL を使用して DynamoDB stream にアクセスする

次の例では、リージョン us-east-1 および VPC エンドポイント ID https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
//client build with endpoint config
final AmazonDynamoDBStreams dynamodbstreams =
 AmazonDynamoDBStreamsClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration(
                "https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com",
                Regions.DEFAULT_REGION.getName()
        )
).build();
```

SDK for Java 2.x

例: エンドポイント URL を使用して DynamoDB stream にアクセスする

次の例では、リージョン us-east-1 および VPC エンドポイント ID https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-east-1.vpce.amazonaws.com をユーザー自身の情報に置き換えます。

```
Region region = Region.US_EAST_1;
dynamoDbStreamsClient = DynamoDbStreamsClient.builder().region(region)
.endpointOverride(URI.create("https://vpce-1a2b3c4d-5e6f.streams.dynamodb.us-
east-1.vpce.amazonaws.com"))
.build()
```

DynamoDB Streams 用 Amazon VPC エンドポイントポリシーの作成

Amazon VPC エンドポイントに DynamoDB Streams へのアクセスをコントロールするエンドポイントポリシーをアタッチできます。このポリシーでは、以下の情報を指定します。

- アクションを実行できる AWS Identity and Access Management (IAM) プリンシパル
- 実行可能なアクション
- アクションを実行できるリソース

トピック

- 例: Amazon VPC エンドポイントから特定のストリームへのアクセスの制限

例: Amazon VPC エンドポイントから特定のストリームへのアクセスの制限

特定の DynamoDB Streams へのアクセスのみを制限するエンドポイントポリシーを作成できます。このタイプのポリシーは、DynamoDB Streams を使用する他の AWS のサービス が Amazon VPC にある場合に便利です。次のストリームポリシーは、*DOC-EXAMPLE-TABLE* にアタッチされた *2025-02-20T11:22:33.444* ストリームのみへのアクセスを制限します。このエンドポイントポリシーを使用するには、*DOC-EXAMPLE-TABLE* をテーブルの名前で、*2025-02-20T11:22:33.444* をストリームラベルで、それぞれ置き換えます。

JSON

```
{
"Version":"2012-10-17",
  "Id": "Policy1216114807515",
  "Statement": [
    { "Sid": "Access-to-specific-stream-only",
      "Principal": "*",
      "Action": [
        "dynamodb:DescribeStream",
        "dynamodb:GetRecords"
      ],
      "Effect": "Allow",
      "Resource": ["arn:aws:dynamodb:us-east-1:111122223333:table/table-name/
stream/2025-02-20T11:22:33.444"]
    }
  ]
}
```

> **Note**
>
> ゲートウェイエンドポイントは DynamoDB Streams ではサポートされていません。

AWS マネジメントコンソール プライベートアクセスでの DynamoDB エンドポイントの使用

AWS マネジメントコンソール プライベートアクセスの DynamoDB コンソールで VPC エンドポイントを使用する場合は、DynamoDB と DynamoDB Streams の DNS 設定を設定する必要があります。

AWS マネジメントコンソール プライベートアクセスでアクセスできるように DynamoDB を設定するには、次の 2 つの VPC エンドポイントを作成する必要があります。

- `com.amazonaws.<region>.dynamodb`
- `com.amazonaws.<region>.dynamodb-streams`

VPC エンドポイントを作成する場合、Route53 コンソールに移動し、リージョンエンドポイント `dynamodb.us-east-1.amazonaws.com` を使用して DynamoDB のプライベートホストゾーンを作成します。

プライベートホストゾーンに次の 2 つのエイリアスレコードを作成します。

- トラフィックを VPC エンドポイント `com.amazonaws.<region>.dynamodb` にルーティングする `dynamodb.<region>.amazonaws.com`。
- トラフィックを VPC エンドポイント `com.amazonaws.<region>.dynamodb-streams` にルーティングする `streams.dynamodb.<region>.amazonaws.com`。

DynamoDB Accelerator (DAX) での AWS PrivateLink の使用

DynamoDB Accelerator (DAX) 用 AWS PrivateLink を使用すると、仮想プライベートクラウド (VPC) 内のプライベート IP アドレスを介して、`CreateCluster`、`DescribeClusters`、`DeleteCluster` などの DAX 管理 API に安全にアクセスできます。この機能を使用すると、パブリックインターネットにトラフィックを公開することなく、アプリケーションから DAX サービスにプライベートにアクセスできます。

DAX PrivateLink はデュアルスタックエンドポイント (`dax.{region}.api.aws`) をサポートし、IPv4 と IPv6 の両方の接続を可能にします。DAX 用 AWS PrivateLink を使用すると、お客様はプライベート DNS 名を使用してサービスにアクセスできます。デュアルスタックのエンドポイントサポートにより、ネットワークのプライバシーを維持しながら透過的な接続が確保されます。これにより、SDK 設定を変更することなく、パブリックインターネットと VPC エンドポイントの両方から DAX にアクセスできます。

DynamoDB Accelerator (DAX) 用 AWS PrivateLink を使用する際の考慮事項

DynamoDB Accelerator (DAX) 用 AWS PrivateLink を実装するときは、いくつかの重要な考慮事項を考慮する必要があります。

DAX のインターフェイスエンドポイントを設定する前に、次の点を考慮してください。

- DAX インターフェイスエンドポイントは、同じ AWS リージョン内の DAX 管理 API へのアクセスのみをサポートします。インターフェイスエンドポイントを使用して、他のリージョンの DAX 管理 API にアクセスすることはできません。
- DAX 管理のためにAWS マネジメントコンソールにプライベートにアクセスするには、`com.amazonaws.region.console` や関連サービスなどのサービス用の追加の VPC エンドポイントを作成する必要がある場合があります。
- DAX へのインターフェイスエンドポイントの作成と使用には料金がかかります。料金情報については、「AWS PrivateLink の料金」を参照してください。

AWS PrivateLink と DAX の連携方法

DAX のインターフェイスエンドポイントを作成する場合。

1. AWS は、インターフェイスエンドポイントに対して有効にする各サブネットにエンドポイントネットワークインターフェイスを作成します。
2. これらは、DAX 宛てのトラフィックのエントリポイントとして機能するリクエスタ管理型ネットワークインターフェイスです。
3. その後、VPC 内のプライベート IP アドレスを介して DAX にアクセスできます。
4. このアーキテクチャでは、VPC セキュリティグループを使用してエンドポイントへのアクセスを管理できます。
5. アプリケーションは、VPC 内の各インターフェイスエンドポイント経由で DynamoDB と DAX の両方にアクセスできますが、オンプレミスアプリケーションは Direct Connect または VPN 経由で接続することもできます。
6. これにより、両方のサービス間で一貫した接続モデルが提供され、アーキテクチャが簡素化され、トラフィックを AWS ネットワーク内に維持することでセキュリティが向上します。

DAX のインターフェイスエンドポイントの作成

インターフェイスエンドポイントを作成して、AWS マネジメントコンソール、AWS SDK、CloudFormation、または AWS API を使用して DAX に接続できます。

コンソールを使用して DAX のインターフェイスエンドポイントを作成するには

1. https://console.aws.amazon.com/vpc/ で Amazon VPC コンソールに移動します。
2. ナビゲーションペインで、[エンドポイント] を選択します。
3. [エンドポイントの作成] を選択します。

4. [サービスカテゴリ] で [AWS のサービス]を選択し、[サービス名] で `com.amazonaws.region.dax` を検索して選択します。
5. VPC の場合は、DAX にアクセスする VPC を選択し、サブネットの場合は、AWS がエンドポイントネットワークインターフェイスを作成するサブネットを選択します。
6. [セキュリティグループ] で、エンドポイントネットワークインターフェイスに関連付けるセキュリティグループを選択します。
7. [ポリシー] で、デフォルトの [フルアクセス] を維持するか、必要に応じてカスタマイズします。
8. [DNS 名を有効化] を選択して、エンドポイントのプライベート DNS を有効化します。SDK 設定の変更を防ぐために、プライベート DNS 名を有効にしておきます。有効にすると、アプリケーションは標準サービスの DNS 名 (例: `dax.region.amazonaws.com`) を引き続き使用できます。AWS は、この名前をエンドポイントのプライベート IP アドレスに解決するプライベートホストゾーンを VPC 内に作成します。

   > **Note**
   >
   > 必要に応じてリージョン DNS 名を使用します。ゾーン DNS 名の使用はお勧めしません。また、3 つ以上の AZ からサブネットを選択し、PrivateLink を介して最大可用性を確保します。

9. エンドポイントの作成 を選択します。

AWS CLI を使用して DAX のインターフェイスエンドポイントを作成するには

`vpc-endpoint-type` パラメータを `Interface` に設定し、`service-name` パラメータを `com.amazonaws.region.dax` に設定して、`create-vpc-endpoint` コマンドを使用します。

```
aws ec2 create-vpc-endpoint \
    --vpc-id vpc-ec43eb89 \
    --vpc-endpoint-type Interface \
    --service-name com.amazonaws.us-east-1.dax \
    --subnet-ids subnet-abcd1234 subnet-1a2b3c4d \
    --security-group-ids sg-1a2b3c4d \
    --private-dns-enabled
```

その他のリソース

AWS PrivateLink と VPC エンドポイントの使用の詳細については、以下のリソースを参照してください。

- AWS PrivateLink for DynamoDB
- AWS PrivateLink for DynamoDB Streams
- を使用して VPC をサービスに接続するAWS PrivateLink
- を使用して DynamoDB へのプライベート接続を簡素化するAWS PrivateLink
- AWS PrivateLink ホワイトペーパー

Amazon DynamoDB での設定と脆弱性の分析

AWS は、ゲストオペレーティングシステム (OS) やデータベースへのパッチ適用、ファイアウォール設定、ディザスターリカバリーなどの基本的なセキュリティタスクを処理します。これらの手順は適切な第三者によって確認され、証明されています。詳細については、以下のリソースを参照してください。

- Amazon DynamoDB のコンプライアンス検証
- 責任共有モデル
- アマゾン ウェブ サービス: セキュリティプロセスの概要 (ホワイトペーパー)

以下のセキュリティのベストプラクティスも Amazon DynamoDB での設定と脆弱性の分析に対処します。

- を使用した DynamoDB コンプライアンスのモニタリングAWS Config ルール
- を使用した DynamoDB 設定のモニタリングAWS Config

Amazon DynamoDB のセキュリティベストプラクティス

Amazon DynamoDB には、独自のセキュリティポリシーを策定および実装する際に考慮すべきさまざまなセキュリティ機能が用意されています。以下のベストプラクティスは一般的なガイドラインであり、完全なセキュリティソリューションに相当するものではありません。これらのベストプラクティスはお客様の環境に適切ではないか、十分ではない場合があるため、これらは処方箋ではなく、有用な考慮事項と考えてください。

トピック

- DynamoDB での予防的セキュリティのベストプラクティス
- DynamoDB Detective によるセキュリティのベストプラクティス

DynamoDB での予防的セキュリティのベストプラクティス

以下のベストプラクティスは、Amazon DynamoDB のセキュリティインシデントの予防に役立ちます。

保管時の暗号化

DynamoDB は、AWS Key Management Service (AWS KMS) に保存されているテーブル、インデックス、ストリーミング、Backup に保存されているすべてのユーザーデータを保存時に暗号化します。この機能は、基になるストレージへの不正アクセスからデータを保護することによって、データ保護の追加レイヤーを提供します。

DynamoDB でユーザーデータを暗号化するために AWS 所有のキー (デフォルトの暗号化タイプ)、AWS マネージドキー を使用するか、カスタマーマネージドキーを使用するかを指定できます。詳細については、「保管時の Amazon DynamoDB 暗号化」を参照してください。

IAM ロールを使用した DynamoDB へのアクセス認証

ユーザー、アプリケーション、およびその他の AWS サービスが DynamoDB にアクセスするには、有効な AWS 認証情報が AWS API リクエストに含まれている必要があります。AWS 認証情報を、アプリケーションまたは EC2 インスタンスに直接保存しないことをお勧めします。自動更新されない長期認証情報条件のため、漏洩すると業務に深刻な悪影響が及ぶ可能性があります。IAM ロールでは、AWS サービスおよびリソースにアクセスするために使用できる一時的なアクセスキーを有効にすることができます。

詳細については、「Amazon DynamoDB の Identity and Access Management」を参照してください。

IAM ポリシーを使用した DynamoDB ベース認可

許可を付与する場合、許可を取得するユーザー、取得する許可の対象となる DynamoDB API、およびそれらのリソースに対して許可される特定のアクションを決定します。最小限の特権の実装は、セキュリティリスクはもちろん、エラーや悪意ある行動によってもたらされる可能性のある影響を減らす上での鍵となります。

IAM アイデンティティ (ユーザー、グループ、ロール) にアクセス権限ポリシーをアタッチし、DynamoDB リソースでオペレーションを実行する許可を付与します。

これを行うには、次を使用します。

- AWS 管理 (事前定義) ポリシー
- カスタマー管理ポリシー

詳細に設定されたアクセスコントロールのための IAM ポリシー条件を使用する

DynamoDB でアクセス権限を付与するときは、アクセス権限ポリシーを有効にする方法を決める条件を指定できます。最小限の特権の実装は、セキュリティリスクはもちろん、エラーや悪意ある行動によってもたらされる可能性のある影響を減らす上での鍵となります。

IAM ポリシーを使用して、アクセス許可を付与するときに条件を指定できます。例えば、次の操作を実行できます。

- テーブルまたはセカンダリインデックス内の項目と属性に対する読み込み専用アクセスをユーザーに許可する許可を付与します。
- ユーザーのアイデンティティに基づいて、テーブルの特定の属性への書き込み専用アクセスのアクセス権限をユーザーに付与します。

詳細については、「詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用」を参照してください。

VPC エンドポイントとポリシーを使用した DynamoDB へのアクセス

仮想プライベートクラウド (VPC) 内からのみ DynamoDB にアクセスする必要がある場合は、VPC エンドポイントを使用して、必要な VPC からのみのアクセスに制限することをお勧めします。これにより、トラフィックがオープンインターネットを通過してその環境に晒されるのを防ぐことができます。

DynamoDB の VPC エンドポイントを使用すると、以下を使用してアクセスを制御および制限できます。

- VPC エンドポイントポリシー — これらのポリシーは DynamoDB VPC エンドポイントに適用されます。DynamoDB テーブルへの API アクセスを制御および制限が許可されます。
- IAM ポリシー — ユーザー、グループ、またはロールにアタッチされたポリシーで `aws:sourceVpce` 条件を使用すると、DynamoDB テーブルへのすべてのアクセスが指定の VPC エンドポイントを経由するよう強制できます。

詳細については、「Amazon DynamoDB のエンドポイント」を参照してください。

クライアント側の暗号化を考慮する

DynamoDB にテーブルを実装する前に、暗号化戦略を計画することをお勧めします。機密データまたは機密データを DynamoDB に保存する場合は、クライアント側の暗号化をプランに含めることを検討してください。これにより、データをできるだけ送信元に近い状態で暗号化し、ライフサイクル全体にわたってデータを確実に保護できます。転送中および保管時の機密データを暗号化することで、サードパーティーがプレーンテキストデータを使用できないようにします。

DynamoDB 用の AWS Database Encryption SDK は、DynamoDB に送信する前にテーブルデータを保護するのに役立つソフトウェアライブラリです。DynamoDB テーブル項目を暗号化、署名、検証、復号します。どの属性を暗号化して署名するかを制御できます。

プライマリキーに関する考慮事項

テーブルやグローバルセカンダリインデックスでは、プライマリキーに機密性の高い名前やプレーンテキストデータを使用しないでください。キー名はテーブル定義に表示されます。例えば、プライマリキー名は、DescribeTable を呼び出すアクセス許可を持つすべてのユーザーがアクセスできます。キー値は、AWS CloudTrail やその他のログに表示される場合があります。さらに、DynamoDB はデータの配信とリクエストのルーティングにキー値を使用します。AWS 管理者は、この値を観察してサービスのヘルスを維持する場合があります。

テーブルや GSI のキー値に機密データを使用する必要がある場合は、エンドツーエンドのクライアント暗号化を使用することをお勧めします。エンドツーエンドのクライアント暗号化により、データへのキー値の参照を実行する際に、データが暗号化されないまま DynamoDB 関連ログに表示されることを回避できます。これを実現する 1 つの方法は、AWS Database Encryption SDK for DynamoDB を使用することですが、これは必須ではありません。独自のソリューションを使用する場合は、十分に安全な暗号化アルゴリズムを常に使用してください。ハッシュなどの非暗号化オプションは、ほとんどの状況で十分に安全とは見なされないため、使用しないでください。

プライマリキーの名前が機密である場合は、代わりに `pk` と `sk` を使用することをお勧めします。これは、パーティションキーの設計を柔軟に保つ一般的なベストプラクティスです。

どれが適切な選択肢であるか不安な場合は、必ずセキュリティエキスパートまたは AWS アカウントチームに相談してください。

DynamoDB Detective によるセキュリティのベストプラクティス

以下は、潜在的なセキュリティ上の弱点とインシデントを検出するために役立つ Amazon DynamoDB でのベストプラクティスです。

AWS CloudTrail を使用した AWS マネージド KMS キーの使用状況モニタリング

保管時の暗号化に AWS マネージドキー を使用している場合、このキーの使用状況が AWS CloudTrail に記録されます。CloudTrail は、アカウントで実行されたアクションをレコードすることで、ユーザーのアクティビティを可視化します。CloudTrail は、リクエストを行ったユーザー、使用されたサービス、実行されたアクション、アクションのパラメータ、AWS のサービスから返されたレスポンス要素など、各アクションに関する重要な情報をレコードします。この

情報は、AWS リソースに加えられた変更を追跡し、オペレーション上の問題をトラブルシューティングするサポートになります。CloudTrail を使用すると、社内ポリシーや規制スタンダードへのコンプライアンスが容易になります。

CloudTrail を使用して、キーの使用状況を監査できます。CloudTrail は、アカウントの AWS API コールおよび関連イベントの履歴を含むログファイルを作成します。これらのログファイルには、統合された AWS サービスを通じて行われたものに加えて、AWS マネジメントコンソール、AWS SDK、およびコマンドラインツールを使用して行われたすべての AWS KMS API リクエストが含まれます。これらのログファイルを使って、KMS が使用された日時、リクエストされたオペレーション、リクエスタの ID、リクエスト元の IP アドレスなどについての情報を取得できます。詳細については、「AWS CloudTrail を使用した AWS KMS API 呼び出しのログ記録」と「AWS CloudTrail ユーザーガイド」を参照してください。

CloudTrail を使用した DynamoDB オペレーションのモニタリング

CloudTrail は、コントロールプレーンイベントとデータプレーンイベントの両方をモニタリングできます。コントロールプレーンのオペレーションでは、DynamoDB テーブルを作成および管理できます。また、インデックス、ストリーム、およびテーブルに依存する他のオブジェクトを操作できます。データプレーンオペレーションでは、テーブルのデータで、作成、読み込み、更新、および削除 (CRUD とも呼ばれる) アクションを実行できます。一部のデータプレーンオペレーションでも、セカンダリインデックスからデータを読み込むことができます。CloudTrail でデータプレーンイベントのログを有効にするには、CloudTrail でデータプレーン API アクティビティのログを有効にする必要があります。詳細については、「トレイルのデータイベントのログ記録」を参照してください。

DynamoDB でアクティビティが発生すると、そのアクティビティはイベント履歴の他の AWS のサービスのイベントと共に CloudTrail イベントにレコードされます。詳細については、「AWS CloudTrail を使用した DynamoDB オペレーションのログ記録」を参照してください。最近のイベントは、AWS アカウントで表示、検索、ダウンロードできます。詳細については、AWS CloudTrailユーザーガイドの「CloudTrail イベント履歴でのイベントの表示」を参照してください。

DynamoDB のイベントなど、AWS アカウントのイベントの継続的なレコードについては、追跡を作成します。証跡により、CloudTrail はログファイルを Amazon Simple Storage Service (Amazon S3) バケットに配信できます。デフォルトでは、コンソールで証跡を作成すると、証跡がすべての AWS リージョンに適用されます。証跡では、AWS パーティションのすべてのリージョンからのイベントがログに記録され、指定した S3 バケットにログファイルが配信されます。さらに、その他の AWS サービスを設定して、CloudTrail ログで収集したデータをより詳細に分析し、それに基づく対応を行うことができます。

DynamoDB Streams を使用してデータプレーンオペレーションをモニタリングする

DynamoDB は AWS Lambda と統合されているため、トリガー (DynamoDB Streams 内のイベントに自動的に応答するコードの一部) を作成できます。トリガーを使用すると、DynamoDB テーブル内のデータ変更に対応するアプリケーションを構築できます。

テーブルで DynamoDB Streams を有効にした場合、書き込む Lambda 関数にストリームの Amazon リソースネーム (ARN) を関連付けることができます。テーブルの項目が変更されるとすぐに、新しいレコードがテーブルのストリーミングに表示されます。AWS Lambda はストリーミングをポーリングし、新しいストリーミングレコードを検出すると、 Lambda 関数を同期的に呼び出します。Lambda 関数は、通知の送信やワークフローの開始など、指定したアクションを実行できます。

例については、「チュートリアル: Amazon DynamoDB Streams で AWS Lambda を使用する」を参照してください。この例では、DynamoDB イベント入力を受け取り、それに含まれるメッセージを処理して、受信イベントデータの一部を Amazon CloudWatch Logs に書き込みます。

を使用した での DynamoDB 設定のモニタリングAWS Config

AWS Config を使用すると、AWS リソースの設定変更を継続的にモニタリングおよびレコードできます。また、AWS Config を使用して AWS リソースのインベントリも作成できます。以前の状態からの変更が検出されると、Amazon Simple Notification Service (Amazon SNS) 通知が配信されるので、内容を確認してアクションを実行できます。「コンソールを使用して AWS Config を設定する」のガイダンスに従い、DynamoDB リソースタイプが含まれていることを確認します。

設定変更と通知を Amazon SNS トピックにストリーミングするように AWS Config を設定できます。例えば、リソースの更新時に E メールアドレスに通知が送信されれば、変更を確認できます。AWS Config でカスタムルールまたはマネージドルールを適用してリソースを評価したときにも通知を受け取ることができます。

例については、AWS Config デベロッパーガイドの「AWS Config が Amazon SNS トピックに送信する通知」を参照してください。

AWS Config ルール を使用した での DynamoDB コンプライアンスのモニタリング

AWS Config は、リソースで発生する設定変更を継続的に追跡し、これらの変更がルールの条件に違反していないかどうかをチェックします。リソースがルールに違反している場合、AWS Config はリソースとルールに非準拠を示す [noncompliant] のフラグを付けます。

AWS Config を使用してリソースの設定を評価することで、自社プラクティス、業界ガイドライン、および規制に対するリソース設定の準拠状態を確認できます。AWS Config は、AWS マ

ネージドルール]を提供しますが、これは事前定義されているカスタマイズ可能なルールで、AWS Config はこのルールを使用して AWS リソースが一般的なベストプラクティスに準拠しているかどうかを評価します。

識別とオートメーションのために DynamoDB リソースにタグを付ける

AWS のリソースにメタデータをタグ付け形式で割り当てることができます。各タグは、カスタマー定義のキーと値 (オプション) で構成されるシンプルなラベルです。タグを使用すると、リソースの管理、検索、フィルターが容易になります。

タグ付けを行うと、グループ化されたコントロールを実装できます。タグには固有のタイプはありませんが、 用途、所有者、環境などの基準でリソースを分類できます。次に例をいくつか示します。

- セキュリティ — 暗号化などの要件を決定するために使用されます。
- 機密性 — リソースがサポートするデータ機密性レベルの識別子。
- 環境 — 開発、テスト、本番稼働用インフラストラクチャを区別するために使用されます。

詳細については、「AWS タグ付け戦略」および「DynamoDB のタグ付け」を参照してください。

AWS Security Hub CSPM を使用して、セキュリティのベストプラクティスに関連する Amazon DynamoDB の使用状況をモニタリングします。

Security Hub CSPM は、セキュリティコントロールを使用してリソース設定とセキュリティ標準を評価し、お客様がさまざまなコンプライアンスフレームワークに準拠できるようサポートします。

Security Hub CSPM を使用して DynamoDB リソースを評価する方法の詳細については、「AWS Security Hub CSPM ユーザーガイド」の「Amazon DynamoDB コントロール」を参照してください。

DynamoDB でのモニタリングとログ記録

モニタリングは、DynamoDB および AWS ソリューションの信頼性、可用性、パフォーマンスを維持する上で重要な部分です。マルチポイント障害を簡単にデバッグできるように、AWS ソリューションのすべての部分からモニタリングデータを収集する必要があります。

トピック

- モニタリング計画
- パフォーマンスのベースライン
- 統合サービス
- 自動モニタリングツール
- Amazon CloudWatch で DynamoDB のメトリクスをモニタリングする
- AWS CloudTrail を使用して DynamoDB オペレーションをログに記録する
- CloudWatch Contributor Insights for DynamoDB を使用してデータアクセスを分析する

モニタリング計画

DynamoDB のモニタリングを開始する前に、以下の質問に対する回答を反映したモニタリング計画を作成します。

- どのような目的でモニタリングしますか?
- どのリソースをモニタリングしますか?
- どのくらいの頻度でこれらのリソースをモニタリングしますか?
- どのモニタリングツールを利用しますか?
- 誰がモニタリングタスクを実行しますか?
- 問題が発生したときに誰が通知を受け取りますか?

パフォーマンスのベースライン

さまざまなタイミングと負荷条件でパフォーマンスを測定することにより、使用環境における通常の DynamoDB パフォーマンスのベースラインを確定します。DynamoDB をモニタリングするには、モニタリングの履歴データを保存することを検討します。保存データを最新のパフォーマンスデータと比較するベースラインとして使用し、通常のパフォーマンスのパターンやパフォーマンスの異常を検

出して、問題への対応を検討することができます。ベースラインを確立するには、少なくとも、次の項目をモニタリングする必要があります。

- 指定された期間に消費された読み込みまたは書き込み容量ユニットの数。プロビジョニングされたスループットの使用量を追跡できます。
- テーブルにプロビジョニングされた書き込みまたは読み込み容量を指定期間中に超えたリクエストで、テーブルにプロビジョニングされたスループットクォータをどのリクエストが超えたかを判断できます。
- エラー。リクエストがエラーになったかどうかを判断できます。

統合サービス

DynamoDB は、ユーザーに代わってテーブルを自動的にモニタリングし、Amazon CloudWatch を介してメトリクスを報告します。さらに、以下の AWS のサービスとの統合により、DynamoDB リソースのモニタリングとトラブルシューティングも促進します。

- AWS CloudTrail は、AWS アカウント により、またはそのアカウントに代わって行われた API コールおよび関連イベントを取得し、指定した Amazon S3 バケットにログファイルを配信します。詳細については、「AWS CloudTrail を使用して DynamoDB オペレーションを口グに記録する」を参照してください。
- Contributor Insights は、テーブルやインデックス内で最も頻繁にアクセスおよびスロットリングされるキーをすばやく特定するための診断ツールです。詳細については、「CloudWatch Contributor Insights for DynamoDB を使用してデータアクセスを分析する」を参照してください。

自動モニタリングツール

AWS は、DynamoDB のモニタリングに使用できるさまざまなツールを提供しています。モニタリングタスクをできるだけ自動化することをお勧めします。以下の自動化されたモニタリングツールを使用して、DynamoDB を監視し、問題が発生したときにレポートできます。

- Amazon CloudWatch のアラーム – 単一のメトリクスを指定した期間モニタリングし、特定のしきい値に対する複数の期間にわたるメトリクスの値に基づいて、1 つ以上のアクションを実行します。

  アクションは、Amazon Simple Notification Service (Amazon SNS) のトピックまたは Amazon EC2 Auto Scaling のポリシーに送信される通知です。Amazon CloudWatch のアラームは、メトリ

クスが特定の状態にあるだけではアクションを呼び出しません。アクションを呼び出すには、指定した期間継続している必要があります。詳細については、「Amazon CloudWatch で DynamoDB のメトリクスをモニタリングする」を参照してください。

- AWS CloudTrail ログのモニタリング - アカウント間でログファイルを共有し、AWS CloudTrail ログファイルを AWS CloudTrail Logs に送信してリアルタイムでモニタリングします。また、ログ処理アプリケーションを Java で作成し、AWS CloudTrail による配信後にログファイルが変更されていないことを検証します。詳細については、「AWS CloudTrail ユーザーガイド」の「Amazon CloudWatch Logs とは」を参照してください。

Amazon CloudWatch で DynamoDB のメトリクスをモニタリングする

CloudWatch を使用して DynamoDB をモニタリングすることで、DynamoDB から raw データを収集し、リアルタイムに近い読み込み可能なメトリクスに加工することができます。これらの統計は一定期間保持されるため、履歴情報にアクセスしてウェブアプリケーションやサービスの動作をより的確に把握できます。デフォルトでは、DynamoDB メトリクスデータは CloudWatch に自動的に送信されます。詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon CloudWatch とは」および「メトリクスの保持」を参照してください。

トピック

- DynamoDB メトリクスの使用方法
- CloudWatch コンソールでのメトリクスの表示
- AWS CLI でのメトリクスの表示
- DynamoDB のメトリクスとディメンション
- DynamoDB に CloudWatch アラームを作成する

DynamoDB メトリクスの使用方法

DynamoDB によってレポートされるメトリクスが提供する情報は、さまざまな方法で分析できます。以下のリストは、メトリクスの一般的な利用方法をいくつか示しています。ここで紹介するのは開始するための提案事項です。すべてを網羅しているわけではありません。

DynamoDB メトリクスの使用方法

目的	関連するメトリクス
テーブルの TTL 削除率は、どのようにしてモニタリングできますか?	指定した期間にわたって TimeToLiveDeletedItemCount をモニタリングし、テーブルの TTL 削除率を追跡できます。TimeToLiveDeletedItemCount メトリクスを使用するサーバーレスアプリケーションの例については、「Automatically archive items to S3 using DynamoDB time to live (TTL) with AWS Lambda and Amazon Data Firehose」を参照してください。
プロビジョニングされたスループットの消費量は、どのようにして確認できますか?	指定した期間 ConsumedReadCapacityUnits または ConsumedWriteCapacityUnits をモニタリングし、プロビジョニング済みスループットの使用量を追跡できます。
どのリクエストがテーブルのプロビジョニングされたスループットクォータを超えているかは、どのようにして確認できますか?	ThrottledRequests リクエスト内で任意のイベントがプロビジョニングされたスループットクォータを超過した場合、は 1 ずつ増加します。次に、どのイベントがリクエストをスロットリングしているかについてのインサイトを取得するには、ThrottledRequests とテーブルおよびそのインデックスの ReadThrottleEvents メトリクスと WriteThrottleEvents メトリクスを比較します。
どうすればシステムエラーの発生の有無を判断できますか。	SystemErrors をモニタリングして HTTP 500 (サーバーエラー) コードを発生させたリクエストがあるかどうかを判断できます。DynamoDB などの分散システムでは、ときどき HTTP 500 エラーが発生することがありますが、影響を受けたリクエストを再試行できます。システムエラーが長期間続く場合は、AWS サポート (https://aws.amazon.com/support) にお問い合わせください。
テーブルオペレーションのレイテンシー値は、どのようにしてモニタリングできますか?	パーセンタイルメトリクス (p50) を通じて平均レイテンシーとレイテンシー中央値を追跡することで、SuccessfulRequestLatency をモニタリングできます。時折レイテンシーが急上昇しても問題ありません。ただし、平均レイテンシーまたは p50 (中央値) が高い場合は、解決すべき根本的な問題が存在する可能性があります。詳細については「Amazon

目的	関連するメトリクス
	DynamoDB でのレイテンシーの問題のトラブルシューティング」を参照してください。

CloudWatch コンソールでのメトリクスの表示

メトリクスは、まずサービスの名前空間ごとにグループ化されます。次に各名前空間内でさまざまなディメンションの組み合わせごとにグループ化されます。

CloudWatch コンソールでメトリクスを表示する

1. CloudWatch コンソールの https://console.aws.amazon.com/cloudwatch/ を開いてください。
2. ナビゲーションペインで、[メトリクス]、[すべてのメトリクス] の順に選択します。
3. DynamoDB 名前空間を選択します。また、Usage (使用状況) 名前空間を選択して、DynamoDB の使用状況メトリクスを表示できます。使用状況メトリクスの詳細については、「AWS 使用状況メトリクス」を参照してください。
4. [参照] タブには、名前空間内のすべてのメトリクスが表示されます。
5. (オプション) メトリクスグラフを CloudWatch ダッシュボードに追加するには、[アクション]、[ダッシュボードに追加] の順に選択します。

AWS CLI でのメトリクスの表示

AWS CLI を使用してメトリクス情報を取得するには、CloudWatch の `list-metrics` コマンドを使用します。次の例では、`AWS/DynamoDB` 名前空間にすべてのメトリクスがリストされています。

```
aws cloudwatch list-metrics --namespace "AWS/DynamoDB"
```

メトリクスの統計情報を取得するには、 コマンドを使用します`get-metric-statistics` 次のコマンドは、特定の 24 時間にわたるテーブル `ProductCatalog` の `ConsumedReadCapacityUnits` 統計情報を、5 分間隔で取得します。

```
aws cloudwatch get-metric-statistics –namespace AWS/DynamoDB \
     –metric-name ConsumedReadCapacityUnits \
     –start-time 2023-11-01T00:00:00Z \
```

```
    --end-time 2023-11-02T00:00:00Z \
    --period 360 \
    --statistics Average \
    --dimensions Name=TableName,Value=ProductCatalog
```

出力例は次のとおりです。

```
{
    "Datapoints": [
        {
            "Timestamp": "2023-11-01T 09:18:00+00:00",
            "Average": 20,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-11-01T 04:36:00+00:00",
            "Average": 22.5,
            "Unit": "Count"
        },
        {
            "Timestamp": "2023-11-01T 15:12:00+00:00",
            "Average": 20,
            "Unit": "Count"
        }, ...
        {
            "Timestamp": "2023-11-01T 17:30:00+00:00",
            "Average": 25,
            "Unit": "Count"
        }
    ],
    "Label": " ConsumedReadCapacityUnits "
}
```

DynamoDB のメトリクスとディメンション

DynamoDB を操作すると、メトリクスとディメンションが CloudWatch に送信されます。

DynamoDB は、消費されたプロビジョンドスループットを 1 分間出力します。消費したキャパシティが、設定したターゲット使用率を 2 分間連続して超過すると、自動スケーリングがトリガーされます。CloudWatch アラームは、自動スケーリングをトリガーする前に、最大数分の短い遅延を伴う場合があります。この遅延により、CloudWatch メトリクスの正確な評価が保証されます。消費されたスループットのスパイク間隔が 1 分を超えると、自動スケーリングはトリガーされない場

合があります。同様に、15 個の連続するデータポイントがターゲット使用率を下回ると、スケールダウンイベントが発生する場合があります。いずれの場合も、自動スケーリングのトリガー後に、UpdateTable API が呼び出されます。テーブルやインデックスのプロビジョンドキャパシティの更新には、数分かかる場合があります。この間に、テーブルの前のプロビジョンドキャパシティを超えるリクエストはスロットリングされます。

メトリクスおよびディメンションの表示

CloudWatch は、以下の DynamoDB のメトリクスを表示します。

DynamoDB のメトリクス

> **Note**
>
> Amazon CloudWatch は 1 分間隔でこれらのメトリクスを集計します。

- ConditionalCheckFailedRequests
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ReadAccountLimitThrottleEvents
- ReadKeyRangeThroughputThrottleEvents
- ReadMaxOnDemandThroughputThrottleEvents
- ReadProvisionedThroughputThrottleEvents
- ReadThrottleEvents
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteAccountLimitThrottleEvents
- WriteKeyRangeThroughputThrottleEvents

- `WriteMaxOnDemandThroughputThrottleEvents`
- `WriteProvisionedThroughputThrottleEvents`
- `WriteThrottleEvents`
- `FaultInjectionServiceInducedErrors`

その他のすべての DynamoDB メトリクスでは、集計の間隔は 5 分です。

Average や Sum など、すべての統計が必ずしも常にすべてのメトリクスに適用可能であるとは限りません。ただし、これらの値はすべて Amazon DynamoDB コンソール経由で利用できます。また、すべてのメトリクスで CloudWatch コンソール、AWS CLI、AWS SDK を使用することによっても利用できます。

次のリストは、各メトリクスに適用可能な有効な統計のセットを示します。

利用可能なメトリクスのリスト

- AccountMaxReads
- AccountMaxTableLevelReads
- AccountMaxTableLevelWrites
- AccountMaxWrites
- AccountProvisionedReadCapacityUtilization
- AccountProvisionedWriteCapacityUtilization
- AgeOfOldestUnreplicatedRecord
- 条件チェックが失敗したリクエスト
- ConsumedChangeDataCaptureUnits
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- FailedToReplicateRecordCount
- MaxProvisionedTableReadCapacityUtilization
- MaxProvisionedTableWriteCapacityUtilization
- OnDemandMaxReadRequestUnits
- OnDemandMaxWriteRequestUnits

- OnlineIndexConsumedWriteCapacity
- OnlineIndexPercentageProgress
- OnlineIndexThrottleEvents
- PendingReplicationCount
- ProvisionedReadCapacityUnits
- ProvisionedWriteCapacityUnits
- ReadAccountLimitThrottleEvents
- ReadKeyRangeThroughputThrottleEvents
- ReadMaxOnDemandThroughputThrottleEvents
- ReadProvisionedThroughputThrottleEvents
- ReadThrottleEvents
- ReplicationLatency
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledPutRecordCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteAccountLimitThrottleEvents
- WriteKeyRangeThroughputThrottleEvents
- WriteMaxOnDemandThroughputThrottleEvents
- WriteProvisionedThroughputThrottleEvents
- WriteThrottleEvents
- 使用状況メトリクス
- FaultInjectionServiceInducedErrors

- VectorSearchRequestBytes
- VectorWriteRequestBytes

AccountMaxReads

アカウントで使用できる読み込み容量ユニットの最大数。この制限は、オンデマンドテーブルやグローバルセカンダリインデックスには適用されません。

単位: `Count`

有効な統計:

- `Maximum` — アカウントで使用できる読み込み容量ユニットの最大数。

AccountMaxTableLevelReads

アカウントのテーブルまたはグローバルセカンダリインデックスで使用できる読み込み容量ユニットの最大数。オンデマンドテーブルの場合、この制限は、テーブルやグローバルセカンダリインデックスで使用できる読み込みリクエストユニットの上限数を示します。

単位: `Count`

有効な統計:

- `Maximum` — アカウントのテーブルまたはグローバルセカンダリインデックスで使用できる読み込み容量ユニットの最大数。

AccountMaxTableLevelWrites

アカウントのテーブルまたはグローバルセカンダリインデックスで使用できる書き込み容量ユニットの最大数。オンデマンドテーブルの場合、この制限は、テーブルやグローバルセカンダリインデックスで使用できる書き込みリクエストユニットの上限数を示します。

単位: `Count`

有効な統計:

- `Maximum` — アカウントのテーブルまたはグローバルセカンダリインデックスで使用できる書き込み容量ユニットの最大数。

AccountMaxWrites

アカウントで使用できる書き込み容量ユニットの最大数。この制限は、オンデマンドテーブルやグローバルセカンダリインデックスには適用されません。

単位: Count

有効な統計:

- Maximum - アカウントで使用できる書き込み容量ユニットの最大数。

AccountProvisionedReadCapacityUtilization

アカウントで利用されるプロビジョニング済み読み込み容量ユニットの割合。

単位: Percent

有効な統計:

- Maximum — アカウントで使用されるプロビジョニング済み読み込み容量ユニットの最大割合。
- Minimum — アカウントで使用されるプロビジョニング済み読み込み容量ユニットの最小割合。
- Average — アカウントで使用されるプロビジョニング済み読み込み容量ユニットの平均割合。メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み読み込み容量ユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

AccountProvisionedWriteCapacityUtilization

アカウントで利用されるプロビジョニング済み書き込み容量ユニットの割合。

単位: Percent

有効な統計:

- Maximum - アカウントで利用されるプロビジョニング済み書き込み容量ユニットの最大割合。
- Minimum — アカウントで使用されるプロビジョニング済み読み込み容量ユニットの最小割合。
- Average — アカウントで使用されるプロビジョニング済み書き込み容量ユニットの平均割合。メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み書き込み容量ユニットをすばやく調整すると、この統計には実際の平均値が反映されないことがあります。

AgeOfOldestUnreplicatedRecord

Kinesis データストリームにレプリケートされていないレコードからの経過時間が DynamoDB テーブルに最初に出現してからの経過時間。

単位: `Milliseconds`

ディメンション: `TableName, DelegatedOperation`

有効な統計:

- `Maximum`.
- `Minimum`.
- `Average`.

条件チェックが失敗したリクエスト

条件付き書き込みの実行に失敗した回数。`PutItem`、`UpdateItem`、および `DeleteItem` オペレーションを使用すると、オペレーションを続行する前に true と評価される必要がある論理条件を指定できます。この条件が false に評価される場合は、`ConditionalCheckFailedRequests` は 1 つ増加します。`ConditionalCheckFailedRequests` も、論理条件が提供され、その条件が false に評価される PartiQL Update および Delete ステートメントで 1 つ増加します。

> **Note**
>
> 条件付き書き込みに失敗すると HTTP 400 エラー (Bad Request) が発生します。これらのイベントは `ConditionalCheckFailedRequests` メトリクスに反映されますが、`UserErrors` メトリクスには反映されません。

単位: `Count`

ディメンション: `TableName`

有効な統計:

- `Minimum`
- `Maximum`
- `Average`

- SampleCount
- Sum

ConsumedChangeDataCaptureUnits

消費された変更データキャプチャユニットの数。

単位: Count

ディメンション: TableName, DelegatedOperation

有効な統計:

- Minimum
- Maximum
- Average

ConsumedReadCapacityUnits

プロビジョンドキャパシティとオンデマンドキャパシティの両方について、指定された期間内に消費された読み取りキャパシティユニットの数。これにより、使用されたスループットの量を追跡できます。テーブルとそのすべてのグローバルセカンダリインデックス、または特定のグローバルセカンダリインデックスの消費された読み込み容量の合計を取得できます。詳細については、「読み込み/書き込み容量モード」を参照してください。

TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの ConsumedReadCapacityUnits を返します。グローバルセカンダリインデックスの ConsumedReadCapacityUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

> **Note**
>
> つまり、キャパシティ消費量の急増が 1 秒間だけであれば、CloudWatch グラフには正確に反映されず、該当する 1 分間の消費率は低く見える可能性があります。
> Sum 統計を使用して、消費されたスループットを計算します。例えば、1 分間の Sum 値を取得し、1 分間の秒数 (60) で除算して 1 秒あたりの平均 ConsumedReadCapacityUnits を計算します。計算された値と、DynamoDB が提供するプロビジョニング済みスループット値を比較できます。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName

有効な統計:

- Minimum — テーブルまたはインデックスへの個々のリクエストによって消費される読み込み容量ユニットの最小数。
- Maximum — テーブルまたはインデックスへの個々のリクエストによって消費される読み込み容量ユニットの最大数。
- Average — 消費されたリクエストごとの平均読み込み容量。

> **Note**
>
> Average 値は、サンプル値がゼロになる非活動期間によって影響を受けます。

- Sum — 消費された読み込み容量ユニットの合計。これは、ConsumedReadCapacityUnits メトリクスの最も有用な統計です。
- SampleCount – メトリクスが出力される頻度を表します。トラフィックがゼロのテーブルでも、SampleCount は定期的に出力されます。ただし、サンプル値は常にゼロになります。

> **Note**
>
> SampleCount 値は、サンプル値がゼロになる非活動期間によって影響を受けます。

ConsumedWriteCapacityUnits

プロビジョンドキャパシティとオンデマンドキャパシティの両方について、指定された期間内に消費された書き込みキャパシティユニットの数。これにより、使用されたスループットの量を追跡できます。テーブルとそのすべてのグローバルセカンダリインデックス、または特定のグローバルセカンダリインデックスの消費された書き込み容量の合計を取得できます。詳細については、「読み込み/書き込み容量モード」を参照してください。

TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの ConsumedWriteCapacityUnits を返します。グローバルセカンダリインデックスの ConsumedWriteCapacityUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。Source ディメンション

は、Customer と GlobalTable の 2 つの値のいずれかを返すことができます。レプリケートされた書き込みにはソース GlobalTable の ConsumedWriteCapacityUnits が設定されますが、リージョンテーブルの書き込みはソース Customer の ConsumedWriteCapacityUnits が設定されます。

> **Note**
>
> Sum 統計を使用して、消費されたスループットを計算します。例えば、Sum 値を 1 分にわたって取得し、1 分間の秒数 (60) で除算して 1 秒あたりの平均 ConsumedWriteCapacityUnits を計算します (この平均では 1 分の間に書き込みアクティビティで発生した大きくて短いスパイクは強調されないことを認識します)。計算された値と、DynamoDB が提供するプロビジョニング済みスループット値を比較できます。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName, Source

有効な統計:

- Minimum — テーブルまたはインデックスへの個々のリクエストによって消費される書き込み容量ユニットの最小数。
- Maximum — テーブルまたはインデックスへの個々のリクエストによって消費される書き込み容量ユニットの最大数。
- Average — 消費されたリクエストごとの平均書き込み容量。

  > **Note**
  >
  > Average 値は、サンプル値がゼロになる非活動期間によって影響を受けます。

- Sum — 消費された書き込み容量ユニットの合計。これは、ConsumedWriteCapacityUnits メトリクスの最も有用な統計です。
- SampleCount – メトリクスが出力される頻度を表します。トラフィックがゼロのテーブルでも、SampleCount は定期的に出力されます。ただし、サンプル値は常にゼロになります。

  > **Note**
  >
  > SampleCount - 値は、サンプル値がゼロになる非活動期間によって影響を受けます。

FailedToReplicateRecordCount

DynamoDB が Kinesis Data Streams にレプリケートできなかったレコードの数。

単位: Count

ディメンション: TableName、DelegatedOperation

有効な統計:

- Sum

MaxProvisionedTableReadCapacityUtilization

プロビジョニング済み読み込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み読み込みテーブルまたはグローバルセカンダリインデックスが使用している割合。

単位: Percent

有効な統計:

- Maximum – プロビジョニング済み読み込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み読み込みテーブルまたはグローバルセカンダリインデックスが使用している割合の最大値。
- Minimum – プロビジョニング済み読み込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み読み込みテーブルまたはグローバルセカンダリインデックスが使用している割合の最小値。
- Average — プロビジョニング済み書き込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み書き込みテーブルまたはグローバルセカンダリインデックスが使用している割合の平均。メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み読み込み容量ユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

MaxProvisionedTableWriteCapacityUtilization

プロビジョニング済み書き込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み書き込みテーブルまたはグローバルセカンダリインデックスが使用している割合。

単位: Percent

有効な統計:

- Maximum — プロビジョニング済み書き込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み書き込みテーブルまたはグローバルセカンダリインデックスが使用している最大割合。
- Minimum — プロビジョニング済み書き込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み書き込みテーブルまたはグローバルセカンダリインデックスが使用している最小割合。
- Average — プロビジョニング済み書き込み容量ユニットのうち、アカウントの最も高いプロビジョニング済み書き込みテーブルまたはグローバルセカンダリインデックスが使用している平均割合。メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み書き込み容量ユニットをすばやく調整すると、この統計には実際の平均値が反映されないことがあります。

OnDemandMaxReadRequestUnits

テーブルまたはグローバルセカンダリインデックスに指定されたオンデマンド読み込みリクエストユニットの数。

テーブルの OnDemandMaxReadRequestUnits を表示するには、TableName を指定する必要があります。グローバルセカンダリインデックスの OnDemandMaxReadRequestUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: カウント

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Minimum – オンデマンド読み込みリクエストユニットの最小設定。UpdateTable を使用して読み込みリクエストユニットを増やす場合、このメトリクスは、この期間中のオンデマンド ReadRequestUnits の最小値を示します。
- Maximum – オンデマンド読み込みリクエストユニットの最大設定。UpdateTable を使用して読み込みリクエストユニットを減らす場合、このメトリクスは、この期間中のオンデマンド ReadRequestUnits の最大値を示します。
- Average – オンデマンド読み込みリクエストユニットの平均。OnDemandMaxReadRequestUnits メトリクスは 5 分間隔で発行されます。したがって、オンデマンド読み込みリクエストユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

OnDemandMaxWriteRequestUnits

テーブルまたはグローバルセカンダリインデックスに指定されたオンデマンド書き込みリクエストユニットの数。

テーブルの OnDemandMaxWriteRequestUnits を表示するには、TableName を指定する必要があります。グローバルセカンダリインデックスの OnDemandMaxWriteRequestUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Minimum – オンデマンド書き込みリクエストユニットの最小設定。UpdateTable を使用して書き込みリクエストユニットを増やす場合、このメトリクスは、この期間中のオンデマンド WriteRequestUnits の最小値を示します。
- Maximum – オンデマンド書き込みリクエストユニットの最大設定。UpdateTable を使用して書き込みリクエストユニットを減らす場合、このメトリクスは、この期間中のオンデマンド WriteRequestUnits の最大値を示します。
- Average – オンデマンド書き込みリクエストユニットの平均。OnDemandMaxWriteRequestUnits メトリクスは 5 分間隔で発行されます。したがって、オンデマンド書き込みリクエストユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

OnlineIndexConsumedWriteCapacity

このメトリクスは、インデックスのビルド中に 0 を示すことが予想されます。このメトリクスは以前は、新しいグローバルセカンダリインデックスをテーブルに追加するときに消費される書き込み容量ユニットの数を示していました。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName

有効な統計:

- Minimum

- Maximum
- Average
- SampleCount
- Sum

OnlineIndexPercentageProgress

新しいグローバルセカンダリインデックスがテーブルに追加されるときの完了率。DynamoDB は、まず新しいインデックスにリソースを割り当てて、次にテーブルの属性でインデックスを埋めていく必要があります。大きなテーブルの場合、この処理には長時間かかる場合があります。DynamoDB がインデックスを構築する際の相対的な進行状況を表示するには、この統計をモニタリングする必要があります。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName

有効な統計:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

OnlineIndexThrottleEvents

このメトリクスは、インデックスのビルド中に 0 を示すことが予想されます。このメトリクスは以前は、新しいグローバルセカンダリインデックスをテーブルに追加するときに発生する書き込みスロットリングイベントの数を示していました。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName

有効な統計:

- Minimum

- Maximum
- Average
- SampleCount
- Sum

PendingReplicationCount

グローバルテーブルバージョン 2017.11.29 (レガシー) のメトリクス (グローバルテーブルのみ)。1 つのレプリカテーブルに書き込まれていても、グローバルテーブル内の別のレプリカにはまだ書き込まれていない項目の更新の数。

単位: Count

ディメンション: TableName, ReceivingRegion

有効な統計:

- Average
- Sample Count
- Sum

ProvisionedReadCapacityUnits

テーブルまたはグローバルセカンダリインデックスのプロビジョニング済み読み込み容量ユニットの数。TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの ProvisionedReadCapacityUnits を返します。グローバルセカンダリインデックスの ProvisionedReadCapacityUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Minimum — プロビジョニング済み読み込み容量の最小設定。UpdateTable を使用して読み込み容量を増やす場合、このメトリクスは、この期間中のプロビジョニング済み ReadCapacityUnits の最小値を示します。

- Maximum — プロビジョニング済み読み込み容量の最大設定。UpdateTable を使用して読み込み容量を減らす場合、このメトリクスは、この期間中のプロビジョニング済み ReadCapacityUnits の最大値を示します。
- Average — プロビジョニング済み読み込み容量の平均。ProvisionedReadCapacityUnits メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み読み込み容量ユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

ProvisionedWriteCapacityUnits

テーブルまたはグローバルセカンダリインデックスのプロビジョニング済み書き込み容量ユニットの数。

TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの ProvisionedWriteCapacityUnits を返します。グローバルセカンダリインデックスの ProvisionedWriteCapacityUnits を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Minimum — プロビジョニング済み書き込み容量の最小設定。UpdateTable を使用して書き込み容量を増やす場合、このメトリクスは、この期間中のプロビジョニング済み WriteCapacityUnits の最小値を示します。
- Maximum — プロビジョニング済み書き込み容量の最大設定。UpdateTable を使用して書き込み容量を減らす場合、このメトリクスは、この期間中のプロビジョニング済み WriteCapacityUnits の最大値を示します。
- Average — プロビジョニング済み書き込み容量の平均。ProvisionedWriteCapacityUnits メトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み書き込み容量ユニットをすばやく調整すると、この統計には実際の平均値が反映されないことがあります。

ReadAccountLimitThrottleEvents

アカウント制限によりスロットリングされた読み取りリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

ReadKeyRangeThroughputThrottleEvents

パーティションの制限によりスロットリングされた読み取りリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

ReadMaxOnDemandThroughputThrottleEvents

オンデマンドの最大スループットによってスロットリングされた読み取りリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

ReadProvisionedThroughputThrottleEvents

プロビジョニングされたスループット制限によってスロットリングされた読み取りリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

ReadThrottleEvents

テーブルまたはグローバルセカンダリインデックス用にプロビジョニングされた読み込み容量ユニットを超える DynamoDB へのリクエスト。

1 つのリクエストで複数のイベントが発生する可能性があります。例えば、10 の項目を読み込む BatchGetItem は、10 個の GetItem イベントとして処理されます。各イベントでは、そのイベントがスロットリングされている場合、ReadThrottleEvents は 1 つ増加します。10 個すべての GetItem がスロットリングされない限り、BatchGetItem 全体の ThrottledRequests メトリクスは増加しません。

TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの ReadThrottleEvents を返します。グローバルセカンダリインデックスの ReadThrottleEvents を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- SampleCount
- Sum

ReplicationLatency

(このメトリクスは DynamoDB グローバルテーブル用です)。更新された項目が 1 つのレプリカテーブルの DynamoDB Streams に表示され、その項目がグローバルテーブルの別のレプリカに表示されるまでの経過時間。

単位: `Milliseconds`

ディメンション: `TableName, ReceivingRegion`

有効な統計:

- `Average`
- `Minimum`
- `Maximum`

ReturnedBytes

指定した期間中に `GetRecords` オペレーション (Amazon DynamoDB Streams) によって返されるバイト数。

単位: `Bytes`

ディメンション: `Operation, StreamLabel, TableName`

有効な統計:

- `Minimum`
- `Maximum`
- `Average`
- `SampleCount`
- `Sum`

ReturnedItemCount

指定した期間中に `Query`、`Scan`、または `ExecuteStatement` (選択) オペレーションによって返される項目の数。

返された項目の数は、評価された項目の数と必ずしも同じではありません。例えば、100 の項目があるテーブルまたはインデックス上の `Scan` をリクエストし、15 の項目だけが返されるように結果を

絞り込む FilterExpression を指定した場合を考えてみます。この場合、Scan からのレスポンスには、100 の ScanCount と 15 の返された項目の Count が含まれます。

単位: Count

ディメンション: TableName, Operation

有効な統計:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

ReturnedRecordsCount

指定した期間中に GetRecords オペレーション (Amazon DynamoDB Streams) によって返されるストリームレコードの数。

単位: Count

ディメンション: Operation, StreamLabel, TableName

有効な統計:

- Minimum
- Maximum
- Average
- SampleCount
- Sum

SuccessfulRequestLatency

指定した期間中に成功した DynamoDB または Amazon DynamoDB Streams へのリクエストのレイテンシー。SuccessfulRequestLatency は、次の 2 種類の異なる情報を提供できます。

- リクエストが成功するまでの経過時間 (Minimum、Maximum、Sum、Average、または Percentile)。

- 成功したリクエストの数 (SampleCount)。

SuccessfulRequestLatency は DynamoDB または Amazon DynamoDB Streams 内のアクティビティのみを反映し、ネットワークレイテンシーやクライアント側のアクティビティは考慮されません。

> **Note**
>
> カスタムパーセンタイル値 (p99.9 など) を分析するには、CloudWatch メトリクス統計フィールドに目的のパーセンタイル (p99.9 など) を手動で入力できます。これにより、ドロップダウンに表示されるデフォルトのパーセンタイルを超えるレイテンシー分布を評価できます。

単位: Milliseconds

ディメンション: TableName, Operation, StreamLabel

有効な統計:

- Minimum
- Maximum
- Sum
- Average
- Percentile
- SampleCount

SystemErrors

指定された期間に HTTP 500 ステータスコードを生成する DynamoDB または Amazon DynamoDB Streams へのリクエスト。HTTP 500 は通常、内部サービスエラーを示します。

> **Note**
>
> DynamoDB がシステムエラー (HTTP 500) を返すと、ほとんどの AWS SDK は設定可能な回数の再試行を自動的に実行します。再試行中に問題が解決した場合、アプリケーションはエラーを表示せずに続行しますが、クライアント側でレイテンシーの増加が認識されることが

あります。すべての再試行後もエラーが解決しない場合は、アプリケーションコードに伝播されます。

> **Note**
>
> フォールトインジェクションの実験中に AWS FIS が挿入する HTTP 500 エラーは、FaultInjectionServiceInducedErrors でカウントされるだけでなく、SystemErrors でもカウントされます。その結果、実験中に SystemErrors のアラームが発生します。実験誘発エラーを有機的なサービスエラーと区別するには、同じ期間の SystemErrors と FaultInjectionServiceInducedErrors を比較します。

単位: Count

ディメンション: TableName, Operation

> **Note**
>
> DynamoDB は、TableName および Operation ディメンションペアでのみ SystemErrors を公開します。TableName のみでこのメトリクスを公開することはありません。TableName ディメンションのみを指定する CloudWatch アラームは、データと一致せず、INSUFFICIENT_DATA 状態のままになるため、TableName と Operation の両方を指定します。

有効な統計:

- Sum
- SampleCount

TimeToLiveDeletedItemCount

指定した期間中に有効期限 (TTL) によって削除された項目の数。このメトリクスは、テーブルの TTL 削除率を監視するのに役立ちます。

単位: Count

ディメンション: TableName

有効な統計:

- Sum

ThrottledPutRecordCount

Kinesis Data Streams のキャパシティが不足しているために Kinesis データストリームによってスロットリングされたレコードの数。

単位: Count

ディメンション: TableName, DelegatedOperation

有効な統計:

- Minimum
- Maximum
- Average
- SampleCount

ThrottledRequests

リソース (テーブルやインデックスなど) のプロビジョニング済みスループット制限を超える DynamoDB へのリクエスト。

ThrottledRequestsリクエスト内でいずれかのイベントがプロビジョニング済みスループットクォータを超過した場合、 が 1 つ増加します。例えば、グローバルセカンダリインデックスを持つテーブル内の項目を更新する場合、テーブルへの書き込みと各インデックスへの書き込みという複数のイベントが発生します。これらのイベントの 1 つまたは複数がスロットリングされている場合、ThrottledRequests が 1 つ増加します。

> **Note**
>
> バッチリクエスト (BatchGetItem または BatchWriteItem) では、ThrottledRequests は、バッチ内の各リクエストがスロットリングされた場合にのみ増加します。
> バッチ内の個々のリクエストがスロットリングされると、次のいずれかのメトリクスが増加します。

- ReadThrottleEvents — BatchGetItem 内のスロットリングされた GetItem イベント。
- WriteThrottleEvents — BatchWriteItem 内のスロットリングされた PutItem または DeleteItem。

どのイベントがリクエストをスロットリングしているかについてのインサイトを取得するには、ThrottledRequests とテーブルおよびそのインデックスの ReadThrottleEvents と WriteThrottleEvents を比較します。

> **Note**
>
> スロットリングされたリクエストは HTTP 400 ステータスコードになります。このようなイベントはすべて、ThrottledRequests メトリクスに反映されますが、UserErrors メトリクスには反映されません。

単位: Count

ディメンション: TableName, Operation

有効な統計:

- Sum
- SampleCount

TransactionConflict

同じ項目に対する同時になされた要求のトランザクション競合が原因で、品目レベルの要求が拒否されました。詳細については、「DynamoDB でのトランザクション競合の処理」を参照してください。

単位: Count

ディメンション: TableName

有効な統計:

- Sum — トランザクションの競合により拒否された項目レベルのリクエストの数。

> **Note**
>
> TransactWriteItems または TransactGetItems への呼び出し内の項目レベルの複数のリクエストが拒否された場合、Sum は、各項目レベルの Put、Update、Delete、または Get リクエストに対して 1 つ増加します。

- SampleCount — トランザクションの競合により拒否されたリクエストの数。

> **Note**
>
> TransactWriteItems または TransactGetItems への呼び出し内の項目レベルの複数のリクエストが拒否された場合、SampleCount が 1 つだけ増加します。

- Min — TransactWriteItems、TransactGetItems、PutItem、UpdateItem、または DeleteItem に対する呼び出し内で拒否された項目レベルのリクエストの最小数。
- Max — TransactWriteItems、TransactGetItems、PutItem、UpdateItem、または DeleteItem に対する呼び出し内で拒否された項目レベルのリクエストの最大数。
- Average — TransactWriteItems、TransactGetItems、PutItem、UpdateItem、または DeleteItem に対する呼び出し内で拒否された項目レベルのリクエストの平均数。

UserErrors

指定された期間に HTTP 400 ステータスコードを生成する DynamoDB または Amazon DynamoDB Streams へのリクエスト。HTTP 400 は通常、無効なパラメータの組み合わせ、存在しないテーブルへの更新の試み、不正なリクエスト署名など、クライアント側のエラーを示します。

UserErrors に関連するメトリクスをログに記録する例外の例は次のようになります。

- ResourceNotFoundException
- ValidationException
- TransactionConflict

以下を除き、このようなイベントはすべて UserErrors メトリクスに反映されます。

- ProvisionedThroughputExceededException — このセクションの ThrottledRequests メトリクスを参照してください。

- ConditionalCheckFailedException —このセクションの ConditionalCheckFailedRequests メトリクスを参照してください。

UserErrors は、現在の AWS リージョンおよび現在の AWS アカウントの DynamoDB または Amazon DynamoDB Streams リクエストの HTTP 400 エラーの集計を表します。

単位: Count

有効な統計:

- Sum
- SampleCount

WriteAccountLimitThrottleEvents

アカウント制限によりスロットリングされた書き込みリクエストの数。

単位: Count

ディメンション: TableName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

WriteKeyRangeThroughputThrottleEvents

パーティションの制限によりスロットリングされた書き込みリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。

- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

WriteMaxOnDemandThroughputThrottleEvents

オンデマンドの最大スループットによってスロットリングされた書き込みリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

WriteProvisionedThroughputThrottleEvents

プロビジョニングされたスループット制限によってスロットリングされた書き込みリクエストの数。

単位: Count

ディメンション: TableName、GlobalSecondaryIndexName

有効な統計:

- Sum – スロットリングされたイベントの合計数。
- SampleCount – スロットリングの発生回数。
- Minimum – 任意のサンプル内のスロットリングされたイベントの最小数。
- Maximum – 任意のサンプル内のスロットリングされたイベントの最大数。

WriteThrottleEvents

テーブルまたはグローバルセカンダリインデックス用にプロビジョニングされた書き込み容量ユニットを超える DynamoDB へのリクエスト。

1 つのリクエストで複数のイベントが発生する可能性があります。例えば、3 つのグローバルセカンダリインデックスを持つテーブル上の PutItem リクエストでは、テーブル書き込みと 3 つのインデックス書き込みという 4 つのイベントが発生します。各イベントでは、そのイベントがスロットリングされている場合、WriteThrottleEvents メトリクスは 1 つ増加します。単一の PutItem リクエストでは、いずれかのイベントがスロットリングされている場合、ThrottledRequests も 1 つ増加します。BatchWriteItem では、個々の PutItem または DeleteItem イベントがすべてスロットリングされた場合を除き、BatchWriteItem 全体の ThrottledRequests メトリクスは増加しません。

TableName ディメンションは、グローバルセカンダリインデックスではなくテーブルの WriteThrottleEvents を返します。グローバルセカンダリインデックスの WriteThrottleEvents を表示するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。

単位: Count

ディメンション: TableName, GlobalSecondaryIndexName

有効な統計:

- Sum
- SampleCount

使用状況メトリクス

CloudWatch の使用状況メトリクスを使用して、使用状況をプロアクティブに管理することができます。これは、CloudWatch コンソールでのメトリクスの可視化、カスタムダッシュボードの作成、CloudWatch 異常検出によるアクティビティの変化の検出、使用量がしきい値に近づいたときに警告するアラームの設定などによって実現します。

DynamoDB では、これらの使用状況メトリクスをサービスクォータと統合します。CloudWatch を使用して、アカウントのサービスクォータの使用を管理できます。詳細については、「サービスクォータの視覚化とアラームの設定」を参照してください。

利用可能な使用状況メトリクスのリスト

- AccountProvisionedWriteCapacityUnits
- AccountProvisionedReadCapacityUnits
- TableCount

AccountProvisionedWriteCapacityUnits

アカウントのすべてのテーブルおよびグローバルセカンダリインデックスのプロビジョニング済み書き込みキャパシティ単位の数。

単位: Count

有効な統計:

- Minimum - 一定期間中のプロビジョニング済み書き込みキャパシティ単位の最小数。
- Maximum - 一定期間中のプロビジョニング済み書き込みキャパシティ単位の最大数。
- Average - 一定期間中のプロビジョニング済み書き込みキャパシティ単位の平均数

このメトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み書き込み容量ユニットをすばやく調整すると、この統計には実際の平均値が反映されないことがあります。

AccountProvisionedReadCapacityUnits

アカウントのすべてのテーブルおよびグローバルセカンダリインデックスのプロビジョニング済み読み込みキャパシティ単位の数。

単位: Count

有効な統計:

- Minimum - 一定期間中のプロビジョニング済み読み込みキャパシティ単位の最小数。
- Maximum - 一定期間中のプロビジョニング済み読み込みキャパシティ単位の最大数。
- Average - 一定期間中のプロビジョニング済み読み込みキャパシティ単位の平均数。

このメトリクスは 5 分間隔で発行されます。したがって、プロビジョニング済み読み込み容量ユニットをすばやく調整すると、この統計には実際の平均が反映されないことがあります。

TableCount

アカウントのアクティブなテーブルの数。

単位: Count

有効な統計:

- Minimum - 一定期間中のテーブルの最小数。
- Maximum - 一定期間中のテーブルの最大数。
- Average - 一定期間中のテーブルの平均数。

FaultInjectionServiceInducedErrors

AWS FIS 実験の結果として、指定された期間およびキャッチアップ中にシミュレートされた HTTP 500 ステータスコードを生成する DynamoDB へのリクエスト。

> **Note**
>
> これらのエラーは SystemErrors メトリクスのサブセットです。AWS FIS が挿入する各エラーは、FaultInjectionServiceInducedErrors と SystemErrors の両方でカウントされます。このメトリクスを使用して、SystemErrors のうちどれだけが、有機的なサービスエラーではなく、フォールトインジェクションの実験によるものかを特定します。

単位: Count

ディメンション: TableName、Operation

有効な統計:

- Sum
- SampleCount

VectorSearchRequestBytes

指定された期間にベクトルインデックスに対する SearchVectors オペレーションによって処理されたバイト数。この値は、検索対象および検索結果として返されるベクトルデータのサイズに応じてスケールされ、インデックス内のディメンション数に応じて大きくなります。

単位: Bytes

ディメンション: TableName、VectorIndexName

有効な統計:

- Minimum

- Maximum
- Average
- Sum
- SampleCount

VectorWriteRequestBytes

指定された期間にデータをベクトルインデックスにレプリケートする書き込みオペレーション (PutItem、UpdateItem、DeleteItem、BatchWriteItem、TransactWriteItems) によって処理されたバイト数。この値は、インデックスにレプリケートされたデータのサイズに応じてスケーリングされます。

単位: Bytes

ディメンション: TableName、VectorIndexName

有効な統計:

- Minimum
- Maximum
- Average
- Sum
- SampleCount

DynamoDB のメトリクスとディメンションについて

DynamoDB のメトリクスは、アカウント、テーブル名、グローバルセカンダリインデックス名、オペレーションなどの値で修飾されます。CloudWatch コンソールを使用して、DynamoDB データ、および以下の表に示すいずれかのディメンションを取得できます。

使用できるディメンションのリスト

- DelegatedOperation
- GlobalSecondaryIndexName
- Operation
- OperationType
- Verb

- ReceivingRegion
- ソース
- StreamLabel
- TableName

DelegatedOperation

このディメンションは、 DynamoDB がユーザーに代わって実行するオペレーションにデータを制限します。以下のオペレーションがキャプチャされます。

- Kinesis Data Streams でのデータキャプチャの変更

GlobalSecondaryIndexName

このディメンションは、テーブルのグローバルセカンダリインデックスにデータを制限します。`GlobalSecondaryIndexName` を指定する場合は、`TableName` も指定する必要があります。

Operation

このディメンションは、以下の DynamoDB オペレーションタイプのいずれかにデータを制限します。

- `PutItem`
- `DeleteItem`
- `UpdateItem`
- `GetItem`
- `BatchGetItem`
- `Scan`
- `Query`
- `BatchWriteItem`
- `TransactWriteItems`
- `TransactGetItems`
- `ExecuteTransaction`
- `BatchExecuteStatement`
- `ExecuteStatement`

さらに、データを次の Amazon DynamoDB Streams オペレーションに制限することもできます。

- `GetRecords`

OperationType

このディメンションは、以下のオペレーションタイプのいずれかにデータを制限します。

- `Read`
- `Write`

このディメンションは `ExecuteTransaction` リクエストと `BatchExecuteStatement` リクエストに発行されます。

Verb

このディメンションは、次の DynamoDB PartiQL 動詞のいずれかにデータを制限します。

- Insert: `PartiQLInsert`
- Select: `PartiQLSelect`
- Update: `PartiQLUpdate`
- Delete: `PartiQLDelete`

このディメンションは、`ExecuteStatement` オペレーションに発行されます。

ReceivingRegion

このディメンションは、特定の AWS リージョン内にデータを制限します。これは、DynamoDB グローバルテーブル内のレプリカテーブルから生成されたメトリクスで使用されます。

ソース

DynamoDB は、`ConsumedWriteCapacityUnits` メトリクスに対してこのディメンションを出力し、書き込み容量をより多く消費するものを区別できます。これには、次のいずれかの値を指定できます。

`Customer`

テーブルに対して直接行う書き込みによって消費される書き込み容量 (グローバルテーブルのレプリカテーブルへの書き込みを含む)。

`GlobalTable`

DynamoDB がレプリカの同期を保つためにレプリカテーブルに適用する、グローバルテーブルレプリケーションの書き込みによって消費される書き込み容量。

このディメンションを使用して、容量を計画する場合などに、独自の書き込みの消費をグローバルテーブルレプリケーションの消費から分離します。

StreamLabel

このディメンションは、特定のストリームラベルにデータを制限します。これは、Amazon DynamoDB Streams の `GetRecords` オペレーションから送信されたメトリクスとともに使用されます。

TableName

このディメンションは、特定のテーブルにデータを制限します。この値は、現在のリージョンおよび現在の AWS アカウントの任意のテーブル名にすることができます。

DynamoDB に CloudWatch アラームを作成する

CloudWatch アラームは、指定した期間にわたって単一のメトリクスを監視し、一定期間にわたってしきい値と比較したメトリクスの値に基づいて、1 つ以上の指定したアクションを実行します。アクションは、Amazon SNS のトピックまたは自動スケーリングのポリシーに送信される通知です。ダッシュボードにアラームを追加すると、複数のリージョンにまたがって AWS リソースやアプリケーションをモニタリングし、アラートを受け取ることもできます。作成できるアラームの数に制限はありません。CloudWatch のアラームは、メトリクスが特定の状態にあるだけではアクションを呼び出しません。アクションを呼び出すには、指定した期間継続している必要があります。推奨される DynamoDB アラームのリストについては、「推奨アラーム」を参照してください。

> **Note**
>
> CloudWatch アラームを作成するときは、必要なすべてのディメンションを指定する必要があります。これは、CloudWatch が欠落しているディメンションのメトリクスを集計しないためです。ディメンションが欠落している CloudWatch アラームを作成しても、アラームの作成時にエラーは発生しません。

1 つのプロビジョニングされたテーブル内に、5 つの読み込みキャパシティユニットがあるとします。この場合、プロビジョニングされた読み込みキャパシティ全体を消費する前に通知を受け取る

には、テーブルのプロビジョニングされたキャパシティの消費量が 80% に達したときに通知を送信するように CloudWatch アラームを作成します。アラームは、CloudWatch コンソールまたは AWS CLI を使用して作成できます。

CloudWatch コンソールでのアラームの作成

CloudWatch コンソールでアラームを作成するには

1. AWS マネジメントコンソール にサインインして、CloudWatch コンソール (https://console.aws.amazon.com/cloudwatch/) を開きます。
2. ナビゲーションペインで、[アラーム]、[すべてのアラーム] の順に選択します。
3. [アラームの作成] を選択します。
4. [メトリクス名] 列で、モニタリングするテーブルと **ConsumeReadCapacityUnits** を含む行を見つけます。この行の横にあるチェックボックスをオンにして、[メトリクスを選択] を選択します。
5. [メトリクスと条件の指定] で、[統計] の [合計] を選択します。[期間] として [1 分] を選択します。
6. [Conditions (条件)] で、次のように指定します。
   a. [Threshold type (しきい値タイプ)] で [静的] を選択します。
   b. [**ConsumedReadCapacityUnits** が以下のとき] で、[より大きい/等しい] を選択し、しきい値を 240 に指定します。
7. [次へ] を選択します。
8. [通知] で、[**In alarm**] を選択し、アラームが ALARM 状態のときに通知する SNS トピックを選択します。
9. 完了したら、[次へ] を選択します。
10. 次のページで、アラームの名前と説明を入力し、[次へ] を選択します。
11. [Preview and create (プレビューして作成)] で、情報と条件が正しいことを確認し、[アラームの作成] を選択します。

AWS CLI でのアラームの作成

```
aws cloudwatch put-metric-alarm \
    -\-alarm-name ReadCapacityUnitsLimitAlarm \
    -\-alarm-description "Alarm when read capacity reaches 80% of my provisioned read
 capacity" \
```

```
    -\-namespace AWS/DynamoDB \
    -\-metric-name ConsumedReadCapacityUnits \
    -\-dimensions Name=TableName,Value=myTable \
    -\-statistic Sum \
    -\-threshold 240 \
    -\-comparison-operator GreaterThanOrEqualToThreshold \
    -\-period 60 \
    -\-evaluation-periods 1 \
    -\-alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

アラームのテストを行います。

```
aws cloudwatch set-alarm-state -\-alarm-name ReadCapacityUnitsLimitAlarm -\-state-
reason "initializing" -\-state-value OK
```

```
aws cloudwatch set-alarm-state -\-alarm-name ReadCapacityUnitsLimitAlarm -\-state-
reason "initializing" -\-state-value ALARM
```

その他の AWS CLI の例

次の手順では、テーブルのプロビジョニングされたスループットクォータを超えるリクエストがある場合に、通知を受け取る方法を示します。

1. Amazon SNS トピック `arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput` を作成します。詳細については、「Amazon Simple Notification Service をセットアップする」を参照してください。
2. アラームを作成します。

```
aws cloudwatch put-metric-alarm \
        -\-alarm-name ReadCapacityUnitsLimitAlarm \
        -\-alarm-description "Alarm when read capacity reaches 80% of my
 provisioned read capacity" \
        -\-namespace AWS/DynamoDB \
        -\-metric-name ConsumedReadCapacityUnits \
        -\-dimensions Name=TableName,Value=myTable \
        -\-statistic Sum \
        -\-threshold 240 \
        -\-comparison-operator GreaterThanOrEqualToThreshold \
        -\-period 60 \
        -\-evaluation-periods 1 \
```

```
        -\-alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

3. アラームのテストを行います。

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --
state-reason "initializing" --state-value ALARM
```

次の手順では、システムエラーが発生した場合に通知を受け取る方法を示します。

1. Amazon SNS トピック `arn:aws:sns:us-east-1:123456789012:notify-on-system-errors` を作成します。詳細については、「Amazon Simple Notification Service をセットアップする」を参照してください。
2. アラームを作成します。

```
aws cloudwatch put-metric-alarm \
    --alarm-name SystemErrorsAlarm \
    --alarm-description "Alarm when system errors occur" \
    --namespace AWS/DynamoDB \
    --metric-name SystemErrors \
    --dimensions Name=TableName,Value=myTable
 Name=Operation,Value=aDynamoDBOperation \
    --statistic Sum \
    --threshold 0 \
    --comparison-operator GreaterThanThreshold \
    --period 60 \
    --unit Count \
    --evaluation-periods 1 \
    --treat-missing-data breaching \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. アラームのテストを行います。

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
 "initializing" --state-value OK
```

```
aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason
 "initializing" --state-value ALARM
```

AWS CloudTrail を使用して DynamoDB オペレーションを口グに記録する

DynamoDB は、DynamoDB のユーザー、ロール、または AWS のサービスによって実行されたアクションをレコードするサービスである AWS CloudTrail と統合されています。CloudTrail は、DynamoDB のすべての API コールをイベントとしてキャプチャします。キャプチャされた呼び出しには、DynamoDB コンソールからの呼び出しと PartiQL とクラシック API の両方を使用した DynamoDB API オペレーションへのコード呼び出しが含まれます。証跡を作成する場合は、DynamoDB のイベントなど、Amazon S3 バケットへの CloudTrail イベントの継続的な配信を有効にすることができます。追跡を設定しない場合でも、CloudTrail コンソールの [Event history] (イベント履歴) で最新のイベントを表示できます。CloudTrail で収集された情報を使用して、DynamoDB に対するリクエスト、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

堅牢なモニタリングとアラートのために、CloudTrail イベントを Amazon CloudWatch Logs と統合することもできます。DynamoDB サービスアクティビティの分析を強化し、AWS アカウントのアクティビティの変更を特定するには、Amazon Athena を使用して、AWS CloudTrail をクエリできます。例えば、クエリを使用して傾向を識別したり、アクティビティを属性 (ソース IP アドレスやユーザーなど) でさらに分離したりできます。

設定や有効化の方法など、CloudTrail の詳細については、「 AWS CloudTrail ユーザーガイド」を参照してください。

トピック

- CloudTrail 内の DynamoDB 情報
- DynamoDB ログファイルのエントリについて

CloudTrail 内の DynamoDB 情報

AWS アカウントを作成すると、そのアカウントに対して CloudTrail が有効になります。DynamoDB でサポートされているイベントアクティビティが発生すると、そのアクティビティは [イベント履歴] の他の AWS のサービスのイベントとともに CloudTrail イベントにレコードされます。AWS アカウ

ントで最近のイベントを表示、検索、ダウンロードできます。詳細については、「CloudTrail イベント履歴の操作」を参照してください。

DynamoDB のイベントなど、AWS アカウントのイベントの継続的なレコードについては、証跡を作成します。追跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成すると、すべての AWS リージョンに証跡が適用されます。証跡は、AWS パーティションのすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集したイベントデータをより詳細に分析し、それに基づいて対応するため、他の AWS サービスを構成できます。詳細については、次を参照してください:

- 追跡を作成するための概要
- CloudTrail がサポートされているサービスと統合
- CloudTrail の Amazon SNS 通知の設定
- 複数のリージョンから CloudTrail ログファイルを受け取るおよび複数のアカウントから CloudTrail ログファイルを受け取る

CloudTrail のコントロールプレーンイベント

デフォルトで、以下の API アクションはイベントとして CloudTrail ファイルに記録されます。

Amazon DynamoDB

- CreateBackup
- CreateGlobalTable
- CreateTable
- DeleteBackup
- DeleteTable
- DescribeBackup
- DescribeContinuousBackups
- DescribeGlobalTable
- DescribeLimits
- DescribeTable
- DescribeTimeToLive
- ListBackups

- ListTables
- ListTagsOfResource
- ListGlobalTables
- RestoreTableFromBackup
- RestoreTableToPointInTime
- TagResource
- UntagResource
- UpdateGlobalTable
- UpdateTable
- UpdateTimeToLive
- DescribeReservedCapacity
- DescribeReservedCapacityOfferings
- PurchaseReservedCapacityOfferings
- DescribeScalableTargets
- RegisterScalableTarget

DynamoDB Streams

- DescribeStream
- ListStreams

DynamoDB Accelerator (DAX)

- CreateCluster
- CreateParameterGroup
- CreateSubnetGroup
- DecreaseReplicationFactor
- DeleteCluster
- DeleteParameterGroup
- DeleteSubnetGroup
- DescribeClusters

- DescribeDefaultParameters
- DescribeEvents
- DescribeParameterGroups
- DescribeParameters
- DescribeSubnetGroups
- IncreaseReplicationFactor
- ListTags
- RebootNode
- TagResource
- UntagResource
- UpdateCluster
- UpdateParameterGroup
- UpdateSubnetGroup

CloudTrail の DynamoDB データプレーンイベント

CloudTrail ファイルで以下の API アクションのロギングを有効にするには、CloudTrail でデータプレーン API アクティビティのログ記録を有効にする必要があります。詳細については、「証跡のデータイベントの記録」を参照してください。

データプレーンイベントは、リソースタイプ別にフィルタリングして、CloudTrail でログと支払いを選択的に実行する DynamoDB API 呼び出しをきめ細かくコントロールできます。例えば、リソースタイプに `AWS::DynamoDB::Stream` を指定することで、DynamoDB ストリーム API への呼び出しのみをログに記録できます。ストリームが有効になっているテーブルの場合、データプレーンイベントのリソースフィールドには `AWS::DynamoDB::Stream` と `AWS::DynamoDB::Table` 両方が含まれます。リソースタイプに `AWS::DynamoDB::Table` を指定すると、デフォルトで DynamoDB テーブルと DynamoDB ストリームイベントの両方がログに記録されます。ストリームイベントをログに記録しない場合、ストリームイベントを除外するフィルターを追加できます。詳細については、「AWS CloudTrail API リファレンス」の「DataResource」を参照してください。

Amazon DynamoDB

- BatchExecuteStatement
- BatchGetItem
- BatchWriteItem

- DeleteItem
- ExecuteStatement
- ExecuteTransaction
- GetItem
- PutItem
- Query
- Scan
- SearchVectors
- TransactGetItems
- TransactWriteItems
- UpdateItem

> **Note**
>
> CloudTrail では DynamoDB の有効期限 (TTL) データプレーンのアクションはログに記録されません。

DynamoDB Streams

- GetRecords
- GetShardIterator

> **Important**
>
> `GetRecords` データイベントをログに記録すると、グローバルテーブルのレプリケーションなどの DynamoDB 内部オペレーションからの `GetRecords` 呼び出しが表示される場合があります。これらの `GetRecords` 呼び出しに対して DynamoDB から課金されることはありませんが、データイベントログに対して CloudTrail から課金されます。これにより、予期しない CloudTrail 料金が発生する可能性があります。
> 予期しない CloudTrail 料金を回避するには、次のいずれかを実行します。
>
> - AWS サービスによって開始されたイベントを除外するログセレクタテンプレートを使用します。

- `userIdentity.arn` を `NotStartsWith` `AWSServiceRoleFor` に設定した高度なイベントセレクタフィルターを追加します。

詳細については、「AWS CloudTrail ユーザーガイド」の「データイベントをログ記録する」を参照してください。

DynamoDB ログファイルのエントリについて

「トレイル」は、指定した Amazon S3 バケットにイベントをログファイルとして配信するように設定できます。CloudTrail のログファイルは、単一か複数のログエントリを含みます。イベントは任意の発生元からの 1 つのリクエストを表し、リクエストされたアクション、アクションの日時、リクエストのパラメータなどに関する情報が含まれます。

各イベントまたはログエントリには、リクエストの生成者に関する情報が含まれます。ID 情報は次の判断に役立ちます。

- リクエストが、ルートとユーザー認証情報のどちらを使用して送信されたか。
- リクエストが、ロールとフェデレーションユーザーの一時的なセキュリティ認証情報のどちらを使用して送信されたか。
- リクエストが、別の AWS サービスによって送信されたかどうか。

> **Note**
>
> 非キー属性値は、PartiQL API を使用してアクションの CloudTrail ログで編集され、クラシック API を使用するアクションのログには表示されません。

詳細については、「CloudTrail userIdentity エレメント」を参照してください。

以下の例は、これらのイベントタイプの CloudTrail ログを示しています。

Amazon DynamoDB

- `UpdateTable`
- `DeleteTable`
- `CreateCluster`

- PutItem (成功)
- UpdateItem (失敗)
- TransactWriteItems (成功)
- TransactWriteItems (TransactionCanceledException を含む)
- ExecuteStatement
- BatchExecuteStatement

DynamoDB Streams

- GetRecords

UpdateTable


```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2015-05-04T02:14:52Z",
            "eventSource": "dynamodb.amazonaws.com",
```

```
            "eventName": "UpdateTable",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "console.aws.amazon.com",
            "requestParameters": {
                "provisionedThroughput": {
                    "writeCapacityUnits": 25,
                    "readCapacityUnits": 25
                }
            },
            "responseElements": {
                "tableDescription": {
                    "tableName": "Music",
                    "attributeDefinitions": [
                        {
                            "attributeType": "S",
                            "attributeName": "Artist"
                        },
                        {
                            "attributeType": "S",
                            "attributeName": "SongTitle"
                        }
                    ],
                    "itemCount": 0,
                    "provisionedThroughput": {
                        "writeCapacityUnits": 10,
                        "numberOfDecreasesToday": 0,
                        "readCapacityUnits": 10,
                        "lastIncreaseDateTime": "May 3, 2015 11:34:14 PM"
                    },
                    "creationDateTime": "May 3, 2015 11:34:14 PM",
                    "keySchema": [
                        {
                            "attributeName": "Artist",
                            "keyType": "HASH"
                        },
                        {
                            "attributeName": "SongTitle",
                            "keyType": "RANGE"
                        }
                    ],
                    "tableStatus": "UPDATING",
                    "tableSizeBytes": 0
                }
```

```
            },
            "requestID": "AALNP0J2L244N5O15PKISJ1KUFVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "eb834e01-f168-435f-92c0-c36278378b6e",
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

DeleteTable

```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2015-05-04T13:38:20Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "DeleteTable",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "console.aws.amazon.com",
            "requestParameters": {
```

```
                "tableName": "Music"
            },
            "responseElements": {
                "tableDescription": {
                    "tableName": "Music",
                    "itemCount": 0,
                    "provisionedThroughput": {
                        "writeCapacityUnits": 25,
                        "numberOfDecreasesToday": 0,
                        "readCapacityUnits": 25
                    },
                    "tableStatus": "DELETING",
                    "tableSizeBytes": 0
                }
            },
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

CreateCluster

```
{
    "Records": [
        {
            "eventVersion": "1.05",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "userName": "bob"
            },
            "eventTime": "2019-12-17T23:17:34Z",
            "eventSource": "dax.amazonaws.com",
            "eventName": "CreateCluster",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
```

```
            "userAgent": "aws-cli/1.16.304 Python/3.6.9
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.13.40",
            "requestParameters": {
                "sSESpecification": {
                    "enabled": true
                },
                "clusterName": "daxcluster",
                "nodeType": "dax.r4.large",
                "replicationFactor": 3,
                "iamRoleArn": "arn:aws:iam::111122223333:role/
DAXServiceRoleForDynamoDBAccess"
            },
            "responseElements": {
                "cluster": {
                    "securityGroups": [
                        {
                            "securityGroupIdentifier": "sg-1af6e36e",
                            "status": "active"
                        }
                    ],
                    "parameterGroup": {
                        "nodeIdsToReboot": [],
                        "parameterGroupName": "default.dax1.0",
                        "parameterApplyStatus": "in-sync"
                    },
                    "clusterDiscoveryEndpoint": {
                        "port": 8111
                    },
                    "clusterArn": "arn:aws:dax:us-west-2:111122223333:cache/
daxcluster",
                    "status": "creating",
                    "subnetGroup": "default",
                    "sSEDescription": {
                        "status": "ENABLED",
                        "kMSMasterKeyArn": "arn:aws:kms:us-
west-2:111122223333:key/764898e4-adb1-46d6-a762-e2f4225b4fc4"
                    },
                    "iamRoleArn": "arn:aws:iam::111122223333:role/
DAXServiceRoleForDynamoDBAccess",
                    "clusterName": "daxcluster",
                    "activeNodes": 0,
                    "totalNodes": 3,
                    "preferredMaintenanceWindow": "thu:13:00-thu:14:00",
                    "nodeType": "dax.r4.large"
```

```
                }
            },
            "requestID": "585adc5f-ad05-4e27-8804-70ba1315f8fd",
            "eventID": "29158945-28da-4e32-88e1-56d1b90c1a0c",
            "eventType": "AwsApiCall",
            "recipientAccountId": "111122223333"
        }
    ]
}
```

PutItem (成功)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-05-28T18:06:01Z"
                    },
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    }
                }
            },
            "eventTime": "2019-01-19T15:41:54Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "PutItem",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
 botocore/1.10.63",
```

```
            "requestParameters": {
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Scared of My Shadow"
                },
                "item": [
                    "Artist",
                    "SongTitle",
                    "AlbumTitle"
                ],
                "returnConsumedCapacity": "TOTAL"
            },
            "responseElements": null,
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

`UpdateItem` (失敗)

```
{
    "Records": [
        {
            "eventVersion": "1.07",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
```

```
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2020-09-03T22:27:15Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "UpdateItem",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
botocore/1.10.63",
            "errorCode": "ConditionalCheckFailedException",
            "errorMessage": "The conditional request failed",
            "requestParameters": {
                "tableName": "Music",
                "key": {
                    "Artist": "No One You Know",
                    "SongTitle": "Call Me Today"
                },
                "updateExpression": "SET #Y = :y, #AT = :t",
                "expressionAttributeNames": {
                    "#Y": "Year",
                    "#AT": "AlbumTitle"
                },
                "conditionExpression": "attribute_not_exists(#Y)",
                "returnConsumedCapacity": "TOTAL"
            },
            "responseElements": null,
            "requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
            "readOnly": false,
            "resources": [
```

```
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

`TransactWriteItems` (成功)

```
{
    "Records": [
        {
            "eventVersion": "1.07",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2020-09-03T21:48:12Z",
            "eventSource": "dynamodb.amazonaws.com",
```

```
            "eventName": "TransactWriteItems",
            "awsRegion": "us-west-1",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.15.64 Python/2.7.16 Darwin/17.7.0
botocore/1.10.63",
            "requestParameters": {
                "requestItems": [
                    {
                        "operation": "Put",
                        "tableName": "Music",
                        "key": {
                            "Artist": "No One You Know",
                            "SongTitle": "Call Me Today"
                        },
                        "items": [
                            "Artist",
                            "SongTitle",
                            "AlbumTitle"
                        ],
                        "conditionExpression": "#AT = :A",
                        "expressionAttributeNames": {
                            "#AT": "AlbumTitle"
                        },
                        "returnValuesOnConditionCheckFailure": "ALL_OLD"
                    },
                    {
                        "operation": "Update",
                        "tableName": "Music",
                        "key": {
                            "Artist": "No One You Know",
                            "SongTitle": "Call Me Tomorrow"
                        },
                        "updateExpression": "SET #AT = :newval",
                        "ConditionExpression": "attribute_not_exists(Rating)",
                        "ExpressionAttributeNames": {
                            "#AT": "AlbumTitle"
                        },
                        "returnValuesOnConditionCheckFailure": "ALL_OLD"
                    },
                    {
                        "operation": "Delete",
                        "TableName": "Music",
                        "key": {
                            "Artist": "No One You Know",
```

```
                        "SongTitle": "Call Me Yesterday"
                    },
                    "conditionExpression": "#P between :lo and :hi",
                    "expressionAttributeNames": {
                        "#P": "Price"
                    },
                    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "ConditionCheck",
                    "TableName": "Music",
                    "Key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Now"
                    },
                    "ConditionExpression": "#P between :lo and :hi",
                    "ExpressionAttributeNames": {
                        "#P": "Price"
                    },
                    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                }
            ],
            "returnConsumedCapacity": "TOTAL",
            "returnItemCollectionMetrics": "SIZE"
        },
        "responseElements": null,
        "requestID": "45EN32OM6TQSMV2MI65O4L5TNFVV4KQNSO5AEMVJF66Q9ASUAAJG",
        "eventID": "4f1cc78b-5c94-4174-a6ad-3ee78605381c",
        "readOnly": false,
        "resources": [
            {
                "accountId": "111122223333",
                "type": "AWS::DynamoDB::Table",
                "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
            }
        ],
        "eventType": "AwsApiCall",
        "apiVersion": "2012-08-10",
        "managementEvent": false,
        "recipientAccountId": "111122223333",
        "eventCategory": "Data"
    }
]
```

```
}
```

TransactWriteItems (TransactionCanceledException を含む)

```
{
    "Records": [
        {
            "eventVersion": "1.06",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2019-02-01T00:42:34Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "TransactWriteItems",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.16.93 Python/3.4.7
 Linux/4.9.119-0.1.ac.277.71.329.metal1.x86_64 botocore/1.12.83",
            "errorCode": "TransactionCanceledException",
            "errorMessage": "Transaction cancelled, please refer cancellation reasons
 for specific reasons [ConditionalCheckFailed, None]",
            "requestParameters": {
                "requestItems": [
                    {
                        "operation": "Put",
                        "tableName": "Music",
```

```
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Today"
                    },
                    "items": [
                        "Artist",
                        "SongTitle",
                        "AlbumTitle"
                    ],
                    "conditionExpression": "#AT = :A",
                    "expressionAttributeNames": {
                        "#AT": "AlbumTitle"
                    },
                    "returnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "Update",
                    "tableName": "Music",
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Tomorrow"
                    },
                    "updateExpression": "SET #AT = :newval",
                    "ConditionExpression": "attribute_not_exists(Rating)",
                    "ExpressionAttributeNames": {
                        "#AT": "AlbumTitle"
                    },
                    "returnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "Delete",
                    "TableName": "Music",
                    "key": {
                        "Artist": "No One You Know",
                        "SongTitle": "Call Me Yesterday"
                    },
                    "conditionExpression": "#P between :lo and :hi",
                    "expressionAttributeNames": {
                        "#P": "Price"
                    },
                    "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                },
                {
                    "operation": "ConditionCheck",
```

```
                        "TableName": "Music",
                        "Key": {
                            "Artist": "No One You Know",
                            "SongTitle": "Call Me Now"
                        },
                        "ConditionExpression": "#P between :lo and :hi",
                        "ExpressionAttributeNames": {
                            "#P": "Price"
                        },
                        "ReturnValuesOnConditionCheckFailure": "ALL_OLD"
                    }
                ],
                "returnConsumedCapacity": "TOTAL",
                "returnItemCollectionMetrics": "SIZE"
            },
            "responseElements": null,
            "requestID": "A0GTQEKLBB9VD8E05REA5A3E1VVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "43e437b5-908a-46af-84e6-e27fffb9c5cd",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

ExecuteStatement

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
```

```
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-03-03T23:06:45Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "ExecuteStatement",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.7 Python/3.6.13
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
            "requestParameters": {
                "statement": "SELECT * FROM Music WHERE Artist = 'No One You Know' AND
SongTitle = 'Call Me Today' AND nonKeyAttr = ***(Redacted)"
            },
            "responseElements": null,
            "requestID": "V7G2KCSFLP83ORB7MMFG6RIAD3VV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "0b5c4779-e169-4227-a1de-6ed01dd18ac7",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
```

```
            "eventCategory": "Data"
        }
    ]
}
```

BatchExecuteStatement

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-03-03T23:24:48Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "BatchExecuteStatement",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.7 Python/3.6.13
 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 botocore/1.20.7",
            "requestParameters": {
                "requestItems": [
                    {
                        "statement": "UPDATE Music SET Album = ***(Redacted) WHERE
 Artist = 'No One You Know' AND SongTitle = 'Call Me Today'"
```

```
                    },
                    {
                        "statement": "INSERT INTO Music VALUE {'Artist' :
 ***(Redacted), 'SongTitle' : ***(Redacted), 'Album' : ***(Redacted)}"
                    }
                ]
            },
            "responseElements": null,
            "requestID": "23PE7ED291UD65P9SMS6TISNVBVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "eventID": "f863f966-b741-4c36-b15e-f867e829035a",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

GetRecords

```
{
    "Records": [
        {
            "eventVersion": "1.08",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
                "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
```

```
                        "arn": "arn:aws:iam::444455556666:role/admin-role",
                        "accountId": "444455556666",
                        "userName": "bob"
                    },
                    "attributes": {
                        "creationDate": "2020-09-03T22:14:13Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2021-04-15T04:15:02Z",
            "eventSource": "dynamodb.amazonaws.com",
            "eventName": "GetRecords",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "aws-cli/1.19.50 Python/3.6.13
 Linux/4.9.230-0.1.ac.224.84.332.metal1.x86_64 botocore/1.20.50",
            "requestParameters": {
                "shardIterator": "arn:aws:dynamodb:us-west-2:123456789012:table/
Music/stream/2021-04-15T04:02:47.428|1|AAAAAAAAAAH7HF3xwDQHBrvk2UBZ1PKh8bX3F
+JeH0rFwHCE7dz4VGV1ZoJ5bMxQwkmerA3wzCTL+zSseGLdSXNJP14EwrjLNvDNoZeRSJ/
n6xc3I4NYOptR4zR8d7VrjMAD6h5nR12NtxGIgJ/
dVsUpluWsHyCW3PPbKsMlJSruVRWoitRhSd3S6slEWEPB0bDC7+
+ISH5mXrCHOnvyezQKlqNshTSPZ5jWwqRj2VNSXCMTGXv9P01/
U0bpOUI2cuRTchgUpPSe3ur2sQrRj3KlbmIyCz7P
+H3CYlugafi8fQ5kipDSkESkIWS6O5ejzibWKg/3izms1eVIm/
zLFdEeihCYJ7G8fpHUSLX5JAk3ab68aUXGSFEZLONntgNIhQkcMo00/
mJlaIgkEdBUyqvZO1vtKUBH5YonIrZqSUhv8Coc+mh24vOg1YI+SPIXlr
+Lnl54BG6AjrmaScjHACVXoPDxPsXSJXC4c9HjoC3YSskCPV7uWi0f65/
n7JAT3cskcX2ISaLHwYzJPaMBSftxOgeRLm3BnisL32nT8uTj2gF/
PUrEjdyoqTX7EerQpcaekXmOgay5Kh8n4T2uPdM83f356vRpar/
DDp8pLFD0ddb6Yvz7zU2zGdAvTod3IScC1GpTqcjRxaMhlBVZy1TnI9Cs
+7fXMdUF6xYScjR2725icFBNLojSFVDmsfHabXaCEpmeuXZsLbp5CjcPAHa66R8mQ5tSoFjrzOEzeB4uconEXAMPLE=="
            },
            "responseElements": null,
            "requestID": "1M0U1Q80P4LDPT7A7N1A758N2VVV4KQNSO5AEMVJF66Q9EXAMPLE",
            "eventID": "09a634f2-da7d-4c9e-a259-54aceexample",
            "readOnly": true,
            "resources": [
                {
                    "accountId": "111122223333",
                    "type": "AWS::DynamoDB::Table",
                    "ARN": "arn:aws:dynamodb:us-west-2:123456789012:table/Music"
                }
```

```
            ],
            "eventType": "AwsApiCall",
            "apiVersion": "2012-08-10",
            "managementEvent": false,
            "recipientAccountId": "111122223333",
            "eventCategory": "Data"
        }
    ]
}
```

CloudWatch Contributor Insights for DynamoDB を使用してデータアクセスを分析する

Amazon CloudWatch Contributor Insights for Amazon DynamoDB は、テーブルやインデックス内で最も頻繁にアクセスおよび調整されるキーをすばやく特定する診断ツールです。このツールでは、CloudWatch Contributor Insights を使用します。

テーブルまたはグローバルセカンダリインデックスで CloudWatch Contributor Insights for DynamoDB を有効にすることで、これらのリソースで最もアクセスが多く調整された項目を表示できます。

> **Note**
>
> DynamoDB の Contributor Insights には CloudWatch の料金が適用されます。料金の詳細については、「Amazon CloudWatch の料金」を参照してください。

トピック

- CloudWatch Contributor Insights for DynamoDB: 仕組み
- CloudWatch Contributor Insights for DynamoDB の開始方法
- IAM と CloudWatch Contributor Insights for DynamoDB の使用

CloudWatch Contributor Insights for DynamoDB: 仕組み

Amazon DynamoDB は、Amazon CloudWatch Contributor Insights との統合により、テーブルやグローバルセカンダリインデックスでアクセスとスロットリングが最も多い項目に関する情報を提供し

ます。DynamoDB は、CloudWatch Contributor Insights のルール、レポート、レポートデータのグラフを通じて、この情報を提供します。

CloudWatch Contributor Insights for DynamoDB は、DynamoDB テーブルのパフォーマンスに影響を与えないように設計されています。

詳細については、「Amazon CloudWatch ユーザーガイド」の「Contributor Insights を使用して高カーディナリティデータを分析する」を参照してください。

次のセクションでは、CloudWatch Contributor Insights for DynamoDB の基本概念と動作について説明します。

トピック

- CloudWatch Contributor Insights for DynamoDB のモード
- CloudWatch Contributor Insights for DynamoDB のルール
- CloudWatch Contributor Insights for DynamoDB のグラフを理解する
- 他の DynamoDB 機能との相互作用
- CloudWatch Contributor Insights for DynamoDB の請求

CloudWatch Contributor Insights for DynamoDB のモード

CloudWatch Contributor Insights for DynamoDB には、さまざまなモニタリングニーズを満たす 2 つの異なるモードが用意されています。

スロットリングされたキーモード

このモードは、スロットリングが発生したときにのみイベントを処理することで、スロットリングされたリクエストのみに焦点を当てます。すべてのアクセスパターンを追跡するオーバーヘッドなしで、パフォーマンスの問題に関するインサイトを提供します。このモードでは、DynamoDB は以下のみを追跡します。

- 最もスロットリングされた項目 — 最もスロットリングイベントが発生する項目

このモードは、次の場合に最適です。

- 主な懸念事項は、スロットリングの問題の特定と解決です
- Contributor Insights をリアルタイムスロットリング検出で継続的に有効にしておく必要がある
- スロットリング問題をモニタリングするためのコスト最適化アプローチが必要です

> **Note**
>
> スロットリングされたキーモードは、スロットリングが発生したときにのみイベントを処理するため、継続的なモニタリングに対して費用対効果が向上します。このターゲットを絞ったアプローチにより、コストへの影響を最小限に抑えながら、スロットリングの問題が発生したときにすぐに可視化しながら、この機能を永続的に有効にすることができます。

テーブルにスロットリングが発生していない場合、Contributor Insights グラフにはデータが表示されません。これは正常なパフォーマンスを示します。スロットリングが検出されると、生成されたグラフは、パフォーマンスの問題を引き起こす特定のアクセスパターンを特定するのに役立ちます。この情報は、不均一なアクセスパターンに対処するための戦略を実装するのに役立ちます。

包括的なモニタリング戦略では、これらのスロットリングインサイトを他の CloudWatch メトリクスと統合して、スロットリングイベントと全体的なテーブルパフォーマンスを関連付ける統合ダッシュボードを作成できます。

アクセスおよびスロットリングされたキーモード

このモードでは、アクセスされた項目とスロットリングされた項目の両方を包括的にモニタリングできます。このモードでは、DynamoDB は以下を追跡します。

- 最もアクセスされた項目 — 最も読み取りおよび書き込みキャパシティを消費する項目
- 最もスロットリングされた項目 — 最もスロットリングイベントが発生する項目

このモードは、テーブルのアクセスパターンを完全に可視化する必要があり、トラフィックの多い項目とスロットリングの問題の両方を理解する場合に最適です。

モードの切り替え

DynamoDB コンソール、AWS CLI、または API を使用して、いつでもモードを切り替えることができます。モードを切り替える場合:

- 既存の CloudWatch ルールが新しいモードに合わせて更新されます
- スロットリングされたキー CloudWatch ルールはそのまま残り、スロットリングメトリクスの継続的な履歴データを維持します。
  - スロットリングされたキーモードからアクセスおよびスロットリングされたキーモードに切り替えると、既存のスロットリングされたキールールが保持され、新しいアクセスキールールが作成されます。

- アクセスおよびスロットリングされたキーモードからスロットリングされたキーモードに切り替えると、スロットリングされたキールールのみが保持され、アクセスキールールは削除されます。
- 請求は、新しいモードのイベント処理を反映するようにすぐに調整されます。

CloudWatch Contributor Insights for DynamoDB のルール

テーブルまたはグローバルセカンダリインデックスで DynamoDB の CloudWatch Contributor Insights を有効にすると、DynamoDB は選択したモードに基づいてユーザーに代わってルールを作成します。

> **Note**
>
> DynamoDB テーブルで Contributor Insights を有効にする場合、Contributor Insights ルールによる制限が適用されます。詳細については、「CloudWatch サービスのクォータ」を参照してください。

アクセスおよびスロットリングされたキーモードのルール

アクセスおよびスロットリングされたキーモードでは、DynamoDB は次のルールを作成します。

- 最もアクセスされた項目 (パーティションキー) - テーブルまたはグローバルセカンダリインデックスで最もアクセスされた項目のパーティションキーを識別します。

  CloudWatch ルール名の形式: `DynamoDBContributorInsights-PKC-[resource_name]-[creationtimestamp]`
- 最もスロットリングされたキー (パーティションキー) - テーブルまたはグローバルセカンダリインデックスで最もスロットリングされた項目のパーティションキーを識別します。

  CloudWatch ルール名の形式: `DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]`

テーブルまたはグローバルセカンダリインデックスにソートキーがある場合、DynamoDB はソートキーに固有の次のルールも作成します。

- 最もアクセスされたキー (パーティションキーとソートキー) - テーブルまたはグローバルセカンダリインデックスで最もアクセスされた項目のパーティションキーとソートキーを識別します。

CloudWatch ルール名の形式: `DynamoDBContributorInsights-SKC-[resource_name]-[creationtimestamp]`

- 最もスロットリングされたキー (パーティションキーとソートキー) - テーブルまたはグローバルセカンダリインデックスで最もスロットリングされた項目のパーティションキーとソートキーを識別します。

  CloudWatch ルール名の形式: `DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]`

スロットリングされたキーモードのルール

スロットリングされたキーモードでは、DynamoDB はスロットリング関連のルールのみを作成します。

- 最もスロットリングされたキー (パーティションキー) - テーブルまたはグローバルセカンダリインデックスで最もスロットリングされた項目のパーティションキーを識別します。

  CloudWatch ルール名の形式: `DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]`

テーブルまたはグローバルセカンダリインデックスにソートキーがある場合、DynamoDB は以下も作成します。

- 最もスロットリングされたキー (パーティションキーとソートキー) - テーブルまたはグローバルセカンダリインデックスで最もスロットリングされた項目のパーティションキーとソートキーを識別します。

  CloudWatch ルール名の形式: `DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]`

この焦点を絞ったアプローチにより、アクティブなルールの数が減少し、スロットリングイベントをより適切に診断するために処理されるイベントの量が減少します。

> **Note**
>
> - CloudWatch コンソールまたは API を使用して CloudWatch Contributor Insights for DynamoDB を表示すると、選択したモードに対応するルールのみが表示されます。

- CloudWatch コンソールまたは API を使用して、CloudWatch Contributor Insights for DynamoDB によって作成されたルールを直接変更または削除することはできません。テーブルまたはグローバルセカンダリインデックスで CloudWatch Contributor Insights for DynamoDB を無効にすると、そのテーブルまたはグローバルセカンダリインデックス用に作成されたルールが自動的に削除されます
- DynamoDB によって作成された CloudWatch Contributor Insights ルールで GetInsightRuleReport オペレーションを使用すると、`MaxContributorValue` と `Maximum` のみが有用な統計を返します。このリストの他の統計は、意味のある値を返しません。
- CloudWatch Contributor Insights for DynamoDB の寄稿者の上限は 25 です。25 人を超える寄稿者をリクエストすると、エラーが返されます。

CloudWatch アラームは、CloudWatch Contributor Insights for DynamoDB ルールを使用して作成できます。これにより、項目が `ConsumedThroughputUnits` または `ThrottleCount` の特定のしきい値を超えたとき、または満たすときに通知を受け取ることができます。詳細については、「Contributor Insights メトリクスデータでのアラームの設定」を参照してください。

CloudWatch Contributor Insights for DynamoDB のグラフを理解する

CloudWatch Contributor Insights for DynamoDB は、選択したモードに応じて、DynamoDB コンソールと CloudWatch コンソールの両方で異なる種類のグラフを表示します。

モード別のグラフの可用性

表示されるグラフは、選択した Contributor Insights モードによって異なります。

- アクセスおよびスロットリングされたキーモードは、最もアクセスされた項目と最もスロットリングされた項目のグラフの両方を表示します
- スロットリングされたキーモードは最もスロットリングされた項目のグラフのみを表示します

最もアクセスされた項目

このグラフは、アクセスキーモードとスロットリングされたキーモードでのみ使用できます。このグラフを使用して、テーブルまたはグローバルセカンダリインデックスで最もアクセスの多い項目を特定します。グラフの Y 軸に `ConsumedThroughputUnits` が、X 軸に時間が表示されます。上位 N 個のキーはそれぞれ独自の色で表示され、X 軸の下に凡例が表示されます。

DynamoDB は、ConsumedThroughputUnits を使用してキーアクセス頻度を測定します。これは、読み取りと書き込みのトラフィックを組み合わせて測定します。ConsumedThroughputUnits は次のように定義されます。

- プロビジョンド - (3 x 消費された書き込み容量の単位) + 消費された読み込み容量の単位
- オンデマンド - (3 x 書き込みリクエスト単位) + 読み込みリクエスト単位

DynamoDB コンソールで、グラフの各データポイントは 1 分間にわたる ConsumedThroughputUnits の最大値を表します。例えば、ConsumedThroughputUnits のグラフ値が 180,000 である場合は、1 分間以内の 60 秒間にわたり、項目別の最大スループットを 1,000 の書き込みリクエスト単位または 3,000 の読み取りリクエスト単位として、項目が継続的にアクセスされたことを示します (3,000 x 60 秒)。つまり、グラフ化された値は、各 1 分間以内で最もトラフィックの多い 1 分間を表します。ConsumedThroughputUnits メトリクスの時間粒度は CloudWatch コンソールで変更できます (1 分間ではなく 5 分間のメトリクスを表示するなど)。

明確な異常値がなく、いくつかの密接にクラスター化された行が表示されている場合、指定された時間枠内で項目間でワークロードのバランスが比較的取れていることを示しています。グラフに接続線ではなく孤立したポイントが表示されている場合は、項目が短時間だけ頻繁にアクセスされたこと示します。

テーブルまたはグローバルセカンダリインデックスにソートキーがある場合、DynamoDB は 2 つのグラフを作成します。1 つは最もアクセス頻度の高いパーティションキー用、もう 1 つは最もアクセス頻度の高いパーティションキーとソートキーのペア用です。パーティションキーレベルのトラフィックは、パーティションキー専用のグラフでのみ確認できます。項目レベルのトラフィックは、パーティションキーとソートキーのペア用グラフで確認できます。

最もスロットリングされた項目

このグラフは両方のモードで使用できます。このグラフを使用して、テーブルまたはグローバルセカンダリインデックスで最もスロットリングされた項目を特定します。グラフの Y 軸に ThrottleCount が、X 軸に時間が表示されます。上位 N 個の各キーは独自の色で表示され、X 軸の下に凡例が表示されます。

DynamoDB は ThrottleCount を使用してスロットル頻度を測定します。これは、ProvisionedThroughputExceededException、ThrottlingException、および RequestLimitExceeded エラーのカウントです。

グローバルセカンダリインデックスの書き込み容量が不十分なため発生した書き込みスロットルは測定されません。グローバルセカンダリインデックスの [最もアクセスされた項目] グラフを使用し

て、書き込みスロットリングを引き起こす可能性のある不均衡なアクセスパターンを特定できます。詳細については、「グローバルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項」を参照してください。

DynamoDB コンソールの場合、グラフの各データポイントは 1 分間のスロットルイベント数を表します。

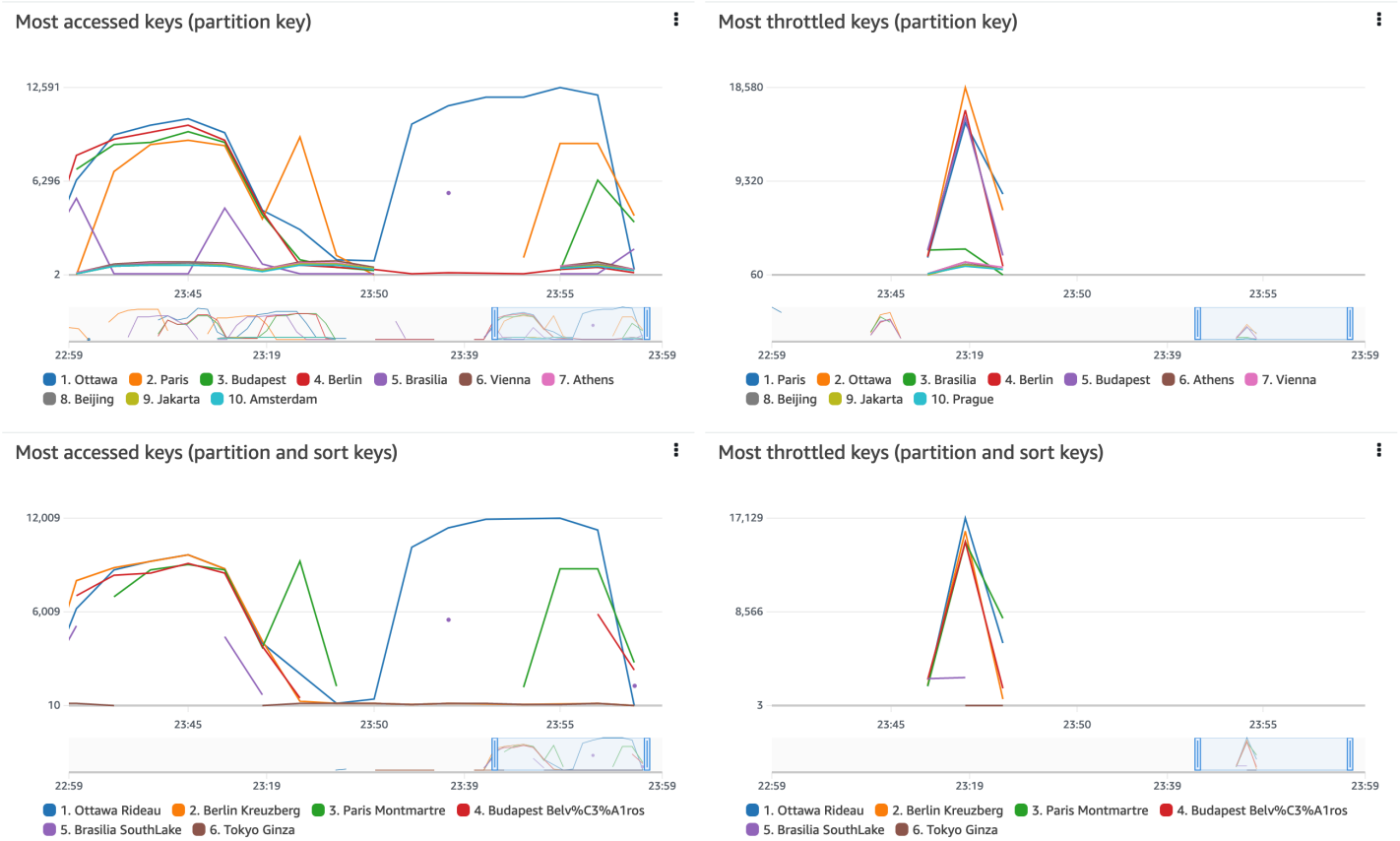
このグラフにデータが表示されない場合は、リクエストがスロットリングされていないことを示しています。グラフに接続線ではなく孤立したポイントが表示されている場合は、項目が短時間に頻繁にスロットリングされたことを示します。

テーブルまたはグローバルセカンダリインデックスにソートキーがある場合、DynamoDB は 2 つのグラフを作成します。1 つは最もスロットリングされたパーティションキー用、もう 1 つは最もスロットリングされたパーティションキーとソートキーのペア用です。パーティションキーのみのグラフではパーティションキーレベルのスロットルカウントを、パーティションキーとソートキーのグラフでは項目レベルのスロットルカウントを確認できます。

> **Note**
>
> スロットリングされたキーモードでは、表示されるグラフのタイプはこれだけです。これらのグラフにデータが表示されない場合は、スロットリングが発生しておらず、テーブルのパフォーマンスが正常であることを示しています。

レポートの例

以下の例は、パーティションキーとソートキーの両方がアクセスおよびスロットリングされたキーモードにあるテーブルに対して作成されたレポートを示しています。スロットリングされたキーモードでは、このレポートのスロットリング関連の部分のみが表示されます。



他の DynamoDB 機能との相互作用

次のセクションでは、CloudWatch Contributor Insights for DynamoDB がどのように動作し、DynamoDB の他のいくつかの機能とどのようにやり取りするかについて説明します。これらの動作は、特に指定がない限り、両方のモードに適用されます。

グローバルテーブル

CloudWatch Contributor Insights for DynamoDB は、グローバルテーブルのレプリカを個別のテーブルとしてモニタリングします。Contributor Insights のグラフに表示されるレプリカのパターンは、AWS リージョン間で異なる場合があります。これは、書き込みデータはグローバルテーブル内のすべてのレプリカにレプリケートされますが、リージョンにバインドされた読み込みトラフィックは各レプリカで処理できるためです。

各レプリカは、異なる Contributor Insights モードを個別に設定できます。例えば、プライマリリージョンでアクセスおよびスロットリングされたキーモードを使用して包括的なモニタリングを行い、セカンダリリージョンでスロットリングされたキーモードを使用してパフォーマンスの問題の可視性を維持できます。

DynamoDB Accelerator (DAX)

CloudWatch Contributor Insights for DynamoDB は、DAX のキャッシュ応答を表示しません。テーブルまたはグローバルセカンダリインデックスへのアクセスに対する応答のみを表示します。

> **Note**
>
> DynamoDB CloudWatch Contributor Insights は PartiQL リクエストをサポートしていません。

保管中の暗号化

CloudWatch Contributor Insights for DynamoDB は、DynamoDB での暗号化の動作には影響しません。CloudWatch で発行されるプライマリキーデータは、AWS 所有のキー で暗号化されます。ただし、DynamoDB は AWS マネージドキー やカスタマーマネージドキーもサポートしています。

CloudWatch Contributor Insights for DynamoDB は、頻繁にアクセスされ、スロットリングされる項目のパーティションキーとソートキー (該当する場合) を表示します。CloudWatch Contributor Insights は暗号化された DynamoDB テーブルで動作しますが、テーブルの設定された暗号化とは別の独自の Amazon 所有の暗号化コンテキストを使用することに注意してください。

DynamoDB テーブルのプライマリキーに機密情報が含まれており、組織のセキュリティポリシーで暗号化プロセスを完全に制御する必要がある場合、CloudWatch Contributor Insights を有効にすることが適切ではない場合があります。

きめ細かなアクセスコントロール

CloudWatch Contributor Insights for DynamoDB は、きめ細かなアクセスコントロール (FGAC) を使用してテーブル別に動作を調整することはありません。つまり、適切な CloudWatch 許可を持つユーザーは、CloudWatch Contributor Insights のグラフで FGAC で保護されたプライマリキーを表示できます。

テーブルのプライマリキーに、CloudWatch に公開したくない FGAC で保護されたデータが含まれている場合、そのテーブルに対して CloudWatch Contributor Insights for DynamoDB を有効にしないでください。

アクセスコントロール

DynamoDB コントロールプレーンの許可と CloudWatch データプレーンの許可を制限することにより、AWS Identity and Access Management (IAM) を使用して、CloudWatch Contributor Insights

for DynamoDB へのアクセスを制御します。詳細については、「CloudWatch Contributor Insights for DynamoDB での IAM の使用」を参照してください。

CloudWatch Contributor Insights for DynamoDB の請求

Contributor Insights for DynamoDB の料金は、毎月の請求書の CloudWatch セクションに表示されます。これらの料金には、機能を有効にしたときに作成される各 Contributor Insights ルールの固定月額料金と、処理される DynamoDB イベントの数に基づく料金 (選択したモードによって異なる) の 2 つのコンポーネントがあります。

モード別の請求

2 つの Contributor Insights モードの請求特性は異なります。

- アクセスおよびスロットリングされたキーモードの請求 - このモードでは、データプレーンオペレーションを介して書き込みまたは読み取りされる各項目は、リクエストが成功したかスロットリングされたかに関係なく、1 つのイベントを表します。テーブルまたはグローバルセカンダリインデックスにソートキーが含まれている場合、読み込みまたは書き込みされる各項目は 2 つのイベントを表します。これは、DynamoDB が別々の時系列から上位の貢献者を識別するためです。1 つはパーティションキー専用の時系列、もう 1 つはパーティションキーとソートキーのペア用の時系列です。
- スロットリングされたキーモードの請求 - このモードでは、スロットリングされたリクエストのみが請求可能なイベントを生成します。イベントは、リクエストが `ProvisionedThroughputExceededException`、`ThrottlingException`、または `RequestLimitExceeded` エラーになった場合にのみ生成されます。テーブルまたはグローバルセカンダリインデックスにソートキーが含まれている場合、スロットリングされた各項目は 2 つのイベント (パーティションキーの追跡とパーティション + ソートキーの追跡) を表します。

請求例

例えば、5 つの項目を配置する `GetItem`、`PutItem`、`BatchWriteItem` の DynamoDB オペレーションがアプリケーションによって実行されたとします。また、`PutItem` オペレーションはスロットリングされますが、他のすべてのオペレーションは成功すると仮定します。

- アクセスおよびスロットリングされたキーモード
  - テーブルまたはグローバルセカンダリインデックスにパーティションキーのみがある場合、イベント数は 7 になります (`GetItem` に 1、`PutItem` に 1、`BatchWriteItem` に 5)。

  - テーブルまたはグローバルセカンダリインデックスにパーティションキーとソートキーがある場合、イベント数は 14 になります (GetItem に 2、PutItem に 2、BatchWriteItem に 10)。
- スロットリングされたキーモード
  - テーブルまたはグローバルセカンダリインデックスにパーティションキーのみがある場合、イベント数は 1 になります (スロットリングされた PutItem のみ)。
  - テーブルまたはグローバルセカンダリインデックスにパーティションキーとソートキーがある場合、イベント数は 2 になります (スロットリングされた PutItem に 2 つ)。

  正常な GetItem および BatchWriteItem オペレーションの場合、スロットリングされたキーモードではイベントは生成されません。

一般的な請求要因

Query オペレーションでは、モードや返される項目の数に関係なく、イベント数は常に 1 です。

DynamoDB の他の機能とは異なり、CloudWatch Contributor Insights for DynamoDB の請求は以下に基づいて変動しません。

- 容量モード (プロビジョンドとオンデマンド)
- 読み込みリクエストと書き込みリクエストのどちらを実行するか
- 読み込みまたは書き込みされた項目のサイズ (KB)

CloudWatch Contributor Insights for DynamoDB の開始方法

このセクションでは、Amazon DynamoDB コンソールまたは AWS Command Line Interface (AWS CLI) で、モニタリングのニーズに合わせて Amazon CloudWatch Contributor Insights をさまざまなモードで有効化し、使用する方法について説明します。

次の例では、「DynamoDB の使用開始」チュートリアルで定義された DynamoDB テーブルを使用します。

トピック

- Contributor Insights モードの選択
- Contributor Insights の使用 (コンソール)
- Contributor Insights の使用 (AWS CLI)

Contributor Insights モードの選択

Contributor Insights を有効にする前に、使用可能な 2 つのモードを理解しておく必要があります。モード比較を確認して、特定の要件に最も適したオプションを選択します。

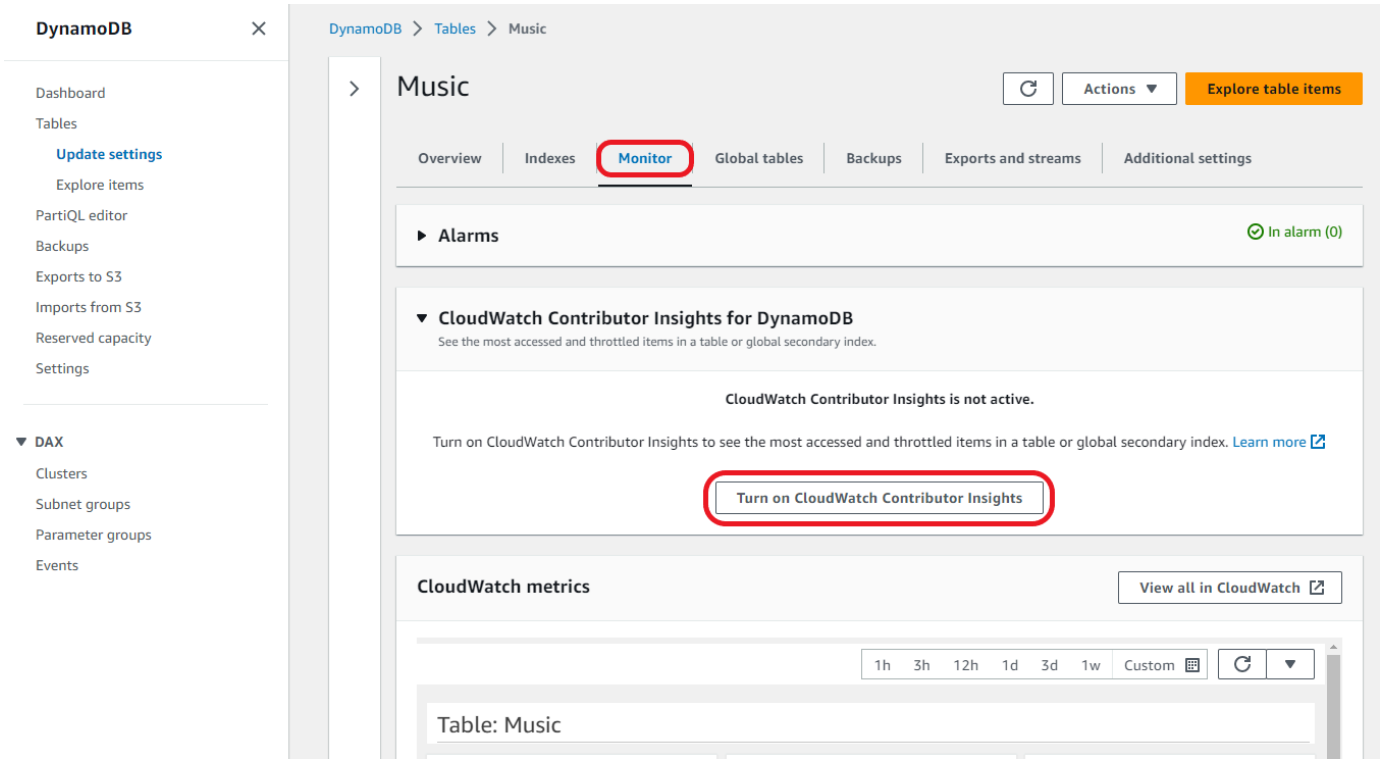
側面	アクセスおよびスロットリングされたキーモード	スロットリングされたキーモード
モニタリング	すべてのリクエスト (成功およびスロットリング)	スロットリングされたリクエストのみ
グラフ:	最もアクセスされた項目 + 最もスロットリングされた項目	最もスロットリングされた項目のみ
最適な用途	ターゲットを絞った分析と最適化	スロットリングの監視
次の場合に使用	アクセスパターンを完全に可視化する必要があります。短期分析またはデバッグを実行している。	主な懸念事項は、スロットリングの問題の特定と解決です。Contributor Insights をリアルタイムスロットリングアラートで継続的に有効にしておく必要がある。

Contributor Insights の使用 (コンソール)

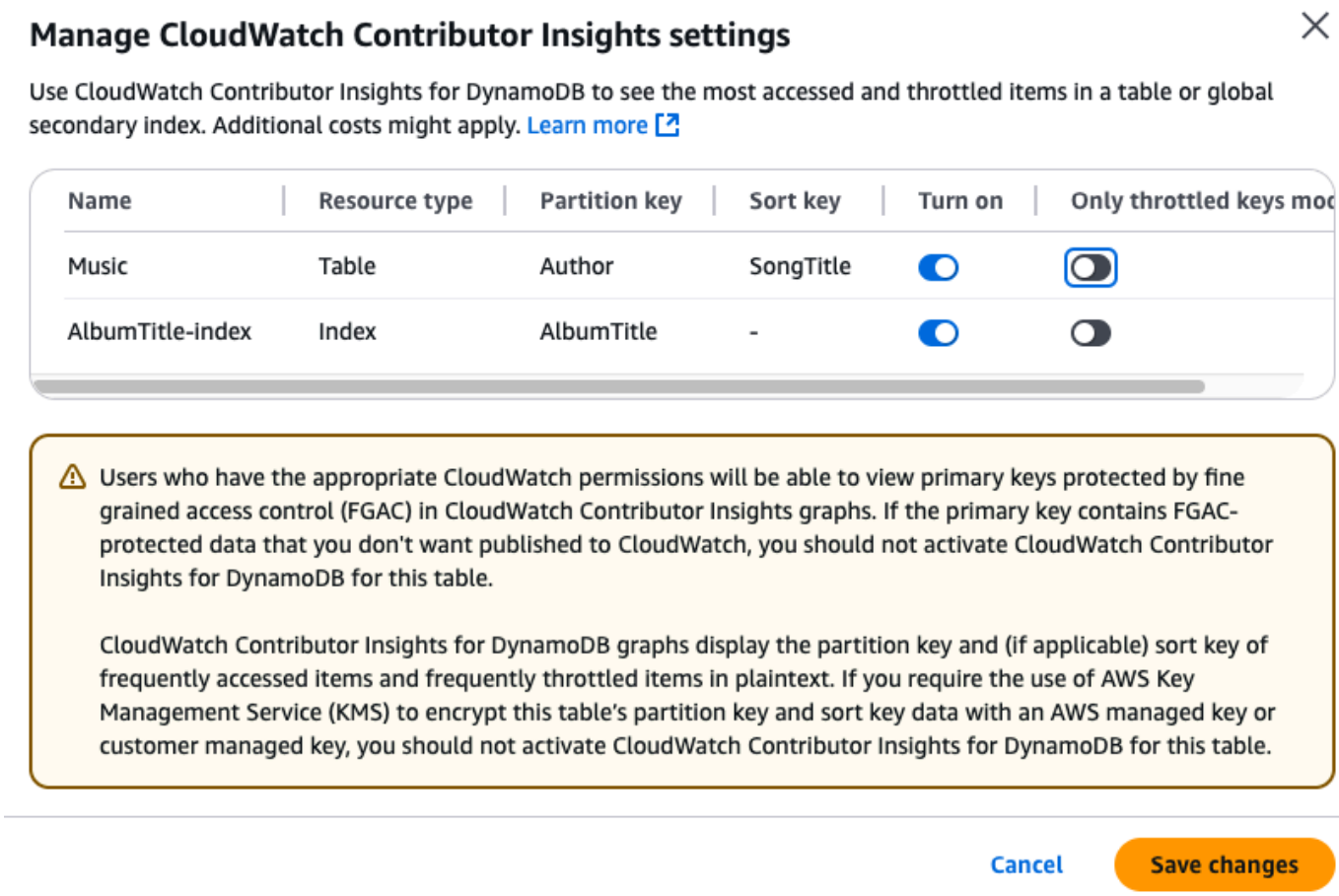
コンソールは、Contributor Insights を有効にし、モニタリングニーズに適したモードを選択する直感的な方法を提供します。

コンソールで Contributor Insights を使用するには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. `Music` テーブルを選択します。
4. [Monitor] (モニタリング) タブを選択します。
5. [Turn on CloudWatch Contributor Insights] (CloudWatch Contributor Insights を有効にする) を選択します。



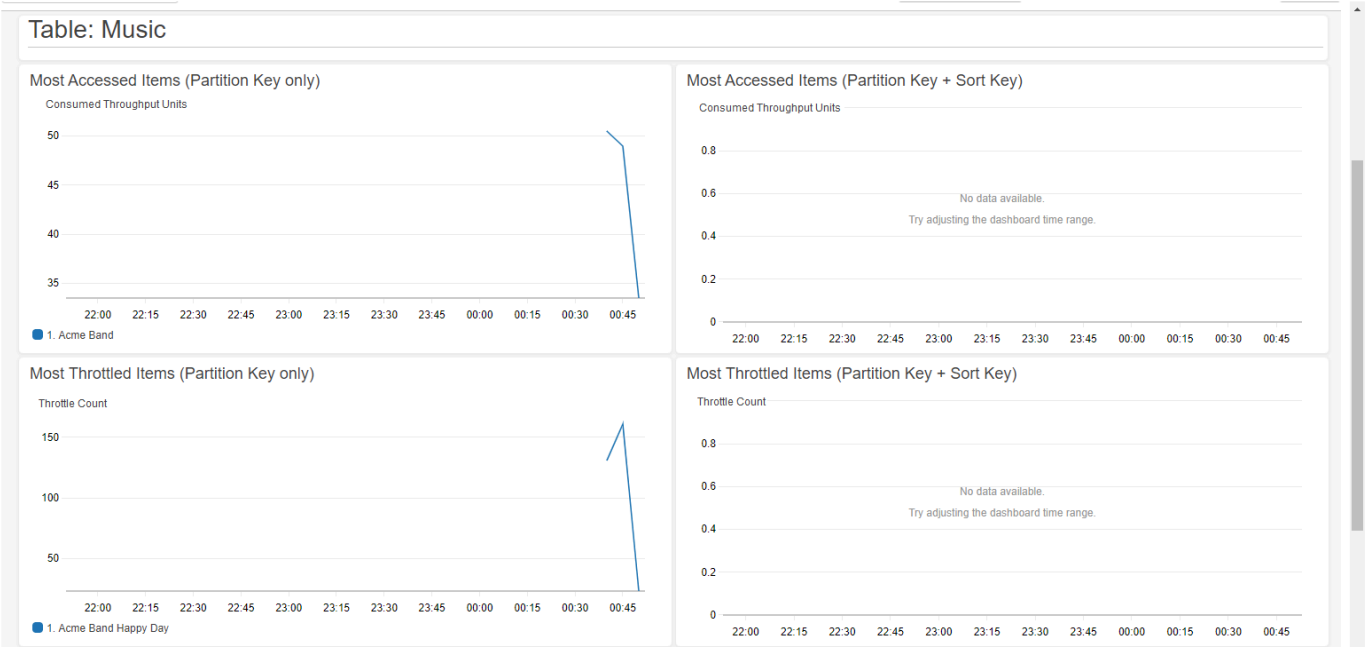
6. [CloudWatch Contributor Insights 設定の管理] ダイアログボックスで、`Music` 基本テーブルと `AlbumTitle-index` グローバルセカンダリインデックスの両方に対して [オンにする] に切り替えます。
7. [スロットルキーモードのみ] の切り替えを両方のオフ位置にしてから、[変更を保存] を選択します。



これにより、テーブルと GSI の両方でデフォルトの [アクセスおよびスロットリングされたキー] モードが有効になり、アクセスされた項目とスロットリングされた項目の両方をモニタリングできます。[スロットルキーモードのみ] をオンに切り替えると、[スロットリングされたキー] モードが有効になります。

このオペレーションが失敗した場合は、Amazon DynamoDB API リファレンスの「DescribeContributorInsights FailureException」を参照して考えられる原因を検討してください。

8. CloudWatch Contributor Insights グラフが `Music` テーブルの [モニタリング] タブに表示されます。[アクセスおよびスロットリングされたキー] モードを有効にしたため、アクセスされた項目とスロットリングされた項目の両方のグラフが表示されます。



モードの切り替え

Contributor Insights を無効にすることなく、いつでもモードを切り替えることができます。

Contributor Insights モードを切り替えるには

1. テーブルの [モニタリング] タブで、[CloudWatch Contributor Insights を管理] を選択します。
2. [Contributor Insights 設定の管理] ダイアログボックスで、各基本テーブルまたは GSI に対して以下を行います。
   - [スロットリングされたキーのみモード] をオンまたはオフに切り替えて、[スロットリングされたキー] モードを有効にするか、デフォルトの [アクセスおよびスロットリングされたキー] モードに戻ります。
   - [オンにする] をオフに切り替えて、テーブルまたは GSI の CloudWatch Contributor Insight を無効にします。
3. [Save changes] (変更の保存) をクリックします。

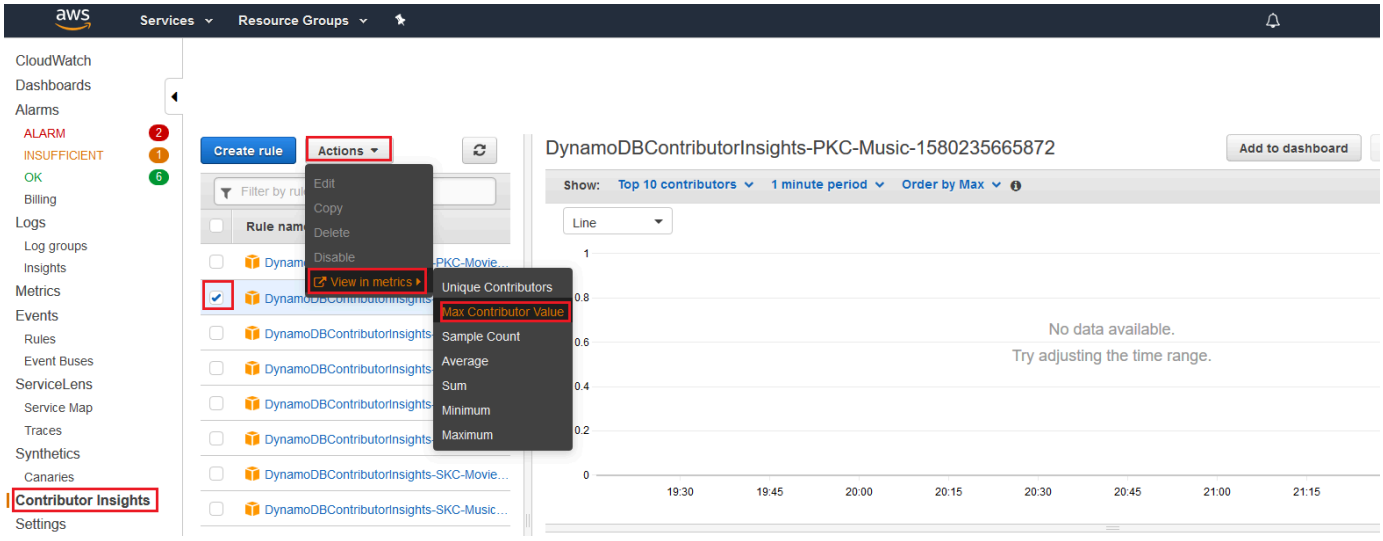
   これが完了すると、グラフに新しいモードが反映されます。

CloudWatch アラームの作成d\

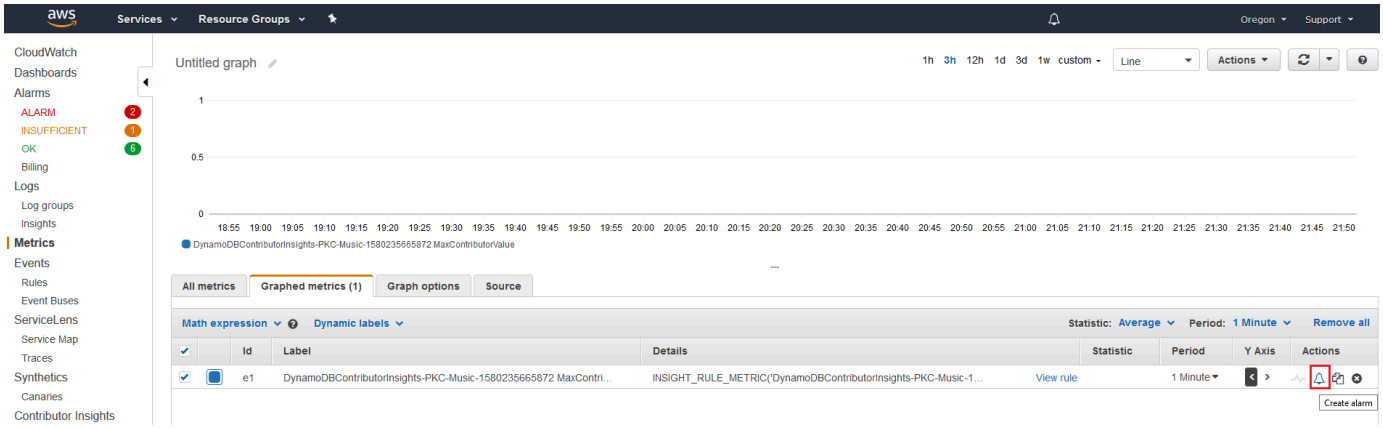
以下のステップに従って、CloudWatch アラームを作成し、パーティションキーが 50,000 件の ConsumedThroughputUnits を超えた場合、またはスロットリングが発生した場合に通知を受け取ります。

1. AWS マネジメントコンソール にサインインして、CloudWatch コンソール (https://console.aws.amazon.com/cloudwatch/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Contributor Insights] を選択します。
3. モードとモニタリングする内容に基づいて、適切なルールを選択します。
   - アクセスされた項目のモニタリング (アクセスおよびスロットリングされたキーモードのみ): [DynamoDBContributorInsights-PKC-Music] を選択します
   - スロットリングされた項目のモニタリング (両方のモード): [DynamoDBContributorInsights-PKT-Music] を選択します
4. [アクション] ドロップダウンを選択します。
5. [View in metrics (メトリクスで表示)] を選択します。
6. [Max Contributor Value (寄稿者の最大値)] を選択します。

> **Note**
>
> `Max Contributor Value` と `Maximum` のみが有用な統計を返します。このリストの他の統計は、意味のある値を返しません。

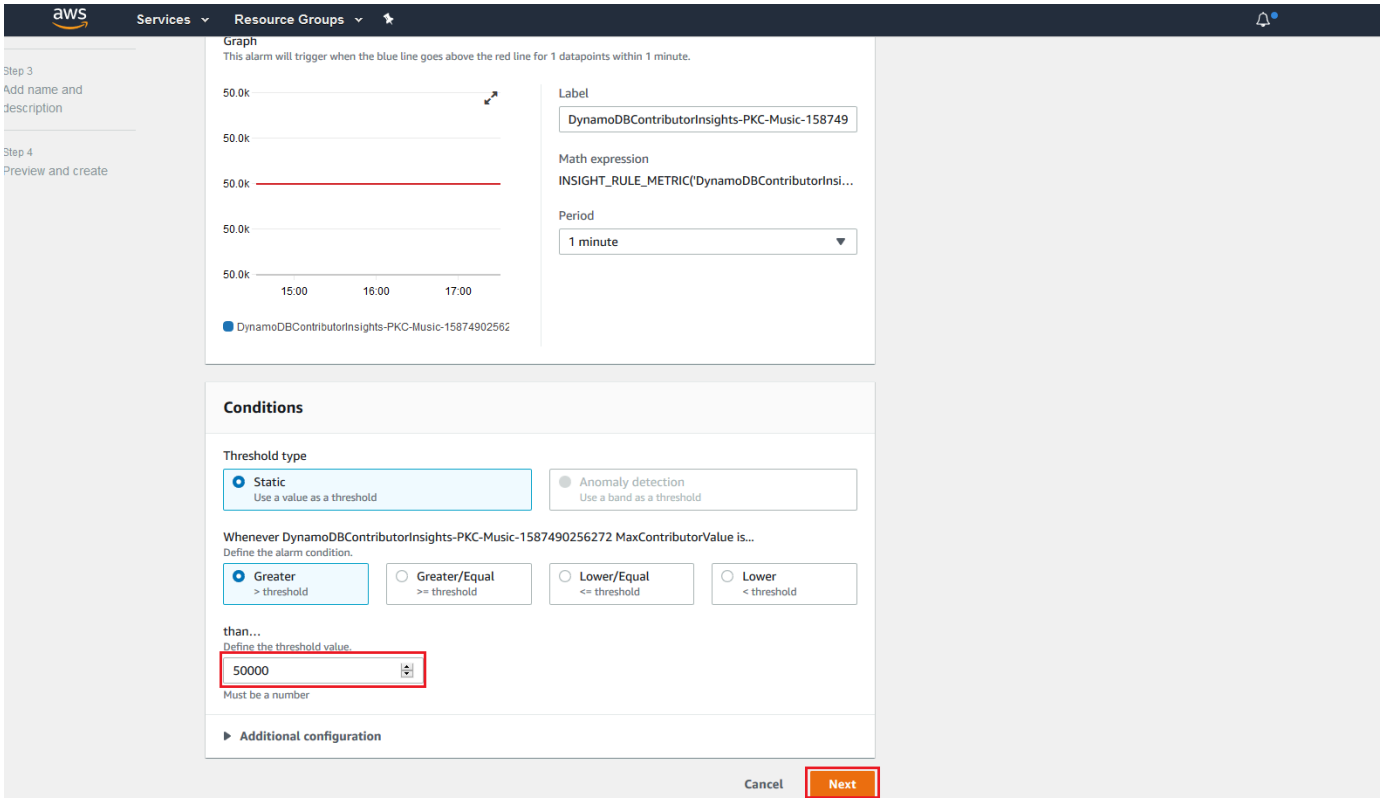


7. [アクション] 列で、[Create Alarm (アラームの作成)] を選択します。



8. 適切なしきい値を入力し、[次へ] を選択します。

   - アクセスされた項目 (PKC ルール): `ConsumedThroughputUnits` に 50,000 を入力します
   - スロットリングされた項目 (PKT ルール): スロットリングでアラートを受け取る `ThrottleCount` に 1 を入力します



9. アラームの通知を設定する方法の詳細については、「Amazon CloudWatch アラームの使用」を参照してください。

Contributor Insights の使用 (AWS CLI)

AWS CLI は、Contributor Insights へのプログラムによるアクセスを提供し、両方のモードを完全にサポートします。Contributor Insights を有効にするとき、または後でモードを切り替えるときにモードを指定できます。

デフォルトモードの基本的な操作

デフォルト設定で Contributor Insights を使用するには

1. `Music` 基本テーブルで、[アクセスおよびスロットリングされたキー] モードを使用して、CloudWatch Contributor Insights for DynamoDB を有効化します。`ACCESSED_AND_THROTTLED_KEYS` がデフォルトモードであるため、`--contributor-insights-mode=ACCESSED_AND_THROTTLED_KEYS`パラメータを省略できます。

   ```
   aws dynamodb update-contributor-insights \
                       --table-name Music \
                       --contributor-insights-action=ENABLE
   ```

2. `AlbumTitle-index` グローバルセカンダリインデックスで Contributor Insights for DynamoDB を有効化します。

   ```
   aws dynamodb update-contributor-insights \
                       --table-name Music \
                       --index-name AlbumTitle-index \
                       --contributor-insights-action=ENABLE
   ```

3. `Music` テーブルとそのすべてのインデックスのステータスとルールを取得します。

   ```
   aws dynamodb describe-contributor-insights
                       --table-name Music
   ```

   レスポンスには、`ACCESSED_AND_THROTTLED_KEYS` を示す `ContributorInsightsMode` フィールドが含まれます。

4. `Music` テーブルとそのすべてのインデックスのステータスを一覧表示します。

   ```
   aws dynamodb list-contributor-insights --table-name Music
   ```

スロットリングされたキーモードの有効化

スロットリングされたキーモードで Contributor Insights を有効にするには

1. `Music` 基本テーブルで、スロットリングされたキーモードを使用して CloudWatch Contributor Insights for DynamoDB を有効化します。

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

2. `AlbumTitle-index` グローバルセカンダリインデックスのスロットリングされたキーモードで Contributor Insights を有効にします。

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --index-name AlbumTitle-index \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

3. Contributor Insights の設定を記述してモードを確認します。

```
aws dynamodb describe-contributor-insights --table-name Music
```

レスポンスは `THROTTLED_KEYS` として `ContributorInsightsMode` を表示し、デフォルトモードと比較してルールの数が少なくなります。

モードの切り替え

Contributor Insights モードを切り替えるには

1. スロットリングされたキーモードからアクセスおよびスロットリングされたキーモードに切り替えます。

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=ACCESSED_AND_THROTTLED_KEYS
```

2. アクセスおよびスロットリングされたキーモードからスロットリングされたキーモードに切り替えます。

```
aws dynamodb update-contributor-insights \
    --table-name Music \
    --contributor-insights-action=ENABLE \
    --contributor-insights-mode=THROTTLED_KEYS
```

3. 移行中にステータスを確認します。

```
aws dynamodb describe-contributor-insights --table-name Music
```

モードの切り替え中、ContributorInsightsStatus は ENABLING として表示されます。これが完了すると、新しいモードでは ENABLED と表示されます。

Contributor Insights の管理

Contributor Insights の設定を管理するには

1. AlbumTitle-index グローバルセカンダリインデックスで CloudWatch Contributor Insights for DynamoDB を無効化します。

```
aws dynamodb update-contributor-insights \
                    --table-name Music --index-name AlbumTitle-index \
                    --contributor-insights-action=DISABLE
```

2. アカウント内のすべての Contributor Insights 設定を一覧表示します。

```
aws dynamodb list-contributor-insights
```

これにより、Contributor Insights が有効になっているすべてのテーブルとインデックス、およびそれらのモードが表示されます。

3. 特定の設定に関する詳細情報を取得します。

```
aws dynamodb describe-contributor-insights \
                    --table-name Music \
                    --index-name AlbumTitle-index
```

レスポンス例

モード間の違いを示すレスポンスの例を次に示します。

アクセスおよびスロットリングされたキーモードのレスポンス

```
{
    "TableName": "Music",
    "ContributorInsightsRuleList": [
        "DynamoDBContributorInsights-PKC-Music-1234567890123",
        "DynamoDBContributorInsights-PKT-Music-1234567890123",
        "DynamoDBContributorInsights-SKC-Music-1234567890123",
        "DynamoDBContributorInsights-SKT-Music-1234567890123"
    ],
    "ContributorInsightsStatus": "ENABLED",
    "ContributorInsightsMode": "ACCESSED_AND_THROTTLED_KEYS",
    "LastUpdateDateTime": "2024-01-15T10:30:00.000Z"
}
```

スロットリングされたキーモードのレスポンス

```
{
    "TableName": "Music",
    "ContributorInsightsRuleList": [
        "DynamoDBContributorInsights-PKT-Music-1234567890123",
        "DynamoDBContributorInsights-SKT-Music-1234567890123"
    ],
    "ContributorInsightsStatus": "ENABLED",
    "ContributorInsightsMode": "THROTTLED_KEYS",
    "LastUpdateDateTime": "2024-01-15T10:35:00.000Z"
}
```

スロットリングされたキーモードのルールが少ないこと (PKT と SKT のみ) に注意してください。これは、より焦点を絞ったモニタリングに対応しています。

IAM と CloudWatch Contributor Insights for DynamoDB の使用

Amazon DynamoDB の Amazon CloudWatch Contributor Insights を初めて有効にすると、DynamoDB は AWS Identity and Access Management (IAM) サービスリンクロールを自動的に作成します。このロール (`AWSServiceRoleForDynamoDBCloudWatchContributorInsights`) により、DynamoDB がユーザーに代わって CloudWatch Contributor Insights ルールを管理できます。このサービスリンクロールは削除しないでください。ロールを削除してしまうと、テーブルまたはグ

ローバルセカンダリインデックスを削除したときに、すべてのユーザー管理ルールがクリーンアップされなくなります。

サービスリンクロールの詳細については、「IAM ユーザーガイド」の「サービスリンクロールの使用」を参照してください。

以下のアクセス権限が必要です。

- CloudWatch Contributor Insights for DynamoDB を有効または無効にするには、テーブルまたはインデックスに対する `dynamodb:UpdateContributorInsights` 許可が必要です。
- DynamoDB グラフの CloudWatch Contributor Insights を表示するには、`cloudwatch:GetInsightRuleReport` の権限が必要です。
- 特定の DynamoDB テーブルまたはインデックスに関して、DynamoDB の CloudWatch Contributor Insights を記述するには、`dynamodb:DescribeContributorInsights` 許可が必要です。
- 各テーブルおよびグローバルセカンダリインデックスに関して、DynamoDB ステータスの CloudWatch Contributor Insights を一覧表示するには、`dynamodb:ListContributorInsights` の権限が必要です。

例: DynamoDB の CloudWatch Contributor Insights を有効または無効にする

以下の IAM ポリシーは、DynamoDB の CloudWatch Contributor Insights を有効または無効にするための許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/contributorinsights.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",
            "Condition": {"StringLike": {"iam:AWSServiceName":
 "contributorinsights.dynamodb.amazonaws.com"}}
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*"
        }
    ]
}
```

KMS キーで暗号化されたテーブルの場合、Contributor Insights を更新するには、ユーザーが kms:Decrypt アクセス許可を持っている必要があります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-
service-role/contributorinsights.dynamodb.amazonaws.com/
AWSServiceRoleForDynamoDBCloudWatchContributorInsights",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName":
 "contributorinsights.dynamodb.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/*"
        },
        {
            "Effect": "Allow",
            "Resource": "arn:aws:kms:*:*:key/*",
            "Action": [
                "kms:Decrypt"
```

```
            ]
        }
    ]
}
```

例: CloudWatch Contributor Insights のルールレポートを取得する

以下の IAM ポリシーは、CloudWatch Contributor Insights のルールレポートを取得するための許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetInsightRuleReport"
            ],
            "Resource": "arn:aws:cloudwatch:*:*:insight-rule/
DynamoDBContributorInsights*"
        }
    ]
}
```

例: リソースに基づいて DynamoDB の CloudWatch Contributor Insights の許可を選択的に適用する

以下の IAM ポリシーは、`ListContributorInsights` と `DescribeContributorInsights` のアクションを許可するアクセス権限を付与し、特定のグローバルセカンダリインデックスに対する `UpdateContributorInsights` アクションを拒否します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListContributorInsights",
                "dynamodb:DescribeContributorInsights"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": [
                "dynamodb:UpdateContributorInsights"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books/
index/Author-index"
        }
    ]
}
```

CloudWatch Contributor Insights for DynamoDB のサービスリンクロールの使用

CloudWatch Contributor Insights for DynamoDB は、AWS Identity and Access Management (IAM) サービスリンクロールを使用します。サービスリンクロールは、CloudWatch Contributor Insights for DynamoDB に直接リンクされた一意のタイプの IAM ロールです。サービスリンクロールは、CloudWatch Contributor Insights for DynamoDB で事前定義され、ユーザーの代わりに該当サービスから他の AWS のサービスを呼び出すために必要なすべての許可が含まれされます。

サービスリンクロールを使用することで、必要なアクセス権限を手動で追加する必要がなくなるため、CloudWatch Contributor Insights for DynamoDB の設定が簡単になります。CloudWatch Contributor Insights for DynamoDB は、サービスリンクロールの許可を定義します。特に定義されている場合を除き、DynamoDB の CloudWatch Contributor Insights のみがその口ールを引き受けることができます。定義される許可は、信頼ポリシーと許可ポリシーに含まれており、その許可ポリシーを他の IAM エンティティにアタッチすることはできません。

サービスにリンクされたロールをサポートする他のサービスについては、「IAM と連携する AWS サービス」を参照して、[サービスにリンクされたロール] 列が [はい] になっているサービスを見つけてください。サービスリンクロールに関するドキュメントをサービスで表示するには、[Yes] (はい) リンクを選択します。

CloudWatch Contributor Insights for DynamoDB のサービスリンクロールの許可

CloudWatch Contributor Insights for DynamoDB は、サービスリンクロール AWSServiceRoleForDynamoDBCloudWatchContributorInsights を使用します。サービスリンクロールの目的は、Amazon DynamoDB が DynamoDB テーブルおよびグローバルセカンダリインデックス用に作成された Amazon CloudWatch Contributor Insights ルールをユーザーに代わって管理できるようにすることです。

`AWSServiceRoleForDynamoDBCloudWatchContributorInsights` サービスリンクロールは、ロールの引き受けについて以下のサービスを信頼します。

- `contributorinsights.dynamodb.amazonaws.com`

このロールの許可ポリシーは、CloudWatch Contributor Insights for DynamoDB が指定されたリソースで以下のアクションを実行できるようにします。

- アクション: `DynamoDBContributorInsights` での `Create and manage Insight Rules`

サービスにリンクされたロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) に許可するには、権限を設定する必要があります。詳細については、IAM ユーザーガイドの「サービスにリンクされたロールのアクセス権限」を参照してください。

CloudWatch Contributor Insights for DynamoDB のサービスリンクロールの作成

サービスリンクロールを手動で作成する必要はありません。AWS マネジメントコンソール、AWS CLI、または AWS API でコントリビューターインサイトを有効にすると、CloudWatch Contributor Insights for DynamoDB によってサービスリンクロールが作成されます。

このサービスリンクロールを削除した後で再度作成する必要が生じた場合は、同じ方法でアカウントにロールを再作成できます。Contributor Insights を有効にすると、CloudWatch Contributor Insights for DynamoDB によって、サービスリンクロールが再度作成されます。

CloudWatch Contributor Insights for DynamoDB のサービスリンクロールの編集

CloudWatch Contributor Insights for DynamoDB では、`AWSServiceRoleForDynamoDBCloudWatchContributorInsights` サービスリンクロールを編集することはできません。サービスリンクロールを作成すると、多くのエンティティによってロールが参照される可能性があるため、ロール名を変更することはできません。ただし、IAM を使用してロールの説明を編集することはできます。詳細については、「IAM ユーザーガイド」の「サービスリンクロールの編集」を参照してください。

CloudWatch Contributor Insights for DynamoDB のサービスリンクロールの削除

`AWSServiceRoleForDynamoDBCloudWatchContributorInsights` ロールを手動で削除する必要はありません。AWS マネジメントコンソール、AWS CLI、または AWS API で Contributor Insights を無効にすると、CloudWatch Contributor Insights for DynamoDB によってリソースがクリーンアップされます。

サービスリンクロールは、IAM コンソール、AWS CLI、または AWS API を使用して手動で削除することもできます。そのためにはまず、サービスリンクロールのリソースをクリーンアップする必要があります。その後で、手動で削除できます。

> **Note**
>
> リソースを削除する際に、CloudWatch Contributor Insights for DynamoDB サービスでロールが使用されている場合、削除が失敗することがあります。失敗した場合は、数分待ってから再度オペレーションを実行してください。

IAM を使用してサービスリンクロールを手動で削除するには

IAM コンソール、AWS CLI、または AWS API を使用して、`AWSServiceRoleForDynamoDBCloudWatchContributorInsights` サービスリンクロールを削除します。詳細については、「IAM ユーザーガイド」の「サービスにリンクされたロールの削除」を参照してください。

DynamoDB を使用した設計とアーキテクチャの設計に関するベストプラクティス

このセクションでは、DynamoDB の使用時にパフォーマンスを最大化してスループットコストを最小限に抑えるための推奨事項をすばやく確認できます。

トピック

- DynamoDB 用の NoSQL
- DynamoDB の Well-Architected レンズを使用して DynamoDB ワークロードを最適化する
- DynamoDB でパーティションキーを効率的に設計し、使用するためのベストプラクティス。
- ソートキーを使用してデータを DynamoDB で整理するためのベストプラクティス
- DynamoDB でセカンダリインデックスを使用するためのベストプラクティス。
- DynamoDB で大きな項目と属性を格納するベストプラクティス
- DynamoDB で時系列データを処理するベストプラクティス。
- DynamoDB テーブルで多対多リレーションシップを管理するためのベストプラクティス
- DynamoDB でのデータのクエリとスキャンのベストプラクティス
- DynamoDB テーブル設計のベストプラクティス
- DynamoDB グローバルテーブルの使用
- DynamoDB でコントロールプレーンを管理するためのベストプラクティス
- DynamoDB での一括データオペレーション使用についてのベストプラクティス
- DynamoDB で同時更新を処理するためのベストプラクティス
- DynamoDB における AWS の請求および使用状況レポートを理解するためのベストプラクティス
- DynamoDB テーブルをアカウントから別のアカウントに移行する
- DAX を DynamoDB アプリケーションと統合するための規範ガイダンス
- DynamoDB での PITR 復旧のベストプラクティス

- AWS PrivateLink for Amazon DynamoDB を使用する場合の考慮事項

DynamoDB 用の NoSQL

Amazon DynamoDB などの NoSQL データベースシステムでは、キーと値のペアやドキュメントストレージなど、データ管理のための代替モデルを使用します。リレーショナルデータベース管理システム (RDBMS) から DynamoDB のような NoSQL データベースシステムに切り替えるときは、主な相違点と特定の設計アプローチを理解することが重要です。

トピック

- リレーショナルデータ設計と NoSQL の相違点
- NoSQL 設計の 2 つの重要な概念
- NoSQL 設計へのアプローチ
- DynamoDB 用の NoSQL Workbench

リレーショナルデータ設計と NoSQL の相違点

リレーショナルデータベースシステム (RDBMS) と NoSQL データベースにはそれぞれ異なる長所と短所があります。

- RDBMS では、データは柔軟にクエリできますが、クエリは比較的コストが高く、トラフィックが多い状況ではスケールがうまくいかない場合があります (DynamoDB でリレーショナルデータをモデル化するための最初のステップ 参照)。
- 一方、DynamoDB のような NoSQL データベースでは、データは限られた数の方法で効率的にクエリできますが、その範囲外では、クエリは高コストで低速になりがちです。

これらの相違点により、2 つのシステム間でデータベース設計が異なるものになります。

- RDBMS では、実装の詳細やパフォーマンスを気にせずに柔軟に設計できます。クエリの最適化は一般的にスキーマ設計には影響しませんが、正規化は重要です。
- DynamoDB では、最も一般的で重要なクエリをできるだけ速く、安価にするために、具体的にスキーマを設計します。データ構造は、ビジネスユースケースの特定の要件に合わせて調整されています。

NoSQL 設計の 2 つの重要な概念

NoSQL 設計では、RDBMS 設計とは異なる考え方が必要です。RDBMS の場合は、アクセスパターンを考慮せずに正規化されたデータモデルを作成できます。その後、新しい課題とクエリの要件が発

生したら、そのデータモデルを拡張することができます。各タイプのデータを独自のテーブルに整理できます。

NoSQL の設計の違い

- 対照的に、DynamoDB の場合は答えが必要な質問が分かるまで、スキーマの設計を開始すべきではありません。ビジネス上の問題とアプリケーションのユースケースを理解することが不可欠です。
- DynamoDB アプリケーションで維持するテーブルはできるだけ少なくする必要があります。テーブルの数が少なくなると、スケーラビリティが向上し、必要なアクセス権限の管理が少なくなり、DynamoDB アプリケーションのオーバーヘッドが削減されます。また、バックアップコストを全体的に低く抑えるのにも役立ちます。

NoSQL 設計へのアプローチ

DynamoDB アプリケーションを設計する最初のステップは、システムが満たす必要がある特定のクエリパターンを見極めることです。

始める前に、特にアプリケーションのアクセスパターンの 3 つの基本的な特性を理解することが重要です。

- データサイズ: 一度に格納され、リクエストされるデータの量を把握することで、データパーティションの最も効果的な方法を決定できます。
- データシェイプ: クエリが処理される際 (RDBMS システムのように) データを再形成するのではなく、データベースの形状がクエリ処理に対応するように、NoSQL データベースでデータを整理します。これは、スピードとスケーラビリティを向上させる重要な要素です。
- データ速度: DynamoDB では、クエリを処理するために使用可能な物理パーティションの数を増やし、それらのパーティション間で効率的にデータを分散させることでスケーリングします。ピーク時のクエリのロードを事前に把握することは、I/O 容量を最大限に活用するためにデータを分割する方法を決定する上で役立ちます。

特定のクエリ要件を特定したら、パフォーマンスを管理する一般的な原則に従ってデータを整理できます。

- 関連データをまとめて保存します。 調査により、関連データを 1 か所にまとめて保存する「参照の局所性」の原則は、NoSQL システムのパフォーマンスと応答時間を向上させる重要な要素であ

ることが判明しています。これは、何年も前にルーティングテーブルを最適化するために重要であることが判明したのと同じです。

一般的なルールとして、DynamoDB アプリケーションでは維持するテーブルをできるだけ少なくする必要があります。

例外として、大キャパシティの時系列データが必要な場合や、データセットのアクセスパターンが大きく異なる場合などがあります。反転されたインデックスを含む単一のテーブルでは通常、シンプルなクエリを使用してアプリケーションで必要とされる複雑な階層データ構造を構築および取得できます。

- ソート順を使用します。　キーの設計が原因でソートされている場合は、関連項目をグループ化して、効率的にクエリできます。これは重要な NoSQL 設計方法です。
- クエリを分散します。　大量のクエリがデータベースの一部に集中しないことも重要です。集中すると、I/O キャパシティを超えるおそれがあります。代わりに、パーティション間でトラフィックをできるだけ均等に分散するようにデータキーを設計し、ホットスポットを避ける必要があります。
- グローバルセカンダリインデックスを使用します。　特定のグローバルセカンダリインデックスを作成することで、メインテーブルでサポートできるクエリとは異なるクエリを有効にすることができ、このクエリは高速で比較的安価です。

このような一般原則は、DynamoDB でデータを効率的にモデル化するために使用できる一般的なデザインパターンに変換されます。

DynamoDB 用の NoSQL Workbench

DynamoDB 用の NoSQL Workbench は、最新のデータベース開発および運用で使用できるクロスプラットフォームのクライアント側 GUI アプリケーションです。Windows、macOS、Linux で使用できます。NoSQL Workbench は、DynamoDB テーブルの設計、作成、クエリ、管理に役立つデータモデリング、データ可視化、サンプルデータ生成、クエリ開発といった特徴を提供する視覚的開発ツールです。DynamoDB 用の NoSQL Workbench を使用すると、アプリケーションのデータアクセスパターンを満たす既存のデータモデルから新しいデータモデルを構築したり、既存のデータモデルに基づいてモデルを設計したりできます。プロセスの最後に、設計されたデータモデルをインポートおよびエクスポートすることもできます。詳細については、「NoSQL Workbench を使用したデータモデルの構築」を参照してください。

DynamoDB の Well-Architected レンズを使用して DynamoDB ワークロードを最適化する

このセクションでは、優れたアーキテクチャの DynamoDB ワークロードを設計するための設計原則とガイダンスの集合である Amazon DynamoDB Well-Architected レンズについて説明します。

DynamoDB テーブルのコストの最適化

このセクションでは、既存の DynamoDB テーブルのコストを最適化する方法のベストプラクティスについて説明します。次の戦略を見て、どのコスト最適化戦略がニーズに最も適しているかを確認し、それらに繰り返しアプローチする必要があります。各戦略には、コストに影響する可能性のある要素、注意が必要な兆候、およびその削減方法に関する規範的なガイダンスの概要が示されています。

トピック

- テーブルレベルでコストを評価する
- DynamoDB テーブルのキャパシティモードの評価
- DynamoDB テーブルの自動スケーリング設定を評価する
- DynamoDB テーブルクラスの選択を評価する
- DynamoDB の未使用のリソースを特定する
- DynamoDB テーブルの使用パターンを評価する
- DynamoDB Streams の使用状況を評価する
- DynamoDB テーブルで適切なサイズのプロビジョニングを行うために、プロビジョンドキャパシティを評価する

テーブルレベルでコストを評価する

AWS マネジメントコンソール 内にある Cost Explorer ツールを使用すると、読み取り、書き込み、ストレージ、バックアップの料金など、コストをタイプ別に分類して確認できます。また、これらのコストを月や日などの期間別にまとめて表示することもできます。

管理者が直面する課題として、ある特定のテーブルのコストのみを確認する必要がある場合があります。一部のデータは DynamoDB コンソールまたは `DescribeTable` API の呼び出しを介して取得することができます。ただし Cost Explorer では、デフォルトでは特定のテーブルに関連するコストで

フィルタリングまたはグループ化することができません。このセクションでは、タグ付けを使用して Cost Explorer で個々のテーブルのコスト分析を実行する方法を説明します。

トピック

- 単一の DynamoDB テーブルのコストを表示する方法
- Cost Explorer のデフォルトビュー
- Cost Explorer でのテーブルタグの使用方法および適用方法

単一の DynamoDB テーブルのコストを表示する方法

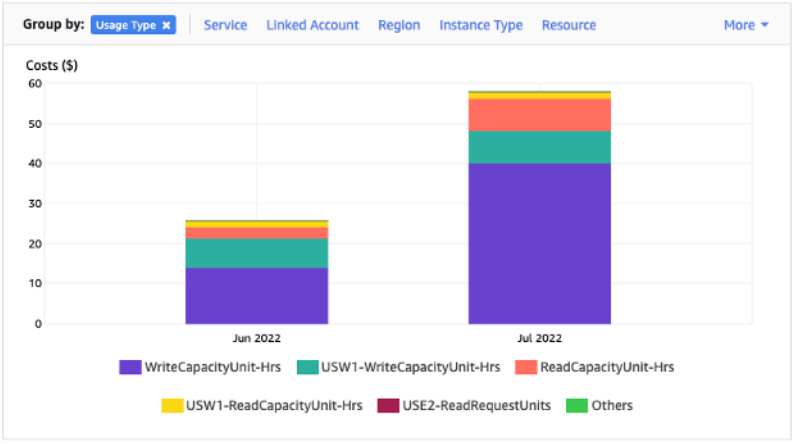
Amazon DynamoDB AWS マネジメントコンソール と`DescribeTable` API はどちらも、プライマリキースキーマ、テーブルのすべてのインデックス、テーブルとインデックスのサイズと項目数など、単一のテーブルに関する情報を表示します。テーブルのサイズとインデックスのサイズから、テーブルの月々のストレージコストを計算することができます。たとえば、us-east-1 リージョンでは 1 GB あたり 0.25 USD です。

テーブルがプロビジョンドキャパシティーモードの場合、現在の RCU と WCU の設定も返されます。これらを使用してテーブルの現在の読み取りと書き込みのコストを計算することもできますが、特にテーブルが Auto Scaling で設定されている場合、これらのコストは変わる可能性があります。

> **Note**
>
> テーブルがオンデマンドキャパシティーモードの場合、`DescribeTable` はスループットコストの見積もりには役立ちません。これはスループットコストが、ある期間内のプロビジョニングされた使用量ではなく、実際の使用量に基づいて請求されるためです。

Cost Explorer のデフォルトビュー

Cost Explorer のデフォルトビューには、スループットやストレージなどの消費されたリソースのコストを示すグラフが表示されます。月ごとや日ごとの合計など、期間ごとにコストをグループ化することを選択できます。ストレージ、読み取り、書き込み、その他の機能のコストも分類して比較できます。



Cost Explorer でのテーブルタグの使用方法および適用方法

デフォルトでは、Cost Explorer は複数のテーブルのコストを合計にまとめるため、特定の 1 つのテーブルのコストの概要は表示されません。ただし、AWS リソースへのタグ付けを使用すると、メタデータタグで各テーブルを識別できます。タグとは、プロジェクトや部門に属するすべてのリソースを識別するなど、さまざまな目的に使用できるキーと値のペアです。この例では、MyTable という名前のテーブルがあると仮定します。

1. table_name のキーと MyTable の値を使用してタグを設定します。
2. Cost Explorer 内でタグをアクティブ化し、タグ値でフィルタリングすると、各テーブルのコストをより明確に把握できます。

> **Note**
>
> タグが Cost Explorer に表示され始めるまでには、1 ~ 2 日かかる場合があります。

メタデータタグは、コンソールで自分で設定することも、AWS CLI やAWS SDK などのオートメーションを使用して設定することもできます。組織の新しいテーブル作成プロセスの一環として、table_name タグの設定を義務付けることを検討してください。既存のテーブルについては、これらのタグを検索してアカウントの特定のリージョンにあるすべての既存のテーブルに適用する Python ユーティリティが用意されています。詳細については、GitHub の「Eponymous Table Tagger」を参照してください。

DynamoDB テーブルのキャパシティモードの評価

このセクションでは、DynamoDB テーブルで、適切なキャパシティモードを選択する方法の概要を説明します。各モードは、スループットの変化に対する応答性や使用に対する請求方法という点で、さまざまなワークロードのニーズを満たすように調整されています。決定を下す際には、これらの要素のバランスを取る必要があります。

トピック

- 利用可能なテーブルキャパシティモード
- オンデマンドキャパシティモードを選択する場合
- プロビジョンドキャパシティモードを選択する場合
- テーブルキャパシティモードを選択する際に考慮すべきその他の要素

利用可能なテーブルキャパシティモード

DynamoDB テーブルを作成する場合、オンデマンドまたはプロビジョンドキャパシティモードのいずれかを選択する必要があります。

24 時間のローリングウィンドウで、テーブルをプロビジョンドキャパシティモードからオンデマンドモードに最大 4 回切り替えることができます。テーブルは、オンデマンドモードからプロビジョンドキャパシティモードにいつでも切り替えることができます。

オンデマンドキャパシティモード

オンデマンドキャパシティモードは、DynamoDB テーブルのキャパシティをプランニングまたはプロビジョニングする必要がないように設計されています。このモードでは、テーブルへのリクエストにテーブルが即座に対応します。リソースをスケールアップまたはスケールダウンする必要はありません (テーブルの以前のピークスループットの最大 2 倍まで)。

DynamoDB オンデマンドは、読み込みおよび書き込みリクエストの料金が従量制であるため、使用した分だけを支払います。

プロビジョンドキャパシティモード

プロビジョンドキャパシティモードは従来型のモデルで、リクエストに対して利用できるテーブルのキャパシティを直接、または自動スケーリングを使って定義する必要があります。特定のキャパシティが指定された時間にテーブルにプロビジョニングされるため、請求は消費されたリクエストの数ではなく、プロビジョニングされたキャパシティの合計に基づいて行われます。割り当てられたキャ

パシティを超えると、テーブルがリクエストを拒否し、アプリケーションユーザーのエクスペリエンスが低下する可能性があります。

プロビジョンドキャパシティモードでは、テーブルのオーバープロビジョニングや過少プロビジョニングを避けるために継続的にモニタリングしてバランスを図る必要があります。スロットリングを低く抑え、コストを調整するためです。

オンデマンドキャパシティモードを選択する場合

コストを最適化する際、次のグラフのようなワークロードを実行する場合は、オンデマンドモードが最適です。

この種のワークロードには、次の要素が影響します。

- 時間の経過とともに進化するトラフィックパターン
- リクエストの量が変動する (バッチワークロードに起因)
- リクエストのタイミングが予測できない (トラフィックの急増につながる)
- 特定の時間でピーク時のゼロから 30% 未満に低下



上記の要素があるワークロードでは、トラフィックの急増に対応するために自動スケーリングを使用して十分なキャパシティをテーブルに維持しようとすると、テーブルが過剰にプロビジョニングされて必要以上にコストがかかったり、テーブルがプロビジョニング不足でリクエストが必要以上にスロットリングされたりする可能性があります。オンデマンドキャパシティモードは、キャパシティを予測または調整することなく、変動するトラフィックを処理できるため、より適しています。

オンデマンドモードのリクエストごとの支払い料金モデルでは、実際に使用したスループットに対してのみ料金を支払うため、アイドル状態のキャパシティについて心配する必要はありません。消費した読み取りまたは書き込みリクエストごとに課金されるため、コストは実際の使用量を直接反映します。そのため、コストとパフォーマンスのバランスを簡単に取ることができます。オプションで、個々のオンデマンドテーブルとグローバルセカンダリインデックスに対して 1 秒あたりの読み込み

や書き込み (または両方) の最大スループットを設定して、コストと使用量を制限し続けることもできます。詳細については、「オンデマンドテーブルの最大スループット」を参照してください。

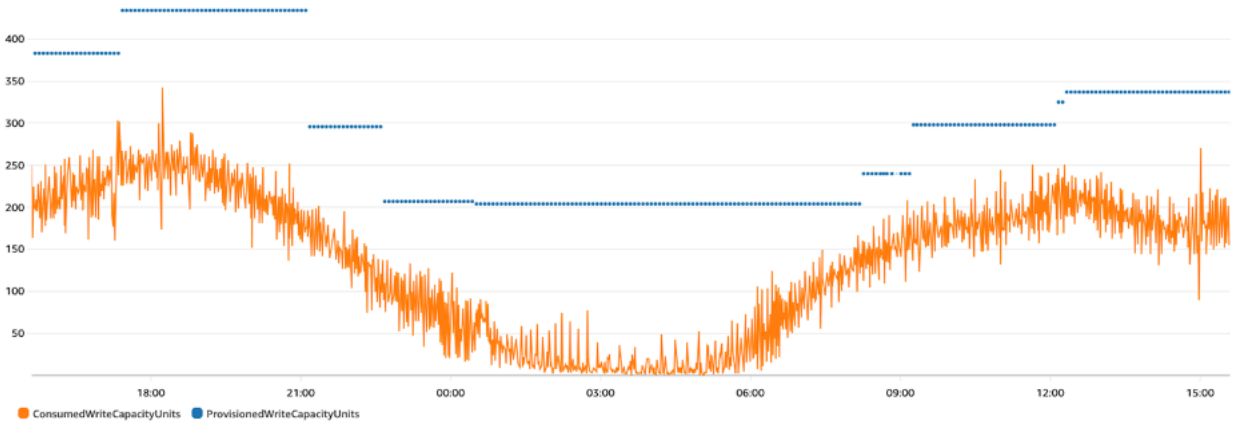
プロビジョンドキャパシティモードを選択する場合

プロビジョンドキャパシティモードに最適なワークロードは、以下のグラフのように、使用パターンがより安定しており、予測可能な場合です。

> **Note**
>
> プロビジョンドキャパシティでアクションを実行する前に、14 日などの短い期間でメトリクスを確認することをお勧めします。

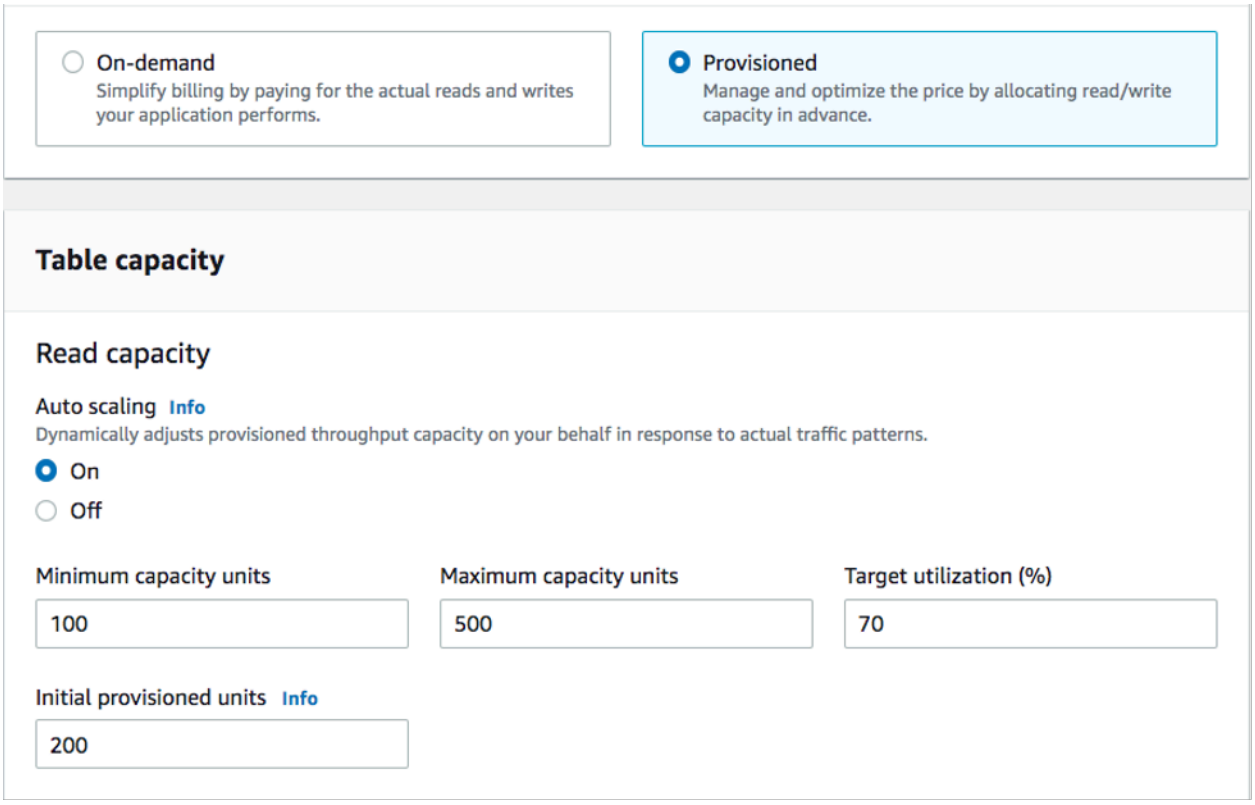
この種のワークロードには、次の要素が影響します。

- 特定の時間または日における予測可能で安定した周期的なトラフィック
- トラフィックの急増が短期間で限定的



一定時間内または 1 日のトラフィック量が安定しているため、テーブルのプロビジョンドキャパシティは、テーブルで実際に消費されるキャパシティに比較的近い値に設定できます。プロビジョンドキャパシティテーブルのコストを最適化することは、最終的には、テーブル上で `ThrottledRequests` を増やすことなく、プロビジョンドキャパシティ (青色の線) を消費キャパシティ (オレンジ色の線) にできるだけ近づけるための練習になります。2 つの線の間にある空白は、未使用のキャパシティであると同時に、スロットリングによるユーザーエクスペリエンスの低下に備えた保険でもあります。アプリケーションのスループット要件を予測でき、読み取りキャパシティと書き込みキャパシティを制御するコスト予測可能性を希望する場合は、プロビジョニングされたテーブルを引き続き使用することをお勧めします。

DynamoDB は、Auto Scaling を提供しており、ユーザーに代わって自動的にプロビジョンドキャパシティテーブルのバランスを調整します。これにより、1 日を通して消費されたキャパシティを追跡し、いくつかの変数に基づいてテーブルのキャパシティを設定できます。自動スケーリングを使用する場合、テーブルは過剰にプロビジョニングされるため、ワークロードのニーズに合わせてスロットリング数と過剰にプロビジョニングされたキャパシティユニットの比率をファインチューニングする必要があります。



キャパシティユニットの最小数

テーブルの最小キャパシティを設定してスロットリングを制限することはできますが、これによってテーブルのコストが削減されるわけではありません。テーブルの使用量が少ない期間が続いた後に突然使用量が急増した場合、最小値を設定しておくと、Auto Scaling がテーブルキャパシティを低く設定しすぎるのを防ぐことができます。

キャパシティユニットの最大数

テーブルの最大キャパシティを設定すると、テーブルが意図したより大きくスケールしてしまうことを制限できます。大規模な負荷テストを実施しない開発テーブルまたはテストテーブルには最大値を適用することを検討してください。最大数はどのテーブルにも設定できますが、本番環境で使用する場合は、誤ってスロットリングされないように、この設定をテーブルのベースラインと照らし合わせて定期的に評価してください。

目標使用率

テーブルの目標使用率を設定することは、プロビジョンドキャパシティテーブルのコストを最適化するための主な手段です。ここでパーセント値を低く設定すると、テーブルでオーバープロビジョニングされるキャパシティが増え、コストが増加しますが、スロットリングのリスクは軽減されます。パーセント値を高く設定すると、テーブルでオーバープロビジョニングされるキャパシティは減少しますが、スロットリングのリスクは増大します。

テーブルキャパシティモードを選択する際に考慮すべきその他の要素

2 つのモードのどちらかを決める際には、考慮すべき点が他にもいくつかあります。

プロビジョンドキャパシティの使用率

オンデマンドモードのコストがプロビジョンドキャパシティよりも安くなるタイミングを判断するには、プロビジョンドキャパシティの使用率を調べると便利です。これは、割り当てられた (またはプロビジョニングされた) リソースがどの程度効率的に使用されているかを示します。オンデマンドモードのコストは、プロビジョニングされた平均キャパシティ使用率が 35% 未満のワークロードで低くなります。多くの場合、プロビジョンドキャパシティ使用率が 35% を超えるワークロードでも、オンデマンドモードを使用する方が費用対効果が高くなります。特に、ワークロードのアクティビティが少ない期間とピークが混在している場合はそう言えます。

リザーブドキャパシティ

プロビジョンドキャパシティテーブルの場合、DynamoDB では読み込みおよび書き込みキャパシティ用にリザーブドキャパシティを購入できます (レプリケートされた書き込みキャパシティユニット (rWCU) と Standard-IA テーブルは現在対象外です)。リザーブドキャパシティは、標準のプロビジョンドキャパシティ料金を大幅に割引します。

2 つのテーブルモードのどちらかを決める際には、この追加の割引がテーブルのコストに与える影響を考慮してください。場合によっては、比較的予測が困難なワークロードでも、リザーブドキャパシティを利用してオーバープロビジョニングされたプロビジョンドキャパシティテーブルで実行する方が安く済む場合があります。

ワークロードの予測可能性の向上

状況によっては、ワークロードに予測可能なパターンと予測不可能なパターンの両方があるように見えることがあります。これはオンデマンドテーブルで簡単にサポートできますが、ワークロードの予測不可能なパターンを改善できれば、コストも改善できる可能性があります。

これらのパターンの最も一般的な原因の 1 つは、バッチインポートです。このタイプのトラフィックでは、ワークロードを実行するとスロットリングが発生してしまうほど、テーブルのベースライン

キャパシティを超えることがよくあります。このようなワークロードをプロビジョンドキャパシティテーブルで実行し続ける場合は、次のオプションを検討してください。

- スケジュールされた時間にバッチが行われる場合は、バッチの実行前に Auto Scaling のキャパシティを増やすようにスケジュールできます。
- バッチがランダムに行われる場合は、できるだけ速く実行するよりも、実行する時間を延長することを検討してください。
- 自動スケーリングでテーブルキャパシティの調整を開始できるようになるまで、インポートを低速で開始し、数分間かけてゆっくりと速度を上げていくランプアップ時間をインポートに追加します。

DynamoDB テーブルの自動スケーリング設定を評価する

このセクションでは、DynamoDB テーブルの Auto Scaling 設定を評価する方法の概要を説明します。Amazon DynamoDB Auto Scaling は、アプリケーショントラフィックとターゲット使用率メトリクスに基づいてテーブルとグローバルセカンダリインデックス (GSI) のスループットを管理する機能です。これにより、テーブルや GSI がアプリケーションパターンに必要なキャパシティを確保できるようになります。

AWS Auto Scaling サービスは現在のテーブル使用率をモニタリングし、ターゲット使用率値である `TargetValue` と比較します。割り当てられたキャパシティを増減する時期になると、通知されます。

トピック

- Auto Scaling 設定について
- 目標使用率が低い (50% 未満) テーブルを特定する方法
- 季節変動のあるワークロードに対処する方法
- パターンが不明な急増するワークロードに対処する方法
- リンクされたアプリケーションのワークロードに対応する方法

Auto Scaling 設定について

ターゲット使用率、初期ステップ、最終値の正しい値を定義するには、運用チームの関与が必要です。これにより、AWS Auauto Scaling ポリシーのトリガーに使用されるアプリケーションの使用履歴に基づいて値を適切に定義することができます。ターゲット使用率は、Auto Scaling ルールが適用される前の期間に達成する必要がある合計キャパシティの割合です。

高いターゲット使用率 (90% 前後) を設定すると、Auto Scaling が起動する前に、一定期間トラフィックが 90% を超える必要があります。アプリケーションが常に一定で、トラフィックが急増しない場合を除いて、高いターゲット使用率を使用しないでください。

非常に低い使用率 (50% 未満) を設定する場合は、アプリケーションが Auto Scaling ポリシーをトリガーする前に、プロビジョニングされたキャパシティの 50% に達する必要があります。アプリケーショントラフィックが非常に速い速度で増加しない限り、これは通常、キャパシティの未使用およびリソースの浪費につながります。

目標使用率が低い (50% 未満) テーブルを特定する方法

AWS CLI または AWS マネジメントコンソール を使用して、DynamoDB リソース内の Auto Scaling ポリシーの `TargetValues` をモニタリングおよび識別できます。

AWS CLI

1. リソースのリスト全体を返すには、次のコマンドを実行します。

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb
   ```

   このコマンドは、任意の DynamoDB リソースに発行された Auto Scaling ポリシーのリスト全体を返します。特定のテーブルからのみリソースを取得する場合は、`–resource-id parameter` を追加できます。例えば、次のようになります。

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb --resource-id "table/<table-name>”
   ```

2. 特定の GSI の Auto Scaling ポリシーのみを返すには、次のコマンドを実行します。

   ```
   aws application-autoscaling describe-scaling-policies --service-namespace
    dynamodb --resource-id "table/<table-name>/index/<gsi-name>”
   ```

   Auto Scaling ポリシーで注目している値は以下のとおりです。オーバープロビジョニングを避けるため、ターゲット値を 50% 以上に設定したいと考えています。次のような結果を取得します。

   ```
   {
       "ScalingPolicies": [
           {
   ```

```
            "PolicyARN": "arn:aws:autoscaling:<region>:<account-
id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-
name>:policyName/$<full-gsi-name>-scaling-policy",
            "PolicyName": $<full-gsi-name>”,
            "ServiceNamespace": "dynamodb",
            "ResourceId": "table/<table-name>/index/<index-name>",
            "ScalableDimension": "dynamodb:index:WriteCapacityUnits",
            "PolicyType": "TargetTrackingScaling",
            "TargetTrackingScalingPolicyConfiguration": {
                "TargetValue": 70.0,
                "PredefinedMetricSpecification": {
                    "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
                }
            },
            "Alarms": [
                ...
            ],
            "CreationTime": "2022-03-04T16:23:48.641000+10:00"
        },
        {
            "PolicyARN": "arn:aws:autoscaling:<region>:<account-
id>:scalingPolicy:<uuid>:resource/dynamodb/table/<table-name>/index/<index-
name>:policyName/$<full-gsi-name>-scaling-policy",
            "PolicyName":$<full-gsi-name>”,
            "ServiceNamespace": "dynamodb",
            "ResourceId": "table/<table-name>/index/<index-name>",
            "ScalableDimension": "dynamodb:index:ReadCapacityUnits",
            "PolicyType": "TargetTrackingScaling",
            "TargetTrackingScalingPolicyConfiguration": {
                "TargetValue": 70.0,
                "PredefinedMetricSpecification": {
                    "PredefinedMetricType": "DynamoDBReadCapacityUtilization"
                }
            },
            "Alarms": [
                ...
            ],
            "CreationTime": "2022-03-04T16:23:47.820000+10:00"
        }
    ]
}
```

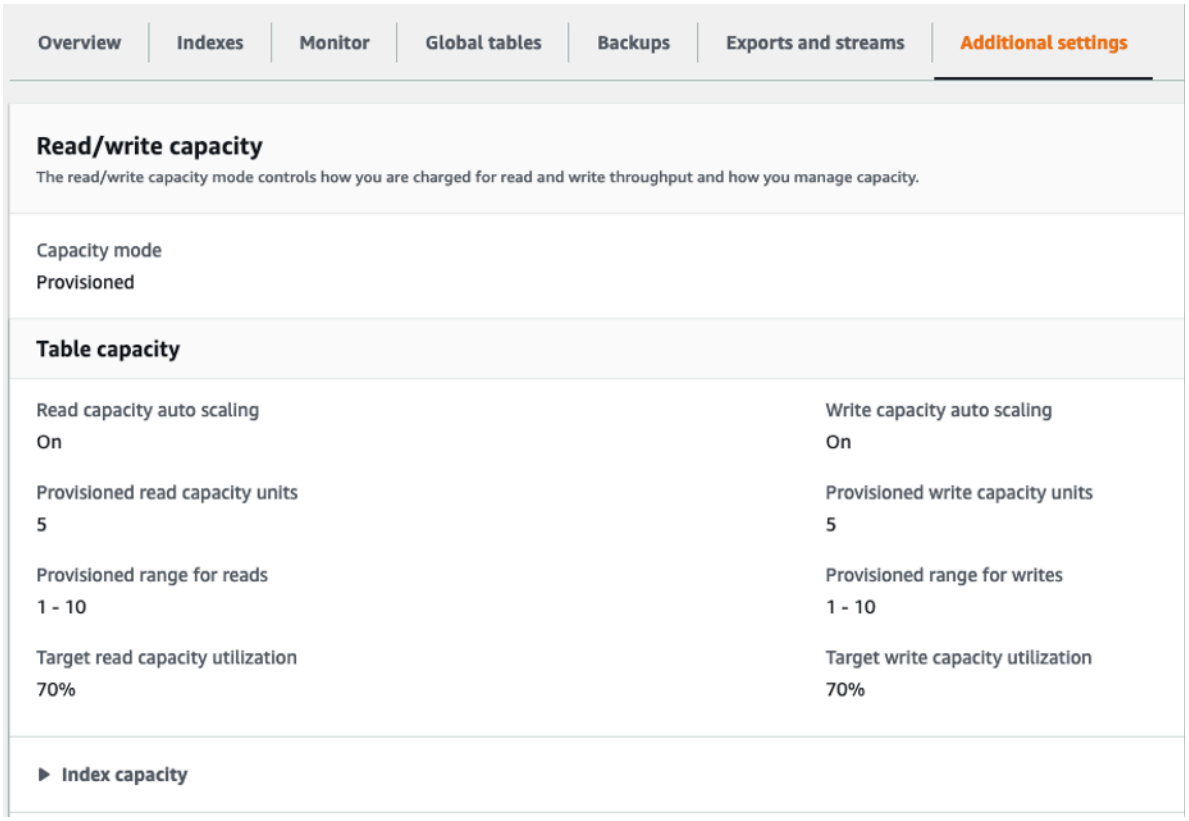
AWS マネジメントコンソール

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。

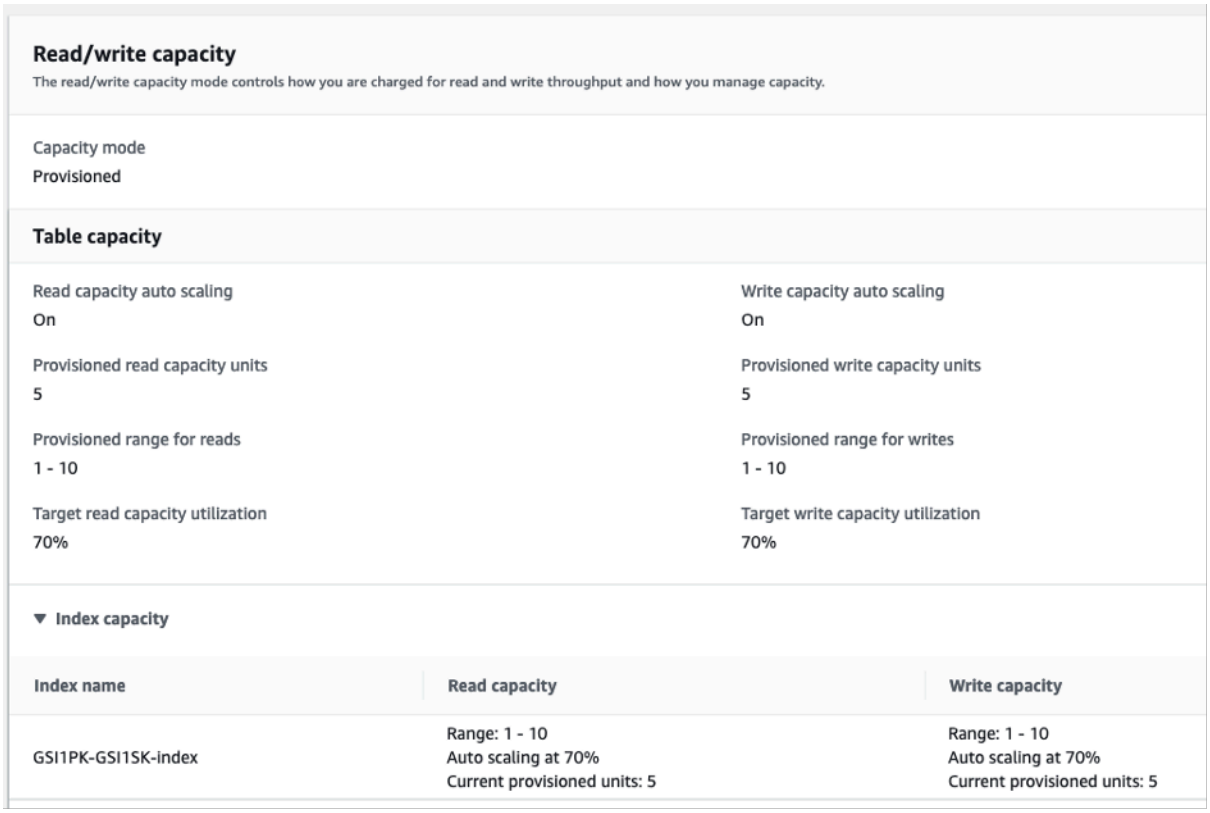
   必要に応じて、適切な AWS リージョン を選択します。

2. 左のナビゲーションバーで、[Tables (テーブル)] を選択します。[Tables (テーブル)] ページで、テーブルの [Name (名)] を選択します。

3. [テーブルの詳細] ページで [追加の設定] を選択して、テーブルの Auto Scaling 設定を確認します。



   インデックスの場合は、[インデックスキャパシティー] セクションを展開して、インデックスの Auto Scaling 設定を確認します。



目標使用率が 50% 以下の場合は、テーブルの使用率メトリクスを調べて、プロビジョニングが不十分か過剰かを確認する必要があります。

季節変動のあるワークロードに対処する方法

次のシナリオを考えてみましょう。アプリケーションはほとんどの場合最小平均値で動作していますが、目標使用率は低いため、アプリケーションは 1 日の特定の時間に発生するイベントに迅速に対応でき、十分なキャパシティがあり、スロットリングを回避できます。これは、アプリケーションが通常の勤務時間 (午前 9 時 ~ 午後 5 時) には非常にビジーで、勤務時間外には基本的なレベルで動作する場合に一般的なシナリオです。一部のユーザーは午前 9 時前に接続を開始するため、アプリケーションはこの低いしきい値を使用して、ピーク時に必要なキャパシティに達するようにすばやく拡張します。

このシナリオは次のようなものです。

- 午後 5 時から午前 9 時までの間、`ConsumedWriteCapacity` ユニットの単位は 90 から 100 の間にとどまる
- ユーザーが午前 9 時前にアプリケーションに接続し始めると、キャパシティユニットが大幅に増加する (最大値は 1500 WCU)

- 勤務時間中のアプリケーション使用量は、平均して 800 ~ 1,200 の間で推移する

前述のシナリオが当てはまる場合は、スケジュールされた Auto Scaling の使用を検討してください。この場合、テーブルにはアプリケーションの Auto Scaling ルールを設定できますが、ターゲット使用率はそれほど高くなく、必要な間隔で追加のキャパシティのみをプロビジョニングできます。

AWS CLI を使用して以下の手順を実行し、時間帯と曜日に基づいて実行されるスケジュールされた Auto Scaling ルールを作成することができます。

1. DynamoDB テーブルまたは GSI をスケーラブルなターゲットとして Application Auto Scaling に登録します。スケーラブルなターゲットは、Application Auto Scaling でスケールアウトまたはスケールインできるリソースです。

```
aws application-autoscaling register-scalable-target \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --min-capacity 90 \
    --max-capacity 1500
```

2. 要件に従ってスケジュールされたアクションをセットアップします。

   このシナリオに対応するには、2 つのルールが必要です。1 つはスケールアップ用、もう 1 つはスケールダウン用です。スケジュールされたアクションをスケールアップするための最初のルール:

```
aws application-autoscaling put-scheduled-action \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
    --resource-id table/<table-name> \
    --scheduled-action-name my-8-5-scheduled-action \
    --scalable-target-action MinCapacity=800,MaxCapacity=1500 \
    --schedule "cron(45 8 ? * MON-FRI *)" \
    --timezone "Australia/Brisbane"
```

   スケジュールされたアクションをスケールダウンするための 2 つ目のルール:

```
aws application-autoscaling put-scheduled-action \
    --service-namespace dynamodb \
    --scalable-dimension dynamodb:table:WriteCapacityUnits \
```

```
    --resource-id table/<table-name> \
    --scheduled-action-name my-5-8-scheduled-down-action \
    --scalable-target-action MinCapacity=90,MaxCapacity=1500 \
    --schedule "cron(15 17 ? * MON-FRI *)" \
    --timezone "Australia/Brisbane"
```

3. 次のコマンドを実行して、両方のルールがアクティブになっていることを確認します。

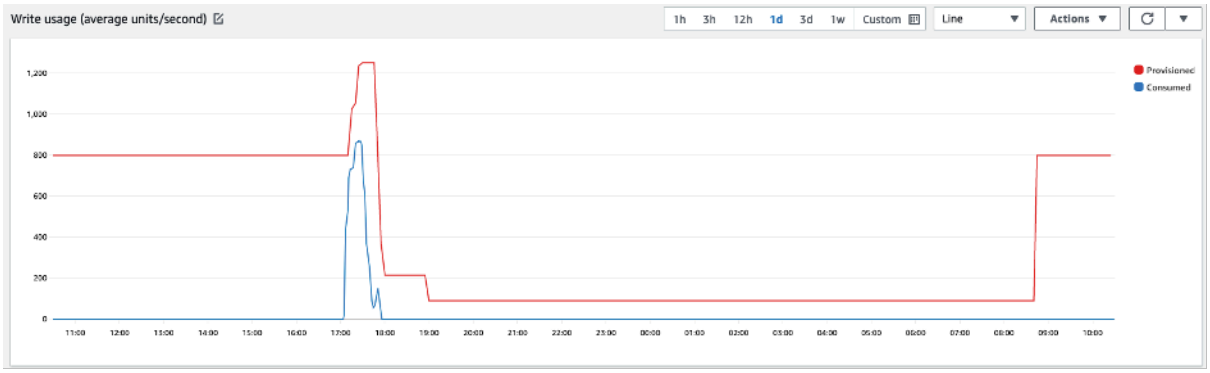
```
aws application-autoscaling describe-scheduled-actions --service-namespace dynamodb
```

次のような結果が表示されます。

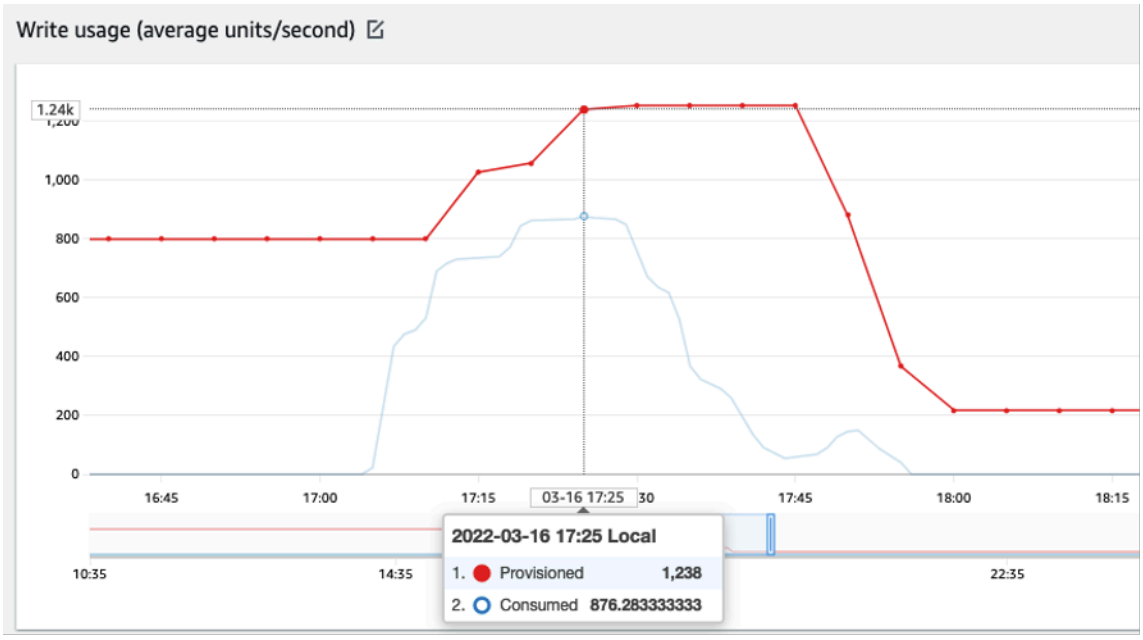
```
{
    "ScheduledActions": [
        {
            "ScheduledActionName": "my-5-8-scheduled-down-action",
            "ScheduledActionARN":
 "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/
table/<table-name>:scheduledActionName/my-5-8-scheduled-down-action",
            "ServiceNamespace": "dynamodb",
            "Schedule": "cron(15 17 ? * MON-FRI *)",
            "Timezone": "Australia/Brisbane",
            "ResourceId": "table/<table-name>",
            "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
            "ScalableTargetAction": {
                "MinCapacity": 90,
                "MaxCapacity": 1500
            },
            "CreationTime": "2022-03-15T17:30:25.100000+10:00"
        },
        {
            "ScheduledActionName": "my-8-5-scheduled-action",
            "ScheduledActionARN":
 "arn:aws:autoscaling:<region>:<account>:scheduledAction:<uuid>:resource/dynamodb/
table/<table-name>:scheduledActionName/my-8-5-scheduled-action",
            "ServiceNamespace": "dynamodb",
            "Schedule": "cron(45 8 ? * MON-FRI *)",
            "Timezone": "Australia/Brisbane",
            "ResourceId": "table/<table-name>",
            "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
            "ScalableTargetAction": {
                "MinCapacity": 800,
                "MaxCapacity": 1500
```

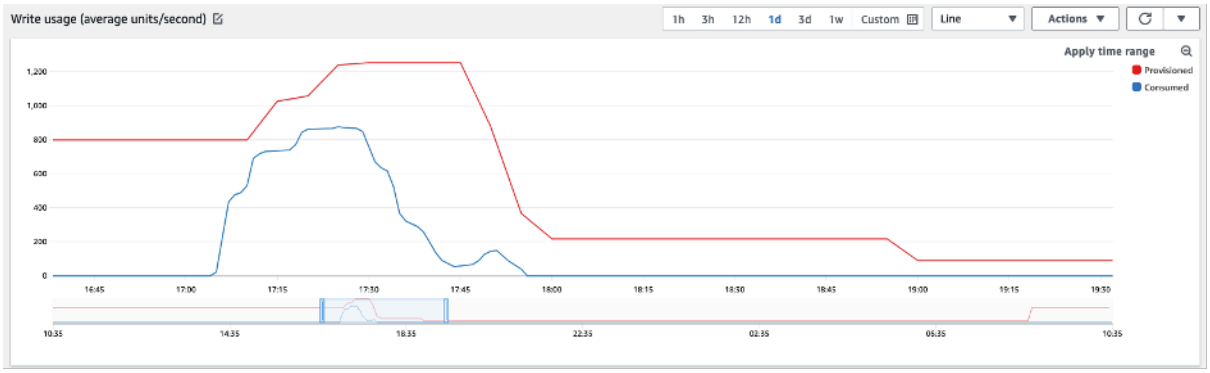
```
            },
            "CreationTime": "2022-03-15T17:28:57.816000+10:00"
        }
    ]
}
```

次の図は、常に 70% のターゲット使用率を維持するサンプルワークロードを示しています。Auto Scaling ルールが適用され、スループットが低下していないことに注目してください。



拡大すると、アプリケーションにスパイクが発生して自動スケーリングのしきい値が 70% に達し、自動スケーリングが強制的に開始され、テーブルに必要な追加キャパシティが提供されたことがわかります。スケジュールされた Auto Scaling アクションは最大値と最小値に影響するので、設定するのはユーザーの責任です。





パターンが不明な急増するワークロードに対処する方法

このシナリオでは、アプリケーションのパターンがまだわからないため、アプリケーションのターゲット使用率が非常に低く、ワークロードがスロットリングされないようにする必要があります。

代わりにオンデマンドキャパシティモードを使用することを検討してください。オンデマンドテーブルは、トラフィックパターンが不明で急増するワークロードに最適です。オンデマンドキャパシティモードでは、アプリケーションがテーブルに対して実行するデータの読み取りと書き込みに対して、リクエストごとに料金を支払います。DynamoDB はワークロードの増減に即座に対応するため、アプリケーションに期待する読み取りと書き込みのスループットを指定する必要はありません。

リンクされたアプリケーションのワークロードに対応する方法

このシナリオでは、アプリケーションは他のシステムに依存します。例えば、バッチ処理シナリオでは、アプリケーションロジックのイベントに応じてトラフィックが大幅に急増する可能性があります。

特定のニーズに応じてテーブルのキャパシティと `TargetValues` を増やすことができる、これらのイベントに反応するカスタムの Auto Scaling ロジックの開発を検討してください。Amazon EventBridge のメリットを得たり、Lambda や Step Functions のような AWS を組み合わせることで、特定のアプリケーションのニーズに対応することができます。

DynamoDB テーブルクラスの選択を評価する

このセクションでは、DynamoDB テーブルで、適切なテーブルクラスを選択する方法の概要を説明します。Standard-Infrequent Access (Standard-IA) テーブルクラスのリリースにより、テーブルを最適化してストレージコストまたはスループットコストを削減できるようになりました。

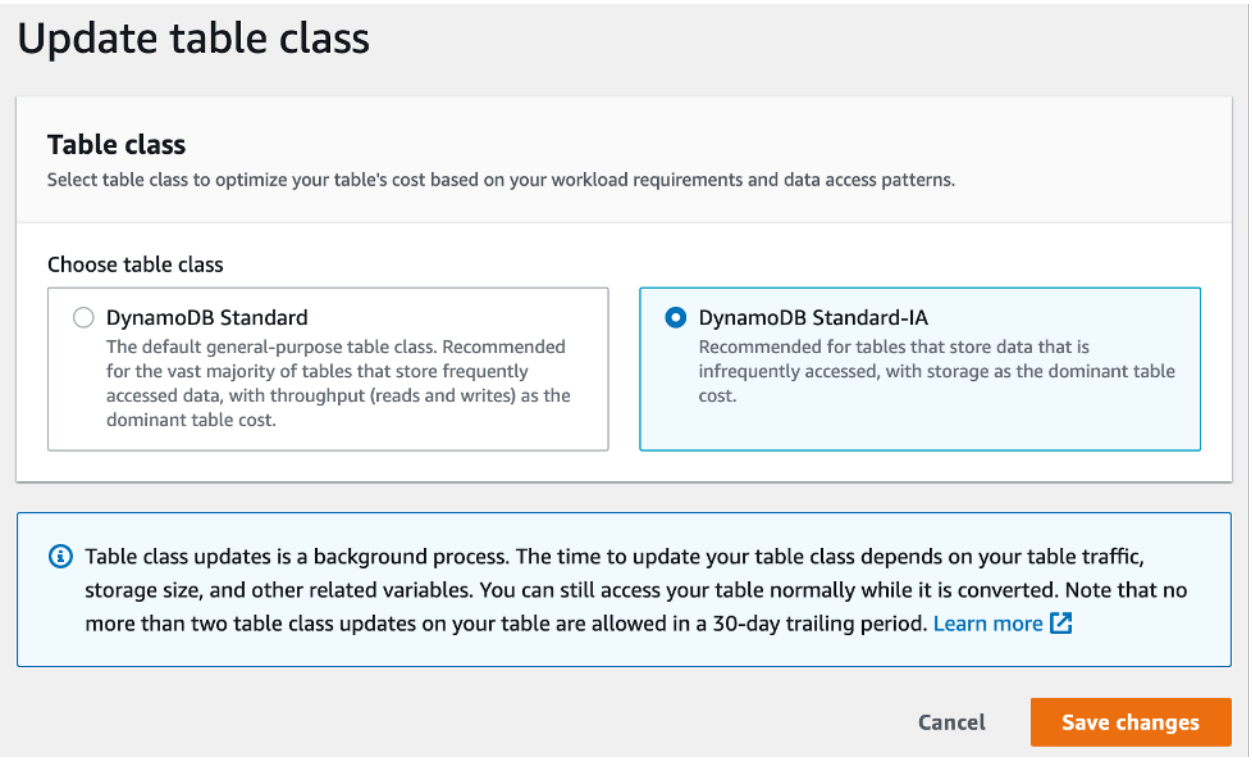
トピック

- 利用可能なテーブルクラス

- DynamoDB Standard テーブルクラスを選択する場合
- DynamoDB Standard-IA テーブルクラスを選択する場合
- テーブルクラスを選択する際に考慮すべきその他の要素

利用可能なテーブルクラス

DynamoDB テーブルを作成する場合は、テーブルクラスに DynamoDB Standard または DynamoDB Standard-IA のいずれかを選択する必要があります。テーブルクラスは 30 日間に 2 回変更できるため、将来いつでも変更できます。どちらのテーブルクラスを選択しても、テーブルのパフォーマンス、可用性、信頼性、耐久性には影響しません。



Standard テーブルクラス

Standard テーブルクラスは新しいテーブルのデフォルトのオプションです。このオプションでは、スループットと、アクセス頻度の高いデータを含むテーブルのストレージコストのバランスが取れる DynamoDB の元の請求残高が維持されます。

Standard-IA テーブルクラス

Standard-IA テーブルクラスは、更新や読み込みの頻度が低いデータの長期保存を必要とするワークロードのストレージコストを削減 (最大 60% 削減) するのに役立ちます。このクラスはアクセス頻度

の低いクラス向けに最適化されているため、読み込みと書き込みの料金は Standard テーブルクラスよりも若干高く (最大 25% 高い) なります。

DynamoDB Standard テーブルクラスを選択する場合

DynamoDB Standard テーブルクラスは、ストレージコストがテーブルの月額総コストの約 50% 以下のテーブルに最適です。このコストバランスは、DynamoDB 内に既に保存されている項目に定期的にアクセスするか、項目を更新するワークロードを示しています。

DynamoDB Standard-IA テーブルクラスを選択する場合

DynamoDB Standard-IA テーブルクラスは、ストレージコストがテーブルの月額総コストの約 50% 以上であるテーブルに最適です。このコストバランスは、1 か月あたりの項目の作成または読み込みがストレージに保持される量よりも少ないワークロードを示しています。

Standard-IA テーブルクラスの一般的な使用方法は、アクセス頻度の低いデータを個々の Standard-IA テーブルに移動することです。詳細については、「Optimizing the storage costs of your workloads with Amazon DynamoDB Standard-IA table class (Amazon DynamoDB Standard-IA テーブルクラスによるワークロードのストレージコストの最適化)」を参照してください。

テーブルクラスを選択する際に考慮すべきその他の要素

2 つのテーブルクラスのどちらかを決める際には、決定するうえで考慮すべき要素が他にもいくつかあります。

リザーブドキャパシティ

Standard-IA テーブルクラスを使用するテーブルのリザーブドキャパシティの購入は、現在サポートされていません。リザーブドキャパシティを利用している Standard テーブルからリザーブドキャパシティを利用していない Standard-IA テーブルに移行する場合、コスト上のメリットが得られない可能性があります。

DynamoDB の未使用のリソースを特定する

このセクションでは、未使用のリソースを定期的に評価する方法の概要を説明します。アプリケーションの要件が変化するにつれて、未使用のリソースによって不要な Amazon DynamoDB のコストが生じないようにする必要があります。以下に説明する手順では、Amazon CloudWatch メトリクスを使用して未使用のリソースを特定します。さらにこの手順は、リソースを特定してコストを削減するための対策を講じるのに役立ちます。

CloudWatch を使用して DynamoDB をモニタリングすることで、DynamoDB から raw データを収集し、リアルタイムに近い読み込み可能なメトリクスに加工することができます。これらの統計は一定

期間保持されるため、履歴情報にアクセスして使用率をより正確に調べることができます。デフォルトでは、DynamoDB メトリクスデータは CloudWatch に自動的に送信されます。詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon CloudWatch とは」および「メトリクスの保持」を参照してください。

トピック

- 未使用のリソースを特定する方法
- 未使用のテーブルリソースを特定する
- 未使用のテーブルリソースをクリーンアップする
- 未使用の GSI リソースを特定する
- 未使用の GSI リソースをクリーンアップする
- 未使用のグローバルテーブルをクリーンアップする
- 未使用のバックアップまたはポイントインタイムリカバリ (PITR) をクリーンアップする

未使用のリソースを特定する方法

未使用のテーブルやインデックスを特定するには、30 日間にわたって次の CloudWatch メトリクスを調べ、テーブルに対するアクティブな読み込みまたは書き込み、あるいはグローバルセカンダリインデックス (GSI) の読み込みがあるかどうかを調べます。

ConsumedReadCapacityUnits

指定された期間に消費された読み込みキャパシティユニットの数。消費されたキャパシティの使用量を追跡できます。テーブルとそのすべてのグローバルセカンダリインデックス、または特定のグローバルセカンダリインデックスの消費された読み込み容量の合計を取得できます。

ConsumedWriteCapacityUnits

指定された期間に消費された書き込みキャパシティユニットの数。消費されたキャパシティの使用量を追跡できます。テーブルとそのすべてのグローバルセカンダリインデックス、または特定のグローバルセカンダリインデックスの消費された書き込み容量の合計を取得できます。

未使用のテーブルリソースを特定する

Amazon CloudWatch は、モニタリングおよびオブザーバビリティサービスで、未使用のリソースの特定に使用する DynamoDB テーブルのメトリクスを提供します。CloudWatch メトリクスは、AWS マネジメントコンソール と AWS Command Line Interface を使って表示できます。

AWS Command Line Interface

AWS Command Line Interface を使ってテーブルのメトリクスを表示する場合は、次のコマンドを使用できます。

1. まず、テーブルの読み込みを評価します。

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
```

テーブルが未使用と誤って識別されないようにするには、長期間にわたってメトリクスの評価を行います。たとえば、30 日間と、適切な開始時刻と終了時刻の範囲、および 86400 などで適切な期間を選択します。

返されるデータで、[合計] が 0 を超える場合は、評価対象のテーブルがその期間に受信した読み込みトラフィックを示します。

次の結果は、評価期間中に読み込みトラフィックを受信したテーブルを示しています。

```
        {
            "Timestamp": "2022-08-25T19:40:00Z",
            "Sum": 36023355.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2022-08-12T19:40:00Z",
            "Sum": 38025777.5,
            "Unit": "Count"
        },
```

次の結果は、評価期間中に読み込みトラフィックを受信しなかったテーブルを示しています。

```
        {
            "Timestamp": "2022-08-01T19:50:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
        {
```

```
            "Timestamp": "2022-08-20T19:50:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
```

2. 次に、テーブルの書き込みを評価します。

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedWriteCapacityUnits --start-time <start-time> --end-time <end-
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
```

テーブルが未使用と誤って識別されないようにするには、長期間にわたってメトリクスを評価する必要があります。30 日間などの適切な開始時刻と終了時刻の範囲、および 86400 などの適切な期間を選択します。

返されるデータで、[Sum (合計)] が 0 を超える場合は、評価対象のテーブルがその期間に受信した読み込みトラフィックを示します。

次の結果は、評価期間中に書き込みトラフィックを受信したテーブルを示しています。

```
        {
            "Timestamp": "2022-08-19T20:15:00Z",
            "Sum": 41014457.0,
            "Unit": "Count"
        },
        {
            "Timestamp": "2022-08-18T20:15:00Z",
            "Sum": 40048531.0,
            "Unit": "Count"
        },
```

次の結果は、評価期間中に書き込みトラフィックを受信しなかったテーブルを示しています。

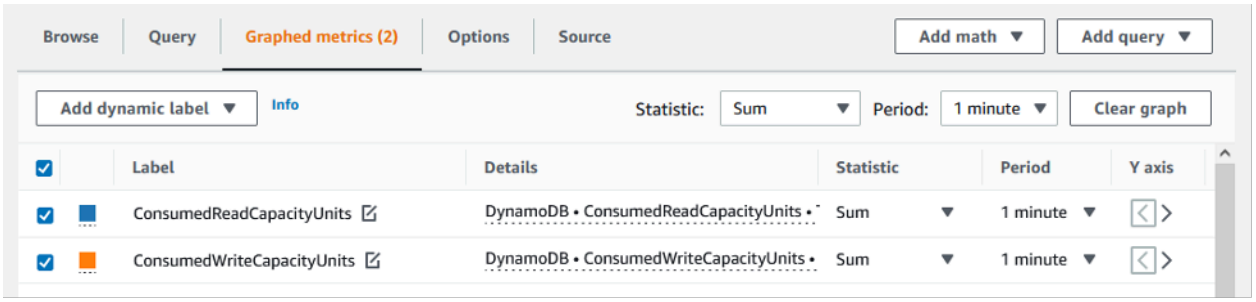
```
        {
            "Timestamp": "2022-07-31T20:15:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
        {
```

```
            "Timestamp": "2022-08-19T20:15:00Z",
            "Sum": 0.0,
            "Unit": "Count"
        },
```

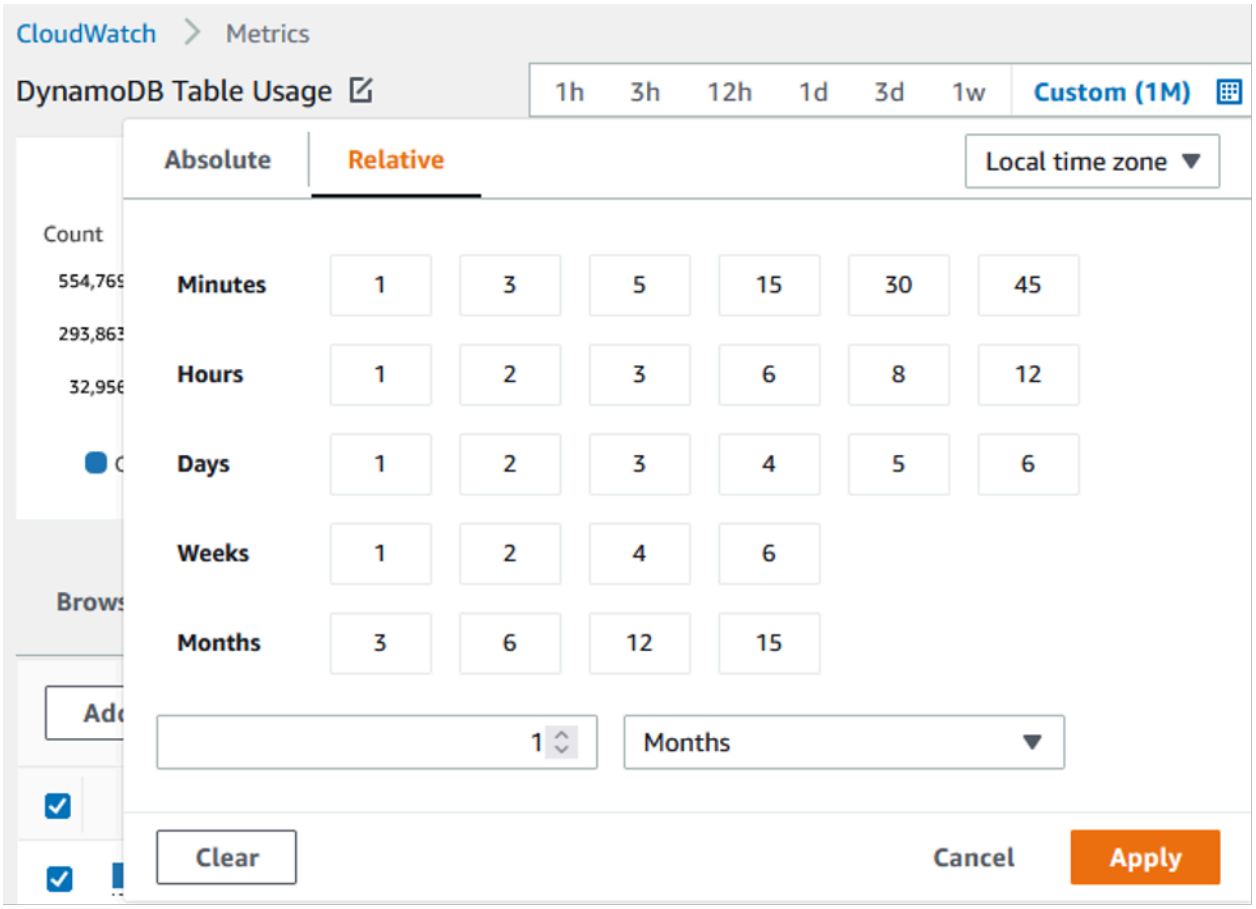
AWS マネジメントコンソール

次のステップでは、AWS マネジメントコンソール を使用してリソースの使用率を評価できます。

1. AWS コンソールにログインして、CloudWatch サービスページ (https://console.aws.amazon.com/cloudwatch/) を開きます。必要に応じて、コンソールの右上で適切な AWS リージョンを選択します。
2. 左のナビゲーションバーで、[Metrics] (メトリクス) セクションを探し、[All metrics] (すべてのメトリクス) を選択します。
3. これにより、2 つのパネルで構成されるダッシュボードが開きます。上部のパネルには、現在グラフ化されているメトリクスが表示されます。下部のパネルでは、グラフ化できるメトリクスを選択できます。下部のパネルで DynamoDB を選択します。
4. DynamoDB メトリクスの選択パネルで [Table Metrics] (テーブルメトリクス) カテゴリを選択し、現在のリージョンのテーブルのメトリクスを表示します。
5. メニューを下にスクロールしてテーブル名を確認し、テーブルの `ConsumedReadCapacityUnits` および `ConsumedWriteCapacityUnits` メトリクスを選択します。
6. [Graphed metrics (2)] (グラフ化したメトリクス (2)) タブを選択し、[Statistic] (統計) 列を [Sum] (合計) に設定します。

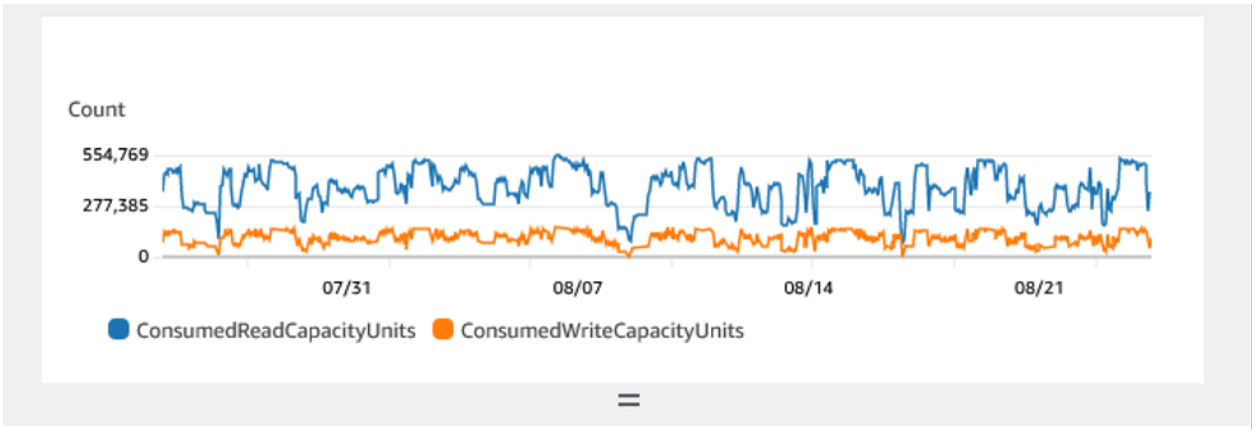


7. テーブルが未使用と誤って識別されないようにするには、長期間にわたってメトリクスを評価する必要があります。グラフパネルの上部で、テーブルを評価するための適切な時間枠 (1 か月など) を選択します。[Custom] (カスタム) を選択し、ドロップダウンで [1 Months] (1 か月間) を選択し、[Apply] (適用) を選択します。

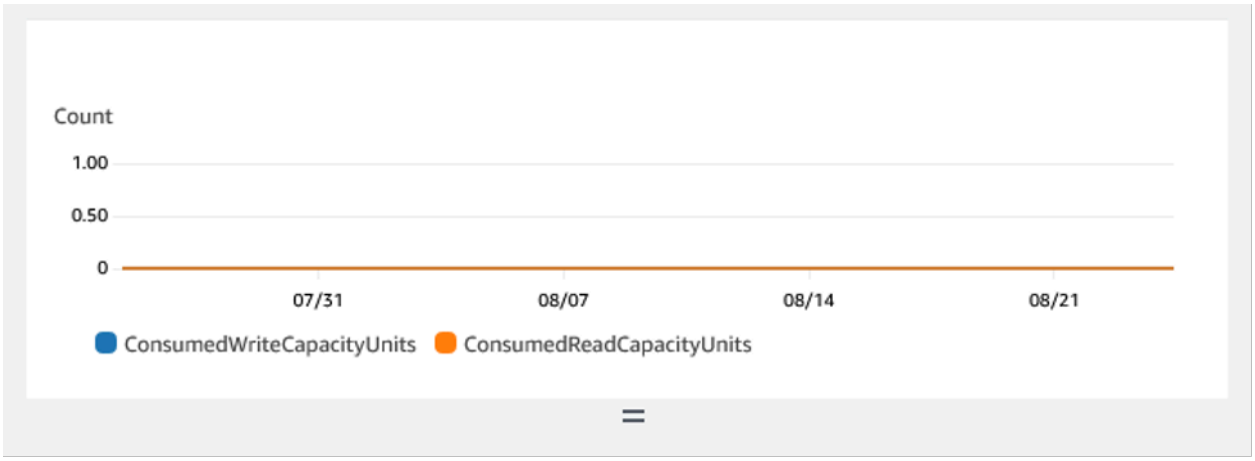


8. テーブルのグラフ化されたメトリクスを評価して、使用されているかどうかを判断します。0 を超えるメトリクスは、評価期間中にテーブルが使用されたことを示します。読み込みと書き込みの両方が 0 の平らなグラフは、テーブルが未使用であることを示します。

   以下の画像は、読み込みトラフィックがあるテーブルを示しています。



   以下の画像は、読み込みトラフィックがないテーブルを示しています。



未使用のテーブルリソースをクリーンアップする

未使用のテーブルリソースを特定したら、次の方法でその継続的なコストを削減できます。

> **Note**
>
> 未使用のテーブルを特定したものの、将来アクセスする必要が生じた場合に備えて使用できるようにしておきたい場合は、オンデマンドモードに切り替えることを検討してください。それ以外の場合、テーブル全体をバックアップして削除することを検討できます。

キャパシティモード

DynamoDB テーブル内のデータの読み込み、書き込み、保存には料金がかかります。

DynamoDB には 2 種類のキャパシティモードがあり、テーブルの読み書き処理について別個の請求オプション (オンデマンドとプロビジョンド) があります。読み取り/書き込みキャパシティモードは、読み取りおよび書き込みスループットの課金方法とキャパシティの管理方法を制御します。

オンデマンドモードのテーブルでは、アプリケーションで実行することが予測される読み込みおよび書き込みスループットを指定する必要はありません。DynamoDB では、読み込みリクエスト単位と書き込みリクエスト単位に関して、アプリケーションがテーブルに対して実行する読み込みと書き込みに対して料金が請求されます。テーブル/インデックスにアクティビティがない場合、スループットに対する支払いは発生しませんが、ストレージ料金は発生します。

テーブルクラス

DynamoDB には、コストの最適化に役立つように設計された 2 つのテーブルクラスも用意されています。デフォルトは DynamoDB Standard テーブルクラスで、大半のワークロードで推奨されていま

す。DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) テーブルクラスは、ストレージが主要なコストとなるテーブル向けに最適化されています。

テーブルまたはインデックスにアクティビティがない場合は、ストレージがコストの大きな割合を占める可能性が高く、テーブルクラスを変更することで大幅な節約が実現します。

テーブルの削除

未使用のテーブルを発見し、それを削除する場合は、まずデータのバックアップまたはエクスポートを行うことをお勧めします。

AWS Backup によるバックアップでは、コールドストレージの階層化を活用できるため、コストをさらに削減できます。AWS Backup を使用してバックアップを有効にする方法については「DynamoDB での AWS Backup の使用」のドキュメントを、ライフサイクルを使用してバックアップをコールドストレージに移動する方法については、「Managing backup plans」(バックアッププランの管理) のドキュメントを参照してください。

または、テーブルのデータを S3 にエクスポートすることもできます。その場合は、「Export to Amazon S3」(Amazon S3 へのエクスポート) のドキュメントを参照してください。データをエクスポートした後に、S3 Glacier Instant Retrieval、S3 Glacier Flexile Retrieval、または S3 Glacier Deep Archive を活用してさらにコストを削減する場合は、「Managing your storage lifecycle」(ストレージのライフサイクルの管理) を参照してください。

テーブルをバックアップしたら、AWS マネジメントコンソール または AWS Command Line Interface を使用してテーブルを削除できます。

未使用の GSI リソースを特定する

未使用のグローバルセカンダリを特定するステップは、未使用のテーブルを特定するステップと似ています。ベーステーブルに書き込まれた項目に GSI のパーティションキーとして使用される属性が含まれている場合、DynamoDB は項目を GSI にレプリケートするため、ベーステーブルが使用中であれば、未使用の GSI でも `ConsumedWriteCapacityUnits` が 0 を超える可能性があります。その結果、`ConsumedReadCapacityUnits` メトリクスのみを評価して、GSI が未使用かどうかを判断することになります。

AWS AWS CLI を使って GSI メトリクスを表示する場合は、次のコマンドを使用してテーブルの読み込みを評価できます。

```
aws cloudwatch get-metric-statistics --metric-name
ConsumedReadCapacityUnits --start-time <start-time> --end-time <end-
```

```
time> --period <period> --namespace AWS/DynamoDB --statistics Sum --
dimensions Name=TableName,Value=<table-name>
Name=GlobalSecondaryIndexName,Value=<index-name>
```

テーブルが未使用と誤って識別されないようにするには、長期間にわたってメトリクスを評価する必要があります。30 日間などの適切な開始時刻と終了時刻の範囲、および 86400 などの適切な期間を選択します。

返されるデータで、[Sum] (合計) が 0 を超える場合は、評価対象のテーブルがその期間に受信した読み込みトラフィックを示します。

次の結果は、評価期間中に読み込みトラフィックを受信した GSI を示しています。

```
        {
          "Timestamp": "2022-08-17T21:20:00Z",
          "Sum": 36319167.0,
          "Unit": "Count"
        },
        {
          "Timestamp": "2022-08-11T21:20:00Z",
          "Sum": 1869136.0,
          "Unit": "Count"
        },
```

次の結果は、評価期間中に最小限の読み込みトラフィックを受信した GSI を示しています。

```
        {
          "Timestamp": "2022-08-28T21:20:00Z",
          "Sum": 0.0,
          "Unit": "Count"
        },
        {
          "Timestamp": "2022-08-15T21:20:00Z",
          "Sum": 2.0,
          "Unit": "Count"
        },
```

次の結果は、評価期間中に読み込みトラフィックを受信しなかった GSI を示しています。

```
        {
          "Timestamp": "2022-08-17T21:20:00Z",
          "Sum": 0.0,
```

```
      "Unit": "Count"
    },
    {
      "Timestamp": "2022-08-11T21:20:00Z",
      "Sum": 0.0,
      "Unit": "Count"
    },
```

未使用の GSI リソースをクリーンアップする

未使用の GSI が見つかった場合は、それを削除することができます。GSI に存在するすべてのデータはベーステーブルにも存在するため、GSI を削除する前に追加のバックアップを行う必要はありません。今後、GSI が再び必要になった場合は、テーブルに追加し直すことができます。

使用頻度の低い GSI が見つかった場合は、それを削除したりコストを削減したりできるように、アプリケーションの設計変更を検討する必要があります。例えば、DynamoDB のスキャンは大量のシステムリソースを消費するため、控え目に使用する必要がありますが、サポートするアクセスパターンが極めて低頻度であれば、GSI よりもコスト効率が高くなる可能性があります。

さらに、低頻度のアクセスパターンを GSI でサポートする必要がある場合は、より限定された属性セットを射影することを検討してください。この場合、低頻度のアクセスパターンをサポートするために、ベーステーブルに対して後続のクエリが必要になる場合がありますが、ストレージと書き込みのコストを大幅に削減できる可能性があります。

未使用のグローバルテーブルをクリーンアップする

Amazon DynamoDB グローバルテーブルは、マルチリージョンにマルチアクティブデータベースをデプロイするための完全マネージド型のソリューションです。独自のレプリケーションソリューションを構築および管理する必要はありません。

グローバルテーブルは、ユーザーに近いデータに低レイテンシーでアクセスしたり、ディザスタリカバリのためのセカンダリリージョンにしたりするのに最適です。

データへの低レイテンシーアクセスを提供するためにリソースに対してグローバルテーブルオプションが有効になっていても、ディザスタリカバリ戦略の一部でない場合は、CloudWatch メトリクスを評価して、両方のレプリカが読み込みトラフィックをアクティブに処理していることを確認します。1 つのレプリカが読み込みトラフィックを処理しない場合、そのレプリカは未使用のリソースである可能性があります。

グローバルテーブルがディザスタリカバリ戦略の一部である場合、アクティブ/スタンバイパターンでは、1 つのレプリカが読み込みトラフィックを受信しないことが予想されます。

未使用のバックアップまたはポイントインタイムリカバリ (PITR) をクリーンアップする

DynamoDB には 2 種類のバックアップスタイルがあります。ポイントインタイムリカバリでは最長 35 日間までの継続的バックアップを利用できるため、意図しない書き込みや削除を防ぐことができます。一方、オンデマンドバックアップでは、長期的に保存が可能なスナップショットを作成できます。リカバリ期間は、1～35 日間の任意の値を設定できます。どちらのタイプのバックアップにも、関連するコストがあります。

「DynamoDB のバックアップと復元」および「DynamoDB のポイントインタイムバックアップ」のマニュアルを参照して、不要になった可能性のあるバックアップがテーブルで有効になっているかどうかを確認してください。

DynamoDB テーブルの使用パターンを評価する

このセクションでは、DynamoDB テーブルを効率的に使用しているかどうかを評価する方法の概要を説明します。DynamoDB には最適ではない特定の使用パターンがあり、パフォーマンスとコストの両方の観点から最適化の余地があります。

トピック

- 強力な整合性のある読み込みオペレーションの数を減らす
- 読み込み操作のトランザクション数を減らす
- スキャンの回数を減らす
- 属性名を短くする
- 有効期限 (TTL) の有効化
- グローバルテーブルをクロスリージョンバックアップに置き換える

強力な整合性のある読み込みオペレーションの数を減らす

DynamoDB では、リクエストごとに読み込み整合性を設定できます。デフォルトでは、読み込みリクエストは結果的に整合性が保たれます。結果整合性のある読み込みは、最大 4 KB のデータに対して 0.5 RCU で課金されます。

分散型ワークロードのほとんどの部分は柔軟性があり、最終的な一貫性を許容できます。ただし、強力な整合性のある読み込みを必要とするアクセスパターンもあり得ます。強力な整合性のある読み込みには、最大 4 KB のデータに対して 1 RCU の料金が課金されるため、読み込みコストは実質的に 2 倍になります。DynamoDB では、同じテーブルで両方の整合性モデルを柔軟に使用できます。

ワークロードとアプリケーションコードを評価して、強力な整合性のある読み込みが必要な場合にのみ使用されているかどうかを確認できます。

読み込み操作のトランザクション数を減らす

DynamoDB では、特定のアクションを「すべて」か「なし」でグループ化できます。つまり、DynamoDB で ACID トランザクションを実行することが可能です。ただし、リレーショナルデータベースの場合と同様に、すべてのアクションでこのアプローチに従う必要はありません。

最終的に一貫した方法で同じ量のデータを読み取るためにデフォルトでは 0.5 RCU が使用される一方、最大 4 KB のトランザクション読み込みオペレーションでは 2 RCU が使用されます。書き込み操作のコストは 2 倍になります。つまり、1 KB までのトランザクション書き込みは、2 WCU に相当します。

テーブルのすべてのオペレーションがトランザクションかどうかを判断するには、テーブルの CloudWatch メトリクスをトランザクション API に絞り込むことができます。テーブルの `SuccessfulRequestLatency` メトリクスで使用できるグラフがトランザクション API だけであれば、すべてのオペレーションがこのテーブルのトランザクションであることを確認できます。あるいは、全体的なキャパシティ使用率の傾向がトランザクション API の傾向と一致する場合は、トランザクション API が大部分を占めていると考えられるため、アプリケーション設計を見直すことを検討してください。

スキャンの回数を減らす

`Scan` オペレーションを多用するのは、一般的に DynamoDB データに対して分析クエリを実行する必要があるためです。大きなテーブルで頻繁に `Scan` オペレーションを実行するのは、非効率的でコストが高額になる可能性があります。

また、Export to S3 機能を使用し、特定の時点を選択してテーブルの状態を S3 にエクスポートする方法もあります。AWS には、テーブルの容量を消費することなく、データに対して分析クエリを実行できる Athena のようなサービスがあります。

`Scan` オペレーションの頻度は、`Scan` の `SuccessfulRequestLatency` メトリクスの `SampleCount` 統計を使用して決定できます。`Scan` オペレーションが非常に頻繁に行われる場合は、アクセスパターンとデータモデルの再評価が推奨されます。

属性名を短くする

DynamoDB の項目の合計サイズは、属性名と属性値の長さの合計です。属性名が長いと、ストレージコストだけでなく、RCU/WCU の消費量も増える可能性があります。属性名は長いものよりも短いものにすることをお勧めします。属性名を短くすると、項目サイズを次の 4KB/1KB の境界内に

制限でき、それを超えるとデータにアクセスするために追加の RCU/WCU を消費することになります。

有効期限 (TTL) の有効化

有効期限 (TTL) では、項目に設定した有効期限を過ぎた項目を特定してテーブルから削除できます。データが時間の経過とともに増加し、古いデータが重要でなくなった場合は、テーブルで TTL を有効にすると、データを削減してストレージコストを節約できます。

TTL のもう 1 つの便利な点は、期限切れの項目が DynamoDB ストリームで発生することです。そのため、データからデータを削除するだけでなく、ストリームからそれらの項目を消費して低コストのストレージ階層にアーカイブすることができます。さらに、TTL による項目の削除に追加コストはかかりません。容量を消費せず、クリーンアップアプリケーションを設計するオーバーヘッドもありません。

グローバルテーブルをクロスリージョンバックアップに置き換える

グローバルテーブルを使用すると、異なるリージョンに複数のアクティブなレプリカテーブルを管理できます。これらのテーブルはすべて書き込み操作を受け付け、相互にデータをレプリケートできます。レプリカの設定は簡単で、同期は自動的に管理されます。レプリカは、最新書き込み優先ストラテジーを使用して一貫した状態に収束します。

グローバルテーブルをフェイルオーバーまたはディザスタリカバリ (DR) 戦略の一環としてのみ使用している場合は、比較的緩やかな復旧ポイント目標と復旧時間目標の要件を満たすために、グローバルテーブルをクロスリージョンバックアップコピーに置き換えることができます。高速なローカルアクセスと 99.999% の高可用性が必要ない場合は、グローバルテーブルレプリカを維持することはフェイルオーバーに最適な方法ではないかもしれません。

別の方法として、AWS Backup を使用して DynamoDB バックアップを管理することを検討してください。定期的なバックアップをスケジュールしてリージョン間でコピーすることで、グローバルテーブルを使用するよりも費用対効果の高い方法で、DR 要件を満たすことができます。

DynamoDB Streams の使用状況を評価する

このセクションでは、DynamoDB Streams の使用量を評価する方法についての概要を説明します。DynamoDB には最適ではない特定の使用パターンがあり、パフォーマンスとコストの両方の観点から最適化の余地があります。

ストリーミングとイベントドリブンのユースケースには、次の 2 つのネイティブストリーミング統合が用意されています。

- Amazon DynamoDB Streams
- Amazon Kinesis Data Streams

このページでは、これらのオプションのコスト最適化戦略のみに焦点を当てます。2 つのオプションのどちらかを選択する方法については、「変更データキャプチャのストリーミングオプション」を参照してください。

トピック

- DynamoDB Streams のコストの最適化
- Kinesis Data Streams のコストの最適化
- どちらのタイプの Streams 使用にも適したコスト最適化戦略

DynamoDB Streams のコストの最適化

DynamoDB Streams の料金ページに記載されているように、テーブルのスループットキャパシティモードに関係なく、DynamoDB ではテーブルの DynamoDB Streams に対して行われた読み取りリクエストの数に応じて料金が発生します。DynamoDB Streams に対して行われる読み取りリクエストは、DynamoDB テーブルに対して行われる読み取りリクエストとは異なります。

Streams に関する各読み取りリクエストは `GetRecords` API コールの形式で行われ、最大 1,000 件のレコードまたは 1 MB 相当のレコードがレスポンスとして返されます。他のいずれの DynamoDB Streams API、またはアイドル状態の DynamoDB Streams では料金が発生しません。つまり、DynamoDB Streams に対して読み取りリクエストが行われない場合、テーブルで DynamoDB Streams を有効にしても料金は発生しません。

DynamoDB Streams コンシューマーアプリケーションをいくつか紹介します。

- AWS Lambda 関数
- Amazon Kinesis Data Streams ベースのアプリケーション
- AWS SDK を使用して構築されたお客様向けコンシューマーアプリケーション

DynamoDB Streams の AWS Lambda ベースのコンシューマーによる読み取りリクエストは無料ですが、他の種類のコンシューマーによる呼び出しには料金が発生します。毎月、Lambda 以外のユーザーによる読み取りリクエストのうち、最初の 2,500,000 件も無料です。これは、各 AWS リージョンの AWS アカウント内の DynamoDB Streams に対するすべての読み込みリクエストに適用されます。

DynamoDB Streams 使用量のモニタリング

請求コンソールの DynamoDB Streams の料金は、AWS アカウント内の AWS リージョンにあるすべての DynamoDB Streams でまとめられます。現在、DynamoDB Streams のタグ付けはサポートされていないため、コスト配分タグを使用して DynamoDB Streams の詳細なコストを特定することはできません。`GetRecords` の呼び出し量を DynamoDB Streams レベルで取得して、ストリームあたりの料金を計算することは可能です。呼び出し量は DynamoDB Streams の CloudWatch メトリクス `SuccessfulRequestLatency` とその `SampleCount` 統計によって表されます。このメトリクスには、継続的なレプリケーションを行うためのグローバルテーブルによる `GetRecords` コールや、AWS Lambda コンシューマによるコールも含まれますが、いずれも料金は発生しません。DynamoDB Streams によって公開されるその他の CloudWatch メトリクスについては、「DynamoDB のメトリクスとディメンション」を参照してください。

AWS Lambda をコンシューマーとして使用する

AWS Lambda 関数を DynamoDB Streams のコンシューマーとして使用できるかどうかを評価してください。これにより、DynamoDB Streams からの読み取りに関連するコストを削減できるためです。一方、DynamoDB Streams Kinesis Adapter または SDK ベースのコンシューマーアプリケーションでは、DynamoDB Streams に対して行った `GetRecords` 呼び出しの数に応じて料金が発生します。

Lambda 関数の呼び出しは Lambda の標準料金に基づいて課金されますが、DynamoDB Streams では料金は発生しません。Lambda は、レコードの DynamoDB Streams にあるシャードを 1 秒あたり 4 回の基本レートでポーリングします。レコードが利用可能になると、Lambda は関数を呼び出し、結果を待機します。処理が成功すると、Lambda は、レコードをさらに受け取るまでポーリングを再開します。

DynamoDB Streams Kinesis Adapter ベースのコンシューマーアプリケーションのチューニング

Lambda ベースではないコンシューマーによる読み取りリクエストは DynamoDB Streams に対して課金されるため、ほぼリアルタイムの要件と、コンシューマーアプリケーションが DynamoDB Streams をポーリングしなければならない回数とのバランスをとることが重要です。

DynamoDB Streams Kinesis Adapter ベースのアプリケーションを使用して DynamoDB Streams をポーリングする頻度は、設定された `idleTimeBetweenReadsInMillis` 値によって決まります。このパラメータは、同じシャードへの前回の `GetRecords` 呼び出しでレコードが返されなかった場合に、コンシューマーがシャードを処理する前に待機する時間をミリ秒単位で決定します。デフォルトでは、このパラメータの値は 1,000 ミリ秒です。ほぼリアルタイムの処理が必要ない場合は、このパラメータを増やしてコンシューマーアプリケーションが行う `GetRecords` 呼び出しを減らし、DynamoDB Streams 呼び出しを最適化できます。

Kinesis Data Streams のコストの最適化

Kinesis Data Streams が DynamoDB テーブルの変更データキャプチャイベントの送信先として設定されている場合、Kinesis Data Streams は個別のサイズ管理を必要とする場合があり、これは全体のコストに影響を与えます。DynamoDB は、変更データキャプチャユニット (CDU) 単位で課金されます。この場合、各ユニットは、DynamoDB サービスが送信先の Kinesis Data Streams に対して試行する 1 KB の DynamoDB 項目サイズで構成されます。

DynamoDB サービスによる料金に加えて、Kinesis Data Streams の標準の料金が発生します。料金ページに記載されているように、サービスの料金は、DynamoDB テーブルのキャパシティモードとは異なり、ユーザー定義のキャパシティモード (プロビジョニングまたはオンデマンド) によって異なります。おおまかに言うと、Kinesis Data Streams は、キャパシティモードと DynamoDB サービスによってストリームに取り込まれたデータに基づいて、時間単位で課金されます。Kinesis Data Streams のユーザー設定によっては、データ検索 (オンデマンドモードの場合)、データ保持の延長 (デフォルトの 24 時間を超える場合)、拡張ファンアウトコンシューマーの取得などの追加料金が発生する場合があります。

Kinesis Data Streams の使用量のモニタリング

DynamoDB 用 Kinesis Data Streams は、標準の Kinesis Data Streams CloudWatch メトリクスに加えて、DynamoDB からのメトリクスを発行します。Kinesis Data Streams のキャパシティ不足の場合は Kinesis によって、または Kinesis Data Streams の保管中のデータを暗号化するように設定された AWS KMS サービスなどの依存コンポーネントによって、DynamoDB サービスによる `Put` の試みがスロットリングされる可能性があります。

DynamoDB サービスが Kinesis Data Streams 用に発行する CloudWatch メトリクスの詳細については、「Kinesis Data Streams を使用した変更データキャプチャのモニタリング」を参照してください。スロットリングが原因のサービスの再試行による追加コストを回避するために、プロビジョニングモードの場合は Kinesis Data Streams のサイズを適切に設定することが重要です。

Kinesis Data Streams に適したキャパシティモードの選択

Kinesis Data Streams は、プロビジョニングモードとオンデマンドモードの 2 つのキャパシティモードでサポートされています。

- Kinesis Data Streams を含むワークロードのアプリケーショントラフィックが予測可能な場合や、トラフィックが安定しているか徐々に増加している場合、またはトラフィックが正確に予測できる場合は、Kinesis Data Streams プロビジョニングモードが適しており、コスト効率も高くなります。

- ワークロードが新しい場合や、アプリケーショントラフィックが予測できない場合、またはキャパシティを管理したくない場合は、Kinesis Data Streams のオンデマンドモードが適しており、コスト効率も高くなります。

コストを最適化するためのベストプラクティスは、Kinesis Data Streams に関連する DynamoDB テーブルが、Kinesis Data Streams のプロビジョニングモードを活用できる予測可能なトラフィックパターンを持っているかどうかを評価することです。ワークロードが新しい場合は、最初の数週間は Kinesis Data Streams にオンデマンドモードを使用し、CloudWatch メトリクスを確認してトラフィックパターンを理解してから、同じストリームをワークロードの性質に基づいてプロビジョニングモードに切り替えることができます。プロビジョニングモードの場合、Kinesis Data Streams のシャード管理に関する考慮事項に従うことでシャード数を推定できます。

DynamoDB 用 Kinesis Data Streams を使用したコンシューマーアプリケーションの評価

Kinesis Data Streams は DynamoDB Streams のように `GetRecords` の呼び出し回数で課金されないため、`GetRecords` のスロットリング制限内であれば、コンシューマーアプリケーションで何度でも呼び出しを行えます。Kinesis Data Streams のオンデマンドモードでは、データ読み取りは GB 単位で課金されます。プロビジョニングモードの Kinesis Data Streams では、データが 7 日未満の場合、読み取りは課金されません。Kinesis Data Streams コンシューマーとしての Lambda 関数の場合、Lambda は 1 秒あたり 1 回の基本レートで、レコードの Kinesis Data Streams にある各シャードをポーリングします。

どちらのタイプの Streams 使用にも適したコスト最適化戦略

AWS Lambda コンシューマー向けイベントフィルタリング

Lambda イベントフィルタリングを使用すると、フィルター条件に基づくイベントを Lambda 関数呼び出しバッチで使用できないように破棄できます。これにより、コンシューマー関数ロジック内の不要なストリームレコードを処理または破棄するための Lambda コストが最適化されます。イベントフィルタリングの設定とフィルタリング条件の記述の詳細については、「Lambda のイベントフィルタリング」を参照してください。

AWS Lambda コンシューマーの調整

`BatchSize` を大きくして呼び出しごとに処理する量を増やす、`BisectBatchOnFunctionError` を有効にして重複処理 (追加コストが発生する) を防ぐ、再試行回数が多すぎることのないように `MaximumRetryAttempts` を設定するなど、Lambda の設定パラメータを調整することで、コストをさらに最適化できます。デフォルトでは、コンシューマーの Lambda 呼び出しが失敗すると、ストリームからレコードの有効期限が切れるまで無限に再試行されます。DynamoDB Streams 場

合は 24 時間前後、Kinesis Data Streams では 24 時間から最長 1 年まで設定できます。上記の DynamoDB Streams コンシューマー向けの Lambda 設定オプションなど、その他の Lambda 設定オプションについては、「AWS Lambda デベロッパーガイド」を参照してください。

DynamoDB テーブルで適切なサイズのプロビジョニングを行うために、プロビジョンドキャパシティを評価する

このセクションでは、DynamoDB テーブルのプロビジョニングが適切なサイズであるかどうかを評価する方法の概要を説明します。ワークロードの変化に応じて、運用手順を適切に変更する必要があります。特に DynamoDB テーブルがプロビジョニングモードで設定されていて、テーブルを過剰にプロビジョニングしたり、プロビジョニング不足になったりするリスクがある場合はそうです。

以下に説明する手順では、本稼働アプリケーションをサポートしている DynamoDB テーブルから取得する必要がある統計情報が必要です。アプリケーションの動作を理解するには、アプリケーションからのデータの季節性を把握するのに十分な期間を定義する必要があります。たとえば、アプリケーションに週単位のパターンが見られる場合は、3 週間の期間を使用することで、アプリケーションのスループットニーズを分析するための十分な余裕が得られます。

何から始めればよいかわからない場合は、以下の計算に少なくとも 1 か月分のデータ使用量を使用してください。

容量を評価する際、DynamoDB テーブルでは、読み込みキャパシティユニット (RCU) と書き込みキャパシティユニット (WCU) を個別に設定できます。テーブルにグローバルセカンダリインデックス (GSI) が設定されている場合は、そのテーブルが消費するスループットを指定する必要があります。これもベーステーブルの RCU や WCU から独立しています。

> **Note**
>
> ローカルセカンダリインデックス (LSI) はベーステーブルの容量を消費します。

トピック

- DynamoDB テーブルの消費メトリクスを取得する方法
- プロビジョニング不足の DynamoDB テーブルを識別する方法
- 過剰にプロビジョニングされた DynamoDB テーブルを識別する方法

DynamoDB テーブルの消費メトリクスを取得する方法

テーブルと GSI の容量を評価するには、次の CloudWatch メトリクスをモニタリングし、適切なディメンションを選択してテーブルまたは GSI 情報を取得します。

読み込みキャパシティユニット	書き込みキャパシティユニット
ConsumedReadCapacityUnits	ConsumedWriteCapacityUnits
ProvisionedReadCapacityUnits	ProvisionedWriteCapacityUnits
ReadThrottleEvents	WriteThrottleEvents

この操作は、AWS CLI または AWS マネジメントコンソール で実行できます。

AWS CLI

テーブル消費メトリクスを取得する前に、まず CloudWatch API を使用して履歴データポイントを取得する必要があります。

まず、write-calc.json と read-calc.json の 2 つのファイルを作成します。これらのファイルは、テーブルまたは GSI の計算を表します。下の表に示すように、一部のフィールドを環境に合わせて更新する必要があります。

フィールド名	定義	例
<table-name>	分析するテーブルの名前	サンプルテーブル
<period>	ターゲット使用率の評価に使用する期間 (秒単位)	1 時間の場合は 3600 と指定します
<start-time>	ISO8601 形式で指定された評価間隔の開始	2022-02-21T23:00:00
<end-time>	ISO8601 形式で指定された評価間隔の終了	2022-02-22T06:00:00

書き込み計算ファイルに、指定された日付範囲の期間にプロビジョニングおよび消費された WCU の数が取得されます。また、分析に使用される使用率も生成されます。write-calc.json ファイルのすべての内容は以下のようになります。

```
{
  "MetricDataQueries": [
    {
      "Id": "provisionedWCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ProvisionedWriteCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
          ]
        },
        "Period": <period>,
        "Stat": "Average"
      },
      "Label": "Provisioned",
      "ReturnData": false
    },
    {
      "Id": "consumedWCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ConsumedWriteCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>""
            }
          ]
        },
        "Period": <period>,
        "Stat": "Sum"
      },
      "Label": "",
      "ReturnData": false
```

```
    },
    {
      "Id": "m1",
      "Expression": "consumedWCU/PERIOD(consumedWCU)",
      "Label": "Consumed WCUs",
      "ReturnData": false
    },
    {
      "Id": "utilizationPercentage",
      "Expression": "100*(m1/provisionedWCU)",
      "Label": "Utilization Percentage",
      "ReturnData": true
    }
  ],
  "StartTime": "<start-time>",
  "EndTime": "<ent-time>",
  "ScanBy": "TimestampDescending",
  "MaxDatapoints": 24
}
```

読み込み計算ファイルにも同様のファイルを使用します。このファイルには、指定された日付範囲にプロビジョニングおよび消費された RCU の数が取得されます。read-calc.json ファイルの内容は以下のようになります。

```
{
  "MetricDataQueries": [
    {
      "Id": "provisionedRCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ProvisionedReadCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
          ]
        },
        "Period": <period>,
        "Stat": "Average"
      },
      "Label": "Provisioned",
```

```
      "ReturnData": false
    },
    {
      "Id": "consumedRCU",
      "MetricStat": {
        "Metric": {
          "Namespace": "AWS/DynamoDB",
          "MetricName": "ConsumedReadCapacityUnits",
          "Dimensions": [
            {
              "Name": "TableName",
              "Value": "<table-name>"
            }
          ]
        },
        "Period": <period>,
        "Stat": "Sum"
      },
      "Label": "",
      "ReturnData": false
    },
    {
      "Id": "m1",
      "Expression": "consumedRCU/PERIOD(consumedRCU)",
      "Label": "Consumed RCUs",
      "ReturnData": false
    },
    {
      "Id": "utilizationPercentage",
      "Expression": "100*(m1/provisionedRCU)",
      "Label": "Utilization Percentage",
      "ReturnData": true
    }
  ],
  "StartTime": "<start-time>",
  "EndTime": "<end-time>",
  "ScanBy": "TimestampDescending",
  "MaxDatapoints": 24
}
```

ファイルを作成したら、使用率データの取得を開始できます。

1. 書き込み使用率データを取得するには、次のコマンドを実行します。

```
aws cloudwatch get-metric-data --cli-input-json file://write-calc.json
```

2. 読み込み使用率データを取得するには、次のコマンドを実行します。

```
aws cloudwatch get-metric-data --cli-input-json file://read-calc.json
```

両方のクエリの結果は、分析に使用される JSON 形式の一連のデータポイントになります。結果は、指定したデータポイントの数、期間、および特定のワークロードデータによって異なります。次のように表示されます。

```
{
    "MetricDataResults": [
        {
            "Id": "utilizationPercentage",
            "Label": "Utilization Percentage",
            "Timestamps": [
                "2022-02-22T05:00:00+00:00",
                "2022-02-22T04:00:00+00:00",
                "2022-02-22T03:00:00+00:00",
                "2022-02-22T02:00:00+00:00",
                "2022-02-22T01:00:00+00:00",
                "2022-02-22T00:00:00+00:00",
                "2022-02-21T23:00:00+00:00"
            ],
            "Values": [
                91.55364583333333,
                55.066631944444445,
                2.6114930555555556,
                24.9496875,
                40.94725694444445,
                25.61819444444444,
                0.0
            ],
            "StatusCode": "Complete"
        }
    ],
    "Messages": []
}
```

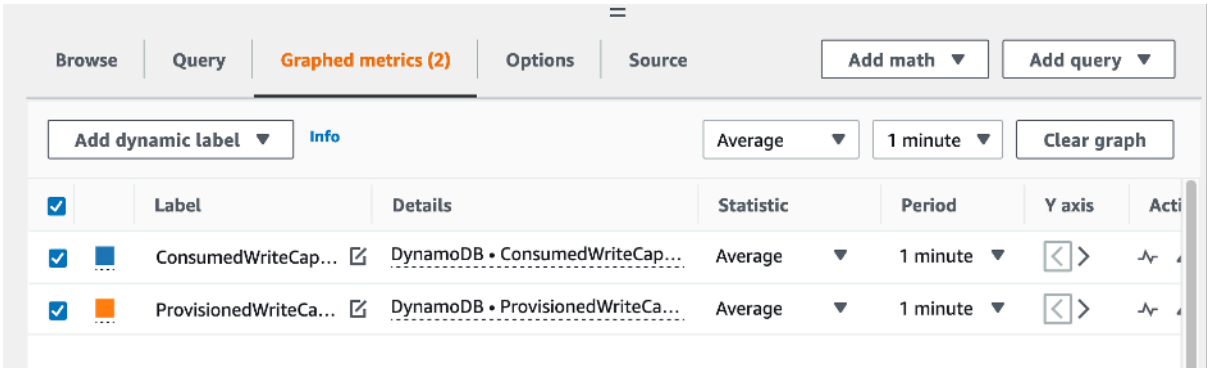
> **Note**
>
> 短い期間および長い時間範囲を指定する場合、スクリプトのデフォルトで 24 に設定されている `MaxDatapoints` を変更する必要があるかもしれません。これは、1 時間あたり 1 データポイント、1 日あたり 24 データポイントに相当します。

AWS マネジメントコンソール

1. AWS マネジメントコンソール にログインして、CloudWatch サービスページに移動します。必要に応じて、適切な AWS リージョン を選択します。
2. 左側のナビゲーションバーの [メトリクス] セクションで、[すべてのメトリクス] を選択します。
3. これにより、2 つのパネルで構成されるダッシュボードが開きます。上のパネルにはグラフィックが表示され、下のパネルにはグラフ化するメトリクスが表示されます。[DynamoDB] を選択します。
4. [テーブルメトリクス] を選択します。これにより、現在のリージョンのテーブルが表示されます。
5. 検索ボックスを使用してテーブル名を検索し、書き込みオペレーションメトリクス (`ConsumedWriteCapacityUnits` および `ProvisionedWriteCapacityUnits`) を選択します。

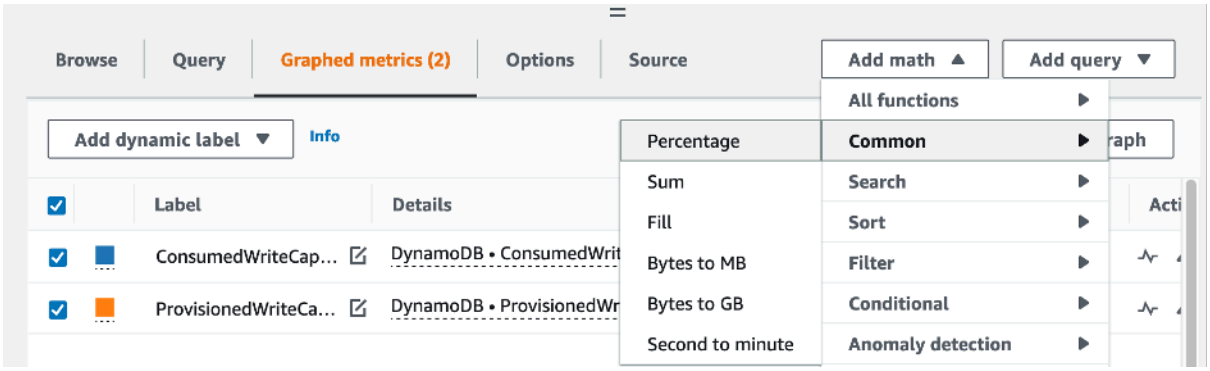
   > **Note**
   >
   > この例では書き込みオペレーションメトリクスについて説明していますが、これらの手順を使用して読み込みオペレーションメトリクスをグラフ化することもできます。

6. [グラフ化したメトリクス (2)] タブを選択して、式を変更します。デフォルトでは、CloudWatch はグラフの統計関数 [平均] を選択します。

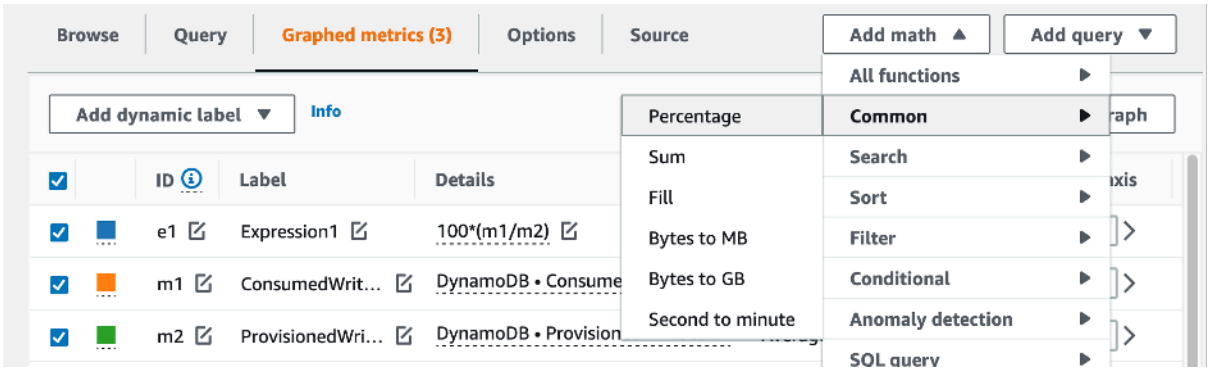


7. グラフ化された両方のメトリクスを選択した状態 (左側のチェックボックス) で、[Add math (算術の追加)] メニュー、[Common (共通)] の順に選択し、次に [Percentage (パーセンテージ)] 関数を選択します。この手順を 2 回繰り返します。

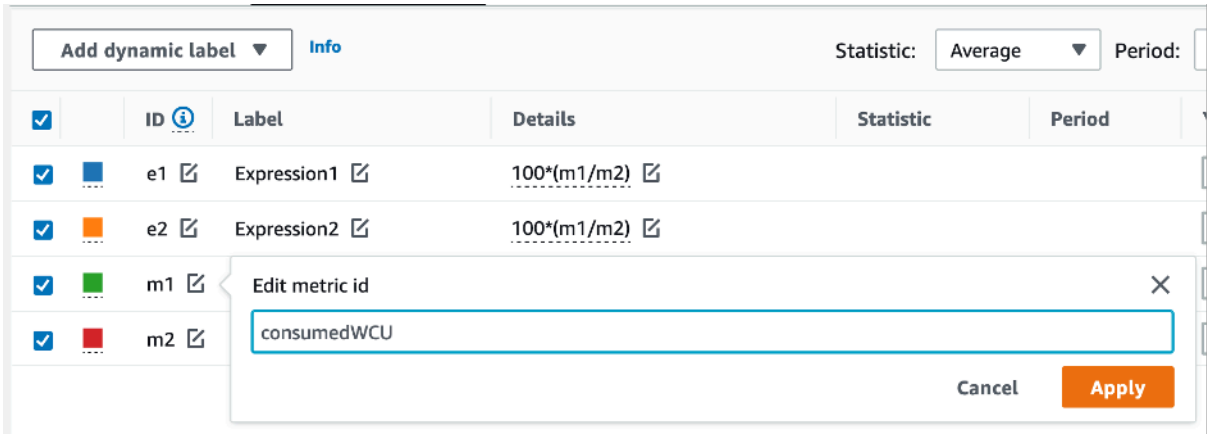
   [Percentage] (パーセンテージ) 関数を初めて選択する場合:



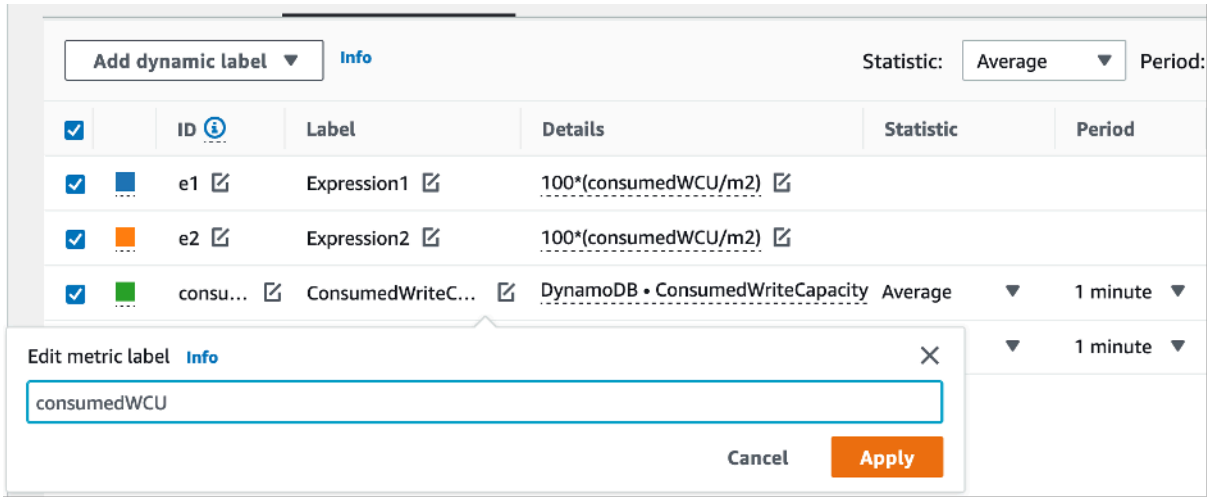
   [Percentage] (パーセンテージ) 関数を 2 回目に選択する場合:



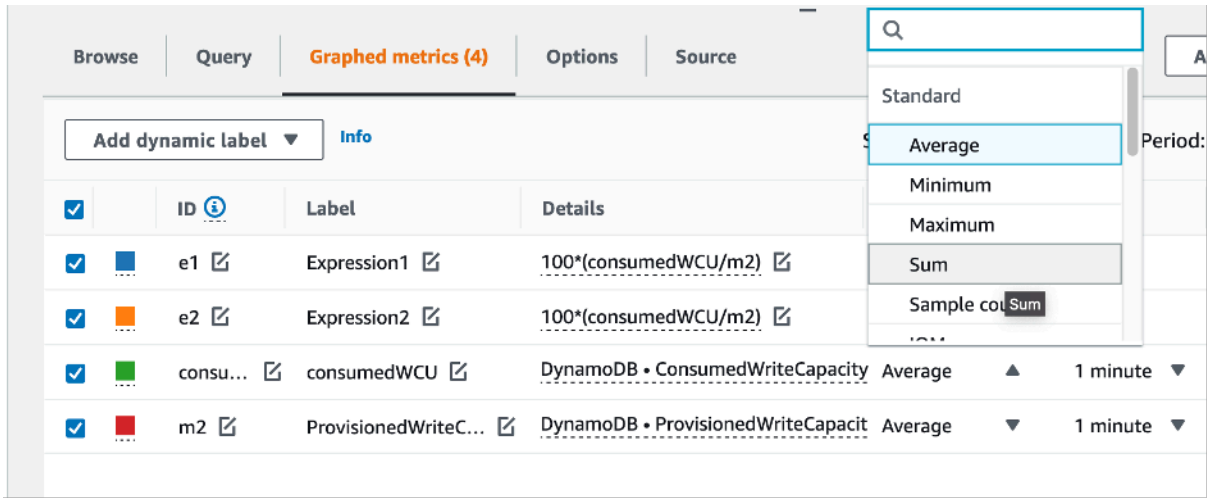
8. この時点で、下部のメニューに 4 つのメトリクスが表示されているはずです。`ConsumedWriteCapacityUnits` の計算に取り掛かりましょう。一貫性を保つために、AWS CLI セクションで使用した名前と一致させる必要があります。m1 ID を選択し、この値を consumedWCU に変更します。

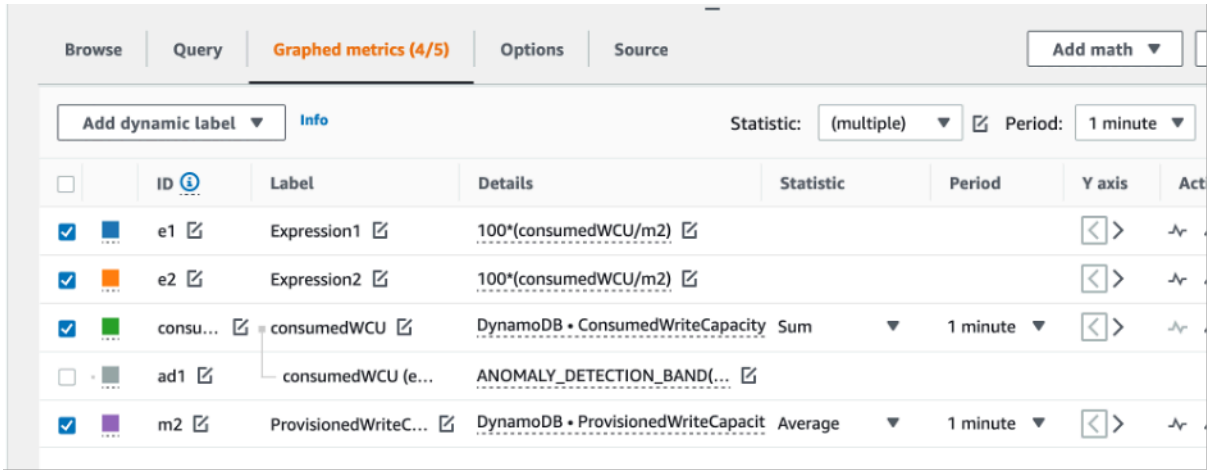


ConsumedWriteCapacityUnit ラベルの名前を **consumedWCU** に変更します。

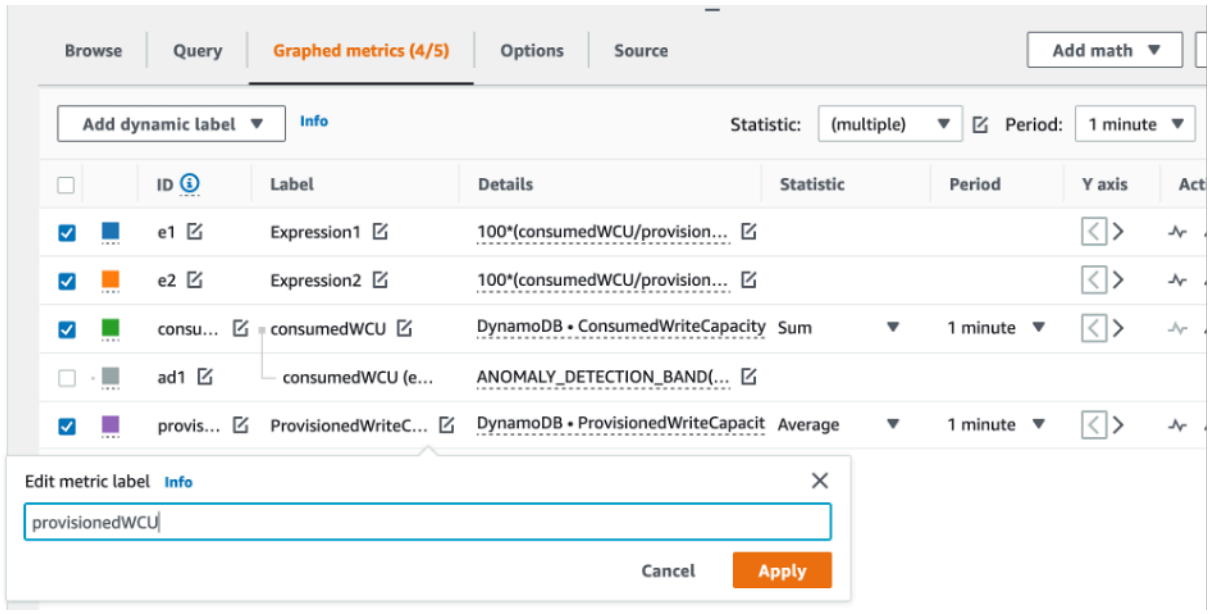


9. 統計を Average から Sum に変更します。このアクションにより、ANOMALY_DETECTION_BAND という名前の別のメトリクスが自動的に作成されます。この手順の対象範囲では、新しく生成された [ad1 metric] のチェックボックスをオフにして、これを無視します。



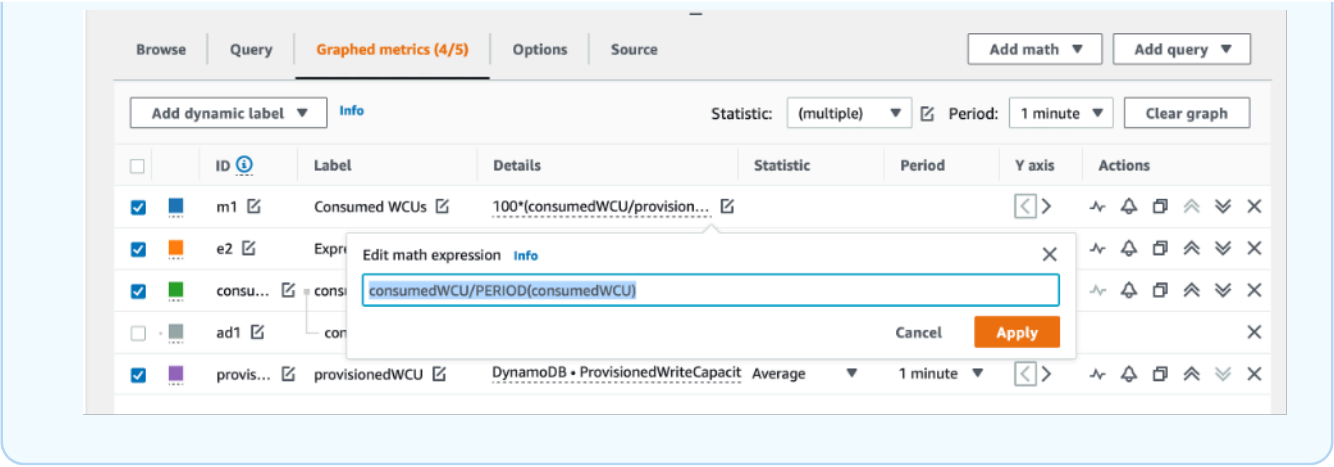


10. 手順 8 を繰り返して m2 ID の名前を ProvisionedWCU に変更します。統計は [Average (平均)] に設定したままにします。

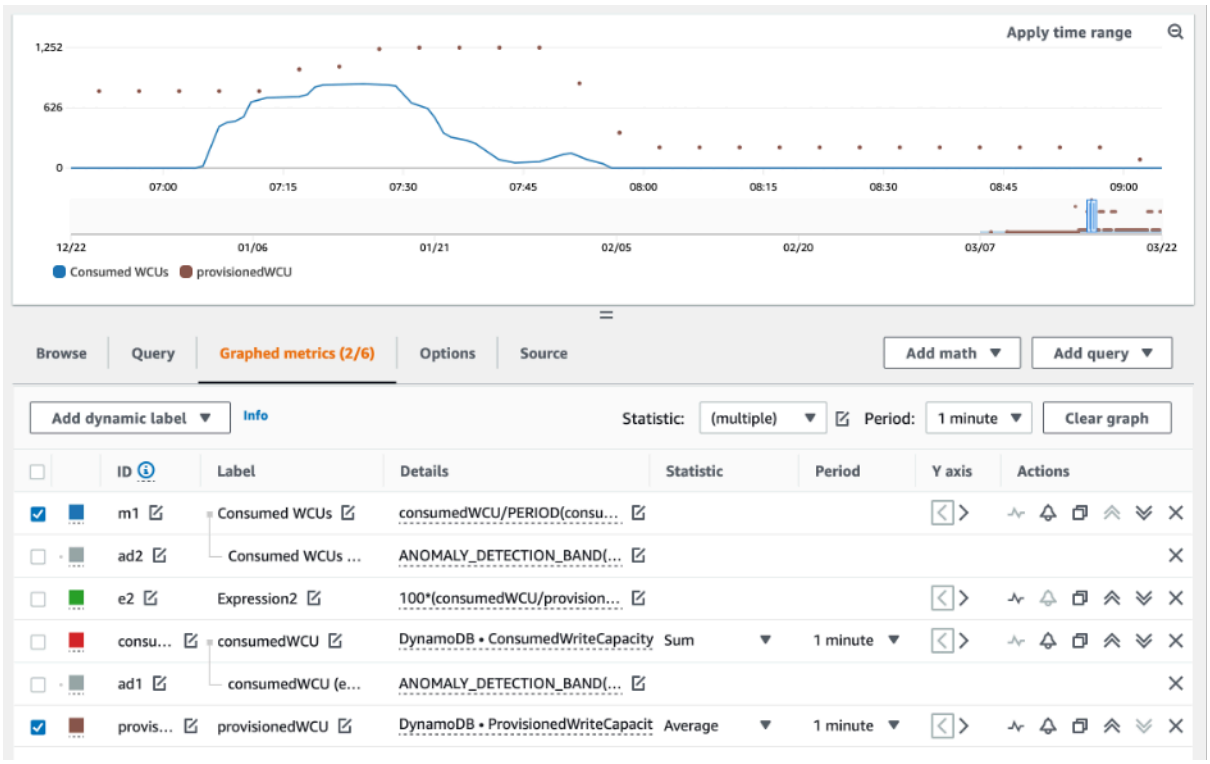


11. [Expression1] ラベルを選択し、値を [m1] に更新し、ラベルを [Consumed WCUs] に更新します。

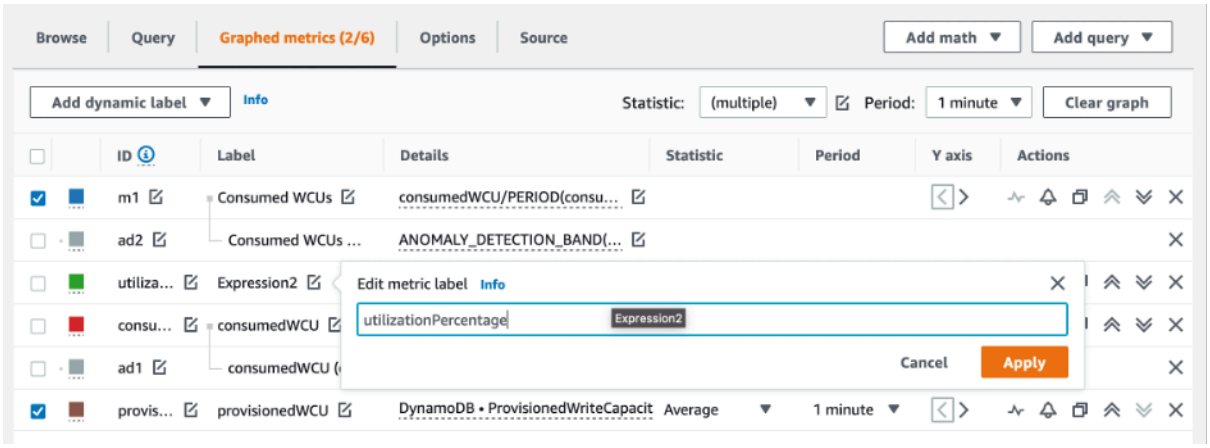
> **Note**
>
> データを正しく視覚化するには、必ず [m1] (左側のチェックボックス) と [ProvisionedWCU] のみを選択してください。[Details (詳細)] をクリックし、数式を [consumedWCU/PERIOD(consumedWCU)] に変更して、数式を更新します。このステップで別の [ANOMALY_DETECTION_BAND] メトリクスを生成することもありますが、この手順の範囲では無視できます。

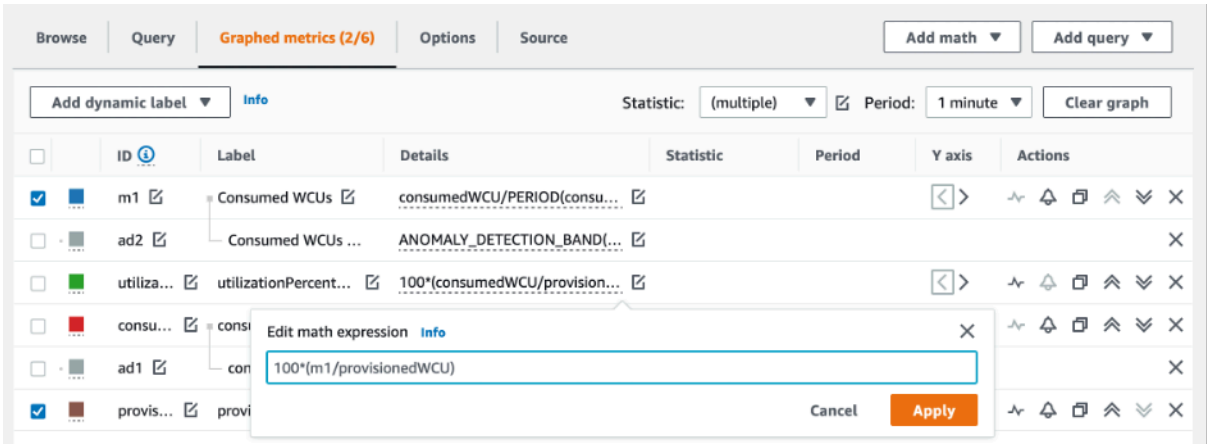


12. これで 2 つのグラフィックができたはずです。1 つはテーブル上にプロビジョニングされた WCU を示し、もう 1 つは消費された WCU を示しています。グラフィックの形状は下の図と異なる場合がありますが、参考にしてください。



13. Expression2 グラフィック (e2) を選択して、パーセンテージ数式を更新します。ラベルと ID の名前を utilizationPercentage に変更します。100*(m1/provisionedWCU) と一致するように数式の名前を変更します。

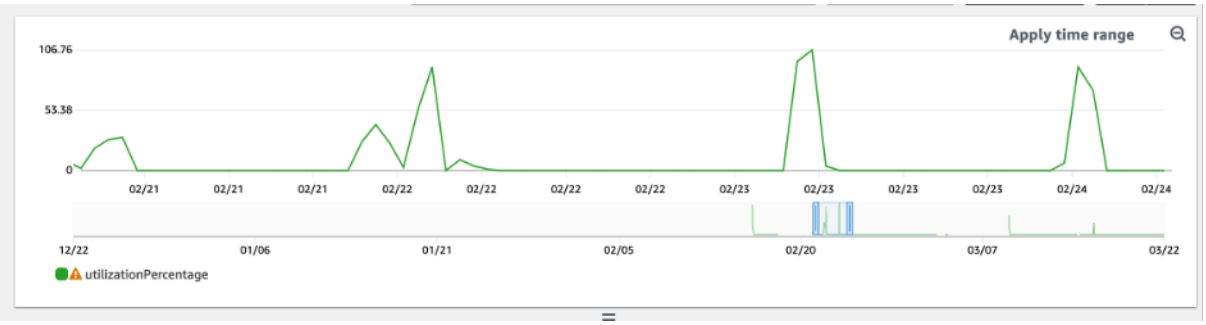




14. 使用率パターンを視覚化するには、utilizationPercentage 以外のすべてのメトリクスのチェックボックスをオフにします。デフォルトの間隔は 1 分に設定されていますが、必要に応じて自由に変更できます。



こちらは、1 時間の拡大表示と長時間の表示です。使用率が 100% を超える間隔がいくつかあることがわかりますが、この特定のワークロードでは、使用率がゼロの間隔が長くなっています。



この時点では、この例の写真とは異なる結果が得られるかもしれません。すべてはワークロードのデータ次第です。使用率が 100% を超える間隔では、スロットリングイベントが発生しやすくなります。DynamoDB にはバーストキャパシティがありますが、バーストキャパシティが完了すると、100% を超えるものはすべてスロットリングされます。

プロビジョニング不足の DynamoDB テーブルを識別する方法

ほとんどのワークロードでは、テーブルがプロビジョンドキャパシティの 80% 以上を絶えず消費している場合、そのテーブルはプロビジョニング不足と見なされます。

バーストキャパシティは DynamoDB の機能の 1 つで、お客様が当初のプロビジョニング量よりも多くの (表で定義されている 1 秒あたりのプロビジョニングスループットを超える) RCU/WCU を一時的に消費できるようにするものです。バーストキャパシティは、特別なイベントや使用量の急増によるトラフィックの急激な増加を吸収するために作成されました。このバーストキャパシティはいつまでも続くわけではありません。未使用の RCU と WCU がなくなると同時に、プロビジョニングされた容量よりも多くの容量を消費しようとすると、スロットリング状態になります。アプリケーショントラフィックの使用率が 80% に近づくと、スロットリングのリスクが大幅に高まります。

80% 使用率ルールは、データの季節性やトラフィックの増加によって異なります。次のシナリオを考えてみます。

- 過去 12 か月間、トラフィックの使用率が約 90% で安定していれば、テーブルのキャパシティは適切であると言えます
- アプリケーショントラフィックが 3 か月以内に毎月 8% の割合で増加した場合、100% に到達します
- アプリケーショントラフィックが 4 か月余りで 5% の割合で増加している場合でも、100% に到達します

上記のクエリの結果から、使用率の全体像がわかります。これらを参考にして、必要に応じてテーブルのキャパシティを増やす方法を選択するのに役立つ他のメトリクス (月間または毎週の増加率など) をさらに評価してください。運用チームと協力して、ワークロードとテーブルに適したパーセンテージを定義してください。

データを毎日または毎週分析すると、データが歪んでしまう特別なシナリオがあります。例えば、季節性のアプリケーションで、勤務時間中に使用量が急増する (ただし、勤務時間外はほぼゼロになる) 場合は、自動スケーリングをスケジュールして時間帯 (および曜日) を指定してプロビジョンドキャパシティを増やすと効果的です。繁忙期に対応できるように容量を増やすかわりに、季節性がそれほどない場合には、DynamoDB テーブルの自動スケーリング設定を利用することもできます。

> **Note**
>
> ベーステーブルに DynamoDB Auto Scaling 設定を作成するときは、そのテーブルに関連付けられている GSI にも別の設定を含めることを忘れないでください。

過剰にプロビジョニングされた DynamoDB テーブルを識別する方法

上記のスクリプトから取得したクエリ結果から、初期分析を実行するために必要なデータポイントが得られます。データセットの使用率が複数の間隔で 20% 未満の値を示す場合は、テーブルが過剰にプロビジョニングされている可能性があります。WCU と RCU の数を減らす必要があるかどうかをさらに明確にするには、その間隔で他の測定値を見直す必要があります。

テーブルに使用頻度の低い間隔が複数ある場合は、自動スケーリングをスケジュールするか、使用率に基づくテーブルのデフォルトの自動スケーリングポリシーを設定することで、自動スケーリングポリシーの使用から大きなメリットが得られます。

使用率が低いのにスロットリング率 (間隔内の Max(ThrottleEvents)/Min(ThrottleEvents)) が高いワークロードがある場合、この現象は、ある日 (または時間) にトラフィックが大幅に増加するが一般的にトラフィックは常に少ない、非常にスパイクの多いワークロードがある場合に発生する可能性があります。このようなシナリオでは、スケジュールされた自動スケーリングを使用すると有益な場合があります。

AWS Well-Architected フレームワークは、クラウドアーキテクトがさまざまなアプリケーションやワークロード向けに、安全で高性能、かつ回復力のある効率的なインフラストラクチャを構築するのに役立ちます。AWS Well-Architected は、運用上の優秀性、セキュリティ、信頼性、パフォーマンス効率、コスト最適化、持続可能性の 6 つの柱に基づいて構築されており、顧客とパートナーがアーキテクチャを評価し、スケーラブルな設計を実装するための一貫したアプローチを提供します。

AWS Well-Architected レンズによって、AWS Well-Architected が提供するガイダンスを特定の業界や技術分野に拡張できます。Amazon DynamoDB Well-Architected レンズでは、DynamoDB ワークロードに焦点を当てています。このレンズは、DynamoDB ワークロードを評価およびレビューするためのベストプラクティス、設計原則、質問を提供します。Amazon DynamoDB Well-Architected レンズによるレビューを完了すると、AWS Well-Architected の各柱に関連する推奨設計原則に関する教育とガイダンスを利用できるようになります。このガイダンスは、さまざまな業界、セグメント、規模、地域のお客様との作業経験に基づいています。

Well-Architected レンズによるレビューの直接的な結果として、DynamoDB ワークロードを最適化および改善するための実用的な推奨事項の概要が得られます。

Amazon DynamoDB Well-Architected レンズによるレビューの実施

DynamoDB Well-Architected レンズによるレビューは通常、AWSソリューションアーキテクトがお客様と一緒に実施しますが、お客様がセルフサービスとして実施することもできます。Amazon DynamoDB Well-Architected レンズの一環として Well-Architected の 6 本の柱をすべて確認することをお勧めしますが、まずは 1 本または複数の柱に優先的に焦点を当てでもかまいません。

Amazon DynamoDB の Well-Architected レンズによるレビューを実施するための追加情報と手順については、このビデオと DynamoDB Well-Architected Lens GitHub ページをご覧ください。

Amazon DynamoDB の Well-Architected レンズの柱

Amazon DynamoDB の Well-Architected レンズには 6 本の柱で構成されています。

パフォーマンス効率の柱

パフォーマンス効率の柱には、コンピューティングリソースを効率的に使用してシステム要件を満たし、需要の変化や技術の進歩に合わせてこの効率性を維持する能力が含まれます。

この柱の主な DynamoDB の設計原則は、データのモデル化、パーティションキーとソートキーの選択、およびアプリケーションアクセスパターンに基づくセカンダリインデックスの定義を中心に展開されています。その他の考慮事項には、ワークロードに最適なスループットモードの選択、AWS SDK によるチューニング、および必要な場合は、最適なキャッシュ戦略の使用などがあります。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズのパフォーマンス効率の柱に関するこのディープダイブビデオを参照してください。

コスト最適化の柱

コスト最適化の柱は、不必要なコストを回避することに焦点を当てています。

主なトピックは、資金の用途の把握と管理、最適なリソースタイプとその適切な数の選択、経時的な支出の分析、アプリケーション固有のアクセスパターンに応じてコストを最適化するデータモデルの設計、浪費することなくビジネスニーズを満たすためのスケーリングです。

DynamoDB の主なコスト最適化設計原則は、テーブルに最適なキャパシティモードとテーブルクラスの選択、オンデマンドキャパシティモードまたは自動スケーリングを使用するプロビジョンドキャパシティモードのいずれかを使用したキャパシティの過剰プロビジョニングの回避を中心に展開されています。その他の考慮事項には、効率的なデータモデリングとクエリによる消費される容量の削減、消費容量の一部の割引価格での予約、項目サイズの最小化、未使用のリソースの特定と削除、TTL を使用した古くなったデータの無料の自動削除などがあります。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズのコスト最適化の柱に関するこのディープダイブビデオを参照してください。

DynamoDB のコスト最適化のベストプラクティスの詳細については、「DynamoDB テーブルのコストの最適化」を参照してください。

運用上の優秀性の柱

運用上の優秀性の柱では、ビジネス価値をもたらし、プロセスと手順の継続的な向上を実現するために、システムを実行およびモニタリングすることに焦点を当てています。主なトピックは、変更の自動化、イベントへの対応、日常業務を管理するための標準の定義です。

DynamoDB の運用上の優秀性の主な設計原則は、Amazon CloudWatch および AWS Config を通じて DynamoDB メトリックスを監視すること、事前に定義されたしきい値に違反した場合や準拠していないルールが検出された場合に自動的に警告して修正することです。その他の考慮事項としては、インフラストラクチャをコードとして使用して DynamoDB リソースを定義し、タグを活用して DynamoDB リソースの整理、識別、およびコスト計算を改善することがあります。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズの運用上の優秀性の柱に関するこのディープダイブビデオを参照してください。

信頼性の柱

信頼性の柱では、期待されるときにワークロードが意図された機能を正しく一貫して実行することを重視しています。回復力のあるワークロードは、障害から迅速に復旧し、ビジネスや顧客の需要を満たします。主なトピックは、分散システムの設計、復旧計画、および変更への対応方法です。

DynamoDB の重要な信頼性設計の原則は、RPO と RTO の要件に基づいてバックアップ戦略と保持期間を選択し、RTO の低いマルチリージョンワークロードまたはクロスリージョンディザスタリカバリシナリオに DynamoDB グローバルテーブルを使用することです。また、AWS SDK でこれらの機能を設定して、アプリケーションにエクスポネンシャルバックオフを含む再試行ロジックを実装

し、Amazon CloudWatch を介して DynamoDB メトリクスをモニタリングして、事前定義されたしきい値を超えたときに自動的にアラートの実行と修復を行う必要があります。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズの信頼性の柱に関するこのディープダイブビデオを参照してください。

セキュリティの柱

セキュリティの柱は、情報とシステムの保護に焦点を当てています。主なトピックは、データの機密性と完全性、権限管理による誰が何を実行できるのかの特定と管理、システムの保護、セキュリティイベントを検出するための制御の確立です。

DynamoDB のセキュリティの主な設計原則は、転送中のデータを HTTPS で暗号化すること、保管中のデータを暗号化するためのキーの種類を選択すること、DynamoDB リソースに対して認証および認可を行い、きめ細かなアクセスを提供するための IAM ロールとポリシーを定義することです。その他の考慮事項は、AWS CloudTrail を通じた DynamoDB のコントロールプレーンオペレーションとデータプレーンオペレーションです。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズのセキュリティの柱に関するこのディープダイブビデオを参照してください。

DynamoDB のセキュリティの詳細については、「Amazon DynamoDB のセキュリティとコンプライアンス」を参照してください。

持続可能性の柱

持続可能性の柱は、クラウドワークロードの実行による環境に対する影響を最小限に抑えることに重点を置いています。主なトピックは、持続可能性に関する責任共有モデル、影響の把握、必要なリソースを最小限に抑えてダウンストリームへの影響を軽減するための使用率の最大化です。

DynamoDB の主な持続可能性の設計原則には、未使用の DynamoDB リソースの特定と削除、およびオンデマンドキャパシティモードまたは自動スケーリング付きプロビジョンドキャパシティモードの使用によるオーバープロビジョニングの回避が含まれます。また、効率的なクエリによって容量の消費量を削減したり、データを圧縮し TTL を使用して古いデータを削除することでストレージフットプリントを削減したりすることもできます。これらの設計原則の詳細については、DynamoDB の Well-Architected レンズの持続可能性の柱に関するこのディープダイブビデオを参照してください。

DynamoDB でパーティションキーを効率的に設計し、使用するためのベストプラクティス。

Amazon DynamoDB テーブルの各項目を一意に識別する主キーは、シンプル (パーティションキーのみ)、または複合 (パーティションキーとソートキーの組み合わせ) で構成できます。

テーブルとそのセカンダリインデックスのすべてのパーティションキーにわたって、アクティビティが均一であるアプリケーションを設計する必要があります。アプリケーションが必要とするアクセスパターンと、各テーブルとセカンダリインデックスが必要とする読み込みユニットと書き込みユニットを決定することができます。

> **Note**
>
> アダプティブキャパシティは、オンデマンドモードとプロビジョンドキャパシティに適用されます。

DynamoDB テーブル内のすべてのパーティションは、1 秒あたり 3,000 の読み取りユニットと 1 秒あたり 1,000 の書き込みユニットを最大キャパシティとして提供するように設計されています。1 つの読み取りユニットは、最大サイズが 4 KB の項目に対する、1 秒あたり 1 回の強力な整合性のある読み込み、または 1 秒あたり 2 回の結果整合性のある読み込みを表します。1 つの書き込みユニットは、最大サイズが 1 KB の項目に対する、1 秒あたり 1 回の書き込みオペレーションを表します。

テーブルのパーティションスループット制限を評価するときは、項目のサイズを考慮する必要があります。例えば、テーブルの項目サイズが 20 KB である場合、1 回の整合性のある読み取りオペレーションでは 5 つの読み取りユニットを消費します。つまり、パーティションの制限に達する前に、その 1 つの項目に対して 1 秒あたり 600 回の整合性のある読み取りオペレーションを同時に実行できます。テーブル内のすべてのパーティションの合計スループットは、プロビジョンドモードでプロビジョニングされたスループット、またはオンデマンドモードのテーブルレベルのスループット制限によって制約される場合があります。詳細については、「Service Quotas」を参照してください。

トピック

- パーティションキーを設計してワークロードを DynamoDB で分散する
- 書き込みシャーディングを使用して DynamoDB テーブルでワークロードを均等に分散させる
- DynamoDB でデータアップロード中の書き込みアクティビティを効率的に分散させる

パーティションキーを設計してワークロードを DynamoDB で分散する

テーブルの主キーのパーティションキー部分は、テーブルのデータが格納されている論理パーティションを決定します。これにより、基本的な物理パーティションに影響を及ぼします。I/O リクエストを効率的に分散しないパーティションキー設計では、「ホット」パーティションが作成される場合があります。これにより、スロットリングが発生することや、プロビジョニングされた I/O 容量が効率的に使用されないことがあります。

テーブルのプロビジョニングされたスループットの最適な使用は、個々の項目のワークロードパターンだけでなく、パーティションキー設計にも左右されます。この操作によって、スループットレベルを達成するためにすべてのパーティションキーバリューにアクセスしたり、アクセスされるパーティションキーバリューの割合を高くしたりする必要があるわけではありません。これは、ワークロードがアクセスするパーティションキーバリューが大きくなるほど、リクエストが分割された空間に分散されることを意味します。一般的に、パーティションキー値の総数に対するアクセスされたパーティションキー値の比率が高くなるほど、プロビジョンドスループットの使用がより効率的になります。

一般的なパーティションキースキーマのプロビジョニングされたスループット効率の比較を次に示します。

パーティションキーバリュー	均一性
ユーザー ID (アプリケーションに多くのユーザーが存在する場合)	良好
ステータスコード (可能性のあるステータスコードが少しだけある場合)	不良
項目の作成日。直近の期間 (日、時、分など) に切り上げられます。	不良
デバイス ID (各デバイスが比較的類似した間隔でデータにアクセスする場合)。	良好
デバイス ID (追跡中のデバイスが多数あり、そのうちの 1 つのデバイスが他のすべてのデバイスよりも頻繁に使用される場合)	不良

単一のテーブル内にあるパーティションキーバリューの数が少ない場合は、より明確に区別できるパーティションキーバリュー全体を対象として、書き込みオペレーションを分散することを検討してください。つまり、「ホット」パーティションキーバリュー (何度もリクエストされるパーティションキーバリュー) が原因で全体のパフォーマンス速度が低下する場合があるので、このようなパーティションキーバリューを回避するようにプライマリキー要素を構築してください。

例えば、複合プライマリキーを持つテーブルがあるとします。パーティションキーは項目の作成日を表します (直近の日付に切り上げられます)。ソートキーは項目の識別子を表します。特定の日付

(2014-07-09 など) で、すべての新しい項目が 1 つのパーティションキーバリュー (およびその物理パーティション) として書き込まれます。

テーブル全体が単一のパーティションに収まり (時間の経過に伴うデータの増加を考慮)、アプリケーションで必要となる読み込みスループットと書き込みスループットが単一のパーティションにおける読み込みと書き込みの容量を超過しない場合は、パーティション分割をしても、アプリケーションが予想外のスロットリングを受けることはありません。

NoSQL Workbench for DynamoDB を使用してパーティションキー設計を視覚化する方法については、「NoSQL Workbench を使用したデータモデルの構築」を参照してください。

書き込みシャーディングを使用して DynamoDB テーブルでワークロードを均等に分散させる

Amazon DynamoDB のパーティションキースペースで効率的な書き込みの分散を行う方法として、スペースの拡張があります。これにはさまざまな方法があります。パーティションキーバリューに乱数を追加して、パーティション間で項目を分散させることができます。または、クエリ対象に基づいて計算された数値を使用することができます。

ランダムなサフィックスを使用したシャーディング

パーティションキースペース全体に均等にロードを分散するための方法として、パーティションキーバリューの最後に乱数を追加する方法があります。次に、より大きなスペース全体の書き込みをランダム化します。

例えば、今日の日付を表すパーティションキーの場合は、1 と 200 の間の乱数を選択し、それをサフィックスとして日付に連結することができます。これにより、2014-07-09.1 や 2014-07-09.2 などのパーティションキーバリューが生成されます。この場合、最大のパーティションキーバリューは 2014-07-09.200 になります。パーティションキーをランダム化しているため、毎日のテーブルへの書き込みは複数のパーティション間で均等に分散されます。これにより、並列性が強化され、全体的なスループットが向上します。

ただし、特定の日のすべての項目を読み込むには、すべてのサフィックスの項目をクエリしてから結果をマージする必要があります。例えば、最初にパーティションキーバリュー Query に対して 2014-07-09.1 リクエストを発行します。次に Query に対して別の 2014-07-09.2 を発行します。この処理を 2014-07-09.200 まで繰り返します。最終的には、アプリケーションですべての Query リクエストの結果をマージする必要があります。

計算されたサフィックスを使用したシャーディング

ランダム化の方法は、書き込みスループットを大幅に向上させることができます。ただし、項目を書き込むときに使用されたサフィックスの値がわからないため、特定の項目を読み込むのは困難です。別の方法を使用して個々の項目の読み込みを容易にすることができます。パーティション間で項目を分散させるには、乱数を使用せずに、クエリする項目に基づいて計算できる数値を使用します。

前の例では、パーティションキーで今日の日付が使用されています。各項目にはアクセス可能な `OrderId` 属性があり、日付だけでなく、順序 ID を使用して、項目を最も頻繁に見つける必要があるとします。アプリケーションから項目をテーブルに書き込む前に、この順序 ID に基づいてハッシュサフィックスを計算し、その値をパーティションキーの日付に追加します。この計算では、1 から 200 までの数値が生成され、ランダムな方法と同様に、完全に均一に分散されます。

この計算はシンプルな計算で十分です (順序 ID の文字の UTF-8 コードポイント値を乗算して、それを 200 で割った余りに + 1 するなど)。パーティションキーバリューは、日付に計算結果を連結した値になります。

この方法を使用すると、書き込みがパーティションキーバリュー全体に均一に分散されます。したがって、物理パーティション全体にも均一に分散されます。これで、特定の `GetItem` 値のパーティションキーバリューを計算できるため、`OrderId` オペレーションを実行して、特定の項目や日付を簡単に読み込むことができます。

特定の日のすべての項目を読み込むには、`2014-07-09.N` キー (ここで `N` は 1 〜 200) を`Query`する必要があり、アプリケーションはすべての結果をマージする必要があります。この方法には、すべてのワークロードを利用する単一の「ホット」パーティションキーバリューを使用しないというメリットがあります。

> **Note**
>
> 大容量の時系列データを処理するために特別に設計されている効率的な方法については、「時系列データ」を参照してください。

DynamoDB でデータアップロード中の書き込みアクティビティを効率的に分散させる

通常、他のデータソースからデータをロードする場合、Amazon DynamoDB がテーブルデータを複数のサーバーでパーティション化します。割り当てられたすべてのサーバーに同時にデータをアップロードするとパフォーマンスが向上します。

例えば、パーティションキーとして `UserID` とソートキーとして `MessageID` を含む複合プライマリキーを使用する DynamoDB テーブルにユーザーメッセージをアップロードするとします。

データをアップロードする場合、1 つのアプローチとして、ユーザーごとにすべてのメッセージ項目を別々にアップロードできます。

UserId	MessageID
U1	1
U1	2
U1	...
U1	... 最大 100
U2	1
U2	2
U2	...
U2	... 最大 200

この場合の問題は、パーティションキーバリュー全体で DynamoDB への書き込みリクエストを分散していないことです。一度に 1 つのパーティションキーバリューを取得し、そのすべての項目をアップロードした後に、次のパーティションキーバリューに移動して同じ処理が行われます。

その裏では、DynamoDB がテーブル内のデータを複数のサーバーでパーティション化しています。テーブルにプロビジョニングされたすべてのスループットキャパシティを完全に使用するには、ワークロードをパーティションキーバリューに分散する必要があります。

同じパーティションキーバリューを持つ項目に対して不均等な量のアップロード作業を指示すると、DynamoDB がテーブル用にプロビジョニングしたすべてのリソースを完全に使用することはできません。

アップロード作業を分散するには、ソートキーを使用して各パーティションキーバリューから 1 つの項目をロードし、次に各パーティションキーバリューから別の項目をロードします。

UserId	MessageID
U1	1
U2	1
U3	1
...	...
U1	2
U2	2
U3	2
...	...

このシーケンスの各アップロードでは、それぞれ異なるパーティションキーバリューが使用されるので、多くの DynamoDB サーバーが同時にビジー状態になり、スループットパフォーマンスが向上します。

ソートキーを使用してデータを DynamoDB で整理するためのベストプラクティス

Amazon DynamoDB テーブルでは、テーブルの各アイテムを一意に識別するプライマリキーは、パーティションキーとソートキーで構成することができます。

設計が優れたソートキーには、2 つの主な利点があります。

- 関連情報を 1 つの場所にまとめて、効率的にクエリを実行することができます。ソートキーを慎重に設計することで、`begins_with`、`between`、`>`、`<` などの演算子による範囲のクエリを使用して、一般的に必要な関連項目のグループを検索することができます。
- 複合ソートキーを作成すれば、データの階層的 (1 対多) な関係を定義して、任意の階層レベルでクエリを実行することができます。

  例えば、地理的場所を示すテーブルでは、次のようにソートキーを構成できます。

```
[country]#[region]#[state]#[county]#[city]#[neighborhood]
```

これにより、これらの集計レベルのいずれかの場所のリスト (country から neighborhood、またその間にあるすべてのもの) に対して、範囲のクエリを効率的に行うことができます。

バージョンコントロールにソートキーを使用する

多くのアプリケーションでは、監査またはコンプライアンス目的で項目レベルのリビジョン履歴を保持し、最新バージョンを簡単に取得できる必要があります。ソートキープレフィックスを使用してこれを実現する効果的な設計パターンがあります。

- 新しい項目ごとに、コピーを 2 つ作成します。一方は、ソートキーの冒頭にバージョン番号 0 のプレフィックス (v0_ など) を、もう一方には、バージョン番号 1 のプレフィックス (v1_ など) を付加します。
- 項目が更新されるたびに、更新されたバージョンのソートキーの次の上位バージョンプレフィックスを使用して、更新後の内容をバージョンのプレフィックス 0 の項目にコピーします。つまり、任意の項目の最新バージョンは、0 のプレフィックスを使用して簡単に見つけることができます。

例えば、部品メーカーは、以下に示すようなスキーマを使用する場合があります。

Primary Key		Data-Item Attributes...				
Partition Key *Equipment_ID*	Sort Key *(varies)*	Attribute 1	Attribute 2	Attribute 3	Attribute 4	...
Equipment_1	Details	Name: Biphasic Cardiometer *(equipment name)*	Factory_ID: S14_Tukwila *(factory where manufactured)*	Line_ID R_7 *(assembly-line ID)*		
	v0_Audit	Auditor: Padma *(name of the auditor)*	Latest: 3 *(most recent audit version)*	Time: 2018-04-15T11:00 *(audit date and time)*	Result Passed *(audit result)*	...etc.
	v1_Audit	Auditor: Rick *(name of the auditor)*	Time: 2018-03-14T11:00 *(audit date and time)*	Result Open *(audit result)*	Report: 0943922EKG14 *(detailed problem report in S3)*	...etc.
	v2_Audit	Auditor: George *(name of the auditor)*	Time: 2018-03-18T11:00 *(audit date and time)*	Result Open *(audit result)*	Report: 0943923EKG15 *(detailed problem report in S3)*	...etc.
	v3_Audit	Auditor: Padma *(name of the auditor)*	Time: 2018-04-15T11:00 *(audit date and time)*	Result Passed *(audit result)*	Report: x792 *(pass confirmation report)*	...etc.

Equipment_1 項目は、さまざまな監査人によって、一連の監査が行われます。新しい各監査の結果は、テーブルの新しい項目に取り込まれます。バージョン番号 1 から始まり、その後のリビジョンごとに番号が増えます。

新しいリビジョンが追加されるたびに、アプリケーション層でゼロバージョンの項目 (v0_Audit のソートキー) の内容が、新しいリビジョンの内容に置き換えられます。

アプリケーションで最新の監査ステータスを取得する必要がある場合は、ソートキープレフィックス `v0_` をクエリできます。

アプリケーションでリビジョン履歴全体を取得する必要がある場合は、その項目のパーティションキーの下にあるすべての項目をクエリして、`v0_` 項目をフィルターで除外することができます。

この設計では、ソートキープレフィックスの後にソートキーの各部品 ID が含まれている場合、機器の複数の部分を監査することもできます。

DynamoDB でセカンダリインデックスを使用するためのベストプラクティス。

多くの場合、セカンダリインデックスは、アプリケーションに必要なクエリパターンをサポートするために必要です。同時に、セカンダリインデックスを過度に使用するか、非効率的に使用すると、コストがかかり、不要にパフォーマンスが低下する可能性があります。

目次

- DynamoDB のセカンダリインデックスの一般的なガイドライン
  - インデックスを効率的に使用する
  - 計画を慎重に選択する
  - フェッチを回避するための頻繁なクエリの最適化
  - ローカルセカンダリインデックス作成時に項目コレクションのサイズ制限に注意する
- スパースインデックスの利用
  - DynamoDB のスパースインデックスの例
- DynamoDB のマテリアライズされた集計クエリでグローバルセカンダリインデックスを使用する
  - シナリオとアクセスパターンの例
  - 事前計算された集計を行う理由
  - テーブル設計
  - Streams と AWS Lambda を使用した集約パイプライン
  - スパース GSI 設計
  - GSI のクエリ
  - 考慮事項
- DynamoDB のグローバルセカンダリインデックスの多重定義

- DynamoDB の選択テーブルクエリでグローバルセカンダリインデックス書き込みシャーディングを使用する
  - パターンの設計
  - シャーディング戦略
  - シャーディングされた GSI のクエリ
  - 並列クエリの実行に関する考慮事項
  - コード例
- DynamoDB のグローバルセカンダリインデックスを使用して結果的に整合性のあるレプリカを作成する

DynamoDB のセカンダリインデックスの一般的なガイドライン

Amazon DynamoDB では、次の 2 種類のセカンダリインデックスをサポートしています。

- グローバルセカンダリインデックス (GSI) — パーティションキーおよびソートキーを持つインデックス。ベーステーブルのものとは異なる場合があります。このインデックスに関するクエリがすべてのパーティションにまたがり、ベーステーブル内のすべてのデータを対象とする可能性があるため、グローバルセカンダリインデックスは「グローバル」と見なされます。グローバルセカンダリインデックスにはサイズの制限はなく、読み込みアクティビティと書き込みアクティビティ用にプロビジョニングされた独自のスループット設定がテーブルのものとは異なります。
- ローカルセカンダリインデックス (LSI) - パーティションキーはベーステーブルと同じですが、ソートキーが異なるインデックスです。ローカルセカンダリインデックスは、ローカルセカンダリインデックスのすべてのパーティションの範囲が同じパーティションキーバリューを持つベーステーブルのパーティションに限定されるという意味で「ローカル」です。その結果、任意の 1 つのパーティションキーバリューに対してインデックスが作成された項目の合計サイズが、10 GB を超えることはありません。また、ローカルセカンダリインデックスでは、読み込みアクティビティおよび書き込みアクティビティのプロビジョニングされたスループット設定が、インデックス作成中のテーブルと共有されます。

DynamoDB の各テーブルには、最大で 20 のグローバルセカンダリインデックス (デフォルトのクォータ) と 5 つのローカルセカンダリインデックスを持つことができます。

多くの場合、グローバルセカンダリインデックスはローカルセカンダリインデックスよりも便利です。どのタイプのインデックスを使用するかの決定は、アプリケーションの要件によっても異なります。グローバルセカンダリインデックスとローカルセカンダリインデックスの比較および、どちらを

選ぶかの詳細については、「the section called “セカンダリインデックスの操作”」を参照してください。

DynamoDB でインデックスを作成する際に留意すべき一般的な原則と設計パターンは次のとおりです。

トピック

- インデックスを効率的に使用する
- 計画を慎重に選択する
- フェッチを回避するための頻繁なクエリの最適化
- ローカルセカンダリインデックス作成時に項目コレクションのサイズ制限に注意する

インデックスを効率的に使用する

インデックス数は最小限に抑えます。頻繁にクエリを行わない属性では、セカンダリインデックスを作成しないようにします。ほとんど使用されていないインデックスは、ストレージおよび I/O のコスト増大の一因になり、アプリケーションのパフォーマンスには効果がありません。

計画を慎重に選択する

セカンダリインデックスはストレージとプロビジョニング済みのスループットを消費するため、インデックスのサイズは可能な限り小さくすべきです。また、インデックスが小さいほど、テーブル全体に対してクエリを行うのに比べてパフォーマンスが向上します。使用するクエリが属性の一部しか返さないことが多く、それらの属性のサイズを合計しても項目全体より大幅に小さい場合には、頻繁にリクエストを行う属性だけを対象とするようにします。

読み込みに比べて、テーブルでの書き込みアクティビティが多くなることが予想される場合には、次のベストプラクティスに従います。

- インデックスに書き込まれる項目のサイズが最小になるように、計画される属性が少なくなるようにします。ただし、計画される属性のサイズが単一の書き込み容量ユニット (1 KB) より大きい場合にのみ適用されます。例えば、インデックスエントリのサイズが 200 バイトである場合、DynamoDB ではこれが 1 KB に切り上げられます。言い換えれば、インデックス項目のサイズが小さい間は、追加コストが発生することなく、より多くの属性を計画することができます。
- クエリではほとんど必要にならないことが分かっている属性は計画しないでください。インデックスで計画されている属性を更新する度に、インデックス更新による追加コストが発生します。プロビジョニングされたスループットの高いコストでは、`Query` では射影されない属性を引き続き検

索できますが、クエリのコストは頻繁にインデックスを更新するコストよりも大幅に低くなる可能性があります。

- ALL は、異なるソートキーによってソートされるテーブル項目全体を返るようにする場合にのみ指定します。すべての属性を計画することでテーブルをフェッチする必要がなくなりますが、ほとんどの場合、ストレージおよび書き込みアクティビティに要するコストが倍加します。

次のセクションで説明するように、フェッチを最小限に抑える必要性に応じて、インデックスをできるだけ小さくします。

フェッチを回避するための頻繁なクエリの最適化

レイテンシーを可能な限り小さくしてクエリを最速にするには、クエリによって返ることが予想されるすべての属性を計画します。特に、計画されていない属性のローカルセカンダリインデックスにクエリを実行すると、DynamoDB は自動的にテーブルからこれらの属性を取得します。そのため、項目全体をテーブルから読み取る必要があります。これにより、レイテンシーと不要な I/O オペレーションを減らすことができます。

「不定期な」クエリは、「必須」のクエリに変化することがあります。不定期にのみ、クエリを実行することが予想されるために計画しない属性がある場合は、状況が変わる可能性があるかどうかを検討します。

テーブルのフェッチの詳細については、「ローカルセカンダリインデックスに対するプロビジョニングされたスループットに関する考慮事項」を参照してください。

ローカルセカンダリインデックス作成時に項目コレクションのサイズ制限に注意する

項目コレクションとは、テーブルとそのローカルセカンダリインデックス内で、同じパーティションキーを持つすべての項目を意味します。10 GB を超えることができる項目コレクションはないため、パーティションキーバリューによっては容量が不足する可能性があります。

テーブル項目を追加または更新すると、DynamoDB は影響を受けるローカルセカンダリインデックスをすべて更新します。インデックスが付けられた属性がテーブル内で定義されている場合は、ローカルセカンダリインデックスも増加します。

ローカルセカンダリインデックスを作成する場合は、インデックスに書き込まれるデータの量と、および同じパーティションキーバリューを含むデータ項目数を考慮します。特定のパーティションキーバリューに対するテーブルおよびインデックス項目の合計が 10 GB を超えると予想される場合は、そのインデックスの作成を回避できないかどうかを検討してください。

ローカルセカンダリインデックスの作成を回避できない場合は、項目コレクションのサイズ制限を超える前に対処する必要があります。ベストプラクティスとして、10 GB のサイズ制限に近づいたアイテムコレクションのサイズを監視して警告するアイテムを書き込む際は、`ReturnItemCollectionMetrics` パラメータを使用します。アイテムコレクションの最大サイズを超えると、書き込みが失敗します。アプリケーションに影響が出る前にアイテムコレクションのサイズを監視してアラートを出すことで、アイテムコレクションのサイズに関する問題を軽減できます。

> **Note**
>
> ローカルセカンダリインデックスは、一度作成すると削除できません。

制限の範囲内での作業と是正措置のための方法については、「項目コレクションのサイズ制限」を参照してください。

スパースインデックスの利用

テーブル内の項目について、DynamoDB は項目内にインデックスキー属性がある場合のみ、対応するインデックスエントリを書き込みます。グローバルセカンダリインデックスの場合、これは、インデックスパーティションキーを項目で定義する必要があり、インデックスにソートキーも存在する場合、その属性も存在している必要があることを意味します。いずれかのキー属性が項目にない場合、その項目はインデックスに表示されません。ベーステーブルの項目のサブセットのみが表示されるインデックスは、スパースインデックスと呼ばれます。

スパースインデックスは、テーブルの小さなサブセクションに対して行うクエリに役立ちます。例えば、次のキー属性を使用して顧客の注文をすべて格納するテーブルがあるとします。

- パーティションキー: `CustomerId`
- ソートキー: `OrderId`

クローズされていない注文を追跡するために、未出荷の注文項目に `isOpen` というブール属性を挿入することができます。その後、注文が出荷されたら、その属性は削除できます。`CustomerId` (パーティションキー) と `isOpen` (ソートキー) でインデックスを作成した場合は、`isOpen` が定義されている注文のみ、その中に表示されます。多くの注文のうち、少数の注文がクローズされていない場合は、テーブル全体をスキャンするよりも、クローズされていない注文のインデックスをクエリする方が時間もコストもかかりません。

isOpen のようなタイプの属性を使用する代わりに、インデックスに有益なソート順になる値を持つ属性を使用できます。例えば、各注文時の日付に設定された OrderOpenDate 属性を使用して、注文が確定した後でそれを削除することができます。このように、スパースなインデックスをクエリすると、各注文の確定日でソートされた項目が返されます。

DynamoDB のスパースインデックスの例

グローバルセカンダリインデックスは、デフォルトでスパースです。グローバルセカンダリインデックスを作成する際、パーティションキーおよびソートキー (オプション) を指定します。必要なキー属性を含むベーステーブル内の項目のみがインデックスに表示されます。項目にインデックスパーティションキー (または定義されている場合はソートキー) がない場合、その項目はインデックスから除外されます。

グローバルセカンダリインデックスをスパースに設計することにより、優れたパフォーマンスを達成しながら、ベーステーブルの書き込みスループットよりも低くプロビジョニングできます。

例えば、ゲームアプリケーションは、各ユーザーのすべてのスコアを追跡することができますが、一般的に、いくつかの高いスコアをクエリするだけで済みます。このシナリオは、次のように効率的に処理されます。

Table

Primary Key		Data Attributes...		
Partition Key	*Sort Key*			
Player_ID	Game_ID	Attribute 1	Attribute 2	Attribute 3
Rick	Game_1	Score: 36,750 *(game score)*	Date: 2017-11-14 *(date of game)*	
	Game_2	Score: 69,450 *(game score)*	Date: 2017-12-31 *(date of game)*	
	Game_3	Score: 135,900 *(game score)*	Date: 2018-01-19 *(date of game)*	Award: Champ *(type of award)*
Padma	Game_4	Score: 25,350 *(game score)*	Date: 2018-01-27 *(date of game)*	
	Game_5	Score: 69,450 *(game score)*	Date: 2028-01-19 *(date of game)*	
	Game_6	Score: 147,300 *(game score)*	Date: 2018-02-02 *(date of game)*	Award: Champ *(type of award)*
	Game_7	Score: 169,100 *(game score)*	Date: 2018-03-10 *(date of game)*	Award: Champ *(type of award)*

ここで、Rick は 3 つのゲームをプレイし、そのうちの 1 つで Champ ステータスを達成しました。Padma は 4 つのゲームをプレイし、そのうちの 2 つで Champ ステータスを達成しまし

た。Award 属性は、ユーザーが賞を獲得した項目にのみ存在することに注意してください。関連するグローバルセカンダリインデックスは次のようになります。

GSI	Primary Key	Projected Attributes...			
	Partition Key				
	Award	Player_ID	Game_ID	Score	Date
	Champ	Rick	Game_3	135,900	2018-01-19
		Padma	Game_6	147,300	2018-02-02
		Padma	Game_7	169,100	2018-03-10

グローバルセカンダリインデックスには、頻繁にクエリされる上位スコアのみ含まれます。これらのスコアは、ベーステーブルの項目の小さなサブセットです。

DynamoDB のマテリアライズされた集計クエリでグローバルセカンダリインデックスを使用する

急速に変化するデータ上で、ほぼリアルタイムの集約や主要メトリクスを維持することは、企業が意思決定を迅速に行う上で、ますます価値が高まりつつあります。例えば、音楽ライブラリでは最もダウンロードされた曲をほぼリアルタイムで紹介したい場合や、e コマースプラットフォームではトレンド商品をカテゴリ別に表示する必要がある場合があります。

DynamoDB は項目間の SUM や COUNT などの集計オペレーションをネイティブにサポートしていないため、読み取り時にこれらの値を計算するには、多数の項目をスキャンする必要があり、時間がかかり、コストがかかる場合があります。代わりに、データ変更時に集計を事前計算し、結果を通常の項目としてテーブルに保存できます。このパターンはマテリアライズされた集計と呼ばれます。

トピック

- シナリオとアクセスパターンの例
- 事前計算された集計を行う理由
- テーブル設計
- Streams と AWS Lambda を使用した集約パイプライン
- スパース GSI 設計
- GSI のクエリ
- 考慮事項

シナリオとアクセスパターンの例

以下の要件を満たす音楽ライブラリアプリケーションを検討してください。

- アプリケーションは、個々の曲のダウンロードを大容量 (1 秒あたり数千件) で記録します。
- ユーザーは、特定の月の最もダウンロードされた曲を 1 桁ミリ秒のレイテンシーで確認する必要があります。
- アプリケーションは、「今月の上位 10 曲」や「特定の月にダウンロードされたすべての曲」などのクエリもサポートする必要があります。

すべてのダウンロードレコードをスキャンして読み取り時にダウンロード数を計算すると、この規模ではコストがかかる場合があります。代わりに、ダウンロードが発生するたびに更新する実行中のカウントを維持し、効率的なクエリをサポートする方法で保存できます。

事前計算された集計を行う理由

集計を計算するには、いくつかのアプローチがあります。次の表は、一般的な代替案を比較し、DynamoDB でのマテリアライズされた集計がこのタイプのユースケースに最適であることが多い理由を示しています。

アプローチ	トレードオフ	どのようなときに使うか
読み取り時のスキャンとカウント	クエリごとにすべてのダウンロードレコードを読み取る必要があります。レイテンシーはデータ量とともに増加し、大量の読み取り容量を消費します。	レイテンシーが懸念されない非常に小さなデータセットにのみ適しています。
外部集約ストア (Amazon ElastiCache など)	管理する別のサービスにより運用が複雑になります。DynamoDB とキャッシュ間の同期ロジックが必要です。	1 ミリ秒未満の読み取りや、単純なカウントを超えた複雑な集約ロジックが必要な場合。
書き込み時のアプリケーションレベルの集約	集約ロジックを書き込みパスに結合します。ダウンロードを記録した後、カウントを更新する前にアプリケーションが失敗すると、集約が不整合になります。	同期的で強力な整合性のある集約が必要であり、追加の書き込みレイテンシーを許容できる場合。

アプローチ	トレードオフ	どのようなときに使うか
Streams と Lambda を使用したマテリアライズされた集計	書き込みパスから集計を切り離します。集計は結果整合性があります (通常は数秒遅れます)。Lambda 呼び出しコストを追加します。	読み取りレイテンシーが低く、結果整合性を許容できるほぼリアルタイムの集計が必要な場合。これがこのページで説明されているアプローチです。

マテリアライズされた集計アプローチは、書き込みパスをシンプルに保ち (ダウンロードを記録するだけで)、集約を非同期プロセスにオフロードし、結果を DynamoDB に保存して、1 桁ミリ秒のレイテンシーでクエリできるようにします。

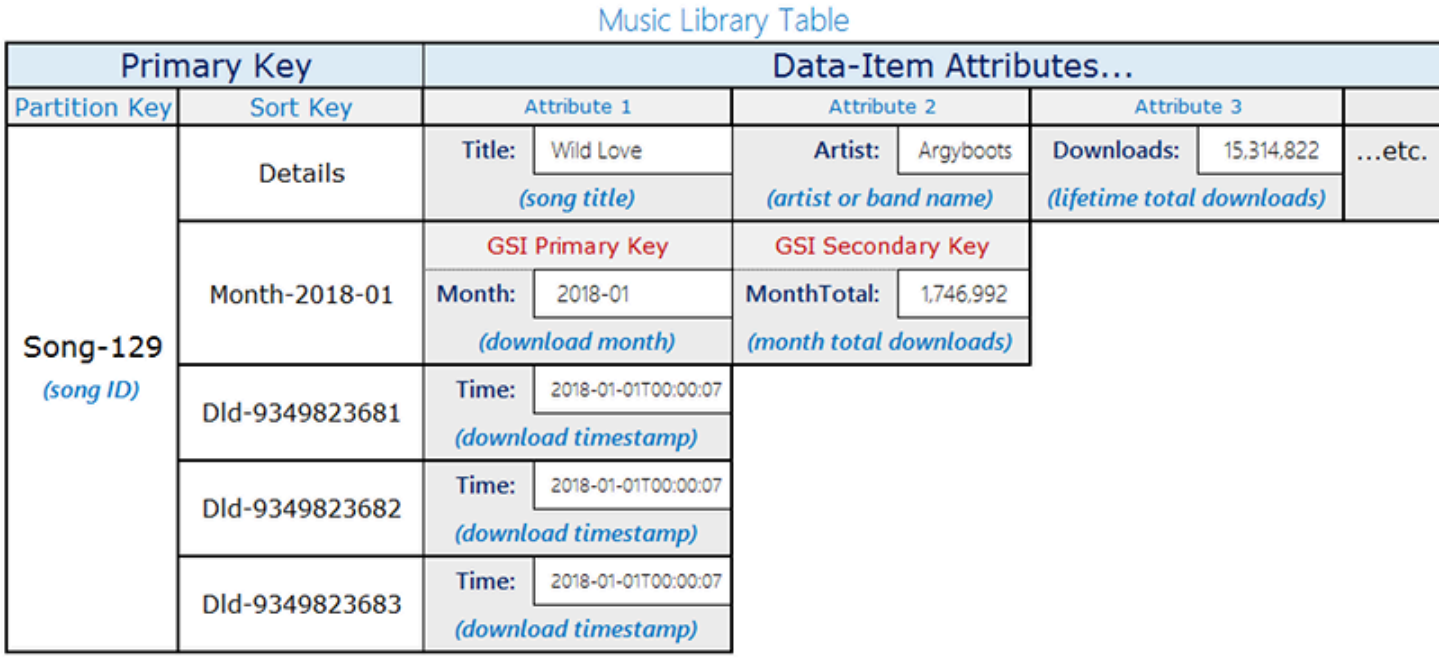
テーブル設計

この設計では、同じパーティションキー (songID) を共有する 2 つの項目タイプを持つ単一のテーブルを使用しますが、異なるソートキーパターンを使用してそれらを区別します。

- ダウンロードレコード – 個々のダウンロードイベント。ソートキーは DownloadID (ダウンロードごとに一意の識別子) です。
- 毎月の集計項目 – 1 曲あたり 1 か月あたりのダウンロード数を事前計算します。ソートキーは、YYYY-MM 形式の月です (例: 2018-01)。これらの項目には、累計を示す DownloadCount 属性も含まれます。

Month 属性を含むのは、毎月の集計項目のみです。この区別は、後で説明するスパース GSI 設計にとって重要です。

次の図は、両方の項目タイプを含むテーブルレイアウトを示しています。



項目タイプ	パーティションキー (songID)	ソートキー	その他の属性
ダウンロードレコード	song1	download-abc123	UserID, Timestamp
毎月の集計	song1	2018-01	Month=2018-01, DownloadCount =1,746,992

Streams と AWS Lambda を使用した集約パイプライン

集約パイプラインは次のように機能します。

1. 曲がダウンロードされると、アプリケーションは Partition-Key=songID と Sort-Key=DownloadID を使用してテーブルに新しい項目を書き込みます。
2. DynamoDB Streams は、この書き込みをストリームレコードとしてキャプチャします。
3. ストリームにアタッチされた Lambda 関数は、新しいレコードを処理します。songID と現在の月を識別し、DownloadCount 属性を増分して対応する毎月の集計項目を更新します。
4. 更新された集計項目は、スパース GSI を介したクエリに使用できます。

Lambda 関数は、ADD 式を使用した UpdateItem 呼び出しを使用して、ダウンロード数をアトミックに増分します。これにより、読み取り/変更/書き込みの競合状態を回避できます。

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MusicLibrary')

def handler(event, context):
    for record in event['Records']:
        if record['eventName'] == 'INSERT':
            new_image = record['dynamodb']['NewImage']
            song_id = new_image['songID']['S']
            # Derive the month from the download timestamp
            timestamp = new_image['Timestamp']['S']
            month = timestamp[:7]  # Extract YYYY-MM

            table.update_item(
                Key={
                    'songID': song_id,
                    'SK': month
                },
                UpdateExpression='ADD DownloadCount :inc SET #m = :month',
                ExpressionAttributeNames={
                    '#m': 'Month'
                },
                ExpressionAttributeValues={
                    ':inc': 1,
                    ':month': month
                }
            )
```

> **Note**
>
> 更新された集計値を書き込んだ後に Lambda の実行が失敗した場合、ストリームレコードが再試行される可能性があります。ADD オペレーションを実行するたびにカウントが増分されるため、再試行すると同じダウンロードに対してカウントが複数回増分され、おおよその値が返されます。ほとんどの分析とリーダーボードのユースケースでは、このわずかな誤差は許容範囲です。正確なカウントが必要な場合は、べき等性ロジックを追加することを検討してください。例えば、特定の DownloadID が既に処理されているかどうかをチェックする条件式を使用します。

スパース GSI 設計

集計結果を効率的にクエリするには、次のキースキーマを使用してグローバルセカンダリインデックスを作成します。

- GSI パーティションキー: `Month` (文字列)
- GSI ソートキー: `DownloadCount` (数値)

毎月の集計項目にのみ `Month` 属性が含まれているため、この GSI はスパースです。個々のダウンロードレコードにはこの属性がないため、自動的にインデックスから除外されます。つまり、GSI には事前に計算された集計項目のみが含まれます。これは、テーブル内の合計項目のごく一部です。

スパース GSI には 2 つの主な利点があります。

- コストの削減 – 集計項目のみがインデックスにレプリケートされるため、テーブル内のすべての項目を含むインデックスと比較して、書き込み容量とストレージの消費がはるかに少なくなります。
- クエリの高速化 – インデックスにはクエリが必要なデータのみが含まれているため、読み取りは効率的で、1 桁ミリ秒のレイテンシーで結果を返します。

スパースインデックスの動作の詳細については、「スパースインデックスの利用」を参照してください。

GSI のクエリ

スパース GSI を使用すると、いくつかのタイプのクエリに効率的に回答できます。

特定の月に最もダウンロードされた曲を取得します。

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false \
    --limit 1
```

ScanIndexForward を false に設定すると、結果が DownloadCount で降順にソートされ、Limit=1 は先頭の曲のみを返します。

特定の月の上位 10 曲を取得します。

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false \
    --limit 10
```

特定の月にダウンロードされたすべての曲を取得します (ダウンロード数でソート)。

```
aws dynamodb query \
    --table-name "MusicLibrary" \
    --index-name "MonthDownloadsIndex" \
    --key-condition-expression "#m = :month" \
    --expression-attribute-names '{"#m": "Month"}' \
    --expression-attribute-values '{":month": {"S": "2018-01"}}' \
    --scan-index-forward false
```

考慮事項

このパターンを実装するときは、次の点に注意してください。

- 結果整合性 – 集約値は DynamoDB Streams と Lambda を介して非同期的に更新されます。通常、ダウンロードが記録されてから集計が更新されるまでに数秒の遅延があります。つまり、GSI はリアルタイムデータではなく、ほぼリアルタイムのデータを反映します。
- Lambda の同時実行 – テーブルの書き込みボリュームが大きい場合、複数の Lambda 呼び出しが同じ集計項目を同時に更新しようとすることがあります。アトミック ADD オペレーションはこれを安全に処理しますが、関数がストリームに対応できることを確認するために、Lambda の同時実行性とスロットリングメトリクスをモニタリングする必要があります。
- GSI 書き込み容量 – スパース GSI には集計項目のみが含まれているため、必要な書き込み容量はベーステーブルよりも大幅に少なくなります。ただし、集約更新のレートを処理するのに十分な容量をプロビジョニング (またはオンデマンドモードを使用) する必要があります。

- おおよそのカウント – 前述のように、Lambda の再試行により、カウントがわずかに多くカウントされる可能性があります。正確なカウントを必要とするユースケースの場合は、Lambda 関数にべき等性チェックを実装します。

DynamoDB のグローバルセカンダリインデックスの多重定義

Amazon DynamoDB には、テーブルあたりの 20 グローバルセカンダリインデックスのデフォルトのクォータがありますが、実際には 20 以上のデータフィールドでインデックスを作成できます。スキーマが均一であるリレーショナルデータベース管理システム (RDBMS) のテーブルとは対照的に、DynamoDB のテーブルでは、一度に多数の異なるタイプのデータ項目を保持することができます。さらに、異なる項目に含まれている同じ属性に、まったく異なる種類の情報を含めることができます。

さまざまな種類のデータを保存する DynamoDB テーブルレイアウトの次の例を考えてみましょう。

Primary Key		Data-Item Attributes...		
Partition Key	Sort Key	Attribute 1	Attribute 2	...
HR-974 *(employee ID)*	Employee_Name	Data: Murphy, John *(employee name)*	Start: 2008-11-08 *(start date)*	...etc.
	YYYY-Q1	Data: $5,477 *(order totals in USD)*	Name: Murphy, John *(employee name)*	
	HR_confidential	Data: 2008-11-08 *(hire date)*	Name: Murphy, John *(employee name)*	...etc.
	Warehouse_01	Data: Murphy, John *(employee name)*		
	v0_Job_title	Data: Operator-1 *(job title)*	Start: 2008-11-08 *(start date)*	...etc.
	v1_Job_title	Data: Operator-2 *(job title)*	Start: 2016-11-04 *(start date)*	...etc.
	v2_Job_title	Data: Supervisor-1 *(job title)*	Start: 2017-11-01 *(start date)*	...etc.

Data 属性。すべての項目に共通ですが、親項目に応じて内容が異なります。テーブルのソートキーをパーティションキーとして使用し、Data 属性をソートキーとして使用するテーブルのグローバルセカンダリインデックスを作成した場合、その 1 つのグローバルセカンダリインデックスを使用してさまざまな異なるクエリを作成できます。例えば、次のようなクエリがあります。

- Employee_Name をパーティションキーバリューとして使用し、従業員の名前 (Murphy, John など) をソートキーバリューとして使用し、グローバルセカンダリインデックスの名前で従業員を検索します。
- グローバルセカンダリインデックスを使用して、倉庫 ID (Warehouse_01 など) を検索して特定の倉庫で作業しているすべての従業員を検索します。
- 最新の採用者のリストを取得して、HR_confidential のグローバルセカンダリインデックスをパーティションキーバリューとして使用し、日付範囲をソートキーバリューとして使用してクエリします。

DynamoDB の選択テーブルクエリでグローバルセカンダリインデックス書き込みシャーディングを使用する

特定の時間枠内で最近のデータをクエリする必要がある場合、ほとんどの読み取りオペレーションにパーティションキーを提供する DynamoDB の要件が課題となる可能性があります。このようなシナリオに対処するには、書き込みシャーディングとグローバルセカンダリインデックス (GSI) の組み合わせを使用して、効果的なクエリパターンを実装できます。

このようなアプローチを採用すると、リソースを大量に使用し、コスト高となる可能性がある、テーブル全体のスキャンを実行する必要なく、時間的制約のあるデータを効率的に取得して分析できます。テーブル構造とインデックスを戦略的に設計することで、最適なパフォーマンスを維持しながら、時間ベースのデータ取得をサポートする柔軟なソリューションを作成できます。

トピック

- パターンの設計
- シャーディング戦略
- シャーディングされた GSI のクエリ
- 並列クエリの実行に関する考慮事項
- コード例

パターンの設計

DynamoDB を使用する場合、書き込みシャーディングとグローバルセカンダリインデックスを組み合わせた高度なパターンを実装して、最近のデータウィンドウ全体にわたる柔軟で効率的なクエリを実現することで、時間ベースのデータ取得の課題を克服することができます。

テーブルの構造

- パーティションキー (PK):「ユーザー名」

GSI の構造

- GSI パーティションキー (PK_GSI): 「ShardNumber#」
- GSI ソートキー (SK_GSI): ISO 8601 タイムスタンプ (「2030-04-01T12:00:00Z」など)

[TABLE] - TimeBounded　Metadata　Actions

PK	SK_GSI	Address	PK_GSI
Trenton69	2023-12-05T06:40:31.312Z	117 Hudson Divide	ShardNumber#0
Scot_Langworth-Prosacco	2024-10-09T14:44:45.819Z	84021 Herzog Skyway	ShardNumber#6
Juliet.Morissette	2025-03-28T07:20:56.538Z	4871 N Broadway	ShardNumber#4
Kay.Von71	2024-11-23T16:19:26.763Z	14554 Odell Throughway	ShardNumber#2
Kiarra43	2023-11-15T16:21:57.078Z	8471 London Road	ShardNumber#1
Marcella91	2024-12-17T09:02:31.623Z	10271 Nick Crescent	ShardNumber#6
Bernard.Lemke36	2025-09-09T00:18:34.482Z	5438 Douglas Groves	ShardNumber#2
Daisha83	2024-01-24T06:38:08.482Z	8786 Armstrong Radial	ShardNumber#3
Marcos_Schmitt58	2024-04-06T17:38:58.551Z	510 S Center Street	ShardNumber#3
Jeremie_VonRueden-Mills	2025-04-06T04:24:23.701Z	157 Koch Drives	ShardNumber#1
Verna28	2025-10-10T18:42:21.059Z	70040 Baylee Trafficway	ShardNumber#1
Trycia.Doyle	2024-01-10T17:00:08.360Z	9511 Tillman Extensions	ShardNumber#2

シャーディング戦略

10 個のシャードを使用する場合、シャード番号の範囲は 0～9 の範囲になります。アクティビティをログに記録する際は、シャード番号 を計算し (ユーザー ID でハッシュ関数を使用し、シャード数の係数を取得するなど)、これを GSI パーティションキーに追加します。このような方法を採用する

と、エントリをさまざまなシャードに分散するため、ホットパーティションのリスクの軽減につながります。

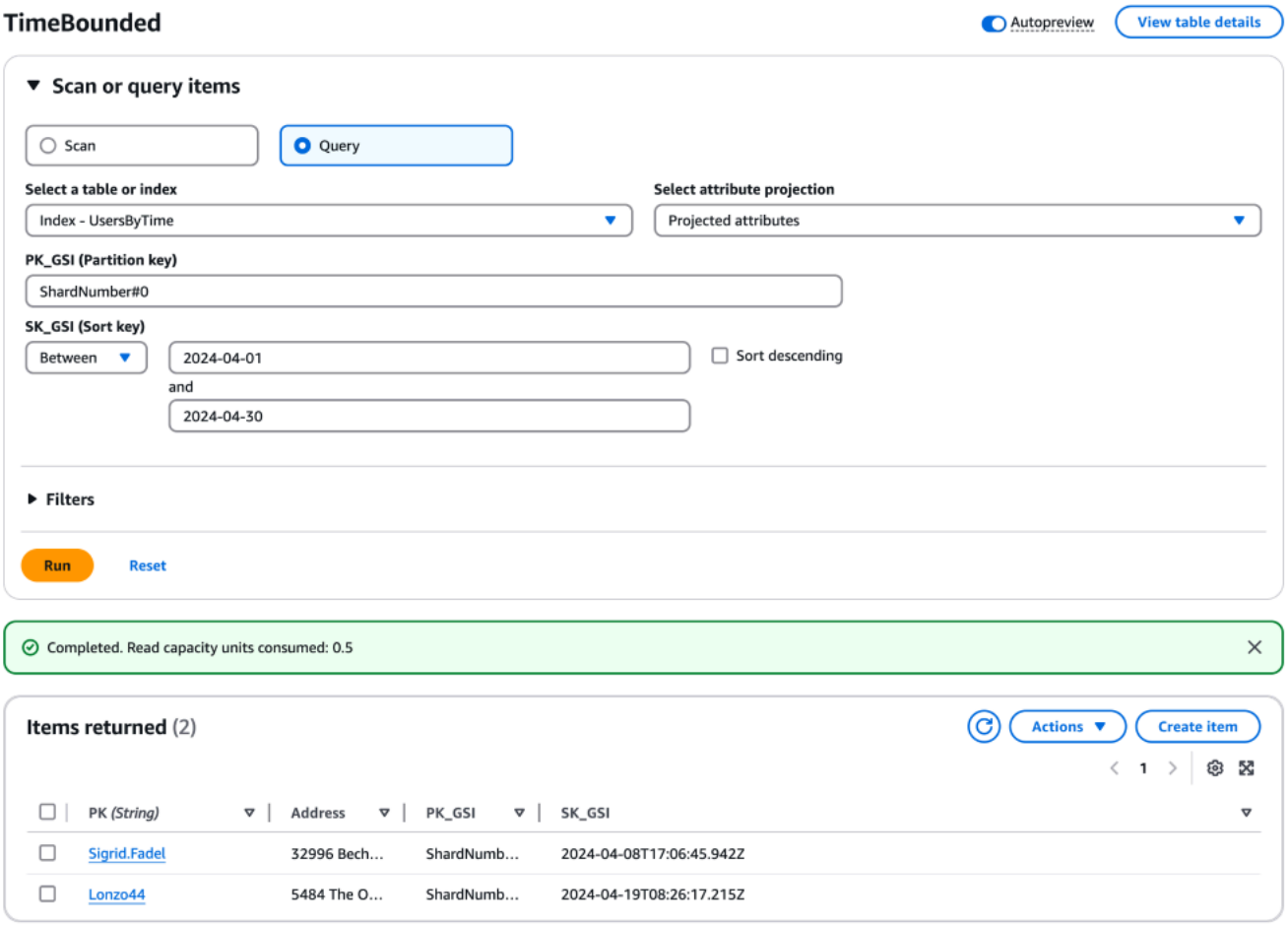
シャーディングされた GSI のクエリ

データが複数のパーティションキーにシャーディングされている DynamoDB テーブルで、特定の時間範囲内の項目について、すべてのシャードにわたってクエリを実行するには、単一のパーティションのクエリとは異なるアプローチが必要になります。DynamoDB クエリは一度に 1 つのパーティションキーに制限されているため、1 回のクエリオペレーションで複数のシャードにわたって直接クエリを実行することはできません。ただし、複数のクエリを実行し、それぞれが特定のシャードをターゲットにして結果を集計すると、アプリケーションレベルのロジックを介して目的の結果を実現できます。次の手順では、これを行う方法について説明します。

シャードをクエリして集計するには

1. シャーディング戦略で使用するシャード番号の範囲を特定します。例えば、10 個のシャードがある場合、シャード番号の範囲は 0~9 の範囲になります。
2. 目的の時間範囲内の項目を取得するクエリをシャードごとに構築して実行します。このようなクエリは、効率を向上させるために並行して実行できます。これらのクエリでは、シャード番号のパーティションキーと時間範囲のソートキーを使用します。単一のシャードのクエリサンプルは、次のとおりです。

```
aws dynamodb query \
    --table-name "YourTableName" \
    --index-name "YourIndexName" \
    --key-condition-expression "PK_GSI = :pk_val AND SK_GSI BETWEEN :start_date
 AND :end_date" \
    --expression-attribute-values '{
        ":pk_val": {"S": "ShardNumber#0"},
        ":start_date": {"S": "2024-04-01"},
        ":end_date": {"S": "2024-04-30"}
    }'
```



このクエリをシャードごとにレプリケートし、シャードに応じてパーティションキーを調整します (「ShardNumber#1」、「ShardNumber#2」...「ShardNumber#9」など)。

3. すべてのクエリが完了したら、各クエリの結果を集計します。この集計はアプリケーションコードで実行し、指定した時間範囲内のすべてのシャードの項目を表す単一のデータセットになるように、結果を結合します。

並列クエリの実行に関する考慮事項

各クエリは、テーブルまたはインデックスからの読み取り容量を使用します。プロビジョン済みのスループットを使用している場合は、並列クエリのバースト処理に十分な容量でテーブルがプロビジョンされていることを確認する必要があります。オンデマンド容量を使用している場合は、潜在的なコストへの影響に注意する必要があります。

コード例

Python を使用して DynamoDB シャード間で並列クエリを実行するには、Amazon Web Services SDK for Python である boto3 ライブラリを使用できます。この例は、boto3 がインストール済みで、適切な AWS 認証情報で設定されていることを前提としています。

次の Python コードは、特定の時間範囲で複数のシャードに対して並列クエリを実行する方法を説明しています。concurrent.futures を使用してクエリを並行して実行するため、順次実行と比べて全体的な実行時間が短縮されます。

```
import boto3
from concurrent.futures import ThreadPoolExecutor, as_completed

# Initialize a DynamoDB client
dynamodb = boto3.client('dynamodb')

# Define your table name and the total number of shards
table_name = 'YourTableName'
total_shards = 10  # Example: 10 shards numbered 0 to 9
time_start = "2030-03-15T09:00:00Z"
time_end = "2030-03-15T10:00:00Z"

def query_shard(shard_number):
    """
    Query items in a specific shard for the given time range.
    """
    response = dynamodb.query(
        TableName=table_name,
        IndexName='YourGSIName',  # Replace with your GSI name
        KeyConditionExpression="PK_GSI = :pk_val AND SK_GSI BETWEEN :date_start
 AND :date_end",
        ExpressionAttributeValues={
            ":pk_val": {"S": f"ShardNumber#{shard_number}"},
            ":date_start": {"S": time_start},
            ":date_end": {"S": time_end},
        }
    )
    return response['Items']

# Use ThreadPoolExecutor to query across shards in parallel
with ThreadPoolExecutor(max_workers=total_shards) as executor:
    # Submit a future for each shard query
```

```
    futures = {executor.submit(query_shard, shard_number): shard_number for
 shard_number in range(total_shards)}

    # Collect and aggregate results from all shards
    all_items = []
    for future in as_completed(futures):
        shard_number = futures[future]
        try:
            shard_items = future.result()
            all_items.extend(shard_items)
            print(f"Shard {shard_number} returned {len(shard_items)} items")
        except Exception as exc:
            print(f"Shard {shard_number} generated an exception: {exc}")

# Process the aggregated results (e.g., sorting, filtering) as needed
# For example, simply printing the count of all retrieved items
print(f"Total items retrieved from all shards: {len(all_items)}")
```

このコードを実行する前に、YourTableName と YourGSIName を必ず DynamoDB セットアップの実際のテーブル名と GSI 名に置き換えます。特定の要件に応じて、total_shards 変数、time_start 変数、time_end 変数を調整する必要もあります。

このスクリプトは、指定された時間範囲内の項目について各シャードをクエリして、結果を集計します。

DynamoDB のグローバルセカンダリインデックスを使用して結果的に整合性のあるレプリカを作成する

グローバルセカンダリインデックスを使用して、テーブルの結果整合性レプリカを作成できます。レプリカを作成すると、次のことが可能になります。

- リーダーごとに異なるプロビジョニングされた読み込み容量を設定する。例えば、優先順位の高いクエリを処理し、最高レベルの読み込みパフォーマンスを必要とするアプリケーションと読み込みアクティビティのスロットリングを許容できる優先順位の低いクエリを処理するアプリケーションの 2 つのアプリケーションがあるとします。

  これらのアプリケーションが同じテーブルから読み込むと、優先順位の低いアプリケーションからの読み込み負荷が高くなると、テーブルで使用可能な読み込み容量がすべて消費される可能性があります。これにより、優先度の高いアプリケーションの読み込みアクティビティが抑制されます。

代わりに、テーブル自体の読み込み容量とは別に設定できるグローバルセカンダリインデックスを通じてレプリカを作成できます。その後、テーブルの代わりに、優先度の低いアプリでレプリカにクエリすることができます。

- テーブルからの読み込みを完全に排除する。例えば、Web サイトから大量のクリックストリームアクティビティをキャプチャするアプリケーションがある場合、読み込みが妨げられる危険性はありません。このテーブルを分離し、グローバルセカンダリインデックスを使用して作成されたレプリカを他のアプリケーションに読み取らせながら、他のアプリケーションによる読み込みを防止することができます (「詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用」を参照)。

レプリカを作成するには、親テーブルと同じキースキーマを持つグローバルセカンダリインデックスを設定します。このインデックスには、キー以外の属性の一部またはすべてが入力されます。アプリケーションでは、一部またはすべての読み込みアクティビティを親テーブルではなく、このグローバルセカンダリインデックスに送信できます。その後、親テーブルのプロビジョニングされた読み込み容量を変更せずに、グローバルセカンダリインデックスのプロビジョニングされた読み込み容量を調整して、これらの読み込みを処理できます。

親テーブルへの書き込みから、書き込みデータがインデックスに表示されるまでの間には、常に短い伝播遅延があります。つまり、アプリケーションでは、グローバルセカンダリインデックスレプリカの結果整合性は親テーブルのものであるということを考慮に入れる必要があります。

複数のグローバルセカンダリインデックスレプリカを作成して、異なる読み込みパターンをサポートできます。レプリカを作成するときは、各読み込みパターンが実際に必要とする属性のみを計画します。これにより、アプリケーションは、親テーブルから項目を読み取る必要はなく、必要なデータだけを取得するために、プロビジョニングされた読み込み容量を少なくすることができます。この最適化により、長期的には大幅なコスト削減を実現できます。

DynamoDB で大きな項目と属性を格納するベストプラクティス

Amazon DynamoDB は、テーブルに格納する各アイテムのサイズを 400 KB に制限します (「項目のサイズ」を参照)。アプリケーションで DynamoDB のサイズ制限よりも多くのデータを格納する必要がある場合は、1 つ以上の大きな属性を圧縮するか、項目を複数の項目に分割することができます (ソートキーによって効率的にインデックス付けされます)。Amazon Simple Storage Service (Amazon S3) にオブジェクトとして項目を保存して、Amazon S3 オブジェクト識別子を DynamoDB 項目に格納することもできます。

ベストプラクティスとして、400 KB の最大アイテムサイズに近づいたアイテムサイズを監視して警告するアイテムを書き込む際は、`ReturnConsumedCapacity` パラメータを使用します。アイテムの最大サイズを超えると、書き込みが失敗します。DynamoDB は ValidationException error を返します。アプリケーションに影響が出る前にアイテムサイズを監視してアラートを出すことで、アイテムサイズに関する問題を軽減できます。

大量の属性値を圧縮する

大きな属性値を圧縮すると、DynamoDB の項目制限に収まり、ストレージコストが削減されます。GZIP や LZO などの圧縮アルゴリズムはバイナリ出力を生成し、これをアイテム内の `Binary` 属性タイプに格納することができます。

例えば、フォーラムユーザーの書いたメッセージを格納するテーブルを考えてみましょう。このようなメッセージは、圧縮に適した長い文字列を含むことがよくあります。圧縮するとアイテムサイズを小さくすることができますが、圧縮された属性値はフィルタリングで使用できないという問題があります。

DynamoDB で長いメッセージを圧縮する方法を示すサンプルコードについては、次のトピックを参照してください。

- 例: AWS SDK for Java ドキュメント API を使用したバイナリタイプ属性の処理
- 例: AWS SDK for .NET 低レベル API を使用したバイナリタイプ属性の処理

垂直パーティショニング

大きなアイテムを処理する際の代替解決策は、アイテムを小さなデータに分割し、関連するすべてのアイテムをパーティションキー値で関連付けることです。その後、ソートキー文字列を使用して、一緒に格納されている関連情報を特定します。さらに、複数のアイテムを同じパーティションキー値でグループ化することで、アイテムコレクションを作成できます。

このアプローチの詳細については、以下を参照してください。

- Use vertical partitioning to scale data efficiently in Amazon DynamoDB
- Implement vertical partitioning in Amazon DynamoDB using AWS Glue

大きな属性値を Amazon S3 に格納する

前述のように、Amazon S3 を使用して、DynamoDB 項目に収まらない大きな属性値を格納することもできます。Amazon S3 のオブジェクトとして保存してから、DynamoDB 項目にオブジェクト識別子を格納することができます。

また、Amazon S3 のオブジェクトメタデータサポートを使用して、DynamoDB の親項目にリンクを戻すこともできます。項目のプライマリキーバリューを Amazon S3 のオブジェクトの Amazon S3 メタデータとして格納します。この操作は、Amazon S3 オブジェクトのメンテナンスに役立ちます。

例えば、`ProductCatalog` テーブルについて考えてみましょう。このテーブル内の項目は、価格、説明、書籍の著者、その他の製品の寸法に関する情報を格納します。大きすぎて項目に収まらない各製品のイメージを保存する場合は、DynamoDB ではなく、Amazon S3 にイメージを保存できます。

この方法を実装するときは、次の制限事項に注意してください。

- DynamoDB は、Amazon S3 と DynamoDB 間のトランザクションをサポートしていません。このため、アプリケーションでは失敗が発生する状況に対処する必要があります。例えば、孤立した Amazon S3 オブジェクトのクリーンアップなどがあります。
- Amazon S3 では、オブジェクト ID の長さが制限されます。そのため、過度に長いオブジェクト識別子を生成したり、他の Amazon S3 制約に違反しないようにデータを整理する必要があります。

Amazon S3 の使用方法の詳細については、Amazon Simple Storage Service ユーザーガイドを参照してください。

DynamoDB で時系列データを処理するベストプラクティス。

Amazon DynamoDB の一般的な設計原則では、使用するテーブルの数を最小限に抑えることをお勧めします。ほとんどのアプリケーションで、必要なテーブルは 1 つだけです。ただし、時系列データでは、期間あたりアプリケーションごとにテーブルを 1 個使用すると、多くの場合最適に処理できます。

時系列データの設計パターン

大容量のイベントを追跡する一般的な時系列シナリオを考えてみます。書き込みアクセスパターンは、記録されているすべてのイベントが今日の日付であることです。お客様の読み込みアクセスパ

ターンでは、今日のイベントは最も頻繁に、昨日のイベントはそれよりはるかに低い頻度で読み込まれ、それ以前の古いイベントはほとんど読み込まれません。これを処理する 1 つの方法として、現在の日時をプライマリキーに構築できます。

次の設計パターンでは、このようなシナリオを効率的に処理します。

- 必要な読み込みおよび書き込み容量と必要なインデックスでプロビジョニングされたテーブルを 1 つの期間に 1 つ作成します。
- 各期間が終了する前に、次の期間でテーブルを事前に作成します。現在の期間が終了するように、イベントトラフィックを新しいテーブルにリダイレクトします。これらのテーブルには、記録した期間を指定する名前を割り当てることができます。
- テーブルに書き込まれなくなったら、プロビジョニングされた書き込み容量を小さい値 (1 WCU など) に減らし、適切な読み込み容量をプロビジョニングします。時間の経過と共に、以前のテーブルのプロビジョニングされた読み込み容量を減らします。コンテンツがほとんど必要ないテーブルやコンテンツがまったく必要ないテーブルをアーカイブまたは削除することもできます。

トラフィックボリュームが最も多くなる現在の期間に必要なリソースを割り当て、アクティブに使用されなくなった以前のテーブルのプロビジョニングをスケールダウンすることでコストを削減できます。ビジネスニーズによっては、トラフィックを論理パーティションキーに均等に分散するために書き込みシャーディングを検討します。詳細については、「書き込みシャーディングを使用して DynamoDB テーブルでワークロードを均等に分散させる」を参照してください。

Standard-IA テーブルクラスを使用してストレージコストを最適化する

DynamoDB には、DynamoDB Standard と DynamoDB Standard-Infrequent Access (DynamoDB Standard-IA) の 2 つのテーブルクラスがあります。Standard-IA テーブルクラスでは、読み取りおよび書き込みスループットのコストは上がりますが、ストレージコストは下がります。このクラスは、ストレージがテーブルのコストの主要な部分で、テーブルへのアクセス頻度が低い場合に適しています。

このトレードオフは、時系列パターンに自然に当てはまります。現在の期間のテーブルは、ほとんどの読み取りトラフィックと書き込みトラフィックを受け取るため、スループットのコストが低い DynamoDB Standard テーブルクラスに保持してください。時間の経過とともにテーブルがアクティブに使用されなくなると、データの保持は継続されますが、ほとんど (またはまったく) 書き込まれず、読み取り頻度が低くなります。このような古いテーブルの場合、ストレージのコストが一般的に最も高くなるため、標準 IA テーブルクラスに切り替えることで、全体的なコストを削減できます。

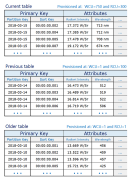
時系列テーブルにテーブルクラスを適用する場合は、次の点を考慮してください。

- DynamoDB Standard テーブルクラスで現在の (アクティブな) 期間を維持します。Standard-IA のスループットコストが高いと、大量の読み取りと書き込みを処理するテーブルのストレージ削減額を上回る可能性があります。
- 期間テーブルがアクティブに書き込まれなくなり、アクセス頻度が低くなった後も、データを保持する必要がある場合 (コンプライアンス目的や、履歴クエリをときどき実行するなど) は、テーブルを Standard-IA に切り替えてください。
- テーブルクラスはテーブルの作成時に設定でき、後で変更することもできます。切り替え前に各テーブルのストレージ対スループットのコスト比を評価し、変更後のコストをモニタリングします。

テーブルクラスとその選択方法の詳細については、「DynamoDB テーブルクラス」および「DynamoDB テーブルクラスの選択を評価する」を参照してください。

時系列テーブルの例

読み込み/書き込み容量が高いときに現在のテーブルがプロビジョニングされ、頻繁にアクセスされないために以前のテーブルが縮小される時系列データの例を以下に示します。



DynamoDB テーブルで多対多リレーションシップを管理するためのベストプラクティス

隣接関係のリストは、Amazon DynamoDB の多対多の関係のモデリングに役立つ設計パターンです。より一般的には、DynamoDB のグラフデータ (ノードとエッジ) を表現する方法です。

隣接関係のリスト設計パターン

異なるエンティティのアプリケーションの間に、多対多の関係がある場合、その関係は隣接関係のリストとしてモデル化することができます。このパターンでは、最上位エンティティ (グラフモデル内のノードと同義) はすべて、パーティションキーを使用して表現されます。他のエンティティとの関係 (グラフのエッジ) は、ソートキーの値をターゲットエンティティ ID (ターゲットノード) に設定することによって、パーティション内の項目として表現されます。

このパターンの利点として、ターゲットエンティティ (ターゲットノードへのエッジを含む) に関連するすべてのエンティティ (ノード) を見つけるために重複データを最小限に抑えられることや、クエリパターンを簡素化できる点などがあります。

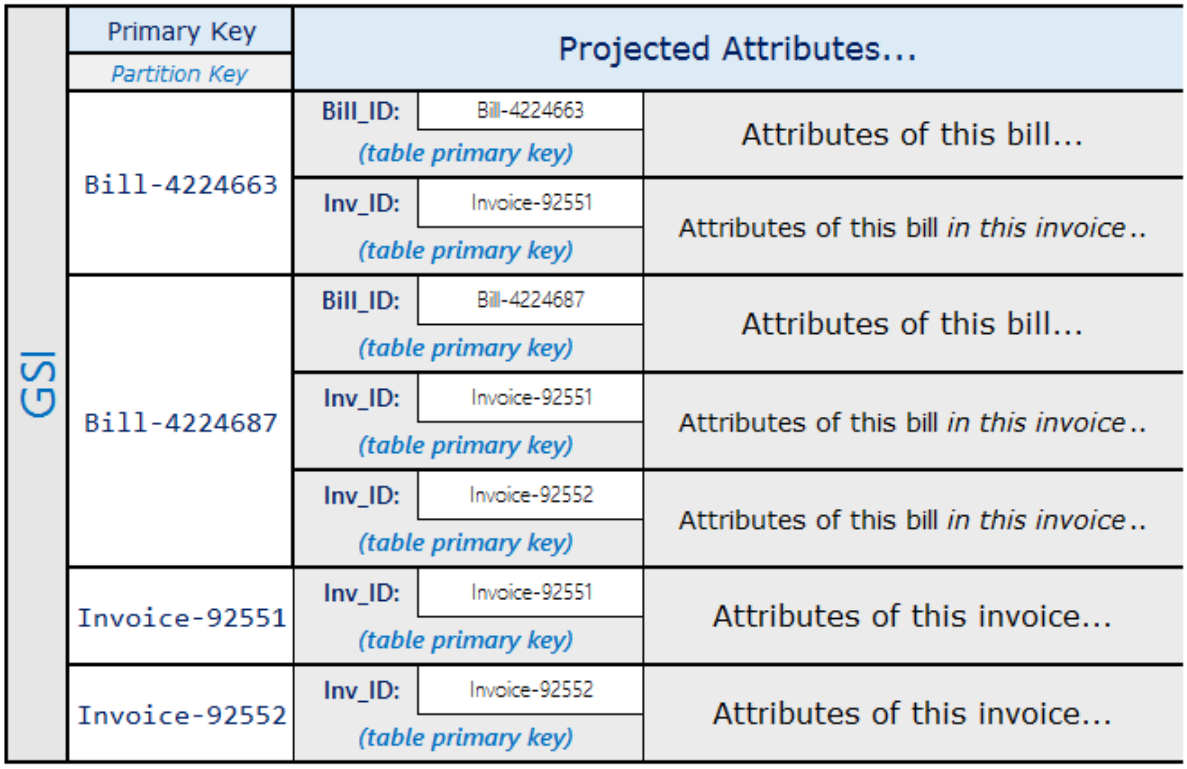
このパターンが実際に役に立った例として、請求書に複数の請求書を含めることができる請求システムがあります。1 つの請求書を複数の請求書に含めることができます。この例のパーティションキーは、InvoiceID または BillID です。BillID パーティションには、請求書固有の属性がすべて含まれます。InvoiceID パーティションには、請求書固有の属性を格納する項目と、各 BillID の項目が含まれており、この項目は、1 つの請求書にまとめられます。

スキーマは次のようになります。

Table	Primary Key		Data Attributes...	
	Partition Key	*Sort Key (and GSI PK)*		
	Invoice-92551	**Inv_ID:** Invoice-92551 *(invoice ID)*	**Dated:** 2018-02-07 *(date created)*	More attributes of this invoice...
		Bill_ID: Bill-4224663 *(bill ID)*	**Dated:** 2017-12-03 *(date created)*	Attributes of this bill *in this invoice*..
		Bill_ID: Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	Attributes of this bill *in this invoice*..
	Invoice-92552	**Inv_ID:** Invoice-92552 *(invoice ID)*	**Dated:** 2018-03-04 *(date created)*	More attributes of this invoice...
		Bill_ID: Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	Attributes of this bill *in this invoice*..
	Bill-4224663	**Bill_ID:** Bill-4224663 *(bill ID)*	**Dated:** 2017-12-03 *(date created)*	More attributes of this bill...
	Bill-4224687	**Bill_ID:** Bill-4224687 *(bill ID)*	**Dated:** 2018-01-09 *(date created)*	More attributes of this bill...

上記のスキーマを使用すると、テーブルのプライマリキーを使用して、請求書の請求内容をすべて照会することができます。請求書の一部を含むすべての請求書を検索するには、テーブルのソートキーにグローバルセカンダリインデックスを作成します。

グローバルセカンダリインデックスの射影は、次のようになります。



マテリアライズされたグラフパターン

多くのアプリケーションは、ピア間のランキング、エンティティ間の関係、およびネイバーエンティティの状態を把握する必要があります。アプリケーションでこれらのタイプのグラフスタイルのワークフローを使用している場合は、次のスキーマ設計パターンを検討してください。

実際の例として、ソーシャルネットワーキングアプリケーションを考えてみましょう。このアプリケーションでは、人々は他の人と関係を持ち、スキルを持ち、特定の場所に住み、関連する日付 (生年月日など) を持っています。各個人はグラフ内のノードです。友だちなどの人々の間のつながりは、エッジです。人と属性 (スキル、場所、日付) との間の関連もエッジです。

マテリアライズされたグラフパターンを使用すると、ノードとエッジの両方を単一の DynamoDB テーブルに保存し、関係を効率的にトラバースできます。次の図は、このソーシャルネットワーキンググラフをモデル化する方法を示しています。最初の図は、プライマリテーブル構造を示しています。次の図は、グローバルセカンダリインデックスの射影を示しています。

TABLE	Primary Key		Attributes		
	PK (NodeId)	SK (TypeTarget, GSI 2 SK)	Data	GSI PK	Graph Projections
	1	DATE|2|BIRTH	1980-12-19	Hash(Person.Data)	
		PERSON|1	Data (GSI1 SK)	GSI PK	
			John Doe	Hash(Person.Data)	
		PERSON|5|FRIEND	Data	GSI PK	
			Jane Smith	Hash(Person.Data)	
		PLACE|4|BIRTH	Data	GSI PK	
			USA|Texas|Austin	Hash(Person.Data)	
		SKILL|6	Data	GSI PK	
			Java Developer|Senior	Hash(Person.Data)	
	2	DATE|2	Data	GSI PK	
			1980-12-19	0	
	3	PLACE|3	Data	GSI PK	
			UK|England|London	0	
	4	PLACE|4	Data	GSI PK	
			USA|Texas|Austin	0	...
	5	DATE|2|BIRTH	Data	GSI PK	
			1980-12-19	Hash(Person.Data)	
		PERSON|5	Data	GSI PK	
			Jane Smith	Hash(Person.Data)	
		PERSON|1|FRIEND	Data	GSI PK	
			John Doe	Hash(Person.Data)	
		PLACE|3|BIRTH	Data	GSI PK	
			UK|England|London	Hash(Person.Data)	
		SKILL|7	Data	GSI PK	
			Guitar|Advanced	Hash(Person.Data)	
	6	SKILL|6	Data	GSI PK	
			Java Developer	0	
	7	SKILL|7	Data	GSI PK	
			Guitar	0	

	Primary Key		Attributes		
	GSI PK	GSI 1 SK (Data)			
GSI 1	O-N	1980-12-19	NodeID	TypeTarget	Graph Projections
			2	DATE|2	...
			NodeID	TypeTarget	
			1	DATE|2|BIRTH	
			NodeID		
			5		
		Guitar	NodeID	TypeTarget	
			7	SKILL|7	
		Guitar|Advanced	NodeID		
			5		
		Jane Smith	NodeID	TypeTarget	
			5	Person|5	
			NodeID	TypeTarget	
			1	Person|5|FRIEND	
		Java Developer	NodeID	TypeTarget	
			6	SKILL|6	
		Java Developer|Senior	NodeID		
			1		
		John Doe	NodeID	TypeTarget	
			1	Person|1	
			NodeID	TypeTarget	
			5	Person|1|FRIEND	
		UK|England|London	NodeID	TypeTarget	
			3	PLACE|3	
			NodeID	TypeTarget	
			1	PLACE|3|BIRTH	
		USA|Texas|Austin	NodeID	TypeTarget	
			4	PLACE|4	
			NodeID	TypeTarget	
			5	PLACE|4|BIRTH	

GSI 2	Primary Key		Attributes		
	GSI PK	GSI 2 SK (TypeTarget)	NodeID	Data	Graph Projections
	O-N	DATE|2	2	1980-12-19	...
		DATE|2|BIRTH	1		
			5		
		PERSON|1	NodeID	Data	
			1	John Doe	
		PERSON|1|FRIEND	NodeID		
			5		
		PERSON|5	NodeID	Data	
			5	Jane Smith	
		PERSON|5|FRIEND	NodeID		
			1		
		PLACE|3	NodeID	Data	
			3	UK|England|London	
		PLACE|3|BIRTH	NodeID		
			5		
		PLACE|4	NodeID	Data	
			4	USA|texas|Austin	
		PLACE|4|BIRTH	NodeID		
			1		
		SKILL|6	NodeID	Data	
			6	Java Developer	
			NodeID	Data	
			1	Java Developer|Senior	
		SKILL|7	NodeID	Data	
			7	Guitar	
			NodeID	Data	
			5	Guitar|Advanced	

このテーブルは次のキー構造を使用します。

- パーティションキー – エンティティ ID (例: `Person-1`、`Person-2`)。各パーティションには、1 つのノード項目と複数のエッジ項目が含まれます。
- ソートキー – ノード項目の場合、エンティティの ID。エッジ項目の場合、エッジタイプとターゲットの組み合わせ (例: `Friend-Person-2` または `Skill-DynamoDB`)。

エッジ項目には、`Target` 属性および `Type` 属性が含まれます。これらは、プライマリテーブルと 2 つ目のグローバルセカンダリインデックスの項目を識別する複合キー「TypeTarget」を形成します。例えば、「Person-1 は Person-2 の友だち」からは、`Type=Friend`、`Target=Person-2`、および `TypeTarget=Friend-Person-2` が生成されます。

最初のグローバルセカンダリインデックスは、`Data` 属性で構築されます。この属性は、グローバルセカンダリインデックスのオーバーロードを使用して、同一インデックス内で複数の属性タイプをインデックス化します。

- `Dates` – 生年月日、参加日 (例: `1971-12-21`)

- Names – 表示名 (例: Ana Carolina Silva)
- Places – 場所 (例: Seattle)
- Skills – コンピテンシー (例: DynamoDB)

この単一のグローバルセカンダリインデックスを使用して、特定の日に生まれたすべての人、特定の場所のすべての人、または特定のスキルを持つすべての人をクエリできます。

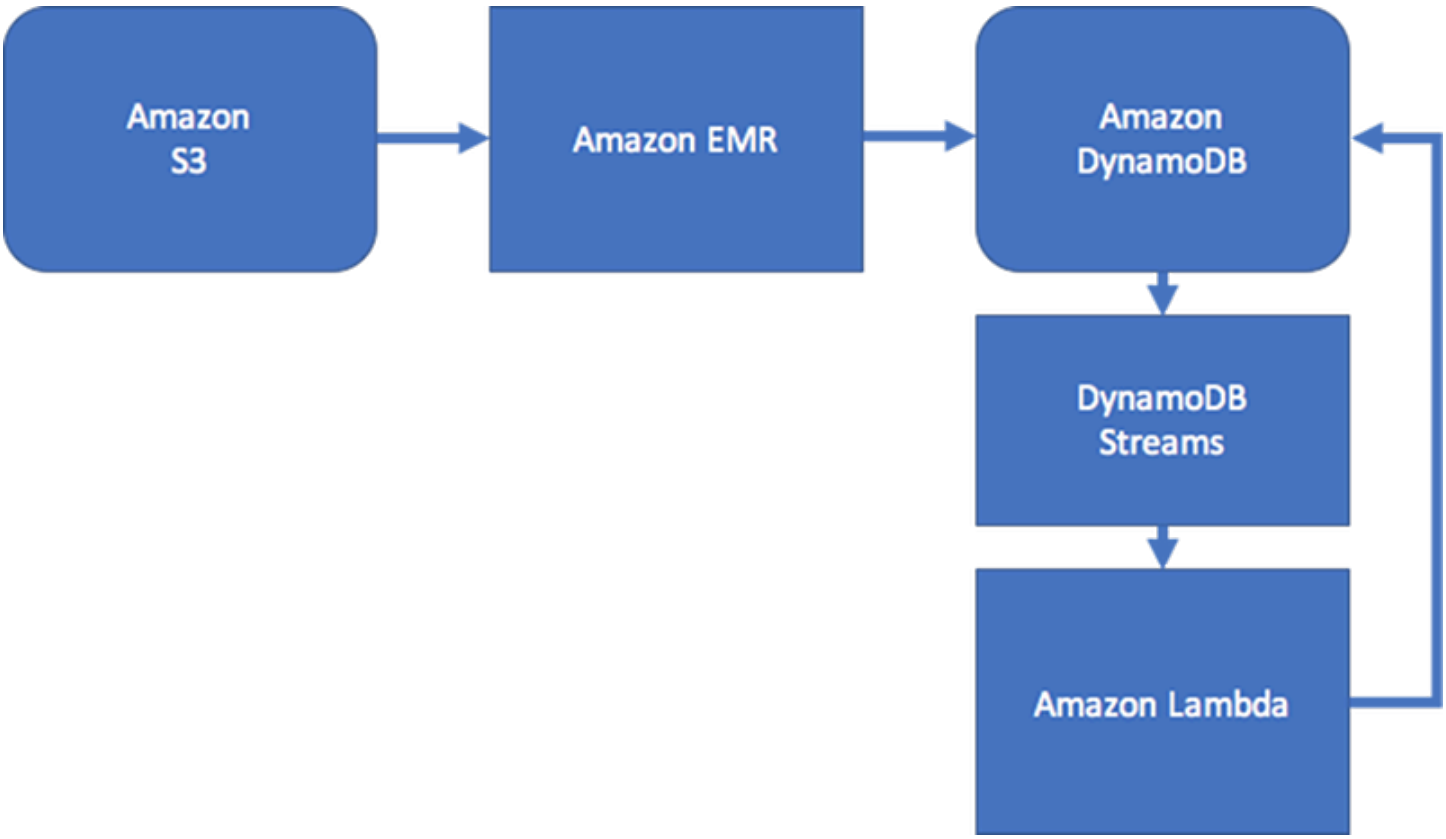
2 つ目のグローバルセカンダリインデックスは、リバースルックアップのパーティションキーとして TypeTarget を使用します。例えば、TypeTarget=Friend-Person-2 をクエリすることで、友だちとして Person-2 をリストしているすべてのユーザーを検索できます。

項目をテーブルに挿入する際に、インテリジェントなシャーディング戦略を使用して、大量の集計 (生年月日、スキル) を含む項目セットを、ホット読み書き問題を回避するために必要なグローバルなセカンダリインデックスの論理パーティションに分散させることができます。

このような設計パターンの組み合わせによって、効率の高いリアルタイムグラフワークフローを実現するソリッドなデータストアを構築できます。これを使用して、推奨エンジン、ソーシャルネットワーキングアプリケーション、ノードのランキング、サブツリーの集約、およびその他の一般的なグラフのユースケースに対して、高パフォーマンスなネイバーエンティティの状態とエッジの集約クエリを構築できます。

ユースケースがリアルタイムデータの一貫性に優れていない場合は、スケジュールされた Amazon EMR プロセスを使用して、エッジをワークフローに関連するグラフサマリー集約に追加することができます。グラフにエッジが追加されたときにアプリケーションですぐに認識する必要がない場合は、スケジュールされたプロセスを使用して結果を集約できます。

一定レベルの一貫性を維持するために、この設計には、エッジ更新を処理する Amazon DynamoDB Streams や AWS Lambda が含まれることがあります。また、Amazon EMR ジョブを使用して、定期的に結果を検証することもできます。このアプローチを次の図に示します。このアプローチは、リアルタイムクエリのコストが高く、各ユーザーの更新をすぐに知る必要性が低いソーシャルネットワーキングアプリケーションで一般的に使用されます。



IT サービス管理 (ITSM) およびセキュリティアプリケーションでは通常、複雑なエッジ集約で構成されるエンティティ状態の変化にリアルタイムで応答する必要があります。このようなアプリケーションでは、第 2 レベルと第 3 レベルの関係や複雑なエッジトラバーサルの複数のリアルタイムノード集約をサポートできるシステムが必要です。ユースケースに、これらのタイプのリアルタイムグラフクエリワークフローが必要な場合は、これらのワークフローの管理に Amazon Neptune の使用を検討することをお勧めします。

> **Note**
>
> 高度に接続されたデータセットをクエリする必要がある場合、またはミリ秒のレイテンシーで複数のノードを走査する (マルチホップクエリ) 必要がある場合は、Amazon Neptune の使用を検討してください。Amazon Neptune は、専用のハイパフォーマンスなグラフデータベースエンジンです。数十億の関係を保存し、ミリ秒単位のレイテンシーでグラフをクエリできるよう最適化されています。

DynamoDB でのデータのクエリとスキャンのベストプラクティス

このセクションでは、Amazon DynamoDB で Query オペレーションと Scan オペレーションを使用するためのベストプラクティスについて説明します。

スキャンのパフォーマンスに関する考慮事項

一般的に、Scan オペレーションは、DynamoDB の他のオペレーションよりも効率が低くなります。Scan オペレーションは常にテーブルまたはセカンダリインデックス全体をスキャンします。次に、値をフィルターして必要な結果を提供し、本質的に結果セットからデータを削除する余分なステップを追加します。

可能な場合、多くの結果を削除するフィルターを使用して大きなテーブルやインデックスで Scan オペレーションを実行することは避けるべきです。また、テーブルまたはインデックスが大きくなるにつれて、Scan オペレーションの処理速度は遅くなります。Scan オペレーションは、すべての項目でリクエストされた値を調べるので、大きなテーブルまたはインデックスに対してプロビジョニングされたスループットを 1 回のオペレーションで使い果たすことがあります。応答時間を短縮するには、アプリケーションが Scan ではなく Query を使用できるようにテーブルおよびインデックスを設計します。(テーブルの場合、GetItem API と BatchGetItem API を使用することを検討できます。)

リクエスト率への影響を最小限に抑える方法で Scan オペレーションを使用できるようアプリケーションを設計することもできます。これには、Scan オペレーションの代わりにグローバルセカンダリインデックスを使用する方が効率的な場合のモデリングが含まれる場合があります。このプロセスの詳細については、次の動画をご覧ください。

低速アクセスパターンのモデリング

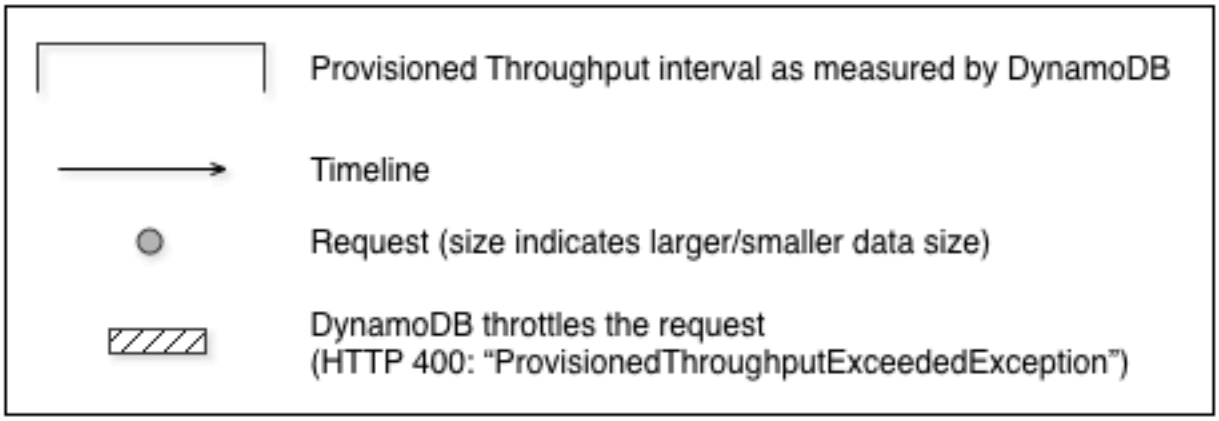
読み込みアクティビティの突然のスパイクの回避

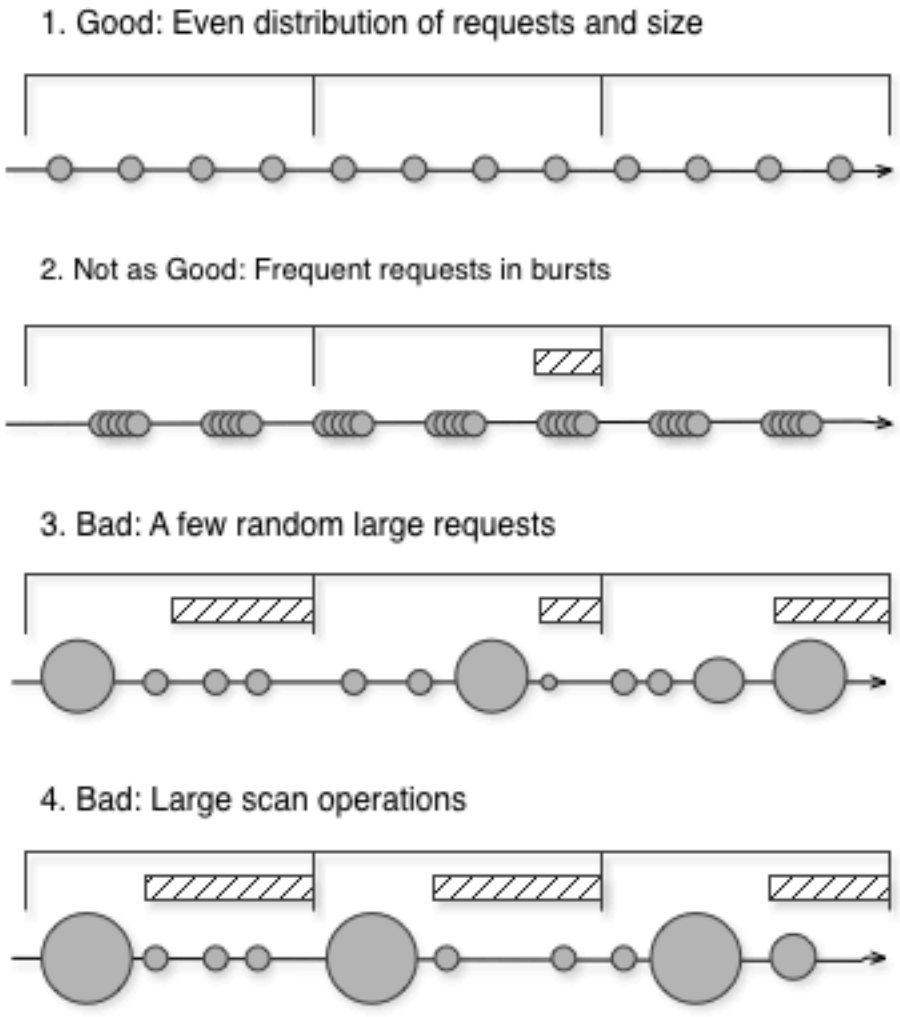
テーブルを作成する際、読み込み容量ユニットと書き込み容量ユニットの要件を設定します。読み込みの場合、容量ユニットは、1 秒間に強い整合性のある 4 KB データ読み込みリクエストの数として表されます。結果整合性のある読み込みの場合、読み込み容量ユニットは 1 秒あたり 2 回の 4 KB 読み込みリクエストです。Scan オペレーションは、デフォルトで結果整合性のある読み込みを実行し、最大 1 MB (1 ページ) のデータを返すことができます。したがって、単一の Scan リクエストは、(1 MB ページサイズ/ 4 KB 項目サイズ)/2 (結果整合性のある読み込み) = 128 の読み込みオペレーションを使用できます。強力な整合性のある読み込みをリクエストした場合、Scan オペレーションでは、プロビジョニングされたスループット (256 回の読み込みオペレーション) が 2 倍消費されます。

これは、テーブルに設定されている読み込み容量と比較して使用量が急増することを表します。スキャンによる容量ユニットの使用により、同じテーブルに対する他の潜在的に重要なリクエストで利用可能な容量ユニットが使用できなくなります。その結果、これらのリクエストで ProvisionedThroughputExceeded 例外が発生する可能性が高くなります。

問題は、Scan が使用する容量ユニットの急激な増加だけではありません。スキャンは、パーティション上で隣り合うアイテムの読み込みをリクエストするため、同じパーティションのすべての容量ユニットを消費する可能性もあります。これは、リクエストが同じパーティションに到達し、そのすべての容量ユニットが消費され、そのパーティションへの他のリクエストがスロットリングされることを意味します。データの読み込みリクエストが複数のパーティションに分散されている場合、オペレーションは特定のパーティションをスロットリングしません。

次の図は、Query オペレーションと Scan オペレーションによって容量ユニットの使用率が急増したことによる影響、および同じテーブルに対するその他のリクエストへの影響を示しています。





この図に示すように、使用率の急増は、テーブルのプロビジョニングされたスループットにいくつかの点で影響する可能性があります。

1. 良い: リクエストとサイズの均等な分散

2. あまり良くない: バーストでの頻繁なリクエスト

3. 悪い: いくつかのランダムな大きなリクエスト

4. 悪い: 大規模なスキャンオペレーション

大きな Scan オペレーションを使用する代わりに、次のテクニックを使用して、テーブルのプロビジョニングされたスループットに対するスキャンの影響を最小限に抑えることができます。

- ページサイズの削減

  スキャンオペレーションはページ全体 (デフォルトでは 1 MB) を読み込むため、ページサイズを小さくすることで、スキャンオペレーションの影響を軽減できます。Scan オペレーションは、リクエストのページサイズを設定するために使用できる Limit パラメータを提供します。同じようなページサイズの個々の Query または Scan リクエストでは使用される読み込みオペレーションの数が少なくなり、各リクエストの間で「一時停止」が作成されます。例えば、各項目が 4 KB で、ページサイズを 40 項目に設定するとします。Query リクエストは、結果整合性のある読み込み操作を 20 回しか消費せず、強く整合性のある読み込み操作を 40 回しか消費しません 多くの小さい Query または Scan オペレーションを使用すると、他の重要なリクエストをスロットリングなしで正常に完了できます。

- 複数のスキャンオペレーションの分離

  DynamoDB は、簡単にスケーリングできるように設計されています。その結果、アプリケーションは、複数の異なる目的のためにテーブルを作成することができます。場合によっては、複数のテーブル間でコンテンツを複製することも可能です。「ミッションクリティカル」なトラフィックを取らないテーブルでスキャンを実行する場合、アプリケーションによっては、2 つのテーブル間で、重要なトラフィックと記録用トラフィックなど、トラフィックを 1 時間ごとにローテーションすることで、このロードを処理します。他のアプリケーションでは、「ミッションクリティカル」なテーブルと「シャドウ」テーブルの 2 つのテーブルに対してすべての書き込みを実行することで、これを実行できます。

プロビジョニングされたスループットを超えたことを示すレスポンスコードを受け取ったリクエストを再試行するようにアプリケーションを設定します。または、UpdateTable オペレーションを使用して、テーブルのプロビジョニングされたスループットを増やします。ワークロードの一時的なスパイクが発生して、スループットがプロビジョニングされたレベルを超えた場合、エクスポネンシャルバックオフを使用してリクエストを再試行します。エクスポネンシャルバックオフの実装の詳細については、「エラーの再試行とエクスポネンシャルバックオフ」を参照してください。

並列スキャンを活用する

多くのアプリケーションでは、シーケンシャルスキャンよりも、並列 Scan オペレーションを使用することで利点が得られます。例えば、履歴データの大きなテーブルを処理するアプリケーションは、シーケンシャルデータよりもはるかに高速に並列スキャンを実行できます。バックグラウンドの「sweeper」プロセスに複数のワーカースレッドがあると、本番トラフィックに影響を与えずに、低い優先度でテーブルをスキャンできる可能性があります。これらの例では、並列 Scan は、プロビジョニングされたスループットリソースの他のアプリケーションが枯渇しないように使用されます。

並列スキャンは有益ですが、プロビジョニングされたスループットに大量のリクエストが発生する可能性があります。並列スキャンでは、アプリケーションには、Scan オペレーションが同時実行する複数のワーカーがあります。これにより、テーブルのプロビジョニングされた読み込み容量が短時間で消費されることがあります。その場合、テーブルにアクセスする必要がある他のアプリケーションでスロットリングが発生する可能性があります。

次の条件が満たされている場合、並列スキャンは正しい選択と考えられます。

- テーブルのサイズが 20 GB 以上である。
- テーブルのプロビジョニングされた読み込みスループットが完全に使用されていない。
- シーケンシャル Scan オペレーションが遅すぎる。

TotalSegments の選択

TotalSegments の最適な設定は、特定のデータ、テーブルのプロビジョニングされたスループット設定、およびパフォーマンス要件によって異なります。正しい設定を行うには、実験が必要な場合があります。データの 2 GB あたり 1 セグメントなど、単純な比率から始めることをお勧めします。例えば、30 GB のテーブルの場合、TotalSegments を 15 に設定してみます (30 GB/2 GB)。アプリケーションは 15 のワーカーを使用し、各ワーカーは異なるセグメントをスキャンします。

クライアントリソースに基づいて TotalSegments の値を選択することもできます。TotalSegments を 1 ~ 1000000 の任意の数に設定し、その数のセグメントを DynamoDB でスキャンできます。例えば、クライアントが同時に実行できるスレッドの数を制限している場合、アプリケーションで最高の Scan パフォーマンスを得るまで TotalSegments を段階的に増やすことができます。

並列スキャンをモニタリングして、プロビジョニングされたスループットの使用を最適化し、他のアプリケーションのリソースが不足していないことを確認します。プロビジョニングされたス

ループットがすべて消費されなくても Scan リクエストで引き続きスロットリングが発生する場合、TotalSegments の値を増やします。Scan リクエストが意図したものよりも多くのプロビジョニング済みスループットを消費する場合、TotalSegments の値を減らします。

DynamoDB テーブル設計のベストプラクティス

Amazon DynamoDB の一般的な設計原則では、使用するテーブルの数を最小限に抑えることをお勧めします。大部分のケースで、1 つのテーブルを使用することをお勧めします。ただし、1 つまたは少数のテーブルで対応できない場合は、以下のガイドラインを参考にしてください。

- 1 アカウントあたりのテーブル数は 10,000 が上限です。これを超えることはできません。アプリケーションでさらにテーブルが必要な場合は、テーブルを複数のアカウントに分散するように計画してください。詳細については、「Amazon DynamoDB のサービス、アカウント、およびテーブルのクォータ」を参照してください。
- テーブル管理に影響する可能性がある同時実行コントロールプレーンオペレーションについては、コントロールプレーンの制限を考慮します。
- AWS ソリューションアーキテクトと協力して、マルチテナント設計の設計パターンを検証します。

DynamoDB グローバルテーブルの使用

グローバルテーブルは、Amazon DynamoDB のグローバルフットプリントに基づいて構築され、フルマネージド、マルチリージョン、マルチアクティブのデータベースにより、大規模にスケールされたグローバルアプリケーションの高速なローカルの読み取り/書き込みパフォーマンスを実現できます。グローバルテーブルは、選択する AWS リージョン 全体で DynamoDB テーブルを自動的にレプリケートします。グローバルテーブルは既存の DynamoDB API を使用するため、アプリケーションを変更する必要はありません。グローバルテーブルの使用に伴う前払い料金やコミットメントはなく、使用したリソース分のみの支払いで済みます。

このガイドでは、DynamoDB グローバルテーブルを効果的に使用する方法について説明します。ここでは、次の項目を取り上げます。グローバルテーブルに関する重要な情報、機能の主なユースケース、2 つの整合性モード、考慮すべき 3 種類の書き込みモデルの分類基準、実装可能な 4 つの主なリクエストルーティング、稼働しているリージョンまたはオフラインのリージョンを退避させる方法、スループットキャパシティプランニングに関する考え方、グローバルテーブルをデプロイする際の考慮事項のチェックリスト。

このガイドは、「AWS マルチリージョンの基礎」ホワイトペーパーや「AWS を使用したデータ回復力の設計パターン」という動画で説明されているように、AWS マルチリージョンデプロイという、より広いコンテキストに適しています。

トピック

- DynamoDB グローバルテーブル設計に関する重要な事実
- MREC に関する重要な事実
- MRSC に関する重要な事実
- MREC DynamoDB グローバルテーブルのユースケース
- マルチアカウントグローバルテーブルに関する重要な事実
- DynamoDB グローバルテーブルによる書き込みモード
- DynamoDB のルーティング戦略
- 退避プロセス
- DynamoDB グローバルテーブルのスループットキャパシティプランニング
- DynamoDB グローバルテーブルの準備チェックリスト
- 結論とリソース

DynamoDB グローバルテーブル設計に関する重要な事実

- グローバルテーブルには、現行バージョンであるグローバルテーブルバージョン 2019.11.21 (現行) (「V2」とも呼ばれます) と グローバルテーブルバージョン 2017.11.29 (レガシー) (「V1」とも呼ばれます) の 2 つのバージョンがあります。このガイドは、現行バージョンのみを対象としています。
- DynamoDB (グローバルテーブルなし) は、リージョンサービスとして提供されるため、可用性に優れ、本質的に、インフラストラクチャの障害 (アベイラビリティーゾーン全体の障害を含む) に対するレジリエンスを備えています。単一リージョンの DynamoDB テーブルは、99.99% の可用性を実現するように設計されています。詳細については、「DynamoDB サービスレベルアグリーメント (SLA)」を参照してください。
- DynamoDB グローバルテーブルでは、2 つ以上のリージョン間でデータがレプリケートされます。マルチリージョンの DynamoDB テーブルは、99.999% の可用性を実現するように設計されています。グローバルテーブルを適切に計画すると、リージョンでの障害に対するレジリエンスが確保されたアーキテクチャを構築しやすくなります。

- DynamoDB にはグローバルエンドポイントはありません。どのリクエストも、リージョンエンドポイントに対し行われ、そのエンドポイントが、そのリージョンでローカルなグローバルテーブルインスタンスにアクセスします。
- DynamoDB への呼び出しは、リージョンを越えて実行してはなりません。ベストプラクティスとしては、あるリージョンをホームとするアプリケーションが、そのリージョンのローカル DynamoDB エンドポイントにのみ直接アクセスするようにします。リージョン内 (DynamoDB レイヤー内や周辺スタック内) で問題を検出した場合は、エンドユーザーのトラフィックを別のリージョンでホストしている別のアプリケーションエンドポイントにルーティングする必要があります。グローバルテーブルを使用すると、どのリージョンをホームとするアプリケーションも同じデータにアクセスできるようになります。

整合性モード

グローバルテーブルを作成する際には、その整合性モードを設定できます。グローバルテーブルは 2 つの整合性モードに対応しています。1 つは、マルチリージョンの結果整合性 (MREC)、もう 1 つは、2025 年 6 月に導入された、マルチリージョンの強力な整合性 (MRSC) です。

グローバルテーブルの作成時に整合性モードを指定しない場合、デフォルトでは MREC が設定されます。グローバルテーブルには、異なる整合性モードを設定したレプリカを含めることはできません。グローバルテーブルの作成後に整合性モードを変更することもできません。マルチアカウントグローバルテーブルは MREC のみをサポートします。

MREC に関する重要な事実

- MREC を使用するグローバルテーブルでは、アクティブ-アクティブレプリケーションモデルを使用します。DynamoDB の観点から見ると、各リージョンのテーブルは読み取りと書き込みのリクエストを同じように受け入れることができます。書き込みリクエストを受信したローカルレプリカテーブルでは、バックグラウンドで、書き込みオペレーションが他の参加リモートリージョンにレプリケートされます。
- 項目は個別にレプリケートされます。1 回のトランザクションで更新した複数の項目を、まとめてレプリケートすることはできません。
- ソースリージョンの各テーブルパーティションでは、他のすべてのパーティションと並行して書き込みオペレーションがレプリケートされます。リモートリージョン内の書き込みオペレーションの順序は、ソースリージョン内で発生した書き込みオペレーションの順序と一致しない場合があります。テーブルパーティションの詳細については、ブログ記事「DynamoDB のスケーリング: パーティション、ホットキー、ヒートに応じた分割がパフォーマンスに与える影響」を参照してください。

- 新しく書き込まれた項目は、通常、1 秒以内にすべてのレプリカテーブルに伝播されます。近くのリージョンにはより速く伝播される傾向があります。
- Amazon CloudWatch は、リージョンのペアごとに ReplicationLatency メトリクスを提供します。到着した項目を確認し、到着時間と最初の書き込み時間を比較し、平均を算出することでメトリクスが計算されます。タイミングはソースリージョンの CloudWatch 内に保存されます。平均タイミングと最大タイミングを表示すると、平均および最悪の場合のレプリケーションラグを判断するのに役立ちます。このレイテンシーには SLA はありません。
- 個々の項目が 2 つの異なるリージョンでほぼ同時に (この ReplicationLatency ウィンドウ内で) 更新され、最初の書き込みオペレーションがレプリケートされる前に次の書き込みオペレーションが行われると、書き込みが競合する可能性があります。MREC を使用するグローバルテーブルでは、こうした競合を解決できます。これには、書き込みオペレーションのタイムスタンプを基に、最終書き込み者優先の仕組みを使用します。つまり、最初のオペレーションよりも、2 番目のオペレーションが優先されることになります。こうした競合は CloudWatch や AWS CloudTrail には記録されません。
- 各項目には、最終書き込みタイムスタンプがプライベートシステムプロパティとして保持されます。最終書き込み者優先のアプローチを実装するには、受け取る項目のタイムスタンプが既存の項目のタイムスタンプよりも後であることを要求する条件付き書き込みオペレーションを使用します。
- グローバルテーブルでは、すべての項目がすべての参加リージョンにレプリケートされます。異なるレプリケーションスコープが必要な場合は、複数のグローバルテーブルを作成し、各テーブルに異なる参加リージョンを割り当てることができます。
- レプリカリージョンがオフラインの場合や、ReplicationLatency が増加した場合でも、ローカルリージョンでは書き込みオペレーションを受け付けます。ローカルテーブルは、各項目が成功するまで、リモートテーブルへの項目のレプリケーションを試行し続けます。
- 万が一、リージョンが完全にオフラインになっても、その後オンラインに戻った際に、すべての保留中のアウトバウンドレプリケーションとインバウンドレプリケーションが再試行されます。テーブルを同期状態に戻すために特別なアクションは必要ありません。最終書き込み者優先という仕組みにより、最終的にはデータの整合性が確保されます。
- DynamoDB MREC テーブルには、いつでも新しいリージョンを追加できます。DynamoDB では、初期同期と継続的レプリケーションが処理されます。元のリージョンも含め、リージョンの削除も可能で、これによって、そのリージョンのローカルテーブルも削除されます。

MRSC に関する重要な事実

- MRSC を使用するグローバルテーブルでも、アクティブ-アクティブレプリケーションモデルを使用します。DynamoDB の観点から見ると、各リージョンのテーブルは読み取りと書き込みのリクエストを同じように受け入れることができます。MRSC グローバルテーブルレプリカの項目の変更は、書き込み操作が正常なレスポンスを返す前に、少なくとも 1 つの他のリージョンに同期的にレプリケートされます。
- MRSC レプリカに対する強力な整合性のある読み込み操作は、常に項目の最新バージョンを返します。条件付き書き込みオペレーションでは、常に項目の最新バージョンが条件式を基に評価されます。更新は、常に、項目の最新バージョンに実行されます。
- MRSC レプリカへの、結果整合性のある読み込みオペレーションには、別のリージョンで最近発生した変更が含まれていない、あるいは、同じリージョンでごく最近発生した変更さえ含まれていない可能性があります。
- 別のリージョンで既に変更済みの項目を変更しようとすると、書き込みオペレーションは `ReplicatedWriteConflictException` の例外が発生し失敗します。`ReplicatedWriteConflictException` の例外によって失敗した書き込みオペレーションは、再試行が可能で、別のリージョンでアイテムが変更済みでなければ成功します。
- MRSC では、書き込みオペレーションと、強力な整合性のある読み込みオペレーションのレイテンシーが大きくなります。これらのオペレーションには、クロスリージョンの通信が必要です。この通信は、アクセスされるリージョンとグローバルテーブルに参加している最も近いリージョンとの間のラウンドトリップレイテンシーに基づいて増加するレイテンシーを追加できます。詳細については、「AWS re:Invent 2024 のプレゼンテーション」、「DynamoDB グローバルテーブルを使用したマルチリージョンの強力な整合性」を参照してください。結果整合性のある読み込みオペレーションでは、レイテンシーが増大することはありません。
- 項目は個別にレプリケートされます。MRSC を使用しているグローバルテーブルでは、トランザクション API はサポートされていません。
- MRSC グローバルテーブルは、厳密に 3 つのリージョンにデプロイする必要があります。MRSC グローバルテーブルは、3 つのレプリカ、または 2 つのレプリカと 1 つの監視で設定できます。監視とは、MRSC グローバルテーブルのコンポーネントであり、これには、グローバルテーブルレプリカに書き込まれた最近のデータが含まれます。監視により、フルレプリカに代わるオプションを得られる同時に、MRSC の可用性アーキテクチャに対応できます。監視に対して読み取りまたは書き込み操作を実行することはできません。監視にストレージや書き込みのコストは発生しません。監視は、2 つのレプリカとは異なるリージョン内に存在します。
- MRSC グローバルテーブルを作成するには、データが含まれていない既存の DynamoDB テーブルに、1 つのレプリカと 1 つの監視を追加するか、2 つのレプリカを追加します。既存の MRSC グ

ローバルテーブルに追加のレプリカを追加することはできません。MRSC グローバルテーブルから単一のレプリカまたは監視を削除することはできません。MRSC グローバルテーブルからは、2 つのレプリカを削除するか、1 つのレプリカと 1 つの監視を削除できます。2 番目のシナリオでは、残りのレプリカを単一リージョンの DynamoDB テーブルに変換しています。

- MRSC グローバルテーブルに監視が設定されているかどうか、またどのリージョンに設定されているかは、DescribeTable API の出力から判断できます。監視は、DynamoDB によって所有および管理されており、監視を設定したリージョンの AWS アカウントには表示されません。
- MRSC グローバルテーブルは、次のリージョンセットで使用できます。
  - US リージョンセット: 米国東部 (バージニア北部)、米国東部 (オハイオ)、米国西部 (オレゴン)
  - EU リージョンセット: 欧州 (アイルランド)、欧州 (ロンドン)、欧州 (パリ)、欧州 (フランクフルト)
  - AP リージョンセット: アジアパシフィック (東京)、アジアパシフィック (ソウル)、アジアパシフィック (大阪)
- MRSC グローバルテーブルには、リージョンセットをまたいだ設定は行えません。例えば、MRSC グローバルテーブルに、米国と欧州のリージョンセットの両方を含めることはできません。
- MRSC グローバルテーブルでは、有効期限 (TTL) はサポートされていません。
- ローカルセカンダリインデックス (LSI) は、MRSC グローバルテーブルではサポートされていません。
- CloudWatch Contributor Insights の情報は、オペレーションが発生したリージョンについてのみ報告されます。
- ローカルリージョンでは、レプリカまたは監視をホストする 2 番目のリージョンでクォーラムが確立できる限り、すべての読み取りおよび書き込みオペレーションを受け付けます。2 番目のリージョンが利用できない場合、ローカルリージョンでは、結果整合性のある読み込みのみを処理できます。
- 万が一、リージョンが完全にオフラインになっても、その後オンラインに戻った際に、自動的にキャッチアップが開始されます。キャッチアップされるまで、キャッチアップリージョンのみへの書き込みオペレーションと強力な整合性のある読み込みオペレーションは エラーを返しますが、他のリージョンへのリクエストは通常どおり実行が続行されます。キャッチアップリージョンへの結果整合性のある読み込みオペレーションにより、これまでリージョンに伝搬されたデータが、リーダーノードとローカルレプリカ間で通常生じるローカル整合性動作に従って返ります。テーブルを同期状態に戻すために特別なアクションは必要ありません。

MREC DynamoDB グローバルテーブルのユースケース

MREC グローバルテーブルにより、以下の利点を得られます。

- 低レイテンシーの読み取りオペレーション。データのコピーをエンドユーザーの近くに配置して、読み込みオペレーション中のネットワークレイテンシーを削減できます。キャッシュは `ReplicationLatency` 値と同じ最新の状態に保たれます。
- 低レイテンシーの書き込みオペレーション。近くのリージョンに書き込むことで、ネットワークレイテンシーと書き込みにかかる時間を短縮できます。書き込みトラフィックは、競合が発生しないように慎重にルーティングする必要があります。ルーティングの手法の詳細については、「DynamoDB のルーティング戦略」を参照してください。
- シームレスなリージョンの移行。新しいリージョンを追加してから古いリージョンを削除し、リージョン間でデプロイを移行できます。これに伴うデータレイヤーでのダウンタイムは発生しません。

MREC と MRSC のグローバルテーブルの両方で、以下の利点を得られます。

- 回復力とディザスタリカバリの向上。リージョンのパフォーマンスが低下したり、完全に停止した場合は、そのリージョンから退避できます。退避とは、そのリージョンに送信されるリクエストの一部またはすべてを遠ざけることを意味します。また、グローバルテーブルを使用すると、月間稼働率の DynamoDB SLA が 99.99% から 99.999% に向上します。MREC を使用すると、目標復旧時点 (RPO) と目標復旧時間 (RTO) を秒単位で測定可能になります。MRSC を使用すると、RPO をゼロにすることができます。

  例えば、Fidelity Investments 社は、re:Invent 2022 で、注文管理システムでの DynamoDB グローバルテーブルの活用についてプレゼンテーションを行いました。同社の目標は、オンプレミス処理では到達できなかった規模で信頼性に優れた低レイテンシーの処理を実現すると同時に、アベイラビリティーゾーンやリージョンでの障害に対するレジリエンスを確保することでした。

レジリエンス確保とディザスタリカバリを目標とする場合に、MRSC テーブルを使用すると、テーブルへの書き込み時と、強力な整合性のある読み込み時のレイテンシーは増大しますが、RPO ゼロのサポートが可能です。MREC グローバルテーブルの場合は、レプリカ間で生じるレプリケーション遅延と同じ長さの RPO (レプリカリージョンにもよるが、通常は数秒) に対応できます。

マルチアカウントグローバルテーブルに関する重要な事実

DynamoDB グローバルテーブルを使用すると、異なる AWS アカウントのテーブル間でデータをレプリケートできます。セキュリティまたはガバナンスコントロールでワークロードリージョンを個別の AWS アカウントに分離する必要がある場合は、マルチアカウントグローバルテーブルを使用できます。アカウントは明確な境界であり、複数のアカウント間でワークロードを分離することで、アカウント侵害シナリオにおける潜在的な影響を軽減できます。アカウントは、他のアカウントのレプリカとの間でのみデータをレプリケートできます。これらのレプリカを削除したり、ポイントインタイムリカバリ (PITR) またはバックアップ設定を変更したりすることはできません。したがって、1 つのアカウントの認証情報が侵害された場合、そのアカウントのデータのレプリカのみが影響を受けます。

アカウントは、コスト配分の明確な境界も提供します。各アカウントは、読み取り、書き込み、ストレージ、データ転送など、自分のレプリカが使用するリソースとリクエストに対してのみ料金を支払います。DynamoDB グローバルテーブルは、リージョン間のレプリケーションデータ転送には料金を請求しません。別のチームがレプリカを使用する場合、別のアカウントを使用することで、タグのみを使用する場合よりもコストの分離が明確になります。

- マルチアカウントグローバルテーブルは MREC のみをサポートします。
- 競合解決では、同一アカウントの MREC グローバルテーブルと同様の最後の書き込みを優先するメカニズムを使用します。
- マルチアカウントグローバルテーブル内のすべてのレプリカは、異なる AWS アカウントおよびリージョンに存在する必要があります。DynamoDB は、同一リージョンでのレプリケーションをサポートしていません。
- 各レプリカに一意の AWS アカウントが必要であるため、最初のクロスアカウントレプリカを追加する前に、テーブルは単一リージョンである必要があります。既存のシングルアカウントグローバルテーブルは、マルチアカウントグローバルテーブルになる前にレプリカを削除する必要があります。
- レプリケーショングループ内のすべてのレプリカには、DynamoDB レプリケーションサービスプリンシパル (`replication.dynamodb.amazonaws.com`) がレプリカ間でデータを読み書きできるようにするリソースベースのポリシーが必要です。新しいレプリカを追加する前に、このポリシーをソースレプリカに追加します。DynamoDB コンソール、AWS SDK、またはコードとしてのインフラストラクチャツールを使用できます。リソースポリシーの例については、「マルチアカウントグローバルテーブルの設定」を参照してください。
- DynamoDB はポイントインタイムリカバリ (PITR) 設定をレプリケートしません。すべてのアカウントで完全な耐障害性を実現するには、すべてのレプリカで PITR を有効にします。これにより、

いずれかのアカウントが侵害され、悪意のある書き込みまたは削除が他のアカウントにレプリケートされた場合でも、復旧パスが提供されます。

- ソースレプリカが AWS KMS キーで暗号化されている場合は、キーポリシーを更新して、DynamoDB レプリケーションサービスプリンシパル (`replication.dynamodb.amazonaws.com`) に `kms:Decrypt`、`kms:ReEncrypt*`、`kms:GenerateDataKey*`、および `kms:DescribeKey` へのアクセス許可を付与する必要があります。AWS 所有キーを使用する場合、これらのアクセス許可は必要ありません。マルチアカウントグローバルテーブルは AWS マネージドキーをサポートしていません。
- コンソールからレプリカを作成するには、レプリケート先アカウントとレプリケート先リージョンでアクションを開始します。
- 暗号化キー管理モードは、レプリカ間で一致させる必要はありません。
- コンソールを使用して、AWS 所有の暗号化キーでレプリカを作成できます。カスタマーマネージドキーを使用してレプリカを作成する必要がある場合は、AWS SDK を使用します。
- マルチアカウントグローバルテーブルの作成後、最初の 24 時間はソースレプリカを削除できません。それ以外の場合はいつでも、レプリケーショングループからレプリカを削除したり、レプリカの非レプリケート設定を変更したりできます。ただし、他のアカウントのレプリカを削除または変更することはできません。

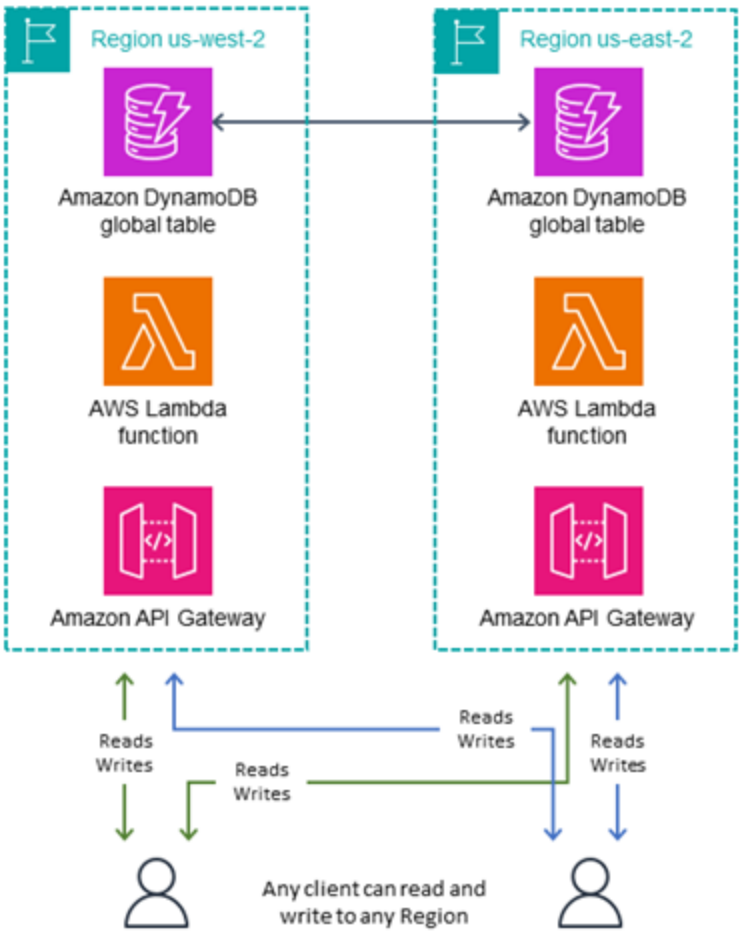
DynamoDB グローバルテーブルによる書き込みモード

グローバルテーブルは、テーブルレベルでは常にアクティブ/アクティブです。ただし、書き込みリクエストのルーティング方法を制御して、アクティブ/パッシブで構成することをお勧めします。特に MREC テーブルの場合にその方法が当てはまります。例えば、MREC テーブルで発生しかねない書き込みの競合を避けるには、書き込みリクエストを単一リージョンにルーティングするように設定すると良いでしょう。

マネージド書き込みには、主に 3 つのパターンがあります。これについては、次の 3 つのセクションで説明します。どの書き込みパターンがユースケースに適しているかを検討する必要があります。この選択は、リクエストのルーティング、リージョンの退避、ディザスタリカバリの処理方法に影響します。以降のセクションでは、アプリケーションの書き込みモードによって異なるガイダンスを提示します。

任意のリージョンへの書き込みモード (プライマリなし)

次の図に示す、任意のリージョンへの書き込みモードは、完全にアクティブ/アクティブの構成であり、書き込みが発生する場所に制限は適用されません。任意のリージョンにいつでも書き込むことができます。これは最もシンプルなモードですが、これを使用できるのは、一部のタイプのアプリケーションのみです。このモードは、すべての MRSC テーブルに適しています。すべてのライターが冪等であるため、安全に繰り返し実行可能であり、リージョンをまたいでの同時または繰り返しの書き込みオペレーションが競合しないという場合にも MREC テーブルに適しています。例えば、ユーザーが連絡先データを更新する場合などに使用できます。このモードは、冪等である特殊なケースとして、確定的なプライマリキーにおけるすべての書き込みが一意の挿入であるという、追加専用のデータセットにも適しています。最後に、このモードは、書き込みの競合リスクを許容できる場合に MREC 適しています。



任意のリージョンへの書き込みモードは、実装が最も簡単なアーキテクチャです。いつでも任意のリージョンを書き込み先にすることができるため、ルーティングが容易になります。フェイルオーバーも容易に行えます。なぜなら、MRSC テーブルの場合、その項目は常に同期され、MRSC テー

ブルの場合、最近の書き込みを任意のセカンダリリージョンで何度も実行できるからです。可能な限り、この書き込みモード向けの設計を行うようにします。

例えば、ビデオストリーミングサービスの一部では、グローバルテーブルを使用して、ブックマーク、レビュー、視聴ステータスフラグなどを追跡しています。こうしたデプロイで MREC テーブルが使用されるのは、世界中にレプリカを分散させ、各レプリカで、低レイテンシーの読み取りおよび書き込みオペレーションを行う必要があるからです。こうしたデプロイでは、すべての書き込みオペレーションがべき等である限り、任意のリージョンへの書き込みモードを使用できます。この方法を適用できるのは、いずれの更新によってもユーザーの新しい状態が直接割り当てられ、ある項目の次の正しい値が現在の値に依存しない場合です。最新のタイムコードの設定、新しいレビューの割り当て、新しい監視ステータスの設定などがその例と言えます。万が一、ユーザーの書き込みリクエストが異なるリージョンにルーティングされた場合でも、最後の書き込みオペレーションが保持され、最後の割り当てに従ってグローバル状態が確定します。このモードでの読み取りオペレーションは、最新の `ReplicationLatency` 値に応じた遅延の後、最終的に一貫性が保たれます。

別の例として、ある金融サービス会社では、システムの一部としてグローバルテーブルを使用し、各顧客のデビットカード購入の累計を管理して、その顧客のキャッシュバック特典を計算しています。また、顧客ごとに `RunningBalance` 項目を保持したいと考えています。この書き込みモードは、本質的に、べき等ではありません。トランザクションを次々と受け付け、`ADD` 式を使用してバランスを変更するからです。この式では、正しい新規の値が現在の値に依存します。そうした状況でも、MRSC テーブルでは、任意のリージョンへの書き込みモードを維持できます。いずれの `ADD` 呼び出しでも、常に項目の最新値がオペレーションの対象になるからです。

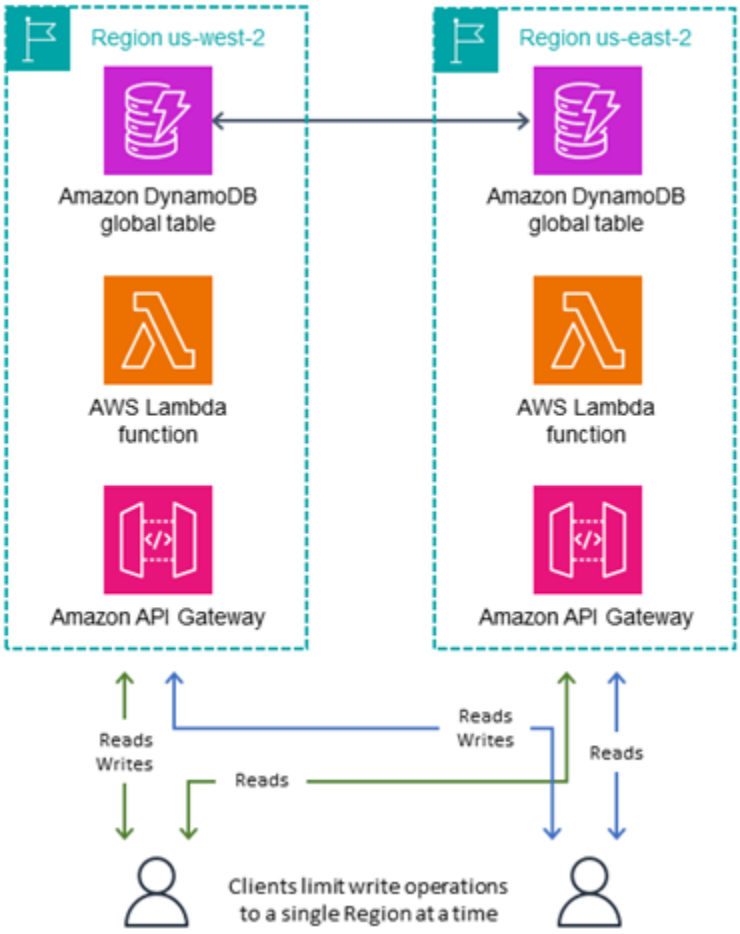
3 番目の例は、オンライン広告掲載サービスを提供する企業を示しています。この企業は、任意のリージョンへの書き込みモードによる設計の簡素化を実現するには、低リスクのデータ損失は許容できると判断しました。広告の配信時には、わずか数ミリ秒で十分なメタデータを取得してどの広告を表示するか決定すると共に、広告インプレッションを記録して同じ広告が何度もすぐに表示されないようにしています。グローバルテーブルを使用して、低レイテンシーの読み取りオペレーションで世界中のエンドユーザーに配慮し、低レイテンシーの書き込み操作も実現しています。ユーザーの広告インプレッションはすべて、単一の項目に記録し、それを成長するリストと見なしています。項目コレクションに追加する代わりに 1 つの項目を使用できます。これにより、各書き込みオペレーションの一部として古い広告インプレッションを削除できるため、削除オペレーションにコストが発生しません。この書き込み操作は、べき等ではありません。なぜなら、同じエンドユーザーに、ほぼ同時に複数のリージョンから配信された広告が表示された場合、広告インプレッションの 1 つの書き込みオペレーションによって別の書き込みオペレーションが上書きされる可能性があるからです。ここでのリスクは、ときどき、特定の広告が繰り返し表示される可能性があることです。これは許容できると同社は判断しました。

単一プライマリ (「1 つのリージョンへの書き込み」)

次の図に示すように、1 つのリージョンへの書き込みモードでは、アクティブ/パッシブの処理を行い、すべてのテーブルの書き込みを 1 つのアクティブなリージョンにルーティングします。DynamoDB には 1 つのアクティブリージョンという概念がないことに注意してください。DynamoDB の外部のアプリケーションルーティングがこれを管理します。1 つのリージョンへの書き込みモードは、書き込みオペレーションを一度に 1 つのリージョンにのみルーティングすることで、書き込みの競合を防ぐ必要がある MREC テーブルで効果を発揮します。この書き込みモードは、条件式を使用する場合や、何らかの理由で MRSC を使用できない場合、またはトランザクションの実行が必要な場合に有用です。これらの式は、オペレーション対象のデータが最新であることを把握していない限り、実行できないため、すべての書き込みリクエストを、最新のデータがある単一のリージョンに送信しなければなりません。

MRSC テーブルを使用する場合、利便性を考慮して、多くの場合、1 つのリージョンへの書き込みを選択できます。例えば、これにより、DynamoDB 以外のインフラストラクチャの構築を最小限に抑えられます。そうした状況でも、書き込みは任意のリージョンへの書き込みモードを維持できます。なぜなら、MRSC を使用すると、いつでも任意のリージョンに安全に書き込める一方で、MREC テーブルで 1 つのリージョンへの書き込みを選択した場合の競合解決を懸念する必要がないからです。

結果整合性のある読み込みは、任意のレプリカリージョンに送信してレイテンシーを短縮できます。強力な整合性のある読み込みは、単一のプライマリリージョンに送信する必要があります。



場合によっては、リージョンの障害に応じてアクティブリージョンを変更する必要があります。一部のユーザーは、フォローザサンデプロイの実装などで、現在のアクティブリージョンを定期的に変更しています。これにより、アクティビティが最も多い地域 (フォローザサンの名前のとおり、一般的に、日中である地域) の近くにアクティブリージョンを配置して、読み取りおよび書き込みオペレーションのレイテンシーを最小化しています。これには、リージョン変更コードを毎日呼び出して、ディザスタリカバリに備えたテストをそうしたコードに十分に行えるという副次的な利点もあります。

パッシブリージョンでは、DynamoDB を取り巻くインフラストラクチャをダウンスケールしたままにして、アクティブリージョンになったときにのみ構築を行う場合があります。このガイドでは、パイロットライトとウォームスタンバイの設計は取り上げていません。詳細については、「AWS でのディザスタリカバリ (DR) アーキテクチャ、パート III: パイロットライトとウォームスタンバイ」を参照してください。

1 つのリージョンへの書き込みモードが適しているのは、グローバルに分散された低レイテンシーの読み取りオペレーションにグローバルテーブルを使用する場合です。例としては、世界中のすべての

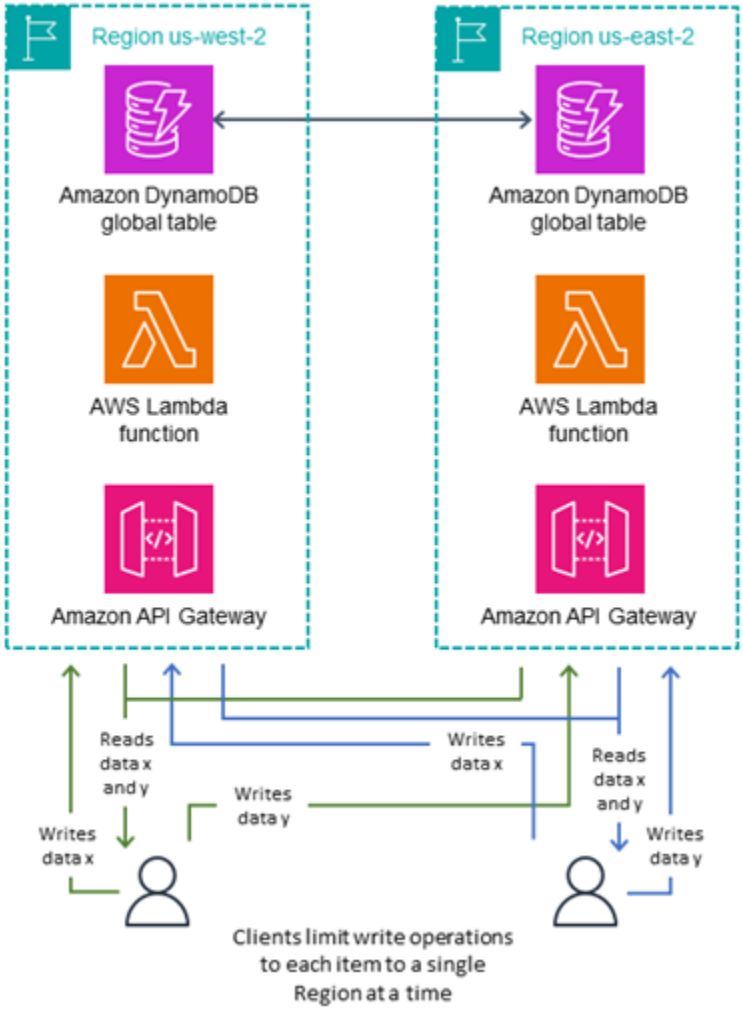
リージョンで同じ参照データを用意する必要がある大規模ソーシャルメディア企業が挙げられます。データが頻繁に更新されることはありませんが、更新されると、書き込みは、競合を避けるために 1 つのリージョンにのみ実行されます。読み取りオペレーションは、いずれのリージョンからでも常に許可されます。

別の例として、毎日のキャッシュバック計算を実装した金融サービス会社 (前述) を考えてみましょう。同社では、任意のリージョンへの書き込みモードを使用して残高を計算する一方で、1 つのリージョンへの書き込みモードを使用して支払いを追跡していました。この処理には、MRSC テーブルではサポートされていないトランザクションが必要であり、この処理を効果的に行うために、別の MREC テーブルと、1 つのリージョンへの書き込みモードを使用しています。

自分のリージョンへの書き込み (「混合プライマリ」)

次の図に示す自分のリージョンへの書き込みモードは、MREC テーブルで機能します。このモードでは、異なるデータサブセットを異なるホームリージョンに割り当て、そのホームリージョンを介してのみ項目への書き込みオペレーションを許可します。運用はアクティブ/パッシブで行いますが、項目に基づいてアクティブなリージョンを割り当てます。各リージョンは重複しない独自のデータセットを持つプライマリとして機能するため、書き込み操作を制限して適切な局所性を確保しなければなりません。

このモードは 1 つのリージョンへの書き込みと似ていますが、各ユーザーに関連するデータを、そのユーザーに近いネットワークに配置できるため、書き込みオペレーションのレイテンシーを削減できる点が異なります。また、周囲のインフラストラクチャがリージョン間でより均等に分散され、すべてのリージョンでインフラストラクチャの一部が既にアクティブになっているため、フェイルオーバーシナリオ時にインフラストラクチャを構築する作業が少なくなります。



項目のホームリージョンは、いくつかの方法で決定できます。

- 組み込み: 特殊な属性や、パーティションキーに埋め込まれた値といった、データの一面により、ホームリージョンが明確になります。この手法については、ブログ記事「リージョンのピン留めを使用して Amazon DynamoDB グローバルテーブル内の項目のホームリージョンを設定する」の解説をご覧ください。
- ネゴシエート済み: 各データセットのホームリージョンは、割り当てを管理する別のグローバルサービスなど、何らかの外部的な方法でネゴシエート済みです。割り当てには有効期間があり、その後は再ネゴシエーションの対象となります。
- テーブル指向: レプリケートするグローバルテーブルを 1 つ作成するのではなく、レプリケートするリージョンと同じ数のグローバルテーブルを作成します。各テーブルの名前はホームリージョンを示します。標準オペレーションでは、すべてのデータがホームリージョンに書き込まれ、他のリージョンは読み取り専用のコピーを保持します。フェイルオーバー中、別のリージョンがそのテーブルの書き込みタスクを一時的に引き継ぎます。

例えば、ゲーム会社で働いているとしましょう。世界中のすべてのゲーマーに配慮して、低レイテンシーの読み取りおよび書き込みオペレーションを可能にする必要があります。各ゲーマーに最も近いリージョンに各ゲーマーを割り当てます。そのリージョンでは、すべての読み取りおよび書き込みオペレーションを実行すると同時に、書き込み後の読み取りで、優れた一貫性を維持できます。ただし、ゲーマーが旅行中の場合や、ホームリージョンで障害が発生した場合は、代替リージョンでもデータの完全なコピーを利用可能にし、ゲーマーを別のホームリージョンに割り当てることができます。

別の例として、ビデオ会議会社で働いているとしましょう。各電話会議のメタデータは、特定のリージョンに割り当てられます。発信者は、最も近いリージョンを利用してレイテンシーを最小限に抑えることができます。リージョンが停止した場合、グローバルテーブルを使用していると、呼び出し処理が、レプリケート済みのデータコピーが既に存在する別のリージョンに移動されるため、迅速な復旧が可能になります。

要約

- 任意のリージョンへの書き込みモードは、MRSC テーブルと、MREC テーブルへのべき等呼び出しに適しています。
- 1 つのリージョンへの書き込みモードは、MREC テーブルへのべき等でない呼び出しに適しています。
- 自分のリージョンへの書き込みモードは、MREC テーブルへのべき等でない呼び出しに適しています。ここでは、クライアントが自身に近いリージョンに書き込みを行えるようにすることが重要です。

DynamoDB のルーティング戦略

グローバルテーブルのデプロイで最も複雑な部分は、おそらくリクエストルーティングの管理です。リクエストは、まずエンドユーザーから、何らかの方法で経路が選定されたリージョンに送る必要があります。リクエストは、そのリージョンでサービスのスタックに遭遇します。これには、AWS Lambda 関数、コンテナ、または Amazon Elastic Compute Cloud (Amazon EC2) ノードでバッキングされたロードバランサーを構成要素とするコンピューティングレイヤーや、場合によっては他のサービス (別のデータベースなど) が含まれます。このコンピューティングレイヤーは DynamoDB と通信します。この通信は、そのリージョンのローカルエンドポイントを使用して行う必要があります。グローバルテーブルのデータは、参加している他のすべてのリージョンにレプリケートされ、各リージョンには DynamoDB テーブルを中心に同様のサービスのスタックがあります。

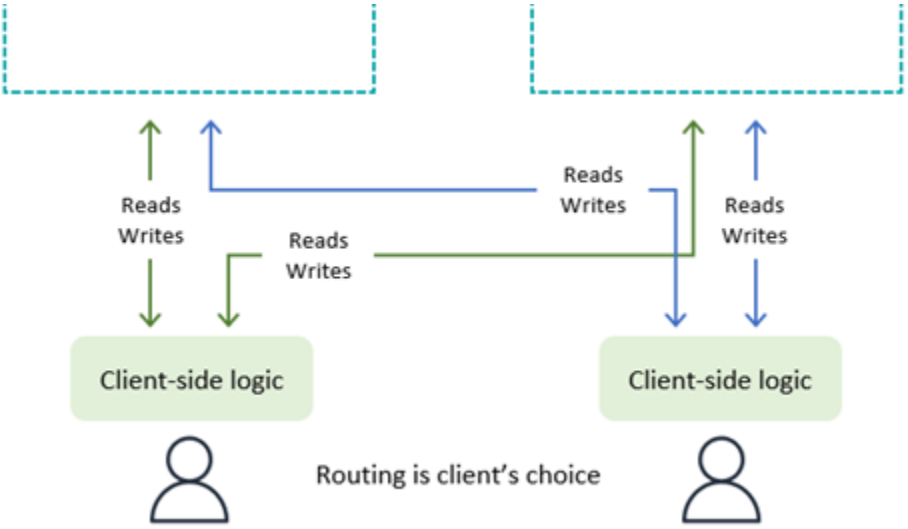
グローバルテーブルは、さまざまなリージョンの各スタックに同じデータのローカルコピーを提供します。1 つのリージョンの 1 つのスタックを対象とした設計を行い、ローカルの DynamoDB

テーブルに問題が発生した場合は、セカンダリリージョンの DynamoDB エンドポイントにリモート呼び出しを行うということも考えられます。しかし、これは、ベストプラクティスではありません。あるリージョンで DynamoDB が原因で問題が生じた場合 (または、スタック内の他の要素や、DynamoDB に依存する別のサービスが原因である可能性が高い場合) の最善策として、エンドユーザーを別のリージョンにルーティングして処理を行い、その別のリージョンのコンピューティングレイヤーを使用して、ローカル DynamoDB エンドポイントと通信すると良いでしょう。このアプローチを取ると、問題のあるリージョンを完全に迂回してルーティングを行えます。レジリエンスを確保するには、複数リージョン間でのレプリケーション、かつ、コンピューティングおよびデータレイヤーのレプリケーションが必要です。

エンドユーザーのリクエストをリージョンにルーティングして処理するための代替手法は数多くあります。最適な選択は、書き込みモードとフェイルオーバーに関する考慮事項によって異なります。このセクションでは、クライアント主導、コンピューティングレイヤー、Route 53、Global Accelerator の 4 つのオプションについて説明します。

クライアント主導のリクエストルーティング

次の図に示すように、クライアント主導のリクエストルーティングでは、エンドユーザークライアント (アプリケーション、JavaScript を使用したウェブページ、別のクライアントなど) が有効なアプリケーションエンドポイント (リテラル DynamoDB エンドポイントではなく Amazon API Gateway エンドポイントなど) を追跡します。さらに、独自の埋め込みロジックを使用して、通信対象のリージョンを選択します。この選択は、ランダムな選択、観測された最低のレイテンシー、観測された最大の帯域幅測定値、ローカルで実行されたヘルスチェックなどに基づいて行われると考えられます。



クライアント主導で行うリクエストルーティングの利点の 1 つは、パフォーマンスの低下が検出された際にリージョンを切り替えるなどして、実際のパブリックインターネットトラフィック状況に適

応できることです。クライアントは、すべての潜在的なエンドポイントを認識している必要がありますが、新しいリージョンエンドポイントの立ち上げは頻繁に起こることではありません。

任意のリージョンへの書き込みモードでは、クライアントが優先エンドポイントを一方的に選択できます。1 つのリージョンへのアクセスが損なわれた場合、クライアントは別のエンドポイントにルーティングできます。

1 つのリージョンへの書き込みモードでは、クライアントは現在アクティブなリージョンに書き込みをルーティングするメカニズムが必要になります。これは、どのリージョンが現在書き込みを受け入れているかを経験的にテストする (書き込み拒否を確認して別のリージョンにフォールバックする) という基本的なものから、グローバルコーディネーターを呼び出して現在のアプリケーションの状態をクエリするという複雑なものまであります。後者の場合は、このようなニーズに対応してグローバル状態を維持するために、Amazon Application Recovery Controller (ARC) ルーティング制御に基づいて構築し、5 リージョンのクォーラム駆動型システムを提供するようなメカニズムとなります。クライアントは、読み取りを任意のリージョンに送って結果整合性を確保するのか、アクティブなリージョンにルーティングして強力な整合性を確保するのかを決定できます。詳細については、「Route 53 の仕組み」を参照してください。

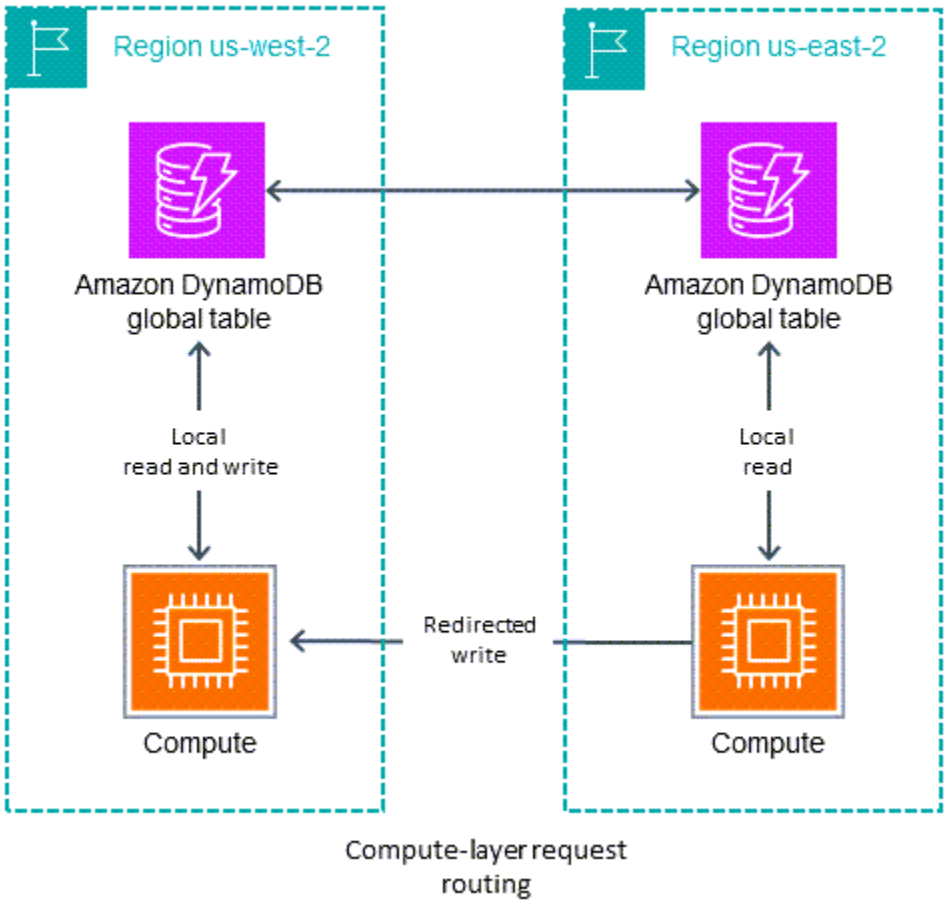
自分のリージョンへの書き込みモードでは、クライアントは使用するデータセットのホームリージョンを決定する必要があります。例えば、クライアントがユーザーアカウントと一致し、ユーザーアカウントごとに 1 つのリージョンをホームとする場合、クライアントはグローバルログインシステムに対して適切なエンドポイントをリクエストできます。

例えば、ユーザーがウェブ経由で財務を管理できるよう支援する金融サービス会社では、自分のリージョンへの書き込みモードでグローバルテーブルを使用できます。各ユーザーは中央サービスにログインする必要があります。このサービスは、認証情報と共に、認証情報が機能するリージョンのエンドポイントを返します。認証情報の有効期間は短期です。有効期間後に、ウェブページは新しいログインを自動ネゴシエートし、ユーザーのアクティビティを新しいリージョンにリダイレクトする機会を提供します。

コンピューティングレイヤーのリクエストルーティング

次の図に示すコンピューティングレイヤーのリクエストルーティングでは、コンピューティングレイヤーで実行するコードによって、リクエストをローカルで処理するか、別のリージョンで実行中の自身のコピーに渡すかを決定します。1 つのリージョンへの書き込みモードを使用する場合、コンピューティングレイヤーでは、アクティブリージョンでないことが検出すると、ローカルでの読み取りオペレーションを許可すると同時に、すべての書き込みオペレーションを別のリージョンに転送することがあります。このコンピューティングレイヤーコードは、データトポロジとルーティングルールを認識する必要があります。また、どのリージョンをどのデータに対してアクティブにするかを指

定する最新の設定に基づいて、これらのデータトポロジとルーティングルールを確実に適用する必要があります。リージョン内の外側のソフトウェアスタックは、読み取りリクエストと書き込みリクエストがマイクロサービスによってどのようにルーティングされるかを認識する必要はありません。堅牢な設計では、受信側リージョンが書き込みオペレーションの現在のプライマリであるかどうかを検証します。プライマリでない場合は、グローバル状態を修正する必要があることを示すエラーが生成されます。プライマリリージョンが変更中の場合、受信側リージョンが書き込みオペレーションをしばらくバッファリングすることもあります。いずれの場合も、リージョン内のコンピューティングスタックはローカルの DynamoDB エンドポイントにのみ書き込みますが、コンピューティングスタック間で相互に通信する可能性があります。

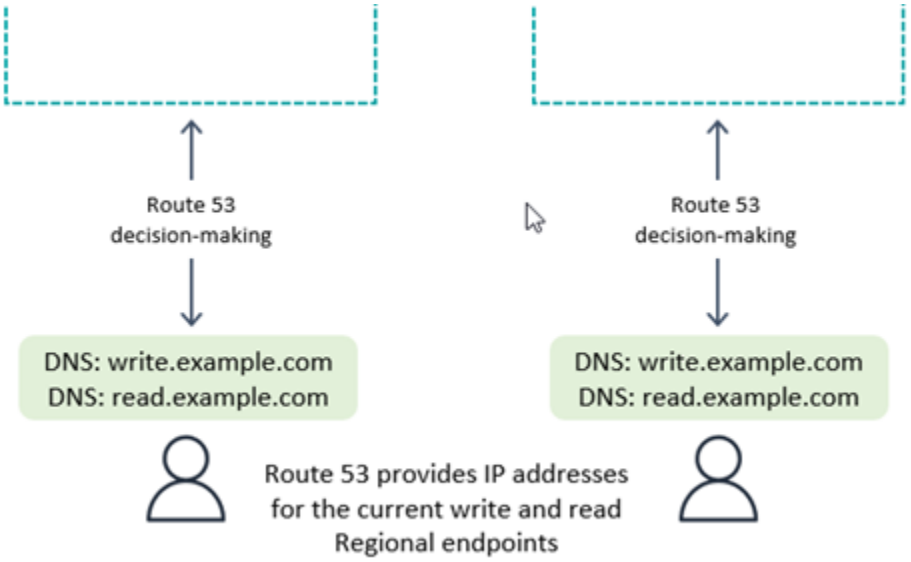


Vanguard グループでは、re:Invent 2022 での説明のように、グローバルオーケストレーションとステータスツール (GOaST) というシステムと、グローバルマルチリージョンライブラリ (GMRlib) というライブラリをこのルーティングプロセスに使用しています。フォローザサンの単一プライマリモデルを採用しています。GOaST では、前のセクションで説明した ARC ルーティングコントロールと同様に、グローバル状態が維持されます。このシステムでは、グローバルテーブルを使用して、どのリージョンがプライマリで、プライマリ状態の次回の切り替えがいつ予定されているかを追跡しています。すべての読み取りおよび書き込みオペレーションは、GOaST と連携する GMRlib を経由し

てルーティングされます。GMRlib を使用すると、ローカルでの読み取りオペレーションを低レイテンシーで実行できます。書き込みオペレーションでは、GMRlib によって、ローカルリージョンが現在のプライマリリージョンかどうかを確認します。プライマリリージョンである場合、書き込みオペレーションは直接完了します。プライマリでない場合は、GMRlib によって、書き込みタスクがプライマリリージョンの GMRlib に転送されます。受信側のライブラリは、自身もプライマリリージョンであると認識したことを確認します。そうでない場合は、エラーを生成します。これにより、グローバル状態の伝播が遅延します。このアプローチでは、リモート DynamoDB エンドポイントに直接書き込まないため、検証上の利点があります。

Route 53 のリクエストルーティング

Amazon Application Recovery Controller (ARC) は、ドメインネームサービス (DNS) テクノロジーです。Route 53 の場合、クライアントは既知の DNS ドメイン名を検索してエンドポイントをリクエストし、Route 53 は最も適切と思われるリージョンエンドポイントに対応する IP アドレスを返します。これについては、以下の図で示されています。Route 53 には、ルーティングポリシーの長いリストがあり、これを使用して適切なリージョンを決定します。フェイルオーバールーティングも実行し、これによって、ヘルスチェックに失敗したリージョンからトラフィックを迂回させます。



任意のリージョンへの書き込みモードを使用するか、バックエンドでコンピューティングレイヤーのリクエストルーティングと組み合わせることで、ネットワークに最も近いリージョン、地理的に最も近いリージョンなど、複雑な内部ルールに基づいてリージョンを返すフルアクセスを Route 53 に許可できます。

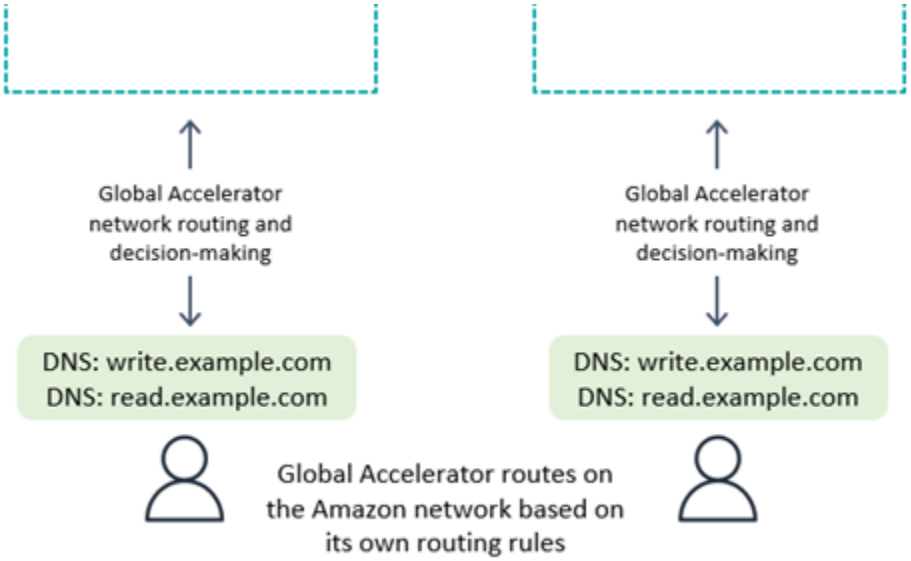
1 つのリージョンへの書き込みモードでは、現在アクティブなリージョンを返すように Route 53 を設定できます (Route 53 ARC を使用)。読み取りオペレーションなどを行うためにパッシブリージョンに接続するクライアントは、別の DNS 名を検索できます。

> **Note**
>
> クライアントは、ドメイン名の有効期間 (TTL) 設定で指定された期間にわたって、Route 53 からの応答に含まれている IP アドレスをキャッシュします。TTL を長くすると、すべてのクライアントが新しいエンドポイントを認識するまでの目標復旧時間 (RTO) が長くなります。フェイルオーバーを使用する場合は通常 60 秒です。必ずしもすべてのソフトウェアが DNS TTL の有効期限に完全に準拠しているとは限らないため、オペレーティングシステム、仮想マシン、アプリケーションなど、複数のレベルの DNS キャッシュが存在する可能性があります。

自分のリージョンへの書き込みモードでは、コンピューティングレイヤーのリクエストルーティングを一緒に使用する場合を除いて、Route 53 を避けるのが最善です。

Global Accelerator のリクエストルーティング

次の図に示すように、AWS Global Accelerator を使用する場合、クライアントは Route 53 で既知のドメイン名を検索します。ただし、クライアントはリージョンのエンドポイントに対応する IP アドレスを取得する代わりに、最も近い AWS エッジロケーションにルーティングするエニーキャスト静的 IP アドレスを受け取ります。このエッジロケーションを起点に、すべてのトラフィックはプライベート AWS ネットワークでルーティングされます。ルーティング先は、Global Accelerator 内で維持されるルーティングルールで選択された、リージョンの一部のエンドポイント (ロードバランサーや API ゲートウェイなど) です。Global Accelerator のリクエストルーティングは、Route 53 ルールに基づくルーティングと比較して、パブリックインターネットのトラフィック量を削減するため、レイテンシーが短縮されます。さらに、Global Accelerator はルーティングルールを変更するために DNS TTL の有効期限に依存しないため、ルーティングをより迅速に調整できます。



任意のリージョンへの書き込みモード、またはバックエンドのコンピューティングレイヤーのリクエストルーティングと組み合わせることで、Global Accelerator はシームレスに動作します。クライアントは最も近いエッジロケーションに接続するため、どのリージョンがリクエストを受け取るか気にする必要はありません。

1 つのリージョンに書き込む場合、Global Accelerator のルーティングルールにより、現在アクティブなリージョンにリクエストを送信する必要があります。ヘルスチェックを使用すると、グローバルシステムによってアクティブなリージョンとは見なされていないリージョンの障害を人為的に報告できます。DNS と同様に、読み取りリクエストがどのリージョンからでも可能な場合は、代替の DNS ドメイン名を使用して読み取りリクエストをルーティングできます。

自分のリージョンへの書き込みモードでは、コンピューティングレイヤーのリクエストルーティングを一緒に使用する場合を除いて、Global Accelerator を避けるのが最善です。

退避プロセス

リージョンの退避とは、アクティビティ (通常は読み取りおよび書き込みアクティビティだが、読み取りアクティビティの場合もある) をそのリージョンから移行するプロセスを意味します。

ライブリージョンからの退避

ライブリージョンからの退避の決定には、さまざまな理由が考えられます。例えば、通常のビジネスアクティビティの一部である (例: 1 つのリージョンへの書き込みモードを使い、フォローザサン方式を取っている)、現在アクティブなリージョンをビジネス上変更する、DynamoDB 外で生じたソフト

ウェアスタック障害に対応する、リージョン内のレイテンシーが通常よりも増加したといった一般的な問題が発生している、などです。

任意のリージョンへの書き込みモードを使用すると、ライブリージョンから簡単に退避できます。任意のルーティングシステムを使用して、トラフィックを代替リージョンにルーティングでき、退避したリージョンでの書き込みオペレーションを通常どおりレプリケートすることが可能です。

1 つのリージョンへの書き込みモードと、自分のリージョンへの書き込みモードは、一般的に、MREC テーブルで使用されます。したがって、新しいアクティブリージョンで書き込みオペレーションを開始する前に、アクティブリージョンへのすべての書き込みオペレーションが完全に記録、ストリーム処理、グローバルに伝播されたことを確認し、その後は、最新バージョンのデータに書き込みオペレーションが行われるようにする必要があります。

例えば、リージョン A がアクティブ、リージョン B がパッシブだとします (リージョン A をホームとするテーブル全体または項目のいずれかが対象)。退避を実行する一般的なメカニズムは、A への書き込みオペレーションを一時停止し、そのオペレーションが B に完全に伝播されるまで待ち、アーキテクチャスタックを更新して B がアクティブであることを認識してから、B への書き込みオペレーションを再開することです。リージョン A がリージョン B にデータを完全にレプリケートしたことを確実に示すメトリクスはありません。リージョン A が正常であれば、リージョン A への書き込みオペレーションを一時停止し、`ReplicationLatency` メトリクスの最新の最大値の 10 倍を待てば、通常はレプリケーションが完了したことを十分に確認できます。リージョン A に異常があり、他の領域でのレイテンシーの増加も示している場合は、待機時間の倍数を大きくします。

オフラインリージョンからの退避

考慮すべき特別なケースがあります。リージョン A が予告なしに完全にオフラインになった場合、どのような影響が生じるでしょうか? 非常にまれなケースですが、考慮する必要があります。

オフライン MRSC テーブルの退避

> そのような状況が MRSC テーブルで発生した場合、特別な操作は必要ありません。MRSC グローバルテーブルでは、目標復旧時点 (RPO) ゼロがサポートされています。オフラインリージョンの MRSC テーブルに実行され成功したすべての書き込みオペレーションは、他のすべてのリージョンテーブルに適用されるため、リージョンが予告なしに完全にオフラインになっても、データに差が生じる可能性はありません。他のリージョンにあるレプリカを業務に引き続き使用できます。

オフライン MREC テーブルの退避

そのような状況が MREC テーブルで発生した場合、リージョン A でまだ伝播されていない書き込みオペレーションはすべて保持され、リージョン A がオンラインに戻った後に伝播されます。書き込みオペレーションは失われませんが、その伝播は無期限に遅延します。

このイベントの処理方法は、アプリケーションの決定によります。ビジネスの継続性を確保するために、書き込みオペレーションを新しいプライマリリージョン B に移行することが必要になる場合があります。ただし、リージョン A の項目に対する書き込みオペレーションの伝播が保留されているときに、リージョン B でその項目が更新を受け取ると、最終書き込み者優先モデルに従ってその伝播は抑制されます。リージョン B での更新は、着信する書き込みリクエストを抑制する可能性があります。

任意のリージョンへの書き込みモードでは、リージョン A の項目が最終的にリージョン B に伝播されることを信頼し、リージョン A がオンラインに戻るまで項目が欠落する可能性を認識しながら、リージョン B で読み取りおよび書き込みオペレーションを続行できます。可能であれば、べき等書き込みオペレーションなどでは、最近の書き込みトラフィックを (アップストリームのイベントソースを使用するなどして) 再生することを検討してください。これにより、欠落している可能性のある書き込みオペレーションのギャップを埋め、最終書き込み者優先の競合解決により、着信する書き込みオペレーションの最終的な伝播を抑制します。

他の書き込みモードでは、どの程度の作業を継続できるかを、少し時代遅れの方法で考慮する必要があります。リージョン A がオンラインに戻るまで、ReplicationLatency で追跡する書き込みオペレーションの一部の短い期間は失われます。ビジネスを継続できるでしょうか。一部のユースケースでは可能ですが、他のユースケースでは追加の緩和メカニズムがないと難しい場合があります。

例えば、あるリージョン全体で障害が発生した後でも、利用可能なクレジット残高を中断することなく維持する必要があるとします。残高を 2 つの異なる項目に分割し、1 つはリージョン A に、もう 1 つはリージョン B に置き、それぞれ利用可能な残高を半分にして開始できます。この場合は、自分のリージョンへの書き込みモードを使用します。各リージョンで処理されたトランザクションの更新は、残高のローカルコピーに対して書き込まれます。リージョン A が完全にオフラインになっても、リージョン B でトランザクション処理を続行でき、書き込みオペレーションはリージョン B に保持されている残高部分に制限されます。このように残高を分割すると、残高が少なくなった場合やクレジットの再調整が必要になった場合に複雑になりますが、保留中の書き込みオペレーションが不安定でも安全にビジネスを回復できる一例となります。

別の例として、ウェブフォームのデータをキャプチャするとします。オプティミスティック同時実行制御 (OCC) (OCC) を使用してデータ項目にバージョンを割り当て、最新バージョンを

非表示フィールドとしてウェブフォームに埋め込むことができます。送信するたびに、データベース内のバージョンがフォーム作成時のバージョンとまだ一致する場合にのみ、書き込みオペレーションが成功します。バージョンが一致しない場合、データベース内の現在のバージョンに基づいてウェブフォームを更新 (または慎重にマージ) することで、ユーザーは再度続行できます。OCC モデルは通常、別のクライアントがデータを上書きして新しいバージョンを生成するのを防ぎますが、フェイルオーバー中にクライアントが古いバージョンのデータに遭遇する可能性がある場合にも役立ちます。タイムスタンプをバージョンとして使用しているとします。フォームを当初 12:00 にリージョン A に対して作成しましたが、(フェイルオーバー後に) リージョン B に書き込もうとして、データベースの最新バージョンが 11:59 であることに気付いたとします。このシナリオの場合、クライアントは 12:00 バージョンがリージョン B に伝播されるのを待ってからそのバージョンに書き込むか、11:59 バージョンに基づいて構築し、新しい 12:01 バージョンを作成することができます (書き込み後、リージョン A の回復後に着信したバージョンは抑制されます)。

3 つ目の例として、金融サービス会社が顧客アカウントおよび金融取引に関するデータを DynamoDB データベースに保持しているとします。同社は、リージョン A が完全に停止した場合、アカウントに関連する書き込みアクティビティのすべてがリージョン B で完全に利用可能であることを確認するか、リージョン A がオンラインに戻るまでアカウントを部分的なものとして隔離したいと考えていました。すべての業務を一時停止するのではなく、伝播されていないトランザクションがあると判断したごく一部のアカウントに対してのみ業務を一時停止することにしました。これを実現するために、同社は 3 番目のリージョン (リージョン C と呼びます) を使用しました。リージョン A で書き込みオペレーションを処理する前に、保留中のオペレーション (アカウントの新規トランザクション数など) の簡潔な要約をリージョン C に配置しました。この要約は、リージョン B のビューが完全に最新であるかどうかを判断するのに十分でした。このアクションにより、リージョン C での書き込み時点から、リージョン A が書き込みオペレーションを受け入れてリージョン B がそれを受け取るまでの間、アカウントは事実上ロックされました。リージョン C のデータは、フェイルオーバープロセスの一部として以外は使用されませんでした。その後、リージョン B はリージョン C とデータを相互チェックして、古いアカウントがないかどうかを確認できました。これらのアカウントは、リージョン A リカバリによって部分的なデータがリージョン B に伝播されるまで隔離済みとしてマークされます。リージョン C に障害が発生した場合は、代わりに新しいリージョン D をスピンアップして使用できます。リージョン C のデータはごく一時的なもので、数分後には処理中の書き込みオペレーションの最新の記録がリージョン D に反映されて十分に役立つようになります。リージョン B に障害が発生しても、リージョン A はリージョン C と協力して書き込みリクエストを引き続き受け入れることができます。同社は、より高いレイテンシーの書き込み (C に続けて A という 2 つのリージョンへの書き込み) をあえて受け入れ、アカウントの状態を簡潔に要約できるデータモデルを持つことができたことを喜んでいます。

DynamoDB グローバルテーブルのスループットキャパシティプランニング

リージョン間でトラフィックを移行する場合は、キャパシティに関する DynamoDB テーブルの設定を慎重に考慮する必要があります。

書き込みキャパシティの管理に関する考慮事項は以下のとおりです。

- グローバルテーブルは、オンデマンドモードにするか、自動スケーリングを有効にしてプロビジョニングする必要があります。
- 自動スケーリングでプロビジョニングした場合、書き込み設定 (最小使用率、最大使用率、およびターゲット使用率) はリージョン間でレプリケートされます。自動スケーリングの設定は同期されますが、実際にプロビジョニングされた書き込みキャパシティは、リージョン間で独立して変動する可能性があります。
- プロビジョニングされた書き込みキャパシティが異なる場合がある理由の 1 つは、TTL 機能によるものです。DynamoDB で TTL を有効にすると、項目の有効期限を秒単位の Unix エポック時間形式の値として示す属性名を指定できます。その後、DynamoDB は書き込みコストを発生させずに項目を削除できます。グローバルテーブルを使用すると、任意のリージョンで TTL を設定でき、その設定はグローバルテーブルに関連付けられている他のリージョンに自動的にレプリケートされます。項目が TTL ルールで削除の対象となった場合、削除はどのリージョンでも実行できます。削除オペレーションはソーステーブルの書き込みユニットを消費せずに実行されますが、レプリカテーブルは削除オペレーションのレプリケートされた書き込みを受け取るため、レプリケートされた書き込みユニットのコストが発生します。TTL は MRSC テーブルではサポートされていません。
- 自動スケーリングを使用する場合は、プロビジョニングされた最大書き込みキャパシティの設定が、すべての書き込みオペレーションとすべての潜在的な TTL 削除オペレーションを処理するのに十分な大きさであることを確認してください。自動スケーリングは、書き込み消費量に応じて各リージョンを調整します。オンデマンドテーブルには、プロビジョニングされた最大書き込みキャパシティの設定はありませんが、テーブルレベルの最大書き込みスループット制限により、オンデマンドテーブルが許可する最大持続書き込みキャパシティが指定されます。デフォルトの制限は 40,000 ですが、調整可能です。オンデマンドテーブルに必要となるすべての書き込みオペレーション (TTL 書き込みオペレーションを含む) を処理できるように、この値を十分に高く設定することをお勧めします。グローバルテーブルを設定する場合、この値は参加しているすべてのリージョンで同じでなければなりません。

読み取りキャパシティの管理に関する考慮事項は以下のとおりです。

- 読み取りキャパシティの管理に関する設定は、リージョンごとに読み取りパターンが異なることが想定されているため、リージョン間で異なることが許容されます。グローバルレプリカを初めてテーブルに追加すると、ソースリージョンのキャパシティが反映されます。作成後、読み取りキャパシティの設定は調整できますが、この調整した設定は反対側には転送されません。
- DynamoDB 自動スケーリングを使用するときは、プロビジョニングされた最大読み取りキャパシティの設定が、すべてのリージョンのすべての読み取りオペレーションを処理するのに十分な大きさであることを確認してください。標準的なオペレーション時には、読み取りキャパシティがリージョン全体に分散される可能性がありますが、フェイルオーバー時には、テーブルが読み取りワークロードの増大に自動的に対応できなければなりません。オンデマンドテーブルにはプロビジョニングされた最大読み取りキャパシティの設定はありませんが、テーブルレベルの最大読み取りスループット制限により、オンデマンドテーブルが許容する最大持続読み取りキャパシティが指定されます。デフォルトの制限は 40,000 ですが、調整可能です。すべての読み取りオペレーションをこの単一のリージョンにルーティングする場合、テーブルが必要とするすべての読み取りオペレーションを処理できるように、この値を十分に高く設定することをお勧めします。
- 1 つのリージョン内にあるテーブルで、通常は読み取りトラフィックを受信しないが、フェイルオーバー後に大量の読み取りトラフィックを受信する必要がある場合は、さらに大量の読み取りトラフィックを受信可能となるようにキャパシティを事前ウォーミングできます。

ARC には、準備状況チェック機能があり、Route 53 を使用してリクエストをルーティングするかどうかにかかわらず、DynamoDB リージョン間でテーブル設定とアカウントクォータが同じであることを確認するのに役立ちます。準備状況チェックは、アカウントレベルのクォータを調整して相互に一致させる場合にも役立ちます。

DynamoDB グローバルテーブルの準備チェックリスト

グローバルテーブルをデプロイするときの決定事項とタスクには、次のチェックリストを使用してください。

- 対象のユースケースの場合、MRSC と、MREC の整合性モードでは、どちらの利点が大きいかを判断します。レイテンシーなどのトレードオフが大きい場合でも、強力な整合性が必要ですか?
- グローバルテーブルに参加するリージョンの数と各リージョンを決定します。MRSC の使用を計画している場合は、3 番目のリージョンをレプリカにするか、監視にするかを決定します。
- アプリケーションの書き込みモードを決定します。これは整合性モードとは異なります。詳細については、「DynamoDB グローバルテーブルによる書き込みモード」を参照してください。
- 書き込みモードに基づいて「DynamoDB のルーティング戦略」戦略を計画します。
- 整合性モード、書き込みモード、ルーティング戦略に基づいて、

リージョンの退避とは、アクティビティ (通常は読み取りおよび書き込みアクティビティだが、読み取りアクティビティの場合もある) をそのリージョンから移行するプロセスを意味します。

ライブリージョンからの退避

ライブリージョンからの退避の決定には、さまざまな理由が考えられます。例えば、通常のビジネスアクティビティの一部である (例: 1 つのリージョンへの書き込みモードを使い、フォローザサン方式を取っている)、現在アクティブなリージョンをビジネス上変更する、DynamoDB 外で生じたソフトウェアスタック障害に対応する、リージョン内のレイテンシーが通常よりも増加したといった一般的な問題が発生している、などです。

任意のリージョンへの書き込みモードを使用すると、ライブリージョンから簡単に退避できます。任意のルーティングシステムを使用して、トラフィックを代替リージョンにルーティングでき、退避したリージョンでの書き込みオペレーションを通常どおりレプリケートすることが可能です。

1 つのリージョンへの書き込みモードと、自分のリージョンへの書き込みモードは、一般的に、MREC テーブルで使用されます。したがって、新しいアクティブリージョンで書き込みオペレーションを開始する前に、アクティブリージョンへのすべての書き込みオペレーションが完全に記録、ストリーム処理、グローバルに伝播されたことを確認し、その後は、最新バージョンのデータに書き込みオペレーションが行われるようにする必要があります。

例えば、リージョン A がアクティブ、リージョン B がパッシブだとします (リージョン A をホームとするテーブル全体または項目のいずれかが対象)。退避を実行する一般的なメカニズムは、A への書き込みオペレーションを一時停止し、そのオペレーションが B に完全に伝播されるまで待ち、アーキテクチャスタックを更新して B がアクティブであることを認識してから、B への書き込みオペレーションを再開することです。リージョン A がリージョン B にデータを完全にレプリケートしたことを確実に示すメトリクスはありません。リージョン A が正常であれば、リージョン A への書き込みオペレーションを一時停止し、`ReplicationLatency` メトリクスの最新の最大値の 10 倍を待てば、通常はレプリケーションが完了したことを十分に確認できます。リージョン A に異常があり、他の領域でのレイテンシーの増加も示している場合は、待機時間の倍数を大きくします。

オフラインリージョンからの退避

考慮すべき特別なケースがあります。リージョン A が予告なしに完全にオフラインになった場合、どのような影響が生じるでしょうか? 非常にまれなケースですが、考慮する必要があります。

オフライン MRSC テーブルの退避

そのような状況が MRSC テーブルで発生した場合、特別な操作は必要ありません。MRSC グローバルテーブルでは、目標復旧時点 (RPO) ゼロがサポートされています。オフラインリージョンの MRSC テーブルに実行され成功したすべての書き込みオペレーションは、他のすべてのリージョンテーブルに適用されるため、リージョンが予告なしに完全にオフラインになっても、データに差が生じる可能性はありません。他のリージョンにあるレプリカを業務に引き続き使用できます。

オフライン MREC テーブルの退避

そのような状況が MREC テーブルで発生した場合、リージョン A でまだ伝播されていない書き込みオペレーションはすべて保持され、リージョン A がオンラインに戻った後に伝播されます。書き込みオペレーションは失われませんが、その伝播は無期限に遅延します。

このイベントの処理方法は、アプリケーションの決定によります。ビジネスの継続性を確保するために、書き込みオペレーションを新しいプライマリリージョン B に移行することが必要になる場合があります。ただし、リージョン A の項目に対する書き込みオペレーションの伝播が保留されているときに、リージョン B でその項目が更新を受け取ると、最終書き込み者優先モデルに従ってその伝播は抑制されます。リージョン B での更新は、着信する書き込みリクエストを抑制する可能性があります。

任意のリージョンへの書き込みモードでは、リージョン A の項目が最終的にリージョン B に伝播されることを信頼し、リージョン A がオンラインに戻るまで項目が欠落する可能性を認識しながら、リージョン B で読み取りおよび書き込みオペレーションを続行できます。可能であれば、べき等書き込みオペレーションなどでは、最近の書き込みトラフィックを (アップストリームのイベントソースを使用するなどして) 再生することを検討してください。これにより、欠落している可能性のある書き込みオペレーションのギャップを埋め、最終書き込み者優先の競合解決により、着信する書き込みオペレーションの最終的な伝播を抑制します。

他の書き込みモードでは、どの程度の作業を継続できるかを、少し時代遅れの方法で考慮する必要があります。リージョン A がオンラインに戻るまで、`ReplicationLatency` で追跡する書き込みオペレーションの一部の短い期間は失われます。ビジネスを継続できるでしょうか。一部のユースケースでは可能ですが、他のユースケースでは追加の緩和メカニズムがないと難しい場合があります。

例えば、あるリージョン全体で障害が発生した後でも、利用可能なクレジット残高を中断することなく維持する必要があるとします。残高を 2 つの異なる項目に分割し、1 つはリー

ジョン A に、もう 1 つはリージョン B に置き、それぞれ利用可能な残高を半分にして開始できます。この場合は、自分のリージョンへの書き込みモードを使用します。各リージョンで処理されたトランザクションの更新は、残高のローカルコピーに対して書き込まれます。リージョン A が完全にオフラインになっても、リージョン B でトランザクション処理を続行でき、書き込みオペレーションはリージョン B に保持されている残高部分に制限されます。このように残高を分割すると、残高が少なくなった場合やクレジットの再調整が必要になった場合に複雑になりますが、保留中の書き込みオペレーションが不安定でも安全にビジネスを回復できる一例となります。

別の例として、ウェブフォームのデータをキャプチャするとします。オプティミスティック同時実行制御 (OCC) (OCC) を使用してデータ項目にバージョンを割り当て、最新バージョンを非表示フィールドとしてウェブフォームに埋め込むことができます。送信するたびに、データベース内のバージョンがフォーム作成時のバージョンとまだ一致する場合にのみ、書き込みオペレーションが成功します。バージョンが一致しない場合、データベース内の現在のバージョンに基づいてウェブフォームを更新 (または慎重にマージ) することで、ユーザーは再度続行できます。OCC モデルは通常、別のクライアントがデータを上書きして新しいバージョンを生成するのを防ぎますが、フェイルオーバー中にクライアントが古いバージョンのデータに遭遇する可能性がある場合にも役立ちます。タイムスタンプをバージョンとして使用しているとします。フォームを当初 12:00 にリージョン A に対して作成しましたが、(フェイルオーバー後に) リージョン B に書き込もうとして、データベースの最新バージョンが 11:59 であることに気付いたとします。このシナリオの場合、クライアントは 12:00 バージョンがリージョン B に伝播されるのを待ってからそのバージョンに書き込むか、11:59 バージョンに基づいて構築し、新しい 12:01 バージョンを作成することができます (書き込み後、リージョン A の回復後に着信したバージョンは抑制されます)。

3 つ目の例として、金融サービス会社が顧客アカウントおよび金融取引に関するデータを DynamoDB データベースに保持しているとします。同社は、リージョン A が完全に停止した場合、アカウントに関連する書き込みアクティビティのすべてがリージョン B で完全に利用可能であることを確認するか、リージョン A がオンラインに戻るまでアカウントを部分的なものとして隔離したいと考えていました。すべての業務を一時停止するのではなく、伝播されていないトランザクションがあると判断したごく一部のアカウントに対してのみ業務を一時停止することにしました。これを実現するために、同社は 3 番目のリージョン (リージョン C と呼びます) を使用しました。リージョン A で書き込みオペレーションを処理する前に、保留中のオペレーション (アカウントの新規トランザクション数など) の簡潔な要約をリージョン C に配置しました。この要約は、リージョン B のビューが完全に最新であるかどうかを判断するのに十分でした。このアクションにより、リージョン C での書き込み時点から、リージョン A が書き込みオペレーションを受け入れてリージョン B がそれを受け取る

> までの間、アカウントは事実上ロックされました。リージョン C のデータは、フェイルオーバープロセスの一部として以外は使用されませんでした。その後、リージョン B はリージョン C とデータを相互チェックして、古いアカウントがないかどうかを確認できました。これらのアカウントは、リージョン A リカバリによって部分的なデータがリージョン B に伝播されるまで隔離済みとしてマークされます。リージョン C に障害が発生した場合は、代わりに新しいリージョン D をスピンアップして使用できます。リージョン C のデータはごく一時的なものので、数分後には処理中の書き込みオペレーションの最新の記録がリージョン D に反映されて十分に役立つようになります。リージョン B に障害が発生しても、リージョン A はリージョン C と協力して書き込みリクエストを引き続き受け入れることができます。同社は、より高いレイテンシーの書き込み (C に続けて A という 2 つのリージョンへの書き込み) をあえて受け入れ、アカウントの状態を簡潔に要約できるデータモデルを持つことができたことを喜んでいます。

the section called “退避プロセス” を定義します。

- リージョンごとのヘルス、レイテンシー、エラーに関するメトリクスをキャプチャします。監視すべき DynamoDB メトリクスのリストについては、AWS Database Blog で「Monitoring Amazon DynamoDB for Operational Awareness」を参照してください。また、合成 canary (炭鉱のカナリアにちなんで名付けられた、障害を検出するための人工的なリクエスト) の使用や、顧客トラフィックのライブ観察も必要です。すべての問題が DynamoDB メトリクスに反映されるわけではありません。
- MREC を使用している場合は、`ReplicationLatency` に継続的な増加が見られた際のアラームを設定します。この増加は、グローバルテーブルの書き込み設定がリージョンごとに異なるという偶発的な設定ミスを示している可能性があり、その結果、レプリケートされたリクエストが失敗し、レイテンシーが高くなります。また、リージョンに混乱が生じていることを示している可能性もあります。良い例としては、最近の平均が 180,000 ミリ秒を超えた場合にアラートを生成することが挙げられます。また、`ReplicationLatency` が 0 に下がり、レプリケーションの停止状態を示していることを監視することもできます。
- 各グローバルテーブルに十分な最大読み取り/書き込み設定を割り当てます。
- リージョンから退避する理由を事前に特定します。決定に人間の判断が関与する場合は、すべての考慮事項を文書化します。この作業は、必要に迫られて行うのではなく、事前に慎重に行う必要があります。
- リージョンから退避する際に実行する必要があるすべてのアクションのランブックを用意しておきます。通常、グローバルテーブルに関係する作業はほとんどありませんが、スタックの残りの部分を移動するのは複雑な作業になる場合があります。

> **Note**
>
> フェイルオーバー手順のベストプラクティスを実装するには、コントロールプレーンでのオペレーションではなくデータプレーンでのオペレーションにのみ依存すると良いでしょう。リージョンでの障害発生中には、一部のコントロールプレーンオペレーションのパフォーマンスが低下する可能性があるからです。

詳細については、AWS ブログ記事「Amazon DynamoDB グローバルテーブルを使用した回復性の高いアプリケーションの構築: パート 4」を参照してください。

- リージョンの退避を含め、ランブックのあらゆる側面を定期的にテストします。テストしないと、ランブックの信頼性が低下します。
- AWS Resilience Hub を使用して、アプリケーション全体 (グローバルテーブルを含む) のレジリエンスを評価することを検討します。ダッシュボードを通じて、アプリケーションポートフォリオ全体のレジリエンスステータスを包括的に確認できます。
- ARC の準備状況チェックを使用して、アプリケーションの現在の設定を評価し、ベストプラクティスから逸脱していないか追跡することを検討します。
- Route 53 または Global Accelerator で使用するヘルスチェックを記述する際には、データベースフロー全体をカバーする一連の呼び出しを行います。チェックを制限し、DynamoDB エンドポイントが起動していることを確認するだけでは、AWS Identity and Access Management (IAM) 設定エラー、コードデプロイの問題、DynamoDB 外のスタックでの障害、平均読み取りまたは書き込みレイテンシーを超える状況といった、障害のさまざまな状態をカバーできません。

グローバルテーブルのデプロイに関するよくある質問 (FAQ)

グローバルテーブルの料金はどのように設定されていますか?

グローバルテーブルの料金は、レプリケートされた書き込みユニット、ストレージ、読み取りなど、各レプリカがそれぞれのリージョンで消費するリソースに基づいて設定されます。グローバルテーブルの料金の詳細については、「グローバルテーブルの料金」を参照してください。

グローバルテーブルはどのリージョンをサポートしていますか?

グローバルテーブルバージョン 2019.11.21 (現行) では、MREC テーブルの場合すべての AWS リージョンがサポートされています。

MRSC グローバルテーブルは、米国東部 (バージニア北部)、米国東部 (オハイオ)、米国西部 (オレゴン)、カナダ (中部)、欧州 (アイルランド)、欧州 (ロンドン)、欧州 (パリ)、欧州 (フランクフルト)、欧州 (ストックホルム)、欧州 (スペイン)、アジアパシフィック (東京)、アジアパシフィック (ソウル)、アジアパシフィック (大阪)、アジアパシフィック (ムンバイ)、およびアジアパシフィック (シンガポール)の各 AWS リージョンで利用できます。MRSC グローバルテーブルには、異なる大陸のリージョンを含め、これらの中から任意の 3 つのリージョンを組み合わせたレプリカを含めることができます。

GSI はグローバルテーブルでどのように処理されますか?

グローバルテーブルバージョン 2019.11.21 (現行) では、あるリージョンで GSI を作成すると、他の参加リージョンでも自動的に作成され、自動的にバックフィルされます。

グローバルテーブルのレプリケーションを停止するにはどうしたらいいですか?

- レプリカテーブルは、他のテーブルを削除するのと同じ方法で削除できます。グローバルテーブルを削除すると、そのリージョンへのレプリケーションが停止し、そのリージョンに保持されているテーブルのコピーが削除されます。ただし、テーブルのコピーを独立したエンティティとして保持している間は、レプリケーションを停止または一時停止することはできません。
- MRSC テーブルは、厳密に 3 つのリージョンにデプロイする必要があります。レプリカを削除するには、MRSC テーブルがローカルテーブルとなるように、すべてのレプリカと監視を削除する必要があります。

DynamoDB Streams はグローバルテーブルとどのようにやり取りするのですか?

- 各グローバルテーブルは、書き込み元に関係なく、すべての書き込みオペレーションに基づいて独立したストリームを生成します。DynamoDB ストリームを 1 つのリージョンで使用するか、すべてのリージョンで (個別に) 使用するかを選択できます。レプリケートされた書き込みオペレーションではなく、ローカル書き込みオペレーションを処理する場合は、各項目に独自のリージョン属性を追加して、書き込みリージョンを識別できます。次に、Lambda イベントフィルターを使用して、ローカルリージョンでの書き込みオペレーションに対してのみ Lambda 関数を呼び出すことができます。これは挿入および更新オペレーションには役立ちますが、削除オペレーションには役立ちません。
- マルチリージョンの結果整合性 (MREC テーブル) を設定したグローバルテーブルでは、レプリカテーブルの DynamoDB ストリームから変更を読み取り、それを他のすべてのレプリカテーブルに適用して、変更をレプリケートします。したがって、デフォルトの場合 DynamoDB ストリームは、MREC グローバルテーブル内のすべてのレプリカで有効になっており、こうしたレプリカで無効にすることはできません。MREC のレプリケーションプロセスでは、短時間に発生した複数

の変更を 1 度の書き込みオペレーションとして結合し、レプリケートを行えます。その結果、各レプリカのストリームには、わずかに異なるレコードが含まれている可能性があります。MREC レプリカの DynamoDB Streams レコードは、常に項目ごとに順序付けられますが、項目間の順序はレプリカによって異なる場合があります。

- マルチリージョンの強力な整合性を設定したグローバルテーブル (MRSC テーブル) では、レプリケーションに DynamoDB Streams が使用されないため、デフォルトの場合、MRSC レプリカではこの機能は有効になっていません。しかし、設定により、MRSC レプリカで DynamoDB Streams を有効にすることができます。MRSC レプリカの DynamoDB Streams レコードはすべてのレプリカで同一であり、常に項目ごとに順序付けされますが、項目間の順序付けはレプリカ間で異なる場合があります。

グローバルテーブルはトランザクションをどのように処理するのですか?

- MRSC テーブルにトランザクションオペレーションを行うと、エラーが返ります。
- MREC テーブルでのトランザクションオペレーションでは、書き込みオペレーションが最初に発生したリージョン内でのみ、アトミック性、整合性、分離性、耐久性 (ACID) が保証されます。グローバルテーブルのリージョン間では、トランザクションはサポートされていません。例えば、米国東部 (オハイオ) および米国西部 (オレゴン) リージョンにレプリカを持つ MREC グローバルテーブルがあり、米国東部 (オハイオ) リージョンで `TransactWriteItems` オペレーションを実行すると、変更がレプリケートされたときに米国西部 (オレゴン) では部分的に完了したトランザクションが観察されることがあります。変更は、ソースリージョンでコミットされた後でのみ、他のリージョンにレプリケートされます。

グローバルテーブルは DynamoDB Accelerator キャッシュ (DAX) とどのようにやり取りするのですか?

グローバルテーブルは DynamoDB を直接更新することで DAX をバイパスするため、DAX はそれが古いデータを保持していることを認識しません。DAX キャッシュは、キャッシュの TTL が期限切れになったときにのみ更新されます。

テーブルのタグは伝播されますか?

いいえ、タグは自動的には伝播されません。

すべてのリージョンのテーブルをバックアップすべきですか? 1 つのリージョンのテーブルのみをバックアップすべきですか?

答えはバックアップの目的によって異なります。

- データの耐久性を確保したい場合、DynamoDB には既に保護手段が用意されています。このサービスにより、耐久性が保証されます。
- 履歴記録のスナップショットを保持する場合 (規制要件を満たすためなど)、1 つのリージョンでバックアップすれば十分です。AWS Backup を使用してバックアップを別のリージョンにコピーできます。
- 誤って削除または変更したデータを復元する場合は、1 つのリージョンで DynamoDB ポイントインタイムリカバリ (PITR) を使用します。

を使用してグローバルテーブルをデプロイするにはどうすればいいですか?CloudFormation

- CloudFormation は、DynamoDB テーブルとグローバルテーブルを 2 つの独立したリソース (`AWS::DynamoDB::Table` と `AWS::DynamoDB::GlobalTable`) として表します。1 つの方法としては、`GlobalTable` コンストラクトを使用してグローバルになる可能性のあるすべてのテーブルを作成します。これらのテーブルは、最初はスタンドアロンテーブルのままにし、必要に応じて後でリージョンを追加します。
- CloudFormation では、レプリカの数に関係なく、各グローバルテーブルは単一リージョン内の単一スタックによって制御されます。テンプレートをデプロイすると、CloudFormation はすべてのレプリカを単一スタックのオペレーションの一部として作成および更新します。同じ AWS::DynamoDB::GlobalTable リソースを複数のリージョンにデプロイしないでください。これは、エラーとなるため、サポートされていません アプリケーションテンプレートを複数のリージョンにデプロイする場合は、条件を使用して単一リージョンに `AWS::DynamoDB::GlobalTable` リソースを作成できます。または、アプリケーションスタックとは別のスタック内に `AWS::DynamoDB::GlobalTable` リソースを定義し、それが単一リージョンにのみデプロイされるようにします。
- 通常のテーブルがあり、それを CloudFormation で管理したままグローバルテーブルに変換する場合は、次のようにします。削除ポリシーを `Retain` に設定して、テーブルをスタックから削除し、コンソールでテーブルをグローバルテーブルに変換します。次に、グローバルテーブルを新しいリソースとしてスタックにインポートします。詳細については、「AWS GitHub リポジトリ」を参照してください。
- CloudFormation (および AWS CDK) を使用して、マルチアカウント (クロスアカウント) グローバルテーブルを設定できます。詳細については、「DynamoDB マルチアカウントグローバルテーブル」を参照してください。

結論とリソース

DynamoDB グローバルテーブルには、制御上の項目はほとんどありませんが、慎重な検討は必要です。書き込みモード、ルーティングモデル、および退避プロセスを決定する必要があります。すべてのリージョンにわたってアプリケーションをインストルメントし、グローバルヘルスを維持するためのルーティングの調整や退避に備える必要があります。その見返りとして、グローバルに分散させたデータセットで低レイテンシーの読み取りおよび書き込みオペレーションを行える上、99.999% の可用性も確保できるのです。

DynamoDB グローバルテーブルの詳細については、次のリソースを参照してください。

- グローバルテーブル - マルチアクティブ、マルチリージョンレプリケーション
- Amazon Application Recovery Controller
- Readiness check in ARC (AWS ドキュメント)
- Route 53 ルーティングポリシー
- AWS Global Accelerator
- DynamoDB サービスレベルアグリーメント
- AWS マルチリージョンの基礎 (AWS ホワイトペーパー)
- Data resiliency design patterns with AWS (AWS re:Invent 2022 プレゼンテーション)
- How Fidelity Investments and Reltio modernized with Amazon DynamoDB (AWS re:Invent 2022 プレゼンテーション)
- Multi-Region design patterns and best practices (AWS re:Invent 2022 プレゼンテーション)
- Disaster Recovery (DR) Architecture on AWS, Part III: Pilot Light and Warm Standby (AWS ブログ記事)
- リージョンのピン留めを使用して Amazon DynamoDB グローバルテーブル内の項目のホームリージョンを設定する (AWS ブログ記事)
- Amazon DynamoDB をモニタリングして運用状況を把握する (AWS ブログ記事)
- DynamoDB のスケーリング: パーティション、ホットキー、ヒートに応じた分割がパフォーマンスに与える影響 (AWS ブログ記事)
- DynamoDB グローバルテーブルを使用したマルチリージョンの強力な整合性 (AWS re:Invent 2024 プレゼンテーション)

DynamoDB でコントロールプレーンを管理するためのベストプラクティス

> **Note**
>
> DynamoDB では、1 秒あたり 2,500 リクエストというコントロールプレーンのスロットル制限を導入しており、再試行のオプションもあります。詳細については、以下を参照してください。

DynamoDB コントロールプレーンオペレーションでは、DynamoDB テーブルだけでなく、インデックスなどのテーブルに依存するオブジェクトも管理できます。これらのオペレーションの詳細については、コントロールプレーン を参照してください。

状況によっては、アクションを実行し、コントロールプレーンの呼び出しによって返されたデータをビジネスロジックの一部として使用する必要がある場合があります。例えば、`DescribeTable` によって返される `ProvisionedThroughput` の値を知っておく必要がある場合があります。このような場合は、次のベストプラクティスに従ってください。

- DynamoDB コントロールプレーンに過度にクエリを実行しないでください。
- 同じコード内でコントロールプレーンの呼び出しとデータプレーンの呼び出しを混在させないでください。
- コントロールプレーンのリクエストでスロットリングを処理し、バックオフして再試行します。
- 1 つのクライアントから特定のリソースを呼び出して変更を追跡します。
- 同じテーブルのデータを短い間隔で複数回取得する代わりに、データをキャッシュして処理します。

DynamoDB での一括データオペレーション使用についてのベストプラクティス

DynamoDB は、最大 25 の `PutItem` リクエストと `DeleteItem` リクエストを同時に実行できる、`BatchWriteItem` などのバッチオペレーションをサポートしています。ただし、`BatchWriteItem` は、`UpdateItem` オペレーションをサポートしていません。一括更新の場合、更新の要件と内容によって異なります。最大 100 のバッチサイズには、`TransactWriteItems` などの他の DynamoDB API を使用できます。これ以上の項目が

含まれる場合は、AWS Glue、Amazon EMR、AWS Step Functions などのサービスを使用するか、DynamoDB シェルなどのカスタムスクリプトやツール、またはオープンソースの Bulk Executor for DynamoDB を使用して一括更新できます。

トピック

- 条件付きバッチ更新
- 効率的な一括オペレーション

条件付きバッチ更新

DynamoDB は、最大 25 の `PutItem` リクエストと `DeleteItem` リクエストを単一のバッチで実行できる、`BatchWriteItem` などのバッチオペレーションをサポートしています。ただし、`BatchWriteItem` は `UpdateItem` オペレーションはサポートしておらず、条件式もサポートしていません。この回避策として、最大 100 のバッチサイズには、`TransactWriteItems` などのその他の DynamoDB API を使用できます。

これ以上の項目が含まれており、大量のデータの変更が必要な場合は、AWS Glue、Amazon EMR AWS Step Functions などのサービスを使用するか、DynamoDB シェルなどのカスタムスクリプトやツールを使用すると、効率的に一括更新できます。

このパターンを使用すべきケース

- DynamoDB シェルは、本番稼働のユースケースではサポートされていません。
- `TransactWriteItems` – 条件付きまたは条件なしの最大 100 の個別の更新。all-or-nothing の ACID バンドルとして実行されます。アプリケーションでべき等性が必要な場合、つまり複数の同一の呼び出しが 1 回の呼び出しと同じ効果を持つような場合は、`TransactWriteItems` コールに `ClientRequestToken` を指定することもできます。これにより、同じトランザクションを複数回実行してもデータの状態が正しくないという状況を避けることができます。

  トレードオフ – 追加のスループットが使用されます。標準の 1 KB の書き込みあたり 1 WGU ではなく、1 KB の書き込みあたり 2 WCU になります。
- PartiQL `BatchExecuteStatement` – 条件付きまたは条件なしで、最大 25 件の更新。`BatchExecuteStatement` は常にリクエスト全体に成功応答を返し、順序を維持する個別のオペレーション応答のリストも返します。

  トレードオフ – バッチが大きい場合、リクエストを 25 のバッチで分散するために、クライアント側で追加のロジックが必要です。再試行戦略を決定するには、個別のエラーレスポンスを考慮する必要があります。

コードの例

これらのコード例では、AWS SDK for Python である boto3 ライブラリを使用しています。これらの例では、boto3 がインストール済みで、適切な AWS 認証情報で設定されていることを前提としています。

ヨーロッパの都市に複数の倉庫がある電気機器ベンダーの在庫データベースがあるとします。現在は夏の終わりであるため、ベンダーはデスク用ファンを処分して他の製品の在庫スペースを確保したいと考えています。ベンダーは、イタリアの倉庫から供給されたすべてのデスク用ファンについて、デスク用ファンの予備在庫が 20 ある場合に限り、価格の割引を提供することを検討しています。DynamoDB テーブル名は、inventory であり、このテーブルには、各製品の一意の識別子であるパーティションキー sku のキースキーマと、倉庫の識別子であるソートキー warehouse があります。

以下の Python コードは、`BatchExecuteStatement` API コールを使用して、この条件付きバッチ更新を実行する方法を説明しています。

```
import boto3

client=boto3.client("dynamodb")

before_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("Before update: ", before_image['Items'])

response=client.batch_execute_statement(
        Statements=[
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITTUR1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM2'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITROM5'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
```

```
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN1'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN2'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
            {'Statement': 'UPDATE inventory SET price=price-5 WHERE sku=? AND
 warehouse=? AND quantity > 20', 'Parameters': [{'S':'F123'}, {'S':'WITVEN3'}],
 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'},
        ],
        ReturnConsumedCapacity='TOTAL'
    )

after_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("After update: ", after_image['Items'])
```

実行すると、サンプルデータに以下の出力が生成されます。

```
Before update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price':
 {'N': '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S':
 'WITROM2'}, 'price': {'N': '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '28'}, 'warehouse': {'S': 'WITROM5'}, 'price': {'N': '38'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '40'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price':
 {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S':
 'WITVEN2'}, 'price': {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'},
 'warehouse': {'S': 'WITVEN3'}, 'price': {'N': '35'}, 'sku': {'S': 'F123'}}]
After update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price': {'N':
 '40'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S': 'WITROM2'},
 'price': {'N': '35'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '28'}, 'warehouse':
 {'S': 'WITROM5'}, 'price': {'N': '33'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '35'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price': {'N': '38'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S': 'WITVEN2'}, 'price':
 {'N': '38'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'}, 'warehouse': {'S':
 'WITVEN3'}, 'price': {'N': '30'}, 'sku': {'S': 'F123'}}]
```

これは内部システムの境界オペレーションであるため、べき等性の要件は考慮されていません。価格更新のような追加のガードレールを配置できるのは、価格が 35 より大きく 40 より小さい場合のみとするなど、更新をより堅牢にすることができます。

または、べき等性や ACID 要件をより厳密にする要件の場合は、TransactWriteItems を使用して同じバッチ更新オペレーションを実行できます。ただし、トランザクションバンドル内のすべてのオペレーションが完了するか、バンドル全体が失敗する、という点に注意する必要があります。

イタリアで熱波が発生し、デスク用ファンの需要が急増したと仮定します。ベンダーは、イタリアのすべての倉庫から出荷されるデスク用ファンのコストを 20 ユーロ引き上げたいと考えていますが、規制機関がこのコスト上昇を許可するのは、現在のコストが在庫全体で 70 ユーロ未満の場合にのみです。在庫全体で価格が一度に一度のみ更新され、更新されるのは各倉庫のコストが 70 ユーロ未満の場合にのみである天が重要です。

次の Python コードは、TransactWriteItems API コールを使用して、このバッチ更新を実行する方法を説明しています。

```
import boto3

client=boto3.client("dynamodb")

before_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("Before update: ", before_image['Items'])

response=client.transact_write_items(
        ClientRequestToken='UUIDAWS124',
        TransactItems=[
            {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITTUR1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
            {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
            {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM2'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
```

```
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITROM5'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN1'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN2'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
           {'Update': { 'Key': {'sku': {'S':'F123'}, 'warehouse': {'S':'WITVEN3'}},
 'UpdateExpression': 'SET price = price + :inc', 'ConditionExpression': 'price < :cap',
 'ExpressionAttributeValues': { ':inc': {'N': '20'}, ':cap': {'N': '70'}}, 'TableName':
 'inventory', 'ReturnValuesOnConditionCheckFailure': 'ALL_OLD'}},
       ],
       ReturnConsumedCapacity='TOTAL'
   )

after_image=client.query(TableName='inventory', KeyConditionExpression='sku=:pk_val
 AND begins_with(warehouse, :sk_val)', ExpressionAttributeValues={':pk_val':
{'S':'F123'},':sk_val':{'S':'WIT'}},
 ProjectionExpression='sku,warehouse,quantity,price')
print("After update: ", after_image['Items'])
```

実行すると、サンプルデータに以下の出力が生成されます。

```
Before update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price':
 {'N': '60'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S':
 'WITROM2'}, 'price': {'N': '55'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
 '28'}, 'warehouse': {'S': 'WITROM5'}, 'price': {'N': '53'}, 'sku': {'S': 'F123'}},
 {'quantity': {'N': '26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '55'}, 'sku':
 {'S': 'F123'}}, {'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price':
 {'N': '58'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S':
 'WITVEN2'}, 'price': {'N': '58'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'},
 'warehouse': {'S': 'WITVEN3'}, 'price': {'N': '50'}, 'sku': {'S': 'F123'}}]
After update:  [{'quantity': {'N': '20'}, 'warehouse': {'S': 'WITROM1'}, 'price': {'N':
 '80'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '25'}, 'warehouse': {'S': 'WITROM2'},
 'price': {'N': '75'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '28'}, 'warehouse':
```

```
{'S': 'WITROM5'}, 'price': {'N': '73'}, 'sku': {'S': 'F123'}}, {'quantity': {'N':
'26'}, 'warehouse': {'S': 'WITTUR1'}, 'price': {'N': '75'}, 'sku': {'S': 'F123'}},
{'quantity': {'N': '10'}, 'warehouse': {'S': 'WITVEN1'}, 'price': {'N': '78'}, 'sku':
{'S': 'F123'}}, {'quantity': {'N': '20'}, 'warehouse': {'S': 'WITVEN2'}, 'price':
{'N': '78'}, 'sku': {'S': 'F123'}}, {'quantity': {'N': '50'}, 'warehouse': {'S':
'WITVEN3'}, 'price': {'N': '70'}, 'sku': {'S': 'F123'}}]
```

DynamoDB でバッチ更新を実行する方法は、複数あります。適切なアプローチは、ACID やべき等性の要件、更新する項目数、API の知識などの要因によって異なります。

効率的な一括オペレーション

このパターンを使用すべきケース

これらのパターンは、DynamoDB 項目の一括更新を効率的に実行するうえで役立ちます。

- DynamoDB シェルは、本番稼働のユースケースではサポートされていません。
- `TransactWriteItems` – 条件付きまたは条件なしの最大 100 の個別の更新。all-or-nothing の ACID バンドルとして実行されます。

  トレードオフ — 追加のスループットが使用され、1 KB の書き込みあたり 2 WCU が使用されます。
- PartiQL `BatchExecuteStatement` – 条件付きまたは条件なしで、最大 25 件の更新。

  トレードオフ – リクエストを 25 のバッチで分散するために、追加のロジックが必要です。
- AWS Step Functions – に精通しているデベロッパー向けのレート制限付き一括オペレーション。AWS Lambda

  トレードオフ — 実行時間はレート制限に反比例します。Lambda 関数の最長タイムアウトによって制限されます。この機能では、読み取りと書き込みの間に発生するデータ変更が上書きされる可能性があります。詳細については、「Backfilling an Amazon DynamoDB Time to Live attribute using Amazon EMR: Part 2」を参照してください。
- AWS Glue および Amazon EMR – マネージド並列処理によるレート制限付き一括オペレーション。時間的制約のないアプリケーションや更新の場合、これらのオプションはスループットのごく一部しか使用せずにバックグラウンドで実行できます。どちらのサービスも、emr-dynamodb-connector を使用して DynamoDB オペレーションを実行します。これらのサービスは、大規模な読み取りを実行した後、レート制限のオプションを使用して更新済みの項目を大量に書き込みます。

トレードオフ — 実行時間はレート制限に反比例します。この機能では、読み取りと書き込みの間に発生するデータ変更が上書きされる可能性があります。グローバルセカンダリインデックス (GSI) からの読み取りはできません。詳細については、「Backfilling an Amazon DynamoDB Time to Live attribute using Amazon EMR: Part 2」を参照してください。

- DynamoDB シェル – SQL のようなクエリを使用したレート制限付き一括オペレーション。効率を向上させるために GSI から読み取ることができます。

  トレードオフ — 実行時間はレート制限に反比例します。「Rate limited bulk operations in DynamoDB Shell」を参照してください。

- Bulk Executor for DynamoDB – カウント、検索、削除、更新、コピー、フィル、ロード、差分などの一括タスクを並列実行するためにバックエンドで AWS Glue を使用するオープンソースのコマンドラインツール。組み込みコマンドにコーディングは不要ですが、Python を使用してカスタムコマンドで拡張することも可能です。レート制限をサポートし、実行前にコストの見積もりを提供します。Bulk Executor はオープンソースであり、公式サポートは含まれていません。

  トレードオフ – AWS Glue 環境 (ブートストラップ時に作成) が必要です。実行時間はレート制限に反比例します。カスタムコマンドには Python が必要です。「Introducing open source Bulk Executor for Amazon DynamoDB」を参照してください。

パターンの使用

オンデマンドスループットモードを使用する場合、特に一括更新については、コストに多大な影響が及ぶ可能性があります。プロビジョンドスループットモードを使用する場合、速度とコストの間にトレードオフが発生します。レート制限パラメータを厳密に設定すると、処理時間が非常に長くなる可能性があります。平均項目サイズとレート制限を使用すると、更新の速度を大まかに判断できます。

別の方法として、更新プロセスの予想期間と平均項目サイズに基づいて、プロセスに必要なスループットの量を決定することもできます。各パターンで共有されているブログのリファレンスには、パターン使用の戦略、実装、制限に関する詳細が記載されています。詳細については、「Cost-effective bulk processing with Amazon DynamoDB」を参照してください。

ライブ DynamoDB テーブルに対して一括更新を実行するには、複数の方法があります。適切なアプローチは、ACID やべき等性の要件、更新する項目数、API の知識などの要因によって異なります。コストと時間の間のトレードオフを考慮することが重要となります。上記のほとんどのアプローチでは、一括更新ジョブで使用されるスループットをレート制限するオプションが提供されています。

DynamoDB で同時更新を処理するためのベストプラクティス

分散システムでは、複数のプロセスまたはユーザーが同時に同じデータを変更しようとすることがあります。同時実行制御がない場合、これらの同時書き込みにより、更新の損失、データの不整合、競合状態が発生する可能性があります。DynamoDB には、同時アクセスの管理とデータ整合性の維持に役立つメカニズムがいくつか用意されています。

> **Note**
>
> `UpdateItem` などの個々の書き込みオペレーションはアトミックであり、同時実行性に関係なく、常に項目の最新バージョンに対して実行されます。アプリケーションが項目を読み取り、読み取った値に基づいて書き戻す必要がある場合 (読み取り/変更/書き込みサイクル)、読み取りと書き込みの間に別のプロセスが項目を変更する可能性があるため、ロック戦略が必要になります。

同時更新を処理するには、主に 2 つの戦略があります。

- 楽観的ロック — 競合がまれであると想定します。同時アクセスを許可し、条件付き書き込みを使用して書き込み時に競合を検出します。競合が検出されると、書き込みは失敗し、アプリケーションは再試行できます。
- 悲観的ロック — 競合が発生する可能性が高いと想定します。リソースを変更する前に排他的アクセスを取得することで、同時アクセスを防止します。他のプロセスは、ロックが解除されるまで待つ必要があります。

次の表は、DynamoDB で使用できるアプローチをまとめたものです。

アプローチ	メカニズム	適しているチーム
オプティミスティックロック	バージョン属性 + 条件付き書き込み	競合が少なく、再試行コストが低い
悲観的ロック (トランザクション)	`TransactWriteItems`	複数項目のアトミック性、中程度の競合
悲観的ロック (ロッククライアント)	リースとハートビートを備えた専用ロックテーブル	長時間実行されるワークフロー、分散調整

バージョン番号を使用した楽観的ロック

楽観的ロックは、競合を回避するのではなく、書き込み時に競合を検出する戦略です。各項目には、更新ごとに増分するバージョン属性が含まれています。項目を更新するときは、バージョン番号がアプリケーションが最後に読み取った値と一致するかどうかをチェックする条件式を含めます。その間に別のプロセスが項目を変更した場合、条件は失敗し、DynamoDB は `ConditionalCheckFailedException` を返します。

どのような場合に楽観的ロックを使用するか

楽観的ロックは、次の場合に適しています。

- 複数のユーザーまたはプロセスが同じ項目を更新する可能性がありますが、競合はまれです。
- 失敗した書き込みを再試行することは、アプリケーションにとって安価です。
- 分散ロックの管理に伴うオーバーヘッドや複雑さを回避したい。

一般的な例としては、e コマース在庫の更新、共同編集プラットフォーム、金融取引レコードなどがあります。

トレードオフ

競合率が高い場合の再試行のオーバーヘッド

高同時実行性環境では、競合の可能性が高くなり、再試行や書き込みコストの増大につながる可能性があります。

実装の複雑さ

項目にバージョン管理を追加し、条件付きチェックを処理すると、アプリケーションロジックが複雑になります。AWS SDK for Java v2 拡張クライアントは、自動的にバージョン番号を管理する `@DynamoDbVersionAttribute` 注釈を通じて組み込みサポートを提供します。

パターンの設計

各項目にバージョン属性を含めます。シンプルなスキーマ設計は次のとおりです。

- パーティションキー – 各項目の一意の識別子 (`ItemId` など)。
- 属性:

- ItemId – ジョブの一意の識別子
- Version – 項目のバージョン番号を表す整数
- QuantityLeft – 項目の残りのインベントリ

項目が最初に作成されると、Version 属性は 1 に設定されます。更新される都度、バージョン番号は 1 ずつ増分します。

ItemID (パーティションキー)	バージョン	QuantityLeft
Bananas	1	10
Apples	1	5
Oranges	1	7

実装

楽観的ロックを実装するには、次の手順に従います。

1. 項目の最新バージョンを読み取ります。

```
def get_item(item_id):
    response = table.get_item(Key={'ItemID': item_id})
    return response['Item']

item = get_item('Bananas')
current_version = item['Version']
```

2. バージョン番号をチェックする条件式を使用して項目を更新します。

```
def update_item(item_id, qty_bought, current_version):
    try:
        response = table.update_item(
            Key={'ItemID': item_id},
            UpdateExpression="SET QuantityLeft = QuantityLeft - :qty, Version
 = :new_v",
            ConditionExpression="Version = :expected_v",
            ExpressionAttributeValues={
                ':qty': qty_bought,
```

```
                ':new_v': current_version + 1,
                ':expected_v': current_version
            },
            ReturnValues="UPDATED_NEW"
        )
        return response
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            print("Version conflict: another process updated this item.")
        raise
```

3. 新しい読み取りを再試行して競合を処理します。

   各再試行には追加の読み取りが必要なため、再試行の合計数を制限します。

```
def update_with_retry(item_id, qty_bought, max_retries=3):
    for attempt in range(max_retries):
        item = get_item(item_id)
        try:
            return update_item(item_id, qty_bought, item['Version'])
        except ClientError as e:
            if e.response['Error']['Code'] != 'ConditionalCheckFailedException':
                raise
            print(f"Retry {attempt + 1}/{max_retries}")
    raise Exception("Update failed after maximum retries.")
```

Java アプリケーションの場合、AWS SDK for Java v2 拡張クライアントは、`@DynamoDbVersionAttribute` 注釈を通じて組み込みの楽観的ロックサポートを提供し、自動的にバージョン番号を管理します。

条件式の詳細については、「DynamoDB 条件式 CLI の例」を参照してください。

DynamoDB トランザクションによる悲観的ロック

DynamoDB トランザクションは、グループ化されたオペレーションにオールオアナッシングアプローチを提供します。`TransactWriteItems` を使用すると、DynamoDB はトランザクション内のすべての項目をモニタリングします。トランザクション中に別のオペレーションによって項目が変更された場合、トランザクション全体がキャンセルされ、DynamoDB は `TransactionCanceledException` を返します。この動作は、競合する同時変更が事後に検出されるのではなく防止されるため、悲観的な同時実行制御の一形式を提供します。

どのような場合にロックにトランザクションを使用するか

トランザクションは、次の場合に適しています。

- 同じテーブル内またはテーブル間で、複数の項目をアトミックに更新する必要があります。
- ビジネスロジックでは、オールオアナッシングセマンティクス (すべての変更が成功するか、まったく適用されないかのいずれか) が必要です。

一般的な例としては、アカウント間での資金の送金、在庫テーブルと注文テーブルの両方を更新する注文の配置、ゲーム内のプレイヤー間でのアイテムの交換などがあります。

トレードオフ

書き込みコストの増加

最大 1 KB の項目の場合、トランザクションは項目ごとに 2 WCU (1 つは準備用、もう 1 つはコミット用) を消費しますが、標準書き込みの場合は 1 WCU です。

項目の制限

1 つのトランザクションには、1 つ以上のテーブルで最大 100 個のアクションを含めることができます。

競合の機密性

トランザクション内の項目が別のオペレーションによって変更されると、トランザクション全体が失敗します。競合率が高いシナリオでは、頻繁にキャンセルが発生する可能性があります。

実装

次の例では、`TransactWriteItems` を使用して 2 つの項目間で在庫をアトミックに転送します。トランザクション中に別のプロセスがいずれかの項目を変更すると、オペレーション全体がロールバックされます。

```
import boto3

client = boto3.client('dynamodb')

def transfer_inventory(source_id, target_id, quantity):
    try:
```

```
        client.transact_write_items(
            TransactItems=[
                {
                    'Update': {
                        'TableName': 'Inventory',
                        'Key': {'ItemID': {'S': source_id}},
                        'UpdateExpression': 'SET QuantityLeft = QuantityLeft - :qty',
                        'ConditionExpression': 'QuantityLeft >= :qty',
                        'ExpressionAttributeValues': {
                            ':qty': {'N': str(quantity)}
                        }
                    }
                },
                {
                    'Update': {
                        'TableName': 'Inventory',
                        'Key': {'ItemID': {'S': target_id}},
                        'UpdateExpression': 'SET QuantityLeft = QuantityLeft + :qty',
                        'ExpressionAttributeValues': {
                            ':qty': {'N': str(quantity)}
                        }
                    }
                }
            ]
        )
        return True
    except client.exceptions.TransactionCanceledException as e:
        print(f"Transaction canceled: {e}")
        return False
```

この例では、条件式は十分な在庫が存在することを確認しますが、バージョン属性は必要ありません。DynamoDB は、準備フェーズとコミットフェーズの間に別のオペレーションによってトランザクション内の項目が変更された場合、トランザクションを自動的にキャンセルします。これにより、悲観的な同時実行制御が実現します。競合する同時変更はトランザクション自体によって防止されます。

> **Note**
>
> バージョンチェックを追加の条件式として追加することで、トランザクションと楽観的ロックを組み合わせることができます。これにより、追加の保護レイヤーが提供されますが、トランザクションが競合を検出するためには必須ではありません。

詳細については、「DynamoDB トランザクションで複雑なワークフローを管理する」を参照してください。

DynamoDB ロッククライアントによる分散ロック

従来のロック/取得/解除セマンティクスを必要とするアプリケーションの場合、DynamoDB ロッククライアントは、DynamoDB テーブルをロックストアとして使用して分散ロックを実装するオープンソースライブラリです。このアプローチは、複数のアプリケーションインスタンス間で外部リソース (S3 オブジェクトや共有設定など) へのアクセスを調整する必要がある場合に便利です。

ロッククライアントはオープンソースの Java ライブラリとして使用できます。

仕組み

ロッククライアントは、専用の DynamoDB テーブルを使用してロックを追跡します。各ロックは、次の主要属性を持つ項目として表されます。

- ロックされているリソースを識別するパーティションキー。
- ロックが有効である期間を指定するリース期間。ロックホルダーがクラッシュするか、応答しなくなった場合、ロックはリース期間後に自動的に期限切れになります。
- ロックホルダーがリースを延長するために定期的に送信するハートビート。これにより、ホルダーがまだアクティブに処理している間にロックの有効期限が切れるのを防ぎます。

ロッククライアントは条件付き書き込みを使用して、一度に 1 つのプロセスのみがロックを取得できるようにします。ロックがすでに保持されている場合、呼び出し元は待機して再試行するか、すぐに失敗するかを選択できます。

どのような場合にロッククライアントを使用するか

ロッククライアントは、次の場合に適しています。

- 複数のアプリケーションインスタンスまたはマイクロサービス間で共有リソースへのアクセスを調整する必要があります。
- クリティカルセクションは長時間 (数秒から数分) 実行されるため、競合時にオペレーション全体を再試行するとコストがかかります。
- プロセスの失敗を適切に処理するには、自動ロックの有効期限が必要です。

一般的な例としては、分散ワークフローのオーケストレーション、複数のインスタンス間での cron ジョブの調整、共有外部リソースへのアクセスの管理などがあります。

トレードオフ

追加のインフラストラクチャ

ロック管理には専用の DynamoDB テーブルが必要で、ロック操作とハートビート用の追加の読み取りおよび書き込み容量も必要です。

クロック依存性

ロックの有効期限はタイムスタンプに依存します。クライアント間で大幅なクロックスキューが発生すると、特にリース期間が短い場合、予期しない動作が発生する可能性があります。

デッドロックリスク

アプリケーションが複数のリソースのロックを取得する場合、デッドロックを回避するために、一貫した順序でロックを取得する必要があります。リース期間は、応答しないホルダーからロックを自動的に解放することで、セーフティネットを提供します。

実装

次の例は、DynamoDB ロッククライアントを使用してロックを取得および解除する方法を示しています。

```
import java.io.IOException;
import java.util.Optional;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

final DynamoDbClient dynamoDB = DynamoDbClient.builder()
    .region(Region.US_WEST_2)
    .build();

final AmazonDynamoDBLockClient lockClient = new AmazonDynamoDBLockClient(
    AmazonDynamoDBLockClientOptions.builder(dynamoDB, "Locks")
        .withTimeUnit(TimeUnit.SECONDS)
        .withLeaseDuration(10L)
        .withHeartbeatPeriod(3L)
        .withCreateHeartbeatBackgroundThread(true)
        .build());
```

```
// Try to acquire a lock on a resource
final Optional<LockItem> lock =
    lockClient.tryAcquireLock(AcquireLockOptions.builder("my-shared-
resource").build());

if (lock.isPresent()) {
    try {
        // Perform operations that require exclusive access
        processSharedResource();
    } finally {
        // Always release the lock when done
        lockClient.releaseLock(lock.get());
    }
} else {
    System.out.println("Failed to acquire lock.");
}

lockClient.close();
```

> **Important**
>
> 処理ロジックが例外をスローした場合でもロックが確実に解除されるように、常に finally ブロックでロックを解除します。解除されていないロックは、リースの有効期限が切れるまで他のプロセスをブロックします。

条件付き書き込みを直接使用して、ロッククライアントライブラリを使用せずシンプルなロックメカニズムを実装することもできます。次の例では、条件式を指定した UpdateItem を使用してロックを取得し、DeleteItem を使用してロックを解除します。

```
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Attr

def acquire_lock(table, resource_name, owner_id, ttl_seconds):
    """Attempt to acquire a lock. Returns True if successful."""
    expiry = (datetime.now() + timedelta(seconds=ttl_seconds)).isoformat()
    now = datetime.now().isoformat()
    try:
        table.update_item(
            Key={'LockID': resource_name},
            UpdateExpression='SET #owner = :owner, #expiry = :expiry',
```

```
            ConditionExpression=Attr('LockID').not_exists() |
 Attr('ExpiresAt').lt(now),
            ExpressionAttributeNames={'#owner': 'OwnerID', '#expiry': 'ExpiresAt'},
            ExpressionAttributeValues={':owner': owner_id, ':expiry': expiry}
        )
        return True
    except table.meta.client.exceptions.ConditionalCheckFailedException:
        return False

def release_lock(table, resource_name, owner_id):
    """Release a lock. Only succeeds if the caller is the lock owner."""
    try:
        table.delete_item(
            Key={'LockID': resource_name},
            ConditionExpression=Attr('OwnerID').eq(owner_id)
        )
        return True
    except table.meta.client.exceptions.ConditionalCheckFailedException:
        return False
```

このアプローチでは、条件式を使用して、ロックが存在しないか期限切れの場合にのみロックを取得でき、ロックを取得したプロセスによってのみロックが解除されることを保証します。期限切れのロック項目を自動的にクリーンアップするには、ロックテーブルで有効期限 (TTL) を有効にすることを検討してください。

同時実行制御戦略の選択

ワークロードに適したアプローチを選択するには、次のガイドラインを使用します。

次の場合は、楽観的ロックを使用します。

- 競合はまれです。
- 失敗した書き込みを再試行するのは安価です。
- 一度に 1 つの項目を更新します。

次の場合は、トランザクションを使用します。

- 複数の項目をアトミックに更新する必要があります。
- 項目またはテーブル全体でオールオアナッシングセマンティクスが必要です。
- 条件チェックと書き込みを 1 つのオペレーションで組み合わせる必要があります。

次の場合は、ロッククライアントを使用します。

- 分散プロセス全体で外部リソースへのアクセスを調整する必要があります。

- 重要なセクションは長時間実行され、競合時の再試行にはコストがかかります。
- プロセスの失敗を処理するには、自動ロックの有効期限が必要です。

> **Note**
>
> DynamoDB グローバルテーブルを使用する場合は、グローバルテーブルが同時更新に「最終書き込み者優先」調整戦略を使用することに注意してください。バージョン番号を使用した楽観的ロックは、あるリージョンでの書き込みがバージョンチェックなしで別のリージョンでの同時書き込みを上書きする可能性があるため、リージョン間では期待どおりに機能しません。グローバルテーブルを使用する場合は、アプリケーションレベルで競合を処理するようにアプリケーションを設計します。

DynamoDB における AWS の請求および使用状況レポートを理解するためのベストプラクティス

このドキュメントでは、DynamoDB に関連する UsageType 料金の請求コードについて説明します。

AWS は、使用したサービスのデータを含む請求および使用状況レポート (CUR) を提供します。AWS Cost and Usage Report を使用して、請求レポートを CSV 形式で Amazon S3 に公開できます。CUR を設定する際は、時間単位、日単位、または月単位で期間を分割したり、使用量をリソース ID ごとに分割するかどうかを選択できます。詳細については、「AWS Data Exports ユーザーガイド」の「コストと使用状況レポートを作成する」を参照してください。

CSV エクスポートでは、各行に関連する属性が一覧表示されます。レポートに含まれる属性の例を以下に示します。

- lineitem/UsageStartDate: 明細項目の開始日時 (UTC、その時刻を含む)。
- lineitem/UsageEndDate: 対応する明細項目の終了日時 (UTC、その時刻は含まない)。
- lineitem/ProductCode: DynamoDB では「AmazonDynamoDB」。
- lineitem/UsageType: このドキュメントに列挙されている使用タイプの説明コード。
- lineitem/Operation: 料金が発生したオペレーション名などの請求項目の情報 (オプション)。
- lineitem/ResourceId: 使用されたリソースの識別子。CUR にリソース ID 別の内訳が含まれている場合に表示されます。
- lineitem/UsageAmount: 指定した期間に発生した使用量。

- lineitem/UnblendedCost: この使用量の料金。
- lineitem/LineItemDescription: 各明細項目の説明。

CUR データディクショナリの詳細については、「AWS Data Exports ユーザーガイド」の「コストと使用状況レポート (CUR) 2.0」を参照してください。正確な名前はコンテキストによって異なることに注意してください。

`UsageType` は、`ReadCapacityUnit-Hrs`、`USW2-ReadRequestUnits`、`EU-WriteCapacityUnit-Hrs`、`USE1-TimedPITRStorage-ByteHrs` などの値を持つ文字列です。各使用タイプは、オプションのリージョンプレフィックスで始まります。プレフィックスがない場合、us-east-1 リージョンであることを示します。以下の表では、簡易的な請求リージョンコードを従来のリージョンコードとリージョン名にマッピングしています。

例えば、`USW2-ReadRequestUnits` という使用は、us-west-2 で消費される読み取りリクエストユニットを示します。

請求リージョンコード	リージョンコード	リージョン名
AFS1	af-south-1	アフリカ (ケープタウン)
APE1	ap-east-1	アジアパシフィック (香港)
APN1	ap-northeast-1	アジアパシフィック (東京)
APN2	ap-northeast-2	アジアパシフィック (ソウル)
APN3	ap-northeast-3	アジアパシフィック (大阪)
APS1	ap-southeast-1	アジアパシフィック (シンガポール)
APS2	ap-southeast-2	アジアパシフィック (シドニー)
APS3	ap-south-1	アジアパシフィック (ムンバイ)
APS4	ap-southeast-3	アジアパシフィック (ジャカルタ)

請求リージョンコード	リージョンコード	リージョン名
APS5	ap-south-2	アジアパシフィック (ハイデラバード)
APS6	ap-southeast-4	アジアパシフィック (メルボルン)
CAN1	ca-central-1	カナダ (中部)
EU	eu-west-1	欧州 (アイルランド)
EUC1	eu-central-1	欧州 (フランクフルト)
EUC2	eu-central-2	欧州 (チューリッヒ)
EUN1	eu-north-1	欧州 (ストックホルム)
EUS1	eu-south-1	欧州 (ミラノ)
EUS2	eu-south-2	欧州 (スペイン)
EUW1	eu-west-1	欧州 (アイルランド)
EUW2	eu-west-2	欧州 (ロンドン)
EUW3	eu-west-3	欧州 (パリ)
ILC1	il-central-1	イスラエル (テルアビブ)
MEC1	me-central-1	中東 (アラブ首長国連邦)
MES1	me-south-1	中東 (バーレーン)
SAE1	sa-east-1	南米 (サンパウロ)
USE1 (デフォルト)	us-east-1	米国東部 (バージニア北部)
USE2	us-east-2	米国東部 (オハイオ)
UGE1	us-gov-east-1	米国東部政府

請求リージョンコード	リージョンコード	リージョン名
UGW1	us-gov-west-1	米国西部政府
USW1	us-west-1	米国西部 (北カリフォルニア)
USW2	us-west-2	米国西部 (オレゴン)

以下のセクションでは、REG-UsageType パターンを使用して DynamoDB の料金を表示します。REG は使用が発生したリージョンを示し、usageType は請求タイプのコードを示します。例えば、CSV ファイルに USW1- ReadCapacityUnit-Hrs という明細項目がある場合、その項目は US-west-1 でプロビジョニングされた読み込みキャパシティの使用量が発生したことを示します。この場合、リストには「REG-ReadCapacityUnit-Hrs」と表示されます。

トピック

- スループット容量
- Streams
- Storage
- バックアップと復元
- データ転送
- CloudWatch Contributor Insights
- DynamoDB Accelerator (DAX)

スループット容量

プロビジョニングされたキャパシティの読み取りと書き込み

DynamoDB テーブルをプロビジョンドキャパシティモードで作成する場合、アプリケーションに必要と想定される読み取りキャパシティを指定します。使用タイプはテーブルクラス (Standard または Standard-Infrequent Access) によって異なります。1 秒あたりの使用率に基づいて読み取りと書き込みをプロビジョニングしますが、料金はプロビジョニングされた容量に基づいて時間単位で課金されます。

UsageType	単位	詳細度	説明
REG-ReadCapacityUnit-Hrs	RCU 時	時	Standard テーブルクラスを使用するプロビジョンドキャパシティモードでの読み取りに対して発生する料金。
REG-IA-ReadCapacityUnit-Hrs	RCU 時	時	Standard-IA テーブルクラスを使用するプロビジョンドキャパシティモードでの読み取りに対して発生する料金。
REG-WriteCapacityUnit-Hrs	WCU 時	時	Standard テーブルクラスを使用するプロビジョンドキャパシティモードでの書き込みに対して発生する料金。
REG-IA-WriteCapacityUnit-Hrs	WCU 時	時	Standard-IA テーブルクラスを使用するプロビジョンドキャパシティモードでの書き込みに対して発生する料金。

リザーブドキャパシティの読み取りと書き込み

リザーブドキャパシティでは、1 回限りの前払い料金を支払い、期間中、プロビジョニングされた最小使用レベルを支払う契約を結びます。リザーブドキャパシティは、割引された時間単位の料金で請求されます。リザーブドキャパシティを超えてプロビジョニングしたキャパシティには、標準のプロビジョンされたキャパシティの料金が請求されます。リザーブドキャパシティは、Standard テーブルクラスを使用する DynamoDB テーブルの単一リージョンのプロビジョニングされた読み取りおよ

び書き込みキャパシティユニット (RCU および WCU) で使用できます。1 年および 3 年のリザーブドキャパシティはどちらも同じ SKU を使用して請求されます。

UsageType	単位	詳細度	説明
REG-HeavyUsage:dynamodb.read	RCU 時	前払い、その後月払い	リザーブドキャパシティの読み取り料金: 1 回限りの前払い料金と、割引されたコミット済みの RCU 時をすべてカバーする月額料金が毎月請求されます。対応するゼロコスト REG-ReadCapacityUnit-Hrs 明細項目が表示されます。
REG-HeavyUsage:dynamodb.write	WCU 時	前払い、その後月払い	リザーブドキャパシティの書き込み料金: 1 回限りの前払い料金と、割引されたコミット済みの WCU 時をすべてカバーする月額料金が毎月請求されます。対応するゼロコスト REG-WriteCapacityUnit-Hrs 明細項目が表示されます。

オンデマンドキャパシティの読み取りと書き込み

DynamoDB テーブルをオンデマンドキャパシティモードで作成する場合、アプリケーションが実行する読み取りと書き込みに対してのみ料金が発生します。読み取りリクエストと書き込みリクエストの料金は、テーブルクラスによって異なります。

UsageType	単位	詳細度	説明
REG-ReadRequestUnits	RRU	Unit	Standard テーブルクラスを使用するオンデマンドキャパシティモードでの読み取りに対して発生する料金。
REG-IA-ReadRequestUnits	RRU	Unit	Standard-IA テーブルクラスを使用するオンデマンドキャパシティモードでの読み取りに対して発生する料金。
REG-WriteRequestUnits	WRU	Unit	Standard テーブルクラスを使用するオンデマンドキャパシティモードでの書き込みに対して発生する料金。
REG-IA-WriteRequestUnits	WRU	Unit	Standard-IA テーブルクラスを使用するオンデマンドキャパシティモードでの書き込みに対して発生する料金。

グローバルテーブルの読み取りと書き込み

DynamoDB では、各レプリカテーブルで使用されているリソースに基づいてグローバルテーブルの使用料金が請求されます。プロビジョニングされたグローバルテーブルの場合、グローバルテーブルへの書き込みリクエストは標準 WCU ではなくレプリケート WCU (rWCU) で測定され、グローバルテーブルのグローバルセカンダリインデックスへの書き込みは WCU で測定されます。オンデマンドグローバルテーブルでは、書き込みリクエストは標準 WRU ではなくレプリケート WRU (rWRU)

で測定されます。レプリケーションに使用される rWCU の数または rWRU の数は、使用しているグローバルテーブルのバージョンによって異なります。料金は、テーブルクラスによって異なります。

グローバルセカンダリインデックス (GSI) への書き込みは、標準書き込み単位 (WCU と WRU) を使用して請求されます。読み取りリクエストとデータストレージは、単一リージョンのテーブルと同様に請求されます。

テーブルレプリカを追加して新しいリージョンでグローバルテーブルを作成または拡張した場合、DynamoDB では、復元されたデータ 1 GB ごとに、追加されたリージョンのテーブル復元に対して料金が請求されます。復元されたデータは REG-RestoreDataSize-Bytes として請求されます。詳細については、「DynamoDB のバックアップと復元」を参照してください。DynamoDB では、グローバルテーブルのリージョン間でデータをレプリケートする場合、リージョン間のデータ転送料金は発生しません。

DynamoDB グローバルテーブルでオンデマンドキャパシティモードを選択すると、各レプリカテーブルでアプリケーションが使用するリソースに対してのみ請求されます。

UsageType	単位	詳細度	説明
REG-ReplWriteCapacityUnit-Hrs	rWCU 時	時	グローバルテーブル、プロビジョンド、Standard テーブルクラス。
REG-IA-ReplWriteCapacityUnit-Hrs	rWCU 時	時	グローバルテーブル、プロビジョンド、Standard-IA テーブルクラス。
REG-ReplWriteRequestUnits	rWRU	Unit	グローバルテーブル、オンデマンド、Standard テーブルクラス。
REG-IA-ReplWriteRequestUnits	rWRU	Unit	グローバルテーブル、オンデマンド、Standard- IA テーブルクラス。

Streams

DynamoDB には、DynamoDB Streams と Kinesis という 2 つのストリーミングテクノロジーがあります。これらは、それぞれ料金が異なります。

DynamoDB Streams では、読み取りリクエストユニットでのデータ読み取りに対して料金が発生します。各 `GetRecords` API コールは、ストリーム読み取りリクエストとして請求されます。DynamoDB トリガーの一部として `GetRecords` から、またはレプリケーションの一部として DynamoDB グローバルテーブルから呼び出される AWS Lambda API コールに対しては、料金は発生しません。

UsageType	単位	詳細度	説明
REG-Streams-RequestsCount	カウント	Unit	DynamoDB Streams の読み取りリクエストユニット。

Amazon Kinesis Data Streams では、データキャプチャユニットに対して料金が発生します。DynamoDB では、書き込みごとに 1 つの変更データキャプチャユニットが請求されます (最大 1 KB)。1 KB を超えるアイテムには、追加の変更データキャプチャユニットが必要です。テーブルのスループットキャパシティの管理なしに、アプリケーションが実行した書き込みに対してのみ料金が発生します。

UsageType	単位	詳細度	説明
REG-ChangeDataCaptureUnits-Kinesis	CDC ユニット	Unit	Kinesis Data Streams の変更データキャプチャユニット。

Storage

DynamoDB は、データの RAW バイトサイズと有効化した特徴量に応じて、項目あたりのストレージオーバーヘッドを加算して、請求対象データのサイズを測定します。

> **Note**
>
> `DescribeTable` はアイテムごとのストレージオーバーヘッドを含まないため、CUR のストレージ使用量は `DescribeTable` 使用時のストレージ使用量と比較して高くなります。

ストレージは 1 時間単位で計算されますが、1 時間あたりの料金の平均から計算され、月単位で課金されます。

`UsageType` ストレージは `ByteHrs` をサフィックスとして使用しますが、CUR のストレージ使用量は GB 単位で測定され、料金は GB 月単位です。

UsageType	単位	詳細度	説明
REG-TimedStorage-ByteHrs	GB	月	Standard テーブルクラスのテーブルで、DynamoDB テーブルとインデックスが使用するストレージの容量。
REG-IA-TimedStorage- ByteHrs	GB	月	Standard-IA テーブルクラスのテーブルで、DynamoDB テーブルとインデックスが使用するストレージの容量。

バックアップと復元

DynamoDB には、ポイントインタイムリカバリ (PITR) バックアップとオンデマンドバックアップの 2 種類のバックアップがあります。ユーザーは、これらのバックアップから DynamoDB テーブルに復元することもできます。以下の料金は、バックアップと復元の両方を対象としています。

バックアップストレージの料金は各月の初日に発生し、バックアップが追加または削除されるたびに調整されます。詳細については、「Understanding Amazon DynamoDB On-demand Backups and Billing」のブログ記事を参照してください。

UsageType	単位	詳細度	説明
REG-TimedBackupStorage-ByteHrs	GB	月	DynamoDB テーブルとローカルセカンダリインデックスのオンデマンドバックアップで消費されるストレージ。
TimedPITRStorage-ByteHrs	GB	月	ポイントインタイムリカバリ (PITR) バックアップで使用されるストレージ。PITR が有効である間、DynamoDB では、PITR が有効化されているテーブルサイズの 1 か月間の継続的なモニタリングに基づいてバックアップ料金が決定されます。
REG-RestoreDataSize-Bytes	GB	サイズ	DynamoDB バックアップから復元されたデータ (テーブルデータ、ローカルセカンダリインデックス、グローバルセカンダリインデックスを含む) の合計サイズ (GB 単位)。

AWS Backup

AWS Backup はフルマネージド型のバックアップサービスで、クラウドおよびオンプレミスの AWS サービス間でデータのバックアップを簡単に一元化および自動化できます。AWS Backup は、

ストレージ (ウォームストレージまたはコールドストレージ)、復元作業、およびリージョン間のデータ転送に対して請求されます。以下の UsageType 料金は、「AmazonDynamoDB」ではなく「AWSBackup」ProductCode の下に表示されます。

UsageType	単位	詳細度	説明
REG-WarmStorage-ByteHrs-DynamoDB	GB	月	1 か月を通して AWS Backup で管理される DynamoDB バックアップによって使用されたストレージ (GB-月単位)。
REG-CrossRegion-WarmBytes-DynamoDB	GB	サイズ	同一アカウントの異なる AWS リージョン、または異なる AWS アカウントに転送されたデータ。クロスリージョン転送の料金は、あるリージョンから別のリージョンにバックアップをコピーする際に発生します。料金は常にデータの転送元のアカウントに請求されます。
REG-Restore-WarmBytes-DynamoDB	GB	サイズ	ウォームストレージから復元されたデータの合計サイズ。GB 単位で測定されます。
REG-ColdStorage-ByteHrs-DynamoDB	GB	月	1 か月を通して AWS Backup で管理される DynamoDB バックアップによって使用

UsageType	単位	詳細度	説明
			されたコールドストレージ (GB-月単位)。
REG-Restore-ColdBytes-DynamoDB	GB	月	コールドストレージから復元されたデータの合計サイズ。GB 単位で測定されます。

エクスポートおよびインポート

DynamoDB から Amazon S3 にデータをエクスポートしたり、Amazon S3 から新しい DynamoDB テーブルにデータをインポートしたりすることができます。

`UsageType` は `Bytes` をサフィックスとして使用しますが、CUR のエクスポートとインポートの使用量は GB 単位で測定され請求されます。

UsageType	単位	詳細度	説明
REG-ExportDataSize-Bytes	GB	サイズ	S3 へのデータエクスポートの料金。DynamoDB でのデータエクスポートは、エクスポートが作成された時点の DynamoDB ベーステーブル (テーブルデータおよびローカルセカンダリインデックス) のサイズに基づいて課金されます。
REG-ImportDataSize-Bytes	GB	サイズ	S3 からのデータインポートの料金。サイズは、Amazon S3 のデータのオブジェ

UsageType	単位	詳細度	説明
			クトの非圧縮サイズに基づいて計算されます。GSI を使用したテーブルへのインポートには追加料金はかかりません。
REG-IncrementalExportDataSize-Bytes	GB	サイズ	継続バックアップから増分エクスポートを行うために処理されたデータのサイズに基づく料金。

データ転送

データ転送アクティビティは DynamoDB サービスに関連して表示される場合があります。DynamoDB では、インバウンドデータ転送の料金は発生しません。また、DynamoDB と同じ AWS リージョン内の他の AWS サービスとの間のデータ転送の料金も発生しません (つまり、GB あたり 0.00 USD)。AWS リージョン間 (米国東部 [バージニア北部] リージョンの DynamoDB と欧州 [アイルランド] リージョンの Amazon EC2 間など) で転送されるデータは、転送元と転送先の両方で料金が発生します。

UsageType	単位	詳細度	説明
REG-DataTransfer-In-Bytes	GB	単位	インターネットから DynamoDB へのデータ転送。
REG-DataTransfer-Out-Bytes	GB	単位	DynamoDB からインターネットへのデータ転送。

CloudWatch Contributor Insights

CloudWatch Contributor Insights for DynamoDB は、DynamoDB テーブルで最も頻繁にアクセスされスロットリングされるキーを特定する診断ツールです。以下の UsageType 料金は、「AmazonDynamoDB」ではなく「AmazonCloudWatch」ProductCode の下に表示されます。

UsageType	単位	詳細度	説明
REG-CW:ContributorEventsManaged	処理されたイベント	単位	処理された DynamoDB イベントの量。例えば、CloudWatch Contributor Insights が有効になっているテーブルでは、アイテムが読み書きされるたびに 1 イベントとしてカウントされます。テーブルにソートキーがある場合は、2 イベントとしてカウントされます。
REG-CW:ContributorRulesManaged	ルール数	月	Cloud Watch Contributor Insights を有効にすると、DynamoDB は最もアクセス数の多いアイテムと最もスロットリングされたキーを識別するルールを作成します。この料金は、CloudWatch contributor insights のログ用に設定された各エンティティ (テー

UsageType	単位	詳細度	説明
			ブルおよび GSI) に追加されたルールに対して発生します。

DynamoDB Accelerator (DAX)

DynamoDB Accelerator (DAX) は、サービスで選択されたインスタンスタイプに基づいて時間単位で請求されます。以下の料金は、プロビジョニングされた DynamoDB Accelerator インスタンスに関するものです。以下の UsageType 料金は、「AmazonDynamoDB」ではなく「AmazonDAX」ProductCode の下に表示されます。

UsageType	単位	詳細度	説明
REG-NodeUsage:dax-<INSTANCETYPE>	ノード時間	時	特定のインスタンスタイプの 1 時間あたりの使用量。料金は、ノードが起動されてから終了するまでの、消費されたノード時間あたりの価格です。消費されたノード時間が 1 時間に満たない場合でも、1 時間分として請求されます。DAX は DAX クラスター内のノードごとに課金されます。クラスターに複数のノードがある場合、請求レポートには複数の明細項目が表示されます。

インスタンスタイプは、次のリストに示す値の 1 つになります。ノードタイプの詳細については、「ノード」を参照してください。

- r3.2xlarge、r4.8xlarge、または r5.8xlarge
- r3.4xlarge、r4.large、または r5.large
- r3.8xlarge、r4.xlarge、または r5.xlarge
- r3.2xlarge、r5.12xlarge、または t2.medium
- r3.4xlarge、r4.large、または r5.large
- r3.xlarge、r5.16xlarge、または t2.small
- r4.16xlarge、r5.24xlarge、または t3.medium
- r4.2xlarge、r5.2xlarge、または t3.small
- r4.4xlarge または r5.4xlarge

DynamoDB テーブルをアカウントから別のアカウントに移行する

Amazon DynamoDB テーブルをあるアカウントから別のアカウントに移行して、マルチアカウント戦略またはバックアップ戦略を実装できます。また、テスト、デバッグ、またはコンプライアンスを目的として移行を行うこともできます。一般的なユースケースは、各環境で異なる AWS アカウントが使用されている本番環境、ステージング環境、テスト環境、および開発環境間での DynamoDB テーブルのコピーです。

DynamoDB には、ある AWS アカウントから別のアカウントにテーブルを移行するための 2 つのオプションがあります。

- クロスアカウントバックアップと復元用の AWS Backup: AWS Backup は、フルマネージド型のバックアップサービスで、複数の AWS サービス間のバックアップを一元管理できます。クロスアカウントバックアップおよび復元機能を使用すると、あるアカウントで DynamoDB テーブルをバックアップして、同じ AWS Organization 内の別のアカウントにバックアップを復元できます。
- Amazon S3 への DynamoDB のエクスポートとインポート: Amazon S3 への DynamoDB のエクスポートとインポート機能を使用すると、Amazon S3 バケットへの完全なエクスポートを実行して、そのデータを別の AWS アカウントの新しいテーブルにインポートできます。このアプローチは、同じ AWS Organization の一部ではないアカウント間でのデータの移行が必要な場合や、AWS Backup を使用しない場合に適しています。

> **Note**
>
> Amazon S3 からのインポートでは、ローカルセカンダリインデックス (LSI) を使用しているテーブルはサポートされていませんが、グローバルセカンダリインデックス (GSI) はサポートされています。LSI および GSI の詳細については、「DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上」を参照してください。

トピック

- クロスアカウントバックアップと復元用の AWS Backup を使用してテーブルを移行する
- S3 へのエクスポートと S3 からのインポートを使用してテーブルを移行する

クロスアカウントバックアップと復元用の AWS Backup を使用してテーブルを移行する

前提条件

- ソースおよびターゲットの AWS アカウントは、AWS Organizations サービスの同じ組織に属していること
- AWS Backup ボールトを作成し使用するための有効な AWS Identity and Access Management (IAM) アクセス許可があること

詳細については、「Creating backup copies across AWS accounts」を参照してください。

料金情報

AWS では、バックアップ (テーブルサイズに基づく)、AWS リージョン間のデータコピー (データ量に基づく)、復元 (データ量に基づく)、および進行中のストレージ料金に対する費用が発生します。継続的な費用の発生を避けるために、復元後にバックアップが必要ない場合は、バックアップを削除できます。

料金の詳細については、「AWS Backup の料金」を参照してください。

ステップ 1: DynamoDB の高度な機能とクロスアカウントバックアップを有効にする

1. ソースおよびターゲットの AWS アカウントの両方で、AWS マネジメントコンソールにアクセスして、AWS バックアップコンソールを開きます。

2. [設定] オプションを選択します。
3. [Amazon DynamoDB バックアップの高度な機能] で、[高度な機能] が有効になっていることを確認します。有効でない場合は、[有効化] を選択します。
4. [クロスアカウント管理] の下の [クロスアカウントバックアップ] で、[オンにする] を選択します。

ステップ 2: ソースアカウントとターゲットアカウントにバックアップボールトを作成する

1. ソース AWS アカウントで、AWS バックアップコンソールを開きます。
2. [バックアップボールト] を選択します。
3. [バックアップ保管庫を作成] を選択します。
4. 作成したバックアップボールトとターゲット AWS アカウントの [Amazon リソースネーム (ARN)] をコピーして保存します。
5. アカウント間で DynamoDB テーブルバックアップをコピーする際は、ソースバックアップボールトとターゲットバックアップボールトの両方の ARN が必要です。

ステップ 3: ソースアカウントで、DynamoDB テーブルバックアップを作成する

1. [AWS バックアップダッシュボード] ページで、[オンデマンドバックアップを作成] を選択します。
2. [設定] セクションの [リソースタイプ] で [DynamoDB] を選択して、テーブル名を選択します。
3. [バックアップボールト] ドロップダウンリストで、ソースアカウントで作成したバックアップボールトを選択します。
4. 希望する [保持期間] を選択します。
5. [オンデマンドバックアップを作成] を選択します。
6. [AWS バックアップジョブ] ページの [バックアップジョブ] タブで、バックアップジョブのステータスをモニタリングします。

ステップ 4: DynamoDB テーブルのバックアップをソースアカウントからターゲットアカウントにコピーする

1. バックアップジョブが完了したら、ソースアカウントで AWS Backup コンソールを開き、[バックアップボールト] を選択します。
2. [バックアップ] で、DynamoDB テーブルのバックアップを選択します。[アクション]、[コピー] の順に選択します。
3. ターゲットアカウントの AWS リージョンを入力します。
4. [外部ボールトの ARN] に、ターゲットアカウントで作成したバックアップボールトの ARN を入力します。
5. ターゲットアカウントのバックアップボールトで、ソースアカウントからのアクセスを有効にしてバックアップのコピーを許可します。

ステップ 5: ターゲットアカウントに DynamoDB テーブルのバックアップを復元する

1. ターゲット AWS アカウントで AWS Backup コンソールを開き、[バックアップボールト] を選択します。
2. [バックアップ]で、ソースアカウントからコピーしたバックアップを選択します。[アクション]、[復元] の順に選択します。
3. 新しい DynamoDB テーブルの名前、新しいテーブルで使用する暗号化、復元の暗号化に使用するキー、およびその他のオプションを入力します。
4. 復元が完了すると、テーブルのステータスは [アクティブ] と表示されます。

S3 へのエクスポートと S3 からのインポートを使用してテーブルを移行する

前提条件

- S3 へのエクスポートを実行するには、テーブルのポイントインタイムリカバリ (PITR) を有効にする必要があります。詳細については、「DynamoDB でポイントインタイムリカバリを有効にする」を参照してください。
- エクスポートを実行するための有効な IAM アクセス許可。詳細については、「DynamoDB でテーブルのエクスポートをリクエストする」を参照してください。

- インポートを実行するための有効な IAM アクセス許可。詳細については、「DynamoDB でテーブルのインポートをリクエストする」を参照してください。

料金情報

AWS では、PITR の料金 (テーブルのサイズと PITR が有効になっている期間に基づく) が発生します。エクスポート以外の PITR が必要ない場合は、エクスポートが終了した後に無効にできます。AWS では、S3 に対するリクエスト、エクスポートされたデータの S3 への保存とインポート (インポートされたデータの非圧縮サイズに基づく) の費用も発生します。

DynamoDB の料金の詳細については、「DynamoDB の料金表」を参照してください。

> **Note**
>
> S3 から DynamoDB にインポートする際は、オブジェクトのサイズと数に制限があります。詳細については、「インポートクォータ」を参照してください。

Amazon S3 へのテーブルのエクスポートのリクエスト

1. AWS マネジメントコンソールにサインインして、DynamoDB コンソールを開きます。
2. コンソールの左側のナビゲーションペインで、[S3 へのエクスポート] を選択します。
3. コピー元テーブルとエクスポート先の S3 バケットを選択します。`s3://bucketname/prefix` の形式で送信先アカウントバケットの URL を入力します。ここで、`/prefix` は、エクスポート先のバケットを整理するのに役立つオプションのフォルダです。
4. [フルエクスポート] を選択します。フルエクスポートでは、指定した時点のテーブルのフルテーブルスナップショットが出力されます。
   a. [現在の時刻] を選択して、最新のフルテーブルスナップショットをエクスポートします。
   b. [エクスポートされたファイル形式] で、[DynamoDB JSON] または [Amazon Ion] のいずれかを選択します。デフォルトのオプションは DynamoDB JSON です。
5. [エクスポート] を選択してエクスポートを開始します。
6. サイズの小さなテーブルのエクスポートは数分で完了しますが、テラバイトサイズのテーブルには 1 時間以上かかる場合があります。

Amazon S3 からのテーブルのインポートのリクエスト

1. AWS マネジメントコンソールにサインインして、DynamoDB コンソールを開きます。
2. コンソールの左側のナビゲーションペインで、[Import from S3] (S3 からインポート) を選択します。
3. 表示されたページで、[Import from S3] (S3 からインポート) を選択します。
4. Amazon S3 のソース URL を入力します。また、[S3 を参照] ボタンを使用して検索することもできます。予想されるパスの形式は `s3://bucket/prefix/AWSDynamoDB/<XXXXXXXX-XXXXXX>/data/` です。
5. 自分が [S3 バケット所有] であることを確認します。
6. [ファイル圧縮をインポート] で、エクスポートに一致する [GZIP] を選択します。
7. [インポートファイルのフォーマット] で、エクスポートに一致する [DynamoDB JSON] を選択します。
8. [次へ] を選択してください。[テーブルの詳細を指定] で、データを保存するために作成される新しいテーブルのオプションを選択します。
9. [次へ] を選択してください。[テーブルの設定を構成] で、必要に応じて追加のテーブル設定をカスタマイズします。
10. [次へ] をもう一度選択してインポートオプションを確認し、[インポート] を選択してインポートタスクを開始します。新しいテーブルが [インポート中] というステータスで [S3 からのインポート] の下に表示されます。この間はテーブルにアクセスできません。サイズの小さなインポートは数分で完了しますが、テラバイトサイズのインポートは 1 時間以上かかる場合があります。
11. インポートが完了すると、ステータスが [アクティブ] と表示され、テーブルの使用を開始できます。

移行中にテーブルを同期する

移行中にソーステーブルの書き込み操作を一時的に停止できる場合、移行の直後のソースと出力は一致します。書き込み操作を一時的に停止できない場合、移行後のターゲットテーブルは、通常ソースより少し古くなります。ターゲットテーブルとソーステーブルを同期するには、ストリーミング (DynamoDB Streams または Kinesis Data Streams for DynamoDB) を使用して、バックアップまたはエクスポート以降にソーステーブルで発生した書き込みを再生します。

ストリームレコードの読み取りを、ソーステーブルを S3 にエクスポートしたタイムスタンプの前に開始する必要があります。例えば、S3 へのエクスポートが午後 2 時に発生し、ターゲットテーブル

へのインポートが午後 11 時に終了した場合、DynamoDB ストリームの読み取りを午後 1 時 58 分に開始する必要があります。変更データキャプチャテーブルのストリーミングオプションは、各ストリーミングモデルの機能をまとめたものです。

Lambda で DynamoDB Streams を使用すると、ソースとターゲットの DynamoDB テーブル間でデータを同期するための効率的なアプローチが提供されます。Lambda 関数を使用して、ターゲットテーブルの各書き込みを再生できます。

> **Note**
>
> 項目は DynamoDB Streams に 24 時間保持されるため、この時間内にバックアップと復元を完了するか、エクスポートとインポートを行う必要があります。

DAX を DynamoDB アプリケーションと統合するための規範ガイダンス

DynamoDB Accelerator (DAX) は、読み取り負荷の高いアプリケーションなどの要求の厳しいアプリケーションに高速なインメモリパフォーマンスを提供する DynamoDB 互換キャッシュサービスです。DAX を使用すると、頻繁にリクエストされるデータにアクセスするための応答時間をマイクロ秒単位で実現できます。この DynamoDB Accelerator の規範ガイドでは、DAX を DynamoDB アプリケーションと統合するための包括的なインサイトとベストプラクティスを提供します。

このガイドでは、DAX を初めて使用する際や、既存の設定を最適化する際の基本的な知識を提供します。このガイドでは、DAX を使用するタイミングや DAX クラスターの作成など、さまざまなトピックについて説明します。また、プロジェクトへの DAX の効果的な実装に役立つ実用的な例と詳細についても説明します。最後に、このガイドでは、DAX のキャッシュ機能を最大化して、高速でスケーラブルなアプリケーションの実現に必要な高度な戦略を提供します。

トピック

- ユースケースに対する DAX の適合性の評価
- DAX クライアントの設定
- DAX クラスターの設定
- クラスターのサイジング
- クラスターのデプロイ
- クラスターオペレーションの管理

- DAX のモニタリング

ユースケースに対する DAX の適合性の評価

このセクションでは、DAX を使用するタイミングとその理由について説明します。このガイダンスを使用すると、DAX の DynamoDB への統合が、アプリケーションのワークロードパターン、パフォーマンス要件、およびデータ整合性のニーズに効果的かどうかを判断できます。また、書き込み負荷の高いワークロードやアクセス頻度の低いデータなど、DAX が適さないシナリオについても説明します。

このセクションの内容

- DAX を選択するタイミングとその理由
- DAX の使用が適さない場合

DAX を選択するタイミングとその理由

いくつかのシナリオで、アプリケーションスタックに DAX を追加することを検討できます。例えば、DAX を使用して DynamoDB に対する読み取りリクエストの全体的なレイテンシーを短縮したり、テーブルからの同じデータの繰り返し読み取りを最小限に抑えたりできます。次のリストは、DAX と DynamoDB の統合を活用できるシナリオの例を示しています。

- 高パフォーマンス要件
  - 低レイテンシーの読み取り – アプリケーションが結果整合性のある読み取りにマイクロ秒単位の応答時間を必要とする場合は、DAX の使用を検討する必要があります。DAX は、頻繁に読み込まれるデータにアクセスするための応答時間を大幅に短縮することもできます。
- 読み取り負荷の高いワークロード
  - 読み取り負荷の高いアプリケーション – 10:1 以上など、書き込みに対する読み取りの比率が高いアプリケーションの場合、DAX はキャッシュヒットを増やし、古いデータを減らします。これにより、テーブルに対する読み取りが減少します。書き込み負荷が高い場合にアプリケーションがキャッシュから古いデータを読み取らないようにするには、キャッシュの DynamoDB での Time to Live (TTL) の使用 を低く設定します。
  - 一般的なクエリのキャッシュ – アプリケーションが e コマースプラットフォーム上の一般的な製品など、同じデータを頻繁に読み取る場合、DAX はキャッシュから直接これらのリクエストを処理できます。
- バーストトラフィックパターン

  - よりスムーズなテーブルのスケーリング – DAX は、突然のトラフィックスパイクの影響を低減します。DAX には、DynamoDB テーブルのキャパシティを適切にスケールアップするためのバッファが用意されているため、読み取りスロットリングのリスクが軽減されます。
  - 各項目の読み取りスループットの向上 – DynamoDB は、各項目に個々のパーティションを割り当てます。ただし、3,000 読み取りキャパシティーユニット (RCU) に達すると、パーティションは項目の読み取りスロットリングを開始します。DAX では、1 つの項目の読み取りを 3,000 RCU 以上にスケーリングできます。
- コスト最適化
  - DynamoDB コストの削減 – DAX からの読み取りにより、DynamoDB テーブルに送信される読み取りが減少し、コストに直接影響する可能性があります。キャッシュヒットレートが高い場合、テーブルの読み取りコストが DAX クラスターのコストを上回り、総コストの削減につながります。
- データ整合性の要件
  - 結果整合性 — DAX は結果整合性のある読み込みをサポートします。このため、DAX は即時整合性が重要でないユースケースに適しています。
  - 書き込みスルーキャッシュ — DAX に対して行う書き込みは書き込みスルーです。DAX は、項目が DynamoDB に書き込まれたことを確認すると、その項目バージョンを項目キャッシュに保持します。この書き込みスルーの仕組みは、キャッシュとデータベース間のデータ整合性をより厳密に維持するのに役立ちますが、追加の DAX クラスターリソースを使用します。

DAX の使用が適さない場合

DAX は強力な機能ですが、すべてのシナリオに適しているわけではありません。次のリストは、DAX と DynamoDB の統合が適さないシナリオの例を示しています。

- 書き込み負荷の高いワークロード – DAX の主な利点は読み取りを高速化することですが、書き込みでは読み取りよりも多くの DAX リソースが使用されます。主に書き込み負荷の高いアプリケーションの場合、DAX の利点が制限される可能性があります。
- データの読み取り頻度が低い – アプリケーションがデータに低頻度でアクセスしたり、再利用頻度の低いデータ (コールドデータ) に広範囲にわたってアクセスしたりすると、cache hit ratio が低くなります。この場合、キャッシュを維持するためのオーバーヘッドは、パフォーマンスの向上を正当化できない可能性があります。
- 一括読み取りまたは一括書き込み — アプリケーションが個別の書き込みよりも多くの一括書き込みを実行する場合は、DAX を書き込みアラウンドする必要があります。さらに、一括読み取りの場合は、DynamoDB テーブルに対して完全なテーブルスキャンを直接実行する必要があります。

- 強力な整合性要件またはトランザクション要件 – DAX は強力な整合性のある読み込みと TransactGetItems 呼び出しを DynamoDB テーブルに渡します。クラスターリソースの使用を避けるため、これらの読み取りは DAX クラスターを読み取りアラウンドする必要があります。この方法で読み取られた項目はキャッシュされないため、DAX 経由でこれらの項目をルーティングすることは意味がありません。
- パフォーマンス要件の低いシンプルなアプリケーション – パフォーマンス要件が低く、直接の DynamoDB レイテンシーに対する許容度があるアプリケーションの場合、DAX を追加する複雑さとコストは必要ではない場合があります。DynamoDB は単独で高スループットを処理し、1 桁ミリ秒のパフォーマンスを提供します。
- キーと値によるアクセスを超えた複雑なクエリニーズ – DAX はキーと値によるアクセスパターンに最適化されています。アプリケーションがクエリ操作やスキャン操作など、複雑なフィルタリングを伴う複雑なクエリ機能を必要とする場合、DAX キャッシュの利点が制限される可能性があります。

  このような状況では、代わりに Amazon ElastiCache (Redis OSS) を使用してください。ElastiCache (Redis OSS) は、リスト、セット、ハッシュなどの高度なデータ構造をサポートしています。また、pub/sub、地理空間インデックス、スクリプティングなどの機能も提供します。
- コンプライアンス要件 – DAX は現在、DynamoDB と同じコンプライアンス認定を提供していません。例えば、DAX は SOC 認定をまだ取得していません。

DAX クライアントの設定

DAX クラスターは、さまざまな DAX SDK を使用してアクセスできるインタスタンスベースのクラスターです。各 SDK は、特定のアプリケーション要件を満たすために、requestTimeout や接続などの設定可能なオプションを開発者に提供します。

DAX クライアントを設定する場合、クライアントアプリケーションのスケール、特にクライアントインスタンスと DAX サーバーインスタンスの比率 (最大 11) が主な考慮事項です。大規模なクライアントインスタンスフリートは、DAX サーバーインスタンスへの多数の接続を生成し、負担をかける可能性があります。このガイドでは、DAX クライアント設定のベストプラクティスの概要を説明します。

ベストプラクティス

1. クライアントインスタンス – シングルトンクライアントインスタンスを実装して、リクエスト間でインスタンスを再利用します。実装の詳細については、「the section called “ステップ 4: サンプルアプリケーションの実行”」を参照してください。
2. リクエストタイムアウト – アップストリームシステムのレイテンシーを最小限に抑えるために、アプリケーションでは多くの場合、リクエストタイムアウトを短くする必要がありますが、タイムアウトを低く設定しすぎると問題が発生する可能性があります。タイムアウトが短いと、DAX サーバーで一時的なレイテンシーのスパイクが発生した場合に、サーバーインスタンスへの頻繁な再接続がトリガーされる可能性があります。タイムアウトが発生すると、DAX クライアントは既存のサーバーノード接続を終了し、新しいサーバーノード接続を確立します。接続の確立には大量のリソース消費が必要になるため、多数の連続した接続により DAX サーバーが過負荷になる可能性があります。次の構成を推奨します。
   - デフォルトのリクエストタイムアウト設定の維持。
   - 短いタイムアウトが必要な場合は、低いタイムアウト値で個別のアプリケーションスレッドを実装し、指数バックオフによる再試行メカニズムを含めます。
3. 接続タイムアウト – ほとんどのアプリケーションでは、デフォルトの接続タイムアウト設定を維持することをお勧めします。
4. 同時接続 – JavaV2 などの特定の SDK では、DAX サーバーへの同時接続を調整できます。主な考慮事項:
   - DAX サーバーインスタンスは、最大 40,000 の同時接続を処理できます。
   - デフォルト設定は、ほとんどのユースケースに適しています。
   - 大規模なクライアントインスタンスと多数の同時接続が組み合わさると、サーバーが過負荷になる可能性があります。
   - 同時接続値が低いほど、サーバー過負荷のリスクが軽減されます。
   - パフォーマンス計算の例:
     - リクエストレイテンシーが 1 ミリ秒であるとすると、各接続は理論的に 1,000 リクエスト/秒を処理できます。
     - 3 ノードクラスターの場合、すべてのノードに接続する 1 つのクライアントインスタンスで 3,000 リクエスト/秒を処理できます。
     - 接続数が 10 の場合、クライアントは約 30,000 リクエスト/秒を処理できます。

       推奨事項 – 低い同時接続設定で開始し、予想される本番ワークロードパターンを使用して、パフォーマンステストを通じて検証します。

DAX クラスターの設定

DAX クラスターはマネージドクラスターですが、アプリケーションの要件に合わせて設定を調整できます。DynamoDB API オペレーションと緊密に統合されているため、アプリケーションを DAX と統合する際は、以下の点を考慮する必要があります。

このセクションの内容

- DAX の料金
- 項目キャッシュとクエリキャッシュ
- キャッシュの TTL 設定の選択
- DAX クラスターを使用した複数のテーブルのキャッシュ
- DAX および DynamoDB グローバルテーブルのデータレプリケーション
- DAX が利用可能なリージョン
- DAX キャッシュの動作

DAX の料金

クラスターのコストは、プロビジョニングしたノードの数とサイズによって異なります。すべてのノードでは、クラスターで実行される 1 時間ごとに費用が発生します。詳細については、「Amazon DynamoDB 料金」を参照してください。

キャッシュヒットによって DynamoDB の費用が発生することはありませんが、DAX クラスターリソースに影響を与えます。キャッシュミスでは DynamoDB の読み取り費用が発生し、DAX リソースが必要です。書き込みでは DynamoDB の書き込み費用が発生し、書き込みをプロキシするための DAX クラスターリソースに影響を与えます。

項目キャッシュとクエリキャッシュ

DAX は項目キャッシュとクエリキャッシュを保持します。これらのキャッシュの違いを理解することは、キャッシュがアプリケーションに与えるパフォーマンスと整合性の特性を判断するのに役立ちます。

キャッシュの特性	項目キャッシュ	クエリキャッシュ
目的	GetItem および BatchGetItem API オペレーションの結果を保存します。	クエリおよびスキャン API オペレーションの結果を保存します。これらのオペレーショ

キャッシュの特性	項目キャッシュ	クエリキャッシュ
		ンは、特定の項目キーではなく、クエリ条件に基づいて複数の項目を返すことができます。
アクセスタイプ	キーベースのアクセスを使用します。 アプリケーションが GetItem または BatchGetItem を使用してデータをリクエストすると、DAX は最初にリクエストされた項目のプライマリキーを使用して項目キャッシュをチェックします。項目がキャッシュされており、有効期限が切れていない場合、DAX は DynamoDB テーブルにアクセスすることなく、すぐにデータを返します。	パラメータベースのアクセスを使用します。 DAX は、Query および Scan API オペレーションの結果セットをキャッシュします。DAX は、同じクエリ条件、テーブル、インデックスを含む同じパラメータで後続のリクエストをキャッシュから処理します。これにより、応答時間と DynamoDB 読み取りスループットの消費量が大幅に削減されます。

キャッシュの特性	項目キャッシュ	クエリキャッシュ
キャッシュの無効化	DAX は、以下のシナリオで、更新された項目を DAX クラスター内のノードの項目キャッシュに自動的にレプリケートします。 • キャッシュを使用して項目の更新を書き込む。 • 更新された項目バージョンをテーブルから読み取る。	クエリキャッシュは、項目キャッシュよりも無効化が困難です。項目の更新は、キャッシュされたクエリやスキャンに直接マッピングされない場合があります。データ整合性を維持するには、クエリキャッシュの TTL を慎重に調整する必要があります。DAX またはベーステーブルを介した書き込みは、TTL が以前にキャッシュされた応答を期限切れにし、DAX が DynamoDB に対して新しいクエリを実行するまで、クエリキャッシュに反映されません。
グローバルセカンダリインデックス	`GetItem` API オペレーションはローカルセカンダリインデックスまたはグローバルセカンダリインデックスではサポートされていないため、項目キャッシュはベーステーブルからの読み取りのみをキャッシュします。	クエリキャッシュは、テーブルとインデックスの両方に対してクエリをキャッシュします。

キャッシュの TTL 設定の選択

TTL は、データが期限切れになるまでに、キャッシュに保存される期間を決定します。この期間が過ぎると、次のリクエストでデータが自動的に更新されます。DAX キャッシュに適切な TTL 設定を選択する場合は、アプリケーションのパフォーマンスの最適化とデータ整合性のバランスをとる必要があります。すべてのアプリケーションに適した普遍的な TTL 設定はないため、最適な TTL 設定はアプリケーションに固有の特性と要件によって異なります。この規範ガイダンスを使用して、保守的

な TTL 設定から始めることをお勧めします。その後、アプリケーションのパフォーマンスデータとインサイトに基づいて、TTL 設定を繰り返し調整します。

DAX は項目キャッシュの LRU (least recently used) リストを保持します。LRU リストは、項目がいつキャッシュに最初に書き込まれたか、またはいつキャッシュから最後に読み取られたかを追跡します。DAX ノードのメモリがいっぱいになると、(期限切れでない場合でも) DAX はより古い項目を削除し、新しい項目のためのメモリを確保します。LRU アルゴリズムは常に有効で、ユーザーが設定することはできません。

アプリケーションに適した TTL 期間を設定するには、以下の点を考慮します。

データアクセスパターンを理解する

- 読み込み負荷の高いワークロード – 読み込み負荷が高く、データ更新の頻度が低いアプリケーションでは、TTL 期間を長く設定してキャッシュミスの数を減らします。TTL 期間を長くすると、基盤となる DynamoDB テーブルにアクセスする必要も少なくなります。
- 書き込み負荷の高いワークロード – DAX を介して書き込まれない更新が頻繁に行われるアプリケーションの場合、キャッシュとデータベースの整合性が保たれるように TTL 期間を短く設定します。TTL 期間を短くすると、古いデータが使用されるリスクも軽減されます。

アプリケーションのパフォーマンス要件を評価する

- レイテンシー感度 – データの鮮度よりも低レイテンシーがアプリケーションで必要な場合は、TTL 期間を長く設定します。TTL 時間が長くなるとキャッシュヒットが最大化され、平均読み取りレイテンシーが短縮されます。
- スループットとスケーラビリティ — TTL 時間が長くなると、DynamoDB テーブルの負荷が軽減され、スループットとスケーラビリティが向上します。ただし、スループットやスケーラビリティとデータ鮮度のバランスをとる必要があります。

キャッシュエビクションとメモリ使用量の分析

- キャッシュメモリの制限 — DAX クラスターのメモリ使用量をモニタリングします。TTL 時間が長くなるとキャッシュにより多くのデータが保存され、メモリ制限に達し、LRU ベースのエビクションが発生する可能性があります。

メトリクスとモニタリングを使用して TTL を調整する

キャッシュヒットとキャッシュミス、CPU およびメモリの使用率などのメトリクスを定期的に確認します。これらのメトリクスに基づいて TTL 設定を調整して、パフォーマンスとデータ鮮度のバランスを最適化します。キャッシュミスが多く、メモリ使用率が低い場合は、TTL 期間を長くしてキャッシュヒットレートを高めます。

ビジネス要件とコンプライアンスを考慮する

データ保持ポリシーによって、機密情報または個人情報に設定できる最大 TTL 期間が変わる場合があります。

TTL をゼロに設定する場合のキャッシュの動作

TTL を 0 に設定すると、項目キャッシュとクエリキャッシュは以下のように動作します。

- 項目キャッシュ – キャッシュ内の項目は、LRU エビクションまたは書き込みスルーオペレーションが発生した場合にのみ更新されます。
- クエリキャッシュ — クエリレスポンスはキャッシュされません。

DAX クラスターを使用した複数のテーブルのキャッシュ

個別のキャッシュを必要としない複数の小さな DynamoDB テーブルを持つワークロードの場合、単一の DAX クラスターがこれらのテーブルのリクエストをキャッシュします。これにより、特に複数のテーブルにアクセスし、高パフォーマンス読み取りを必要とするアプリケーションでは、DAX をより柔軟かつ効率的に使用できます。

DynamoDB データプレーン API と同様に、DAX リクエストにはテーブル名が必要です。同じ DAX クラスターで複数のテーブルを使用する場合、特定の設定は必要ありません。ただし、クラスターのセキュリティアクセス許可で、キャッシュされたすべてのテーブルへのアクセスが許可されていることを確認する必要があります。

複数のテーブルで DAX を使用する際の考慮事項

複数の DynamoDB テーブルで DAX を使用する場合は、以下の点を考慮する必要があります。

- メモリ管理 – 複数のテーブルで DAX を使用する場合、ワーキングデータセットの合計サイズを考慮する必要があります。データセット内のすべてのテーブルは、選択したノードタイプと同じメモリ領域を共有します。

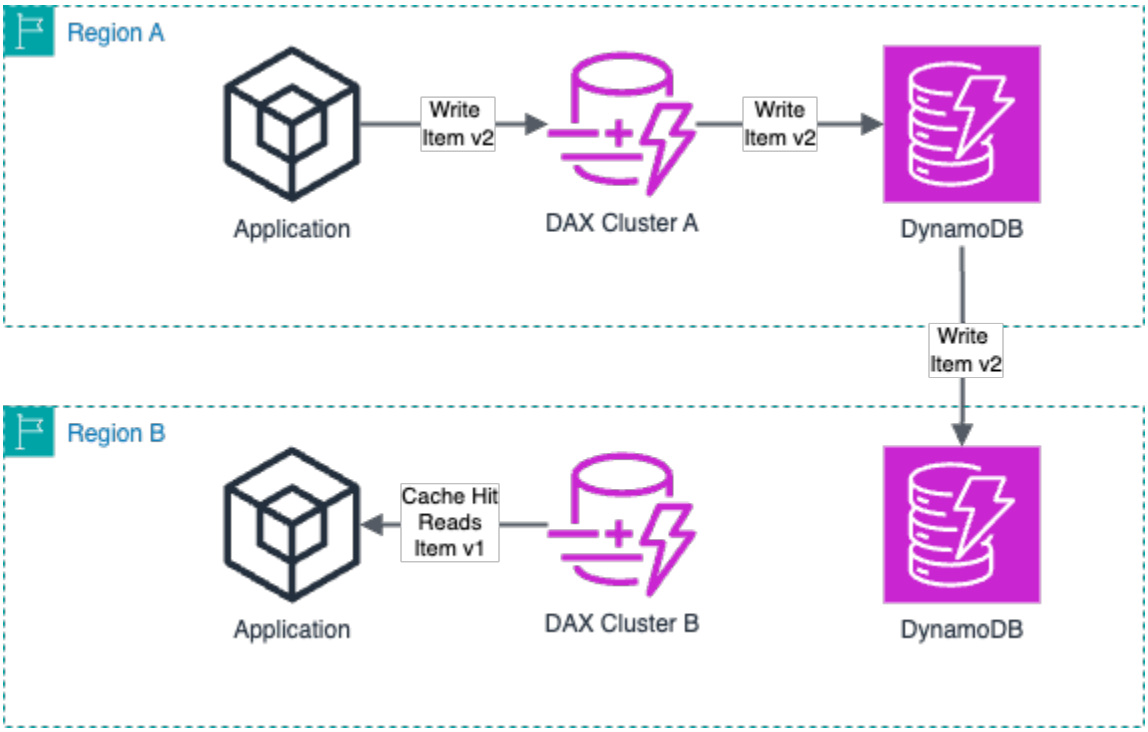
- リソース割り当て — DAX クラスターのリソースは、キャッシュされたすべてのテーブル間で共有されます。ただし、トラフィックの多いテーブルは、隣接する小さなテーブルでのデータエビクションを発生させる可能性があります。
- 規模の経済 — より小さなリソースをより大きな DAX クラスターにグループ化して、より安定したパターンにトラフィックを平均化します。DAX クラスターに必要な読み取りリソースの総数については、3 つ以上のノードを持つことも経済的です。これにより、クラスター内のすべてのキャッシュされたテーブルの可用性も向上します。

DAX および DynamoDB グローバルテーブルのデータレプリケーション

DAX はリージョンベースのサービスであるため、クラスターは自身の AWS リージョン 内のトラフィックのみを認識します。グローバルテーブルは、別のリージョンからデータをレプリケートする際に、キャッシュを書き込みアラウンドします。

TTL 時間が長くなると、古いデータがプライマリリージョンよりもセカンダリリージョンに長く残る可能性があります。これにより、セカンダリリージョンのローカルキャッシュでキャッシュミスが発生する可能性があります。

次の図は、ソースリージョン A のグローバルテーブルレベルで発生するデータレプリケーションを示しています。リージョン B の DAX クラスターは、ソースリージョン A から新しくレプリケートされたデータをすぐには認識しません。



DAX が利用可能なリージョン

DynamoDB テーブルをサポートするすべてのリージョンが DAX クラスターのデプロイをサポートしているわけではありません。アプリケーションで DAX による読み取りレイテンシーが低い場合は、まず DAX をサポートするリージョンのリストを確認します。次に、DynamoDB テーブルのリージョンを選択します。

DAX キャッシュの動作

DAX はメタデータキャッシュおよびネガティブキャッシュを実行します。これらのキャッシュの動作を理解することで、キャッシュされた項目とネガティブキャッシュエントリの属性メタデータを効果的に管理できます。

- メタデータのキャッシュ - DAX クラスターは、キャッシュされた項目の属性名に関するメタデータを無期限に保持します。このメタデータは、項目の有効期限が切れたり、キャッシュから削除された後も保持されます。

  時間の経過とともに、大量の属性名を使用するアプリケーションは、DAX クラスターでメモリ消耗を引き起こすことがあります。この制限は最上位の属性名のみに適用され、ネスト属性名には適用されません。無制限の属性名の例には、タイムスタンプ、UUID や ID があります。タイムスタンプとセッション ID を属性値として使用することはできますが、より短く予測可能な属性名を使用することをお勧めします。
- ネガティブキャッシュ – キャッシュミスが発生し、DynamoDB テーブルからの読み取りで一致する項目がない場合、DAX は対応する項目キャッシュまたはクエリキャッシュにネガティブキャッシュエントリを追加します。このエントリは、キャッシュの TTL 期間が終了するか、書き込みスルーが発生するまで保持されます。DAX は、将来のリクエストに対して、このネガティブキャッシュエントリを返し続けます。

  ネガティブキャッシュの動作がアプリケーションパターンに合わない場合は、DAX が空の結果を返す際に DynamoDB テーブルを直接読み取ります。また、キャッシュ内の空の結果が長期間続くのを防ぎ、テーブルとの整合性を向上させるために、TTL キャッシュ期間を短く設定することをお勧めします。

クラスターのサイジング

DAX クラスターの合計キャパシティと可用性は、ノードのタイプと数によって異なります。クラスター内のノードが増えると、読み取りキャパシティは増加しますが、書き込みキャパシティは増加しません。より大きなノードタイプ (r5.8xlarge まで) では、より多くの書き込みを処理できます

が、ノードが少なすぎる場合は、ノードの障害が発生した際に可用性が影響を受ける可能性があります。DAX クラスターのサイジングの詳細については、「DAX クラスターサイジングガイド」を参照してください。

以下のセクションでは、スケーラブルでコスト効率の高いクラスターを作成する際、ノードタイプとノード数のバランスを取るために考慮すべきさまざまなサイジングの側面について説明します。

このセクションの内容

- 可用性の計画
- 書き込みスループットの計画
- 読み取りスループットの計画
- データセットサイズの計画
- クラスターの概算キャパシティ要件の計算
- ノードタイプ別のクラスタースループットキャパシティの概算
- DAX クラスターでの書き込みキャパシティのスケーリング

可用性の計画

DAX クラスターのサイズを設定する際は、目標とする可用性にフォーカスする必要があります。DAX などのクラスター化されたサービスの可用性は、クラスター内のノードの総数のディメンションです。単一ノードクラスターには障害に対する耐性がないため、その可用性は 1 つのノードに等しくなります。10 ノードのクラスターでは、1 つのノードが失われても、クラスターの全体的なキャパシティへの影響は最小限に抑えられます。残りのノードは引き続き読み取りリクエストを処理できるため、この損失は可用性に直接影響しません。書き込みを再開するために、DAX は新しいプライマリノードをすばやく指定します。

DAX は VPC ベースです。DAX はサブネットグループを使用して、ノードを実行できるアベイラビリティーゾーンを決定し、使用する IP アドレスをサブネットから決定します。本番ワークロードでは、異なるアベイラビリティーゾーンに少なくとも 3 つのノードがある DAX を使用することを強くお勧めします。そうすることで、1 つのノードまたはアベイラビリティーゾーンに障害が発生した場合でも、リクエストを処理するノードがクラスターに 1 つ以上残っていることが保証されます。クラスターには最大 11 個のノードを設定できます。1 個はプライマリノードで、残りの 10 個はリードレプリカです。

書き込みスループットの計画

すべての DAX クラスターには、書き込みスルーリクエスト用のプライマリノードがあります。その書き込みキャパシティは、クラスターのノードタイプのサイズによって決まります。リードレプリカを追加しても、クラスターの書き込みキャパシティは増えません。したがって、クラスターの作成時に書き込みキャパシティを考慮する必要があります。これは、後でノードタイプを変更することはできないためです。

項目キャッシュを更新するためにアプリケーションで書き込みスルー DAX が必要な場合は、書き込みを容易にするためにクラスターリソースの使用を増やすことを検討してください。DAX に対する書き込みは、キャッシュヒット読み取りの約 25 倍のリソースを消費します。このため、読み取り専用クラスターよりも大きなノードタイプが必要になる場合があります。

書き込みスルーまたは書き込みアラウンドがアプリケーションに最適かどうかを判断するための追加のガイダンスについては、「書き込みの方法」を参照してください。

読み取りスループットの計画

DAX クラスターの読み取りキャパシティは、ワークロードのキャッシュヒットレートによって異なります。キャッシュミスが発生すると DAX は DynamoDB からデータを読み取るため、キャッシュヒットの約 10 倍のクラスターリソースを消費します。キャッシュヒットを増やすには、キャッシュの TTL 設定を増やして、項目がキャッシュに保存される期間を定義します。ただし、TTL 期間を長くすると、DAX を介して更新が書き込まれない限り、古い項目バージョンを読み取る可能性が高くなります。

クラスターに十分な読み込みキャパシティがあることを確認するには、「クラスターの水平スケーリング」の説明に従ってクラスターを水平方向にスケーリングします。ノードを追加すると、リソースのプールにリードレプリカが追加され、ノードを削除すると、読み取りキャパシティが減少します。クラスターのノード数とそのサイズを選択する際は、必要な読み取りキャパシティの最小量と最大量の両方を考慮します。読み取り要件を満たすために、小さいノードタイプのクラスターを水平方向にスケーリングできない場合は、より大きいノードタイプを使用してください。

データセットサイズの計画

使用可能な各ノードタイプには、DAX がデータをキャッシュするためのメモリサイズが設定されています。ノードタイプが小さすぎる場合、アプリケーションがリクエストするワーキングデータセットがメモリに収まらず、キャッシュミスが発生します。大きなノードはより大きなキャッシュをサポートしているため、キャッシュが必要な想定されるデータセットよりも大きいノードタイプを使用してください。キャッシュを大きくすることで、キャッシュヒットレートも向上します。

読み取りがほとんど繰り返されない項目をキャッシュすると、リターンが減少する可能性があります。頻繁にアクセスされる項目のメモリサイズを計算し、キャッシュがそのデータセットを保存するのに十分な大きさであることを確認します。

クラスターの概算キャパシティ要件の計算

ワークロードの合計キャパシティを見積もって、クラスターノードの適切なサイズと量を選択することができます。キャパシティを見積もるには、1 秒あたりの正規化されたリクエスト (正規化された RPS) 変数を計算します。この変数は、キャッシュヒット、キャッシュミス、書き込みなど、アプリケーションが DAX クラスターでサポートする必要がある作業の合計単位を表します。正規化された RPS を計算するには、次の入力を使用します。

- `ReadRPS_CacheHit` – キャッシュヒットが発生する 1 秒あたりの読み取り数を指定します。
- `ReadRPS_CacheMiss` – キャッシュミスが発生する 1 秒あたりの読み取り数を指定します。
- `WriteRPS` – DAX を通過する 1 秒あたりの書き込み数を指定します。
- `DaxNodeCount` - DAX クラスターのノード数を指定します。
- `Size` – 書き込みまたは読み取り中の項目のサイズを、最も近い KB に切り上げて KB 単位で指定します。
- `10x_ReadMissFactor` – 値 10 を表します。キャッシュミスが発生すると、DAX はキャッシュヒットの約 10 倍のリソースを使用します。
- `25x_WriteFactor` – 値 25 を表します。これは、DAX 書き込みスルーはキャッシュヒットの約 25 倍のリソースを使用するためです。

次の式を使用して、正規化された RPS を計算します。

```
Normalized RPS = (ReadRPS_CacheHit * Size) + (ReadRPS_CacheMiss * Size *
 10x_ReadMissFactor) + (WriteRequestRate * 25x_WriteFactor * Size * DaxNodeCount)
```

例えば、以下の入力値を考えてみます。

- `ReadRPS_CacheHit` = 50,000
- `ReadRPS_CacheMiss` = 1,000
- `ReadMissFactor` = 1
- `Size` = 2 KB
- `WriteRPS` = 100
- `WriteFactor` = 1

- DaxNodeCount = 3

これらの値を計算式で使用することで、正規化された RPS を次のように計算できます。

```
Normalized RPS = (50,000 Cache Hits/Sec * 2KB) + (1,000 Cache Misses/Sec * 2KB * 10) +
 (100 Writes/Sec * 25 * 2KB * 3)
```

この例では、正規化された RPS の計算値は 135,000 です。ただし、この正規化された RPS 値では、クラスター使用率を 100% 未満に抑えること、またはノード損失が考慮されていません。追加の容量を考慮することをお勧めします。これを行うには、目標使用率とノード損失耐性の 2 つの乗算係数のうち大きい方を決定します。次に、正規化された RPS に大きい方の係数を掛けて、1 秒あたりのターゲットリクエスト (ターゲット RPS) を取得します。

- 目標使用率

  パフォーマンスへの影響によりキャッシュミスが増加するため、DAX クラスターを 100% の使用率で実行することはお勧めしません。理想的には、クラスター使用率を 70% 以下に維持する必要があります。これを実現するには、正規化された RPS に 1.43 を掛けます。

- ノード損失耐性

  ノードに障害が発生した場合、アプリケーションは残りのノード間でリクエストを分散処理する必要があります。ノードの使用率を 100% 未満に維持するには、障害が発生したノードがオンラインに戻るまで、障害によって増加したトラフィックを吸収するのに十分な大きさのノードタイプを選択します。ノード数が少ないクラスターの場合、1 つのノードに障害が発生した場合、各ノードはより大きなトラフィックの増加を許容する必要があります。

  プライマリノードに障害が発生した場合、DAX は自動的にリードレプリカにフェイルオーバーし、新しいプライマリとして指定します。レプリカノードに障害が発生した場合、障害が発生したノードが回復するまで、DAX クラスター内の他のノードがリクエストを処理します。

  例えば、ノード障害のある 3 ノード DAX クラスターでは、残りの 2 つのノードで追加の 50% の容量が必要です。これには 1.5 の乗算係数が必要です。また、障害が発生したノードを持つ 11 ノードクラスターでは、残りのノードに 10% の容量を追加するか、1.1 の乗算係数が必要です。

次の式を使用して、ターゲット RPS を計算します。

```
Target RPS = Normalized RPS * CEILING(TargetUtilization, NodeLossTolerance)
```

例えば、ターゲット RPS の計算で、以下の値を考えてみます。

- `Normalized RPS` = 135,000
- `TargetUtilization` = 1.43

  クラスターの最大使用率を 70% にすることをターゲットとしているため、`TargetUtilization` を 1.43 に設定します。
- `NodeLossTolerance` = 1.5

  3 ノードクラスターを使用するため、`NodeLossTolerance` を 50% キャパシティに設定します。

これらの値を計算式に置き換えることで、ターゲット RPS を次のように計算できます。

```
Target RPS = 135,000 * CEILING(1.43, 1.5)
```

この例では、`NodeLossTolerance` の値が `TargetUtilization` より大きいため、`NodeLossTolerance` を使用してターゲット RPS の値を計算します。計算の結果、ターゲット RPS は 202,500 になります。これが、DAX クラスターがサポートする必要があるキャパシティの合計です。クラスターで必要なノードの数を決定するには、ターゲット RPS を次の表の適切な列にマッピングします。ターゲット RPS が 202,500 の場合、3 つのノードを持つ dax.r5.large ノードタイプが必要です。

ノードタイプ別のクラスタースループットキャパシティの概算

Target RPS formula を使用すると、さまざまなノードタイプのクラスターキャパシティを見積もることができます。次の表は、1、3、5、11 のノードタイプのクラスターのおおよそのキャパシティを示しています。これらのキャパシティは、独自のデータおよびリクエストパターンを使用した DAX の負荷テストの必要性を置き換えるものではありません。さらに、固定 CPU キャパシティがないため、これらのキャパシティには t タイプインスタンスは含まれません。次の表のすべての値の単位は正規化された RPS です。

ノードタイプ (メモリ)	1 ノード	3 ノード	5 ノード	11 ノード
dax.r5.24xlarge (768GB)	100 万回	3M	5M	11M

ノードタイプ (メモリ)	1 ノード	3 ノード	5 ノード	11 ノード
dax.r5.16xlarge (512GB)	100 万回	3M	5M	11M
dax.r5.12xlarge (384GB)	100 万回	3M	5M	11M
dax.r5.8xlarge (256GB)	100 万回	3M	5M	11M
dax.r5.4xlarge (128GB)	600k	1.8M	3M	6.6M
dax.r5.2xlarge (64GB)	300k	900k	1.5M	3.3M
dax.r5.xlarge (32GB)	150k	450k	750k	1.65M
dax.r5.large (16GB)	75k	225k	375k	825k

各ノードの最大制限は 100 万 NPS (1 秒あたりのネットワークオペレーション) であるため、dax.r5.8xlarge 以上のノードタイプによってクラスターキャパシティが増えることはありません。8xlarge を超えるノードタイプは、クラスターの合計スループットキャパシティに影響しない可能性があります。ただし、このようなノードタイプは、より大きなワーキングデータセットをメモリに保存する場合に役立ちます。

DAX クラスターでの書き込みキャパシティのスケーリング

DAX への書き込みごとに、各ノードで 25 個の正規化されたリクエストが消費されます。各ノードには 100 万 RPS の制限があるため、DAX クラスターの書き込みは 1 秒あたり 40,000 回に制限されます (読み取り使用量は考慮されません)。

ユースケースでキャッシュに 1 秒あたり 40,000 回以上の書き込みが必要な場合は、別の DAX クラスターを使用し、クラスター間で書き込みをシャードする必要があります。DynamoDB と同様に、

キャッシュに書き込むデータのパーティションキーをハッシュできます。次に、モジュラスを使用して、データを書き込むシャードを決定します。

次の例では、入力文字列のハッシュを計算します。その後、ハッシュ値のモジュラスを 10 で計算します。

```
def hash_modulo(input_string):
    # Compute the hash of the input string
    hash_value = hash(input_string)

    # Compute the modulus of the hash value with 10
    bucket_number = hash_value % 10

    return bucket_number

#Example usage
if _name_ == "_main_":
    input_string = input("Enter a string: ")
    result = hash_modulo(input_string)
    print(f"The hash modulo 10 of '{input_string}' is: {result}.")
```

クラスターのデプロイ

新しい DAX クラスターを作成するには、DynamoDB で必要な設定よりも多くの設定が必要です。これらの設定は、特にネットワークに関するもので、これは DAX が Amazon VPC に基づいているためです。そのため、リソースの配置、接続、セキュリティなど、仮想ネットワーク環境を完全に制御できます。このセクションでは、クラスターの作成に必要な設定のベストプラクティスについて説明します。

クラスターノードの選択の詳細については、「クラスターのサイジング」を参照してください。

このセクションの内容

- ネットワークの設定
- セキュリティの設定
- パラメータグループ
- メンテナンスウィンドウ

ネットワークの設定

DAX はサブネットグループを使用して、ノードを実行できるアベイラビリティーゾーンを決定し、使用する IP アドレスをサブネットから決定します。アプリケーションと DAX 間のレイテンシーを最小限に抑えるには、アプリケーションサーバーと DAX クラスターのサブネットとアベイラビリティーゾーンが同じである必要があります。

DAX ノードは、複数のアベイラビリティーゾーンに分散することをお勧めします。自動割り当てのデフォルトのオプションを使用することで、ノードを分散することができます。

VPC のセットアップのベストプラクティスについては、「Amazon VPC ユーザーガイド」の「Amazon VPC の使用を開始する」を参照してください。

セキュリティの設定

このセクションでは、DAX を使用するアプリケーションで実装が必要なセキュリティ対策について説明します。このセクションでは、DAX によるデータ暗号化のサポートについても簡単に説明します。

IAM

DAX と DynamoDB には、個別のアクセスコントロールの仕組みがあります。DAX では、DynamoDB テーブルにアクセスするための IAM ロールが必要です。このロールは最小特権の原則に従い、特定のテーブルと DynamoDB オペレーション (GetItem や PutItem など) にのみアクセスを許可する必要があります。DAX が提供するアクセスコントロールの仕組みの詳細については、「DAX のアクセスコントロール」を参照してください。

暗号化

DAX クラスターの作成時に、保管時の暗号化と転送中の暗号化を設定します。これらは、デフォルトで有効化されています。ビジネス要件に基づく無効化が必要な場合を除き、デフォルトの暗号化設定を使用することをお勧めします。詳細については、「保管時の DAX 暗号化」および「転送時の DAX 暗号化」を参照してください。

パラメータグループ

DAX は、クラスター内のすべてのノードにパラメータグループと呼ばれる一連の設定を適用します。この設定は、クラスターの作成後に変更できます。

DAX のパラメータグループは、項目キャッシュとクエリキャッシュの TTL 設定を保持します。デフォルトの TTL 期間は 5 分間です。TTL 期間は、1 ミリ秒以上の任意の整数値に変更することができます。

実行中の DAX インスタンスがパラメータグループを使用している場合、パラメータグループを変更することはできません。パラメータグループの値は、DAX クラスターのダウンタイム中に変更できます。

メンテナンスウィンドウ

ノードへのソフトウェアのアップグレードやパッチ適用を許可するために、DAX クラスターに毎週のメンテナンスウィンドウが設定されます。このウィンドウ中に、DAX はノードにローリング更新を実行します。複数のノードを持つクラスターは、これらの更新中にクラスターの可用性は失われませんが、ノードが再度使用可能になるまでクラスターキャパシティが減少します。組織で使用量が少ない時間帯予測できる場合は、メンテナンスウィンドウをその時間帯に手動で設定することを検討してください。

クラスターオペレーションの管理

DAX は、クラスターのメンテナンスと状態をお客様に代わって管理します。ただし、使用パターンに合わせてクラスターを水平または垂直にスケーリングするには、運用上の入力を提供する必要があります。このセクションでは、DAX クラスターをスケーリングするための推奨されるプロセスについて説明します。

このセクションの内容

- クラスターの水平スケーリング
- クラスターの垂直スケーリング

クラスターの水平スケーリング

DAX クラスターをスケーリングするには、スループットの需要に合わせてキャパシティを調整する必要があります。この調整は、実行中のクラスター内のノード (レプリカ) の数を増減することによって行われます。このプロセスは、水平スケーリングと呼ばれ、需要が低い場合にワークロードをより多くのノードに分散したり、少数のノードに統合したりするのに役立ちます。

AWS CLI の `decrease-replication-factor` コマンドまたは `increase-replication-factor` コマンドを使用して、DAX クラスターを水平方向にスケールインおよびスケールアウトできます。

レプリケーション係数を増やす (スケールアウト)

DAX クラスターのレプリケーション係数を増やすと、クラスターにノードが追加されます。次の例は、`increase-replication-factor` コマンドの使用方法を示しています。

```
aws dax increase-replication-factor \
    --cluster-name yourClusterName  \
    --new-replication-factor desiredReplicationFactor
```

- このコマンドでは、cluster-name 引数はクラスター名を指定します。例えば、*yourClusterName* です。
- new-replication-factor 引数は、スケーリング後のクラスター内の合計ノード数を指定します。これには、プライマリノードとレプリカノードが含まれます。例えば、クラスターに現在 3 つのノードがあり、さらに 2 つのノードを追加する場合は、new-replication-factor の値を 5 に設定します。

レプリケーション係数を減らす (スケールイン)

DAX クラスターのレプリケーション係数を減らすと、クラスターからノードが削除されます。ノードを削除すると、需要の低い時間帯のコストを削減できます。次の例は、decrease-replication-factor コマンドの使用方法を示しています。

```
aws dax decrease-replication-factor \
    --cluster-name yourClusterName  \
    --new-replication-factor desiredReplicationFactor
```

- このコマンドでは、cluster-name 引数はクラスター名を指定します。例えば、*yourClusterName* です。
- new-replication-factor 引数は、スケーリング後のクラスター内のノード数を指定します。この数は、現在のレプリケーション係数よりも小さく、プライマリノードを含める必要があります。例えば、クラスターに現在 5 つのノードがあり、2 つのノードを削除する場合は、new-replication-factor の値を 3 に設定します。

水平スケーリングに関する考慮事項

水平スケーリングを計画する場合は、以下の点を考慮します。

- プライマリノード – DAX クラスターにはプライマリノードが含まれます。レプリケーション係数には、プライマリノードが含まれます。例えば、レプリケーション係数 3 は、1 つのプライマリノードと 2 つのレプリカノードを意味します。
- 可用性 — DAX ノードを追加または削除すると、クラスターの可用性と耐障害性が変わります。ノードが増えると可用性は向上しますが、コストも増加します。

- データ移行 – レプリケーション係数を増やすと、DAX は新しいノードセット全体のデータ分散を自動的に処理します。新しいノードがトラフィックの処理を開始する際、そのキャッシュは既にウォームアップされています。ただし、このプロセスは、データ移行中のパフォーマンスに一時的な影響を与える可能性があります。

スケーリングプロセス中およびスケーリングプロセス後に DAX クラスターを注意深くモニタリングし、クラスターが期待どおりに動作していることを確認し、必要に応じて調整を行います。

クラスターの垂直スケーリング

既存のクラスターのノードサイズを垂直方向にスケーリングするには、新しいクラスターを作成し、アプリケーショントラフィックを新しいクラスターに移行する必要があります。異なるノードを持つ新しいクラスターに移行するには、アプリケーションのパフォーマンスと可用性への影響を最小限に抑えながらスムーズに移行できるようにするためのいくつかのステップが必要です。

ノードサイズを垂直方向にスケーリングするための新しいクラスターを作成するには、以下の点を考慮します。

- 現在のセットアップを確認する – 現在の DAX クラスターのメトリクスを確認して、必要な新しいノードサイズと数を決定します。この情報を入力として使用して、クラスターサイズを定義します。詳細については、「クラスターのサイジング」を参照してください。
- 新しい DAX クラスターをセットアップする – 決定したノードタイプと数で新しい DAX クラスターを作成します。調整が必要な場合を除き、パラメータグループの既存の設定を使用できます。
- データを同期する — DAX は DynamoDB のキャッシュレイヤーであるため、データを直接移行する必要はありません。ただし、トラフィックを送信するまで、新しい DAX クラスターにはワーキングデータセットはメモリに存在しません。
- アプリケーション設定を更新する – 新しい DAX クラスターのエンドポイント を指すようにアプリケーションの設定を更新します。アプリケーションの設定によっては、コードの変更や環境変数の更新が必要になる場合があります。

  新しいクラスターに切り替える際の影響を軽減するには、アプリケーションフリートのごく一部から新しいクラスターに Canary トラフィックを送信します。これは、アプリケーションの更新を徐々にロールアウトするか、DAX エンドポイントの前に重みベースのルーティング DNS エントリを使用することで実行できます。
- モニタリングと最適化 — 新しい DAX クラスターに切り替えたら、パフォーマンスのメトリクスとログに問題がないか注意深くモニタリングします。更新されたワークロードパターンに基づいてノード数を調整する準備を行います。

新しいクラスターがワーキングデータセットを適切にキャッシュするまで、キャッシュミス率とレイテンシーは高くなります。

- 古いクラスターの廃止 – 新しいクラスターが想定どおりに動作していることが確実な場合は、不要なコストが発生しないように、古い DAX クラスターを安全に廃止してください。

DAX のモニタリング

キャッシュヒットレートなどの主要なメトリクスをモニタリングして、最適な DAX クラスターのパフォーマンスを確保し、問題を診断して、クラスターのスケールが必要なタイミングを決定します。主要なメトリクスを定期的にチェックすることで、ワークロードの要件に合わせてクラスターをスケールすることで、パフォーマンス、安定性、コスト効率を維持できます。DAX のモニタリングの詳細については、「本番モニタリング」を参照してください。

次のリストは、モニタリングが必要な主要なメトリクスの一部を示しています。

- キャッシュヒットレート — キャッシュされたデータを DAX がどの程度効果的に処理しているかを示し、基盤となる DynamoDB テーブルにアクセスする必要性を減らします。クラスターのキャッシュミスが少ないほど、キャッシュ効率が高いことを示します。キャッシュヒットが少ない場合、キャッシュの TTL 設定を再確認する必要があるか、ワークロードがキャッシュに適していないことを示します。

  Amazon CloudWatch を使用して、DAX クラスターのキャッシュヒットレートを計算します。`ItemCacheHits`、`ItemCacheMisses`、`QueryCacheHits`、および `QueryCacheMisses` の各メトリクスを比較して、この比率を取得します。次の式は、キャッシュヒットレートの計算方法を示します。この式を使用してヒット率を計算するには、キャッシュヒットをキャッシュヒットとキャッシュミスの合計で割ります。

  ```
  Cache hit ratio = Cache hits / (Cache hits + Cache misses)
  ```

  キャッシュヒットレートは 0 ~ 1 の数値になり、パーセンテージで表されます。パーセンテージが高いほど、全体的なキャッシュ使用率が高いことを示します。

- ErrorRequestCount – ノードまたはクラスターによって報告されたユーザーエラーの原因となったリクエストの数。`ErrorRequestCount` には、ノードまたはクラスターによってスロットリングされたリクエストが含まれます。ユーザーエラーのモニタリングは、アプリケーションのスケーリング設定ミスやホットアイテム/パーティションパターンを特定するのに役立ちます。

- オペレーションのレイテンシー — DAX クラスターとの間で送受信されるオペレーションのレイテンシーをモニタリングすることで、パフォーマンスのボトルネックの特定に役立ちます。レイテンシーの増加は、DAX クラスターの設定、ネットワーク、またはスケールの必要性に問題がある可能性を示します。
- ネットワーク消費 — `NetworkBytesIn` および `NetworkBytesOut` メトリクスに注目して、DAX クラスターのネットワークトラフィックをモニタリングします。ネットワークスループットが予期せず増加すると、クライアントリクエストが増えたり、クエリパターンが非効率になり、より多くのデータが転送される可能性があります。

  ネットワーク消費量のモニタリングは、DAX クラスターのコスト管理に役立ちます。また、ネットワークがクラスターのパフォーマンスのボトルネックにならないようにします。
- エビクション率 — 新しい項目を格納するためにキャッシュから項目が削除される頻度を示します。時間の経過に伴うエビクション率の増加は、キャッシュが小さすぎるか、キャッシュ戦略が有効でない可能性を示します。

  CloudWatch で `EvictedSize` メトリクスをモニタリングして、キャッシュサイズがワークロードに適しているかどうかを判断します。合計エビクションサイズが増加し続ける場合は、より大きなキャッシュに対応するために DAX クラスターをスケールアップする必要がある場合があります。
- CPU 使用率 - ノードまたはクラスターの CPU 使用率の割合を示します。これは、データベースまたはキャッシュシステムをモニタリングするための重要なメトリクスです。CPU 使用率が高いと、DAX クラスターが過負荷になり、需要の増加に対応するためにスケーリングが必要になる可能性があります。

  DAX クラスターの `CPUUtilization` メトリクスのモニタリング。CPU 使用率が常に 70～80% に近づいている場合は、次のセクションで説明するように DAX クラスターをスケールアップすることを検討してください。

  DAX に送信されたリクエストの数がノードのキャパシティを超える場合、DAX は 追加のリクエストの受け入れ率を制限します。これは、ThrottlingException を返すことによって行われます。DAX はクラスターの CPU 使用率を継続的に評価し、正常なクラスター状態を維持しながら処理できるリクエスト量を決定します。

  DAX が CloudWatch に公開する `ThrottledRequestCount` メトリクスをモニタリングできます。これらの例外が定期的に発生する場合は、クラスターをスケールアップすることを検討してください。

モニタリングデータを使用した DAX クラスターのスケール

パフォーマンスメトリクスをモニタリングすることで、DAX クラスターをスケールアップまたはスケールダウンする必要があるかどうかを判断できます。

- スケールアップまたはスケールアウト — DAX クラスターの CPU 使用率が高い場合、(キャッシュ戦略の最適化後に) キャッシュヒットが低い場合、またはオペレーションのレイテンシーが高い場合は、クラスターをスケールアップする必要があります。ノードをさらに追加すると (スケールアウトとも呼ばれる)、負荷をより均等に分散できます。1 秒あたりの書き込み数が増加するワークロードでは、より強力なノードを選択する必要がある場合があります (スケールアップ)。
- スケールダウン — CPU 使用率とオペレーションのレイテンシーが常にしきい値を下回る場合は、リソースが過剰にプロビジョニングされている可能性があります。このような場合は、ノードをスケールダウンしてコストを削減します。使用率が低い時間帯にノードの数を 1 に減らすことができますが、クラスターを完全にシャットダウンすることはできません。

DynamoDB での PITR 復旧のベストプラクティス

ポイントインタイムリカバリ (PITR) を使用してテーブルを以前の状態に戻すためのベストプラクティスを以下に示します。

元に戻したいテーブルへの誤った書き込みに気付いた場合は、これらのベストプラクティスを使用してください。テーブル全体を特定の時点から復元するか、特定の不要な書き込みをインプレースでロールバックできます。

トピック

- テーブルの復元を開始して復旧する
- 不要な書き込みをインプレースでロールバックして復旧する

テーブルの復元を開始して復旧する

PITR を使用すると、PITR 復旧期間内の特定の秒にテーブルを復元できます。PITR 復旧期間は、PITR が有効になったときに開始され、最大 35 日間の履歴が保持されます。より短い期間に設定することもできます。

PITR テーブルの完全復元を開始するときは、次の点を考慮してください。

- 復元により、新しいテーブルが作成されます。テーブル復元アクションは、元のテーブルに対してインプレースでは動作しません。インプレースオプションについては、「不要な書き込みをインプレースでロールバックして復旧する」を参照してください。
- 同じリージョンの新しいテーブルまたは別のリージョンに復元できます。別のリージョンに復元すると、データ転送料金が発生します。
- セカンダリインデックスの有無にかかわらず、テーブルを復元できます。セカンダリインデックスなしで復元すると、より高速でコスト効率が良くなります。
- 復元が完了するには数時間かかる場合があります。復元時間は複数の要因に基づいて異なり、必ずしもテーブルサイズと相関するとは限りません。

PITR テーブルの復元後、新しいテーブルを元のテーブルの完全な置き換えとして運用可能にすることができます。また、元のテーブルと比較して違いを見つけることもできます。より簡単なアプローチについては、「不要な書き込みをインプレースでロールバックして復旧する」を参照してください。

復元されたテーブルを運用可能にするときは、次の点を考慮してください。

- Infrastructure as Code 環境 (AWS CloudFormation、AWS Cloud Development Kit (AWS CDK)、Terraform) では、復元されたテーブルを IaC 環境に導入して管理します。
- クライアント接続を更新して、新しいテーブルを新しい名前で参照します。
- ストリーム設定が元のテーブルにあった場合は、再追加します。DynamoDB ストリーム名にはストリームの作成時刻が含まれているため、新しいストリームは古いストリームとは異なる名前になります。IAM またはダウンストリームコンシューマーコードで古いストリーム名への参照を更新します。
- PITR が元のテーブルにあった場合は、新しいテーブルで PITR を再度有効にします。
- 削除保護が元のテーブルにあった場合は、新しいテーブルで削除保護を再度有効にします。
- タグが元のテーブルにあった場合は、新しいテーブルにタグを再追加します。
- RBP が元のテーブルにあった場合は、新しいテーブルにリソースベースのポリシーを再追加します。
- 元のテーブルを参照するクロスアカウントアクセス IAM ポリシーを更新します。
- 新しいテーブルの自動スケーリング設定を調整します。
- TTL が元のテーブルにあった場合は、新しいテーブルに TTL 属性を設定します。
- カスタムテーブルクラスが元のテーブルにあった場合は、新しいテーブルのテーブルクラスを調整します。

- 新しいテーブルのウォームスループット設定を確認し、必要に応じて読み取りと書き込みの値を増やします。
- 元のテーブルがグローバルテーブルであった場合、そのテーブルと一致するように新しいテーブルのレプリカを追加します。
- 復元されたテーブルが完全に設定され、運用可能になったら、元のテーブルを削除します。

MRSC グローバルテーブルを復元する場合、MRSC テーブルには空のテーブルのみを作成できるため、新しく復元されたテーブルを MRSC テーブルにすることはできません。解決策の 1 つは、テーブルを復元し、復元されたテーブルから空の MRSC テーブルにデータをコピーすることです。

不要な書き込みをインプレースでロールバックして復旧する

元のテーブルを運用可能に保つためには、テーブル全体を復元するのではなく、不要な書き込みを直接ロールバックできます。すべての書き込みをロールバックすることも、間違いとして識別した書き込みのみをロールバックすることもできます。

開始する前に、PITR がテーブルで有効になっており、不要な書き込みの発生した期間が PITR 復旧期間内であることを確認してください。

このアプローチでは、Amazon Simple Storage Service (Amazon S3) への増分エクスポート機能を使用します。増分エクスポートでは、変更された項目を Amazon S3 に書き込みます。エクスポートには、各項目の古いイメージと新しいイメージが含まれます。期間は最短で 15 分、最長で 24 時間です。期間は PITR 復旧期間に含まれている必要があります。

不要な書き込みをロールバックするには

1. Amazon S3 への増分エクスポートを実行します。テーブルに対して誤った書き込みが発生した期間を指定します。「新しいイメージと古いイメージ」のエクスポートビューを選択します。期間が 24 時間を超える場合は、全期間を正確にカバーする複数の増分エクスポートジョブを実行します。
2. 古い項目値を書き戻すことで、Amazon S3 データから一括ジョブを実行し、エクスポートにリストされている変更を元に戻します。必要に応じて変換を適用して、元に戻される内容と方法を正確に制御します (項目の小さな間違いを修正するなど)。

この一括ジョブを実行するには、オープンソースの DynamoDB 用 Bulk Executor を使用できます。ツールの詳細については、AWS Database Blog の「Introducing open source Bulk Executor for Amazon DynamoDB」を参照してください。この方法で不要な書き込みをロールバックする手順の

例については、AWS Database Blog の「Recover from accidental DynamoDB changes using Bulk Executor」を参照してください。Bulk Executor はオープンソースであり、公式サポートは提供されていません。

このアプローチには以下の利点があります。

- 特に大きなテーブルの場合、コスト効率が良くなります。テーブル全体を復元して少数の誤った書き込みを元に戻すのではなく、間違いだけに注目します。
- 特に大きなテーブルの場合、より高速になります。テーブルデータのサブセットのみを処理する必要があります。
- ライブのインプレース修正。修正中、元のテーブルはアクティブなままです。新しいテーブルのメタデータ、設定、外部参照を調整する必要はありません。

DynamoDB を他の AWS サービスで使用する

Amazon DynamoDB を他の AWS サービスと統合することで、繰り返しのタスクを自動化したり、複数のサービスにまたがるアプリケーションを構築できます。

トピック

- DynamoDB の Amazon Cognito を使用した AWS 認証情報の設定
- Amazon Redshift との統合
- Amazon EMR での Apache Hive を使用した DynamoDB データの処理
- DynamoDB の Amazon S3 との統合
- Amazon SageMaker Lakehouse との DynamoDB ゼロ ETL 統合
- DynamoDB と Amazon OpenSearch Service のゼロ ETL 統合
- DynamoDB と Amazon EventBridge の統合
- DynamoDB と Amazon Managed Streaming for Apache Kafka の統合
- DynamoDB との統合に関するベストプラクティス

DynamoDB の Amazon Cognito を使用した AWS 認証情報の設定

ウェブおよびモバイルアプリケーション用の AWS 認証情報を取得する際には、Amazon Cognito を使用することをお勧めします。Amazon Cognito を使用すると、AWS 認証情報をファイル上でハードコーディングする必要がありません。AWS Identity and Access Management (IAM) ロールを使用して、アプリケーションの認証されたユーザーと認証されていないユーザーのために一時的な認証情報が生成されます。

たとえば、Amazon Cognito の認証されていないロールを使用して Amazon DynamoDB ウェブサービスにアクセスするように JavaScript ファイルを設定するには、次の手順を実行します。

Amazon Cognito と統合する認証情報を設定するには

1. 認証されていない ID を許可する Amazon Cognito ID プールを作成します。

```
aws cognito-identity create-identity-pool \
    --identity-pool-name DynamoPool \
    --allow-unauthenticated-identities \
    --output json
```

```
{
    "IdentityPoolId": "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
    "AllowUnauthenticatedIdentities": true,
    "IdentityPoolName": "DynamoPool"
}
```

2. 次のポリシーを myCognitoPolicy.json という名前のファイルにコピーします。ID プール ID (*us-west-2:12345678-1ab2-123a-1234-a12345ab12*) を、前のステップで取得した独自の IdentityPoolId に置き換えます。

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Federated": "cognito-identity.amazonaws.com"
      },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "cognito-identity.amazonaws.com:aud": "us-
west-2:12345678-1ab2-123a-1234-a12345ab12"
        },
        "ForAnyValue:StringLike": {
          "cognito-identity.amazonaws.com:amr": "unauthenticated"
        }
      }
    }
  ]
}
```

3. 以前のポリシーを予期する IAM ロールを作成します。このようにして、Amazon Cognito は Cognito_DynamoPoolUnauth ロールを引き受けられる信頼済みエンティティになります。

```
aws iam create-role --role-name Cognito_DynamoPoolUnauth \
--assume-role-policy-document file://PathToFile/myCognitoPolicy.json --output json
```

4. 管理ポリシー (AmazonDynamoDBReadOnlyAccess) をアタッチして、DynamoDB への読み取り専用アクセスを Cognito_DynamoPoolUnauth ロールに許可します。このロールは認証され

ていないユーザーによって引き受けられるため、最小特権の原則に従い、ロールが実際に必要とするアクセス許可のみを付与します。

```
aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/
AmazonDynamoDBReadOnlyAccess \
--role-name Cognito_DynamoPoolUnauth
```

> **Note**
>
> または、DynamoDB へのきめ細かなアクセスを許可することもできます。詳細については、「詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用」を参照してください。

5. IAM ロールの Amazon リソースネーム (ARN) を取得してコピーします。

```
aws iam get-role --role-name Cognito_DynamoPoolUnauth --output json
```

6. DynamoPool ID プールに Cognito_DynamoPoolUnauth ロールを追加します。指定するフォーマットは KeyName=string です。ここで KeyName は unauthenticated で、文字列は前のステップで取得したロールの ARN です。

```
aws cognito-identity set-identity-pool-roles \
--identity-pool-id "us-west-2:12345678-1ab2-123a-1234-a12345ab12" \
--roles unauthenticated=arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth --
output json
```

7. ファイルで Amazon Cognito 認証情報を指定します。それに応じて IdentityPoolId および RoleArn を変更します。

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
IdentityPoolId: "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
RoleArn: "arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth"
});
```

Amazon Cognito 認証情報を使用して、DynamoDB ウェブサービスに対して JavaScript プログラムを実行できます。詳細については、「AWS SDK for JavaScript 入門ガイド」の「ウェブブラウザ内で認証情報を設定する」を参照してください。

Amazon Redshift との統合

Amazon Redshift は、高速でフルマネージドのペタバイト規模のデータウェアハウスサービスで、既存のビジネスインテリジェンスツールを使用してすべてのデータを簡単かつコスト効率よく効率的に分析できます。

DynamoDB と Amazon Redshift を組み合わせて使用することで、アプリケーションやデータエコシステム内のさまざまなデータストレージおよび処理ニーズに対応することができます。

DynamoDB と Amazon Redshift を統合する方法の詳細については、以下のトピックを参照してください。

トピック

- CMK を使用したクロスアカウント統合に関する考慮事項
- DynamoDB の Amazon Redshift とのゼロ ETL 統合
- COPY コマンドを使用した、DynamoDB から Amazon Redshift へのデータのロード

CMK を使用したクロスアカウント統合に関する考慮事項

DynamoDB から Amazon Redshift に統合しようとすると、最初のアクションが Amazon Redshift から起動されます。適切なアクセス許可がないと、このアクションはサイレントに失敗する可能性があります。以下のセクションでは、このクロスアカウント統合に必要なアクセス許可について詳しく説明します。

必要な AWS KMS ポリシーとアクセス許可

例の中で以下のプレースホルダーを置き換えます。

- `111122223333`: Amazon Redshift がホストされている AWS アカウント ID
- `444455556666`: DynamoDB がホストされている AWS アカウント ID
- `REDSHIFT_ROLE_NAME`: Amazon Redshift で使用されている IAM ロール名
- `REGION`: リソースが存在している AWS リージョン
- `TABLE_NAME`: DynamoDB テーブルの名前
- `KMS_KEY_ID`: KMS キーの ID

DynamoDB アカウントの KMS キーポリシー

次の AWS KMS キーポリシーは、DynamoDB と Amazon Redshift サービス間のクロスアカウントアクセスを有効にします。この例では、アカウント 444455556666 に DynamoDB テーブルと AWS KMS キーが含まれ、アカウント 111122223333 にデータの復号にアクセスする必要がある Amazon Redshift クラスターが含まれています。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Enable IAM User Permissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow Redshift to use the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/REDSHIFT_ROLE_NAME"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey",
                "kms:GenerateDataKeyWithoutPlaintext"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Redshift ロールの IAM ポリシー (Amazon Redshift アカウント内)

次の IAM ポリシーは、Amazon Redshift サービスがクロスアカウントシナリオで DynamoDB テーブルおよび関連する AWS KMS 暗号化キーにアクセスすることを許可します。この例では、アカウント 444455556666 には、Amazon Redshift サービスがアクセスする必要がある DynamoDB リソースと AWS KMS キーが含まれています。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDynamoDBAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME",
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME/stream/*"
            ]
        },
        {
            "Sid": "AllowKMSAccess",
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey",
                "kms:GenerateDataKeyWithoutPlaintext"
            ],
            "Resource": "arn:aws:kms:us-east-1:444455556666:key/KMS_KEY_ID"
```

```
        }
    ]
}
```

Amazon Redshift ロールの信頼関係

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "redshift.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

DynamoDB テーブルポリシー (リソースベースのポリシーを使用している場合)

次のリソースベースのポリシーは、アカウント 111122223333 の Amazon Redshift サービスがアカウント 444455556666 の DynamoDB テーブルとストリームにアクセスすることを許可します。このポリシーを DynamoDB テーブルにアタッチして、クロスアカウントアクセスを有効化します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowRedshiftAccess",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/REDSHIFT_ROLE_NAME"
            },
            "Action": [
```

```
                "dynamodb:DescribeTable",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:BatchGetItem",
                "dynamodb:GetItem",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb:ListStreams"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME",
                "arn:aws:dynamodb:*:444455556666:table/TABLE_NAME/stream/*"
            ]
        }
    ]
}
```

重要な考慮事項

1. KMS キーが DynamoDB テーブルと同じリージョンにあることを確認します。
2. KMS キーは、AWS マネージドキーではなく、カスタマーマネージドキー (CMK) である必要があります。
3. DynamoDB グローバルテーブルを使用している場合は、関連するすべてのリージョンのアクセス許可を設定します。
4. VPC エンドポイントまたは IP 範囲に基づいてアクセスを制限する条件ステートメントを追加することを検討します。
5. セキュリティを強化するには、`aws:PrincipalOrgID` 条件を使用して組織へのアクセスを制限することを検討します。
6. CloudTrail メトリクスと CloudWatch メトリクスを通じて KMS キーの使用状況をモニタリングします。

DynamoDB の Amazon Redshift とのゼロ ETL 統合

Amazon DynamoDB と Amazon Redshift のゼロ ETL 統合により、コーディングなしで DynamoDB のデータのシームレスな分析が可能になります。この完全マネージド型の機能により、DynamoDB テーブルが自動的に Amazon Redshift データベースにレプリケートされるため、複雑な ETL プロセ

スを設定することなく、DynamoDB のデータに対して SQL クエリや分析を実行することができます。統合は、DynamoDB テーブルから Amazon Redshift データベースにデータをレプリケートすることで機能します。

統合を設定するには、ソースとして DynamoDB テーブル、ターゲットとして Amazon Redshift データベースを指定するだけです。有効化すると、この統合機能は DynamoDB テーブル全体を Amazon Redshift データベースにエクスポートします。この初期プロセスが完了するまでにかかる時間は、DynamoDB テーブルのサイズによって異なります。ゼロ ETL 統合により、その後は DynamoDB から Amazon Redshift への更新が、DynamoDB の増分エクスポートを使用して15 ~ 30 分ごとに段階的にレプリケートされます。つまり、Amazon Redshift にレプリケートされる DynamoDB データは、自動的に最新に保たれます。

設定が完了すると、DynamoDB テーブルのパフォーマンスに影響を与えることなく、標準的な SQL クライアントやツールを使用して Amazon Redshift 内の DynamoDB データを分析できるようになります。このゼロ ETL 統合により煩雑な ETL プロセスを排除し、Amazon Redshift の分析機能や機械学習機能を活用して、DynamoDB から迅速かつ簡単にインサイトを引き出すことができます。

トピック

- DynamoDB の Amazon Redshift とのゼロ ETL 統合を作成する前の前提条件
- DynamoDB の Amazon Redshift とのゼロ ETL 統合を使用する場合の制約事項
- DynamoDB の Amazon Redshift とのゼロ ETL 統合の作成
- DynamoDB の Amazon Redshift とのゼロ ETL 統合の表示
- DynamoDB の Amazon Redshift とのゼロ ETL 統合の削除

DynamoDB の Amazon Redshift とのゼロ ETL 統合を作成する前の前提条件

1. 統合を作成する前に、ソースの DynamoDB テーブルとターゲットの Amazon Redshift クラスターを作成しておく必要があります。この情報は、 ステップ 1: ソース DynamoDB テーブルを設定する および ステップ 2: Amazon Redshift データウェアハウスを作成する で説明されています。
2. Amazon DynamoDB と Amazon Redshift 間のゼロ ETL 統合を使用するには、ソースとなる DynamoDB テーブルでポイントインタイムリカバリ（PITR）が有効になっている必要があります。
3. リソースベースのポリシーの場合、ゼロ ETL 統合には、DynamoDB テーブルに直接アタッチされたリソースベースのポリシーが必要です。このインラインポリシーは、レプリケーションのためにテーブルデータにアクセスするアクセス許可を Amazon Redshift サービスに付与しま

す。DynamoDB でのリソースベースのポリシーの使用に関する詳細は、「DynamoDB でのリソースベースのポリシーの使用」を参照してください。

DynamoDB テーブルと Amazon Redshift データウェアハウスが同じアカウント内にある場合、統合の作成ステップで Fix it for me オプションを使用すると、DynamoDB と Amazon Redshift の両方に必要なリソースポリシーを自動的に適用できます。

異なる AWS アカウントにある DynamoDB テーブルと Amazon Redshift データウェアハウスの統合を作成する場合、DynamoDB デーブルに以下のリソースポリシーを手動で適用する必要があります。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid":
 "StatementthatallowsAmazonRedshiftservicetoDescribeTableandExportTable",
            "Effect": "Allow",
            "Principal": {
                "Service": "redshift.amazonaws.com"
            },
            "Action": [
                "dynamodb:ExportTableToPointInTime",
                "dynamodb:DescribeTable"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "111122223333"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
                }
            }
        },
        {
            "Sid":
 "StatementthatallowsAmazonRedshiftservicetoDescribeTableandExportTable",
            "Effect": "Allow",
            "Principal": {
```

```
                "Service": "redshift.amazonaws.com"
            },
            "Action": "dynamodb:DescribeExport",
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/table-
name/export/*",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "111122223333"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
                }
            }
        }
    ]
}
```

また、Amazon Redshift データウェアハウスでリソースポリシーを設定する必要がある場合もあります。詳細については、「Amazon Redshift API を使用して承認を設定する」を参照してください。

4. アイデンティティベースのポリシーについて:

   a. 統合を作成するユーザーには、GetResourcePolicy、PutResourcePolicy および UpdateContinuousBackups のアクションを許可するアイデンティティベースのポリシーが必要です。

> **Note**
>
> 以下の例では、リソースベースを arn:aws:redshift{-serverless} として示します。これは、名前空間が Amazon Redshift クラスターであるか、Amazon Redshift Serverless の名前空間であるかに応じて、arn が arn:aws:redshift または arn:aws:redshift-serverless になることを示す例です。

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:ListTables"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetResourcePolicy",
                "dynamodb:PutResourcePolicy",
                "dynamodb:UpdateContinuousBackups"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:111122223333:table/table-name"
            ]
        },
        {
            "Sid": "AllowRedshiftDescribeIntegration",
            "Effect": "Allow",
            "Action": [
                "redshift:DescribeIntegrations"
            ],
            "Resource": "*"
        },
        {
            "Sid": "AllowRedshiftCreateIntegration",
            "Effect": "Allow",
            "Action": "redshift:CreateIntegration",
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:integration:*"
        },
        {
            "Sid": "AllowRedshiftModifyDeleteIntegration",
            "Effect": "Allow",
            "Action": [
                "redshift:ModifyIntegration",
                "redshift:DeleteIntegration"
            ],
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:integration:uuid"
        },
```

```
        {
            "Sid": "AllowRedshiftCreateInboundIntegration",
            "Effect": "Allow",
            "Action": "redshift:CreateInboundIntegration",
            "Resource": "arn:aws:redshift:us-
east-1:111122223333:namespace:uuid"
        }
    ]
}
```

b. 送信先 Amazon Redshift 名前空間の設定を担当するユーザーには、PutResourcePolicy、DeleteResourcePolicy および GetResourcePolicy のアクションを許可するアイデンティティベースのポリシーが必要です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "redshift:PutResourcePolicy",
                "redshift:DeleteResourcePolicy",
                "redshift:GetResourcePolicy"
            ],
            "Resource": [
                "arn:aws:redshift:us-east-1:111122223333:cluster:cluster-
name"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "redshift:DescribeInboundIntegrations"
            ],
            "Resource": [
                "arn:aws:redshift:us-east-1:111122223333:cluster:cluster-
name"
            ]
        }
    ]
}
```

5. 暗号化キーの許可

ソース DynamoDB テーブルがカスタマー管理 AWS KMS キーを使用して暗号化されている場合は、KMS キーに次のポリシーを追加する必要があります。このポリシーにより、Amazon Redshift は KMS キーを使用して暗号化されたテーブルからデータをエクスポートできます。

```
{
    "Sid": "AllowRedshiftDecryptSourceTable",
    "Effect": "Allow",
    "Principal": {
        "Service": [
            "redshift.amazonaws.com"
        ]
    },
    "Action": "kms:Decrypt",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "aws:SourceAccount": "<account>"
        },
        "ArnEquals": {
            "aws:SourceArn": "arn:aws:redshift:<region>:<account>:integration:*"
        }
    }
}
```

Amazon Redshift 管理ガイドの「ゼロ ETL 統合の開始方法」の手順に従って、Amazon Redshift 名前空間のアクセス許可を設定することもできます。

DynamoDB の Amazon Redshift とのゼロ ETL 統合を使用する場合の制約事項

この統合の現在のリリースには次の制限が適用されます。これらの制限は、後続のリリースで変更される可能性があります。

> **Note**
>
> 以下の制限に加えて、ゼロ ETL 統合を使用する場合の一般的な考慮事項も確認してください。「Amazon Redshift 管理ガイド」の「Amazon Redshift との ゼロ ETL 統合を使用する場合の考慮事項」を参照してください。

- DynamoDB テーブルと Amazon Redshift クラスターは同じリージョンにある必要があります。
- ソース DynamoDB テーブルは、Amazon 所有またはカスタマー管理の AWS KMS キーで暗号化する必要があります。Amazon が管理する暗号化は、ソース DynamoDB テーブルではサポートされていません。

DynamoDB の Amazon Redshift とのゼロ ETL 統合の作成

ゼロ ETL 統合を作成する前に、ソース DynamoDB テーブルを設定し、次にターゲットの Amazon Redshift データウェアハウスを設定する必要があります。

ステップ 1: ソース DynamoDB テーブルを設定する

Amazon Redshift とゼロ ETL 統合を作成するには、テーブルでポイントインタイムリカバリ (PITR) を有効にする必要があります。PITR を有効にしていない場合は、統合セットアッププロセス中にコンソールで修正できます。PITR を有効にする方法の詳細については、「ポイントインタイムリカバリ」を参照してください。

ステップ 2: Amazon Redshift データウェアハウスを作成する

Amazon Redshift データウェアハウスをまだお持ちでない場合は、作成できます。Amazon Redshift Serverless ワークグループを作成するには、「名前空間を伴うワークグループの作成」を参照してください。Amazon Redshift クラスターを作成するには、「クラスターの作成」を参照してください。

正常に統合するには、ターゲット Amazon Redshift ワークグループまたはクラスターで enable_case_sensitive_identifier パラメータが有効になっている必要があります。大文字と小文字の区別を有効にする方法の詳細については、Amazon Redshift 管理ガイドの「データウェアハウスの大文字と小文字の区別を有効にする」を参照してください。

Amazon Redshift ワークグループまたはクラスターのセットアップが完了したら、データウェアハウスを設定する必要があります。詳細については、Amazon Redshift 管理ガイドの「ゼロ ETL 統合」を参照してください。

ステップ 3: DynamoDB ゼロ ETL 統合を作成する

ゼロ ETL 統合を作成する前に、 DynamoDB の Amazon Redshift とのゼロ ETL 統合を作成する前の前提条件 というタイトルのセクションのタスクを完了してください。DynamoDB と Amazon Redshift の統合の作成には、2 つのステップが必要です。まず DynamoDB から統合を作成し、次に Amazon Redshift データベースをこの新しく作成された統合にアタッチします。

ゼロ ETL 統合の作成

1. AWS マネジメントコンソールにサインインし、https://console.aws.amazon.com/dynamodbv2 で Amazon DynamoDB コンソールを開きます。
2. ナビゲーションペインで、[統合] を選択します。
3. [ゼロ ETL 統合の作成] を選択し、[Amazon Redshift]を選択します。
4. この手順により、Amazon Redshift コンソール に移動します。手順を続けるには、「DynamoDB のゼロ ETL 統合を作成する」の DynamoDB セクションを参照してください。

DynamoDB の Amazon Redshift とのゼロ ETL 統合の表示

ゼロ ETL 統合の詳細を表示して、設定情報と現在のステータスを確認できます。

Amazon DynamoDB コンソールでゼロ ETL 統合の詳細を表示するには以下の手順を実行します。

1. AWS マネジメントコンソールにサインインし、https://console.aws.amazon.com/dynamodbv2 で Amazon DynamoDB コンソールを開きます。
2. DynamoDB コンソールで、[統合] を選択します。
3. [ゼロ ETL 統合] ペインで、表示するゼロ ETL 統合を選択します。

Amazon Redshift コンソールでゼロ ETL 統合の詳細を表示するには以下の手順を実行します。

1. AWS マネジメントコンソールにサインインして、https://console.aws.amazon.com/redshiftv2 で Amazon Redshift コンソールを開きます。
2. [ゼロ ETL 統合の表示] のステップに従います。

> **Note**
>
> Amazon Redshift とのゼロ ETL 統合のステータス一覧は、「Amazon Redshift 管理ガイド」の「ゼロ ETL 統合の表示」に記載されています。

DynamoDB の Amazon Redshift とのゼロ ETL 統合の削除

ゼロ ETL 統合を削除すると、データは DynamoDB または Amazon Redshift から削除されないものの、DynamoDB はソーステーブルから Amazon Redshift ターゲットへのデータの送信を停止します。

ゼロ ETL 統合を削除するには

1. AWS マネジメントコンソールにサインインし、https://console.aws.amazon.com/dynamodbv2 で Amazon DynamoDB コンソールを開きます。
2. DynamoDB コンソールで、[統合]を選択します。
3. [ゼロ ETL 統合] ペインで、削除するゼロ ETL 統合を選択します。
4. [Manage] (管理) を選択します。この手順により、統合の詳細ページに移動します。
5. 削除を確認するには、[削除] を選択します。

COPY コマンドを使用した、DynamoDB から Amazon Redshift へのデータのロード

Amazon Redshift は、高度なビジネスインテリジェンス機能と強力な SQL ベースのインターフェイスによって Amazon DynamoDB と連携します。DynamoDB テーブルのデータを Amazon Redshift にコピーすると、Amazon Redshift クラスター内の他のテーブルとの結合など、複雑なデータ分析のクエリを実行できます。

プロビジョニングされたスループットの面から見ると、DynamoDB テーブルからのコピーオペレーションによって、テーブルの読み込みキャパシティーが低下します。データをコピーした後は、Amazon Redshift 内で SQL クエリを実行しても DynamoDB に全く影響はありません。これは、DynamoDB 自体に対してではなく、DynamoDB からのデータのコピーに対してクエリが実行されるためです。

DynamoDB テーブルからデータをロードする前に、まずデータのロード先として Amazon Redshift テーブルを作成する必要があります。注意していただきたいのは、NoSQL 環境から SQL 環境にデータをコピーしていることと、1 つの環境での特定のルールが他の環境には適用されないことです。考慮すべき相違点の例を以下に示します。

- DynamoDB テーブル名には「.」(ドット) や「-」(ダッシュ) を含む最大 255 文字を使用することができ、大文字と小文字が区別されます。Amazon Redshift テーブル名は 127 文字までに制限さ

れます。ドットやダッシュは使用できず、大文字と小文字は区別されません。さらに、テーブル名は Amazon Redshift の予約語と重複してはいけません。

- DynamoDB では、SQL の NULL の概念がサポートされていません。Amazon Redshift において、DynamoDB 内の空白またはブランクの属性値をどのように解釈するか、つまり、NULL と空白のフィールドのどちらで処理するかを指定する必要があります。
- DynamoDB のデータ型は、Amazon Redshift のデータ型と直接対応していません。Amazon Redshift テーブルの各列において、データ型が正しく、DynamoDB からのデータに対応したサイズであることを確認する必要があります。

Amazon Redshift SQL からの COPY コマンドの例を以下に示します。

```
copy favoritemovies from 'dynamodb://my-favorite-movies-table'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-Access-Key>'
readratio 50;
```

この例では、DynamoDB のソーステーブルは、`my-favorite-movies-table` です。Amazon Redshift のターゲットテーブルは、`favoritemovies` です。`readratio 50` 節により、プロビジョニングされたスループットの消費される割合が制限されます。この場合、COPY コマンドでは、`my-favorite-movies-table`用にプロビジョニングされた読み込みキャパシティーの 50% 以下しか使用されません。この割合については、未使用のプロビジョニングされたスループットの平均よりも小さい値に設定することを強くお勧めします。

DynamoDB から Amazon Redshift にデータをロードする詳細な手順については、Amazon Redshift データベースデベロッパーガイドの次のセクションを参照してください。

- DynamoDB テーブルからのデータのロード
- The COPY command
- COPY の例

Amazon EMR での Apache Hive を使用した DynamoDB データの処理

Amazon DynamoDB は、Amazon EMR で実行されるデータウェアハウスアプリケーションである Apache Hive と統合されています。Hive では、DynamoDB テーブルにあるデータの読み込み/書き込みが行えます。これにより、以下のことが可能になります。

- SQL に似た言語 (HiveQL) を使用して、その時点の DynamoDB データがクエリできます。
- DynamoDB テーブルから Amazon S3 バケットに (あるいはその逆の方向で) データをコピーできます。
- DynamoDB テーブルから Hadoop Distributed File System (HDFS) に (あるいはその逆の方向で) データをコピーできます。
- DynamoDB テーブルで JOIN オペレーションを実行できます。

トピック

- 概要
- チュートリアル:Amazon DynamoDB と Apache Hive の使用
- Hive に外部テーブルを作成します
- HiveQL ステートメントの処理
- DynamoDB 内データのクエリ
- Amazon DynamoDB との間でデータをコピーします
- パフォーマンスチューニング

概要

Amazon EMR は、膨大な量のデータの迅速かつコスト効率の良い処理を容易に行えるサービスです。Amazon EMR を使用するには、Hadoop のオープンソースフレームワークを実行する Amazon EC2 インスタンスの、マネージド型クラスターを起動します。Hadoopは、タスクがクラスター内の複数のノードにマッピングされている場合に、MapReduce アルゴリズムを実装するための分散型アプリケーションです。各ノードは、他のノードと並列的に指定された作業を処理します。それらの出力は単一のノードに集約され、最終的な結果が得られます。

Amazon EMR では、永続的または一時的のどちらかを選択して、クラスターを起動することができます。

- 永続的なクラスターは、シャットダウンされるまでその実行が継続します。永続的なクラスターは、データ分析やデータウェアハウスなどを含む、インタラクティブな使用に最適です。
- 一時的なクラスターは、ジョブフローの処理に必要な長さだけ実行された後、自動的にシャットダウンされます。一時的なクラスターは、スクリプトの実行など、定期的な処理タスクに最適です。

Amazon EMR のアーキテクチャと管理の詳細については、Amazon EMR 管理ガイドを参照してください。

ユーザーは、Amazon EMR クラスターを起動する際に、初期段階での Amazon EC2 インスタンスの数とタイプを指定します。また、(Hadoop 自体に加えて) クラスターで実行する他の分散型アプリケーションも指定する必要があります。これらのアプリケーションには Hue、Mahout、Pig、Spark などが含まれます。

Amazon EMR のアプリケーションについては、Amazon EMR リリース ガイドを参照してください。

クラスター構成に応じて、次の中の 1 つ以上のノードタイプが利用可能です。

- リーダーノード – クラスターを管理します。MapReduce の実行可能ファイル、および生データのサブセットを、コアおよびタスクインスタンスグループに適切に分散させます。また、実行された各タスクのステータスを追跡し、インスタンスグループの正常性をモニタリングします。クラスターにはリーダーノードが 1 つのみ存在します。
- コアノード – MapReduce タスクを実行しデータを保存します。その際、Hadoop Distributed File System (HDFS) を使用します。
- タスクノード – (オプション) MapReduce タスクを実行します。

チュートリアル:Amazon DynamoDB と Apache Hive の使用

このチュートリアルでは、まず、Amazon EMR クラスターを起動した上で、DynamoDB テーブルに格納されているデータの処理に Apache Hive を使用していきます。

Hiveは、Hadoop 用のデータウェアハウスアプリケーションで、複数のソースからのデータを処理および分析することを可能にします。Hive では、SQL に似た言語である HiveQL を使用しながら、ローカルの Amazon EMR クラスター、または外部のデータソース (Amazon DynamoDB など) に保存されたデータを操作できます。

詳細については、「Hive Tutorial」を参照してください。

トピック

- 開始する前に
- ステップ 1: Amazon EC2 キーペアを作成する
- ステップ 2: Amazon EMR クラスターを起動します

- ステップ 3: リーダーノードに接続します
- ステップ 4: HDFS にデータをロードします
- ステップ 5: データを DynamoDB にコピーします
- ステップ 6: DynamoDB テーブル内のデータをクエリを行います
- ステップ 7: (オプション) クリーンアップする

開始する前に

このチュートリアルでは、以下が必要になります。

- AWS アカウント。アカウントをお持ちでない場合は、「AWS アカウント へのサインアップ」を参照してください。
- SSH クライアント (セキュアシェル)。SSH クライアントを使用して、Amazon EMR クラスターのリーダーノードに接続し、対話型コマンドを実行します。ほとんどの Linux、Unix、および Mac OS X の実装では、SSH クライアントをデフォルトで利用できます。Windows ユーザーの場合は、SSH がサポートする PuTTY クライアントをダウンロードしてインストールできます。

次のステップ

ステップ 1: Amazon EC2 キーペアを作成する

ステップ 1: Amazon EC2 キーペアを作成する

このステップでは、Amazon EMR リーダーノードに接続し、Hive コマンドを実行するために必要な Amazon EC2 のキーペアを作成します。

1. AWS マネジメントコンソール にサインインし、Amazon EC2 コンソール (https://console.aws.amazon.com/ec2/) を開きます。
2. リージョン (例: `US West (Oregon)`) を選択します。これは、DynamoDB テーブルが配置されているリージョンと同じリージョンにする必要があります。
3. ナビゲーションペインで、[Key Pairs (キーペア)] を選択します。
4. [Create Key Pair] を選択します。
5. [Key pair name] (キーペア名) にキーペアの名前 (例: `mykeypair`) をタイプし、[Create] (作成) を選択します。
6. プライベートキーファイルをダウンロードします。このファイル名は、`.pem` (`mykeypair.pem` など) で終わります。ダウンロードしたプライベートキーファイルは安全な場所に保存します。

ここで作成するキーペアにより起動する Amazon EMR クラスターへのアクセスには、このファイルが必要となります。

> **Important**
>
> キーペアが失われた場合、Amazon EMR クラスターのリーダーノードに接続することはできなくなります。

キーペアの詳細については、「Amazon EC2 ユーザーガイド」の「Amazon EC2 のキーペアと Amazon EC2 インスタンス」を参照してください。

次のステップ

ステップ 2: Amazon EMR クラスターを起動します

ステップ 2: Amazon EMR クラスターを起動します

このステップでは、Amazon EMR クラスターを設定して起動します。このクラスターには、Hive および DynamoDB 用のストレージハンドラーが、既にインストールされています。

1. Amazon EMR コンソール (https://console.aws.amazon.com/emr) を開きます。
2. [クラスターの作成] を選択します。
3. [Create Cluster - Quick Options] (クラスターの作成 – クイックオプション) ページで、以下を実行します。

   a. [Cluster name] (クラスター名) にクラスターの名前を入力します (例: `My EMR cluster` など)。

   b. [EC2 key pair] (EC2 キーペア) で、先に作成してあるキーペアを選択します。

   その他の設定はデフォルト値のままにしておきます。
4. [Create cluster (クラスターの作成)] を選択します。

クラスターを起動するには数分間かかります。このプロセスの進捗状況は、Amazon EMR コンソールの [Cluster Details] (クラスターの詳細) ページで確認できます。

ステータスが `Waiting` に変わると、クラスターは使用可能状態になっています。

クラスターのログファイルと Amazon S3

Amazon EMR クラスターは、クラスターのステータスやデバッグに関する情報が記載されたログファイルを生成します。[Create Cluster - Quick Options (クラスターの作成-クイックオプション)] のデフォルト設定には、Amazon EMR でのログ記録に関する設定が含まれています。

まだ Amazon S3 バケットが存在しない場合は、AWS マネジメントコンソール マネジメントコンソールにより作成されます。このバケットの名前は `aws-logs-`*`account-id`*`-`*`region`* となります。ここで、 *`account-id`* は AWS アカウント番号で、*`region`*は、クラスターを起動したリージョン (例えば `aws-logs-123456789012-us-west-2`) です。

> **Note**
>
> Amazon S3 コンソールを使用して、このログファイルを表示できます。詳細については、「Amazon EMR 管理ガイド」の「ログファイルを表示する」を参照してください。

このバケットは、ログ記録以外の目的でも使用できます。例えば、バケットを Hive スクリプトを保存する場所として使用することや、Amazon DynamoDB から Amazon S3 にデータをエクスポートする際の送信先として使用することが可能です。

次のステップ

ステップ 3: リーダーノードに接続します

ステップ 3: リーダーノードに接続します

Amazon EMR クラスターのステータスが `Waiting` の状態であれば、SSH を使用してリーダーノードに接続し、コマンドラインによるオペレーションを実行できます。

1. Amazon EMR コンソールで、クラスターの名前を選択して、そのステータスを表示します。
2. [Cluster Details] (クラスターの詳細) ページで、[Leader public DNS] (リーダーのパブリック DNS) フィールドを見つけます。これは、Amazon EMR クラスターのリーダーノードのためのパブリック DNS 名です。
3. DNS 名の右側で、SSH リンクを選択します。
4. [Connect to the Leader Node Using SSH] (SSH を使用してマスターノードに接続する) にある手順を実行します。

   使用しているオペレーティングシステムに応じて、Windows タブまたは Mac/Linux タブを選択し、指示に従いリーダーノードに接続します。

SSH または PuTTY を使用してリーダーノードへの接続が完了すると、次のようなコマンドプロンプトが表示されます。

```
[hadoop@ip-192-0-2-0 ~]$
```

次のステップ

ステップ 4: HDFS にデータをロードします

ステップ 4: HDFS にデータをロードします

このステップでは、データファイルを Hadoop Distributed File System (HDFS) にコピーし、データファイルにマップする外部 Hive テーブルを作成します。

サンプルデータをダウンロードします

1. サンプルデータアーカイブ (features.zip) のダウンロード:

   ```
   wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
   features.zip
   ```

2. アーカイブからの features.txt ファイルの抽出:

   ```
   unzip features.zip
   ```

3. features.txt ファイルの最初の数行を表示:

   ```
   head features.txt
   ```

   この結果は以下のようになります。

   ```
   1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794
   875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7
   1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10
   26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681
   1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605
   1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558
   1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024
   533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0
   829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671
   541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
   ```

`features.txt` ファイルには、米国地名委員会 (http://geonames.usgs.gov/domestic/download_data.htm) で提供されるデータのサブセットが含まれます。ここでの各行のフィールドは、次を表しています。

- フィーチャ ID (固有の識別子)
- 名前
- クラス (湖、森林、小川、その他)
- 状態
- 緯度 (度)
- 経度 (度)
- 高度 (フィート)

4. コマンドプロンプトに次のコマンドを入力します。

```
hive
```

コマンドプロンプトがこのように変わります: `hive>`

5. 次の HiveQL ステートメントを入力して、ネイティブ Hive テーブルを作成します。

```
CREATE TABLE hive_features
    (feature_id             BIGINT,
    feature_name            STRING ,
    feature_class           STRING ,
    state_alpha             STRING,
    prim_lat_dec            DOUBLE ,
    prim_long_dec           DOUBLE ,
    elev_in_ft              BIGINT)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '|'
    LINES TERMINATED BY '\n';
```

6. 次の HiveQL ステートメントを入力して、データとともにテーブルをロードします。

```
LOAD DATA
LOCAL
INPATH './features.txt'
OVERWRITE
INTO TABLE hive_features;
```

7. これで、features.txt ファイルからのデータが格納された、ネイティブ Hive テーブルが用意できます。これを確認するには、次の HiveQL ステートメントを入力します。

```
SELECT state_alpha, COUNT(*)
FROM hive_features
GROUP BY state_alpha;
```

出力には、州の一覧といくつかの地理的特徴が表示されます。

次のステップ

ステップ 5: データを DynamoDB にコピーします

ステップ 5: データを DynamoDB にコピーします

このステップでは、データを Hive テーブル (hive_features) から DynamoDB の新しいテーブルにコピーします。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. [テーブルの作成] を選択します。
3. [Create DynamoDB table] (DynamoDB テーブルの作成) ページで、次の操作を行います。

   a. [Table] (テーブル) に **Features** とタイプします。

   b. [Partition key] (パーティションキー) フィールドの [Primary key] (プライマリキー) に **Id** とタイプします。データ型を [数値] に設定します。

      [Use default settings] (デフォルト設定の使用) を消します。[Provisioned Capacity] (プロビジョニングキャパシティ) に以下をタイプします。

      - [Read Capacity Units] (読み込みキャパシティユニット) — 10
      - [Write Capacity Units] (書き込みキャパシティユニット) — 10

   [Create] を選択します。
4. Hive プロンプトが表示された状態で、次の HiveQL ステートメントを入力します。

```
CREATE EXTERNAL TABLE ddb_features
    (feature_id   BIGINT,
    feature_name  STRING,
```

```
    feature_class STRING,
    state_alpha   STRING,
    prim_lat_dec  DOUBLE,
    prim_long_dec DOUBLE,
    elev_in_ft    BIGINT)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES(
    "dynamodb.table.name" = "Features",

 "dynamodb.column.mapping"="feature_id:Id,feature_name:Name,feature_class:Class,state_alpha
);
```

これで、Hive と DynamoDB のフィーチャーテーブル間でのマッピングが確立されました。

5. 次の HiveQL ステートメントを入力して DynamoDB にデータをインポートします。

```
INSERT OVERWRITE TABLE ddb_features
SELECT
    feature_id,
    feature_name,
    feature_class,
    state_alpha,
    prim_lat_dec,
    prim_long_dec,
    elev_in_ft
FROM hive_features;
```

Hive が MapReduce ジョブを送信します。このジョブは Amazon EMR クラスターによって処理されます。このジョブが完了するまで数分間かかります。

6. データが DynamoDB にロードされていることを確認します。

   a. DynamoDB コンソールのナビゲーションペインで、[Tables] (テーブル) を選択します。

   b. フィーチャーテーブルを選択し、[Items] (アイテム) タブを選択して、データを表示します。

次のステップ

ステップ 6: DynamoDB テーブル内のデータをクエリを行います

ステップ 6: DynamoDB テーブル内のデータをクエリを行います

このステップでは、HiveQL を使用して DynamoDB の フィーチャーテーブルをクエリします。以下の Hive クエリを使用します。

1. すべてのフィーチャタイプ (`feature_class`) をアルファベット順に取得:

```
SELECT DISTINCT feature_class
FROM ddb_features
ORDER BY feature_class;
```

2. 名前が文字「M」で始まるすべての湖を取得 :

```
SELECT feature_name, state_alpha
FROM ddb_features
WHERE feature_class = 'Lake'
AND feature_name LIKE 'M%'
ORDER BY feature_name;
```

3. 1 マイル (5,280 フィート) より高度が高く、少なくとも 3 つのフィーチャを持つ州を取得:

```
SELECT state_alpha, feature_class, COUNT(*)
FROM ddb_features
WHERE elev_in_ft > 5280
GROUP by state_alpha, feature_class
HAVING COUNT(*) >= 3
ORDER BY state_alpha, feature_class;
```

次のステップ

ステップ 7: (オプション) クリーンアップする

ステップ 7: (オプション) クリーンアップする

この段階で、チュートリアルは完了しています。引き続きこのセクションを読むことで、Amazon EMR での DynamoDB データの操作について、さらに詳細に学習することができます。このステップを実行する場合は、Amazon EMR クラスターを稼働させ続けることができます。

クラスターが不要である場合は、それを終了し、関連するリソースの削除を行ってください。これにより、不要なリソースに対して課金されることを回避できます。

1. Amazon EMR クラスターを終了します。

   a. Amazon EMR コンソール (https://console.aws.amazon.com/emr) を開きます。

   b. Amazon EMR クラスターを選択した上で、[Terminate] (終了) を選択し確認します。

2. DynamoDB 内のフィーチャーテーブルを削除します。

   a. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。

   b. ナビゲーションペインで、[Tables (テーブル)] を選択します。

   c. フィーチャ-テーブルを選択します。[Actions] (アクション) メニューから、[Delete Table] (テーブルの削除) を選択します。

3. Amazon EMR ログファイルが保存されている Amazon S3 バケットを削除します。

   a. Amazon S3 コンソール (https://console.aws.amazon.com/s3/) を開きます。

   b. バケットのリストから [aws-logs- *accountID*-*region*] を選択します。ここで、*accountID* は使用している AWS アカウント番号であり、*region* はクラスターを起動したリージョンです。

   c. [Actions (アクション)] メニューから、[Delete (削除)] を選択します。

Hive に外部テーブルを作成します

チュートリアル:Amazon DynamoDB と Apache Hive の使用 において、DynamoDB テーブルにマッピングされた外部 Hive テーブルを作成しました。この状態で外部テーブルに対して HiveQL ステートメントを発行すると、読み込み/書き込みの操作が DynamoDB テーブルに渡されます。

外部テーブルは、他の場所で管理および格納されているデータソースへのポインタと捉えることができます。ここでは、基盤となるそのデータソースは、DynamoDB テーブルです。(このテーブルは外部に用意する必要があります。Hive 内から DynamoDB テーブルを作成、更新、または削除することはできません。) 外部テーブルの作成には、`CREATE EXTERNAL TABLE` ステートメントを使用します。その後は、DynamoDB 内のデータが Hive 内にローカルに格納されているかのように、HiveQL を使用してそのデータを操作できます。

> **Note**
>
> `INSERT` ステートメントを使用すると、外部テーブルにデータを挿入でき、また、`SELECT` ステートメントにより、そのテーブルのデータを選択できます。ただし、テーブル内のデー

タを操作するために、UPDATE または DELETE ステートメントを使用することはできません。

外部テーブルが必要でなくなった時に、DROP TABLE を使ってそのテーブルを削除します。この場合でも、DROP TABLE は Hive の外部テーブルだけを削除します。このオペレーションは、基盤となっている DynamoDB テーブルまたはそのデータには影響を与えません。

トピック

- CREATE EXTERNAL TABLE 構文
- データ型のマッピング

CREATE EXTERNAL TABLE 構文

DynamoDB テーブルにマッピングする外部 Hive テーブルを作成するための HiveQL 構文を次に示します。

```
CREATE EXTERNAL TABLE hive_table

 (hive_column1_name hive_column1_datatype, hive_column2_name hive_column2_datatype...)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES (
    "dynamodb.table.name" = "dynamodb_table",
    "dynamodb.column.mapping" =
 "hive_column1_name:dynamodb_attribute1_name,hive_column2_name:dynamodb_attribute2_name..."
);
```

1 行目から CREATE EXTERNAL TABLE ステートメントが始まりますが、作成する Hive テーブルの名前 (hive_table) を指定します。

2 行目では、hive_table の列とデータ型を指定します。DynamoDB テーブルの属性に対応する列とデータ型を定義する必要があります。

3 行目は STORED BY 句で、ここでは、Hive と DynamoDB テーブル間のデータ管理を処理するクラスを指定します。DynamoDB の場合、STORED BY は 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler' に設定する必要があります。

4 行目で TBLPROPERTIES 句が始まり、以下の DynamoDBStorageHandler のパラメータを定義しています。

- dynamodb.table.name – DynamoDB テーブルの名前。
- dynamodb.column.mapping – Hive テーブルの列名と DynamoDB テーブルの対応する属性のペア。各ペアの形式は、hive_column_name:dynamodb_attribute_name とし、それぞれをカンマで区切ります。

次の点に注意してください。

- Hive テーブル名の名前は、DynamoDB テーブル名と同じである必要はありません。
- Hive テーブルの列名は、DynamoDB テーブルの列名と同じである必要はありません。
- dynamodb.table.name で指定されたテーブルは、DynamoDB 内に置いてある必要があります。
- 複数 dynamodb.column.mapping:
  - DynamoDB テーブルには、キースキーマ属性をマッピングする必要があります。これには、パーティションキーと (それが存在する場合は) ソートキー が含まれます。
  - DynamoDB テーブルの非キー属性のマッピングは、必須ではありません。ただし、Hive テーブルに対しクエリを実行しても、マッピングしていないこれらの属性のデータは表示されません。
  - Hive テーブルの列のデータ型と DynamoDB 属性のデータ型に互換性がない場合、Hive テーブルをクエリすると、これらの列には NULL が表示されます。

> **Note**
>
> CREATE EXTERNAL TABLE ステートメントでは、TBLPROPERTIES 句に関する検証は行われません。dynamodb.table.name および dynamodb.column.mapping に指定した値は、テーブルへのアクセスを試みた時点で、 クラスによってのみ評価されます。

データ型のマッピング

次の表に、DynamoDB でのデータ型と、それに互換性のある Hive でのデータ型を示します。

DynamoDB データ型	Hive データ型
文字列	STRING
数値	BIGINT-または-DOUBLE
バイナリ	BINARY
文字列セット	ARRAY<STRING>
数値セット	ARRAY<BIGINT> または ARRAY<DOUBLE>
バイナリセット	ARRAY<BINARY>

> **Note**
>
> 以下の DynamoDB データ型は、DynamoDBStorageHandler クラスではサポートされないため、dynamodb.column.mapping とともに使用することはできません。
>
> - マップ
> - リスト
> - ブール値
> - Null
>
> ただし、これらのデータ型を処理する必要がある場合は、DynamoDB アイテム全体を、マップ内のキーと値の両方の文字列のマップとして表す item という名前の 1 つのエンティティを作成できます。詳細については、列マッピングを使用しないデータをコピーを参照してください。

数値型の DynamoDB 属性をマッピングする場合は、適切な Hive タイプを選択する必要があります。

- Hive BIGINT 型は 8 バイトの符号付き整数です。これは、Java での long データ型と等価です。
- Hive DOUBLE 型は 8 ビット倍精度浮動小数点数です。これは、Java での double 型と等価です。

選択した Hive データ型よりも精度が高い数値データが DynamoDB に格納されている場合、DynamoDB データにアクセスすると精度を損失する可能性があります。

DynamoDB から HDFS (または Amazon S3) にバイナリ型のデータを エクスポートすると、データは Base64 でエンコードされた文字列として格納されます。Amazon S3 または HDFS から DynamoDB に対し、バイナリ型でデータをインポートする場合には、そのデータが Base64 文字列としてエンコードされていることを確認します。

HiveQL ステートメントの処理

Hive は、MapReduceジョブを実行するためのバッチ指向フレームワークとして、Hadoop上で動作するアプリケーションです。HiveQL ステートメントを発行すると、Hive は結果をすぐに返すことができるかどうか、あるいは、MapReduce ジョブを送信する必要があるかどうかを判断します。

例えば、(チュートリアル:Amazon DynamoDB と Apache Hive の使用 より) ddb_features テーブルを考えてみます。次の Hive クエリは、州の省略した名前と、それぞれの州に存在する山頂の数を出力します。

```
SELECT state_alpha, count(*)
FROM ddb_features
WHERE feature_class = 'Summit'
GROUP BY state_alpha;
```

Hive は結果をすぐに返しません。代わりに、Hive からは MapReduceジョブが送信され、Hadoop フレームワークによって処理されます。Hive は、このジョブが完了するまで待機してから、クエリの結果を表示します。

```
AK  2
AL  2
AR  2
AZ  3
CA  7
CO  2
CT  2
ID  1
KS  1
ME  2
MI  1
MT  3
NC  1
```

```
NE  1
NM  1
NY  2
OR  5
PA  1
TN  1
TX  1
UT  4
VA  1
VT  2
WA  2
WY  3
Time taken: 8.753 seconds, Fetched: 25 row(s)
```

ジョブのモニタリングとキャンセル

Hive が Hadoop ジョブを起動すると、そのジョブからの出力が表示されます。表示されるジョブの完了ステータスは、ジョブの進捗に応じて更新されます。このステータスは、長時間更新されないこともあります。(この現象は、読み込みキャパシティーのプロビジョニングが低く設定されている、大型の DynamoDB テーブルをクエリした場合に発生する可能性があります。)

Ctrl+C とタイプすることで、完了する前に任意のタイミングで、このジョブをキャンセルすることが可能です。

DynamoDB 内データのクエリ

次の例で、DynamoDB に保存されたデータへのクエリに、HiveQL を使用する方法をいくつか示します。

これらの例は、チュートリアル (ステップ 5: データを DynamoDB にコピーします) にある ddb_features テーブルを参照しています。

トピック

- 集計関数の使用
- GROUP BY 句および HAVING 句の使用
- 2 つの DynamoDB テーブルの結合
- 異なるソースからのテーブルの結合

集計関数の使用

HiveQL では、データ値を集計するための組み込み関数が利用できます。例えば、選択した列で最大値を検索するためには MAX 関数を使用できます。次の例では、コロラド州で標高が最も高いという特徴を返します。

```
SELECT MAX(elev_in_ft)
FROM ddb_features
WHERE state_alpha = 'CO';
```

GROUP BY 句および HAVING 句の使用

GROUP BY 句を使用して、複数のレコードのデータを収集できます。これは、多くの場合、SUM、COUNT、MIN、または MAX のような集計関数で使用されます。また、一定の基準を満たさない結果を破棄する場合には、HAVING 句を使用することもできます。

次の例では、ddb_features テーブルの中に 5 つ以上の特徴を持つ州での、最も高い標高のリストを返します。

```
SELECT state_alpha, max(elev_in_ft)
FROM ddb_features
GROUP BY state_alpha
HAVING count(*) >= 5;
```

2 つの DynamoDB テーブルの結合

次の例では、DynamoDB 内のテーブルに対し、別の Hive テーブル (east_coast_states) をマッピングしています。SELECT ステートメントにより、これらの 2 つのテーブル間での結合が行われます。結合はクラスターで計算され、以下を返します。結合は DynamoDB 内では発生しません。

以下のデータが保存されている、EastCoastStates という名前の DynamoDB テーブルがあるとします。

```
StateName       StateAbbrev

Maine           ME
New Hampshire   NH
Massachusetts   MA
Rhode Island    RI
Connecticut     CT
```

```
New York        NY
New Jersey      NJ
Delaware        DE
Maryland        MD
Virginia        VA
North Carolina  NC
South Carolina  SC
Georgia         GA
Florida         FL
```

このテーブルは、east_coast_states という名前の Hive 外部テーブルとして利用可能であると仮定します。

```
CREATE EXTERNAL TABLE ddb_east_coast_states (state_name STRING, state_alpha STRING)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "EastCoastStates",
"dynamodb.column.mapping" = "state_name:StateName,state_alpha:StateAbbrev");
```

次の結合により、少なくとも 3 つの特徴を持つ米国東海岸の州が返されます。

```
SELECT ecs.state_name, f.feature_class, COUNT(*)
FROM ddb_east_coast_states ecs
JOIN ddb_features f on ecs.state_alpha = f.state_alpha
GROUP BY ecs.state_name, f.feature_class
HAVING COUNT(*) >= 3;
```

異なるソースからのテーブルの結合

次の例では、s3_east_coast_states は、Amazon S3 に格納された CSV ファイルに関連付けられた Hive テーブルです。ddb_features テーブルは DynamoDB 内のデータに関連付けられています。この例では、これら 2 つのテーブルを結合し、名前が「New」で始まる州における地理的な特徴を返します。

```
create external table s3_east_coast_states (state_name STRING, state_alpha STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://bucketname/path/subpath/';
```

```
SELECT ecs.state_name, f.feature_name, f.feature_class
FROM s3_east_coast_states ecs
JOIN ddb_features f
```

```
ON ecs.state_alpha = f.state_alpha
WHERE ecs.state_name LIKE 'New%';
```

Amazon DynamoDB との間でデータをコピーします

チュートリアル:Amazon DynamoDB と Apache Hive の使用 では、ネイティブの Hive テーブルから外部にある DynamoDB テーブルにデータをコピーし、その外部 DynamoDB テーブルにクエリを実行しました。外部テーブルは Hive の外側に存在するテーブルです。仮に、DynamoDB テーブルにマッピングされた Hive テーブルを削除したとしても、DynamoDB テーブルは影響を受けません。

DynamoDB テーブル、Amazon S3 バケット、ネイティブ Hive テーブル、および Hadoop Distributed File System (HDFS) の間でデータをコピーする場合は、Hive が優れたソリューションとなります。このセクションでは、これらのオペレーションの例を示します。

トピック

- DynamoDB とネイティブな Hive テーブル間でのデータのコピー
- DynamoDB と Amazon S3 間でのデータのコピー
- DynamoDB と HDFS 間のデータをコピー
- データ圧縮を使用する
- 表示できない UTF-8 文字データの読み込み

DynamoDB とネイティブな Hive テーブル間でのデータのコピー

DynamoDB テーブル内に存在するデータは、ネイティブの Hive テーブルにコピーできます。これにより、コピーした時点のデータのスナップショットを作成できます。

多くの HiveQL クエリを実行する必要があるものの、DynamoDB からプロビジョニングされたスループットキャパシティーを消費したくない場合などに、この操作を行います。ネイティブ Hive テーブルのデータは DynamoDB からのデータのコピーであり、その時点でのデータではないため、クエリでは、データが最新であると想定することはできません。

> **Note**
>
> このセクションの例は、チュートリアル:Amazon DynamoDB と Apache Hive の使用 の手順に従い、および、ddb_features という名前の外部テーブルが DynamoDB に存在することを前提にしています。

Example DynamoDB からネイティブ Hive テーブルへ

次のように、ネイティブの Hive テーブルを作成し、ddb_features からのデータを格納することができます。

```
CREATE TABLE features_snapshot AS
SELECT * FROM ddb_features;
```

このデータはいつでも更新できます。

```
INSERT OVERWRITE TABLE features_snapshot
SELECT * FROM ddb_features;
```

ここでの例では、サブクエリ SELECT * FROM ddb_features により、ddb_features からのすべてのデータを取得できます。データのサブセットのみをコピーしたい場合は、このサブクエリで WHERE 句を使用できます。

次の例では、湖と山頂の一部の属性のみを含むネイティブの Hive テーブルを作成します。

```
CREATE TABLE lakes_and_summits AS
SELECT feature_name, feature_class, state_alpha
FROM ddb_features
WHERE feature_class IN ('Lake','Summit');
```

Exampleネイティブの Hive テーブルから DynamoDB へ

次の HiveQL ステートメントを使用して、ネイティブな Hive テーブルから ddb_features に対してデータをコピーします。

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM features_snapshot;
```

DynamoDB と Amazon S3 間でのデータのコピー

DynamoDB テーブルにあるデータは、Hive を使用して Amazon S3 バケットにコピーすることができます。

この操作は、DynamoDB テーブルにデータのアーカイブを作成したい場合などに行います。例えば、DynamoDB にテストデータのベースラインセットが置かれており、テスト環境でそれを操作す

る必要があるとします。このベースラインデータは、Amazon S3 バケットにコピーした上で、必要なテストを実行できます。その後、Amazon S3 バケットから DynamoDB にベースラインデータを復元することで、テスト環境をリセットすることができます。

チュートリアル:Amazon DynamoDB と Apache Hive の使用 の作業を行った場合、Amazon EMR ログを保存する Amazon S3 バケットが、既に作成されています。バケットのルートパスがわかっている場合は、このセクションの例のために、このバケットを利用できます。

1. Amazon EMR コンソール (https://console.aws.amazon.com/emr) を開きます。
2. [Name] (名前) で、クラスターを選択します。
3. URI が、[Configuration Details] (設定の詳細) の [Log URI] (ログの URI) に一覧表示されています。
4. バケットのルートパスを書き留めておきます。この命名規則は以下のとおりです。

   ```
   s3://aws-logs-accountID-region
   ```

   accountID は、使用している AWS アカウント ID であり、region は、バケットが置かれた AWS リージョンです。

> **Note**
>
> これらの例では、ここに示すようなバケット内のサブパスを使用します。
> `s3://aws-logs-123456789012-us-west-2/hive-test`

以下の手順は、チュートリアルの手順が正しく行われたこと、および、ddb_features という名前の外部テーブルが DynamoDB 内に存在することを前提として書かれています。

トピック

- Hive のデフォルト形式を使用したデータのコピー
- ユーザー指定の形式でデータをコピー
- 列マッピングを使用しないデータをコピー
- Amazon S3 内のデータを表示

Hive のデフォルト形式を使用したデータのコピー

Example DynamoDB から Amazon S3 へのコピー

INSERT OVERWRITE ステートメントを使用して、Amazon S3 に直接書き込みます。

```
INSERT OVERWRITE DIRECTORY 's3://aws-logs-123456789012-us-west-2/hive-test'
SELECT * FROM ddb_features;
```

Amazon S3 でのデータファイルは次のようになります。

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

各フィールドは SOH 文字 (0x01、ヘッダーの開始) で区切られます。ファイルでは、SOH は **^A** と表示されています。

Example Amazon S3 から DynamoDB へのコピー

1. Amazon S3 内のフォーマットされていないデータを指す外部テーブルを作成します。

   ```
   CREATE EXTERNAL TABLE s3_features_unformatted
       (feature_id       BIGINT,
       feature_name      STRING ,
       feature_class     STRING ,
       state_alpha       STRING,
       prim_lat_dec      DOUBLE ,
       prim_long_dec     DOUBLE ,
       elev_in_ft        BIGINT)
   LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
   ```

2. データを DynamoDB にコピーします。

   ```
   INSERT OVERWRITE TABLE ddb_features
   SELECT * FROM s3_features_unformatted;
   ```

ユーザー指定の形式でデータをコピー

フィールド区切りに独自の文字を指定する場合は、Amazon S3 バケットにマッピングされる外部テーブルを作成します。この手法は、カンマ区切り値 (CSV) のデータファイルを作成する場合などに使用できます。

Example DynamoDB から Amazon S3 へのコピー

1. Amazon S3 にマッピングされる Hive 外部テーブルを作成します。これを行う際は、データ型が DynamoDB 外部テーブルのデータ型と一致していることを確認します。

```
CREATE EXTERNAL TABLE s3_features_csv
    (feature_id       BIGINT,
    feature_name      STRING,
    feature_class     STRING,
    state_alpha       STRING,
    prim_lat_dec      DOUBLE,
    prim_long_dec     DOUBLE,
    elev_in_ft        BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. DynamoDB からデータをコピーします。

```
INSERT OVERWRITE TABLE s3_features_csv
SELECT * FROM ddb_features;
```

Amazon S3 でのデータファイルは次のようになります。

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example Amazon S3 から DynamoDB へのコピー

単一の HiveQL ステートメントにより、DynamoDB テーブルに Amazon S3 からのデータを書き込むことができます。

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM s3_features_csv;
```

列マッピングを使用しないデータをコピー

DynamoDB からのデータは、データ型や列マッピングを指定せずにそのままの形式でコピーし、Amazon S3 に書き込むことができます。この手法は、DynamoDB データのアーカイブを作成して Amazon S3 に保存する場合などに使用します。

Example DynamoDB から Amazon S3 へのコピー

1. DynamoDB テーブルに関連付けられた外部テーブルを作成します。(この HiveQL ステートメントには `dynamodb.column.mapping` はありません)。

   ```
   CREATE EXTERNAL TABLE ddb_features_no_mapping
       (item MAP<STRING, STRING>)
   STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
   TBLPROPERTIES ("dynamodb.table.name" = "Features");
   ```

2. Amazon S3 バケットに関連付けられた別の外部テーブルを作成します。

   ```
   CREATE EXTERNAL TABLE s3_features_no_mapping
       (item MAP<STRING, STRING>)
   ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '\t'
   LINES TERMINATED BY '\n'
   LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
   ```

3. DynamoDB から Amazon S3 にデータをコピーします。

   ```
   INSERT OVERWRITE TABLE s3_features_no_mapping
   SELECT * FROM ddb_features_no_mapping;
   ```

Amazon S3 でのデータファイルは次のようになります。

```
Name^C{"s":"Soldiers Farewell
 Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"6135"}^BLatitude^C{"n":"32.
Name^C{"s":"Jones
 Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":"1260"}^BLatitude^C{"n":"41.2
```

```
Name^C{"s":"Sentinel
 Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"8133"}^BLatitude^C{"n":"37.
Name^C{"s":"Neversweet
 Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":"2900"}^BLatitude^C{"n":"41
Name^C{"s":"Chacaloochee
 Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":"0"}^BLatitude^C{"n":"30.6979676
```

各フィールドの先頭には、STX文字 (0x02、テキストの開始) が置かれ、末尾には ETX文字 (0x03、テキストの終わり) が置かれます。ファイルでは、STX は **^B** として表示され、ETX は **^C** として表示されます。

Example Amazon S3 から DynamoDB へのコピー

単一の HiveQL ステートメントにより、DynamoDB テーブルに Amazon S3 からのデータを書き込むことができます。

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM s3_features_no_mapping;
```

Amazon S3 内のデータを表示

SSH を使用してリーダーノードに接続している場合には、AWS Command Line Interface (AWS CLI) を使用して、Hive が Amazon S3 に書き込んだデータにアクセスできます。

以下のステップは、このセクションで示された手順のいずれかを使用して、データが DynamoDB から Amazon S3 にコピーされていることを前提として記述されています。

1. 現在 Hive コマンドプロンプトが表示されている場合は、それを終了し、Linux コマンドプロンプトを表示します。

   ```
   hive> exit;
   ```

2. Amazon S3 バケット内の hive-test ディレクトリの内容を一覧表示します。(このディレクトリには、Hive が DynamoDB からデータをコピーしています)。

   ```
   aws s3 ls s3://aws-logs-123456789012-us-west-2/hive-test/
   ```

   結果は以下のようになります。

   ```
   2016-11-01 23:19:54 81983 000000_0
   ```

このファイル名 (000000_0) は、システムにより生成されています。

3. (オプション) Amazon S3 からリーダーノードのローカルファイルシステムに、データファイルをコピーできます。この操作を完了すると、標準の Linux コマンドラインユーティリティを使用して、ファイル内のデータを操作できるようになります。

```
aws s3 cp s3://aws-logs-123456789012-us-west-2/hive-test/000000_0 .
```

結果は以下のようになります。

```
download: s3://aws-logs-123456789012-us-west-2/hive-test/000000_0
to ./000000_0
```

> **Note**
>
> リーダーノードのローカルファイルシステムには、容量に関する制限があります。ローカルファイルシステム内で使用可能な領域より大きいサイズのファイルに対しては、このコマンドを使用しないでください。

DynamoDB と HDFS 間のデータをコピー

DynamoDB テーブルにあるデータであれば、Hive を使用して Hadoop Distributed File System (HDFS) にコピーすることができます。

DynamoDB からのデータを必要とする MapReduce ジョブを実行している場合などに、この操作を行います。DynamoDB から HDFS にデータをコピーする際には、Amazon EMR クラスターで使用可能なすべてのノードを並行して使用しながら、Hadoop がそのデータの処理を行います。MapReduce ジョブの完了時、その結果を HDFSからDDBに書き込むことができます。

次の例では、Hive が、ここに示す HDFS ディレクトリ `/user/hadoop/hive-test` に対する読み書きを実行しています。

> **Note**
>
> このセクションの例は、チュートリアル:Amazon DynamoDB と Apache Hive の使用 の手順が正確に完了していること、および、ddb_features という名前の外部テーブルが DynamoDB に存在することを前提に書かれています。

トピック

- Hive のデフォルト形式を使用したデータのコピー
- ユーザー指定の形式でデータをコピー
- 列マッピングを使用しないデータをコピー
- HDFS 内のデータへのアクセス

Hive のデフォルト形式を使用したデータのコピー

Example DynamoDB から HDFS へのコピー

`INSERT OVERWRITE` ステートメントを使用することで、HDFS に直接書き込みます。

```
INSERT OVERWRITE DIRECTORY 'hdfs:///user/hadoop/hive-test'
SELECT * FROM ddb_features;
```

HDFS 内に置かれるデータファイルは次のようになります。

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

各フィールドは SOH 文字 (0x01、ヘッダーの開始) で区切られます。ファイルでは、SOH は **^A** と表示されています。

Example HDFS から DynamoDB へのコピー

1. HDFS 内のフォーマットされていないデータにマッピングする外部テーブルを作成します。

```
CREATE EXTERNAL TABLE hdfs_features_unformatted
    (feature_id       BIGINT,
    feature_name      STRING ,
    feature_class     STRING ,
    state_alpha       STRING,
    prim_lat_dec      DOUBLE ,
    prim_long_dec     DOUBLE ,
    elev_in_ft        BIGINT)
```

```
LOCATION 'hdfs:///user/hadoop/hive-test';
```

2. データを DynamoDB にコピーします。

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_unformatted;
```

ユーザー指定の形式でデータをコピー

異なるフィールド区切り文字を使用する場合は、外部テーブルを作成し、そのテーブルをHDFS ディレクトリにマッピングします。この手法は、カンマ区切り値 (CSV) のデータファイルを作成する場合などに使用できます。

Example DynamoDB から HDFS へのコピー

1. HDFS にマッピングする Hive 外部テーブルを作成します。このためには、 DynamoDB 外部テーブルのデータ型と正確に一致しているデータ型を使用します。

```
CREATE EXTERNAL TABLE hdfs_features_csv
    (feature_id       BIGINT,
    feature_name      STRING ,
    feature_class     STRING ,
    state_alpha       STRING,
    prim_lat_dec      DOUBLE ,
    prim_long_dec     DOUBLE ,
    elev_in_ft        BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'hdfs:///user/hadoop/hive-test';
```

2. DynamoDB からデータをコピーします。

```
INSERT OVERWRITE TABLE hdfs_features_csv
SELECT * FROM ddb_features;
```

HDFS 内に置かれるデータファイルは次のようになります。

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
```

```
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example HDFS から DynamoDB へのコピー

単一の HiveQL ステートメントにより、DynamoDB テーブルに HDFS からのデータを書き込むことができます。

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_csv;
```

列マッピングを使用しないデータをコピー

DynamoDB からのデータは、データ型や列マッピングを指定せずにそのままの形式でコピーし、HDFS に書き込むことができます。この手法は、DynamoDB データのアーカイブを作成し、HDFS に保存する場合などに使用できます。

> **Note**
>
> DynamoDB テーブルに Map、List、Boolean、または Null 型の属性が含まれている場合、Hive を使用して DynamoDB から HDFS にデータをコピーするには、これが唯一の方法となります。

Example DynamoDB から HDFS へのコピー

1. DynamoDB テーブルに関連付けられた外部テーブルを作成します。(この HiveQL ステートメントには `dynamodb.column.mapping` はありません)。

   ```
   CREATE EXTERNAL TABLE ddb_features_no_mapping
       (item MAP<STRING, STRING>)
   STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
   TBLPROPERTIES ("dynamodb.table.name" = "Features");
   ```

2. HDFS ディレクトリに関連付けられた別の外部テーブルを作成します。

   ```
   CREATE EXTERNAL TABLE hdfs_features_no_mapping
       (item MAP<STRING, STRING>)
   ```

```
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 'hdfs:///user/hadoop/hive-test';
```

3. DynamoDB から HDFS にデータをコピーします。

```
INSERT OVERWRITE TABLE hdfs_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

HDFS 内に置かれるデータファイルは次のようになります。

```
Name^C{"s":"Soldiers Farewell
 Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"6135"}^BLatitude^C{"n":"32.
Name^C{"s":"Jones
 Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":"1260"}^BLatitude^C{"n":"41.2
Name^C{"s":"Sentinel
 Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":"8133"}^BLatitude^C{"n":"37.
Name^C{"s":"Neversweet
 Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":"2900"}^BLatitude^C{"n":"41
Name^C{"s":"Chacaloochee
 Bay"}^BState^C{"s":"AL"}^BClass^C{"s":"Bay"}^BElevation^C{"n":"0"}^BLatitude^C{"n":"30.6979676
```

各フィールドの先頭には、STX文字 (0x02、テキストの開始) が置かれ、末尾には ETX文字 (0x03、テキストの終わり) が置かれます。ファイルでは、STX は **^B** として表示され、ETX は **^C** として表示されます。

Example HDFS から DynamoDB へのコピー

単一の HiveQL ステートメントにより、DynamoDB テーブルに HDFS からのデータを書き込むことができます。

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM hdfs_features_no_mapping;
```

HDFS 内のデータへのアクセス

HDFS は分散ファイルシステムであり、Amazon EMR クラスター内のすべてのノードからアクセス可能です。SSH を使用してリーダーノードに接続することで、Hive が HDFS に書き込んだデータに対し、コマンドラインツールを使用してのアクセスが可能になります。

HDFSは、リーダーノード上のローカルなファイルシステムとは異なります。標準の Linux コマンド (cat、cp、mv、または rm) を使用して、HDFS 内のファイルやディレクトリはを操作することはできません。代わりに hadoop fs コマンドを使用して、これらのタスクを実行します。

以下のステップは、このセクションで示された手順のいずれかを使用して、データが DynamoDB から HDFS にコピーされていることを前提として記述されています。

1. 現在 Hive コマンドプロンプトが表示されている場合は、それを終了し、Linux コマンドプロンプトを表示します。

   ```
   hive> exit;
   ```

2. HDFS内の、/user/hadoop/hive-test ディレクトリの内容を一覧表示します。(このディレクトリには、Hive が DynamoDB からデータをコピーしています)。

   ```
   hadoop fs -ls /user/hadoop/hive-test
   ```

   結果は以下のようになります。

   ```
   Found 1 items
   -rw-r--r-- 1 hadoop hadoop 29504 2016-06-08 23:40 /user/hadoop/hive-test/000000_0
   ```

   このファイル名 (000000_0) は、システムにより生成されています。
3. ファイルのコンテンツを表示します。

   ```
   hadoop fs -cat /user/hadoop/hive-test/000000_0
   ```

   > **Note**
   >
   > この例では、比較的小さい (約 29 KB) ファイルを使用しています。非常に大きなファイルや、表示不可能な文字を含むファイルに対して、ここでのコマンドを使用する際には注意が必要です。

4. (オプション) データファイルは、HDFS からリーダーノード上のローカルファイルシステムにコピーすることができます。そうすることで、標準の Linux コマンドラインユーティリティを通じて、ファイル内のデータを使用できるようになります。

   ```
   hadoop fs -get /user/hadoop/hive-test/000000_0
   ```

このコマンドでは、ファイルの上書きは行われません。

> **Note**
>
> リーダーノードのローカルファイルシステムには、容量に関する制限があります。ローカルファイルシステム内で使用可能な領域より大きいサイズのファイルに対しては、このコマンドを使用しないでください。

データ圧縮を使用する

Hive を使用して異なるデータソース間でデータをコピーする際には、オンザフライのデータ圧縮を要求できます。Hive では、いくつかの圧縮コーデックが利用可能です。その中の 1 つを Hive セッション中に選択します。これにより、データは指定した形式で圧縮されます。

次の例では、LZO(Lempel-Ziv-Oberhumer)アルゴリズムを使用してデータを圧縮します。

```
SET hive.exec.compress.output=true;
SET io.seqfile.compression.type=BLOCK;
SET mapred.output.compression.codec = com.hadoop.compression.lzo.LzopCodec;

CREATE EXTERNAL TABLE lzo_compression_table (line STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE lzo_compression_table SELECT *
FROM hiveTableName;
```

Amazon S3 内に出力されるファイルは、その末尾には .lzo (例えば、8d436957-57ba-4af7-840c-96c2fc7bb6f5-000000.lzo) が付いたシステム生成の名前が付けられます。

以下の圧縮コーデックを利用できます。

- org.apache.hadoop.io.compress.GzipCodec
- org.apache.hadoop.io.compress.DefaultCodec
- com.hadoop.compression.lzo.LzoCodec
- com.hadoop.compression.lzo.LzopCodec

- org.apache.hadoop.io.compress.BZip2Codec
- org.apache.hadoop.io.compress.SnappyCodec

表示できない UTF-8 文字データの読み込み

表示できない UTF-8 文字データの読み込み、および書き込みを行うには、Hive テーブル作成時に STORED AS SEQUENCEFILE 句を使用します。SequenceFile は、Hadoop バイナリファイル形式で作成されています。このファイルを読み取るには Hadoop を使用する必要があります。次の例に、DynamoDB から Amazon S3 にデータをエクスポートする方法を示します。この機能を使用して印刷不可の UTF-8 でエンコードされた文字を処理できます。

```
CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col array<string>)
STORED AS SEQUENCEFILE
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

パフォーマンスチューニング

DynamoDB テーブルにマッピングする Hive 外部テーブルを作成しても、DynamoDB からの読み込みまたは書き込み容量は消費されません。ただし、Hive テーブルでの読み込み/書き込みのアクティビティ (INSERT または SELECT など) は、基盤となっている DynamoDB テーブルでの読み込み/書き込みオペレーションとして、直接反映されます。

Amazon EMR の Apache Hive では、DynamoDB テーブルにおける I/O 負荷を分散するための独自のロジックが実装されています。また、テーブルでプロビジョニングされたスループットの超過を、最小限に抑制することが試みられます。Amazon EMR は各 Hive クエリの終了時点で、プロビジョニング済みのスループットが超過された回数を含め、実行時のメトリクスを返します。この情報を DynamoDB テーブルの CloudWatch メトリックスと共に使用することで、後続のリクエストのパフォーマンスを向上することにつなげられます。

Amazon EMR コンソールには、クラスターのモニタリングに関する基本的なツールが用意されています。詳細については、「Amazon EMR 管理ガイド」の「クラスターを表示し、モニタリングする」を参照してください。

また、クラスターと Hadoop のジョブをモニタリングするために、Hue、Ganglia、Hadoop ウェブインターフェイスなどのウェブベースのツールを使用することもできます。詳細については、

「Amazon EMR 管理ガイド」の「Amazon EMR クラスターでホストされているウェブインターフェイスの表示」を参照してください。

このセクションでは、外部 DynamoDB テーブルで実行される Hive オペレーションのパフォーマンスを、調整する際に使用できる手順について説明します。

トピック

- DynamoDB のプロビジョニングされたスループット
- マッパーの調整
- その他のトピック

DynamoDB のプロビジョニングされたスループット

外部 DynamoDB テーブルに対して HiveQL ステートメントを発行すると、DynamoDBStorageHandler クラスにより、適切な低レベルの DynamoDB API リクエストが生成されます。このリクエストは、プロビジョニングされたスループットを消費します。十分な読み込みまたは書き込み容量が DynamoDB テーブルにない場合、リクエストはスロットリングされ、HiveQL のパフォーマンス低下を引き起こします。このためテーブルには、十分なスループットキャパシティを確保する必要があります。

例えば、DynamoDB テーブルのために 100 個の読み込みをプロビジョニングしているとします。毎秒読み込むことができるのは、409,600 バイト (100×4 KBの読み込みキャパシティーユニットサイズ) です。ここで、このテーブルに、20 GB (21,474,836,480 バイト) のデータが含まれており、HiveQL を使ってすべてのデータを選択する SELECT ステートメントを使いたいとします。この場合、次のようにクエリの実行にかかる時間を見積もることができます。

21,474,836,480 / 409,600 = 52,429 秒 = 14.56 時間

このシナリオでは、DynamoDB テーブルがボトルネックになります。Hive のスループットが 1 秒あたり 409,600 バイトに制限されているため、Amazon EMR ノードを追加することはできません。SELECT ステートメントの処理に必要な時間を短縮する唯一の方法は、DynamoDB テーブルのプロビジョニングされた読み込みキャパシティを増やすことです。

同様の計算により、DynamoDB テーブルにマッピングされた Hive 外部テーブルに、データを一括ロードする際に必要となる時間も推定できます。項目ごとに必要な書き込みキャパシティーユニットの合計数 (1KB = 1 未満、1 ~ 2KB = 2 など) を決定し、それにロードする項目数を掛けます。これにより、必要な書き込みキャパシティーユニットの数がわかります。その数を、1 秒あたりに割り当て

られる書き込みキャパシティーユニットの数で割ります。これにより、テーブルのロードに要する秒数が計算できます。

テーブルの CloudWatch メトリクスは、定期的にモニタリングする必要があります。DynamoDB コンソールでその概要を簡単に確認するには、対象のテーブルを選択し、[Metrics] (メトリクス) タブを開きます。消費された読み込み/書き込みキャパシティーユニット数、および、スロットリングされた読み込み/書き込みリクエスト数を表示できます。

読み込みキャパシティ

Amazon EMR は、テーブルでのプロビジョニング済みスループットの設定に応じて、DynamoDB テーブルに対するリクエスト負荷を管理します。しかし、ジョブ出力に多数の ProvisionedThroughputExceeded メッセージが見られる場合には、デフォルトの読み込みレートを調整することも可能です。このためには、dynamodb.throughput.read.percent 設定変数を変更できます。Hive コマンドプロンプトで SET コマンドを使用して、この変数を設定できます。

```
SET dynamodb.throughput.read.percent=1.0;
```

この変数は、その時点の Hive セッションの間のみ有効です。一度 Hive を終了し、再び再開した場合には、dynamodb.throughput.read.percent はデフォルト値に戻ります。

dynamodb.throughput.read.percent の値は 0.1 と 1.5 の間で設定できます。0.5 は、デフォルトの読み込みレートを表し、Hive がテーブルの読み込みキャパシティの半分を消費しようとすることを意味します。上記の値を 0.5 より上に設定すると、Hive は読み込みリクエストレートを増加させ、0.5 より低くした場合はリクエストレートを減少させます。(実際の読み込みレートは、DynamoDB テーブルに、統一ディストリビューションのキーがあるかどうかなどの要因によって変わります。)

テーブルのプロビジョニングされた読み込みキャパシティを、Hive が頻繁に枯渇させている場合、または読み込みリクエストのスロットリング頻度が高すぎる場合は、dynamodb.throughput.read.percent を 0.5 より低く設定してみてください。テーブルに十分な読み込みキャパシティがある場合で、HiveQL オペレーションにさらに高い応答性を求める場合には、この値を 0.5 より高く設定できます。

書き込みキャパシティ

Amazon EMR は、テーブルでのプロビジョニング済みスループットの設定に応じて、DynamoDB テーブルに対するリクエスト負荷を管理します。しかし、ジョブ出力に多数の ProvisionedThroughputExceeded メッセージが見られる場合には、デフォルトの書き込みレートを調整することも可能です。このためには、dynamodb.throughput.write.percent 設定変

数を変更できます。Hive コマンドプロンプトで SET コマンドを使用して、この変数を設定できます。

```
SET dynamodb.throughput.write.percent=1.0;
```

この変数は、その時点の Hive セッションの間のみ有効です。一度 Hive を終了し、再び再開した場合には、dynamodb.throughput.write.percent はデフォルト値に戻ります。

dynamodb.throughput.write.percent の値は 0.1 と 1.5 の間で設定できます。0.5 は、デフォルトの書き込みレートを表し、Hive がテーブルの書き込みキャパシティの半分を消費しようとすることを意味します。上記の値を 0.5 より上に設定すると、Hive は書き込みリクエストレートを増加させ、0.5 より低くした場合はリクエストレートを減少させます。(実際の書き込みレートは、DynamoDB テーブルに、統一ディストリビューションのキーがあるかどうかなどの要因によって変わります。)

テーブルのプロビジョニングされた書き込みキャパシティを、Hive が頻繁に枯渇させている場合、または書き込みリクエストのスロットリング頻度が高すぎる場合は、dynamodb.throughput.write.percent を 0.5 より低く設定してみてください。テーブルに十分なキャパシティがある場合で、HiveQL オペレーションにさらに高い応答性を求める場合には、この値を 0.5 より高く設定できます。

Hive を使用してデータを DynamoDB に書き込む場合は、書き込みキャパシティユニットの数をクラスター内のマッパーの数より大きくする必要があります。例えば、Amazon EMR クラスターが 10 の m1.xlarge ノードで構成されているとします。m1.xlarge ノードタイプでは、8 のマッパータスクが提供されるため、クラスターには合計 80 のマッパー (10 × 8) が存在することになります。DynamoDB テーブルにある書き込みキャパシティーユニットが 80 個未満の場合、Hive による書き込みオペレーションが、そのテーブルの書き込みスループットをすべて消費してしまう可能性があります。

Amazon EMR での各ノードタイプのマッパーの数については、「Amazon EMR デベロッパーガイド」の「タスクの設定」を参照してください。

マッパーの詳細については、「マッパーの調整」を参照してください。

マッパーの調整

Hive が Hadoop ジョブを起動すると、このジョブは 1 つ以上のマッパータスクによって処理されます。DynamoDB テーブルに十分なスループットキャパシティが用意されているのであれば、クラスター内のマッパーの数が変更可能なので、これによりパフォーマンスを向上させられる可能性があります。

> **Note**
>
> Hadoop ジョブで使用されるマッパータスクの数は、Hadoop がデータを論理ブロックに分割する入力分割による影響を受けます。Hadoop により十分な入力分割が行われない場合、書き込みオペレーションが、DynamoDB テーブルで使用可能なすべての書き込みスループットを消費できないことがあります。

マッパーの数を増やします

Amazon EMR の各マッパーの最大読み込みレートは、1 秒あたり 1 MiB です。クラスター内のマッパーの数は、そのクラスター内のノードのサイズによって異なります。(ノードのサイズとノードあたりのマッパー数については、「Amazon EMR デベロッパーガイド」の「タスクの設定」を参照してください。)

DynamoDB テーブルに十分な読み込み用スループット容量がある場合は、次のいずれかの方法でマッパーの数を増やすことができます。

- クラスター内のノードのサイズを大きくする。例えば、クラスターで (ノードごとに 3 つのマッパー) m1.large ノードを使用しているのであれば、これを (ノードごとに 8 つのマッパー) m1.xlarge ノードにアップグレードしてみることができます。
- クラスター内のノード数を増やす。たとえば、3 つの m1.xlarge ノードを持つクラスターであれば、合計 24 個のマッパーを使用できます。同じタイプのノードでクラスターのサイズを 2 倍にすれば、48 個のマッパーが使用可能になります。

AWS マネジメントコンソール マネジメントコンソールを使用して、クラスター内のノードのサイズまたは数を管理することができます。(これらの変更を有効にするには、クラスターの再起動が必要な場合があります。)

マッパーの数を増やすもう 1 つの方法は、`mapred.tasktracker.map.tasks.maximum` Hadoop の設定パラメータを変更することです。(これは Hadoop パラメータであり、Hive パラメータではないため、コマンドプロンプトからインタラクティブに変更することはできません。) `mapred.tasktracker.map.tasks.maximum` の値を増加することで、ノードのサイズや数を増やすことなく、マッパーの数を増やすことができます。ただし、この値を高く設定しすぎると、クラスターノードのメモリが不足する可能性があります。

`mapred.tasktracker.map.tasks.maximum` の値は、Amazon EMR クラスターを初めて起動する際のブートストラップアクションの中で設定します。詳細については、「Amazon EMR 管理ガイ

ド」の「(オプション) 追加のソフトウェアをインストールするためのブートストラップアクションの作成」を参照してください。

マッパーの数を減らします

SELECT ステートメントを使用して DynamoDB にマッピングする外部 Hive テーブルからデータを選択する場合、Hadoop ジョブはクラスター内のマッパーの最大数まで、必要な数のタスクを使用できます。このシナリオでは、長時間実行される Hive クエリが DynamoDB テーブルのプロビジョニングされた読み込みキャパシティーをすべて消費することで、他のユーザーに悪影響を与える可能性があります。

dynamodb.max.map.tasks パラメータを使用して、マップタスクの上限を設定することができます。

```
SET dynamodb.max.map.tasks=1
```

この値は、1 以上にする必要があります。Hive がクエリを処理する際に実行される Hadoop ジョブは、DynamoDB テーブルからの読み込み時、dynamodb.max.map.tasks を超えるタスクを使用することはありません。

その他のトピック

Hive を使用して DynamoDB にアクセスするアプリケーションを調整するための、他の方法を以下に示します。

再試行の期間

デフォルトでは、2 分以内に DynamoDB からの結果が返されない場合、Hive は Hadoop ジョブを再実行します。この間隔は、以下のように dynamodb.retry.duration パラメータを変更することで調整できます。

```
SET dynamodb.retry.duration=2;
```

この値は、再試行間隔 (分) を表す 0 以外の整数である必要があります。dynamodb.retry.duration のデフォルトは 2 (分) です。

並列データリクエスト

複数のユーザーまたは複数のアプリケーションから、単一のテーブルに複数のデータリクエストが行われると、プロビジョニング済み読み込みスループットの枯渇につながり、パフォーマンス速度を低下させることがあります。

処理間隔

DynamoDB におけるデータの整合性は、各ノードで実行される読み込み/書き込みオペレーションの順序に影響を受けます。Hive クエリの進行中に、別のアプリケーションが DynamoDB テーブルに新しいデータをロードしたり、既存のデータの変更や削除を行ったりする場合があります。この場合、クエリの実行中にデータに対して行われた変更は Hive クエリの結果に反映されないことがあります。

[Request time] (リクエストタイム)

DynamoDB テーブルの需要が低いタイミングでテーブルにアクセスするように、Hive クエリをスケジューリングすると、パフォーマンスを向上させられます。例えば、アプリケーションのほとんどのユーザーがサンフランシスコに在住の場合、大部分のユーザーが就寝して DynamoDB データベースのレコードを更新することがない午前 4 時 (PST) に、毎日のデータエクスポートを行います。

DynamoDB の Amazon S3 との統合

Amazon DynamoDB のインポートおよびエクスポート機能は、コードを記述せずに Amazon S3 と DynamoDB のテーブル間でデータを移動するシンプルで効率的な方法を提供します。

DynamoDB のインポートおよびエクスポート機能は、DynamoDB テーブルアカウントの移動、変換、コピーに役立ちます。S3 ソースからのインポートや、DynamoDB テーブルデータの Amazon S3 へのエクスポートにより、Athena、Amazon SageMaker AI、AWS Lake Formation などの AWS のサービスを使用してデータを分析し、実用的なインサイトを抽出できます。また、データを新しい DynamoDB テーブルに直接インポートして、大規模な 1 桁ミリ秒のパフォーマンスで新しいアプリケーションを構築し、テーブルとアカウント間のデータ共有を容易にして、ディザスタリカバリとビジネス継続性計画を簡素化することもできます。

トピック

- Amazon S3 からの DynamoDB データのインポート: 仕組み
- Amazon S3 への DynamoDB データのエクスポート: 仕組み

Amazon S3 からの DynamoDB データのインポート: 仕組み

データを DynamoDB にインポートするには、データが CSV、DynamoDB JSON、または Amazon Ion 形式で Amazon S3 バケット内にある必要があります。データは ZSTD または GZIP 形式で圧縮することも、非圧縮形式で直接インポートすることもできます。ソースデータは、単一の Amazon

S3 オブジェクトでも、同じプレフィックスを使用する複数の Amazon S3 オブジェクトでもかまいません。

データは新しい DynamoDB テーブルにインポートされ、インポートリクエストを開始すると作成されます。グローバルセカンダリインデックス (ローカルセカンダリインデックスはサポートされません) を使用してこのテーブルを作成し、インポートが完了したらすぐに、すべてのプライマリインデックスとセカンダリインデックスでデータのクエリと更新を行うことができます。インポートした後にグローバルテーブルレプリカを追加することもできます。

> **Note**
>
> Amazon S3 のインポートプロセス中に、DynamoDB はインポートされる新しいターゲットテーブルを作成します。既存のテーブルへのインポートは、現在この機能ではサポートされていません。

Amazon S3 からのインポートでは、新しいテーブルの書き込み容量が消費されないため、データを DynamoDB にインポートするために追加の容量をプロビジョニングする必要はありません。データインポートの料金は、インポートの結果として処理される Amazon S3 のソースデータの非圧縮サイズに基づいています。処理されたが、ソースデータの形式やその他の不整合のためにテーブルに読み込めなかった項目も、インポートプロセスの一部として請求されます。詳細については、「Amazon DynamoDB の料金表」を参照してください。

別のアカウントによって所有されている Amazon S3 バケットからデータをインポートするには、そのバケットから読み取るための正しい許可が必要です。新しいテーブルは、ソース Amazon S3 バケットとは異なるリージョンにある場合もあります。手順については、「Amazon Simple Storage Service の設定とアクセス許可」を参照してください。

インポート時間は、Amazon S3 におけるデータの特性に直接関係します。これには、データサイズ、データ形式、圧縮スキーム、データ分散の均一性、Amazon S3 オブジェクトの数、およびその他の関連変数が含まれます。特に、均一に分散されたキーを持つデータセットは、歪んだデータセットよりもインポートが高速になります。例えば、セカンダリインデックスのキーが月をパーティショニングに使用し、すべてのデータが 12 月のものだった場合、このデータのインポートにはかなり時間がかかることがあります。

キーに関連付けられた属性は、ベーステーブル上で一意であることが想定されます。一意でないキーがある場合、インポートによって、最後の上書きだけが残るまで、関連する項目が上書きされます。例えば、プライマリキーが月で、複数の項目が 9 月に設定されている場合、新しい項目はそれぞれ以前に書き込まれた項目を上書きし、「月」のプライマリキーが 9 月に設定された 1 つの項目だけ

が残ります。このような場合、インポートテーブルの説明で処理される項目の数は、ターゲットテーブルの項目数と一致しません。

AWS CloudTrailテーブルインポートのコンソールと API アクションは、すべて に記録されます。詳細については、「AWS CloudTrail を使用して DynamoDB オペレーションを口グに記録する」を参照してください。

次の動画では、Amazon S3 から DynamoDB に直接インポートする方法について紹介します。

Amazon S3 からのインポート

トピック

- DynamoDB でテーブルのインポートをリクエストする
- DynamoDB 用の Amazon S3 のインポート形式
- インポート形式のクォータと検証
- Amazon S3 から DynamoDB にインポートするためのベストプラクティス

DynamoDB でテーブルのインポートをリクエストする

DynamoDB インポートでは、Amazon S3 バケットから新しい DynamoDB テーブルにデータをインポートできます。テーブルのインポートをリクエストするには、DynamoDB コンソール、CLI、CloudFormation、または DynamoDB を使用できます。

AWS CLI を使用する場合は、最初に設定する必要があります。詳細については、「DynamoDB にアクセスする」を参照してください。

> **Note**
>
> - テーブルのインポート機能は、複数の異なる AWS のサービス (Amazon S3 や CloudWatch など) と連携して動作します。インポートを開始する前に、インポート API を呼び出すユーザーまたはロールに、機能が依存するすべてのサービスとリソースに対するアクセス許可があることを確認します。
> - インポートの進行中に Amazon S3 オブジェクトを変更しないでください。オペレーションが失敗したり、キャンセルされたりする可能性があります。
>
> エラーとトラブルシューティングの詳細については、「インポート形式のクォータと検証」を参照してください。

トピック

- IAM アクセス許可のセットアップ
- AWS マネジメントコンソール を使用してインポートをリクエストする
- AWS マネジメントコンソールで過去のインポートの詳細を取得する
- AWS CLI を使用してインポートをリクエストする
- AWS CLIで過去のインポートの詳細を取得する

IAM アクセス許可のセットアップ

データは、読み取りアクセス許可を持っている Amazon S3 バケットであれば、どのバケットからでもインポートを実行できます。インポート元のバケットは、ソーステーブルと同じリージョンに存在する必要はなく、所有者が同じである必要もありません。AWS Identity and Access Management (IAM) には、ソース Amazon S3 バケットの関連アクションと、デバッグ情報を提供するために必要な CloudWatch アクセス許可を含める必要があります。次にポリシーの例を示します。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowDynamoDBImportAction",
      "Effect": "Allow",
      "Action": [
        "dynamodb:ImportTable",
        "dynamodb:DescribeImport"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table*"
    },
    {
      "Sid": "AllowS3Access",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::your-bucket/*",
        "arn:aws:s3:::your-bucket"
```

```
      ]
    },
    {
      "Sid": "AllowCloudwatchAccess",
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:DescribeLogStreams",
        "logs:PutLogEvents",
        "logs:PutRetentionPolicy"
      ],
      "Resource": "arn:aws:logs:us-east-1:111122223333:log-group/aws-dynamodb/*"
    },
    {
      "Sid": "AllowDescribeLogGroups",
      "Effect": "Allow",
      "Action": "logs:DescribeLogGroups",
      "Resource": "*"
    },
    {
      "Sid": "AllowDynamoDBListImports",
      "Effect": "Allow",
      "Action": "dynamodb:ListImports",
      "Resource": "*"
    }
  ]
}
```

Amazon S3 のアクセス許可

別のアカウントが所有する Amazon S3 バケットソースでインポートを開始するときは、ロールまたはユーザーが Amazon S3 オブジェクトにアクセスできることを確認します。Amazon S3 の `GetObject` コマンドを実行し、認証情報を使用することで、これを確認できます。API を使用する場合、Amazon S3 バケット所有者パラメータは、デフォルトで現在のユーザーのアカウント ID に設定されます。クロスアカウントインポートの場合は、このパラメータにバケット所有者のアカウント ID が正しく入力されていることを確認します。次のコードは、ソースアカウントの Amazon S3 バケットポリシーの例です。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {"Sid": "ExampleStatement",
            "Effect": "Allow",
            "Principal": {"AWS": "arn:aws:iam::123456789012:user/Dave"
            },
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
        }
    ]
}
```

AWS Key Management Service

インポート用の新しいテーブルを作成するときに、DynamoDB が所有していない保管時の暗号化キーを選択した場合は、カスタマーマネージドキーで暗号化された DynamoDB テーブルを操作するために必要なアクセス許可を AWS KMS に付与する必要があります。詳細については、「AWS KMS キーの使用を認可する」を参照してください。Amazon S3 オブジェクトがサーバー側の暗号化 KMS (SSE-KMS) で暗号化されている場合は、インポートを開始するロールまたはユーザーが、AWS KMS キーを使用して復号するアクセス権を持っていることを確認してください。この機能は、お客様が指定したキーによるサーバー側の暗号化 (SSE-C) で暗号化された Amazon S3 オブジェクトはサポートしません。

CloudWatch のアクセス許可

インポートを開始するロールまたはユーザーには、インポートに関連付けられたロググループとログストリームに対する作成と管理のアクセス許可が必要です。

AWS マネジメントコンソール を使用してインポートをリクエストする

DynamoDB コンソールを使用して、`MusicCollection` という新しいテーブルに既存のデータをインポートする方法を次の例に示します。

テーブルのインポートをリクエストするには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[Import from S3] (S3 からインポート) を選択します。
3. 表示されたページで、[Import from S3] (S3 からインポート) を選択します。
4. [Import from S3] (S3 からインポート) を選択します。
5. [ソース S3 の URL] に Amazon S3 のソース URL を入力します。

   ソースバケットを所有している場合は、[S3 を参照] を選択してバケットを検索します。または、バケットの URL を `s3://bucket/prefix` の形式で入力します。`prefix` は Amazon S3 キープレフィックスです。これは、インポートする Amazon S3 オブジェクトの名前であるか、インポートするすべての Amazon S3 オブジェクトが共有するキープレフィックスのいずれかです。

   > **Note**
   >
   > DynamoDB エクスポートリクエストと同じプレフィックスを使用することはできません。エクスポート機能は、すべてのエクスポートのフォルダ構造とマニフェストファイルを作成します。同じ Amazon S3 パスを使用すると、エラーが発生します。
   > 代わりに、特定のエクスポートからのデータを含むフォルダにインポートを指定します。この場合、正しいパスの形式は `s3://bucket/prefix/AWSDynamoDB/<XXXXXXXX-XXXXXX>/data/` になります。ここで、`XXXXXXXX-XXXXXX` はエクスポート ID です。エクスポート ID はエクスポート ARN にあり、形式は `arn:aws:dynamodb:<Region>:<AccountID>:table/<TableName>/export/<XXXXXXXX-XXXXXX>` です。例えば、`arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/export/01234567890123-a1b2c3d4`。

6. 自分が [S3 バケット所有] であることを確認します。ソースバケットが別のアカウントによって所有されている場合は、[別の AWS アカウント] を選択します。次に、バケット所有者のアカウント ID を入力します。
7. [ファイル圧縮をインポート] で、[圧縮なし]、[GZIP]、または [ZSTD] を適宜選択します。

8. 適切なインポートファイルフォーマットを選択します。オプションは、[DynamoDB JSON]、[Amazon Ion]、または [CSV] です。[CSV] を選択する場合は、[CSV ヘッダー] および [CSV 区切り文字] の 2 つの追加オプションがあります。

   [CSV ヘッダー] を使用する場合は、ヘッダーをファイルの最初の行から取得するか、カスタマイズするかを選択します。[Customize your headers] (ヘッダーをカスタマイズ) を選択した場合は、インポートに使用するヘッダーの値を指定できます。この方法で指定される CSV ヘッダーは大文字と小文字が区別され、ターゲットテーブルのキーを含むことが想定されます。

   [CSV 区切り文字] では、項目を区切る文字を設定します。デフォルトではカンマが選択されます。[カスタム区切り文字] を選択した場合、区切り文字は正規表現パターン `[,;:|\t ]` と一致する必要があります。

9. [次へ] ボタンを選択し、データを保存するために作成される新しいテーブルのオプションを選択します。

   > **Note**
   >
   > プライマリキーとソートキーはファイル内の属性と一致する必要があります。一致しない場合、インポートは失敗します。属性では、大文字と小文字が区別されます。

10. [次へ] をもう一度選択してインポートオプションを確認し、[インポート] を選択してインポートタスクを開始します。まず、新しいテーブルが「作成中」のステータスで「テーブル」にリストされます。この時点では、テーブルにアクセスできません。
11. インポートが完了すると、ステータスが [アクティブ] と表示され、テーブルの使用を開始できます。

AWS マネジメントコンソールで過去のインポートの詳細を取得する

過去に実行したインポートタスクに関する情報は、ナビゲーションサイドバーの [Import from S3] (S3 からインポート) を選択し、[インポート] タブを選択して表示できます。インポートパネルには、過去 90 日間に作成されたすべてのインポートのリストが表示されます。[インポート] タブにリストされているタスクの ARN を選択すると、選択した詳細設定など、そのインポートに関する情報が取得されます。

AWS CLI を使用してインポートをリクエストする

次の例では、プレフィックスのプレフィックスが bucket という名前の S3 バケットから、target-table という新しいテーブルに CSV 形式のデータをインポートします。

```
aws dynamodb import-table --s3-bucket-source S3Bucket=bucket,S3KeyPrefix=prefix \
            --input-format CSV --table-creation-parameters '{"TableName":"target-
table","KeySchema":  \
            [{"AttributeName":"hk","KeyType":"HASH"}],"AttributeDefinitions":
[{"AttributeName":"hk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}' \
            --input-format-options '{"Csv": {"HeaderList": ["hk", "title", "artist",
 "year_of_release"], "Delimiter": ";"}}'
```

> **Note**
>
> AWS Key Management Service (AWS KMS) によって保護されたキーを使用してインポートを暗号化することを選択した場合、そのキーはインポート先 Amazon S3 バケットと同じリージョンにある必要があります。

AWS CLIで過去のインポートの詳細を取得する

過去に実行したインポートタスクに関する情報は、`list-imports` コマンドを実行して見つけることができます。このコマンドは、過去 90 日間に作成されたすべてのインポートのリストを返します。インポートタスクのメタデータは 90 日後に期限切れになり、それより古いジョブはこのリストに表示されなくなりますが、Amazon S3 バケット内のオブジェクトまたはインポート中に作成されたテーブルは削除されません。

```
aws dynamodb list-imports
```

詳細な設定など、特定のインポートタスクに関する詳細情報を取得するには、`describe-import` コマンドを使用します。

```
aws dynamodb describe-import \
    --import-arn arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/exp
```

DynamoDB 用の Amazon S3 のインポート形式

DynamoDB は、CSV、DynamoDB JSON、および Amazon Ion の 3 つの形式でデータをインポートできます。

トピック

- CSV

- DynamoDB Json
- Amazon Ion

CSV

CSV 形式のファイルは、改行で区切られた複数の項目で構成されます。デフォルトでは、DynamoDB はインポートファイルの最初の行をヘッダーとして解釈し、列がカンマで区切られることを想定しています。ファイル内の列数と一致する限り、適用されるヘッダーを定義することもできます。ヘッダーを明示的に定義すると、ファイルの最初の行が値としてインポートされます。

> **Note**
>
> CSV ファイルからインポートする場合、ベーステーブルとセカンダリインデックスのハッシュ範囲とキー以外のすべての列が DynamoDB 文字列としてインポートされます。

二重引用符のエスケープ

CSV ファイルに存在する二重引用符文字はすべてエスケープする必要があります。エスケープしないと、次の例に示すように、インポートは失敗します。

```
id,value
"123",Women's Full "Length" Dress
```

引用符を 2 組の二重引用符でエスケープすると、この同じインポートが成功します。

```
id,value
"""123""","Women's Full ""Length"" Dress"
```

テキストを適切にエスケープしてインポートすると、元の CSV ファイルと同じように表示されます。

```
id,value
"123",Women's Full "Length" Dress
```

DynamoDB は標準の CSV 引用規則 (RFC 4180) に従います。フィールド値内の二重引用符は、二重引用符 ("") をもう 1 つ前に付けることでエスケープされます。バックスラッシュによるエスケープ処理 (\") は、プログラミング言語や多くのデータエクスポートツールの出力では一般的ですが、認

識されず、これを使用するファイルはインポートに失敗します。ソースデータでバックスラッシュによるエスケープ処理を使用している場合は、インポートする前に、エスケープされた各引用符を二重引用符に変換します。例えば、次のようにエクスポートされる値があるとします。

```
"Women's \"Full Length\" Dress"
```

この値は、インポートする前に次のように再フォーマットする必要があります。

```
"Women's ""Full Length"" Dress"
```

キー値のフォーマット

DynamoDB では、プライマリキーはシンプルなプライマリキー (パーティションキー) または複合プライマリキー (パーティションキーとソートキー) のいずれかです。各キー属性は 1 つのスカラー値を保持します。例えば、organizationId と barcode を区切り文字を使用して organizationId#barcode のように結合するなど、複数のフィールドを組み合わせたキー値が必要な場合は、それらのフィールドを 1 つの文字列に連結し、1 つのキー属性に保存します。この連結はアプリケーション規則です。DynamoDB はキー属性を 1 つの値として保存して照合します。

このデータを CSV からインポートするときは、ヘッダーがキー属性の名前と一致する 1 つの列に完全結合値を入力します。パートを別々の列に分割しないでください。インポートでは各列が 1 つの属性にマッピングされ、列はキー値に連結されません。例えば、パーティションキー属性の名前が pk、ソートキー属性の名前が sk であるテーブルの場合:

```
pk,sk,barcode
org123#ABC123,number#001,ABC123
org456#XYZ789,number#002,XYZ789
```

キー値自体にカンマ区切り文字が含まれている場合は、値全体を二重引用符で囲みます (例: "org123,east#ABC123")。

異種項目タイプのインポート

1 つの CSV ファイルを使用して、異なる項目タイプを 1 つのテーブルにインポートできます。項目タイプ全体のすべての属性を含むヘッダー行を定義し、特定の項目に適用されない属性の列を空のままにします。空の列は、空の文字列として保存されるのではなく、インポートされた項目から省略されます。

```
PK,SK,EntityType,Name,Email,OrderDate,Amount,ProductName,Quantity
```

```
USER#1,PROFILE,User,Alice,alice@example.com,,,,
USER#1,ORDER#2024-01-15,Order,,,2024-01-15,99.99,,
USER#1,ORDER#2024-02-10,Order,,,2024-02-10,149.50,,
PRODUCT#101,METADATA,Product,,,,,Laptop,50
PRODUCT#102,METADATA,Product,,,,,Mouse,200
USER#2,PROFILE,User,Bob,bob@example.com,,,,
USER#2,ORDER#2024-01-20,Order,,,2024-01-20,75.00,,
PRODUCT#103,METADATA,Product,,,,,Keyboard,150
USER#3,PROFILE,User,Charlie,charlie@example.com,,,,
PRODUCT#104,METADATA,Product,,,,,Monitor,30
```

この例では、ユーザープロファイル、注文、製品が同じテーブルを共有します。各項目タイプは、関連する列のみを使用します。

DynamoDB Json

DynamoDB JSON 形式のファイルは、複数の Item オブジェクトで構成できます。個々のオブジェクトは DynamoDB のスタンダードマーシャリングされた JSON 形式で、改行が項目区切り文字として使用されます。これは JSON Lines (JSONL) 形式です。1 行に 1 つの JSON オブジェクトで、外側の配列や項目間のカンマはありません。追加機能として、特定の時点からのエクスポートは、デフォルトでインポートソースとしてサポートされています。

> **Note**
>
> 新しい行は DynamoDB JSON 形式のファイルの項目区切り文字として使用されるため、項目オブジェクト内では使用しないでください。

```
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "333-3333333333"}, "Id": {"N": "103"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 103 Title"}}}
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "444-444444444"}, "Id": {"N": "104"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 104 Title"}}}
{"Item": {"Authors": {"SS": ["Author1", "Author2"]}, "Dimensions": {"S": "8.5 x 11.0 x
 1.5"}, "ISBN": {"S": "555-5555555555"}, "Id": {"N": "105"}, "InPublication": {"BOOL":
 false}, "PageCount": {"N": "600"}, "Price": {"N": "2000"}, "ProductCategory": {"S":
 "Book"}, "Title": {"S": "Book 105 Title"}}}
```

Amazon Ion

Amazon Ion は、大規模なサービス指向アーキテクチャをエンジニアリングしながら、急速な開発、デカップリング、効率性といった日々の課題に対処するために構築されたリッチタイプの自己記述型階層型データシリアル化フォーマットです。

データを Ion 形式でインポートすると、Ion データ型は新しい DynamoDB テーブルの DynamoDB データ型にマップされます。

S 番号	Ion から DynamoDB へのデータ型変換	B
1	`Ion Data Type`	`DynamoDB Representation`
2	`string`	`String (s)`
3	`bool`	`Boolean (BOOL)`
4	`decimal`	`Number (N)`
5	`blob`	`Binary (B)`
6	`list (with type annotation $dynamodb_SS, $dynamodb_NS, or $dynamodb_BS)`	`Set (SS, NS, BS)`
7	`list`	`List`
8	`struct`	`Map`

Ion ファイル内の項目は、改行で区切られます。各行は Ion バージョンマーカーで始まり、Ion 形式の項目が続きます。

> **Note**
>
> 次の例では、可読性を高めるため、Ion 形式ファイルの項目を複数の行にフォーマットしています。

```
$ion_1_0
[
  {
    Item:{
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"333-3333333333",
      Id:103.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 103 Title"
    }
  },
  {
    Item:{
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"444-4444444444",
      Id:104.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 104 Title"
    }
  },
  {
    Item:{
      Authors:$dynamodb_SS::["Author1","Author2"],
      Dimensions:"8.5 x 11.0 x 1.5",
      ISBN:"555-5555555555",
      Id:105.,
      InPublication:false,
      PageCount:6d2,
      Price:2d3,
      ProductCategory:"Book",
      Title:"Book 105 Title"
    }
  }
]
```

インポート形式のクォータと検証

インポートクォータ

us-east-1、us-west-2、および eu-west-1 の各リージョンでは、Amazon S3 からの DynamoDB のインポートは、一度に 15 TB の合計インポートソースオブジェクトサイズで 50 個の同時インポートジョブに対応できます。他のすべてのリージョンでは、合計サイズが 1 TB で最大 50 個の同時インポートタスクに対応できます。各インポートジョブでは、すべてのリージョンで最大 50,000 個の Amazon S3 オブジェクトを使用できます。これらのデフォルトクォータはすべてのアカウントに適用されます。これらのクォータを改訂する必要があると考える場合は、アカウントチームに連絡してください。ケースバイケースで検討されます。DynamoDB の制限の詳細については、「Service Quotas」を参照してください。

検証エラー

インポートプロセス中に、DynamoDB でデータの解析中にエラーが発生することがあります。エラーごとに、DynamoDB は CloudWatch ログを出力し、発生したエラーの総数を記録します。Amazon S3 オブジェクト自体の形式が正しくない場合や、その内容が DynamoDB 項目を形成できない場合は、オブジェクトの残りの部分の処理がスキップされることがあります。

> **Note**
>
> Amazon S3 データソースに同じキーを共有する項目が複数ある場合、項目は 1 つが残るまで上書きされます。この場合、1 つの項目がインポートされ、他の項目が無視されたかのように見えることがあります。重複した項目はランダムな順序で上書きされ、エラーとしてカウントされず、CloudWatch ログにも出力されません。
> インポートが完了すると、インポートされた項目の総数、エラーの総数、および処理された項目の合計数が表示されます。トラブルシューティングをさらに進めるには、インポートされた項目の合計サイズと処理されたデータの合計サイズを確認することもできます。

インポートエラーには、API 検証エラー、データ検証エラー、および設定エラーの 3 つのカテゴリがあります。

API 検証エラー

API 検証エラーは、同期 API からの項目レベルのエラーです。一般的な原因として、アクセス許可の問題、必須パラメータの欠落、およびパラメータ検証の失敗が挙げられます。API コールが失敗した理由の詳細は、`ImportTable` リクエストによってスローされる例外に含まれています。

データ検証エラー

データ検証エラーは、項目レベルまたはファイルレベルで発生する可能性があります。インポート中、ターゲットテーブルにインポートする前に、DynamoDB ルールに基づいて項目が検証されます。項目が検証に失敗し、インポートされない場合、インポートジョブはその項目をスキップし、次の項目に進みます。ジョブの終了時に、インポートのジョブは FAILED に設定され、FailureCode、ItemValidationError、および FailureMessage「Some of the items failed validation checks and were not imported。Please check CloudWatch error logs for more details.」(一部の項目は検証チェックに失敗したため、インポートされませんでした。詳細については、CloudWatch エラーログを確認してください) が表示されます。

データ検証エラーの一般的な原因として、オブジェクトが解析できない、オブジェクトの形式が正しくない (入力で DYNAMODB_JSON が指定されているが、オブジェクトが DYNAMODB_JSON でない)、および指定されたソーステーブルキーとスキーマの不一致が挙げられます。

設定エラー

設定エラーは、通常、アクセス許可の検証によるワークフローエラーです。インポートワークフローでは、リクエストを受け入れた後、一部のアクセス許可がチェックされます。Amazon S3 や CloudWatch など、必要な依存関係の呼び出しで問題が発生した場合、プロセスでインポートステータスが FAILED とマークされます。`failureCode` および `failureMessage` は、失敗の理由を指します。該当する場合、失敗メッセージには、CloudTrail での失敗の理由を調査するために使用できるリクエスト ID も含まれます。

よくある設定エラーとして、Amazon S3 バケットの URL が間違っていることや、Amazon S3 バケット、CloudWatch Logs、および Amazon S3 オブジェクトの復号に使用される AWS KMS キーにアクセスする許可がないことが挙げられます。詳細については、「Using and data keys」(データキーの使用) を参照してください。

ソース Amazon S3 オブジェクトの検証

ソース S3 オブジェクトの検証を行うには、次のステップを実行します。

1. データ形式と圧縮タイプの検証

   - 指定したプレフィックスで一致するすべての Amazon S3 オブジェクトが、同じ形式 (DYNAMODB_JSON、DYNAMODB_ION、CSV) であることを確認します。
   - 指定したプレフィックスで一致するすべての Amazon S3 オブジェクトが、同じ方法 (GZIP、ZSTD、NONE) で圧縮されていることを確認します。

> **Note**
>
> ImportTable 呼び出しで指定された入力形式が優先されるため、Amazon S3 オブジェクトには、対応する拡張子 (.csv、.json、.ion、.gz、.zstd など) は必要はありません。

2. インポートデータが目的のテーブルスキーマに準拠していることを検証します

   - ソースデータの各項目にプライマリキーがあることを確認します。ソートキーはインポートのオプションです。
   - プライマリキーとソートキーに関連付けられた属性タイプが、テーブル作成パラメータで指定されているように、テーブルおよび GSI スキーマの属性タイプと一致することを確認します。

トラブルシューティング

CloudWatch ログ

インポートジョブが失敗した場合、詳細なエラーメッセージが CloudWatch Logs に投稿されます。これらのログにアクセスするには、まず出力から ImportArn を取得し、次のコマンドを使用して describe-import を実行します。

```
aws dynamodb describe-import --import-arn arn:aws:dynamodb:us-east-1:ACCOUNT:table/
target-table/import/01658528578619-c4d4e311
```

出力例:

```
aws dynamodb describe-import --import-arn "arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658528578619-c4d4e311"
{
    "ImportTableDescription": {
        "ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/
import/01658528578619-c4d4e311",
        "ImportStatus": "FAILED",
        "TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",
        "TableId": "7b7ecc22-302f-4039-8ea9-8e7c3eb2bcb8",
        "ClientToken": "30f8891c-e478-47f4-af4a-67a5c3b595e3",
        "S3BucketSource": {
            "S3BucketOwner": "ACCOUNT",
            "S3Bucket": "my-import-source",
            "S3KeyPrefix": "import-test"
```

```
        },
        "ErrorCount": 1,
        "CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-
dynamodb/imports:*",
        "InputFormat": "CSV",
        "InputCompressionType": "NONE",
        "TableCreationParameters": {
            "TableName": "target-table",
            "AttributeDefinitions": [
                {
                    "AttributeName": "pk",
                    "AttributeType": "S"
                }
            ],
            "KeySchema": [
                {
                    "AttributeName": "pk",
                    "KeyType": "HASH"
                }
            ],
            "BillingMode": "PAY_PER_REQUEST"
        },
        "StartTime": 1658528578.619,
        "EndTime": 1658528750.628,
        "ProcessedSizeBytes": 70,
        "ProcessedItemCount": 1,
        "ImportedItemCount": 0,
        "FailureCode": "ItemValidationError",
        "FailureMessage": "Some of the items failed validation checks and were not
 imported. Please check CloudWatch error logs for more details."
    }
}
```

上記のレスポンスからロググループとインポート ID を取得し、それを使用してエラーログを取得します。インポート ID は、ImportArn フィールドの最後のパス要素です。ロググループ名は /aws-dynamodb/imports です。エラーログストリーム名は import-id/error です。この例では、01658528578619-c4d4e311/error のようになります。

項目にキー pk がない

ソース S3 オブジェクトにパラメータとして指定されたプライマリキーが含まれていない場合、インポートは失敗します。例えば、インポートのプライマリキーを列名「pk」として定義する場合などです。

```
aws dynamodb import-table –s3-bucket-source S3Bucket=my-import-
source,S3KeyPrefix=import-test.csv \
            –input-format CSV --table-creation-parameters '{"TableName":"target-
table","KeySchema":  \
            [{"AttributeName":"pk","KeyType":"HASH"}],"AttributeDefinitions":
[{"AttributeName":"pk","AttributeType":"S"}],"BillingMode":"PAY_PER_REQUEST"}'
```

列「pk」が、次のコンテンツを含むソースオブジェクト import-test.csv にありません。

```
title,artist,year_of_release
The Dark Side of the Moon,Pink Floyd,1973
```

データソースにプライマリキーがないため、項目の検証エラーによりこのインポートは失敗します。

CloudWatch エラーログの例:

```
aws logs get-log-events –log-group-name /aws-dynamodb/imports –log-stream-name
 01658528578619-c4d4e311/error
{
    "events": [
        {
            "timestamp": 1658528745319,
            "message": "{\"itemS3Pointer\":{\"bucket\":\"my-import-source\",
\"key\":\"import-test.csv\",\"itemIndex\":0},\"importArn\":\"arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658528578619-c4d4e311\",\"errorMessages
\":[\"One or more parameter values were invalid: Missing the key pk in the item\"]}",
            "ingestionTime": 1658528745414
        }
    ],
    "nextForwardToken": "f/36986426953797707963335499204463414460239026137054642176/s",
    "nextBackwardToken": "b/36986426953797707963335499204463414460239026137054642176/s"
}
```

このエラーログには、「One or more parameter values were invalid: Missing the key pk in the item」(1 つ以上のパラメータ値が無効でした: 項目にキー pk がありません) と表示されます。このインポートジョブが失敗したため、「target-table」テーブルが存在し、項目がインポートされなかったことから空になっています。最初の項目が処理され、オブジェクトは項目の検証に失敗しました。

問題を解決するには、「target-table」が不要になった場合は削除します。次に、ソースオブジェクトに存在するプライマリキー列名を使用するか、ソースデータを次のように更新します。

```
pk,title,artist,year_of_release
Albums::Rock::Classic::1973::AlbumId::ALB25,The Dark Side of the Moon,Pink Floyd,1973
```

ターゲットテーブルが存在する

インポートジョブを開始して、次のようなレスポンスを受け取った場合:

```
An error occurred (ResourceInUseException) when calling the ImportTable operation:
 Table already exists: target-table
```

このエラーを修正するには、まだ存在しないテーブル名を選択し、インポートを再試行する必要があります。

指定されたバケットは存在しません

ソースバケットが存在しない場合、インポートは失敗し、エラーメッセージの詳細が CloudWatch に記録されます。

インポートの説明の例:

```
aws dynamodb –endpoint-url $ENDPOINT describe-import –import-arn "arn:aws:dynamodb:us-
east-1:531234567890:table/target-table/import/01658530687105-e6035287"
{
    "ImportTableDescription": {
        "ImportArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table/
import/01658530687105-e6035287",
        "ImportStatus": "FAILED",
        "TableArn": "arn:aws:dynamodb:us-east-1:ACCOUNT:table/target-table",
        "TableId": "e1215a82-b8d1-45a8-b2e2-14b9dd8eb99c",
        "ClientToken": "3048e16a-069b-47a6-9dfb-9c259fd2fb6f",
        "S3BucketSource": {
            "S3BucketOwner": "531234567890",
            "S3Bucket": "BUCKET_DOES_NOT_EXIST",
            "S3KeyPrefix": "import-test"
        },
        "ErrorCount": 0,
        "CloudWatchLogGroupArn": "arn:aws:logs:us-east-1:ACCOUNT:log-group:/aws-
dynamodb/imports:*",
        "InputFormat": "CSV",
        "InputCompressionType": "NONE",
        "TableCreationParameters": {
```

```
            "TableName": "target-table",
            "AttributeDefinitions": [
                {
                    "AttributeName": "pk",
                    "AttributeType": "S"
                }
            ],
            "KeySchema": [
                {
                    "AttributeName": "pk",
                    "KeyType": "HASH"
                }
            ],
            "BillingMode": "PAY_PER_REQUEST"
        },
        "StartTime": 1658530687.105,
        "EndTime": 1658530701.873,
        "ProcessedSizeBytes": 0,
        "ProcessedItemCount": 0,
        "ImportedItemCount": 0,
        "FailureCode": "S3NoSuchBucket",
        "FailureMessage": "The specified bucket does not exist (Service: Amazon
 S3; Status Code: 404; Error Code: NoSuchBucket; Request ID: Q4W6QYYFDWY6WAKH;
 S3 Extended Request ID: ObqSlLeIMJpQqHLRX2C5Sy7n+8g6iGPwy7ixg7eEeTuEkg/+chU/JF
+RbliWytMlkUlUcuCLTrI=; Proxy: null)"
    }
}
```


FailureCode は S3NoSuchBucket となり、FailureMessage には、リクエスト ID や、エラーをスローしたサービスなどの詳細が含まれます。データがテーブルにインポートされる前にエラーがキャッチされたため、新しい DynamoDB テーブルは作成されません。場合によっては、データインポートの開始後にこれらのエラーが発生すると、部分的にインポートされたデータを含むテーブルが保持されることがあります。

このエラーを修正するには、ソース Amazon S3 バケットが存在することを確認してから、インポートプロセスを再開します。

Amazon S3 から DynamoDB にインポートするためのベストプラクティス

以下は、Amazon S3 から DynamoDB にデータをインポートするためのベストプラクティスです。

S3 オブジェクトの数を 50,000 個までに制限する

インポートジョブごとに最大 50,000 個の S3 オブジェクトがサポートされています。データセットに 50,000 個を超えるオブジェクトが含まれている場合は、それらをより大きなオブジェクトに統合することを検討してください。

過度に大きな S3 オブジェクトは避ける

S3 オブジェクトは並列してインポートされます。中規模の S3 オブジェクトが多数ある場合は、過剰なオーバーヘッドを発生せずに並列実行が可能です。1 KB 未満の項目については、各 S3 オブジェクトに 4,000,000 個の項目を配置することを検討してください。項目の平均サイズが大きい場合は、それに比例して各 S3 オブジェクトに配置する項目数を減らします。

ソート順のデータをランダム化する

S3 オブジェクトがデータをソート順に保持している場合、ローリングホットパーティションを作成する場合があります。これは、あるパーティションがすべてのアクティビティを受け取り、その後に次のパーティションがアクティビティを受け取る、というような状況です。ソート順のデータは、インポート中に同じターゲットパーティションに書き込まれる S3 オブジェクト内の連続した項目として定義されます。データがソート順序になる一般的な状況の 1 つは、項目がパーティションキーでソートされ、繰り返される項目が同じパーティションキーを共有する CSV ファイルです。

ホットパーティションのローリングを回避するため、こうしたケースでは順序をランダム化することをお勧めします。これにより、書き込みオペレーションを分散してパフォーマンスを向上させることができます。詳細については、「DynamoDB でデータアップロード中の書き込みアクティビティを効率的に分散させる」を参照してください。

データを圧縮して、S3 オブジェクトの合計サイズをリージョン内の制限以下に保つ

S3 からのインポートプロセスでは、インポートする S3 オブジェクトデータの合計サイズに制限があります。us-east-1、us-west-2、eu-west-1 リージョンでは 15 TB、その他のすべてのリージョンでは 1 TB までです。制限は raw S3 オブジェクトサイズに基づいています。

圧縮すると、より多くの raw データを制限内に収めることができます。圧縮だけではインポートが制限内に収まらない場合は、AWS Premium Support に連絡してクォータを増やすこともできます。

項目のサイズがパフォーマンスにどのように影響するか注意する

項目の平均サイズが非常に小さい (200 バイト未満) 場合、インポート処理は項目サイズが大きい場合よりも多少時間を要する場合があります。

アクティブなインポート中に S3 オブジェクトを変更しない

インポートオペレーションの進行中は、ソース S3 オブジェクトが変更されていないことを確認します。インポート中に S3 オブジェクトが変更された場合、オペレーションはエラーコード `ObjectModifiedInS3DuringImport` で失敗し、「S3 オブジェクトは上書きされたためインポートできませんでした」というメッセージが表示されます。

このエラーが発生した場合は、安定したバージョンの S3 オブジェクトを使用してインポートオペレーションを再開します。この問題を回避するには、ソースファイルに変更を加える前に、現在のインポートが完了するまで待ちます。

グローバルセカンダリインデックスなしでのインポートを検討する

インポート作業の所要時間は、1 つまたは複数のグローバルセカンダリインデックス (GSI) の有無によって異なる場合があります。カーディナリティの低いパーティションキーを使用してインデックスを作成する予定の場合、インポート作業が終了するまでインデックスの作成を延期すると (それらをインポートジョブに含めるよりも) インポート速度が向上する可能性があります。

> **Note**
>
> GSI の作成には、インポート中またはインポート後に作成された場合でも、書き込み料金は発生しません。

Amazon S3 への DynamoDB データのエクスポート: 仕組み

DynamoDB の S3 へのエクスポートは、DynamoDB データを Amazon S3 バケットに大規模にエクスポートするためのフルマネージドソリューションです。S3 への DynamoDB エクスポートを使用すると、Amazon DynamoDB テーブルのデータをポイントインタイムリカバリ（PITR) ウィンドウ内の任意の時点から Amazon S3 バケットにエクスポートできます。エクスポート機能を使用するには、テーブルで PITR を有効にする必要があります。この機能を使用すると、Athena、AWS Glue、Amazon SageMaker AI、Amazon EMR や AWS Lake Formation などの他の AWS のサービスを使用して、データの分析や複雑なクエリを有効にできます。

S3 への DynamoDB エクスポートでは、DynamoDB テーブルからフルデータと増分データの両方をエクスポートできます。フルエクスポートと増分エクスポートのいずれにおいても、フィルタリングされたエクスポートを実行して、テーブル全体ではなくテーブルデータのサブセットのみを取得することもできます。詳細については、 を参照してください。テーブルのエクスポートのフィルタリングエクスポートは非同期であり、読み取りキャパシティユニット (RCU) を消費せず、テーブルのパフォーマンスや可用性には影響しません。サポートされているエクスポートファイル形式

は、DynamoDB JSON 形式と Amazon Ion 形式です。また、データを別の AWS アカウントが所有する S3 バケットや、異なる AWS リージョンにエクスポートすることもできます。データは常にエンドツーエンドで暗号化されます。

> **Note**
>
> サービスレベルアグリーメント (SLA) では、エクスポートの完了時間は保証されず、これらの時間は異なる場合があります。エクスポートの所要時間は、エクスポートするテーブルまたはデータのサイズ、テーブル全体でのデータの分散状況、進行中のその他のエクスポートなどの要因によって異なります。このため、エクスポートが一定の時間内に完了することに依存するワークフローを構築しないでください。

DynamoDB のフルエクスポートは、エクスポートが行われた時点の DynamoDB テーブル (テーブルデータとローカルセカンダリインデックス) のサイズ (テーブルデータとローカルセカンダリインデックス) に基づいて課金されます。DynamoDB の増分エクスポートは、エクスポートされている期間に継続的バックアップから処理されたデータのサイズに基づいて課金されます。増分エクスポートは、最低料金が 10 MB を対象とします。フィルタリングされたエクスポートも同じレートで課金され、フィルタリングに追加料金はかかりません。フィルタリングされたエクスポートで、キー条件を使用して 1 つのパーティションキー値を指定すると、エクスポートが読み取るデータ量が少なくなり、一致するデータに対してのみ課金されます。ただし、エクスポートごとに 10 MB の最小値が適用されます。エクスポートされたデータを Amazon S3 に保存する場合と、Amazon S3 バケットに対して行われた PUT リクエストには追加料金が適用されます。これらの料金の詳細については、「Amazon DynamoDB 料金表」と「Amazon S3 料金表」を参照してください。

Service Quotas の詳細については、「Amazon S3 へのテーブルのエクスポート」を参照してください。

トピック

- DynamoDB でテーブルのエクスポートをリクエストする
- テーブルのエクスポートのフィルタリング
- DynamoDB テーブルのエクスポート出力形式

DynamoDB でテーブルのエクスポートをリクエストする

DynamoDB テーブルのエクスポートを使用すると、テーブルデータを Amazon S3 バケットにエクスポートできます。これにより、Athena、AWS Glue、Amazon SageMaker AI、Amazon EMR、お

よび AWS Lake Formation などの他の AWS のサービスを使用して、データに対して分析や複雑なクエリの実行が許可されます。テーブルのエクスポートをリクエストするには、AWS マネジメントコンソール、AWS CLI、または DynamoDB API のいずれかを使用します。

> **Note**
>
> Amazon S3 のリクエスタ支払いバケットはサポートされていません。

DynamoDB はフルエクスポートと増分エクスポートの両方をサポートしています。

- フルエクスポートを使用すると、テーブルの完全なスナップショットをポイントインタイムリカバリ (PITR) ウィンドウ内の任意の時点から Amazon S3 バケットにエクスポートできます。
- 増分エクスポートでは、指定した期間に変更、更新、または削除された DynamoDB テーブルのデータを、PITR ウィンドウ内で Amazon S3 バケットにエクスポートできます。
- フィルタリングされたエクスポートでは、DynamoDB テーブルのデータ全体ではなくサブセットをエクスポートできます。フィルターは、フルエクスポートと増分エクスポートのいずれにも適用できます。フィルタリングされたエクスポートの仕組みの詳細については、「テーブルのエクスポートのフィルタリング」を参照してください。

トピック

- 前提条件
- AWS マネジメントコンソール を使用してエクスポートをリクエストする
- AWS マネジメントコンソール で過去のエクスポートの詳細を取得する
- AWS CLI および AWS SDK を使用してエクスポートをリクエストする
- AWS CLI および AWS SDK を使用して過去のエクスポートの詳細を取得する

前提条件

PITR を有効にする

S3 へのエクスポート機能を使用するには、テーブルで PITR を有効にする必要があります。PITR を有効にする方法の詳細については、「ポイントインタイムリカバリ」を参照してください。PITR が有効になっていないテーブルのエクスポートをリクエストすると、「`ExportTableToPointInTime` オペレーションの呼び出し時にエラーが発生しました (PointInTimeRecoveryUnavailableException): テーブル 'my-dynamodb-table' のポイントインタイム

リカバリが有効になっていません」という例外メッセージが表示されてリクエストが失敗します。リクエストとエクスポートは、設定された PITR RecoveryPeriodInDays 内の特定の時点からのみ行うことができます。

S3 アクセス権限の設定

テーブルデータは、書き込みアクセス許可のある任意の Amazon S3 バケットにエクスポートできます。エクスポート先のバケットは、ソーステーブル所有者と同じ AWS リージョンに存在する必要はなく、所有者が同じである必要もありません。AWS Identity and Access Management (IAM) ポリシーでは、S3 アクション (s3:AbortMultipartUpload、s3:PutObject、および s3:PutObjectAcl) と DynamoDB エクスポートアクション (dynamodb:ExportTableToPointInTime) を実行できるようにする必要があります。S3 バケットへのエクスポートを実行するアクセス許可をユーザーに付与するポリシーの例を次に示します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowDynamoDBExportAction",
            "Effect": "Allow",
            "Action": "dynamodb:ExportTableToPointInTime",
            "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/my-table"
        },
        {
            "Sid": "AllowS3BucketWrites",
            "Effect": "Allow",
            "Action": [
                "s3:AbortMultipartUpload",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
        }
    ]
}
```

別のアカウントにある Amazon S3 バケットに書き込む必要がある場合や書き込みアクセス許可がない場合、Amazon S3 バケット所有者は、DynamoDB からバケットへのエクスポートをユーザーに許可するバケットポリシーを追加する必要があります。エクスポート先の Amazon S3 バケットのポリシー例を次に示します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleStatement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/Dave"
            },
            "Action": [
                "s3:AbortMultipartUpload",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
        }
    ]
}
```

エクスポートの進行中にこれらの権限を取り消すと、ファイルの一部だけがエクスポートされます。

> **Note**
>
> ソーステーブルと同じ AWS アカウントのプリンシパルとして、`ExportTableToPointInTime` を呼び出す必要があります。(上記のように) 送信先バケットは別のアカウントに属していてもかまいませんが、エクスポートリクエスト自体はテーブルのアカウントから送信する必要があります。別のアカウントから行われたリクエストは、「このアクションはリソース所有者のアカウントに一致するアカウントでのみサポートされています」という `ValidationException` で失敗します。別のアカウントが所有するテーブルをエクスポートするには、そのテーブルのアカウントで IAM ロールを引き受け、そのロールの認証情報を使用してエクスポートリクエストを行います。そのロールが別の

アカウントのバケットに書き込む場合、バケット所有者を含め、上記のようにバケットポリシーでロールに書き込みアクセスを許可します。

> **Note**
>
> エクスポート先のテーブルまたはバケットがカスタマーマネージドキーで暗号化されている場合、その KMS キーのポリシーは、これを使用する許可を DynamoDB に与える必要があります。この権限は、エクスポートジョブをトリガーする IAM ユーザー/ロールを通じて付与されます。ベストプラクティスを含む暗号化の詳細については、「Amazon DynamoDB が AWS KMS を使用する方法」および「カスタム KMS キーの使用」を参照してください。

AWS マネジメントコンソール を使用してエクスポートをリクエストする

DynamoDB コンソールを使用して、`MusicCollection` という既存のテーブルをエクスポートする方法を次の例に示します。

> **Note**
>
> この手順は、ポイントインタイムリカバリを有効にしていることを前提としています。これを `MusicCollection` テーブルに対して有効にするには、[Overview] (概要) タブにある [Table details] (テーブルの詳細) セクションの [Point-in-time recovery] (ポイントインタイムリカバリ) で [Enable] (有効化) を選択します。

テーブルのエクスポートをリクエストするには

1. AWS マネジメントコンソール にサインインして DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/) を開きます。
2. コンソールの左側のナビゲーションペインで、[S3 へのエクスポート] を選択します。
3. [S3 にエクスポート] ボタンを選択します。
4. コピー元テーブルとエクスポート先の S3 バケットを選択します。エクスポート先のバケットがお客様のアカウントによって所有されている場合は、[Browse S3] (S3 を参照) ボタンをクリックしてバケットを検索します。そうでない場合は、`s3://bucketname/prefix` format. の形式でバケットの URL を入力します。ここで、**prefix** は、エクスポート先のバケットを整理するのに役立つオプションのフォルダです。

5. [フルエクスポート] または [増分エクスポート] を選択します。フルエクスポートでは、指定した時点のテーブルのフルテーブルスナップショットが出力されます。増分エクスポートでは、指定したエクスポート期間中にテーブルに加えられた変更が出力されます。出力は、エクスポート期間における項目の最終状態のみを含むように圧縮されます。同じエクスポート期間内に複数の更新があったとしても、項目はエクスポートに 1 回しか表示されません。

   Full export

   1. テーブルスナップショット全体をエクスポートする時点を選択します。これは PITR ウィンドウ内のどの時点でもかまいません。または、[現在の時刻] を選択して最新のスナップショットをエクスポートすることもできます。
   2. [エクスポートされたファイル形式] で、[DynamoDB JSON] と [Amazon Ion] のどちらかを選択します。デフォルトで、テーブルは、ポイントインタイムリカバリウィンドウの復元可能な最新の時点から DynamoDB JSON 形式でエクスポートされ、Amazon S3 キー (SSE S3) を使用して暗号化されます。これらのエクスポート設定は、必要に応じて変更できます。

      > **Note**
      >
      > AWS Key Management Service (AWS KMS) によって保護されたキーを使用してエクスポートを暗号化することを選択した場合、そのキーはエクスポート先 S3 バケットと同じリージョンにある必要があります。

   Incremental export

   1. 増分データをエクスポートする [エクスポート期間] を選択します。PITR ウィンドウ内で開始時刻を選択します。エクスポート期間は 15 分～24 時間に設定する必要があります。エクスポート期間の開始時刻は期間に含まれ、終了時刻は期間に含まれません。
   2. [絶対モード] または [相対モード] を選択します。
      a. [絶対モード] では、指定した期間の増分データがエクスポートされます。
      b. [相対モード] では、エクスポートジョブの送信時間を基準にしたエクスポート期間の差分データがエクスポートされます。
   3. [エクスポートされたファイル形式] で、[DynamoDB JSON] と [Amazon Ion] のどちらかを選択します。デフォルトで、テーブルは、ポイントインタイムリカバリウィンドウの復元可能な最新の時点から DynamoDB JSON 形式でエクスポートされ、Amazon S3

キー (SSE S3) を使用して暗号化されます。これらのエクスポート設定は、必要に応じて変更できます。

> **Note**
>
> AWS Key Management Service (AWS KMS) によって保護されたキーを使用してエクスポートを暗号化することを選択した場合、そのキーはエクスポート先 S3 バケットと同じリージョンにある必要があります。

4. [エクスポートビュータイプ] では、[新旧イメージ] または [新しいイメージのみ] を選択します。新しいイメージには、項目の最新の状態が表示されます。古いイメージには、指定した「開始日時」の直前の項目の状態が表示されます。デフォルト設定は [新旧イメージ] です。新旧イメージの詳細については、「増分エクスポート出力」を参照してください。

6. (オプション) テーブルデータのサブセットのみをエクスポートするには、エクスポートのフィルターを指定します。任意の属性のフィルター式、パーティションキー値で項目を選択するキー条件式、出力に含める属性を選択するプロジェクション式を指定できます。これらの式の仕組みや、処理されるデータにどのような影響を与えるかについての詳細は、「テーブルのエクスポートのフィルタリング」を参照してください。

7. [エクスポート] を選択して開始します。

エクスポートされたデータは、トランザクションとして一貫しません。トランザクションオペレーションは、2 つのエクスポート出力間で分断される可能性があります。トランザクションオペレーションで変更された項目のサブセットはエクスポートに反映される一方で、同じトランザクションで変更された別のサブセットは同じエクスポートリクエストに反映されない場合があります。ただし、最終的にエクスポートの一貫性は保たれます。エクスポート中にトランザクションが分断された場合、残りのトランザクションは直後のエクスポートに (重複することなく) 含まれます。エクスポートに使用される時間は内部システムクロックに基づいており、変動する可能性はアプリケーションのローカルクロックの 1 分のみです。

AWS マネジメントコンソール で過去のエクスポートの詳細を取得する

過去に実行したエクスポートタスクに関する情報は、ナビゲーションサイドバーで [S3 へのエクスポート] セクションを選択して確認できます。このセクションには、過去 90 日間に作成したエクスポートの一覧が表示されます。[エクスポート] タブにリストされているタスクの ARN を選択すると、エクスポートに関する情報 (選択した詳細設定など) を取得できます。エクスポートタスクのメタデータは 90 日後に期限切れになり、それより古いジョブはこのリストに表示されなくなります

が、S3 バケット内のオブジェクトは、バケットポリシーで許可されている限り維持されます。エクスポート中に S3 バケットに作成されたオブジェクトが DynamoDB によって削除されることはありません。

AWS CLI および AWS SDK を使用してエクスポートをリクエストする

以下の例は、既存のテーブルを S3 バケットにエクスポートする方法を示しています。

> **Note**
>
> この手順は、ポイントインタイムリカバリを有効にしていることを前提としています。この機能を `MusicCollection` テーブルに対して有効にするには、次のコマンドを実行します。
>
> ```
> aws dynamodb update-continuous-backups \
>     --table-name MusicCollection \
>     --point-in-time-recovery-specification PointInTimeRecoveryEnabled=True
> ```

フルエクスポート

AWS CLI

> **Note**
>
> クロスアカウントテーブルのエクスポートをリクエストする場合は、必ず `--s3-bucket-owner` オプションを含めます。

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-west-2:111122223333:table/MusicCollection \
  --s3-bucket ddb-export-musiccollection-9012345678 \
  --s3-prefix 2020-Nov \
  --export-format DYNAMODB_JSON \
  --export-time 1604632434 \
  --s3-bucket-owner 9012345678 \
  --s3-sse-algorithm AES256
```

Python

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')

client.export_table_to_point_in_time(
    TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
    ExportTime=datetime(2023, 9, 20, 12, 0, 0),
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON'
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

client.exportTableToPointInTime(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE")
    .exportTime(Instant.parse("2023-09-20T12:00:00Z"))
    .s3Bucket("bucket")
    .s3Prefix("prefix")
    .s3SseAlgorithm(S3SseAlgorithm.AES256)
    .exportFormat(ExportFormat.DYNAMODB_JSON));
```

.NET

```
var client = new AmazonDynamoDBClient();

await client.ExportTableToPointInTimeAsync(new ExportTableToPointInTimeRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportTime = new DateTime(2023, 9, 20, 12, 0, 0, DateTimeKind.Utc),
    S3Bucket = "bucket",
    S3Prefix = "prefix",
    S3SseAlgorithm = S3SseAlgorithm.AES256,
    ExportFormat = ExportFormat.DYNAMODB_JSON
});
```

JavaScript

```
import { DynamoDBClient, ExportTableToPointInTimeCommand } from "@aws-sdk/client-
dynamodb";

const client = new DynamoDBClient();

await client.send(new ExportTableToPointInTimeCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportTime: new Date("2023-09-20T12:00:00Z"),
    S3Bucket: "bucket",
    S3Prefix: "prefix",
    S3SseAlgorithm: "AES256",
    ExportFormat: "DYNAMODB_JSON"
}));
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

exportTime := time.Date(2023, 9, 20, 12, 0, 0, 0, time.UTC)
client.ExportTableToPointInTime(context.TODO(),
 &dynamodb.ExportTableToPointInTimeInput{
    TableArn:       aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/
TABLE"),
    ExportTime:     &exportTime,
    S3Bucket:       aws.String("bucket"),
    S3Prefix:       aws.String("prefix"),
    S3SseAlgorithm: types.S3SseAlgorithmAes256,
    ExportFormat:   types.ExportFormatDynamodbJson,
})
```

増分エクスポート

AWS CLI

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:REGION:ACCOUNT:table/TABLENAME \
  --s3-bucket BUCKET --s3-prefix PREFIX \
```

```
  --incremental-export-specification
 ExportFromTime=1693569600,ExportToTime=1693656000,ExportViewType=NEW_AND_OLD_IMAGES
 \
  --export-type INCREMENTAL_EXPORT
```

Python

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')

client.export_table_to_point_in_time(
    TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
    IncrementalExportSpecification={
      'ExportFromTime': datetime(2023, 9, 20, 12, 0, 0),
      'ExportToTime': datetime(2023, 9, 20, 13, 0, 0),
      'ExportViewType': 'NEW_AND_OLD_IMAGES'
    },
    ExportType='INCREMENTAL_EXPORT',
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON'
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

client.exportTableToPointInTime(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE")
    .exportType(ExportType.INCREMENTAL_EXPORT)
    .incrementalExportSpecification(i -> i
        .exportFromTime(Instant.parse("2023-09-20T12:00:00Z"))
        .exportToTime(Instant.parse("2023-09-20T13:00:00Z"))
        .exportViewType(ExportViewType.NEW_AND_OLD_IMAGES))
    .s3Bucket("bucket")
    .s3Prefix("prefix")
    .s3SseAlgorithm(S3SseAlgorithm.AES256)
    .exportFormat(ExportFormat.DYNAMODB_JSON));
```

.NET

```
var client = new AmazonDynamoDBClient();

await client.ExportTableToPointInTimeAsync(new ExportTableToPointInTimeRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportType = ExportType.INCREMENTAL_EXPORT,
    IncrementalExportSpecification = new IncrementalExportSpecification
    {
        ExportFromTime = new DateTime(2023, 9, 20, 12, 0, 0, DateTimeKind.Utc),
        ExportToTime = new DateTime(2023, 9, 20, 13, 0, 0, DateTimeKind.Utc),
        ExportViewType = ExportViewType.NEW_AND_OLD_IMAGES
    },
    S3Bucket = "bucket",
    S3Prefix = "prefix",
    S3SseAlgorithm = S3SseAlgorithm.AES256,
    ExportFormat = ExportFormat.DYNAMODB_JSON
});
```

JavaScript

```
import { DynamoDBClient, ExportTableToPointInTimeCommand } from "@aws-sdk/client-
dynamodb";

const client = new DynamoDBClient();

await client.send(new ExportTableToPointInTimeCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportType: "INCREMENTAL_EXPORT",
    IncrementalExportSpecification: {
        ExportFromTime: new Date("2023-09-20T12:00:00Z"),
        ExportToTime: new Date("2023-09-20T13:00:00Z"),
        ExportViewType: "NEW_AND_OLD_IMAGES"
    },
    S3Bucket: "bucket",
    S3Prefix: "prefix",
    S3SseAlgorithm: "AES256",
    ExportFormat: "DYNAMODB_JSON"
}));
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

fromTime := time.Date(2023, 9, 20, 12, 0, 0, 0, time.UTC)
toTime := time.Date(2023, 9, 20, 13, 0, 0, 0, time.UTC)
client.ExportTableToPointInTime(context.TODO(),
 &dynamodb.ExportTableToPointInTimeInput{
    TableArn:   aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"),
    ExportType: types.ExportTypeIncrementalExport,
    IncrementalExportSpecification: &types.IncrementalExportSpecification{
        ExportFromTime: &fromTime,
        ExportToTime:   &toTime,
        ExportViewType: types.ExportViewTypeNewAndOldImages,
    },
    S3Bucket:       aws.String("bucket"),
    S3Prefix:       aws.String("prefix"),
    S3SseAlgorithm: types.S3SseAlgorithmAes256,
    ExportFormat:   types.ExportFormatDynamodbJson,
})
```

フィルタリングされたエクスポート

次の例では、エクスポートリクエストに FilterSpecification を含めることでテーブルのサブセットをエクスポートします。各例では、1 つのパーティションキー値を選択するためにキー条件式を使用し、非キー属性を一致させるためにフィルター式を使用し、出力する属性を選択するためにプロジェクション式を使用しています。FilterSpecification は、フルエクスポートまたは増分エクスポートのいずれにも適用できます。式のサイズは最大 4 KB、属性名は最大 250 バイト、属性値は最大 2 MB まで指定でき、式には最大 300 個の演算子を含めることができます。キー条件式を指定すると、パーティションキーをフィルター式に含めることはできません。詳細については、「テーブルのエクスポートのフィルタリング」を参照してください。

AWS CLI

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-east-1:111122223333:table/MusicCollection \
  --s3-bucket ddb-export-musiccollection-9012345678 \
  --s3-prefix 2020-Nov \
  --export-format DYNAMODB_JSON \
  --export-time 1604632434 \
```

```
  --s3-sse-algorithm AES256 \
  --filter-specification '{
    "KeyConditionExpression": "customer_id = :cid",
    "FilterExpression": "#tier = :premium",
    "ProjectionExpression": "customer_id, #tier",
    "ExpressionAttributeNames": {"#tier": "tier"},
    "ExpressionAttributeValues": {":cid": {"S": "cust-123"}, ":premium": {"S":
 "premium"}}
  }'
```

Python

```
import boto3
from datetime import datetime

client = boto3.client('dynamodb')

client.export_table_to_point_in_time(
    TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
    ExportTime=datetime(2023, 9, 20, 12, 0, 0),
    S3Bucket='bucket',
    S3Prefix='prefix',
    S3SseAlgorithm='AES256',
    ExportFormat='DYNAMODB_JSON',
    FilterSpecification={
        'KeyConditionExpression': 'customer_id = :cid',
        'FilterExpression': '#tier = :premium',
        'ProjectionExpression': 'customer_id, #tier',
        'ExpressionAttributeNames': {'#tier': 'tier'},
        'ExpressionAttributeValues': {
            ':cid': {'S': 'cust-123'},
            ':premium': {'S': 'premium'}
        }
    }
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

client.exportTableToPointInTime(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE")
    .exportTime(Instant.parse("2023-09-20T12:00:00Z"))
```

```
    .s3Bucket("bucket")
    .s3Prefix("prefix")
    .s3SseAlgorithm(S3SseAlgorithm.AES256)
    .exportFormat(ExportFormat.DYNAMODB_JSON)
    .filterSpecification(f -> f
        .keyConditionExpression("customer_id = :cid")
        .filterExpression("#tier = :premium")
        .projectionExpression("customer_id, #tier")
        .expressionAttributeNames(Map.of("#tier", "tier"))
        .expressionAttributeValues(Map.of(
            ":cid", AttributeValue.fromS("cust-123"),
            ":premium", AttributeValue.fromS("premium")))));
```

.NET

```
var client = new AmazonDynamoDBClient();

await client.ExportTableToPointInTimeAsync(new ExportTableToPointInTimeRequest
{
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportTime = new DateTime(2023, 9, 20, 12, 0, 0, DateTimeKind.Utc),
    S3Bucket = "bucket",
    S3Prefix = "prefix",
    S3SseAlgorithm = S3SseAlgorithm.AES256,
    ExportFormat = ExportFormat.DYNAMODB_JSON,
    FilterSpecification = new FilterSpecification
    {
        KeyConditionExpression = "customer_id = :cid",
        FilterExpression = "#tier = :premium",
        ProjectionExpression = "customer_id, #tier",
        ExpressionAttributeNames = new Dictionary<string, string>
        {
            { "#tier", "tier" }
        },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>
        {
            { ":cid", new AttributeValue { S = "cust-123" } },
            { ":premium", new AttributeValue { S = "premium" } }
        }
    }
});
```

JavaScript

```
import { DynamoDBClient, ExportTableToPointInTimeCommand } from "@aws-sdk/client-
dynamodb";

const client = new DynamoDBClient();

await client.send(new ExportTableToPointInTimeCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE",
    ExportTime: new Date("2023-09-20T12:00:00Z"),
    S3Bucket: "bucket",
    S3Prefix: "prefix",
    S3SseAlgorithm: "AES256",
    ExportFormat: "DYNAMODB_JSON",
    FilterSpecification: {
        KeyConditionExpression: "customer_id = :cid",
        FilterExpression: "#tier = :premium",
        ProjectionExpression: "customer_id, #tier",
        ExpressionAttributeNames: { "#tier": "tier" },
        ExpressionAttributeValues: {
            ":cid": { S: "cust-123" },
            ":premium": { S: "premium" }
        }
    }
}));
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

exportTime := time.Date(2023, 9, 20, 12, 0, 0, 0, time.UTC)
client.ExportTableToPointInTime(context.TODO(),
 &dynamodb.ExportTableToPointInTimeInput{
    TableArn:       aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/
TABLE"),
    ExportTime:     &exportTime,
    S3Bucket:       aws.String("bucket"),
    S3Prefix:       aws.String("prefix"),
    S3SseAlgorithm: types.S3SseAlgorithmAes256,
    ExportFormat:   types.ExportFormatDynamodbJson,
    FilterSpecification: &types.FilterSpecification{
        KeyConditionExpression: aws.String("customer_id = :cid"),
```

```
        FilterExpression:       aws.String("#tier = :premium"),
        ProjectionExpression:   aws.String("customer_id, #tier"),
        ExpressionAttributeNames: map[string]string{
            "#tier": "tier",
        },
        ExpressionAttributeValues: map[string]types.AttributeValue{
            ":cid":     &types.AttributeValueMemberS{Value: "cust-123"},
            ":premium": &types.AttributeValueMemberS{Value: "premium"},
        },
    },
})
```

> **Note**
>
> DynamoDB は、フィルタリングされたエクスポートを開始する前に式を検証します。式が含まれていない FilterSpecification を指定した場合、または式が無効である場合、リクエストは ValidationException となり、失敗します。DescribeExport オペレーションは、エクスポートに適用した FilterSpecification を返します。ListExports オペレーションは変更されず、フィルタリングによって新しいエクスポートタイプが導入されることもありません。エクスポートタイプ (FULL_EXPORT または INCREMENTAL_EXPORT) でも、データがエクスポートされるタイミングが決まります。どのデータがエクスポートされるかは、フィルターによって決まります。

> **Note**
>
> AWS Key Management Service (AWS KMS) によって保護されたキーを使用してエクスポートを暗号化することを選択した場合、そのキーはエクスポート先 S3 バケットと同じリージョンにある必要があります。

AWS CLI および AWS SDK を使用して過去のエクスポートの詳細を取得する

過去に実行したエクスポートリクエストに関する情報は、list-exports コマンドを実行して見つけることができます。このコマンドは、過去 90 日間に作成されたすべてのエクスポートのリストを返します。エクスポートタスクのメタデータは 90 日後に期限切れになり、それより古いジョブは list-exports コマンドによって返されなくなりますが、S3 バケット内のオブジェクトは、バケッ

トポリシーで許可されている限り維持されます。エクスポート中に S3 バケットに作成されたオブジェクトが DynamoDB によって削除されることはありません。

エクスポートのステータスは、成功または失敗するまで PENDING となります。成功すると、ステータスは COMPLETED に変わります。失敗すると、ステータスは FAILED に変わり、failure_message と failure_reason が追加されます。

エクスポートを一覧表示する

AWS CLI

```
aws dynamodb list-exports \
    --table-arn arn:aws:dynamodb:us-east-1:111122223333:table/ProductCatalog
```

Python

```
import boto3

client = boto3.client('dynamodb')

print(
  client.list_exports(
     TableArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE',
  )
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

ListExportsResponse response = client.listExports(b -> b
    .tableArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"));

response.exportSummaries().forEach(System.out::println);
```

.NET

```
var client = new AmazonDynamoDBClient();

var response = await client.ListExportsAsync(new ListExportsRequest
{
```

```
    TableArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"
});

response.ExportSummaries.ForEach(Console.WriteLine);
```

JavaScript

```
import { DynamoDBClient, ListExportsCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient();

const response = await client.send(new ListExportsCommand({
    TableArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"
}));

console.log(response.ExportSummaries);
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

response, _ := client.ListExports(context.TODO(), &dynamodb.ListExportsInput{
    TableArn: aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE"),
})

fmt.Println(response.ExportSummaries)
```

エクスポートを記述する

AWS CLI

```
aws dynamodb describe-export \
    --export-arn arn:aws:dynamodb:us-east-1:111122223333:table/ProductCatalog/
export/01695353076000-a1b2c3d4
```

Python

```
import boto3
```

```
client = boto3.client('dynamodb')

print(
  client.describe_export(
     ExportArn='arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f',
  )['ExportDescription']
)
```

Java

```
DynamoDbClient client = DynamoDbClient.create();

DescribeExportResponse response = client.describeExport(b -> b
    .exportArn("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"));

System.out.println(response.exportDescription());
```

.NET

```
var client = new AmazonDynamoDBClient();

var response = await client.DescribeExportAsync(new DescribeExportRequest
{
    ExportArn = "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"
});

Console.WriteLine(response.ExportDescription);
```

JavaScript

```
import { DynamoDBClient, DescribeExportCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient();

const response = await client.send(new DescribeExportCommand({
    ExportArn: "arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"
}));
```

```
console.log(response.ExportDescription);
```

Go

```
cfg, _ := config.LoadDefaultConfig(context.TODO())
client := dynamodb.NewFromConfig(cfg)

response, _ := client.DescribeExport(context.TODO(), &dynamodb.DescribeExportInput{
    ExportArn: aws.String("arn:aws:dynamodb:us-east-1:111122223333:table/TABLE/
export/01695353076000-06e2188f"),
})

fmt.Println(response.ExportDescription)
```

テーブルのエクスポートのフィルタリング

テーブル全体をエクスポートする代わりに、必要な項目と属性のみをエクスポートできます。DynamoDB は、指定した式に一致するデータのみをエクスポートします。フィルタリングされたエクスポートは、フルエクスポートと増分エクスポートの両方で機能します。

増分エクスポートの場合、DynamoDB は変更された各項目の最新のイメージに対してフィルターを評価します。削除された項目の場合、新しいイメージが存在しないため、DynamoDB は項目の古いイメージに対してフィルターを評価します。

フィルタリングされたエクスポートで何が返されるかは、3 種類の式を使用して制御できます。1 つのエクスポートリクエストにおいて、これらの式を単独で指定することも組み合わせて指定することもできます。

- キー条件式は、DynamoDB の `Query` におけるキー条件式と同じ構文とルールを使用して、プライマリキーで項目を選択します。キー条件式では、等価比較によって単一のパーティションキー値を指定し、オプションでソートキーの条件を指定します。非キー属性を参照することはできず、複数のパーティションキー値を指定することもできません。キー条件式はエクスポートを単一のパーティションキー値に限定するため、DynamoDB がエクスポート時に読み込むデータの量が削減されます。詳細については、「DynamoDB でのクエリオペレーションのキー条件式」を参照してください。
- フィルター式は、`Scan` オペレーションにおけるフィルター式と同じ構文とルールを使用して、キー属性と非キー属性の両方に条件を適用できます。フィルター式は、等しい、等しくない、`OR` などの比較に加え、`contains`、`IN`、`begins_with`、`BETWEEN`、`attribute_exists`、`size` といった関数をサポートしています。DynamoDB はデータの読み取り後にフィルター式を適用す

るため、フィルターを使用しても、エクスポートで DynamoDB が処理するデータ量は減りません。フィルター式とその構文の詳細については、「フィルター式と条件式の構文」を参照してください。

- プロジェクション式は、エクスポート出力に含める特定の属性を指定し、他の DynamoDB オペレーションのプロジェクション式と同じ構文とルールを使用します。詳細については、「DynamoDB でのプロジェクション式の使用」を参照してください。

> **Note**
>
> キー属性をフィルター式に含めることができるかどうかは、キー条件式も指定しているかどうかで決まります。キー条件式を指定すると、キー属性をフィルター式に含めることはできません。フィルター式のみを指定すると、キー属性をフィルター式に含めることができます。

各式には、次の制限が適用されます。

- 式のサイズは最大 4 KB です。
- 属性名は最大 250 バイトです。
- 属性値は最大 2 MB です。
- 式には最大 300 個の演算子を含めることができます。

DynamoDB テーブルのエクスポート出力形式

DynamoDB テーブルのエクスポートには、テーブルデータを含むファイルに加えて、マニフェストファイルが含まれます。これらのファイルはすべて、エクスポート要求で指定した Amazon S3 バケットに保存されます。以下のセクションでは、各出力オブジェクトの形式と内容について説明します。

フルエクスポート出力

マニフェストファイル

DynamoDB では、エクスポートリクエストごとにマニフェストファイルがチェックサムファイルとともに指定した S3 バケットに作成されます。

```
export-prefix/AWSDynamoDB/ExportId/manifest-summary.json
```

```
export-prefix/AWSDynamoDB/ExportId/manifest-summary.checksum
export-prefix/AWSDynamoDB/ExportId/manifest-files.json
export-prefix/AWSDynamoDB/ExportId/manifest-files.checksum
```

テーブルのエクスポートをリクエストする際は、**export-prefix** を選択します。これにより、エクスポート先 S3 バケットのファイルを整理できます。**ExportId** は、同じ S3 バケットおよび export-prefix への複数のエクスポートが相互に上書きされないようにするために、サービスによって生成される一意のトークンです。

エクスポートでは、パーティションごとに少なくとも 1 つのファイルが作成されます。空のパーティションの場合、エクスポートリクエストにより空のファイルが作成されます。各ファイル内のすべての項目は、その特定のパーティションのハッシュされたキースペースからのものです。

> **Note**
>
> また、DynamoDB では、マニフェストファイルと同じディレクトリに _started という名前の空のファイルが作成されます。このファイルは、エクスポート先のバケットが書き込み可能であること、およびエクスポートが開始されたことを検証します。これは安全に削除できます。

要約マニフェスト

manifest-summary.json ファイルには、エクスポートジョブに関する概要情報が含まれています。これにより、共有データフォルダー内のどのデータファイルがこのエクスポートに関連付けられているかがわかります。この形式は次のとおりです。

```
{
   "version": "2020-06-30",
   "exportArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog/
export/01234567890123-a1b2c3d4",
   "startTime": "2020-11-04T07:28:34.028Z",
   "endTime": "2020-11-04T07:33:43.897Z",
   "tableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/ProductCatalog",
   "tableId": "12345a12-abcd-123a-ab12-1234abc12345",
   "exportTime": "2020-11-04T07:28:34.028Z",
   "s3Bucket": "ddb-productcatalog-export",
   "s3Prefix": "2020-Nov",
   "s3SseAlgorithm": "AES256",
   "s3SseKmsKeyId": null,
```

```
   "manifestFilesS3Key": "AWSDynamoDB/01693685827463-2d8752fd/manifest-files.json",
   "billedSizeBytes": 0,
   "itemCount": 8,
   "outputFormat": "DYNAMODB_JSON"
}
```

> **Note**
>
> exportType フィールドは、増分エクスポートを実行するときのみ manifest-summary.json ファイルに存在し、INCREMENTAL_EXPORT という値を持ちます。フルエクスポートを実行するときに概要マニフェストには含まれません。

> **Note**
>
> エクスポートリクエストにフィルターが含まれている場合、概要マニフェストには適用されたフィルター仕様も含まれます。

ファイルマニフェスト

manifest-files.json ファイルには、エクスポートされたテーブルデータを含むファイルに関する情報が含まれています。ファイルは JSON Lines 形式であるため、改行が項目区切り文字として使用されます。次の例では、読みやすくするためにファイルマニフェストの 1 つのデータファイルの詳細が複数行にフォーマットされています。

```
{
"itemCount": 8,
   "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",
   "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",
   "dataFileS3Key": "AWSDynamoDB/01693685827463-2d8752fd/data/asdl123dasas.json.gz"
}
```

データファイル

DynamoDB では、DynamoDB JSON と Amazon Ion の 2 つの形式でテーブルデータをエクスポートできます。選択した形式にかかわらず、データは、 キーによって名前が付けられた複数の圧縮ファイルに書き込まれます。これらのファイルも manifest-files.json ファイル内に一覧表示されます。

フルエクスポート後の Amazon S3 バケットのディレクトリ構造では、export Id フォルダーにマニフェストファイルとデータファイルがすべて含まれます。

```
amzn-s3-demo-bucket/DestinationPrefix
.
### AWSDynamoDB
    ### 01693685827463-2d8752fd     // the single full export
    #   ### manifest-files.json     // manifest points to files under 'data' subfolder
    #   ### manifest-files.checksum
    #   ### manifest-summary.json   // stores metadata about request
    #   ### manifest-summary.md5
    #   ### data                    // The data exported by full export
    #   #   ### asdl123dasas.json.gz
    #   #   ...
    #   ### _started                // empty file for permission check
```

DynamoDB JSON

DynamoDB JSON 形式のテーブルエクスポートは、複数の `Item` オブジェクトで構成されます。個々のオブジェクトは DynamoDB のスタンダードマーシャリングされた JSON 形式です。

DynamoDB JSON エクスポートデータ用のカスタムパーサーを作成する場合、形式は JSON Lines です。これは、改行が項目の区切り文字として使用されていることを意味します。Athena や AWS Glue など、多くの AWS サービスで、この形式は自動的に解析されます。

次の例では、読みやすくするために DynamoDB JSON エクスポートの 1 つの項目が複数行にフォーマットされています。

```
{
    "Item":{
        "Authors":{
            "SS":[
                "Author1",
                "Author2"
            ]
        },
        "Dimensions":{
            "S":"8.5 x 11.0 x 1.5"
        },
        "ISBN":{
            "S":"333-3333333333"
        },
```

```
        "Id":{
            "N":"103"
        },
        "InPublication":{
            "BOOL":false
        },
        "PageCount":{
            "N":"600"
        },
        "Price":{
            "N":"2000"
        },
        "ProductCategory":{
            "S":"Book"
        },
        "Title":{
            "S":"Book 103 Title"
        }
    }
}
```

Amazon Ion

Amazon Ion は、大規模なサービス指向アーキテクチャをエンジニアリングしながら、急速な開発、デカップリング、効率性といった日々の課題に対処するために構築されたリッチタイプの自己記述型階層型データシリアル化フォーマットです。DynamoDB は、Ion のテキストのフォーマットでのテーブルのエクスポートをサポートします。これは JSON のスーパーセットです。

テーブルを Ion 形式にエクスポートすると、テーブルで使用されている DynamoDB データ型が Ion データ型にマップされます。DynamoDB セットは、Ion 型注釈を使用して、ソーステーブルで使用されるデータ型を明確にします。

次の表に、DynamoDB データ型と ion データ型のマッピングを示します。

DynamoDB データ型	Ion 表現
文字列 (S)	文字列
ブール型 (BOOL)	ブール
数値 (N)	小数点

DynamoDB データ型	Ion 表現
バイナリ (B)	blob
セット (SS、NS、BS)	リスト (型注釈 $DynamoDB_SS、$DynamoDB_NS、または $DynamoDB_BS)
リスト	リスト
マップ	構造体

Ion エクスポート内の項目は、改行で区切られます。各行は Ion バージョンマーカーで始まり、Ion 形式の項目が続きます。次の例では、読みやすくするために 1 つの Ion エクスポートの項目が複数の行にフォーマットされています。

```
$ion_1_0 {
    Item:{
        Authors:$dynamodb_SS::["Author1","Author2"],
        Dimensions:"8.5 x 11.0 x 1.5",
        ISBN:"333-3333333333",
        Id:103.,
        InPublication:false,
        PageCount:6d2,
        Price:2d3,
        ProductCategory:"Book",
        Title:"Book 103 Title"
    }
}
```

増分エクスポート出力

マニフェストファイル

DynamoDB では、エクスポートリクエストごとにマニフェストファイルがチェックサムファイルとともに指定した S3 バケットに作成されます。

```
export-prefix/AWSDynamoDB/ExportId/manifest-summary.json
export-prefix/AWSDynamoDB/ExportId/manifest-summary.checksum
export-prefix/AWSDynamoDB/ExportId/manifest-files.json
export-prefix/AWSDynamoDB/ExportId/manifest-files.checksum
```

テーブルのエクスポートをリクエストする際は、**export-prefix** を選択します。これにより、エクスポート先 S3 バケットのファイルを整理できます。**ExportId** は、同じ S3 バケットおよび export-prefix への複数のエクスポートが相互に上書きされないようにするために、サービスによって生成される一意のトークンです。

エクスポートでは、パーティションごとに少なくとも 1 つのファイルが作成されます。空のパーティションの場合、エクスポートリクエストにより空のファイルが作成されます。各ファイル内のすべての項目は、その特定のパーティションのハッシュされたキースペースからのものです。

> **Note**
>
> また、DynamoDB では、マニフェストファイルと同じディレクトリに _started という名前の空のファイルが作成されます。このファイルは、エクスポート先のバケットが書き込み可能であること、およびエクスポートが開始されたことを検証します。これは安全に削除できます。

要約マニフェスト

manifest-summary.json ファイルには、エクスポートジョブに関する概要情報が含まれています。これにより、共有データフォルダー内のどのデータファイルがこのエクスポートに関連付けられているかがわかります。この形式は次のとおりです。

```
{
 "version": "2023-08-01",
 "exportArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test/
export/01695097218000-d6299cbd",
 "startTime": "2023-09-19T04:20:18.000Z",
 "endTime": "2023-09-19T04:40:24.780Z",
 "tableArn": "arn:aws:dynamodb:us-east-1:599882009758:table/export-test",
 "tableId": "b116b490-6460-4d4a-9a6b-5d360abf4fb3",
 "exportFromTime": "2023-09-18T17:00:00.000Z",
 "exportToTime": "2023-09-19T04:00:00.000Z",
 "s3Bucket": "jason-exports",
 "s3Prefix": "20230919-prefix",
 "s3SseAlgorithm": "AES256",
 "s3SseKmsKeyId": null,
 "manifestFilesS3Key": "20230919-prefix/AWSDynamoDB/01693685934212-ac809da5/manifest-
files.json",
 "billedSizeBytes": 20901239349,
 "itemCount": 169928274,
```

```
 "outputFormat": "DYNAMODB_JSON",
 "outputView": "NEW_AND_OLD_IMAGES",
 "exportType": "INCREMENTAL_EXPORT"
}
```

ファイルマニフェスト

manifest-files.json ファイルには、エクスポートされたテーブルデータを含むファイルに関する情報が含まれています。ファイルは JSON Lines 形式であるため、改行が項目区切り文字として使用されます。次の例では、読みやすくするためにファイルマニフェストの 1 つのデータファイルの詳細が複数行にフォーマットされています。

```
{
"itemCount": 8,
   "md5Checksum": "sQMSpEILNgoQmarvDFonGQ==",
   "etag": "af83d6f217c19b8b0fff8023d8ca4716-1",
   "dataFileS3Key": "AWSDynamoDB/data/sgad6417s6vss4p7owp0471bcq.json.gz"
}
```

データファイル

DynamoDB では、DynamoDB JSON と Amazon Ion の 2 つの形式でテーブルデータをエクスポートできます。選択した形式にかかわらず、データは、 キーによって名前が付けられた複数の圧縮ファイルに書き込まれます。これらのファイルも manifest-files.json ファイル内に一覧表示されます。

増分エクスポートのデータファイルはすべて S3 バケットの共通データフォルダーに格納されています。マニフェストファイルはエクスポート ID フォルダーにあります。

```
amzn-s3-demo-bucket/DestinationPrefix
.
### AWSDynamoDB
    ### 01693685934212-ac809da5     // an incremental export ID
    #   ### manifest-files.json     // manifest points to files under 'data' folder
    #   ### manifest-files.checksum
    #   ### manifest-summary.json   // stores metadata about request
    #   ### manifest-summary.md5
    #   ### _started                // empty file for permission check
    ### 01693686034521-ac809da5
    #   ### manifest-files.json
    #   ### manifest-files.checksum
```

```
    #   ### manifest-summary.json
    #   ### manifest-summary.md5
    #   ### _started
    ### data                        // stores all the data files for incremental
exports
    #   ### sgad6417s6vss4p7owp0471bcq.json.gz
    #   ...
```

ファイルをエクスポートする場合、各項目の出力には、テーブル内でその項目が更新された日時を表すタイムスタンプと、それが `insert`、`update`、または `delete` オペレーションであったかどうかを示すデータ構造が含まれます。タイムスタンプは内部システムクロックに基づいており、アプリケーションクロックとは異なる場合があります。増分エクスポートでは、出力構造の 2 つのエクスポートビュータイプ (新旧イメージ、または新しいイメージのみ) を選択できます。

- 新しいイメージには、項目の最新の状態が表示されます。
- 古いイメージには、指定した開始日時の直前の項目の状態が表示されます。

ビュータイプは、エクスポート期間中に項目がどのように変更されたかを確認する場合に役立ちます。また、特にダウンストリームシステムに DynamoDB パーティションキーと異なるパーティションキーがある場合に、ダウンストリームシステムを効率的に更新するのにも役立ちます。

増分エクスポート出力の項目が `insert`、`update`、または `delete` のどれであったかは、出力の構造を見れば推測できます。両方のエクスポートビュータイプについて、増分エクスポートの構造とそれに対応するオペレーションを以下の表にまとめています。

Operation	新しいイメージのみ	新旧イメージ
Insert	キー + 新しいイメージ	キー + 新しいイメージ
更新	キー + 新しいイメージ	キー + 新しいイメージ + 古いイメージ
削除	キー	キー + 古いイメージ
delete + insert	出力なし	出力なし

DynamoDB JSON

DynamoDB JSON 形式のテーブルエクスポートは、項目の書き込み時間を示すメタデータタイムスタンプと、それに続く項目のキーと値で構成されています。以下は、エクスポートビュータイプを新イメージおよび旧イメージとして使用した DynamoDB JSON 出力の例を示しています。

```
// Ex 1: Insert
//   An insert means the item did not exist before the incremental export window
//   and was added during the incremental export window

{
   "Metadata": {
     "WriteTimestampMicros": {
       "N": "1680109764000000"
     }
   },
   "Keys": {
     "PK": {
       "S": "CUST#100"
     }
   },
   "NewImage": {
     "PK": {
       "S": "CUST#100"
     },
     "FirstName": {
       "S": "John"
     },
     "LastName": {
       "S": "Don"
     }
   }
}

// Ex 2: Update
//   An update means the item existed before the incremental export window
//   and was updated during the incremental export window.
//   The OldImage would not be present if choosing "New images only".

{
   "Metadata": {
     "WriteTimestampMicros": {
       "N": "1680109764000000"
```

```
      }
    },
    "Keys": {
      "PK": {
        "S": "CUST#200"
      }
    },
    "OldImage": {
      "PK": {
        "S": "CUST#200"
      },
      "FirstName": {
        "S": "Mary"
      },
      "LastName": {
        "S": "Grace"
      }
    },
    "NewImage": {
      "PK": {
        "S": "CUST#200"
      },
      "FirstName": {
        "S": "Mary"
      },
      "LastName": {
        "S": "Smith"
      }
    }
}

// Ex 3: Delete
//   A delete means the item existed before the incremental export window
//   and was deleted during the incremental export window
//   The OldImage would not be present if choosing "New images only".

{
    "Metadata": {
      "WriteTimestampMicros": {
        "N": "1680109764000000"
      }
    },
    "Keys": {
      "PK": {
```

```
        "S": "CUST#300"
      }
    },
    "OldImage": {
      "PK": {
        "S": "CUST#300"
      },
      "FirstName": {
        "S": "Jose"
      },
      "LastName": {
        "S": "Hernandez"
      }
    }
}

// Ex 4: Insert + Delete
//   Nothing is exported if an item is inserted and deleted within the
//   incremental export window.
```

Amazon Ion

Amazon Ion は、大規模なサービス指向アーキテクチャをエンジニアリングしながら、急速な開発、デカップリング、効率性といった日々の課題に対処するために構築されたリッチタイプの自己記述型階層型データシリアル化フォーマットです。DynamoDB は、Ion のテキストのフォーマットでのテーブルのエクスポートをサポートします。これは JSON のスーパーセットです。

テーブルを Ion 形式にエクスポートすると、テーブルで使用されている DynamoDB データ型が Ion データ型にマップされます。DynamoDB セットは、Ion 型注釈を使用して、ソーステーブルで使用されるデータ型を明確にします。

次の表に、DynamoDB データ型と ion データ型のマッピングを示します。

DynamoDB データ型	Ion 表現
文字列 (S)	文字列
ブール型 (BOOL)	ブール
数値 (N)	小数点
バイナリ (B)	blob

DynamoDB データ型	Ion 表現
セット (SS、NS、BS)	リスト (型注釈 $DynamoDB_SS、$DynamoDB_NS、または $DynamoDB_BS)
リスト	リスト
マップ	構造体

Ion エクスポート内の項目は、改行で区切られます。各行は Ion バージョンマーカーで始まり、Ion 形式の項目が続きます。次の例では、読みやすくするために 1 つの Ion エクスポートの項目が複数の行にフォーマットされています。

```
$ion_1_0 {
    Record:{
        Keys:{
             ISBN:"333-3333333333"
         },
        Metadata:{
            WriteTimestampMicros:1684374845117899.
        },
        OldImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            ISBN:"333-3333333333",
            Id:103.,
            InPublication:false,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        },
        NewImage:{
            Authors:$dynamodb_SS::["Author1","Author2"],
            Dimensions:"8.5 x 11.0 x 1.5",
            ISBN:"333-3333333333",
            Id:103.,
            InPublication:true,
            PageCount:6d2,
            Price:2d3,
            ProductCategory:"Book",
            Title:"Book 103 Title"
        }
    }
```

```
}
```

Amazon SageMaker Lakehouse との DynamoDB ゼロ ETL 統合

Amazon SageMaker Lakehouse との DynamoDB ゼロ ETL 統合により、DynamoDB のデータを Amazon SageMaker Lakehouse に自動的にレプリケートすることで、カスタムデータ移動パイプラインを構築する必要がなくなります。このノーコード統合により、お客様は DynamoDB のテーブル容量を消費することなく、Amazon SageMaker Lakehouse を使用して DynamoDB データ上で分析ワークロードを実行できます。統合により、テーブルからデータが自動的にエクスポートされ、通常は 15 ~ 30 分以内にターゲットを最新の状態に保ちます。

トピック

- Amazon SageMaker Lakehouse との DynamoDB ゼロ ETL 統合

Amazon SageMaker Lakehouse との DynamoDB ゼロ ETL 統合

DynamoDB テーブルと Amazon SageMaker Lakehouse の統合を設定するには、AWS Glue がソースからデータにアクセスしてターゲットに書き込むために使用する IAM ロールの設定や、中間地点またはターゲットの場所にあるデータを暗号化するための KMS キーの使用などの前提条件が必要です。

トピック

- DynamoDB の Amazon SageMaker Lakehouse とのゼロ ETL 統合を作成する前の前提条件
- DynamoDB の Amazon SageMaker Lakehouse とのゼロ ETL 統合の作成
- 統合に関する CloudWatch メトリクスの表示

DynamoDB の Amazon SageMaker Lakehouse とのゼロ ETL 統合を作成する前の前提条件

DynamoDB ソースとのゼロ ETL 統合を設定するには、AWS Glue が DynamoDB テーブルのデータにアクセスしてエクスポートできるようにするリソースベースのアクセス (RBAC) ポリシーを設定する必要があります。ポリシーには、特定の AWS アカウントおよびリージョンへのアクセスを制限する条件付きで、`ExportTableToPointInTime`、`DescribeTable`、`DescribeExport` などの特定のアクセス許可を含める必要があります。詳細については、「Amazon DynamoDB ソースの設定」を参照してください。

テーブルに対してポイントインタイムリカバリ (PITR) を有効にする必要があり、AWS CLI コマンドを使用してポリシーを適用できます。ポリシーをさらに絞り込むには、完全な統合 ARN を指定して、より制限的なアクセス制御を行います。詳細については、「ゼロ ETL 統合を設定するための前提条件」を参照してください。

DynamoDB の Amazon SageMaker Lakehouse とのゼロ ETL 統合の作成

統合の前提条件を満たしたら、以下のガイダンスに従ってゼロ ETL 統合を作成、変更、または削除できます。

統合の作成

統合を作成するには

1. AWS マネジメントコンソールにサインインし、https://console.aws.amazon.com/dynamodbv2 で Amazon DynamoDB コンソールを開きます。
2. ナビゲーションペインで、[統合] を選択します。
3. [Amazon SageMaker Lakehouse とのゼロ ETL 統合を作成] を選択し、[次へ] を選択します。
4. 統合を作成するには、「統合の作成」を参照してください。
5. 統合を変更するには、「統合の変更」を参照してください。
6. 統合を削除するには、「統合の削除」を参照してください。
7. クロスアカウント統合を設定するには、「クロスアカウント統合の設定」を参照してください。

ターゲット Amazon S3 Tables での圧縮の有効化

圧縮を有効にすると、Amazon Athena のクエリパフォーマンスを向上させることができます。

まず、必要な IAM ロールの設定など、圧縮リソースの前提条件の設定を完了します。IAM ロールの詳細な設定手順については、Lake Formation のドキュメントを参照してください。「圧縮のためのテーブルの最適化」を参照してください。

統合中に作成された AWS Glue テーブルで圧縮を有効にするには、Lake Formation の圧縮有効化プロセスに従います。これにより、テーブルのパフォーマンスとクエリ効率を最適化できます。

統合に関する CloudWatch メトリクスの表示

統合が完了すると、AWS Glue ジョブごとにアカウントで生成された以下の CloudWatch メトリクスと EventBridge 通知が表示されます。詳細については、「統合のモニタリング」を参照してください。

DynamoDB と Amazon OpenSearch Service のゼロ ETL 統合

Amazon DynamoDB は、OpenSearch Ingestion 用 DynamoDB プラグインを通じて Amazon OpenSearch Service とのゼロ ETL 統合が可能です。Amazon OpenSearch Ingestion では、コードを必要としないフルマネージド型のエクスペリエンスで Amazon OpenSearch Service にデータを取り込むことができます。

OpenSearch Ingestion 用 DynamoDB プラグインを使用すると、1 つ以上の DynamoDB テーブルをソースとして使用して、1 つ以上の OpenSearch Service インデックスに取り込むことができます。AWS マネジメントコンソールの OpenSearch Ingestion または DynamoDB インテグレーションから、DynamoDB をソースとして使用する OpenSearch Ingestion パイプラインを参照して設定できます。

- OpenSearch Ingestion を初めて使う場合は、「OpenSearch Ingestion の開始方法」に記載されている手順に従ってください。
- DynamoDB プラグインの前提条件とすべての設定オプションについては、OpenSearch Ingestion 用 DynamoDB プラグインのドキュメントを参照してください。

仕組み

このプラグインは DynamoDB の Amazon S3 へのエクスポート機能を使用して、OpenSearch にロードする初期スナップショットを作成します。スナップショットがロードされると、プラグインは DynamoDB Streams を使用してそれ以降の変更をほぼリアルタイムでレプリケートします。すべての項目は OpenSearch Ingestion でイベントとして処理され、プロセッサプラグインで変更できます。属性を削除したり、複合属性を作成してルート経由でさまざまなインデックスに送信したりできます。

Amazon S3 へのエクスポート機能を使用するには、ポイントインタイムリカバリ (PITR) を有効にする必要があります。また、DynamoDB Streams を使用するには、([新旧のイメージ] オプションが選択されている状態で) DynamoDB Streams が有効になっている必要があります。エクスポート設定を除外すると、スナップショットを作ることなくパイプラインを作成できます。

また、ストリーム設定を除外すると、スナップショットのみで更新のないパイプラインを作成することもできます。このプラグインはテーブルの読み取りスループットや書き込みスループットを使用しないため、本番環境のトラフィックに影響を与えることなく安全に使用できます。ストリーム上の並列コンシューマーの数には制限があり、こうした統合を作成する前に考慮する必要があります。その他の考慮事項については、「the section called “統合に関するベストプラクティス”」を参照してください。

単純なパイプラインの場合、1 つの OpenSearch Compute Unit (OCU) は、1 秒あたり約 1 MB の書き込みを処理できます。これは約 1,000 の書き込みリクエスト単位 (WCU) に相当しますが、パイプラインの複雑さやその他の要因によって、これより多くなることも少なくなることもあります。

OpenSearch Ingestion は、回復不能なエラーの原因となるイベントのデッドレターキュー (DLQ) をサポートします。さらに、DynamoDB、パイプライン、または Amazon OpenSearch Service のいずれかでサービスが中断された場合でも、パイプラインは中断したところからユーザーの介入なしに再開することができます。

中断が 24 時間以上続くと、更新が失われる場合があります。ただし、中断から回復すると、パイプラインはまだ利用できる更新を引き続き処理します。デッドレターキューに入っていない限り、中断したイベントに起因する異常を修正するには、新たにインデックスを構築する必要があります。

プラグインのすべての設定と詳細については、OpenSearch Ingestion 用 DynamoDB プラグインのドキュメントを参照してください。

コンソールによる作成エクスペリエンスの統合

DynamoDB と OpenSearch Service は AWS マネジメントコンソールでエクスペリエンスが統合されており、開始プロセスが効率化されています。以下の手順を実行すると、サービスで自動的に DynamoDB ブループリントが選択され、適切な DynamoDB 情報が自動的に追加されます。

統合を作成するには、「OpenSearch Ingestion の開始方法」に記載されている手順に従ってください。「ステップ 3: パイプラインを作成する」まで進んだら、手順の 1 と 2 を以下の手順に置き換えます。

1. DynamoDB コンソールに移動します。
2. 左側のナビゲーションペインで、[統合] を選択します。
3. OpenSearch にレプリケートする DynamoDB テーブルを選択します。
4. [作成] を選択します。

ここから、チュートリアルの以降の部分を続行します。

次のステップ

DynamoDB と OpenSearch Service の統合についての理解を深めるには、以下を参照してください。

- Amazon OpenSearch Ingestion の開始方法
- DynamoDB プラグインの設定と要件

インデックスの重大な変更の処理

OpenSearch ではインデックスに新しい属性を動的に追加できます。ただし、マッピングテンプレートが特定のキーに設定されている場合は、そのキーを変更する際に追加のアクションが必要になります。さらに、変更によって DynamoDB テーブルの全データの再処理が必要になる場合は、新たなエクスポートを開始するための手順を実行する必要があります。

> **Note**
>
> これらのどのオプションでも、DynamoDB テーブルと指定したマッピングテンプレートで型が競合していると、問題が発生する可能性があります。デッドレターキュー (DLQ) を必ず有効にしてください (開発中の場合でも)。これにより、OpenSearch のインデックスでのインデックス作成時に競合を引き起こすレコードについて、何が間違っているのかを把握しやすくなります。

トピック

- 仕組み
- インデックスを削除してパイプラインをリセットする (パイプライン中心のオプション)
- インデックスを再作成してパイプラインをリセットする (インデックス中心のオプション)
- 新しいインデックスとシンクを作成する (オンラインオプション)
- 型の競合の回避とデバッグのベストプラクティス

仕組み

以下は、インデックスの重大な変更を処理するときに実行されるアクションの概要です。詳細な手順については、以降のセクションで説明します。

- パイプラインを停止して開始する: このオプションでは、パイプラインの状態をリセットし、パイプラインを新たにフルエクスポートで再開します。非破壊的なオプションであるため、DynamoDB 内のインデックスやデータは削除されません。これを行う前に新しいインデックスを作成しないと、インデックス内の現在の `_version` よりも古いドキュメントの挿入が工クスポートによって試行されるため、バージョンの競合による工ラーが多数発生する可能性がありま

す。これらのエラーは無視して問題ありません。パイプラインの停止中は、パイプラインの料金が課金されることはありません。

- パイプラインを更新する: このオプションでは、状態を変えることなく、パイプラインの設定をブルー/グリーン方式で更新します。パイプラインに大幅な変更 (既存のインデックスに新しいルート、インデックス、またはキーを追加するなど) を行った場合は、パイプラインの完全なリセットとインデックスの再作成が必要になる場合があります。このオプションではフルエクスポートは実行されません。
- インデックスを削除して再作成する: このオプションでは、インデックスのデータ設定とマッピング設定が削除されます。マッピングに重大な変更を加える前にはこれを行う必要があります。インデックスに依存しているアプリケーションは、インデックスが再作成されて同期されるまで機能しません。インデックスを削除しても、新たなエクスポートは開始されません。インデックスの削除は、パイプラインを更新した後にのみ行ってください。そうしないと、設定を更新する前にインデックスが再作成される可能性があります。

インデックスを削除してパイプラインをリセットする (パイプライン中心のオプション)

開発途中の場合は、通常、この方法が最速のオプションになります。OpenSearch Service でインデックスを削除し、パイプラインの停止と開始を行って、全データの新たなエクスポートを開始します。これにより、マッピングテンプレートが既存のインデックスと競合したり、処理が不完全なテーブルからデータが失われたりすることがなくなります。

1. AWS マネジメントコンソールから、あるいは AWS CLI か SDK で StopPipeline API オペレーションを使用して、パイプラインを停止します。
2. 新しい変更でパイプライン設定を更新します。
3. REST API コールまたは OpenSearch ダッシュボードを使用して OpenSearch Service のインデックスを削除します。
4. コンソールから、あるいは AWS CLI か SDK で `StartPipeline` API オペレーションを使用して、パイプラインを開始します。

   > **Note**
   >
   > これにより新たなフルエクスポートが開始され、追加費用が発生します。

5. 新しいインデックスを作成するための新たなエクスポートが生成されるため、予期しない問題がないか監視します。

6. OpenSearch Service でインデックスが想定どおりであることを確認します。

エクスポートが完了し、ストリームからの読み取りが再開されると、DynamoDB テーブルデータがインデックスで使用できるようになります。

インデックスを再作成してパイプラインをリセットする (インデックス中心のオプション)

この方法は、パイプラインを DynamoDB から再開する前に OpenSearch Service でインデックスの設計を何度も繰り返す必要がある場合に適しています。これは、開発中に検索パターンのイテレーションを迅速に進めたい場合や、イテレーションのたびに新たなエクスポートの完了を待機したくない場合に役立ちます。

1. AWS マネジメントコンソールから、あるいは AWS CLI か SDK で StopPipeline API オペレーションを呼び出して、パイプラインを停止します。
2. OpenSearch でインデックスを削除し、目的のマッピングテンプレートを使って再作成します。サンプルデータを手動で挿入して、検索が意図したとおりに機能しているかを確認できます。サンプルデータが DynamoDB のデータと競合する可能性がある場合は、次のステップに進む前に必ず削除してください。
3. パイプラインにインデックステンプレートがある場合は、削除するか、OpenSearch Service で作成済みのテンプレートに置き換えます。インデックスの名前がパイプラインでの名前と一致していることを確認してください。
4. コンソールから、あるいは AWS CLI か SDK で `StartPipeline` API オペレーションを呼び出して、パイプラインを開始します。

   > **Note**
   >
   > これにより新たなフルエクスポートが開始され、追加費用が発生します。

5. 新しいインデックスを作成するための新たなエクスポートが生成されるため、予期しない問題がないか監視します。

エクスポートが完了し、ストリームからの読み取りが再開されると、DynamoDB テーブルデータがインデックスで使用できるようになります。

新しいインデックスとシンクを作成する (オンラインオプション)

この方法は、マッピングテンプレートを更新する必要があるものの、現在インデックスを本番環境で使用している場合に適しています。このオプションでは、新しいインデックスを作成し、同期と検証が完了した後にアプリケーションをそのインデックスに移動する必要があります。

> **Note**
>
> これにより、ストリーム上に別のコンシューマーが作成されます。AWS Lambda やグローバルテーブルのようなコンシューマーが他にもあると、問題になる可能性があります。新しいインデックスをロードするキャパシティを確保するために、既存のパイプラインの更新を一時停止しなければならなくなる場合があります。

1. 新しい設定と別のインデックス名を使用して新しいパイプラインを作成します。
2. 新しいインデックスに予期しない問題がないか監視します。
3. アプリケーションを新しいインデックスに切り替えます。
4. すべてが正しく動作していることを確認したら、古いパイプラインを停止して削除します。

型の競合の回避とデバッグのベストプラクティス

- 型の競合が生じた場合のデバッグを容易にするためには、デッドレターキュー (DLQ) を常に使用します。
- マッピングが含まれるインデックステンプレートを必ず使用し、`include_keys` を設定します。OpenSearch Service は新しいキーを動的にマッピングしますが、これによって予期しない動作 (`GeoPoint` を想定していたところ、`string` または `object` として作成された場合など) や エラー (`long` 値と `float` 値が混在している `number` など) の問題が発生する可能性があります。
- 既存のインデックスを本番環境で引き続き使用する必要がある場合は、前述のインデックスの削除手順を、パイプライン設定ファイル内のインデックスの名前を変更する手順に置き換えることもできます。これにより、新しいインデックスが作成されます。完了したら、新しいインデックスをポイントするようにアプリケーションを更新する必要があります。
- 型変換の問題をプロセッサで修正する場合は、`UpdatePipeline` を使ってテストできます。そのためには、停止と開始の手順を行うか、デッドレターキューの処理を実行して、以前にスキップされたエラーのあるドキュメントを修正する必要があります。

DynamoDB と OpenSearch Service のゼロ ETL 統合を使用するためのベストプラクティス

DynamoDB は、Amazon OpenSearch Service との DynamoDB のゼロ ETL 統合を行うことができます。詳細については、「DynamoDB plugin for OpenSearch Ingestion」および「specific best practices for Amazon OpenSearch Service」を参照してください。

設定

- 検索を実行する必要があるデータのみをインデックス化します。実装する際は、必ずマッピングテンプレート (`template_type: index_template`、`template_content`) および `include_keys` を使用します。
- ログを監視して、タイプの競合に関連するエラーがないか確認します。OpenSearch Service は、キーのすべての値が同じタイプであることを前提としています。同じでない場合、例外が生成されます。このようなエラーが発生した場合は、プロセッサを追加して、特定のキーが常に同じ値であるかどうかを確認できます。
- 通常、`document_id` 値には `primary_key` メタデータ値を使用します。OpenSearch Service では、ドキュメント ID は DynamoDB のプライマリキーと同等です。プライマリキーを使用するとドキュメントを見つけやすくなり、アップデートが矛盾なく常にそのドキュメントにレプリケートされます。

  プライマリキー (例: `document_id: "${getMetadata('primary_key')}"`) は、`getMetadata` 補助関数を使用して取得できます。複合プライマリキーを使用している場合は、補助関数によって連結されます。
- 通常、`action` 設定には `opensearch_action` メタデータ値を使用します。これにより、OpenSearch Service のデータが DynamoDB の最新の状態と一致するようにアップデートがレプリケートされるようになります。

  プライマリキー (例: `action: "${getMetadata('opensearch_action')}"`) は、`getMetadata` 補助関数を使用して取得できます。フィルタリングなどの用途では、`dynamodb_event_name` を使用してストリームイベントタイプを取得することもできます。ただし、通常は設定に `action` を使用しないでください。

オブザーバビリティ

- ドロップされたイベントを処理するには、必ず OpenSearch シンクでデッドレターキュー (DLQ) を使用します。DynamoDB は一般的に OpenSearch Service ほど構造化されておらず、予期しな

い問題が発生する可能性は常にあります。デッドレターキューを使用すると、個々のイベントを回復できるだけでなく、回復プロセスを自動化することもできます。これにより、インデックス全体を再構築する必要がなくなります。

- レプリケーションの遅延が想定を超えたことを知らせるアラートを必ず設定します。通常は、アラートノイズがあまり大きくならないように、設定を 1 分にしておくと安全です。これは、書き込みトラフィックの急増度や、パイプラインの OpenSearch Compute Unit (OCU) 設定によって異なる場合があります。

  レプリケーションの遅延が 24 時間を超えると、ストリームはイベントをドロップし始め、インデックスを最初から完全に再構築しない限り、精度の問題が発生します。

スケーリング

- パイプラインに自動スケーリングを使用すると、ワークロードに合わせて OCU をスケールアップまたはスケールダウンできます。
- 自動スケーリングを使用しないプロビジョニングされたスループットテーブルでは、書き込みキャパシティユニット (WCU) を 1000 で割った値に基づいて OCU を設定することをお勧めします。最小値をその値より 1 OCU 低く (ただし 1 以上)、最大値をその値を 1 OCU 上回るように設定します。
  - 計算式:

    ```
    OCU_minimum = GREATEST((table_WCU / 1000) - 1, 1)
    OCU_maximum = (table_WCU / 1000) + 1
    ```

  - 例: あるテーブルでは 25000 の WCU がプロビジョニングされています。パイプラインの OCU は、最小値を 24 (25000/1000 - 1) に、最大値を 26 (25000/1000 + 1) 以上に設定する必要があります。
- 自動スケーリングを使用するプロビジョニングされたスループットテーブルでは、最小および最大 WCU を 1000 で割った値に基づいて OCU を設定することをお勧めします。最小値を DynamoDB の最小値より 1 OCU 低く設定し、最大値を DynamoDB の最大値より少なくとも 1 OCU 高く設定します。
  - 計算式:

    ```
    OCU_minimum = GREATEST((table_minimum_WCU / 1000) - 1, 1)
    OCU_maximum = (table_maximum_WCU / 1000) + 1
    ```

  - 例: あるテーブルには、最小 8000、最大 14000 に設定された自動スケーリングポリシーがあります。パイプラインの OCU は、最小値を 7 (8000/1000 - 1) に、最大値を 15 (14000/1000 + 1) に設定する必要があります。
- オンデマンドスループットテーブルでは、1 秒あたりの書き込みリクエストユニット数の一般的な最大値と最小値に基づいて OCU を設定することをお勧めします。利用可能な集計によっては、より長い期間にわたる平均値の算出が必要な場合があります。最小値を DynamoDB の最小値より 1 OCU 低く設定し、最大値を DynamoDB の最大値より少なくとも 1 OCU 高く設定します。
  - 計算式:

```
# Assuming we have writes aggregated at the minute level
OCU_minimum = GREATEST((min(table_writes_1min) / (60 * 1000)) - 1, 1)
OCU_maximum = (max(table_writes_1min) / (60 * 1000)) + 1
```

  - 例: あるテーブルの書き込みリクエストユニット数の平均最小値は 1 秒あたり 300 で、平均最大値は 4300 です。パイプラインの OCU は、最小値を 1 (300/1000 - 1、ただし 1 以上) に、最大値を 5 (4300/1000 + 1) に設定する必要があります。
- 送信先の OpenSearch Service インデックスのスケーリングに関するベストプラクティスに従ってください。インデックスのスケーリングが十分でない場合、DynamoDB からの取り込みが遅くなり、遅延が発生する可能性があります。

> **Note**
>
> `GREATEST` は、引数のセットを指定すると最大値の引数を返す SQL 関数です。

DynamoDB と Amazon EventBridge の統合

Amazon DynamoDB は、変更データキャプチャのための DynamoDB Streams を提供し、DynamoDB テーブルでの項目レベルの変更のキャプチャを行うことができます。DynamoDB Streams は Lambda 関数を呼び出してこれらの変更を処理することができるため、他のサービスやアプリケーションとのイベント駆動型の統合が可能になります。DynamoDB Streams はフィルタリングもサポートしているため、効率的でターゲットを絞ったイベント処理が可能になります。

DynamoDB Streams は、各シャードで最大 2 つの同時コンシューマーをサポートし、Lambda イベントフィルタリングによるフィルタリングをサポートして、特定の条件に一致する項目のみが処理されるようにします。一部のお客様は、3 つ以上のコンシューマーをサポートする要件がある場合があ

ります。また他のお客様は、変更イベントを処理する前にエンリッチメントが必要な場合や、より高度なフィルタリングとルーティングが必要な場合もあります。

DynamoDB を EventBridge を統合することで、これらの要件をサポートできます。

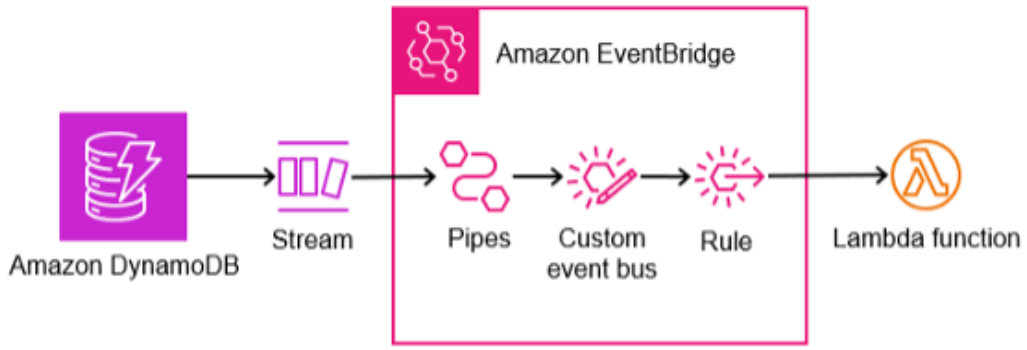
Amazon EventBridge は、イベントを使用してアプリケーションコンポーネント同士を接続するサーバーレスサービスです。これにより、スケーラブルなイベント駆動型アプリケーションを簡単に構築できます。EventBridge は、EventBridge Pipes を介して Amazon DynamoDB とネイティブに統合されるため、DynamoDB から EventBridge バスへのシームレスなデータフローが可能になります。その後、バスは一連のルールとターゲットを通じて複数のアプリケーションやサービスにファンアウトできます。

トピック

- 仕組み
- コンソールを使用した統合の作成
- 次のステップ

仕組み

DynamoDB パイプと EventBridge パイプの統合では、DynamoDB Streams を使用して、DynamoDB テーブル内の項目レベルの変更の時系列シーケンスをキャプチャします。この方法でキャプチャされた各レコードには、テーブルで変更されたデータが含まれます。



EventBridge パイプは DynamoDB Streams からのイベントを消費し、EventBridge バスなどのターゲットにルーティングします (イベントバスはイベントを受信して送信先に配信するルーターで、ターゲットとも呼ばれます)。配信は、イベントの内容に一致するルールに基づいています。このパ

イプには、特定のイベントをフィルタリングするオプションや、ターゲットに送信される前のイベントデータのエンリッチメントオプションも含まれています。

EventBridge は複数のターゲットタイプをサポートしていますが、ファンアウト設計を実装する場合の一般的な選択肢は、ターゲットとして Lambda 関数を使用することです。次の例では、Lambda 関数ターゲットとの統合の例を示します。

コンソールを使用した統合の作成

AWS マネジメントコンソール を使用して統合を作成するには、以下の手順に従います。

1. DynamoDB デベロッパーガイドの「ストリームの有効化」セクションの手順に従って、ソーステーブルで DynamoDB Streams を有効にします。ソーステーブルで DynamoDB Streams がすでに有効になっている場合は、コンシューマーの数が 2 つ以下であることを確認します。コンシューマーには、Lambda 関数、DynamoDB グローバルテーブル、Amazon DynamoDB と Amazon OpenSearch Service のゼロ ETL 統合、または DynamoDB Streams Kinesis Adapter などのストリームから直接読み取るアプリケーションなどが含まれます。
2. 「EventBridge ユーザーガイド」の「Creating an Amazon EventBridge event bus」セクションの手順に従って EventBridge イベントバスを作成します。
   a. イベントバスを作成する際は、[スキーマ検出] を有効化します。
3. 「EventBridge ユーザーガイド」の「Creating an Amazon EventBridge pipe」セクションの手順に従って EventBridge パイプを作成します。
   a. ソースの設定では、[ソース] フィールドで [DynamoDB] を選択し、[DynamoDB Streams] フィールドでソーステーブルストリームの名前を選択します。
   b. ターゲットの設定では、[ターゲットサービス] フィールドで [EventBridge イベントバス] を選択し、[ターゲットのイベントバス] フィールドで手順 2 で作成したイベントバスを選択します。
4. ソース DynamoDB テーブルにサンプル項目を書き込み、イベントをトリガーします。これにより、EventBridge はサンプル項目からスキーマを推測できます。このスキーマを使用して、イベントをルーティングするためのルールを作成できます。例えば、属性のオーバーロードを伴う設計パターンを実装する場合、ソートキーの値に応じて異なるルールをトリガーできます。DynamoDB への項目の書き込み方法の詳細については、「DynamoDB デベロッパーガイド」の「項目と属性の操作」を参照してください。
5. 「Lambda 開発者ガイド」の「Python による Lambda 関数の構築」セクションの手順に従ってターゲットとして使用する Python Lambda 関数の例を作成します。関数を作成する際は、以下の

サンプルコードを使用して統合のデモを行えます。コードが呼び出されると、イベントで受信した NewImage と OldImage を表示し、これは CloudWatch Logs で確認することができます。

```
import json

def lambda_handler(event, context):
    dynamodb = event.get('detail', {}).get('dynamodb', {})
    new_image = dynamodb.get('NewImage')
    old_image = dynamodb.get('OldImage')

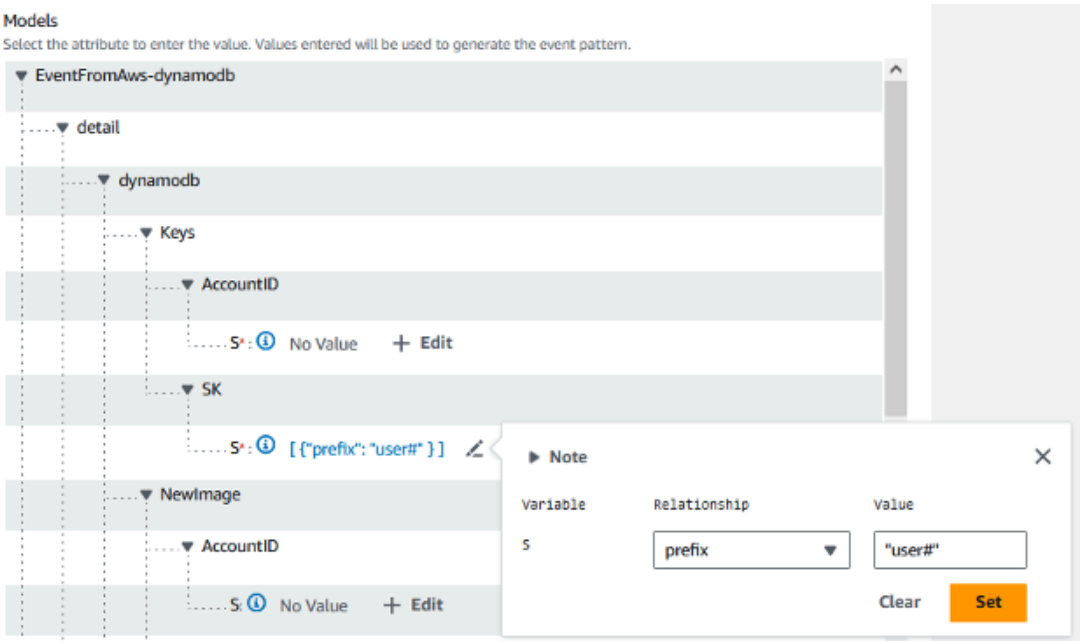
    if new_image:
        print("NewImage:", json.dumps(new_image, indent=2))
    if old_image:
        print("OldImage:", json.dumps(old_image, indent=2))

    return {'statusCode': 200, 'body': json.dumps(event)}
```

6. 「EventBridge ユーザーガイド」の「Creating Amazon EventBridge rules that react to events」セクションの手順に従って、新しい Lambda 関数にイベントをルーティングする EventBridge ルールを作成します。

   a. ルールの詳細を定義する際は、手順 2 で作成したイベントバスの名前を [イベントバス] として選択します。

   b. イベントパターンを構築する際は、既存のスキーマの指示に従います。ここでは、discovered-schemas レジストリおよびイベントで検出されたスキーマを選択できます。これにより、特定の属性に一致するメッセージのみをルーティングするユースケースに固有のイベントパターンを設定できます。例えば、SK が “user#” で始まる DynamoDB 項目のみを一致させる場合は、次のような設定を使用します。



c. スキーマに対するパターンの設計が完了したら、[JSON でイベントパターンを生成する] を選択します。代わりに DynamoDB Streams に表示されるすべてのイベントにマッチさせる場合は、イベントパターンに次の JSON を使用します。

```
{
  "source": ["aws.dynamodb"]
}
```

d. ターゲットを選択する際は、AWS サービスの指示に従います。[ターゲットの選択] フィールドで [Lambda 関数] を選択します。[関数] フィールドで、手順 5 で作成した Lambda 関数を選択します。

7. これで、「EventBridge ユーザーガイド」の「Starting or stopping schema discovery on event buses」セクションの手順に従って、イベントバスでのスキーマの検出を停止することができるようになります。

8. ソース DynamoDB テーブルに 2 つ目のサンプル項目を書き込み、イベントをトリガーします。イベントが各ステップで正常に処理されたことを確認します。

   a. 「EventBridge ユーザーガイド」の「Monitoring Amazon EventBridge」セクションに従ってイベントバスの CloudWatch メトリクス PutEventsApproximateSuccessCount を確認します。

   b. 「Lambda 開発者ガイド」の「Lambda 関数のモニタリングおよびトラブルシューティング」に従って、Lambda 関数の関数ログを確認します。Lambda 関数で提供されているサンプルコードを使用している場合は、CloudWatch Logs ロググループに DynamoDB Streams の `NewImage` と `OldImage` が出力されます。

c. 「Lambda 開発者ガイド」の「Lambda 関数のモニタリングおよびトラブルシューティング」に従って、Lambda 関数のエラー数および成功率 (%) メトリクスを確認します。

次のステップ

これは、単一の Lambda 関数をターゲットとする基本的な統合の例です。複数のルールの作成、複数のターゲットの作成、他のサービスとの統合、イベントのエンリッチメントなど、より複雑な設定をさらに理解するには、EventBridge ユーザーガイド「Getting started with EventBridge」を参照してください。

> **Note**
>
> アプリケーションに関連する可能性のある EventBridge クォータに注意してください。DynamoDB Streams のキャパシティはテーブルに合わせてスケーリングされますが、EventBridge のクォータは別のものです。大規模なアプリケーションで注意すべき一般的なクォータは、1 秒あたりのトランザクションの呼び出しスロットリング制限と 1 秒あたりのトランザクションの PutEvents スロットリング制限です。これらのクォータは、ターゲットに送信できる呼び出しの数と、1 秒あたりにバスに送信できるイベントの数を指定します。

DynamoDB と Amazon Managed Streaming for Apache Kafka の統合

Amazon Managed Streaming for Apache Kafka (Amazon MSK) では、フルマネージドで可用性の高い Apache Kafka サービスを使用して、ストリーミングデータをリアルタイムで簡単に取り込み、処理できます。

Apache Kafka は、ストリーミングデータをリアルタイムで取り込み、処理するために最適化された分散型データストアです。Kafka はレコードのストリームを処理し、レコードが生成された順序でレコードのストリームを効果的に保存し、レコードのストリームを公開およびサブスクライブできます。

これらの機能により、Apache Kafka はリアルタイムストリーミングデータパイプラインの構築によく使用されます。データパイプラインは、あるシステムから別のシステムにデータを確実に処理して移動し、それぞれが異なるユースケースをサポートする複数のデータベースの使用を容易にすることで、目的別データベース戦略を採用する上で重要な部分となる可能性があります。

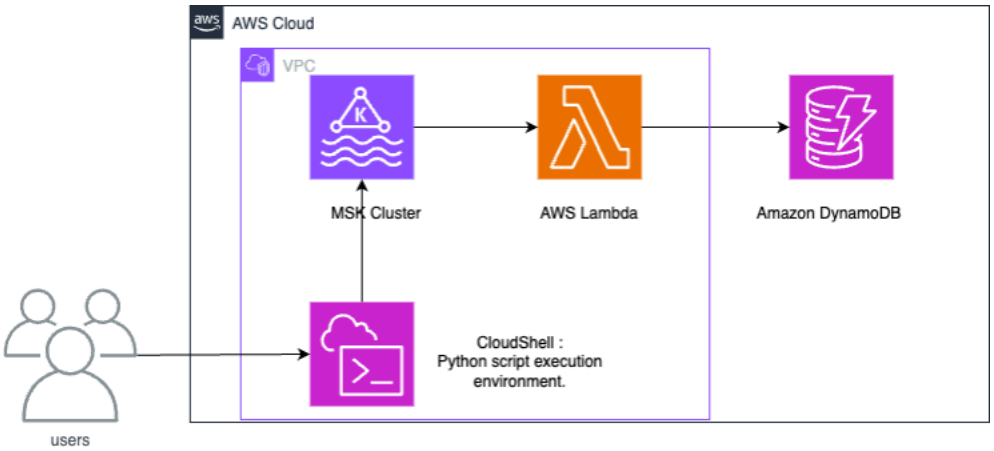
Amazon DynamoDB は、キーバリューまたはドキュメントデータモデルを使用し、一貫した 1 桁ミリ秒のパフォーマンスで無制限のスケーラビリティを求めるアプリケーションをサポートするために、これらのデータパイプラインで一般的なターゲットとなっています。

トピック

- 仕組み
- Amazon MSK と DynamoDB の統合を設定する
- 次のステップ

仕組み

Amazon MSK と DynamoDB の統合では、Lambda 関数を使用して Amazon MSK からのレコードを消費し、DynamoDB に書き込みます。



Lambda は、Amazon MSK からの新しいメッセージを内部的にポーリングした後、ターゲットの Lambda 関数を同期的に呼び出します。Lambda 関数のイベントペイロードには、Amazon MSK からのメッセージのバッチが含まれています。Amazon MSK と DynamoDB の統合のために、Lambda 関数はこれらのメッセージを DynamoDB に書き込みます。

Amazon MSK と DynamoDB の統合を設定する

> **Note**
>
> この例で使用されているリソースは、次の GitHub リポジトリでダウンロードできます。

以下の手順は、Amazon MSK と Amazon DynamoDB 間のサンプル統合を設定する方法を示しています。この例では、モノのインターネット (IoT) デバイスによって生成され、Amazon MSK に取り込まれたデータを表します。データが Amazon MSK に取り込まれると、Apache Kafka と互換性のある分析サービスまたはサードパーティーツールと統合できるため、さまざまな分析ユースケースが可能になります。DynamoDB を統合することで、個々のデバイスレコードのキー値も検索できます。

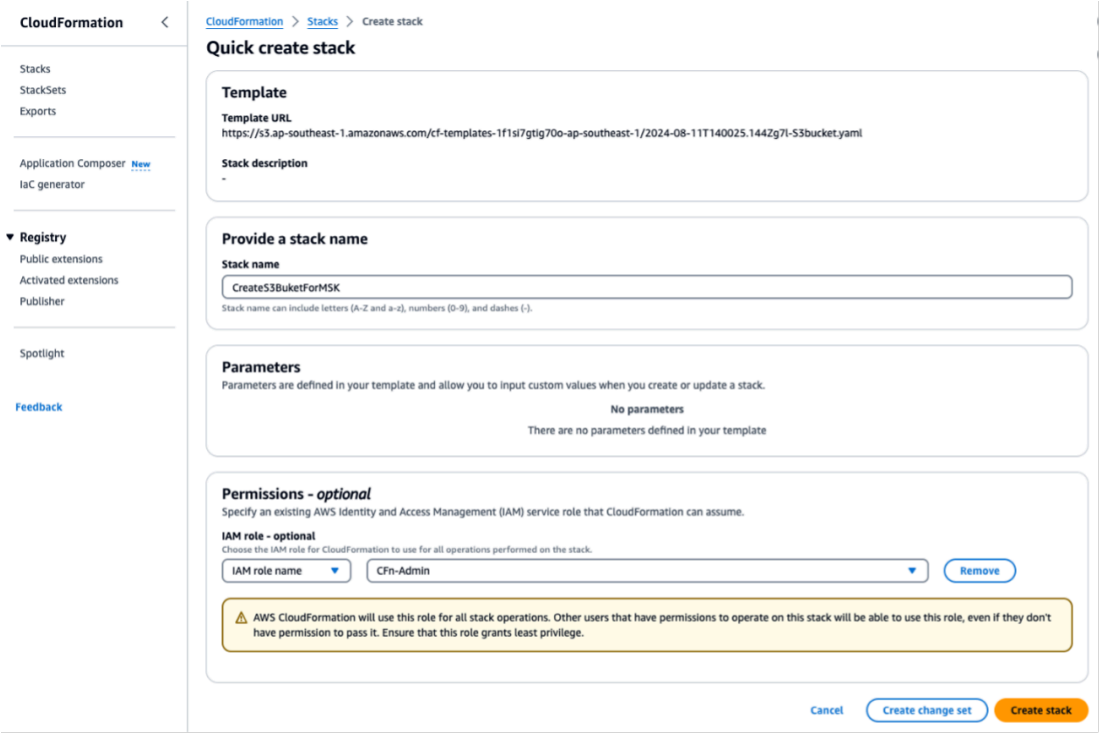
この例では、Python スクリプトが IoT センサーデータを Amazon MSK に書き込む方法を示します。次に、Lambda 関数はパーティションキー「`deviceid`」を持つ項目を DynamoDB に書き込みます。

提供された CloudFormation テンプレートは、Amazon S3 バケット、Amazon VPC、Amazon MSK クラスター、データオペレーションをテストするための AWS CloudShell のリソースを作成します。

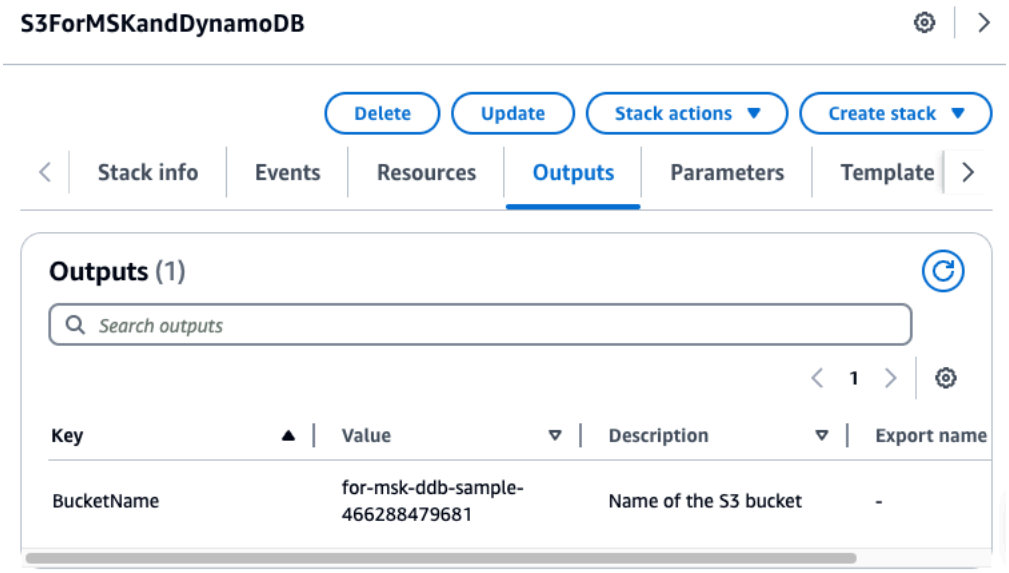
テストデータを生成するには、Amazon MSK トピックを作成し、DynamoDB テーブルを作成します。マネジメントコンソールからセッションマネージャーを使用して CloudShell のオペレーティングシステムにログインし、Python スクリプトを実行できます。

CloudFormation テンプレートを実行したら、次のオペレーションを実行して、このアーキテクチャの構築を完了できます。

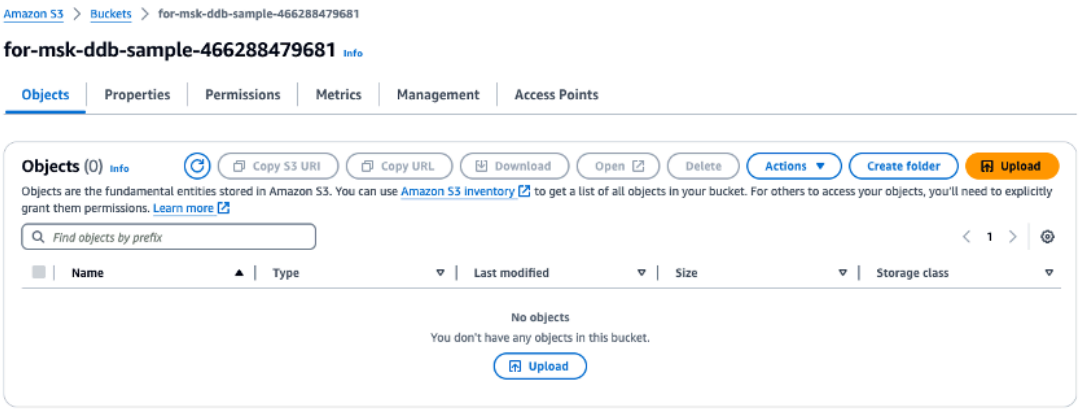
1. CloudFormation テンプレート `S3bucket.yaml` を実行して、S3 バケットを作成します。以降のスクリプトまたはオペレーションについては、同じリージョンで実行してください。CloudFormation スタックの名前として `ForMSKTestS3` を入力します。



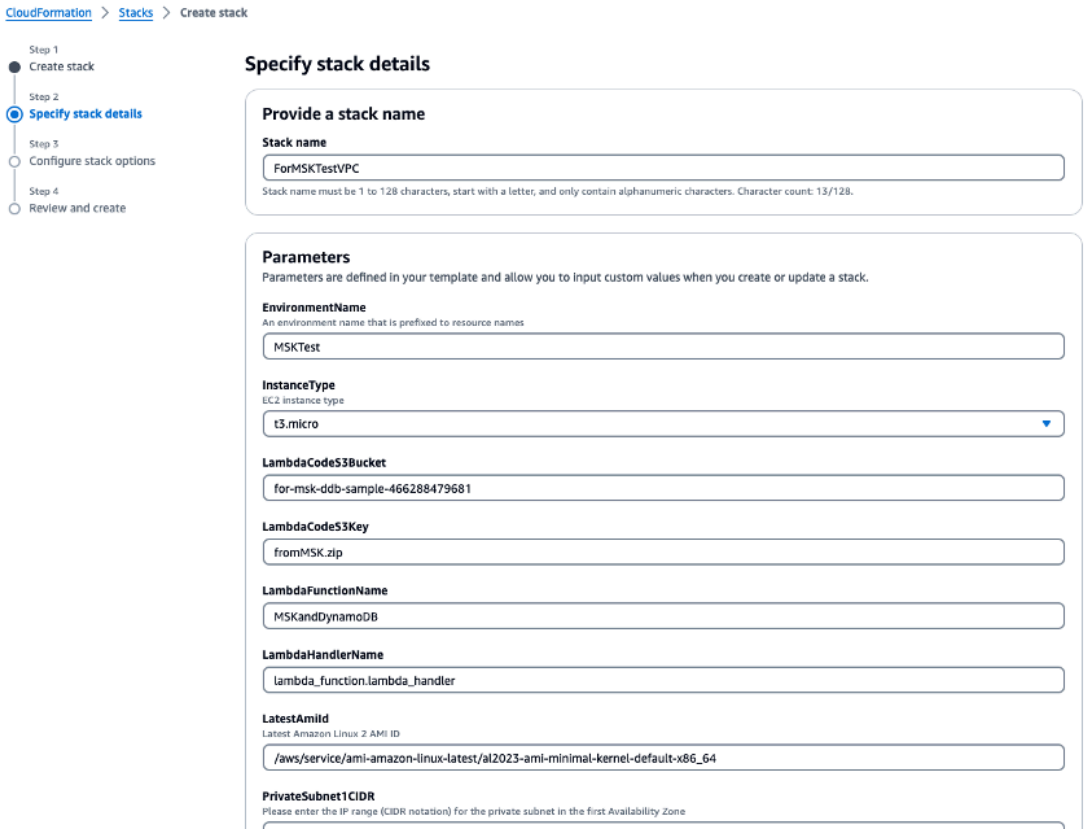
これが完了したら、出力の下にある S3 バケット名出力を書き留めます。ステップ 3 でこの S3 バケット名出力が必要になります。



2. 作成した S3 バケットにダウンロードした ZIP ファイル `fromMSK.zip` をアップロードします。



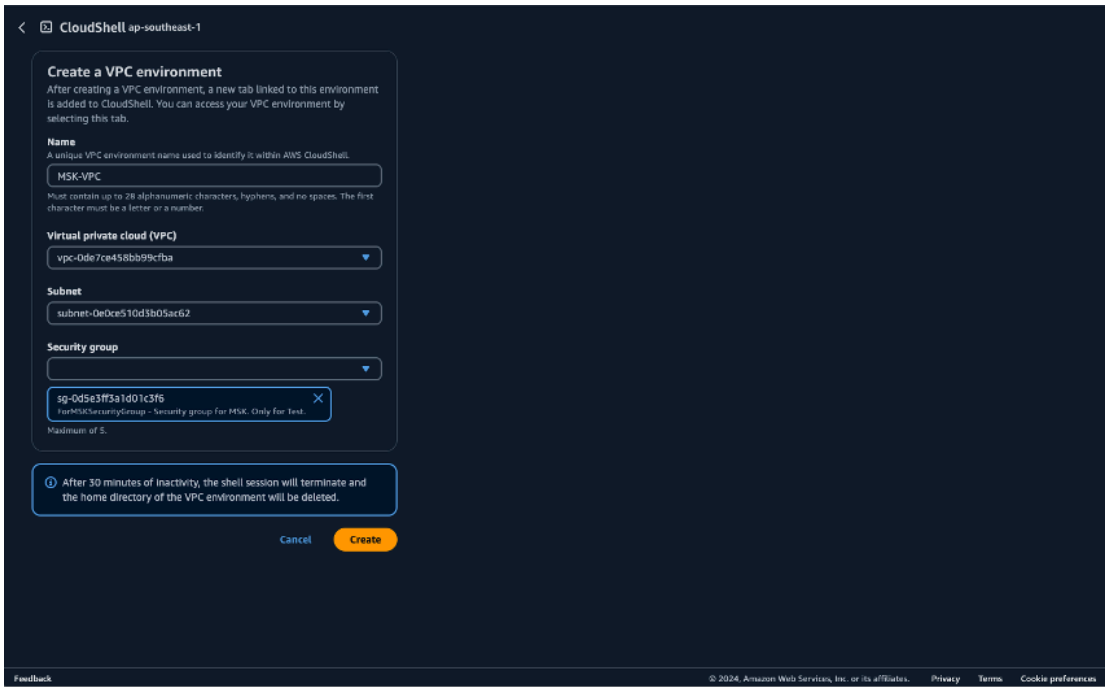
3. CloudFormation のテンプレート `VPC.yaml` を実行して、VPC、Amazon MSK クラスター、Lambda 関数を作成します。パラメータ入力画面で S3 バケットを求められたら、ステップ 1 で作成した S3 バケット名を入力します。CloudFormation スタック名を `ForMSKTestVPC` に設定します。



4. CloudShell で Python スクリプトを実行するための環境を準備します。AWS マネジメントコンソール で CloudShell を使用できます。CloudShell の使用の詳細については、「AWS CloudShell の開始方法」を参照してください。CloudShell を起動したら、Amazon MSK クラスターに接続するために作成した VPC に属する CloudShell を作成します。プライベートサブネットに CloudShell を作成します。以下のフィールドに入力します。

   1. 名前 - 任意の名前に設定できます。例: MSK-VPC
   2. VPC - MSKTest を選択します。
   3. サブネット - MSKTest プライベートサブネット (AZ1) を選択します。
   4. SecurityGroup - ForMSKSecurityGroup を選択します。





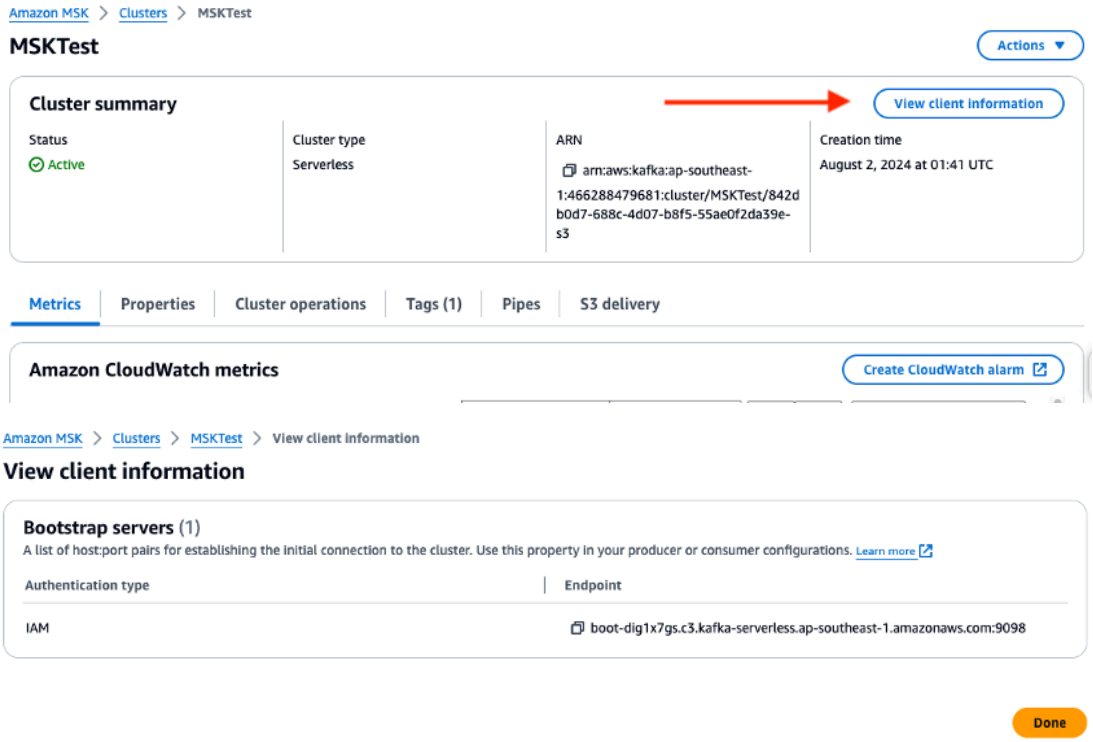
プライベートサブネットに属する CloudShell が開始されたら、次のコマンドを実行します。

```
pip install boto3 kafka-python aws-msk-iam-sasl-signer-python
```

5. S3 バケットから Python スクリプトをダウンロードします。

```
aws s3 cp s3://[YOUR-BUCKET-NAME]/pythonScripts.zip ./
unzip pythonScripts.zip
```

6. マネジメントコンソールを確認し、Python スクリプトでブローカー URL とリージョン値の環境変数を設定します。マネジメントコンソールで Amazon MSK クラスターブローカーエンドポイントを確認します。



7. CloudShell で環境変数を設定します。米国西部 (オレゴン) を使用している場合:

```
export AWS_REGION="us-west-2"
export MSK_BROKER="boot-YOURMSKCLUSTER.c3.kafka-serverless.ap-
southeast-1.amazonaws.com:9098"
```

8. 以下の Python スクリプトを実行します。

   Amazon MSK トピックを作成します。

```
python ./createTopic.py
```

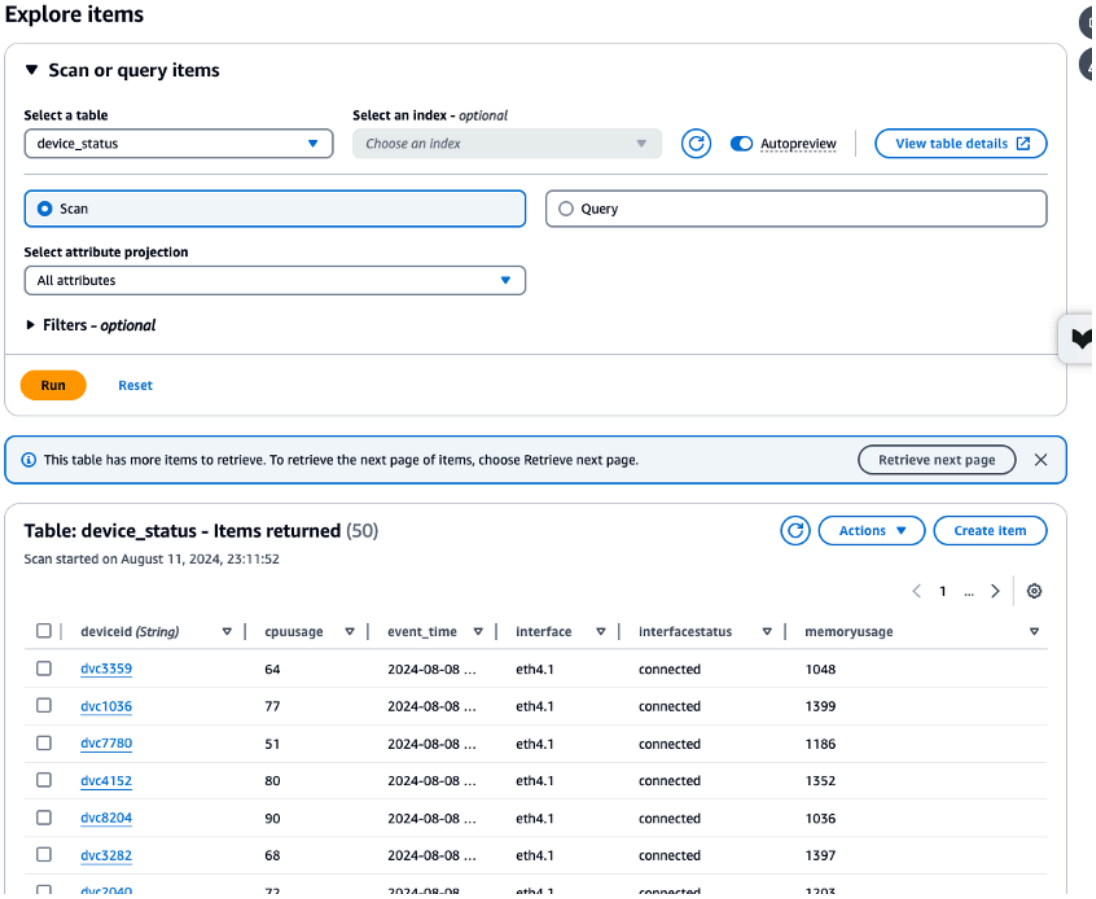
   DynamoDB テーブルを作成します。

```
python ./createTable.py
```

Amazon MSK トピックにテストデータを書き込みます。

```
python ./kafkaDataGen.py
```

9. 作成した Amazon MSK、Lambda、DynamoDB リソースの CloudWatch メトリクスを確認し、DynamoDB Data Explorer を使用して `device_status` テーブルに保存されているデータを検証して、すべてのプロセスが正しく実行されたことを確認します。各プロセスがエラーなしで実行される場合は、CloudShell から Amazon MSK に書き込まれたテストデータも DynamoDB に書き込まれていることを確認できます。



10. この例が完了したら、このチュートリアルで作成したリソースを削除します。`ForMSKTestS3` と `ForMSKTestVPC` の 2 つの CloudFormation スタックを削除します。スタックの削除が正常に完了すると、すべてのリソースが削除されます。

次のステップ

> **Note**
>
> この例に従ってリソースを作成した場合は、予期しない料金が発生しないように、リソースを削除してください。

統合では、Amazon MSK と DynamoDB をリンクして、ストリームデータを OLTP ワークロードをサポートできるようにするアーキテクチャを特定しました。ここから、DynamoDB と Amazon OpenSearch を連携することで、より複雑な検索を実現できます。より複雑なイベント駆動型のニーズには EventBridge との統合を検討し、スループットを高め、レイテンシー要件を低くするには Amazon Managed Service for Apache Flink などの拡張機能を検討してください。

DynamoDB との統合に関するベストプラクティス

DynamoDB を他のサービスと統合する際は、各サービスを使用する際のベストプラクティスに必ず従います。ただし、統合に特有のベストプラクティスもいくつか考慮する必要があります。

トピック

- DynamoDB でスナップショットを作成する
- DynamoDB でデータ変更をキャプチャする

DynamoDB でスナップショットを作成する

- 一般的には、Amazon S3 へのエクスポートを使用して、初期レプリケーションのスナップショットを作成することをお勧めします。この方法は費用対効果が高く、スループットのためにアプリケーションのトラフィックと競合することもありません。また、新しいテーブルへのバックアップと復元の後に、スキャンオペレーションを実行することも検討してください。これにより、アプリケーションとのスループットの競合を避けることができますが、一般的にはエクスポートよりもコスト効率が大きく低下します。
- エクスポートを行う際は、必ず `StartTime` を設定します。これにより、変更データキャプチャ (CDC) をいつ開始するかを簡単に決定できます。
- S3 へのエクスポートを使用する際は、S3 バケットにライフサイクルアクションを設定します。通常、Expiration アクションは 7 日に設定しても問題ありませんが、自社で定められているガイドラ

インに従ってください。取り込み後にアイテムを明示的に削除した場合でも、このアクションは問題を特定するのに役立ち、不要なコストを削減し、ポリシー違反を防ぐことができます。

DynamoDB でデータ変更をキャプチャする

- ほぼリアルタイムの CDC が必要な場合は、DynamoDB Streams または Amazon Kinesis Data Streams (KDS) を使用します。どちらを使用するかを決める際、通常は、どちらがダウンストリームサービスで使いやすいかを検討します。パーティションキーレベルで順番どおりにイベント処理を行う必要がある場合、または非常に大きいアイテムがある場合は、DynamoDB Streams を使用します。
- ほぼリアルタイムの CDC が必要ない場合は、Amazon S3 への増分エクスポートを使用して、2 つの時点の間に発生した変更のみをエクスポートできます。

  S3 へのエクスポートを使用してスナップショットの生成を行った場合は、同様のコードを使用して増分エクスポートを処理できるため、特に便利です。通常、S3 へのエクスポートは以前のストリーミングオプションよりも若干安価ですが、どのオプションを使用するかを決める主な要素はコストではありません。
- 通常、DynamoDB ストリームの同時コンシューマーは、最大 2 人です。統合戦略を計画する際には、この点を考慮してください。
- スキャンを使用して変更を検出しないでください。これは小規模ではうまくいくかもしれませんが、すぐに実用的ではなくなります。

DynamoDB でのエージェンティック AI の使用

Amazon DynamoDB は、サーバーレスのフルマネージド分散 NoSQL データベースで、あらゆる規模で 1 桁ミリ秒のパフォーマンスを実現します。DynamoDB はハイスループットワークロード用に最適化されており、生成 AI モデルと統合することで機能を拡張できます。生成 AI モデルを使用すると、DynamoDB テーブルに保存されているデータをリアルタイムで操作し、コンテキストを認識して高度にパーソナライズされたアプリケーションを構築できます。また、ビジネス、ユーザー、アプリケーションデータを最大限に活用して生成 AI ソリューションをカスタマイズすることで、エンドユーザーエクスペリエンスを向上させることもできます。

生成 AI と、生成 AI アプリケーションを構築するために AWS が提供するソリューションの詳細については、「生成 AI でビジネスを変革する」を参照してください。

DynamoDB はベクトルインデックスを介したネイティブベクトル検索も提供するため、運用データとともにベクトル埋め込みを保存し、個別のベクトルデータベースなしで `SearchVectors` API を使用して類似度検索を実行できます。詳細については、「DynamoDB でのベクトルインデックスの使用」を参照してください。

トピック

- DynamoDB の生成 AI ユースケース
- RAG とエージェントメモリのベクトル検索
- DynamoDB 用の AI フレームワークコネクタ
- DynamoDB の生成 AI ブログ
- OpenSearch Service と DynamoDB のゼロ ETL 統合を活用する

DynamoDB の生成 AI ユースケース

DynamoDB は、チャットボットや基盤モデル (FM) で構築されたコールセンターなど、AI を活用した会話アプリケーションに広く使用されています。Amazon Bedrock、Amazon SageMaker AI、またはその他のモデルプロバイダーから FM にアクセスできます。このようなアプリケーションは、通常、DynamoDB を使用して、パーソナライゼーションを向上させ、アプリケーションデータ、ビジネスデータ、ユーザーデータの 3 つのデータパターンにわたってユーザーエクスペリエンスを向上させます。これらのデータパターンの例は次のとおりです。

- LangChain、 LlamaIndex、またはカスタムコードとの統合による、チャットメッセージ履歴などのアプリケーションデータのストレージ。このコンテキストは、モデルがユーザーと対話できる

ようにすることで、ユーザーエクスペリエンスを向上させます。エージェンティック AI フレームワークとの DynamoDB 統合の詳細については、「DynamoDB 用の AI フレームワークコネクタ」を参照してください。

- インベントリ、料金、ドキュメントなどのビジネスデータを活用して、カスタマイズされたユーザーエクスペリエンスを作成します。
- ウェブ履歴、過去の注文、ユーザー設定などのユーザーデータを適用して、パーソナライズされた回答を提供します。

例えば、保険会社は DynamoDB を使用して、チャットボットを構築し、検索拡張検索拡張生成 (RAG) ベースの生成 AI モデルにほぼリアルタイムのデータへのアクセスを提供することができます。このようなデータの例としては、住宅ローンの金利、商品の価格設定、準拠/標準契約書、ユーザーのウェブ履歴、ユーザーの好みなどのリアルタイムデータが挙げられます。DynamoDB を RAG と組み合わせることで、保険商品やユーザーデータに関する詳細かつ最新の情報が追加されます。これにより、プロンプトと回答が充実し、エンドユーザーに正確でパーソナライズされたほぼリアルタイムのエクスペリエンスを提供できます。

同様に、金融サービス業界のお客様は、DynamoDB、Amazon Bedrock ナレッジベース、Amazon Bedrock エージェントを使用して、RAG ベースの生成 AI アプリケーションを構築しています。これらのアプリケーションは、オープンソースの収益レポートと通話トランスクリプトを使用できます。また、ユーザー固有のポートフォリオとトランザクション履歴を使用して、将来の見通しを含むポートフォリオのオンデマンドの概要を生成することもできます。

RAG とエージェントメモリのベクトル検索

生成 AI アプリケーションは意味によってコンテキストを取得します。RAG アプリケーションはプロンプトに最も関連性の高いドキュメントを見つけ、AI エージェントは現在の会話に類似したメモリを呼び出します。ベクトルインデックスにより、この機能が DynamoDB にネイティブに提供されます。ベクトル埋め込みをテーブル項目の属性として保存すると、`SearchVectors` API は類似度スコアでランク付けされた、クエリベクトルに最も類似した項目を返します。アプリケーションが書き込むデータは検索対象のデータであるため、プロビジョニングが必要な個別のベクトルデータベースや、管理が必要なレプリケーションパイプラインは存在しません。

ベクトルインデックスの検索スキーマでパーティションキーを定義することもできます。各類似度検索は、テナント ID、ユーザー ID、エージェントセッション ID などの単一のパーティションキー値にスコープされます。これにより、ベクトルインデックスはマルチテナント RAG とユーザーごとのエージェントメモリに自然に適合します。

ベクトルインデックスの使用を開始するには、以下のトピックを参照してください。

- DynamoDB でのベクトルインデックスの使用
- チュートリアル: 初めてのベクトル検索
- ベクトルインデックスの作成と検索
- ベクトルインデックスのベストプラクティス

DynamoDB 用の AI フレームワークコネクタ

AI エージェントには、再起動後も維持される会話状態、セッションをまたぐチャット履歴、後の会話で参照できる長期メモリなど、記憶しておく必要があるすべての情報を格納するための耐久性のあるストレージが必要です。AWS およびフレームワークコミュニティによってメンテナンスされるオープンソースコネクタは、DynamoDB を主要な AI エージェントフレームワークと統合します。これにより、状態は 1 桁ミリ秒のレイテンシーを持つサーバーレスデータベースに、お客様自身の AWS アカウント内で保持されます。

次の表は、利用可能なコネクタと各コネクタが保存する情報をまとめたものです。

フレームワーク	パッケージ	DynamoDB に保存される情報	言語
Strands Agents	`strands-dynamodb-storage`	SDK の統合ストレージコントラクトによるセッション状態、長期メモリ、トランスクリプト、およびサイズの大きいツール結果	Python、TypeScript
LangChain	`langchain-community` , `langchain-aws`	チャットメッセージ履歴、およびベクトル類似度検索用の埋め込みを含むドキュメント	Python、JavaScript
LangGraph	`langgraph-checkpoint-aws`	エージェントチェックポイント (短期的な	Python

フレームワーク	パッケージ	DynamoDB に保存される情報	言語
		会話状態) と長期的なクロススレッドメモリ	

トピック

- Strands Agents 用ストレージバックエンドとしての DynamoDB の使用
- DynamoDB と LangChain の使用
- DynamoDB を LangGraph エージェントのチェックポイントストアとして使用する
- LangGraph エージェントのセマンティック長期メモリ

Strands Agents 用ストレージバックエンドとしての DynamoDB の使用

Strands Agents はオープンソース SDK で、数行のコードで AI エージェントを構築するモデル駆動型のアプローチを採用しており、Amazon Bedrock やその他のモデルプロバイダーをファーストクラスでサポートしています。その構成要素の 1 つに、統合ストレージインターフェイスがあります。これは、SDK 内のすべてのステートフルサブシステムが使用するバイト指向のコントラクト (`write`、`read`、`delete`、`list`) です。セッションマネージャーはこのインターフェイスを通じて会話スナップショットを永続化し、メモリマネージャーは長期メモリを保存します。コンテキストオフローダーとトランスクリプトも同じオペレーションを使用します。

strands-dynamodb-storage パッケージは、Python と TypeScript の両方で、単一の DynamoDB テーブル上にそのコントラクトを実装します。SDK の / 区切りストレージキーは DynamoDB キーモデルに直接マッピングされます。キーの最初の 2 つのセグメントがパーティションキーになり、残りがソートキーになるため、ポイントオペレーションは単一項目の呼び出しとなり、プレフィックスのリスト取得はネイティブパーティション `Query` で行われ、テーブルスキャンは不要です。

主な特徴

すべてのエージェント状態を 1 つのテーブルで管理

1 つの `DynamoDBStorage` インスタンスが、セッション永続性、長期メモリ、トランスクリプト、コンテキストオフロードをサポートします。それぞれ独自のキープレフィックスの下に名前空間が設定されています。

セマンティックメモリ検索

テーブルにベクトルインデックスを設定すると、メモリを埋め込み表現で書き込み、`SearchVectors` API を通じて意味に基づいて呼び出すことができます。「ベクトルインデックスを使用したセマンティック長期メモリ」を参照してください。

大きな値のための Amazon S3 オフロード

オフロードはオプトインです。コンストラクタにバケット名を渡すと、400 KB の DynamoDB 項目サイズ制限を超える値は透過的に Amazon Simple Storage Service にオフロードされ、テーブルには小さなポインタ項目が残ります。

オプションの圧縮と有効期限

オプションの gzip 圧縮を使用すると、圧縮可能な値をより低いコストでインラインに保持できます。オプションの有効期限は、DynamoDB ネイティブの有効期限属性をスタンプし、既に有効期限が経過した項目は読み取りおよび一覧表示でフィルタリングされます。

マルチテナントプレフィックス

コンストラクタにバインドされたキープレフィックスは、すべてのオペレーションを独自のキースペース内に固定します。そのため、テーブルを共有する 2 つのテナントは、同じ論理キーを物理的に異なるパーティションに解決します。

前提条件

- DynamoDB テーブルを作成するアクセス許可を持つ AWS アカウント (大きな値のオフロードを設定する場合は、オプションで Amazon S3 バケット)
- Python 3.10 以降 (`strands-agents` 1.48.0 以降)、または Node.js 20 以降 (`@strands-agents/sdk` 1.10.0 以降)
- AWS 認証情報の設定 (認証情報の設定オプションについては、AWS ドキュメントを参照してください)
- セマンティックメモリの場合: Amazon Bedrock の Amazon Titan Text Embeddings V2 などの埋め込みモデルへのアクセス、およびベクトルインデックスで作成されたテーブル (「DynamoDB でのベクトルインデックスの使用」を参照)

インストール

PyPI からパッケージをインストールします。

```
pip install strands-dynamodb-storage "strands-agents>=1.48.0"
```

または TypeScript の場合は npm からインストールします (TypeScript パッケージは機能的に同等のミラーです)。

```
npm install strands-dynamodb-storage
```

テーブルを作成する

パッケージには CreateTable アクセス許可がなく、インフラストラクチャも作成されません。事前にテーブルを作成し、独自のタグ付け、バックアップ、暗号化設定を適用します。必要なのは、文字列パーティションキー pk と文字列ソートキー sk を持つテーブルのみです。

```
aws dynamodb create-table \
  --table-name agent-storage \
  --attribute-definitions AttributeName=pk,AttributeType=S
 AttributeName=sk,AttributeType=S \
  --key-schema AttributeName=pk,KeyType=HASH AttributeName=sk,KeyType=RANGE \
  --billing-mode PAY_PER_REQUEST
```

セマンティックメモリを使用する場合は、同じコマンドでベクトルインデックスを宣言します。インデックスの名前、ディメンション、距離関数は作成後に変更できないため、埋め込みモデルに合わせてディメンションを設定してください。パッケージ README には、インデックスを宣言する完全な呼び出しが示されています。

エージェントセッションを永続化する

ストレージをエージェントのセッションマネージャーに渡します。SDK はキーを名前空間で管理し、呼び出しのたびに会話のスナップショットを作成し、同じセッションが再開されると復元します。

```
from strands import Agent
from strands.session import SnapshotSessionManager
from strands_dynamodb_storage import DynamoDBStorage

storage = DynamoDBStorage("agent-storage", region_name="us-east-1")
session = SnapshotSessionManager(session_id="user-42", storage=storage)

agent = Agent(session_manager=session)
agent("Where did we leave off?")
```

Agent 自体にストレージを 1 回設定することもできます。ストレージを受け入れるすべてのサブシステムはそれを継承し、それぞれが独自のキープレフィックスの下に名前空間化されるため、1 つのテーブルでエージェントの状態全体を管理できます。

```
from strands.vended_plugins.context_offloader import ContextOffloader

agent = Agent(
    storage=storage,
    session_manager=SnapshotSessionManager(),  # persists under session/
    plugins=[ContextOffloader()],              # offloads oversized tool results under
 offloader/
)
```

バイトコントラクトは直接利用することもできます。コントラクトは非同期です。エージェント内では SDK によって自動的に実行され、プレーンスクリプトでは asyncio.run で呼び出しをラップします。

```
import asyncio


async def main():
    await storage.write("session/user-42/notes", b"prefers aisle seats")
    keys = await storage.list("session/user-42/")


asyncio.run(main())
```

ベクトルインデックスを使用したセマンティック長期メモリ

セッション永続化でメモリ問題の半分が解決します。エージェントは再起動後も存続し、会話を再開できます。問題の残りの半分は、ユーザーが数週間前にエージェントに伝えた内容を、以前の会話とキーを共有しない新しい会話の中で思い出すことです。そのためには、キーではなく意味に基づいてメモリを検索する必要があります。DynamoDB ベクトルインデックスを使用すると、エージェントの状態を保持する同じテーブルに対して最近傍検索が可能になります (「DynamoDB でのベクトルインデックスの使用」を参照)。

次の例では、Amazon Bedrock を介して Amazon Titan Text Embeddings V2 でテキストを埋め込みます。モデルはデフォルトで 1,024 次元のベクトルを返すため、これらの例のインデックスは 1,024 次元で作成されます。次のコードは、残りの例で使用する embed( ) 関数を定義します。

```
import json
```

```
import boto3

bedrock = boto3.client("bedrock-runtime", region_name="us-east-1")


def embed(text):
    response = bedrock.invoke_model(
        modelId="amazon.titan-embed-text-v2:0",
        body=json.dumps({"inputText": text, "dimensions": 1024}),
    )
    return json.loads(response["body"].read())["embedding"]
```

メモリを書き込むと、埋め込みとオプションのメタデータがバイトデータとともにアタッチされます。次のコードは、テナントごとのプレフィックスを使用してストアを構築するため、キー memories/m1 は物理パーティション user/u1 に配置されます。

```
from strands_dynamodb_storage import DynamoDBStorage, SearchQuery

storage = DynamoDBStorage("agent-storage", region_name="us-east-1", prefix="user/u1")

await storage.write(
    "memories/m1",
    b"prefers window seats on long flights",
    vector=embed("prefers window seats on long flights"),
    metadata={"kind": "preference"},
)
```

意味に基づく呼び出しは、同じパーティションにスコープされた 1 回の呼び出しです。

```
results = await storage.search(SearchQuery(
    vector=embed("what are this user's seating preferences?"),
    top_k=5,
    pk="user/u1",  # the physical partition: the full key's first two segments
    filter={"kind": "preference"},
))
```

pk 引数に注目してください。ベクトルインデックスはテーブルと同じ方法でパーティション分割され、すべての検索は 1 つのパーティションにスコープされるため、あるテナントの検索が別のテナントのメモリを参照することはありません。各検索が実行する処理は、テーブル全体ではなく、そのテナントのメモリのサイズを追跡します。パーティション値は呼び出し元によって指定されるため、

これは認可境界ではなくクエリスコープです。テーブルに対して dynamodb:SearchVectors を保持するプリンシパルは任意のパーティションを検索できるため、テナントのアクセスコントロールは IAM とアプリケーションレイヤーで管理する必要があります。

結果は類似度の高い順に返されます。raw スコアの方向はインデックスの距離関数に従います。コサイン距離およびユークリッド距離では値が低いほど近く、ドット積では値が高いほど類似度が高くなります。

エージェントへのメモリの接続

実際のエージェントでは、取得したメモリがモデルに自動的に渡されるようにする必要があります。SDK のメモリマネージャーがこれを処理します。各モデル呼び出しの前に関連するエントリを取得してモデル入力に組み込み、モデルがオンデマンドで呼び出せる search_memory ツールを登録します。メモリマネージャーは、SDK の MemoryStore プロトコルを実装するすべてのオブジェクトを受け入れます。パッケージにはメモリマネージャーが付属していないため、書き込み時に埋め込み、検索時に埋め込み、MemoryEntry 値を返す小さなクラスを独自のアプリケーションに定義します。

```
import uuid

from strands.memory import MemoryEntry
from strands_dynamodb_storage import DynamoDBStorage, SearchQuery


class DynamoDBMemoryStore:
    def __init__(self, storage, partition, embed):
        self.storage = storage
        self.partition = partition
        self.embed = embed  # the embed() function defined earlier
        self.name = "dynamodb"
        self.description = "Long-term memories in DynamoDB, searched by meaning"
        self.max_search_results = 3
        self.writable = True
        self.extraction = None

    async def add(self, content, metadata=None):
        await self.storage.write(
            f"memories/{uuid.uuid4().hex[:8]}",
            content.encode(),
            vector=self.embed(content),
            metadata=metadata,
        )
```

```
    async def search(self, query, options=None):
        results = await self.storage.search(SearchQuery(
            vector=self.embed(query),
            top_k=self.max_search_results,
            pk=self.partition,
            include_values=True,
        ))
        return [
            MemoryEntry(content=r.data.decode(), metadata=r.metadata)
            for r in results
            if r.data is not None
        ]
```

次のコードは 3 つのメモリをシードし、ストアをエージェントに接族します。これにより、取得したメモリはオーケストレーションコードなしでモデルに到達します。

```
import asyncio

from strands import Agent
from strands.memory import MemoryManager

storage = DynamoDBStorage("agent-storage", region_name="us-east-1", prefix="user/u1")
store = DynamoDBMemoryStore(storage, partition="user/u1", embed=embed)


async def seed():
    await store.add("Prefers window seats on long flights")
    await store.add("Planning a trip to Tokyo in December")
    await store.add("Allergic to peanuts")


asyncio.run(seed())

memory = MemoryManager(stores=[store], add_tool_config=True)
agent = Agent(memory_manager=memory)
agent("Book me a flight seat for my December trip. Which seat should I pick?")
```

これをライブテーブルに対して実行すると、エージェントは search_memory ツールを呼び出し、DynamoDB から東京への旅行と窓側座席の希望を照合して、フライトの窓側座席を推奨します。add_tool_config=True を渡すと add_memory ツールも登録されるため、モデルは呼び出し元と同じテーブルを通じて新しい事実を保存できます。

必要な IAM 許可

実行時に、パッケージは 4 つの DynamoDB オペレーションを発行し、セマンティック検索を使用する場合は `SearchVectors` をさらに実行します。また、オフロードを設定する場合にのみ Amazon S3 オペレーションを発行するため、最小特権の IAM ポリシーは短くなります。*111122223333* を AWS アカウント ID に置き換え、環境に合わせてリージョンを更新してください。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DeleteItem",
        "dynamodb:Query"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/agent-storage"
    }
  ]
}
```

パッケージ README には、セマンティック検索、Amazon S3 オフロード、埋め込みモデルの呼び出しに関する追加ステートメントを含む、完全なポリシーが記載されています。

考慮事項

- ベクトルインデックスの名前、ディメンション、距離関数は作成後に変更できません。また、新しく作成されたインデックスは検索可能になる前にバックフィル処理されます。テーブルは最大 5 つのベクトルインデックスをサポートしているため、後で別の設定を採用する場合は、テーブルを再構築するのではなく、インデックスを追加することになります。
- ベクトルインデックスには結果整合性があり、グローバルセカンダリインデックスと同じモデルです。直前に書き込まれたメモリは、検索可能になるまでに少し時間がかかる場合があります。
- 有効期限による有効期限フィルタリングは、読み取りとリストオペレーションに適用されます。有効期限による削除は非同期であるため、有効期限が過ぎていても DynamoDB がまだ物理的に削除していない項目が検索結果に一時的に返されることがあります。

- リスト表示には、少なくともスコープ全体と識別子をカバーするプレフィックスが必要です。空のプレフィックスなど、範囲の広いリスト表示は拒否されます。パッケージがテーブルスキャンにフォールバックすることはありません。
- オフロードされた値で有効期限を有効にする場合は、Amazon S3 ライフサイクルルールを追加してください。DynamoDB は期限切れのポインタ項目を削除し、ライフサイクルルールは Amazon S3 オブジェクトを再利用します。

その他のリソース

- GitHub の strands-dynamodb-storage
- PyPI の strands-dynamodb-storage
- npm の strands-dynamodb-storage
- Strands Agents ドキュメント

DynamoDB と LangChain の使用

LangChain は、大規模言語モデル (LLM) を使用してアプリケーションを構築するためのフレームワークです。DynamoDB を使用する LangChain 統合は 2 つあります。`DynamoDBChatMessageHistory` クラスは会話履歴を保存してモデルがセッション間でユーザーとやり取りできるようにし、`DynamoDBVectorStore` クラスは類似性検索のために DynamoDB ベクトルインデックスに LangChain ベクトルストアインターフェイスを実装します。

チャットメッセージ履歴

`DynamoDBChatMessageHistory` クラスは、Python 用の `langchain-community` パッケージと JavaScript 用の `@langchain/community` パッケージで、チャットメッセージを DynamoDB テーブルに保持します。このクラスは、パーティションキーが `SessionId` という名前の文字列属性 (`primary_key_name` を使用して設定可能) である既存のテーブルを前提としています。AWS CLI を使用してテーブルを作成します。

```
aws dynamodb create-table \
  --table-name SessionTable \
  --attribute-definitions AttributeName=SessionId,AttributeType=S \
  --key-schema AttributeName=SessionId,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST
```

パッケージをインストールし、セッション ID をキーとする履歴の読み書きを行います。

```
pip install langchain-community boto3
```

```
from langchain_community.chat_message_histories import DynamoDBChatMessageHistory

history = DynamoDBChatMessageHistory(
    table_name="SessionTable",
    session_id="user-42",
)

history.add_user_message("Hello!")
history.add_ai_message("How can I help you today?")

print(history.messages)
```

各チャットセッションのメッセージは session_id の下に保存されるため、戻ってきたユーザーは中断した会話を再開できます。コンストラクタは、ユーザー ID (key)、古いセッションの TTL ベースの有効期限 (ttl)、保存メッセージの上限 (history_size) など、アプリケーションの詳細別に履歴を分離するための複合キーもサポートしています。完全な API については、「Python リファレンス」および「JavaScript リファレンス」を参照してください。

ベクトルインデックスに基づくベクトルストア

langchain-aws パッケージの DynamoDBVectorStore クラスは、DynamoDB ベクトルインデックスを使用して LangChain ベクトルストアインターフェイスを実装します (「DynamoDB でのベクトルインデックスの使用」を参照)。ドキュメントは通常の DynamoDB 項目として保存され、SearchVectors API を使用してテーブルのベクトルインデックスが検索されるため、LangChain リトリーバーと RAG チェーンは、データを保持するテーブルに対して類似度検索を直接実行します。

パッケージをインストールする:

```
pip install langchain-aws boto3
```

テーブル名と埋め込み関数を指定します。テーブルとベクトルインデックスが存在しない場合、最初の書き込み時に作成されます。

```
from langchain_aws.embeddings import BedrockEmbeddings
from langchain_aws.vectorstores.dynamodb import DynamoDBVectorStore
```

```
vector_store = DynamoDBVectorStore.from_texts(
    ["hello", "developer", "wife"],
    embedding=BedrockEmbeddings(model_id="amazon.titan-embed-text-v2:0"),
    table_name="my-documents",
)

docs = vector_store.similarity_search("greeting", k=2)
```

検索の範囲を指定するには、ストアを構築するときに partition_attribute を渡します。次に、ベクトルインデックスはそのドキュメントメタデータフィールドの検索スキーマパーティションキーを使用して作成され、各検索はコーパス全体ではなく、コレクション、カテゴリ、テナントなどの値を 1 つだけ調べます。その後、すべての検索で、filter={partition_attribute: value} またはストアレベルの default_partition_value を通じて値が提供されます。パーティションキーはインデックススキーマの一部であり、インデックスの作成後に変更することはできません。

以下に留意してください。

- グローバルセカンダリインデックスと同様に、ベクトルインデックスは結果整合性があります。add_texts の直後に発行された検索には、インデックスが追いつくまで、直前に書き込まれたドキュメントが含まれない場合があります。
- SearchVectors はクエリごとに最大 100 件の一致を返すため、ストアの k パラメータは 100 に制限されます。
- インデックスのディメンション、距離関数、および検索スキーマは、作成時に固定されます。ストアは、不一致のまま処理を続行するのではなく、既存のインデックスに対して検証を行います。

API の詳細については、「langchain-aws リポジトリの DynamoDBVectorStore」を参照してください。

その他のリソース

- DynamoDBChatMessageHistory Python リファレンス
- DynamoDBChatMessageHistory JavaScript リファレンス
- GitHub の langchain-aws
- Amazon DynamoDB、Amazon Bedrock、LangChain を使用して、スケーラブルなコンテキスト対応チャットボットを構築する

DynamoDB を LangGraph エージェントのチェックポイントストアとして使用する

LangGraph は、大規模言語モデル (LLM) を用いて、ステートフルなマルチアクター AI アプリケーションを構築するためのフレームワークです。LangGraph エージェントには、会話状態を維持し、ヒューマンインザループワークフローを有効にし、耐障害性をサポートし、タイムトラベルデバッグ機能を提供するために永続的ストレージが必要です。DynamoDB のサーバーレスアーキテクチャ、1 桁ミリ秒のレイテンシー、自動スケーリングは、AWS での本番稼働用 LangGraph デプロイのチェックポイントストアとして理想的です。

`langgraph-checkpoint-aws` パッケージには、LangGraph チェックポイントインターフェイスを実装する `DynamoDBSaver` クラスを提供し、エージェントの状態を DynamoDB に永続化できます。また、大規模なチェックポイントの場合は、オプションで Amazon Simple Storage Service へのオフロードも可能です。

同じパッケージには、DynamoDB ベクトルインデックスに基づくセマンティック検索を使用して、長期のクロススレッドエージェントメモリ用の LangGraph ストアインターフェイスを実装する `DynamoDBStore` クラスも用意されています。詳細については、「LangGraph エージェントのセマンティック長期メモリ」を参照してください。このトピックでは、チェックポイントセーバーについて説明します。

主な特徴

状態永続性

各ステップの後にエージェントの状態を自動的に保存し、エージェントが中断から再開したり、障害から回復したりできるようにします。

有効期限に基づくクリーンアップ

DynamoDB の有効期限を使用して古いチェックポイントを自動的に期限切れにし、ストレージコストを管理します。

圧縮

必要に応じて、gzip でチェックポイントデータを圧縮してストレージコストを削減し、スループットを向上させます。

Amazon S3 へのオフロード

大きなチェックポイント (350 KB 以上) を Amazon Simple Storage Service に自動的にオフロードして、DynamoDB 項目のサイズ制限内で機能させます。

同期と非同期のサポート

同期 API と非同期 API を使用して、さまざまなアプリケーションアーキテクチャに柔軟に対応できます。

前提条件

- Python 3.10 以降
- DynamoDB テーブル (およびオプションで Amazon S3 バケット) を作成するアクセス許可を持つ AWS アカウント。
- AWS 認証情報の設定 (認証情報の設定オプションについては、AWS ドキュメントを参照してください)

インストール

PyPI からチェックポイントパッケージをインストールします。

```
pip install langgraph-checkpoint-aws
```

基本的な使用法

次の例は、DynamoDB を LangGraph エージェントのチェックポイントストアとして設定する方法を示しています。

```
from langgraph.graph import StateGraph
from langgraph_checkpoint_aws import DynamoDBSaver
from typing import TypedDict

# Define your state schema
class State(TypedDict):
    input: str
    result: str

# Initialize the DynamoDB checkpoint saver
checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
    region_name="us-east-1"
)
```

```
# Build your LangGraph workflow
builder = StateGraph(State)
builder.add_node("process", lambda state: {"result": "processed"})
builder.set_entry_point("process")
builder.set_finish_point("process")

# Compile the graph with the DynamoDB checkpointer
graph = builder.compile(checkpointer=checkpointer)

# Invoke the graph with a thread ID to enable state persistence
config = {"configurable": {"thread_id": "session-123"}}
result = graph.invoke({"input": "data"}, config)
```

設定の thread_id は DynamoDB のパーティションキーとして機能し、個別の会話スレッドを維持し、任意のスレッドの履歴状態を取得できます。

本番稼働設定

本番デプロイでは、有効期限、圧縮、Amazon S3 オフロードを有効にできます。endpoint_url パラメータを使用して、テスト用のローカル DynamoDB インスタンスを指定することもできます。

```
import boto3
from botocore.config import Config
from langgraph_checkpoint_aws import DynamoDBSaver

# Production configuration
session = boto3.Session(
    profile_name="production",
    region_name="us-east-1"
)

checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
    session=session,
    ttl_seconds=86400 * 7,           # Expire checkpoints after 7 days
    enable_checkpoint_compression=True,  # Enable gzip compression
    boto_config=Config(
        retries={"mode": "adaptive", "max_attempts": 6},
        max_pool_connections=50
    ),
    s3_offload_config={
        "bucket_name": "my-checkpoint-bucket"
    }
```

```
)

# Local testing with DynamoDB Local
local_checkpointer = DynamoDBSaver(
    table_name="langgraph-checkpoints",
    region_name="us-east-1",
    endpoint_url="http://localhost:8000"
)
```

DynamoDB テーブル設定

チェックポイントセーバーには、複合プライマリキーを持つ DynamoDB テーブルが必要です。次の AWS CloudFormation テンプレートを使用して、テーブルを作成できます。

```
AWSTemplateFormatVersion: '2010-09-09'
Description: 'DynamoDB table for LangGraph checkpoint storage'

Parameters:
  TableName:
    Type: String
    Default: langgraph-checkpoints

Resources:
  CheckpointTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: !Ref TableName
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: PK
          AttributeType: S
        - AttributeName: SK
          AttributeType: S
      KeySchema:
        - AttributeName: PK
          KeyType: HASH
        - AttributeName: SK
          KeyType: RANGE
      TimeToLiveSpecification:
        AttributeName: ttl
        Enabled: true
```

```
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true
      SSESpecification:
        SSEEnabled: true
```

AWS CLI を使用してテンプレートをデプロイします。

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name langgraph-checkpoint \
  --parameter-overrides TableName=langgraph-checkpoints
```

必要な IAM 許可

次の IAM ポリシーは、DynamoDB チェックポイントセーバーに必要な最小限のアクセス許可を提供します。*111122223333* を AWS アカウント ID に置き換え、環境に合わせてリージョンを更新します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ],
      "Resource": "arn:aws:dynamodb:us-east-1:111122223333:table/langgraph-checkpoints"
    }
  ]
}
```

Amazon S3 へのオフロードを有効にする場合は、次のステートメントをポリシーに追加します。

```
{
  "Effect": "Allow",
  "Action": [
    "s3:PutObject",
```

```
    "s3:GetObject",
    "s3:DeleteObject",
    "s3:PutObjectTagging"
  ],
  "Resource": "arn:aws:s3:::my-checkpoint-bucket/*"
},
{
  "Effect": "Allow",
  "Action": [
    "s3:GetBucketLifecycleConfiguration",
    "s3:PutBucketLifecycleConfiguration"
  ],
  "Resource": "arn:aws:s3:::my-checkpoint-bucket"
}
```

非同期使用

非同期アプリケーションの場合は、チェックポイントセーバーが提供する非同期メソッドを使用します。

```
import asyncio
from langgraph.graph import StateGraph
from langgraph_checkpoint_aws import DynamoDBSaver
from typing import TypedDict

class State(TypedDict):
    input: str
    result: str

async def main():
    checkpointer = DynamoDBSaver(
        table_name="langgraph-checkpoints",
        region_name="us-east-1"
    )
    builder = StateGraph(State)
    builder.add_node("process", lambda state: {"result": "processed"})
    builder.set_entry_point("process")
    builder.set_finish_point("process")
    graph = builder.compile(checkpointer=checkpointer)

    config = {"configurable": {"thread_id": "async-session-123"}}
    result = await graph.ainvoke({"input": "data"}, config)
    return result
```

```
asyncio.run(main())
```

エラー処理

一般的なエラーシナリオ。

- テーブルが見つかりません: `table_name` と `region_name` が DynamoDB テーブルと一致していることを確認してください。
- スロットリング: `ProvisionedThroughputExceededException` が表示された場合は、オンデマンド請求モードに切り替えるか、プロビジョンドキャパシティを増やすことを検討してください。
- 項目サイズ超過: チェックポイントが 350 KB を超える場合は、Amazon S3 へのオフロードを有効にします (「本番稼働設定」を参照)。
- 認証情報エラー: AWS 認証情報が有効で、必要なアクセス許可があることを確認します。

その他のリソース

- PyPI の langgraph-checkpoint-aws
- GitHub の langgraph-checkpoint-aws
- LangGraph ドキュメント
- DynamoDB のベストプラクティス
- LangGraph と Amazon DynamoDB を使用して耐久性の高い AI エージェントを構築する

LangGraph エージェントのセマンティック長期メモリ

LangGraph は 2 種類のエージェント状態を分離します。短期的な状態は会話スレッド自体であり、チェックポインターがスレッドを再開、再生、復旧できるように保持します (「DynamoDB を LangGraph エージェントのチェックポイントストアとして使用する」を参照)。長期メモリはエージェントがスレッド間で保持する知識であり、LangGraph はそれをストアとしてモデル化します。ストアとは、エージェントが意図的に書き込み、後で読み返すための名前空間付きキーバリューサーフェスです。

langgraph-checkpoint-aws パッケージの `DynamoDBStore` クラスは、そのストアの DynamoDB 実装です。階層的な名前空間、古くなったメモリを期限切れにする 有効期限 (TTL)、および基本的なフィルタリングを使用して、キーと値の側を処理します。ベクトルインデックスが設定されている

場合 (「DynamoDB でのベクトルインデックスの使用」を参照)、その search() メソッドはセマンティック検索を実行します。メモリは書き込み時に埋め込まれ、非同期的にインデックス化され、同じテーブルから意味に基づいて類似度順に呼び出されます。プロビジョニングが必要な個別のベクトルデータベースも、データをコピーするパイプラインも不要です。

前提条件

- DynamoDB テーブルを作成し、Amazon Bedrock モデルを呼び出すアクセス許可を持つ AWS アカウント。
- お使いのリージョンの Amazon Bedrock の埋め込みモデルへのアクセス。例としては、Amazon Titan Text Embeddings V2 などがあります。
- Python 3.10 以降、langgraph-checkpoint-aws 1.2.2 以降および boto3 1.43.64 以降 (以前の boto3 バージョンには SearchVectors オペレーションがありません)

pip を使用してライブラリをインストールします。

```
pip install langgraph langgraph-checkpoint-aws langchain-aws
```

ストアをセットアップする

index ブロックを使用してストアを設定し、setup() を呼び出します。

```
from langchain_aws import BedrockEmbeddings
from langgraph_checkpoint_aws import DynamoDBStore

store = DynamoDBStore(
    table_name="support-agent-memory",
    region_name="us-east-1",
    index={
        "embed": BedrockEmbeddings(model_id="amazon.titan-embed-text-v2:0"),
        "dims": 1024,
        "fields": ["text"],
        "distance_function": "COSINE",
    },
)

store.setup()
```

index ブロックの 4 つの設定について理解しておく価値があります。

- embed は、LangChain の埋め込みオブジェクト、または文字列のリストをベクトルのリストにマッピングするプレーン呼び出し可能オブジェクトを受け取ります。
- dims はモデルの出力サイズと一致している必要があります。Titan Text Embeddings V2 は、デフォルトで 1,024 次元を返します。これらが一致しない場合、ストアはインデックスが使用できないベクトルを書き込むのではなく、最初の書き込み時に両方の数値を示すディメンション不一致エラーを発生させます。
- fields は、値のどの部分を埋め込むかを選択します。ここでは、text フィールドのみが埋め込まれます。デフォルトの ["$"] は、値全体を JSON にシリアル化して埋め込みます。これは便利ですが、ブックキーピング属性も埋め込まれます。
- distance_function のデフォルトは COSINE であり、 EUCLIDEAN および DOT_PRODUCT も受け入れます。

setup() は、テーブルが存在しない場合はテーブルを作成し、ベクトルインデックスをアタッチして、設定されている場合は有効期限 (TTL) を有効にします。既存のテーブルでは不足しているものだけが追加されるため、既存の DynamoDBStore デプロイでは、テーブルを再作成したり項目を移行したりすることなくセマンティック検索を採用できます。setup() は、インデックスが ACTIVE を報告し、バックフィルが完了するまで返らないことに注意してください。バックフィル中に ACTIVE を報告するインデックスは、 SearchVectors 呼び出しを拒否します。新しい空のテーブルでは、待機時間は短くなります。既存の項目を含むテーブルへの後付けには、バックフィルにかかる時間と同じだけの時間がかかります。

メモリの書き込み

書き込みは通常の put() です。埋め込みは自動的に行われます。

```
namespace = ("memories", "acme-corp", "user-8812")

store.put(namespace, "mem-1", {
    "text": "Account runs a proxy that closes idle sockets after 60 seconds",
    "source": "ticket-4471",
})
store.put(namespace, "mem-2", {
    "text": "Customer prefers email, asked not to be called by phone",
    "source": "ticket-4471",
})
store.put(namespace, "mem-3", {
    "text": "Uses a custom build of the SDK pinned to version 2.14",
    "source": "ticket-4502",
```

```
})
```

意味によるリコール

お客様が新しい会話を開始し、「約 1 分後に接続が何度も切れてしまう」と話したとします。

```
results = store.search(
    namespace,
    query="connection drops after a minute of inactivity",
    limit=3,
)

for item in results:
    print(round(item.score, 3), item.value["text"])
```

出力:

```
0.324 Account runs a proxy that closes idle sockets after 60 seconds
0.039 Customer prefers email, asked not to be called by phone
0.031 Uses a custom build of the SDK pinned to version 2.14
```

関連するメモリが最初にランク付けされ、クエリ内のどのキーとも一致しません。SearchItem.score はLangGraph の規則に従っており、値が高いほど関連性が高いことを示します。一方、DynamoDB は距離を返すため、値が低いほど近いことを示します。そのため、ストアは値を次のように変換します。COSINE の場合は 1 - distance、EUCLIDEAN の場合は 1 / (1 + distance)、DOT_PRODUCT の場合はスコアをそのまま使用します。絶対スコアに基づいてメモリがプロンプトに挿入するのに十分な関連性があるかどうかを判断する場合は、そのしきい値を独自のデータと選択した距離関数に対して調整してください。

検索スキーマを使用したメモリのスコープ

DynamoDBStore がベクトルインデックスを作成するとき、インデックスの検索スキーマで、テーブルのパーティションキーを HASH 要素として宣言します。この要素が存在するため、すべての検索でこの要素に対する条件を指定する必要があり、ストアはその条件として名前空間を指定します。名前空間のタプルは結合されてパーティションキーの値になるため、すべてのセマンティック検索は、正確に 1 つの名前空間に限定されます。これには、次の 3 つの結果があります。

- 分離は構造的なものであり、忘れずに記述する必要があるフィルターではありません。検索は別の名前空間のメモリに到達できないため、多数のテナントにサービスを提供するストアには、別のテナントのメモリを返すクエリ形式が存在しません。これはクエリのスコープ設定であり、認可では

ありません。テーブルに対する dynamodb:SearchVectors 権限を持つプリンシパルは、任意の名前空間の値を直接検索できます。そのため、呼び出し元がどの名前空間を読み取れるかを決定することは、引き続き IAM とアプリケーションの認可レイヤーの役割です。

- リコールのコストは、テーブル全体ではなく 1 ユーザーのメモリ量に応じて決まります。検索処理は、製品全体が保持するメモリの数ではなく、その 1 つの名前空間が保持するメモリの量によって制限されます。そのため、規模が拡大しても、レイテンシーとベクトル検索コストの両方を小さく、安定した状態に保つことができます。
- セマンティック検索は、プレフィックスではなく、完全一致の名前空間を対象とします。検索は、指定した名前空間だけに到達し、その下位の名前空間には到達しません。("memories", "acme-corp") を検索しても、パーティションキーの値が異なるため、("memories", "acme-corp", "user-8812") には到達しません。名前空間がリコールのスコープとなるため、1 回の検索でどの範囲を参照させたいかに合わせて名前空間を選択してください。

以下の表は、名前空間の構成をどのように選択するかをまとめたものです。

名前空間の構成	1 回の **search()** で到達する範囲	この構成を選択する場合
("memories", user_id)	そのユーザーのメモリ	ユーザーごとにリコールを行うシングルテナント製品
("memories", tenant_id, user_id)	そのテナントにおける、そのユーザー	マルチテナント (一般的なケース)
("memories", tenant_id, user_id, agent_name)	そのユーザーに関する 1 つのエージェントのメモ	複数の専門エージェントが互いのメモを読み取るべきではない場合
("account_facts", tenant_id)	テナント全体のナレッジ	アカウント内のすべてのユーザーに適用されるファクト

すべてのメモリを 1 つの名前空間に配置してメタデータでフィルタリングすることは避けてください。フィルターは、DynamoDB が名前空間全体で最も近い一致を選択した後に適用され、1 回の検索で返される一致は最大 100 件です。そのため、ある利用量の多いテナントが一般的なクエリの上位 100 件を占めてしまうと、利用量の少ないテナントのメモリが存在しているにもかかわらず、そのテナントの検索では要求した件数より少ない結果しか返されなくなります。正確性を左右するもの

については、フィルターよりも名前空間のスコープを優先してください。リコールの境界よりも細かくスコープを設定することも、正当な設計方法です。例えば、ユーザー固有のコンテキストとアカウント全体のコンテキストの両方を必要とするエージェントは、両方の名前空間を検索して結果を統合します。

考慮事項

- インデックスは、グローバルセカンダリインデックスと同じモデルである、結果整合性を持ちます。put() の直後に実行した search() では、新しいメモリがまだ含まれていない場合があります。エージェントが同じターン内でメモリを書き込み、その後すぐにそのメモリをリコールする場合は、代わりにキーによって読み戻してください。
- 1 回の検索で返される一致は最大 100 件です。limit と offset の合計がこの上限を超えるリクエストは、上限を超えた分について何も返さずに済ませるのではなく、明確なエラーを返して拒否されます。
- 有効期限 (TTL) を使用して古いメモリを期限切れにし、読み取り時の更新を有効にすると、エージェントが実際にリコールするメモリの有効期限が延長されるため、アクティブに使用中のメモリがひっそりと消えることはありません。
- ベクトル検索の失敗はエラーとして発生するため、アプリケーション側で対応できます。ストアが失敗時に空のリストを返す場合、スロットリング、権限の問題、そして本当に結果が空だったケースがすべて同じように見えてしまいます。
- ベクトル操作は独自の単位で課金され、基本テーブルの読み取りおよび書き込みリクエストユニットとは別に計測されます。また、どちらもディメンション数に応じてスケールします。埋め込みのディメンション数が小さいほど、すべての書き込みとすべての検索でコストが低くなるため、リコールの品質を維持できる最小のディメンション数を使用してください。
- 設定した fields にテキストがない状態で書き込まれたメモリは、埋め込みなしで保存され、セマンティック検索の結果には表示されません。このような場合、ストアは警告をログに記録します。

その他のリソース

- GitHub の DynamoDBStore ドキュメント
- PyPI の langgraph-checkpoint-aws
- LangGraph ドキュメント

DynamoDB の生成 AI ブログ

以下の記事では、高度な AI を活用したアプリケーションの構築に DynamoDB の機能を活用するのに役立つ詳細なユースケース、ベストプラクティス、ステップバイステップガイドを提供します。

- 生成 AI チャットボット用の Amazon DynamoDB データモデル
- Amazon DynamoDB、Amazon Bedrock、LangChain を使用して、スケーラブルなコンテキスト対応チャットボットを構築する
- LangGraph と Amazon DynamoDB を使用して耐久性の高い AI エージェントを構築する

OpenSearch Service と DynamoDB のゼロ ETL 統合を活用する

Amazon Bedrock と DynamoDB を使用して、Amazon Titan やその他のサードパーティーモデルなどの基盤モデル (FM) へのサーバーレスアクセスを提供できます。DynamoDB のデータに対するベクトル検索では、ネイティブベクトルインデックスを使用できます (「DynamoDB でのベクトルインデックスの使用」を参照)。ベクトル検索とともに全文検索、分析、またはハイブリッド検索機能も必要な場合は、 アプリケーションを構築する際に Amazon OpenSearch Service とのゼロ ETL 統合を活用できます。生成 AI による DynamoDB ゼロ ETL から OpenSearch への統合と Amazon Bedrock ワークショップでは、OpenSearch と DynamoDB とのゼロ ETL 統合をセットアップする実践的な経験を提供します。このワークショップでは、次のタスクを実行します。

- DynamoDB テーブルから OpenSearch にパイプラインを作成します。
- OpenSearch に Amazon Bedrock コネクタを作成します。
- OpenSearch をベクトルストアとして使用して Amazon Bedrock にクエリを実行します。
- Amazon Bedrock の Claude FM を使用して、OpenSearch によって返された検索結果を説明する英語の書面によるレスポンスを作成します。

このワークショップでは、DynamoDB を OpenSearch と統合して生成 AI アプリケーションを構築できます。また、データベースエンジン全体で柔軟なクエリ機能を示し、従来のユースケースで DynamoDB と OpenSearch を統合するのに役立ちます。このワークショップは、Amazon DynamoDB Immersion Day の 7 つのモジュールの 1 つです。このワークショップは、どの AWS アカウントでも実行できます。

DynamoDB と OpenSearch Service 間のゼロ ETL 統合を設定する方法については、次のブログ記事を参照してください。このブログ記事では、OpenSearch Service でモデルコネクタをセットアップして、受信データに Amazon Bedrock を使用して、埋め込みを自動的に生成する方法についても説

明します。Amazon OpenSearch Service のゼロ ETL を使用した Amazon DynamoDB のベクトル検索

Amazon DynamoDB のクォータと制約

このトピックは、Amazon DynamoDB 内の現在のクォータ (以前は制限と呼ばれていました) について説明します。このトピックでは、現在のクォータの表示やクォータの引き上げのリクエストなど、クォータ管理タスクを実行する方法についても説明します。

トピック

- DynamoDB でのクォータ管理タスクの実行
- DynamoDB でのクォータ引き上げのリクエスト
- Amazon DynamoDB のクォータ
- Amazon DynamoDB での制約

DynamoDB でのクォータ管理タスクの実行

Amazon DynamoDB には、テーブル、ストリーム、インデックスなど、いくつかのサービスコンポーネントがあります。AWS アカウントを作成すると、これらのコンポーネントにデフォルトのクォータ (以前は制限と呼ばれていました) が設定されます。特に明記していない限り、クォータはリージョン固有です。クォータのいくつかは、リクエストによって引き上げることができます。リソースのクォータに達すると、そのリソースを作成するための追加のリクエストは、失敗して例外が発生します。

DynamoDB クォータへのアクセス

DynamoDB Service Quotas は、次のいずれかの方法で使用できます。

- AWS マネジメントコンソール

  Service Quotas コンソールは、Service Quotas を表示および管理するために使用できるブラウザベースのインターフェイスです。Service Quotas には、上部のナビゲーションバーで選択するか、AWS マネジメントコンソールで Service Quotas を検索することで、任意の AWS マネジメントコンソールページからアクセスできます。
- AWS Command Line Interface のツール

  AWS Command Line Interface ツールを使用する場合、システムのコマンドラインでコマンドを発行して、Service Quotas タスクを実行できます。コマンドラインツールは、AWS タスクを実行するスクリプトを作成する場合にも便利です。

- AWS SDK

  さまざまなプログラミング言語やプラットフォーム (Java、Python、Ruby、.NET、iOS、Android など) に AWS SDK を使用して、Service Quotas タスクを実行できます。

調整可能なクォータが Service Quotas で利用できない場合は、AWS Support Center Console を使用して Service Quotas の引き上げケースを作成します。

コンソールで現在のクォータを表示するには

Service Quotas コンソールを使用して現在の DynamoDB クォータを表示するには

1. https://console.aws.amazon.com/servicequotas/home/services/dynamodb/quotas/ で Service Quotas コンソールを開きます。
2. 画面上部のナビゲーションバーで、リージョンを選択します。
3. コンソールには、DynamoDB [クォータ名]、[適用されたアカウントレベルのクォータ値]、[AWS デフォルトのクォータ値]、[使用率]、およびアカウントレベルまたはリソースレベルのクォータの [調整可能性] に関する詳細が表示されます。

   適用されたクォータ値または使用率が利用できない場合、コンソールには [利用不可] と表示されます。適用されたクォータ値は、サポートセンターコンソールからリクエストできます。
4. 特定の [クォータ名] を選択すると、[詳細] ページが表示されます。このページには、クォータの [説明]、[クォータコード]、[クォータ ARN]、[使用率]、[適用されたアカウントレベルのクォータ値]、[調整可能性]、および [AWS デフォルトのクォータ値] が表示されます。

   該当する場合、[詳細] ページには、[モニタリング] オプション、[アラーム]、[リクエスト履歴]、クォータの [タグ] も表示されます。

AWS CLI を使用した現在のクォータの表示

DynamoDB クォータのデフォルト値を表示するには:

- DynamoDB サービスコード (dynamodb) を使用して `ListDefaultServiceQuotas` オペレーションを呼び出し、Amazon DynamoDB Service Quotas のデフォルト値を取得します。

```
$ aws service-quotas list-aws-default-service-quotas \
        --service-code dynamodb
```

```
 {
    "Quotas": [
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1::dynamodb/L-F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false
        },
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1::dynamodb/L-AB614373",
            "QuotaCode": "L-AB614373",
            "QuotaName": "Table-level write throughput limit",
            "Value": 40000.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false
        }......
    ]
}
```

適用されたクォータ値を表示するには:

- DynamoDB サービスコード (dynamodb) を使用して ListServiceQuotas オペレーションを呼び出すと、パラメータ QuotaAppliedAtLevel の値としてそれぞれ ACCOUNT、RESOURCE、または ALL を渡すことで、アカウントレベル、リソースレベル、またはすべてのレベルで適用されたすべてのクォータ値を取得できます。次の CLI の例では、アカウントレベルで適用されるクォータ値を取得します。

```
$ aws service-quotas list-service-quotas \
        --service-code dynamodb \
        --quota-applied-at-level ACCOUNT


{
```

```
    "Quotas": [
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1:303935678045:dynamodb/L-
F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
{
    "Quotas": [
        {
            "ServiceCode": "dynamodb",
            "ServiceName": "Amazon DynamoDB",
            "QuotaArn": "arn:aws:servicequotas:us-east-1:303935678045:dynamodb/L
-F7858A77",
            "QuotaCode": "L-F7858A77",
            "QuotaName": "Global Secondary Indexes per table",
            "Value": 20.0,
            "Unit": "None",
            "Adjustable": true,
            "GlobalQuota": false,
            "QuotaAppliedAtLevel": "ACCOUNT"
        }.....

        }
    ]
}
```

DynamoDB でのクォータ引き上げのリクエスト

Service Quotas コンソール、AWS CLI またはサポートケースを使用して、各リージョンのクォータの引き上げをリクエストできます。調整可能なクォータが Service Quotas で利用できない場合は、AWS Support Center Console を使用してサービスクォータの引き上げケースを作成します。

サポート は、クォータ引き上げのリクエストを承認、拒否、または部分的に承認できます。引き上げはすぐには許可されず、有効になるまでに数日かかる場合があります。

増加をリクエストするには、Service Quotas コンソールを使用してください。

1. Service Quotas のコンソールを開きます。https://console.aws.amazon.com/servicequotas/home/services/dynamodb/quotas/
2. 画面上部のナビゲーションバーで、リージョンを選択します。
3. リソース名でリストをフィルタリングします。例えば、[オンデマンド] と入力してオンデマンドインスタンスのクォータを確認してください。
4. クォータが調整可能な場合は、クォータを選択し、[クォータの引き上げをリクエスト] を選択します。
5. [クォータ値を変更] に、新しいクォータ値を入力します。
6. [リクエスト] を選択します。
7. コンソールで保留中または最近解決された要求を表示するには、ナビゲーションペインから [ダッシュボード] を選択します。保留中のリクエストの場合は、リクエストのステータスを選択してリクエストの受信をオープンします。リクエストの初期ステータスは [Pending] (保留中) です。ステータスが [要求されたクォータ] に変わると、サポート とケース番号が表示されます。リクエストのチケットを開くには、ケース番号を選択します。

AWS CLI や SDK を使用してクォータの増加をリクエストする方法などの詳細については、「Service Quotas ユーザーガイド」の「クォータ増加のリクエスト」を参照してください。

Amazon DynamoDB のクォータ

このセクションでは、Amazon DynamoDB 内の現在のクォータ (以前は制限と呼ばれていました) について説明します。各クォータは、指定がない限り、リージョン単位で適用されます。

> **Note**
>
> DynamoDB のすべてのサイズ測定値は、バイナリベースの単位を使用します。DynamoDB は、1 KB = 1024 バイト、1 MB = 1024 KB、1 GB = 1024 MB、1 TB = 1024 GB を示します。

トピック

- 読み取り/書き込みスループット
- リザーブドキャパシティ

- テーブル
- グローバルテーブル
- セカンダリインデックス
- ベクトルインデックス
- 射影されたセカンダリインデックスの属性
- DynamoDB Streams
- Amazon S3 からのインポート
- Amazon S3 へのテーブルのエクスポート
- バックアップと復元
- Contributor Insights

読み取り/書き込みスループット

スループットのデフォルトクォータ

AWS では、リージョン内でアカウントがプロビジョニングおよび利用できるスループットについて、いくつかのデフォルトのクォータがあります。

アカウントレベルの読み取りスループットとアカウントレベルの書き込みスループットクォータは、アカウントレベルで適用されます。これらのアカウントレベルのクォータは、特定のリージョン内のすべてのアカウントテーブルとグローバルセカンダリインデックスのプロビジョニングされたスループットキャパシティーの合計に適用されます。アカウントで使用可能なスループットをすべて 1 つのテーブルにプロビジョニングすることも、複数のテーブルに分けてプロビジョニングすることもできます。ーこれらのクォータは、プロビジョンドキャパシティーモードを使用するテーブルにのみ適用されます。

プロビジョンドキャパシティーモードを使用するテーブルと、オンデマンドキャパシティーモードを使用するテーブルに適用されるテーブルレベルの読み込みスループットのクォータとテーブルレベルの書き込みスループットクォータは、それぞれ異なります。

プロビジョンドキャパシティーモードのテーブルと GSI の場合、クォータは、リージョン内の任意のテーブルまたはその GSI にプロビジョニングできる読み取りおよび書き込みキャパシティユニットの最大数です。個々のテーブルとそのすべての GSI の合計も、アカウントレベルの読み取り/書き込みスループットクォータを下回っている必要があります。これに加えて、プロビジョニングされたすべてのテーブルとその GSI の合計がアカウントレベルの読み取り/書き込みスループットクォータを下回っている必要があります。

オンデマンドキャパシティモードテーブルと GSI の場合、テーブルレベルのクォータは、任意のテーブル、またはそのテーブル内の個々の GSI で使用可能な最大読み取り/書き込みキャパシティユニットです。アカウントレベルの読み取り/書き込みスループットクォータは、オンデマンドモードのテーブルには適用されません。

アカウントにデフォルトで適用されるスループットのクォータは以下のとおりです。

> **Note**
>
> すべてのキャパシティーユニットとリクエストユニットのクォータは秒単位で測定されます。例えば、読み込みキャパシティーユニットのクォータが 40,000 の場合、1 秒あたり 40,000 回の読み込みを意味します。

> **Note**
>
> サービスクォータを増やすことで、DynamoDB テーブルの読み取りキャパシティユニット (RCU) または書き込みキャパシティユニット (WCU) をいくつでもリクエストできます。次の表に示す値は、初期のデフォルトクォータです。これらはテーブルの上限値ではありません。

スループットクォータ名	オンデマンド	プロビジョンド	引き上げ可能
テーブルごと	40,000 の読み取りリクエストユニットと 40,000 の書き込みリクエストユニット	読み込みキャパシティーユニット数 40,000 および書き込みキャパシティーユニット数 40,000	はい
アカウントあたり	該当しない	読み込みキャパシティーユニット数 80,000 および書き込みキャパシティーユニット数 80,000	はい

スループットクォータ名	オンデマンド	プロビジョンド	引き上げ可能
任意のテーブルまたはグローバルセカンダリインデックスの最小スループット	該当しない	1 つの読み取りキャパシティユニットと 1 つの書き込みキャパシティユニット	はい

スループットの増加または減少 (プロビジョニングされたテーブルの場合)

プロビジョニングされたスループットの増加

`ReadCapacityUnits` または `WriteCapacityUnits` オペレーションを使用して、必要な回数だけ AWS マネジメントコンソール または `UpdateTable` を増やすことができます。1 回の呼び出しで、テーブル、そのテーブルの任意のグローバルセカンダリインデックス、またはこれらの任意の組み合わせに対して、プロビジョニングされるスループットを増やすことができます。新しい設定は、`UpdateTable` オペレーションが完了するまでは有効になりません。

プロビジョニングされたキャパシティーを追加する場合、アカウントごとのクォータを超えることはできません。また、DynamoDB では、プロビジョニングされたキャパシティーを急速に増やすことはできません。これらの制限に達しない限り、テーブルのプロビジョニング容量を必要なだけ増やすことができます アカウントごとのクォータの詳細については、前述の「スループットのデフォルトクォータ」セクションを参照してください。

プロビジョニングされたスループットの減少

`UpdateTable` オペレーションのすべてのテーブルとグローバルセカンダリインデックスでは、`ReadCapacityUnits` か `WriteCapacityUnits` (またはその両方) を減らすことができます。新しい設定は、`UpdateTable` オペレーションが完了するまでは有効になりません。

1 日あたりの DynamoDB テーブルで実行できるプロビジョンドキャパシティーの減少数には、デフォルトのクォータがあります。日付は、協定世界時 (UTC) に従って定義されます。毎日 4 回の減少が可能です。1 時間ごとに 1 回の追加減少が利用可能になり、最大 4 回まで利用できます。これにより、1 日 24 時間で、最大 27 回 (最初の 1 時間に 4 回、残りの 23 時間ごとに 1 回) 減らすことができます。

> **Important**
>
> テーブルとグローバルセカンダリインデックスの減少制限は別々に設定されているため、特定のテーブルのグローバルセカンダリインデックスにはいずれも、独自の減少制限が設定されています。ただし、1 つのリクエストでテーブルとグローバルセカンダリインデックスのスループットを縮小し、いずれかが現在の制限を超えた場合は拒否されます。リクエストが部分的に処理されることはありません。

Example

1 日の最初の 4 時間で、グローバルセカンダリインデックスが設定されているテーブルは次のように変更できます。

- テーブルの `WriteCapacityUnits` か `ReadCapacityUnits` (または両方) を 4 時間減らします。
- グローバルセカンダリインデックスの `WriteCapacityUnits` か `ReadCapacityUnits` (または両方) を 4 時間減らします。

同じ日の終わりに、テーブルとグローバルセカンダリインデックスのスループットは、合計 27 回ずつ縮小することができる可能性があります。

リザーブドキャパシティ

AWS アカウントで購入できるアクティブなリザーブドキャパシティの量に、デフォルトのクォータを設定します。クォータ制限は、書き込みキャパシティユニット (WCU) と読み取りキャパシティユニット (RCU) のリザーブドキャパシティの組み合わせです。

リザーブドキャパシティのクォータ	アクティブなリザーブドキャパシティ	引き上げ可能
アカウントあたり	1,000,000 のプロビジョンドキャパシティユニット (WCU _ RCU)	はい

1 回の購入で 1,000,000 を超えるプロビジョニング済みキャパシティユニットを購入しようとすると、このサービスクォータ制限に関するエラーが表示されます。アクティブなリザーブドキャパシ

ティがあり、追加のリザーブドキャパシティを購入しようとして、アクティブなプロビジョンドキャパシティユニットが 1,000,000 ユニットを超える場合、このサービスクォータ制限に関するエラーが表示されます。

テーブル

テーブルのサイズ

テーブルのサイズには実用的な制限はありません。テーブルは項目数やバイト数について制限がありません。

1 アカウント (1 リージョン) あたりのドメインの最大数

AWS アカウントについては、AWS リージョンごとに 2,500 個のテーブルという初期クォータがあります。

1 つのアカウントで2,500を超えるテーブルが必要な場合は、AWS アカウントチームに連絡して、最大 10,000 テーブルまで増やすことを検討してください。10,000 を超える場合のベストプラクティスは、複数のアカウントを設定することです。各アカウントは最大 10,000 個のテーブルに対応できます。

グローバルテーブル

グローバルテーブルを使用する場合、次のデフォルトのクォータが適用されます。

グローバルテーブルのデフォルトクォータ	オンデマンド	プロビジョンド
MRSC グローバルテーブルの数 (the section called “整合性モード” を参照)	任意のキャパシティモードで合計 400 の MRSC グローバルテーブル	任意のキャパシティモードで合計 400 の MRSC グローバルテーブル
マルチリージョンの結果整合性 (MREC) 用に設定されたテーブルあたりのスループット	40,000 の読み取りリクエストユニットと 40,000 の書き込みリクエストユニット	読み込みキャパシティーユニット数 40,000 および書き込みキャパシティーユニット数 40,000
マルチリージョンの強力な整合性 (MRSC) 用に設定された	40,000 の読み取りリクエストユニットと 40,000 の書き込みリクエストユニット	読み込みキャパシティーユニット数 40,000 および書き込

グローバルテーブルのデフォルトクォータ	オンデマンド	プロビジョンド
テーブルあたりのスループット		みキャパシティーユニット数 40,000
アカウント、リージョン、1 日あたりの新しいレプリカのバックフィルされたデータ	10 TB	10 TB

Note

場合によっては、AWS サポート を通じてクォータ制限の引き上げをリクエストする必要が生じることがあります。以下のいずれかに該当する場合は、https://aws.amazon.com/support を参照してください。

- レプリカの作成が成功するためには、グローバルテーブルのスループットクォータがテーブルあたりのスループットクォータ以上である必要があります。MREC グローバルテーブルと MRSC グローバルテーブルには、個別のグローバルテーブルスループットクォータがあります。
- 24 時間以内に 1 つまたは複数のレプリカを 1 つの送信先リージョンに追加し、合計が 10 TB を超える場合は、レプリカの追加データのバックフィルクォータについてサービスクォータの引き上げをリクエストする必要があります。
- テーブルレベルの書き込みスループット制限は、レプリカを持つすべてのリージョンで同じである必要があります。1 つのリージョンでテーブルレベルの書き込みスループット制限をデフォルトより高くする場合は、レプリカを追加する前に、他のすべてのレプリカリージョンで一致するクォータの引き上げをリクエストします。そうしないと、レプリカの作成が失敗し、次のような エラーが発生する可能性があります。
  - リージョン「example_region_B」の現在のアカウント制限を超えるため、リージョン「example_region_A」でテーブル「example_table」のレプリカを作成できません。

  このエラーが発生した場合は、レプリカを追加するすべてのリージョンで、テーブルレベルの書き込みスループット制限が一致していることを確認してください。これらの制限は、Service Quotas コンソールから自分で調整できます。テーブルレベルの書き込みスループットのクォータについては、「the section called “読み取り/書き込みスループット”」を参照してください。すべてのリージョンが一致しても エラーが表示される場合

は、AWS サポート (https://aws.amazon.com/support) までお問い合わせください。許可リストへの追加の登録が必要になる場合があります。

セカンダリインデックス

テーブルあたり最大 5 個のローカルセカンダリインデックスを定義できます。

デフォルトクォータとして、テーブルごとに 20 個のグローバルセカンダリインデックスがあります。

ベクトルインデックス

ベクトルインデックスには、次の制限が適用されます。

クォータ	デフォルト	引き上げ可能
テーブルあたりのベクトルインデックス	5	はい (AWS サポートに連絡してください)
ベクトルインデックスあたりの最大ディメンション	4,096	いいえ
`SearchVectors` リクエストあたりの TopK の最大数	100	いいえ
ベクトルインデックスあたりの最大パーティションキー (HASH)	1	いいえ
パーティションキーあたりのベクトル検索レート	1 GBps	はい (AWS サポートに連絡してください)
パーティションキーあたりのベクトルインデックス書き込みレート	10 MBps	はい (AWS サポートに連絡してください)
ベクトルインデックスあたりのインラインフィルターの数	18	いいえ

クォータ	デフォルト	引き上げ可能
許可リストなしのベクトルインデックス作成のベーステーブルの最大サイズ	600 GB	はい (AWS サポートに連絡してください)
テーブルあたりの同時ベクトルインデックスの作成または削除	1	いいえ

例: 768 次元の埋め込み (Cohere Embed v3 など) と 1 KB の非ベクトル項目データでは、各項目は約 4 KB です。この項目サイズでは、パーティションキーあたりのスループット制限は、1 秒あたり約 250,000 個のベクトルの検査 (1 GBps ÷ 4 KB) と 1 秒あたり 2,500 回のベクトル書き込み (10 MBps ÷ 4 KB) をサポートしています。パーティション内のベクトルの数が増えると、各検索はより多くのデータを調べ、検索の制限に達するまでの時間が早まります。パーティションキーの値全体で、スループットは線形にスケールします。ワークロードがこれらの制限を超える場合は、AWS サポートにお問い合わせください。

射影されたセカンダリインデックスの属性

1 つのテーブルのすべてのローカルおよびグローバルセカンダリインデックスを合わせて、最大 100 の属性を射影することができます。このクォータは、ユーザー指定の射影された属性だけに適用されます。

`CreateTable` オペレーションでは、`INCLUDE` として `ProjectionType` を指定した場合には、`NonKeyAttributes` で指定した属性の合計数がすべてのローカルセカンダリインデックス全体を合計して 100 を超えないようにする必要があります。同じ属性名を 2 つの異なるインデックスに射影すると、クォータに対する 2 つの異なる属性として計算されます。

このクォータは、`ProjectionType` が `KEYS_ONLY` または `ALL` であるセカンダリインデックスには適用されません。

DynamoDB Streams

DynamoDB Streams でのシャードの同時読み込み

単一リージョンのテーブルがグローバルテーブルでない場合、同じ DynamoDB Streams のシャードから、同時に 2 つまでの同時プロセスを読み込むように設計できます。この制限を超えると、リク

エストのスロットリングが発生する場合があります。グローバルテーブルでは、リクエストのスロットリングを回避するために、同時リーダーの数を 1 に制限することをお勧めします。

DynamoDB Streams が有効なテーブルの最大書き込みキャパシティ

AWS では、DynamoDB Streams が有効な DynamoDB テーブルの書き込み容量について、いくつかのデフォルトのクォータを用意しています。これらのデフォルトクォータは、プロビジョニングされた読み取り/書き込みキャパシティーモードのテーブルにのみ適用されます。

- テーブル単位 – 40,000 個の書き込みキャパシティーユニット

Amazon S3 からのインポート

us-east-1、us-west-2、および eu-west-1 の各リージョンでは、Amazon S3 からの DynamoDB のインポートは、一度に 15 TB の合計インポートソースオブジェクトサイズで 50 個の同時インポートジョブに対応できます。他のすべてのリージョンでは、合計サイズが 1 TB で最大 50 個の同時インポートタスクに対応できます。各インポートジョブでは、すべてのリージョンで最大 50,000 個の Amazon S3 オブジェクトを使用できます。インポートと検証の詳細については、「インポート形式の割り当てと検証」を参照してください。

Amazon S3 へのテーブルのエクスポート

フルエクスポート: 最大 300 個の同時エクスポートタスク、またはすべての処理中のテーブルエクスポートから合計 100 TB をエクスポートできます。これらの制限は両方とも、エクスポートがキューに入る前に確認されます。

増分エクスポート: Amazon S3 への DynamoDB 増分エクスポートは、最大 300 個の同時エクスポートジョブ、またはすべての処理中のテーブルエクスポートから合計 100 TB までサポートできます。エクスポート期間ウィンドウの制限は、最小 15 分、最大 24 時間です。

バックアップと復元

DynamoDB は、DynamoDB のオンデマンドバックアップまたは継続的バックアップを介して、合計 50 TB の同時復元を最大 50 サポートします。AWS Backup は、合計 25 TB の同時復元を最大 50 サポートします。

Contributor Insights

DynamoDB テーブルで Customer Insights を有効にする場合でも、Contributor Insights のルール制限が適用されます。詳細については、「CloudWatch サービスのクォータ」を参照してください。

Amazon DynamoDB での制約

このセクションでは、Amazon DynamoDB 内の現在の制約 (以前は制限と呼ばれていました) について説明します。

> **Note**
>
> DynamoDB のすべてのサイズ測定値は、バイナリベースの単位を使用します。DynamoDB は、1 KB = 1024 バイト、1 MB = 1024 KB、1 GB = 1024 MB、1 TB = 1024 GB を示します。

トピック

- 読み取り/書き込みキャパシティモード
- セカンダリインデックス
- パーティションキーおよびソートキー
- 名前付けルール
- データ型
- 項目
- 属性
- 式パラメータ
- DynamoDB のトランザクション
- DynamoDB Streams
- DynamoDB Accelerator (DAX)
- API 固有の制約
- 保管時の DynamoDB 暗号化

読み取り/書き込みキャパシティモード

24 時間のローリングウィンドウで、テーブルをプロビジョンドキャパシティモードからオンデマンドモードに最大 4 回切り替えることができます。テーブルは、オンデマンドモードからプロビジョンドキャパシティモードにいつでも切り替えることができます。

読み込みおよび書き込みキャパシティモード間の切り替えの詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。

キャパシティユニットサイズ (プロビジョニングされるテーブルの場合)

最大サイズが 4 KB の項目について、1 つの読み込みキャパシティーユニット = 1 秒あたり 1 回の強力な整合性のある読み込み、あるいは 1 秒あたり 2 回の結果整合性のある読み込み。

最大サイズが 1 KB の項目について、1 つの書き込みキャパシティーユニット = 1 秒あたり 1 回の書き込み

トランザクション読み込みリクエストでは、最大 4 KB の項目を 1 秒あたりに 1 回読み込むために読み込みキャパシティーユニットが 2 個必要です。

トランザクション書き込みリクエストでは、最大 1 KB の項目を 1 秒あたり 1 回書き込むのに書き込みキャパシティーユニットが 2 個必要です。

リクエストユニットサイズ (オンデマンドテーブルの場合)

最大サイズが 4 KB の項目について、1 つの読み込みリクエストユニット = 1 秒あたり 1 回の強力な整合性のある読み込み、あるいは 1 秒あたり 2 回の結果整合性のある読み込み。

最大サイズが 1 KB の項目について、1 つの書き込みリクエストユニット = 1 秒単位の書き込み。

トランザクション読み込みリクエストでは、最大 4 KB の項目を 1 秒に 1 回読み込むのに読み込みリクエストユニットが 2 個必要です。

トランザクション書き込みリクエストでは、1 KB の項目を 1 秒に 1 回書き込むのに書き込みリクエストユニットが 2 個必要です。

セカンダリインデックス

テーブルごとの射影されたセカンダリインデックスの属性

合計最大 100 の属性を、1 つのテーブルのすべてのグローバルセカンダリインデックスに射影することができます。これは、ユーザー指定の射影された属性だけに適用されます。

`CreateTable` オペレーションでは、`INCLUDE` として `ProjectionType` を指定した場合には、`NonKeyAttributes` で指定した、すべてのローカルセカンダリインデックスを含む属性の合計数が 100 を超えてはいけません。同じ属性名を 2 つの異なるインデックスに射影した場合には、合計を計算する際に 2 つの異なる属性として計算されます。

この制限は、`ProjectionType` が `KEYS_ONLY` または `ALL` であるセカンダリインデックスには適用されません。

パーティションキーおよびソートキー

パーティションキーの長さ

パーティションキーと値の最小長は 1 バイトです。最大長は 2048 バイトです

パーティションキーの値

テーブルまたはセカンダリインデックスについて、パーティションキー値の明確な数に関する実質的な制限はありません。

ソートキーの長さ

ソートキーと値の最小長は 1 バイトです。最大長は 1024 バイトです

ソートキー値

一般的に、パーティションキーの値ごとのソートキーの値の数について、実質的に制限はありません。

セカンダリインデックスを持つテーブルは例外です。項目コレクションは、パーティションキー属性の値が同じ項目のセットです。グローバルセカンダリインデックスでは、項目コレクションはベーステーブルから独立しています (そのため、異なるパーティションキー属性を持つこともできます)。ただし、ローカルセカンダリインデックスでは、インデックス付きビューはテーブル内の項目と同じパーティションに配置され、同じパーティションキー属性を共有します。このローカリティのため、テーブルに 1 つ以上の LSI がある場合、項目コレクションを複数のパーティションに分散することはできません。

1 つ以上の LSI を含むテーブルの場合、項目コレクションのサイズは 10 GB を超えることはできません。これには、パーティションキー属性の値が同じであるすべてのベーステーブル項目およびすべての射影された LSI ビューが含まれます。パーティションの最大サイズは 10 GB です。詳細については、「項目コレクションのサイズ制限」を参照してください。

名前付けルール

テーブル名とセカンダリインデックス名

テーブルとセカンダリインデックスの名前は、3 文字以上、255 文字以下である必要があります。使用可能な文字は次のとおりです。

- A-Z

- a-z
- 0-9
- _ (下線)
- - (ハイフン)
- . (ドット)

属性名

一般的に、属性名は 1 文字以上、64 KB 以下である必要があります。

以下の例外があります。これらの属性名は 255 文字以下である必要があります。

- セカンダリインデックスのパーティションキー名。
- セカンダリインデックスのソートキー名。
- ユーザー指定の射影された属性の名前 (ローカルセカンダリインデックスのみに適用)。CreateTable オペレーションでは、ProjectionType を INCLUDE に指定した場合には、NonKeyAttributes パラメータの属性名の長さが制限されます。射影タイプ KEYS_ONLY および ALL には影響しません。

これらの属性名は UTF-8 を使用してエンコードする必要があり、それぞれの名前の合計サイズが (エンコード後に) 255 バイトを超えることはできません。

データ型

String

文字列の長さは、項目の最大サイズである 400 KB に制約されます。

文字列は、UTF-8 バイナリエンコードの Unicode です。UTF-8 は可変幅のエンコードであるため、DynamoDB により、UTF-8 バイトを使用して文字列の長さが決定されます。

Number

数値は、最大 38 桁の精度であり、正、負、または 0 のいずれかです。

- 正の範囲: 1E-130 ~ 9.99999999999999999999999999999999999999E+125
- 負の範囲: -9.99999999999999999999999999999999999999E+125 ~ -1E-130

DynamoDB では、リクエストと返信の数値データが JSON 文字列で表示されます。詳細については、「DynamoDB 低レベル API」を参照してください。

数値の精度が重要な場合は、数値型から変換する文字列を使用して、DynamoDB に数値を渡します。

バイナリ

バイナリの長さは、項目の最大サイズである 400 KB に制約されます。

バイナリ属性を操作するアプリケーションは、データを DynamoDB に送信する前に、base64 形式でエンコードする必要があります。その後 DynamoDB は、受信したデータを署名なしバイト配列にデコードし、それを属性の長さとして使用します。

項目

項目のサイズ

DynamoDB の項目の最大サイズは 400 KB で、属性名のバイナリの長さ (UTF-8 の長さ) と属性値の長さ (こちらもバイナリの長さ) を含みます。属性名はサイズ制限に反映されます。

たとえば、2 つの属性を持つ項目があり、1 つの属性は名前が "shirt-color" で値が "R"、別の属性は名前が "shirt-size" で値が "M" であるとします。この項目の合計サイズは 23 バイトです。

ローカルセカンダリインデックスを持つテーブルの項目のサイズ

テーブルのローカルセカンダリインデックスごとに、次を合計したサイズに関して 400 KB の制限があります。

- テーブルの項目データのサイズ。
- すべてのローカルセカンダリインデックスエントリ (キーの値と射影された属性を含む) の対応するエントリサイズ。

属性

項目あたりの属性名と値のペア

項目あたりの属性の累積サイズは、DynamoDB の項目の最大サイズ (400 KB) 以内である必要があります。

リスト、マップ、またはセットの値の数

値を含む項目が 400 KB の制限内である限り、リスト、マップ、またはセットにおける値の最大数の制限はありません。

属性値

属性がテーブルまたはインデックスのキー属性として使用されていない場合は、空の文字列属性とバイナリ属性値を使用できます。空の文字列とバイナリ値は、セット、リスト、およびマップ型内で許可されます。属性値は空のセット (文字列セット、数値セット、またはバイナリセット) にすることはできません。ただし、空のリストおよびマップは許可されます。

入れ子の属性の深さ

DynamoDB では、深さが最大 32 のレベルの入れ子の属性をサポートします。

式パラメータ

式パラメータには、ProjectionExpression、ConditionExpression、UpdateExpression、および FilterExpression があります。

長さ

任意の式の最大長は 4 KB です。たとえば、ConditionExpression a=b のサイズは 3 バイトです。

1 つの式属性名または式属性値の最大長は 255 バイトです。たとえば、#name は 5 バイト、:val は 4 バイトです。

式のすべての置換変数の最大長は 2 MB です。これはすべての ExpressionAttributeNames および ExpressionAttributeValues の長さの合計です。

演算子およびオペランド

1 つの式で許容される演算子または関数の最大数は 300 です。この制限は、ConditionExpression、FilterExpression、UpdateExpression を含むすべての式タイプに適用されます。

以下の各項目は、この制限に対して 1 つの演算子としてカウントされます。

- =、<>、<、<=、>、>= など、条件式またはフィルター式のコンパレータ。
- 論理キーワード (AND、OR、NOT) または BETWEEN や IN キーワード。

- attribute_exists、begins_with、contains、size などの関数。
- UpdateExpression の算術演算子 (+ または -)。

UpdateExpression では、SET アクションの = は割り当て構文であり、演算子としてカウントされません。条件式またはフィルター式では、= コンパレータはカウントされます。例えば、フィルター式 #a = :val OR #b = :val には 3 つの演算子 (2 つの = コンパレータと 1 つの OR) が含まれ、更新式 SET Price = Price + :val には 1 つの演算子 (+) が含まれます。

> **Note**
>
> 式が 300 演算子の制限を超えると、DynamoDB は The expression contains too many operators; operator count: 301 という形式のメッセージを含む ValidationException を返します。レポートされるカウントは常に 301 であり、式の実際の演算子の数を反映したものではありません。

IN コンパレータのオペランドの最大数は 100 です。

予約語

DynamoDB では、予約語と競合する名前の使用を防ぐことはできません。(詳細な一覧については、「DynamoDB の予約語」を参照してください)。

ただし、式パラメータで予約語を使用する場合は、ExpressionAttributeNames も指定する必要があります。詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

DynamoDB のトランザクション

DynamoDB トランザクションの API オペレーションには次の制約があります。

- トランザクションには、100 個を超える一意のアクションを含めることはできません。
- トランザクションには、4 MB を超えるデータを含めることはできません。
- トランザクション内の 2 つのアクションを、同じテーブルの同じ項目に対して実行することはできません。たとえば、ConditionCheck と Update の両方を 1 つのトランザクションで同じ項目に対して実行することはできません。
- トランザクションは、複数の AWS アカウントまたはリージョンのテーブルで動作できません。
- トランザクションオペレーションは、書き込みが最初に行われた AWS リージョン内でのみ、不可分性、一貫性、分離性、および耐久性 (ACID) を保証します。グローバルテーブルのリージョン間

では、トランザクションはサポートされていません。たとえば、米国東部 (オハイオ) リージョンと米国西部 (オレゴン) リージョンにレプリカを含むグローバルテーブルがあり、米国東部 (バージニア北部) リージョンで `TransactWriteItems` オペレーションを実行するとします。この場合、変更がレプリケートされると、米国西部 (オレゴン) リージョンで部分的に完了したトランザクションを確認できます。変更は、ソースリージョンでコミットされると、他のリージョンにのみレプリケートされます。

DynamoDB Streams

DynamoDB Streams でのシャードの同時読み込み

単一リージョンのテーブルがグローバルテーブルでない場合、同じ DynamoDB Streams のシャードから、同時に 2 つまでのプロセスを読み込むように設計できます。この制限を超えると、リクエストのスロットリングが発生する場合があります。グローバルテーブルでは、リクエストのスロットリングを回避するために、同時リーダーの数を 1 に制限することをお勧めします。

DynamoDB Accelerator (DAX)

AWS が利用可能なリージョン

DAX を使用できる AWS リージョンのリストについては、「AWS 全般のリファレンス」の「DynamoDB アクセラレーター (DAX)」を参照してください。

ノード

DAX クラスターは、1 つのみのプライマリノードと、0 ~ 10 個のリードレプリカノードで構成されます。

ノードの総数 (AWS アカウントごと) は、1 つの AWS リージョン内で 50 を超えることはできません。

パラメータグループ

リージョンごとに最大 20 の DAX パラメータグループを作成できます。

サブネットグループ

リージョンごとに最大 50 の DAX サブネットグループを作成できます。

サブネットグループ内では、最大 20 のサブネットを定義できます。

> **Important**
>
> DAX クラスターは、最大 500 の DynamoDB テーブルをサポートします。DynamoDB テーブルが 500 を超えると、クラスターの可用性とパフォーマンスが低下する可能性があります。

API 固有の制約

CreateTable/UpdateTable/DeleteTable/PutResourcePolicy/DeleteResourcePolicy

一般に、CreateTable、UpdateTable、DeleteTable、PutResourcePolicy、DeleteResourcePolicy の各リクエストは、どのような組み合わせでも最大 500 件まで同時に実行できます。その結果、CREATING、UPDATING、または DELETING の状態のテーブルの合計数が 500 を超えることはできません。

テーブルグループ全体で任意の組み合わせで、ミュータブル (CreateTable、DeleteTable、UpdateTable、PutResourcePolicy、DeleteResourcePolicy) コントロールプレーン API リクエストを 1 秒あたり最大 2,500 回送信できます。ただし、PutResourcePolicy リクエストおよび DeleteResourcePolicy リクエストにはそれぞれ下限があります。詳細については、PutResourcePolicy と DeleteResourcePolicy のクォータに関する以下の詳細を参照してください。

また、リソースベースのポリシーを含む CreateTable リクエストおよび PutResourcePolicy リクエストは、ポリシーの KB ごとに 2 つの追加リクエストとしてカウントされます。例えば、サイズが 5 KB のポリシーの CreateTable リクエストまたは PutResourcePolicy リクエストは 11 リクエストとしてカウントされます。1 つは CreateTable リクエスト、10 はリソースベースのポリシー (2 x 5 KB) です。同様に、サイズが 20 KB のポリシーは 41 件のリクエストとしてカウントされます。1 つはリクエストで、40 はリソースベースのポリシー (2 x 20 KB) です。

PutResourcePolicy

1 つのテーブルで 1 秒あたり最大 25 個の PutResourcePolicy API リクエストを送信できます。個々のテーブルのリクエストが成功すると、その後の 15 秒間、新しい PutResourcePolicy リクエストはサポートされません。

リソースベースのポリシードキュメントでサポートされる最大サイズは 20 KB です。DynamoDB では、この上限に照らしてポリシーのサイズを計算する際に空白はカウントされません。

DeleteResourcePolicy

1 つのテーブルで 1 秒あたり最大 50 個の DeleteResourcePolicy API リクエストを送信できます。個々のテーブルの PutResourcePolicy リクエストが成功すると、その後の 15 秒間、DeleteResourcePolicy リクエストはサポートされません。

BatchGetItem

1 回の BatchGetItem オペレーションで、最大 100 項目を取得できます。取得するすべての項目の合計サイズが 16 MB を超えてはいけません。

BatchWriteItem

1 回の BatchWriteItem オペレーションでは、最大 25 の PutItem、または DeleteItem リクエストを含むことができます。書き込むすべての項目の合計サイズが 16 MB を超えてはいけません。

DescribeStream

DescribeStream は最大で毎秒 10 回呼び出すことができます。

DescribeTableReplicaAutoScaling

DescribeTableReplicaAutoScaling メソッドでは、1 秒あたり 10 リクエストのみサポートされます。

DescribeLimits

DescribeLimits は定期的に呼び出すのみにします。1 分以内に複数回呼び出すと、スロットリングエラーが発生する可能性があります。

DescribeContributorInsights/ListContributorInsights/UpdateContributorInsights

DescribeContributorInsights、ListContributorInsights、およびUpdateContributorInsights は定期的に呼び出すのみにします。DynamoDB では、これらの API のそれぞれで 1 秒あたり最大 5 つのリクエストがサポートされます。

DescribeTable/ListTables/GetResourcePolicy

読み取り専用 (DescribeTable、ListTables、GetResourcePolicy) コントロールプレーン API リクエストを組み合わせて、1 秒あたり最大 2,500 回送信できます。GetResourcePolicy API には、1 秒あたり 100 リクエストという個別の下限があります。

DescribeTimeToLive

DescribeTimeToLive オペレーションは、1 秒あたり 10 個の読み取りリクエストユニットでスロットリングされます。この制限を超えると、DynamoDB は ThrottlingException エラーを返します。

Query

Query の結果セットは、1 回の呼び出しあたり 1 MB に制限されます。クエリ応答から LastEvaluatedKey を使用して、結果をさらに取り出すこともできます。

Scan

Scan の結果セットは、1 回の呼び出しあたり 1 MB に制限されます。スキャン応答から LastEvaluatedKey を使用して、結果をさらに取り出すこともできます。

UpdateKinesisStreamingDestination

UpdateKinesisStreamingDestination オペレーションを実行する場合、24 時間に最大 3 回、ApproximateCreationDateTimePrecision を新しい値に設定できます。

UpdateTableReplicaAutoScaling

UpdateTableReplicaAutoScaling メソッドでは、1 秒あたり 10 リクエストのみサポートされます。

UpdateTableTimeToLive

UpdateTableTimeToLive メソッドでは、指定されたテーブルごとに、Time to Live (TTL) の有効化または無効化のリクエストが 1 時間あたり 1 つだけサポートされます。この変更が完全に処理されるまでに最大で 1 時間かかる場合があります。この 1 時間の間に同じテーブ

ルに対して追加の `UpdateTimeToLive` 呼び出しが行われると、ValidationException が発生します。

保管時の DynamoDB 暗号化

AWS 所有のキー、AWS マネージドキー、カスタマーマネージドキーは 24 時間いつでも切り替えることができます。テーブルの作成時から、テーブル単位で最大 4 回まで切り替え可能です。また、過去 6 時間以内に変更がなかった場合は、追加で変更することができます。これにより、1 日で変更できる最大の回数は 8 回になります (1 日の中で最初の 6 時間は 4 回、その後は 6 時間ごとに 1 回)。

AWS 所有のキー を使用する暗号化キーの切り替えは、上述のクォータを使い尽くしても、必要な回数だけ行うことができます。

クォータの拡大をリクエストしない限り、以下のクォータが適用されます。サービスクォータの増加をリクエストするには、https://aws.amazon.com/support を参照してください。

DynamoDB API リファレンス

Amazon DynamoDB API リファレンスには、以下によりサポートされているオペレーションの詳しいリストが記載されています。

- DynamoDB
- DynamoDB Streams
- DynamoDB Accelerator (DAX)

Amazon DynamoDB のトラブルシューティング

以下のトピックでは、Amazon DynamoDB の使用時に発生する可能性があるエラーや問題のトラブルシューティングに関するアドバイスを提供します。ここに記載されていない問題が見つかった場合は、このページの [フィードバック] ボタンを使用して報告できます。

トラブルシューティングに関するアドバイス、およびサポートへの一般的な質問に対する回答については、AWS ナレッジセンターにアクセスしてください。

トピック

- Amazon DynamoDB の内部サーバーエラーのトラブルシューティング
- Amazon DynamoDB でのレイテンシーの問題のトラブルシューティング
- Amazon DynamoDB でのスロットリングのトラブルシューティング

Amazon DynamoDB の内部サーバーエラーのトラブルシューティング

DynamoDB の場合、内部サーバーエラー (500 エラー) は、サービスがリクエストを処理できないことを示します。これらのエラーは、フリート内の一時的なネットワーク問題、インフラストラクチャの問題、ストレージノード関連の問題など、さまざまな理由で発生する場合があります。

内部サーバーエラーは、DynamoDB テーブルのライフサイクル中に発生する場合があります。これはサービスの分散性が原因であると予想されるため、通常、懸念の原因とはならないはずです。DynamoDB は、ユーザーの介入を必要とせずに、サービスの一時的な問題をリアルタイムで自動的に修復します。ただし、テーブルへのリクエストで内部サーバーエラーが (the section called “SystemErrors” メトリクスに示されるように) 一貫して多く発生する場合は、さらに調査する必要があります。

トピック

- 内部サーバーエラーの調査
- 内部サーバーエラーの影響を最小限に抑える
- 運用意識の向上

内部サーバーエラーの調査

DynamoDB テーブルで内部サーバーエラーが発生した場合は、以下のオプションを検討してください。

1. AWS Health Dashboard を確認します。

   問題を特定するには、最初のステップとして、AWS Service Health Dashboard と AWS Account Health Dashboard を確認します。これらのダッシュボードには、サービス全体の問題、影響を受けたテーブル、進行中の問題、問題解決後の根本原因に関する貴重な情報が表示されます。

   これらのダッシュボードの詳細を確認すると、使用している AWS のサービスの現在のステータスと、アカウントに影響している可能性がある問題をよりよく理解できます。これらの情報は、問題に対処するための次のステップを決定したり、運用の中断を最小限に抑えたりするのに役立ちます。

2. に問い合わせます。サポート

   リクエストでのエラーが長く続く場合は、サービスに問題がある可能性があります。一般的に、直近の 15 分間に全体的な障害率が 1% 以上に達した場合は、問題を AWS サポートチームにエスカレーションするのが適切な対応です。詳細については、「DynamoDB サービスレベルアグリーメント」を参照してください。

   AWS サポートチームへのケースを開くときは、トラブルシューティングプロセスを迅速化するために、以下の詳細を入力します。

   - 影響を受けている DDB、テーブル、またはセカンダリインデックス
   - エラーが観察された時間枠
   - `4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG` などの DynamoDB リクエスト ID (アプリケーションログで確認できます)

   これらの詳細をサポートケースに含めると、AWS チームが問題を理解し、より迅速に解決できるようになります。リクエスト ID がない場合でも、他の利用可能な詳細でケースを記録する必要があります。

内部サーバーエラーの影響を最小限に抑える

DynamoDB の使用時に内部サーバーエラーが発生した場合は、これらがアプリケーションに与える影響を最小限に抑え、以下のベストプラクティスを検討してください。

- バックオフと再試行を使用する: DynamoDB のデフォルトの SDK 動作は、バックオフと再試行戦略の観点から、ほとんどのアプリケーションに適したバランスを見つけるように設計されています。ただし、ダウンタイムとパフォーマンス要件に対するアプリケーションの許容度に基づいて、これらの設定を調整できます。バックオフと再試行の詳細を確認して、これらの再試行設定を微調整する方法を理解します。
- 結果整合性のある読み込みを使用する: アプリケーションに強力な整合性のある読み込みが必要ない場合は、結果整合性のある読み込みを使用することを検討します。これらの読み込みは、使用可能なストレージノードのいずれかで処理されるため、コストが低く、内部サーバーエラーによる一時的な問題が発生する可能性が低くなります。詳細については、「DynamoDB の読み取り整合性」を参照してください。

運用意識の向上

今日のデジタル環境ではアプリケーションの高可用性と信頼性を維持することが不可欠です。この場合、重要な側面の 1 つは、DynamoDB テーブルの内部サーバーエラー (ISE) とグローバルセカンダリインデックス (GSI) をプロアクティブにモニタリングすることです。これらのエラーをモニタリングする CloudWatch アラームを作成することで、運用意識を高め、潜在的な問題がエンドユーザーに影響を与える前にアラートを受け取ることができます。このアプローチは、AWS Well-Architected フレームワークの運用上の優秀性の柱と一致し、DynamoDB ワークロードをパフォーマンス、セキュリティ、信頼性に向けて最適化します。

CloudWatch アラームの作成

内部サーバーエラーの発生数が一貫して高い場合は、メトリクスを手動で監視するのではなく、DynamoDB テーブルに CloudWatch アラームを設定して通知を受け取るようにする必要があります。これは、AWS のあらゆるワークロードに対する Well-Architected フレームワークの運用上の優秀性の柱と一貫します。DynamoDB テーブルの優れた設計の詳細については、「DynamoDB の Well-Architected レンズを使用して DynamoDB ワークロードを最適化する」を参照してください。

the section called “SystemErrors” メトリクスでアラームを作成するときは、`TableName` および `Operation` ディメンション (グローバルセカンダリインデックスの場合は `TableName` および `GlobalSecondaryIndexName`) の両方を指定します。DynamoDB は `TableName` 単独ではなくオペレーションごとに `SystemErrors` を出力するため、`TableName` のみを指定するアラームは `INSUFFICIENT_DATA` 状態のままになり、アラートを出力しません。

Amazon DynamoDB でのレイテンシーの問題のトラブルシューティング

ワークロードのレイテンシーが高いと思われる場合は、CloudWatch `SuccessfulRequestLatency` メトリクスを分析し、平均レイテンシー、およびパーセンタイルメトリクス介したレイテンシーの中央値 (p50) をチェックして、それが DynamoDB に関連しているかどうかを確認できます。報告された `SuccessfulRequestLatency` にある程度のばらつきがあるのは正常であり、時折 (特に `Maximum` 統計および高パーセンタイル) 急上昇が起きても心配する必要はありません。ただし、`Average` 統計または p50 (中央値) が急激に増加し続けている場合は、AWS Service Health Dashboard と Personal Health Dashboard で詳細を確認する必要があります。考えられる原因としては、テーブル内の項目のサイズ (1 KB の項目と 400 KB の項目ではレーテンシーが異なります) やクエリのサイズ (10 項目と 100 項目) などがあります。

パーセンタイルメトリクス (p99、p90 など) は、レイテンシー分布をよりよく理解するのに役立ちます。例えば、次のようになります。

- p50 (中央値) は、ワークロードの一般的なレイテンシーを示します。
- p90 は、リクエストの 90% がこの値よりも高速であることを示しています。
- p99 は、リクエストの 1% に影響する最悪の場合のレイテンシーを特定するのに役立ちます。

p50 値が正常で p99 値が高い場合は、リクエストのごく一部に影響する散発的な問題を示している可能性があり、p50 値が一貫して高い場合は、パフォーマンスの低下を示している可能性があります。

> **Note**
>
> カスタムパーセンタイル値 (p99.9 など) を分析するには、CloudWatch メトリクス統計フィールドに目的のパーセンタイル (p99.9 など) を手動で入力できます。これにより、ドロップダウンに表示されるデフォルトのパーセンタイルを超えるレイテンシー分布を評価できます。

レイテンシーメトリクスのバリエーションは、特にパーセンタイルが高い場合、DynamoDB テーブルに保存されているデータの高可用性と耐久性を維持する DynamoDB 主導のバックグラウンドオペレーションや、一時的なインフラストラクチャの問題の結果として発生する可能性があります。

必要に応じて、AWS サポート でサポートケースを作成することを検討し、作成したランブックに従って、アプリケーションで使用可能なフォールバックオプション (マルチリージョンアーキテクチャの場合はリージョンの退避など) を引き続き評価してください。リクエストが遅い場合は、サポートケースを開くときに AWS サポートに提示できるようにリクエスト ID を記録しておく必要があります。

SuccessfulRequestLatency メトリックスは DynamoDB サービス内部のレイテンシーのみを測定し、クライアント側のアクティビティとネットワークトリップ時間は含まれません。クライアントから DynamoDB サービスへの呼び出しの全体的なレイテンシーについて詳しく知るには、AWS SDK でレイテンシーメトリックスロギングを有効にできます。

> **Note**
>
> ほとんどのシングルトンオペレーション (プライマリキーの値を完全に指定することで 1 つの項目に適用されるオペレーション) では、DynamoDB は 1 桁のミリ秒単位の Average SuccessfulRequestLatency を返します。この値には、DynamoDB エンドポイントにアクセスする呼び出し側コードのトランスポートオーバーヘッドは含まれていません。複数項目のデータオペレーションの場合、レーテンシーは結果セットのサイズ、返されるデータ構造の複雑さ、適用される条件式やフィルター式などの要因によって異なります。同じパラメータで同じデータセットに対して複数項目のオペレーションを繰り返す場合、DynamoDB は一貫した高い Average SuccessfulRequestLatency を提供します。

レイテンシーを減らすには、次の戦略を 1 つ以上検討してください。

- 接続の再利用: DynamoDB リクエストは、デフォルトで HTTPS に設定されている認証済みセッションを経由して行われます。接続を開始するには複数のラウンドトリップが必要で、最初のリクエストのレイテンシーが接続を再利用する後のリクエストよりも長くなります。すでに初期化されている接続でリクエストを行うと、DynamoDB は整合性のある低いレイテンシーを実現できます。新しい接続を確立する際のレイテンシーオーバーヘッドを避けるために、他のリクエストが行われていない場合は 30 秒ごとに GetItem リクエストを送信することで、「キープアライブ」メカニズムを実装できます。
- 結果整合性のある読み込みを使用する:アプリケーションで強い整合性のある読み取りが必要ない場合は、デフォルトの結果整合性のある読み込みを使用することを検討してください。結果整合性のある読み込みはコストが低く、複数のアベイラビリティーゾーンから取得できるため、リクエスタに近接するアベイラビリティーゾーンを選択でき、レイテンシーが短縮されます。詳細については、「DynamoDB の読み取り整合性」を参照してください。

- リクエストヘッジを実装する: p99 レイテンシー要件が非常に低い場合は、リクエストヘッジを実装することを検討してください。リクエストヘッジでは、最初のリクエストが十分な速さでレスポンスを受信しない場合は、2 番目の同等のリクエストを送信して競合させます。最初のレスポンスが優先されます。こうすることで、追加リクエストによりテールレイテンシーを向上させることができます。2 番目のリクエストを送信するまでの待機時間を決定できます。読み取りでは、ヘッジがより簡単になります。書き込みの場合は、タイムスタンプベースの順序付けを使用して、ヘッジされたリクエストが最初のリクエストのタイミングで取り扱われるようにし、誤った順序での処理を防ぎます。このアプローチについては、「Timestamp writes for write hedging in Amazon DynamoDB」を参照してください。
- リクエストのタイムアウトと再試行動作の調整: クライアントから DynamoDB へのパスは多くのコンポーネントを通過しますが、それぞれが冗長性を念頭に置いて設計されています。以下の要素を検討します。
  - ネットワークの耐障害性
  - TCP パケットタイムアウト
  - DynamoDB の分散アーキテクチャ

  デフォルトの SDK 動作は、ほとんどのアプリケーションに最適化されています。ただし、フェイルファスト戦略を実装し、タイムアウト設定を調整することはできます。通常よりも大幅に時間がかかるリクエストは、最終的に成功する可能性が低くなります。フェイルファストして再試行することで、別のパスですぐに成功する可能性があります。これはリクエストヘッジに似ていますが、続行せずに最初のリクエストを終了します。

  低すぎるタイムアウト値の設定を避けます。タイムアウトが低すぎると、クライアントが誘発する可用性の問題が発生する可能性があります。例えば、50 ミリ秒のソケットタイムアウトは、シングルフロートラフィックの Amazon EC2 インスタンス帯域幅制限に近づく場合など、ネットワークレイテンシーの急増中に接続エラーを引き起こす可能性があります。タイムアウトを短くすることの利点をアプリケーションの可用性に対する潜在的なリスクと慎重に比較検討し、短いタイムアウトよりもヘッジを優先します。

  このトピックに関する役立つ説明は、「レイテンシーを考慮した Amazon DynamoDB アプリケーションのための AWS Java SDK HTTP リクエスト設定のチューニング」に記載されています。
- クライアントと DynamoDB エンドポイント間の距離を縮める:ユーザーが世界中に分散している場合は、グローバルテーブル - マルチアクティブ、マルチリージョンレプリケーション を使用することを検討してください。グローバルテーブルを使用すれば、そのテーブルを利用できるように指定した AWS リージョンにテーブルをレプリケートできます。データのコピーをエンドユーザーの近くに配置して、読み込みと書き込みのオペレーション中のネットワークレイテンシーを短縮でき

ます。DynamoDB グローバルテーブルを効果的に使用する方法の詳細については、「AWS 規範ガイダンス」の「Amazon DynamoDB グローバルテーブルの使用」を参照してください。

- キャッシュを使用する: トラフィックの読み取り量が多い場合は、以下のいずれかのキャッシュサービスの使用を検討してください。
  - DynamoDB Accelerator (DAX): フルマネージド型で可用性の高い、DynamoDB 用のインメモリキャッシュです。1 秒あたりのリクエスト数が数百万におよぶ場合であっても、最大 10 倍のパフォーマンスの (ミリセカンドからマイクロセカンドの範囲への) 向上を実現します。DAX の詳細については、「DynamoDB Accelerator (DAX) とインメモリアクセラレーション」を参照してください。
  - Amazon ElastiCache: フルマネージドのインメモリキャッシュサービス。DynamoDB と統合して、キャッシュアサイドパターンを使用して読み取りパフォーマンスを向上させることができます。詳細については、「AWS 規範ガイダンス」の「リードスルーキャッシュを使用した Amazon DynamoDB と Amazon ElastiCache の統合」を参照してください。

Amazon DynamoDB でのスロットリングのトラブルシューティング

DynamoDB は、全体的なサービスパフォーマンスとコスト管理の 2 つの主な目的でスロットリングを実装します。スロットリングは、消費レートがキャパシティを超えたときにパフォーマンスの低下を防ぐ意図的な保護手段として、あるいは最大スループットまたはサービスクォータ制限に達したときにコスト制御メカニズムとして機能します。スロットリングが発生すると、DynamoDB はリクエストがスロットリングされた理由と影響を受けたリソースに関する詳細情報を含む特定の例外を返します。スロットリングの各理由は、スロットリングイベントの頻度とパターンに関する追加のインサイトを提供する特定の CloudWatch メトリクスに対応しています。

次の図は、DynamoDB が保護スロットリングを実装する 4 つの主なシナリオを示しています。

1. キー範囲スループットを超過 (両方のモード)。

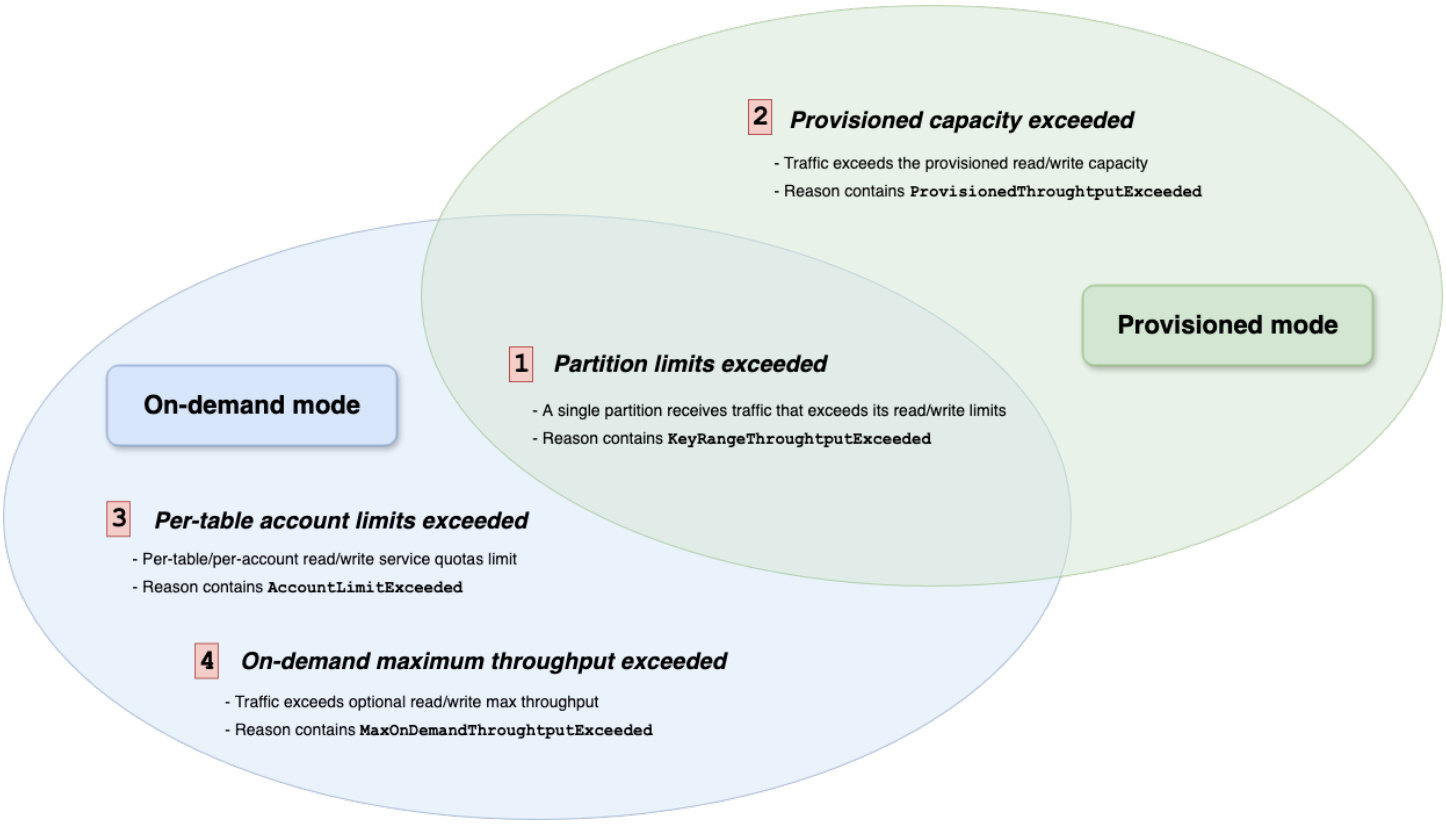
   特定のパーティションに向けられた消費量が、内部パーティションレベルのスループット制限を超えています。
2. プロビジョンドスループットを超過 (プロビジョンドモードの場合)。

   消費が、テーブルまたはグローバルセカンダリインデックス (GSI) に設定されているプロビジョンドキャパシティユニット (読み取りまたは書き込み) を超えています。
3. アカウントレベルのサービスクォータを超過 (オンデマンドモードの場合)。

消費により、テーブルまたは GSI は、現在の AWS リージョンの読み取り/書き込みスループットのテーブルごとのアカウントレベルのサービスクォータを超えます。これらのクォータはバックストップ保護として機能し、引き上げることができます。

4. オンデマンドの最大スループットを超過 (オンデマンドモードの場合)。

   消費が、テーブルまたは GSI に設定されている最大スループット制限を超えています。これらは、コスト管理の目的で特に設定する制限です。



このガイドは、DynamoDB でのスロットリングの理解と操作に役立つようにまとめられています。まず、このガイドは、診断フレームワークを通じてワークロードに影響を与える特定のタイプのスロットリングを特定するのに役立ちます。

次に、解決ガイドセクションでは、検出と分析のためにモニタリングする CloudWatch メトリクス、最適化の推奨ステップなど、スロットリングシナリオごとに具体的なガイダンスを提供します。この構造化されたアプローチに従うことで、スロットリングの根本原因をより適切に診断し、DynamoDB テーブルが効率的に動作するように適切なソリューションを実装できます。

まず、the section called “スロットリングの診断” に従って、ワークロードに影響を与えているスロットリングのタイプを特定し、推奨される解決戦略を実装する方法について学びます。

トピック

- スロットリングの診断
- DynamoDB スロットリング解決ガイド
- DynamoDB のグローバルセカンダリインデックス (GSI) の書き込みスロットリングとバックプレッシャーについて
- CloudWatch スロットリングメトリクス

スロットリングの診断

アプリケーションでスロットリングが発生すると、DynamoDB はこれらのイベントの診断に役立つ詳細な例外情報とターゲットを絞った CloudWatch メトリクスを提供します。

このセクションでは、DynamoDB アプリケーションのスロットリングイベントを理解するための体系的なアプローチを示します。スロットリング例外を解釈し、CloudWatch メトリクスと関連付けてより深いインサイトを得る方法、DynamoDB アプリケーションのスロットリングを減らす変更を理解する方法を示します。

スロットリングの例外について

DynamoDB がリクエストを調整すると、詳細な診断情報を含む特定の例外が返されます。例えば、Java では、これらには `ProvisionedThroughputExceededException`、`RequestLimitExceeded`、または `ThrottlingException` が含まれます。

各例外には、スロットリングを識別して理解するのに役立つ 2 つのキーフィールドを含む個々の `ThrottlingReason` のコレクションである `ThrottlingReasons` が含まれます。

- 理由 - `<ResourceType><OperationType><LimitType>` 形式に従った連結フィールド
- リソース ARN - 影響を受けるテーブルまたはインデックスの Amazon リソースネーム (ARN)

理由フィールドは、何が起こっているのかを正確に理解するのに役立つ一貫したパターンに従います。

- ResourceType (スロットリングの対象): `Table` または `Index`
- OperationType (オペレーションの種類): `Read` または `Write`
- LimitType (スロットリングが発生した理由):

- KeyRangeThroughputExceeded: これは、テーブルまたはインデックスをサポートする特定のパーティションが、パーティションごとの内部スループット制限を超える読み取りまたは書き込みキャパシティを消費した場合に発生します。
- ProvisionedThroughputExceeded: これは、読み取りまたは書き込みの消費率がプロビジョニングされた量を超えた場合に、プロビジョニングされたテーブルまたはグローバルセカンダリインデックスで発生します。
- AccountLimitExceeded: これは、オンデマンドテーブルまたはインデックスで、読み取りまたは書き込みの消費レートが、アカウントレベルで設定されたテーブルとそのインデックスの最大消費レートを超えた場合に発生します。このクォータの引き上げをリクエストできます。
- MaxOnDemandThroughputExceeded: これは、オンデマンドテーブルまたはインデックスで、読み取りまたは書き込みの消費レートが、テーブルまたはインデックスに設定されたユーザー提供の最大消費レートを超えた場合に発生します。この値は、アカウント制限までの任意の値に自分で上げることも、ユーザーが指定した制限がないことを示すために -1 に設定することもできます。

リソース ARN は、スロットリングされているテーブルまたはインデックスを正確に識別します。

- テーブルの場合: arn:aws:dynamodb:[region]:[account-id]:table/[table-name]
- インデックスの場合: arn:aws:dynamodb:[region]:[account-id]:table/[table-name]/index/[index-name]

完全なスロットリングの理由の例:

- TableReadProvisionedThroughputExceeded
- IndexWriteAccountLimitExceeded

これにより、スロットリングされているリソース、スロットリングの原因となったオペレーションのタイプ、スロットリングが発生した理由を正確に特定できます。

例外の例

例 1: GSI でプロビジョニングされたキャパシティを超過

```
{
    "ThrottlingReasons": [
        {
```

```
            "reason": "IndexWriteProvisionedThroughputExceeded",
            "resource": "arn:aws:dynamodb:us-west-2:123456789012:table/CustomerOrders/
index/OrderDateIndex"
        }
    ],
    "awsErrorDetails": {
        "errorCode": "ProvisionedThroughputExceeded",
        "errorMessage": "The level of configured provisioned throughput for the index
 was exceeded",
        "serviceName": "DynamoDB",
        "sdkHttpResponse": {
            "statusText": "Bad Request",
            "statusCode": 400
        }
    }
}
```

この例では、アプリケーションは ProvisionedThroughputExceededException を受け取り、その理由は IndexWriteProvisionedThroughputExceeded です。OrderDateIndex への書き込みは、書き込み消費量が GSI で設定されたプロビジョンド書き込みキャパシティを超えたため、スロットリングされています。

例 2: オンデマンドの最大スループットを超過

```
{
    "ThrottlingReasons": [
        {
            "reason": "TableReadMaxOnDemandThroughputExceeded",
            "resource": "arn:aws:dynamodb:us-east-1:123456789012:table/UserSessions"
        }
    ],
    "awsErrorDetails": {
        "errorMessage": "Throughput exceeds the maximum OnDemandThroughput configured
 on table or index",
        "errorCode": "ThrottlingException",
        "serviceName": "DynamoDB",
        "sdkHttpResponse": {
            "statusText": "Bad Request",
            "statusCode": 400
        }
    }
}
```

この例では、UserSessions テーブルからの読み取りは、テーブルで設定された最大オンデマンドスループット制限を超えているため、スロットリングされています。

DynamoDB スロットリング診断フレームワーク

アプリケーションでスロットリングが発生した場合は、以下の手順に従って問題を診断して解決します。

ステップ 1 - **ThrottlingReason** の詳細を分析する

1. 理由フィールドをチェックして、スロットリングの具体的な理由を特定します。理由には、スロットリングされたリソースのタイプ (テーブルまたはインデックス)、スロットリングイベントの原因となったオペレーションのタイプ (読み取りまたは書き込み)、および超過した制限タイプ (パーティション、プロビジョニングされたスループット、アカウント制限) が詳しく記載されています。
2. resourceArn フィールドをチェックして、スロットリングされているリソース (テーブルまたは GSI) を特定します。
3. この組み合わせ情報を使用して、スロットリングの問題の完全なコンテキストを理解します。

   例えば、スロットリングの理由 ProvisionedThroughputExceededException で次の例外 TableWriteKeyRangeThroughputExceeded が発生するシナリオを考えてみます。影響を受ける resourceARN は arn:aws:dynamodb:us-west-2:123456789012:table/CustomerOrders です。

   この組み合わせにより、CustomerOrders テーブルへの書き込みオペレーションがスロットリングされていることが通知されます。スロットリングはパーティションレベルで発生しています (テーブルレベルではなく、TableWriteProvisionedThroughputExceeded と表示されます)。根本原因は、特定のパーティションキー値または範囲の最大スループットキャパシティを超えており、ホットパーティションの問題を示していることです。

   例外要素間のこの関係を理解することで、適切な緩和策を実装できます。この場合、テーブルの全体的なプロビジョンドキャパシティを増やすのではなく、ホットパーティションに対処します。

ステップ 2 - 関連する CloudWatch メトリクスを特定して分析する

1. メトリクスを特定する: DynamoDB の各スロットリングの理由は、スロットリングイベントを追跡および分析するためにモニタリングできる特定の CloudWatch メトリクスに直接対応します。スロットリングの理由から適切な CloudWatch メトリクス名を体系的に取得できます。

2. このリファレンステーブルを使用して、スロットリングの理由と対応する CloudWatch メトリクスを一致させます。

   完全なスロットリングの理由と CloudWatch メトリクスリファレンス

Category	スロットリングの理由	プライマリ CloudWatch メトリクス
プロビジョンドキャパシティの超過	TableReadProvisionedThroughputExceeded	ReadProvisionedThroughputThrottleEvents
	TableWriteProvisionedThroughputExceeded	WriteProvisionedThroughputThrottleEvents
	IndexReadProvisionedThroughputExceeded	ReadProvisionedThroughputThrottleEvents (GSI)
	IndexWriteProvisionedThroughputExceeded	WriteProvisionedThroughputThrottleEvents (GSI)
パーティション制限を超過	TableReadKeyRangeThroughputExceeded	ReadKeyRangeThroughputThrottleEvents
	TableWriteKeyRangeThroughputExceeded	WriteKeyRangeThroughputThrottleEvents
	IndexReadKeyRangeThroughputExceeded	ReadKeyRangeThroughputThrottleEvents (GSI)
	IndexWriteKeyRangeThroughputExceeded	WriteKeyRangeThroughputThrottleEvents (GSI)

Category	スロットリングの理由	プライマリ CloudWatch メトリクス
オンデマンドの最大を超過	TableReadMaxOnDemandThroughputExceeded	ReadMaxOnDemandThroughputThrottleEvents
	TableWriteMaxOnDemandThroughputExceeded	WriteMaxOnDemandThroughputThrottleEvents
	IndexReadMaxOnDemandThroughputExceeded	ReadMaxOnDemandThroughputThrottleEvents (GSI)
	IndexWriteMaxOnDemandThroughputExceeded	WriteMaxOnDemandThroughputThrottleEvents (GSI)
アカウント制限の超過	TableReadAccountLimitExceeded	ReadAccountLimitThrottleEvents
	TableWriteAccountLimitExceeded	WriteAccountLimitThrottleEvents
	IndexReadAccountLimitExceeded	ReadAccountLimitThrottleEvents (GSIs)
	IndexWriteAccountLimitExceeded	WriteAccountLimitThrottleEvents (GSIs)

例えば、`IndexWriteProvisionedThroughputExceeded` を受け取った場合は、少なくとも `ResourceArn` で識別された特定のインデックスの

WriteProvisionedThroughputThrottleEvents CloudWatchメトリクスをモニタリングする必要があります。

3. CloudWatch でこれらのメトリクスをモニタリングして、スロットリングイベントの頻度とタイミングを理解し、読み取りスロットリングと書き込みスロットリングを区別し、スロットリングの増加時に時間パターンを特定し、キャパシティ使用率の傾向を追跡します。

   DynamoDB は、各テーブルとグローバルセカンダリインデックスの詳細なメトリクスを発行します。メトリクス (ReadThrottleEvents、WriteThrottleEvents、および ThrottledRequests) は、テーブルとそのインデックス全体のすべてのスロットリングイベントを集計します。

ステップ 3 - CloudWatch Contributor Insights を使用してスロットリングされたキーと高アクセス率を特定する (パーティション関連のスロットリング用)

ステップ 1 でパーティション関連の問題 (KeyRangeThroughputExceeded エラーなど) を特定した場合、CloudWatch Contributor Insights for DynamoDB は、トラフィックを駆動し、テーブルまたはインデックスでスロットリングイベントが発生している特定のキーを診断するのに役立ちます。

1. ResourceARN に基づいてスロットリングされたテーブルまたはインデックスに対して CloudWatch Contributor Insights を有効にします。

   スロットリングされたキーモードを選択すると、最もスロットリングされたキーのみに集中できます。このモードは、スロットリングが発生したときにのみイベントを処理するため、継続的なモニタリングに最適です。または、アクセスおよびスロットリングされたキーモードは、最もアクセスされたキーのパターンを探すのに役立ちます。

2. レポートを分析して、問題のあるパターンを特定します。アクセス率またはスロットリング率が不釣り合いに高いキーを探し、スロットリングとトラフィックパターンを関連付けます。Contributor Insights グラフと DynamoDB CloudWatch メトリクスを組み合わせた統合ダッシュボードを作成できます。

3. 個々のキーが過度に高いアクセス率またはスロットリング率を示していない場合、単一のホットキーではなく、複数のキーにまたがる無制限のシーケンシャルアクセスによって、キー範囲のスロットリングが発生する場合があります。継続的な Scan オペレーションを実行する場合、読み取りトラフィックのレートを制限して、キー範囲のパーティションあたりの最大読み取りユニット数を 1 秒あたり 3,000 未満に保ちます。全体的な読み取りスループットを向上させるには、並列 (セグメント化) スキャンを使用して、読み取りトラフィックをパーティション間でより均一に分散します。同様に、Scan によって返されるのと同じキー順序で項目を書き込むと、一度に 1 つのキー範囲に書き込みが集中し、その範囲への書き込みレートが 1 秒あたり 1,000 書き込みユ

ニットを超えると、キー範囲のスロットリングが発生する可能性があります。その制限を下回るように、1 つのキー範囲への書き込みレートを制限します。

CloudWatch Contributor Insights の有効化と使用に関する詳細については、「CloudWatch Contributor Insights for DynamoDB を使用する」を参照してください。

ステップ 4 - 適切なソリューションを決定する

スロットリングの特定の原因を診断したら、特定のコンテキストに基づいて推奨されるソリューションを実装します。適切なソリューションは、スロットリングシナリオ、テーブルのキャパシティモード、テーブルとキーの設計決定、アクセスパターンとクエリ効率、グローバルおよびセカンダリインデックスの設定、および全体的なシステムアーキテクチャと統合ポイントなど、複数の要因によって異なります。

特定のスロットリングシナリオに対処するための詳細な解決策については、「the section called “解決ガイド”」セクションを参照してください。このリソースは、特定のスロットリングの理由とキャパシティモードの設定に合わせてカスタマイズされたターゲットを絞った修復戦略を提供します。

ステップ 5 - 進捗状況をモニタリングする

1. スロットリングシナリオに対応する CloudWatch メトリクスを追跡します。
2. スロットリングイベントの減少を観察することで、緩和戦略が有効であることを確認します。

DynamoDB スロットリング解決ガイド

このセクションでは、DynamoDB が返す可能性のある特定のスロットリングの理由ごとに、ターゲットを絞った解決ガイダンスを提供します。各エントリには、ベストプラクティスとモニタリングする対応する CloudWatch メトリクスに基づいて推奨される解決アプローチが含まれます。

DynamoDB は、4 つの主要なカテゴリにまたがる 16 の異なるスロットリングの理由を実装しています。アプリケーションの例外のスロットリングの理由を使用して、関連するガイダンスに直接移動します。

キー範囲スループットの超過 (ホットパーティション)

これらのスロットリングの理由は、個々のパーティションがスループット制限を超え、プロビジョニングモードとオンデマンドモードの両方に影響を与える場合に発生します。

- TableReadKeyRangeThroughputExceeded

- TableWriteKeyRangeThroughputExceeded
- IndexReadKeyRangeThroughputExceeded
- IndexWriteKeyRangeThroughputExceeded

プロビジョンドスループットの超過

これらのスロットリングの理由は、消費率がプロビジョニングモードでプロビジョニングされた容量制限を超えた場合に発生します。

- TableReadProvisionedThroughputExceeded
- TableWriteProvisionedThroughputExceeded
- IndexReadProvisionedThroughputExceeded
- IndexWriteProvisionedThroughputExceeded

アカウント制限の超過

これらのスロットリングの理由は、消費率が AWS リージョンのアカウントレベルのスループットクォータを超えた場合に発生します。

- TableReadAccountLimitExceeded
- TableWriteAccountLimitExceeded
- IndexReadAccountLimitExceeded
- IndexWriteAccountLimitExceeded

オンデマンドの最大スループットの超過

これらのスロットリングの理由は、オンデマンドモードで消費レートが設定された最大スループット制限を超えた場合に発生します。

- TableReadMaxOnDemandThroughputExceeded
- TableWriteMaxOnDemandThroughputExceeded
- IndexReadMaxOnDemandThroughputExceeded
- IndexWriteMaxOnDemandThroughputExceeded

1- キー範囲スループットの超過 (ホットパーティション)

Amazon DynamoDB は、テーブルとグローバルセカンダリインデックス (GSI) の両方について、パーティションレベルで特定のスループット制限を適用します。各パーティションには、1 秒あたりの読み込みキャパシティーユニット (RCU) と書き込みキャパシティーユニット (WCU) の最大数があります。パーティションがこれらの制限を超える集中トラフィックを受信すると、スロットリングが発生し、他のオペレーションが十分に活用されないままになり、「ホットパーティション」が発生する可能性があります。DynamoDB のパーティションレベルのスロットリングは、読み取りと書き込みに対して独立して動作します。パーティションは、書き込みが正常に継続している間に読み取りをスロットリングするか、その逆を行います。このスロットリングは、テーブルまたは GSI の全体的なキャパシティが十分であっても発生する可能性があります。詳細を確認するトピック

- DynamoDB パーティションの制限とホットパーティション防止に対処する効果的なパーティションキー設計については、「DynamoDB でパーティションキーを効率的に設計し、使用するためのベストプラクティス」を参照してください。
- 一般的なパーティションの概念とデータ分散については、「DynamoDB のパーティション」を参照してください。
- パーティションキーとスループットを管理するためのその他のガイダンスと実際のシナリオについては、「the section called “その他のリソース”」を参照してください。

個々のパーティションがスループット制限を超えると、DynamoDB はスロットリング例外で `KeyRangeThroughputExceeded` スロットリングの理由タイプを返します。この情報により、パーティションのトラフィックが多くなり、どのオペレーションタイプ (読み取りまたは書き込み) が問題の原因になっているかがわかります。

キー範囲スループット超過の緩和策

このセクションでは、パーティションレベルのスロットリングシナリオの解決ガイダンスを提供します。このガイドを使用する前に、アプリケーションの例外処理から特定のスロットリングの理由を特定し、影響を受けるリソースの Amazon リソースネーム (ARN) を決定していることを確認します。スロットリングの理由の取得とスロットリングされたリソースの識別については、「the section called “診断フレームワーク”」を参照してください。

特定のスロットリングシナリオに進む前に、まず問題が自動的に解決されるかどうかを確認します。

- DynamoDB は、多くの場合、自動 split-for-heat メカニズムを通じてホットパーティションに適応します。短期間後に停止するスロットリングイベントが表示される場合は、ホットパーティションを分割してテーブルが既に適応している可能性があります。パーティションが分割されると、新し

い各パーティションはキースペースの小さなセクションを処理するため、負荷をより均等に分散できます。多くの場合、DynamoDB は自動的に問題を解決しているため、それ以上のアクションは必要ありません。

split-for-heat メカニズムの詳細については、「the section called “その他のリソース”」を参照してください。

スロットリングが持続する場合は、以下の特定のスロットリングシナリオを参照して、ターゲットを絞った修復オプションを確認します。

- the section called “TableReadKeyRangeThroughputExceeded”
- the section called “TableWriteKeyRangeThroughputExceeded”
- the section called “IndexReadKeyRangeThroughputExceeded”
- the section called “IndexWriteKeyRangeThroughputExceeded”

TableReadKeyRangeThroughputExceeded

この問題が発生した場合

DynamoDB テーブル内の 1 つ以上のパーティションの消費レートが、パーティションの読み取りスループット制限を超えています。このスロットリングは、テーブルの合計プロビジョンドキャパシティに関係なく発生し、プロビジョンドテーブルとオンデマンドテーブルの両方に影響します。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

プロビジョニングモードとオンデマンドモードの両方の場合:

- 事前ウォーミングキャパシティ: スロットリングが持続する場合は、テーブルの the section called “ウォームスループット” キャパシティが制限されているかどうかを確認します。予想されるトラフィックの増加に備えて、ウォームスループットを使用するか、事前に読み込みプロビジョンドキャパシティを増やします。ウォームスループットを上げると、スロットリングが発生する前に突然のトラフィックのスパイクを処理するテーブルの機能が向上します。時間の経過とともに、実際のスループットがウォームスループットレベルに一貫して近づくと、DynamoDB は観測された使用パターンに基づいて使用率が高いパーティションを分割することがあります。

- ホットキーを特定する: テーブルが自動的に解決せず、ウォームスループットが高い場合、またはウォームスループットを上げても効果がない場合は、特定のホットキーを特定する必要があります。the section called “CloudWatch Contributor Insights を使用したホットキーの識別” を使用して、特定のパーティションキー値がホットかどうかを判断します。これは、緩和作業を効果的にターゲットにするための最初のステップです。特にローリングホットパーティション (時間の経過とともに異なるパーティションがホットになる) や、スキャンなどのオペレーションによってスロットリングがトリガーされる場合、識別が必ずしも簡単であるとは限りません。このような複雑なシナリオでは、アプリケーションのアクセスパターンを分析し、スロットリングイベントのタイミングと関連付ける必要がある場合があります。
- ユースケースに応じて、結果整合性のある読み込みの使用を検討する: 強力な整合性のある読み込みから結果整合性のある読み込みに切り替えると、RCU の半分が消費され、有効な読み込み容量がすぐに 2 倍になります。結果整合性のある読み込みを実装して読み込みキャパシティの消費量を削減するベストプラクティスについては、「the section called “DynamoDB の読み取り整合性”」を参照してください。
- パーティションキー設計の改善: 長期ソリューションとして、the section called “パーティションキー設計の改善” を改善し、パーティション間でアクセスをより均等に分散することを検討します。このアプローチは、多くの場合、根本原因に対処することで、ホットパーティションの問題に対する最も包括的な解決策を提供します。ただし、重要な移行の課題を伴うため、慎重な計画が必要です。

TableWriteKeyRangeThroughputExceeded

この問題が発生した場合

DynamoDB テーブル内の 1 つ以上のパーティションの消費レートが、パーティションの書き込みスループット制限を超えています。このスロットリングは、テーブルの合計プロビジョンドキャパシティに関係なく発生し、プロビジョンドテーブルとオンデマンドテーブルの両方に影響します。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

プロビジョニングモードとオンデマンドモードの両方の場合:

- 事前ウォーミングキャパシティ: スロットリングが持続する場合は、テーブルの DynamoDB ウォームスループットについて キャパシティが制限されているかどうかを確認します。予想され

るトラフィックの増加に備えて、ウォームスループットを使用するか、事前に書き込みプロビジョンドキャパシティを増やします。ウォームスループットを上げると、スロットリングが発生する前に突然のトラフィックのスパイクを処理するテーブルの機能が向上します。時間の経過とともに、実際のスループットがウォームスループットレベルに一貫して近づくと、DynamoDB は観測された使用パターンに基づいて使用率が高いパーティションを分割することがあります。

- ホットキーを特定する: テーブルが自動的に解決せず、ウォームスループットが高い場合、またはウォームスループットを上げても効果がない場合は、特定のホットキーを特定する必要があります。the section called “CloudWatch Contributor Insights を使用したホットキーの識別” を使用して、特定のパーティションキー値がホットかどうかを判断します。これは、緩和作業を効果的にターゲットにするための最初のステップです。以下の一般的なパターンを検討します。
  - スロットリングデータに同じパーティションキーが頻繁に表示される場合は、集中ホットキーであることを示しています。
  - キーが繰り返し表示されないが、順序付けられた方法でデータを書き込んでいる場合 (シーケンシャルタイムスタンプやキースペースの順序に従うスキャンベースのオペレーションなど)、書き込みがキースペースを通過するにつれて、時間の経過とともに異なるキーがホットになるローリングホットパーティションがある可能性があります。

  書き込みスロットリングは、`BatchWriteItem` や複数の項目に同時に影響を与えるトランザクションなどの操作でも発生する可能性があることに注意してください。`BatchWriteItem` リクエスト内の個々の項目がスロットリングされると、DynamoDB はこれらのスロットリングエラーをアプリケーションコードに伝達しません。代わりに、DynamoDB はレスポンス内の未処理の項目に関する情報を返します。アプリケーションは、それらの特定の項目を再試行して処理する必要があります。トランザクションの場合、いずれかの項目でスロットリングが発生すると、オペレーション全体が `TransactionCanceledException` で失敗します。このような複雑なシナリオでは、アプリケーションの書き込みパターンとデータインジェストワークフローを分析し、スロットリングイベントのタイミングと関連付けて、適切な再試行処理戦略を実装する必要がある場合があります。
- パーティションキー設計の改善: 長期ソリューションとして、the section called “パーティションキー設計の改善” を改善し、パーティション間でアクセスをより均等に分散することを検討します。このアプローチは、多くの場合、根本原因に対処することで、ホットパーティションの問題に対する最も包括的な解決策を提供します。ただし、重要な移行の課題を伴うため、慎重な計画が必要です。

IndexReadKeyRangeThroughputExceeded

この問題が発生した場合

DynamoDB GSI 内の 1 つ以上のパーティションの消費レートが、パーティションの読み取りスループット制限を超えています。このスロットリングは、GSI の合計プロビジョンドキャパシティに関係なく発生し、プロビジョンドテーブルとオンデマンドテーブルの両方に影響します。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- 事前ウォーミングキャパシティ: スロットリングが持続する場合は、GSI の DynamoDB ウォームスループットについて キャパシティが制限されているかどうかを確認します。予想されるトラフィックの増加に備えて、ウォームスループットを使用するか、事前に読み込みプロビジョンドキャパシティを増やします。ウォームスループットを上げると、スロットリングが発生する前に突然のトラフィックのスパイクを処理する GSI の機能が向上します。時間の経過とともに、実際のスループットがウォームスループットレベルに一貫して近づくと、DynamoDB は観測された使用パターンに基づいて使用率が高いパーティションを分割することがあります。
- ホットキーを特定する: GSI が自動的に解決せず、ウォームスループットが高い場合、またはウォームスループットを上げても効果がない場合は、特定のホットキーを特定する必要があります。the section called “CloudWatch Contributor Insights を使用したホットキーの識別” を使用して、特定のパーティションキー値がホットかどうかを判断します。これは、緩和作業を効果的にターゲットにするための最初のステップです。GSI の場合、パーティションキーの分散は基本テーブルと大きく異なり、異なるホットキーパターンが発生する可能性があることに注意してください。
- GSI パーティションキーの再設計: GSI キー設計が、少数のパーティションに読み取りを集中させる人工的なホットスポット (ステータスフラグ、日付のみのキー、ブール属性など) を作成しているかどうかを検討します。低カーディナリティ属性を高カーディナリティ属性と組み合わせる複合キーを使用すること (単に「ACTIVE」ではなく「ACTIVE#customer123」など)、または GSI ディストリビューションに影響する基本テーブル項目に the section called “書き込みシャーディング” 手法を適用して複数のパーティションに書き込みを分散することを検討します。シャードデータをクエリするには結果を集約するための追加のアプリケーションロジックが必要ですが、このアプローチではアクセスパターンをより均等に分散することでスロットリングを防止します。

IndexWriteKeyRangeThroughputExceeded

この問題が発生した場合

DynamoDB GSI 内の 1 つ以上のパーティションの消費レートが、パーティションの書き込みスループット制限を超えています。このスロットリングは、GSI の合計プロビジョンドキャパシティに関係なく発生し、プロビジョンドテーブルとオンデマンドテーブルの両方に影響します。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- GSI パーティションキーの再設計: GSI パーティションキー設計を確認して、書き込みを均等に分散するのに十分なカーディナリティ (一意性) があることを確認します。GSI 書き込みスロットリングの一般的な原因は、GSI パーティションキーとして低カーディナリティ属性を使用することです (可能な値が少ないステータスフラグなど)。基本テーブルに適切に分散されたパーティションキーがある場合でも、パーティションキーが少数の値に書き込みが集中すると、GSI でホットパーティションが発生する可能性があります。例えば、項目の 80% に「ACTIVE」というステータスがある場合、ステータスベースの GSI に重大なホットパーティションが作成されます。低カーディナリティ属性を高カーディナリティ属性と組み合わせる複合キーを使用すること (単に「ACTIVE」ではなく「ACTIVE#customer123」など)、または GSI ディストリビューションに影響する基本テーブル項目に the section called “書き込みシャーディング” 手法を適用して複数のパーティションに書き込みを分散することを検討します。シャードデータをクエリするには結果を集約するための追加のアプリケーションロジックが必要ですが、このアプローチではアクセスパターンをより均等に分散することでスロットリングを防止します。
- 事前ウォーミングキャパシティ: GSI が DynamoDB ウォームスループットについて キャパシティによって制限されているかどうかを確認します。予想されるトラフィックの増加に備えて、ウォームスループットを使用するか、事前に書き込みプロビジョンドキャパシティを増やします。ウォームスループットを上げると、スロットリングが発生する前に突然のトラフィックのスパイクを処理する GSI の機能が向上します。時間の経過とともに、実際のスループットがウォームスループットレベルに一貫して近づくと、DynamoDB は観測された使用パターンに基づいて使用率が高いパーティションを分割することがあります。
- GSI 射影の最適化: GSI への書き込み量を減らす the section called “GSI 射影の最適化” 手法を適用します。属性の数を減らすと、各 GSI 更新で消費される書き込みキャパシティが大幅に減少します。

一般的な診断とモニタリング

パーティションレベルのスロットリングをトラブルシューティングする場合、複数の CloudWatch メトリクスがホットパーティションを特定し、根本原因を確認するのに役立ちます。

重要な CloudWatch メトリクス

これらの主要なメトリクスをモニタリングして、パーティションレベルのスロットリングを診断します。

- パーティションレベルのスロットリングイベント: `ReadKeyRangeThroughputThrottleEvents` および `WriteKeyRangeThroughputThrottleEvents` は、個々のパーティションがスループット制限を超えた場合を追跡します。`ReadThrottleEvents` および `WriteThrottleEvents` は、読み取りまたは書き込みリクエストがプロビジョンドキャパシティを超えた場合を追跡します。
- キャパシティ消費: `ConsumedReadCapacityUnits` および `ConsumedWriteCapacityUnits` は全体的な使用パターンを表示します。

解決手順

CloudWatch Contributor Insights を使用したホットキーの識別

この手順を使用して、スロットリングの原因となっているパーティションキーを特定します。

1. テーブルまたは GSI で CloudWatch Contributor Insights を有効にして、最もスロットリングされたキーを追跡します。スロットリングされたキーモードを使用して、CloudWatch Contributor Insights をリアルタイムスロットリングアラートで継続的に有効にすることを検討します。このモードは、スロットリングが発生したときにのみイベントを処理することで、スロットリングされたリクエストのみに焦点を当てます。このターゲットを絞ったモニタリングは、スロットリングの問題を継続的に可視化するためのコスト効率の高い方法です。
2. ホットパーティションの問題の原因となっているキーを特定します。
3. (フルアクセスおよびスロットリングされたキーが有効になっている場合) アクセスパターンを経時的に分析して、ホットキーが一貫しているか、特定の期間に発生するかを判断します。

パーティションキー設計の改善

このアプローチは、パーティション間でトラフィックをより適切に分散するようにテーブルスキーマを変更できる場合に使用します。可能な場合、これはホットパーティションの問題に対する最も効果

的な長期的なソリューションです。パーティションキーの設計は、最初のテーブル設計段階で慎重に検討することをお勧めします。

パーティションキーの再設計は、アプリケーションエコシステム全体に影響を与えるデータモデルへの根本的な変更を表します。このアプローチを進める前に、以下の重要な制限事項を慎重に検討してください。

- データ移行の複雑さ: パーティションキーを再設計するには、既存のすべてのデータを移行する必要があります。大規模なテーブルでは、リソースを大量に消費し、時間がかかる場合があります。
- アプリケーションコードの変更: テーブルを読み書きするすべてのアプリケーションコードは、新しいキー構造を使用するように更新する必要があります。
- 本番稼働への影響: 新しいキー設計に移行するには、移行中にダウンタイムや複雑な二重書き込み戦略が必要になることがよくあります。

パーティションキー設計に関する包括的なガイダンスと原則については、「the section called “パーティションキーの設計”」および「the section called “ワークロードを分散する”」を参照してください。

GSI 射影の最適化

アプリケーションのクエリパターンを確認して、GSI をクエリするときに使用できる属性を正確に決定し、射影をそれらの属性のみに制限します。GSI に射影されていない属性を更新すると、その GSI で書き込みオペレーションは行われず、更新中の書き込みスループットの消費量が減少します。このターゲット射影戦略は、アプリケーションのクエリ要件をサポートしながら、パフォーマンスとコストの両方を最適化します。属性の射影数を減らすと、書き込みキャパシティの消費量は減りますが、基本テーブルの読み取りがさらに必要になる場合があります。

効率的な射影戦略の詳細については、「DynamoDB でセカンダリインデックスを使用するためのベストプラクティス」を参照してください。

その他のリソース

次のブログ投稿では、このガイドで説明されている概念の実践的な例と実践的な詳細について説明します。

- DynamoDB のスケーリングとホットパーティションの管理に関する実践的なガイダンスについては、「Part 1: Scaling DynamoDB - How partitions, hot keys, and split for heat impact performance」を参照してください。

- DynamoDB の split-for-heat メカニズムの仕組み、その利点、実装の詳細については、「Part 3: Summary and best practices」を参照してください。
- 詳細な書き込みシャーディング戦略については、「the section called “書き込みシャーディング”」を参照してください。

2- プロビジョンドスループットの超過

プロビジョンドキャパシティのスロットリングは、アプリケーションの消費率が、テーブルまたはグローバルセカンダリインデックス用に設定された読み取りまたは書き込みキャパシティユニット (RCU/WCU) を超えた場合に発生します。DynamoDB はトラフィックの急増を処理するためのバーストキャパシティを提供しますが、プロビジョニングされた制限を超える持続的なリクエストはスロットリングを引き起こします。この場合、DynamoDB はスロットリング例外で `ProvisionedThroughputExceeded` スロットリングの理由タイプを返します。この理由は、問題が読み取り操作か書き込み操作か、基本テーブルまたはグローバルセカンダリインデックスに影響するかを識別します。

Auto Scaling が有効になっているかどうかにかかわらず、スロットリングが発生する可能性があります。Auto Scaling は消費量の増加に適応しますが、すぐには応答せず、設定した最大キャパシティ制限によって制約されます。つまり、突然のトラフィックの急増時や、消費が Auto Scaling の最大制限を超えた場合でも、スロットリングが発生する可能性があります。

プロビジョンドスループット超過の緩和策

このセクションでは、プロビジョンドキャパシティのスロットリングシナリオの解決ガイダンスを提供します。このガイドを使用する前に、アプリケーションの例外処理から特定のスロットリングの理由を特定し、影響を受けるリソースの Amazon リソースネーム (ARN) を決定していることを確認してください。スロットリングの理由の取得とスロットリングされたリソースの識別については、「the section called “診断フレームワーク”」を参照してください。

特定のスロットリングシナリオに進む前に、まずスロットリングが解決が必要な問題であるかどうかを検討します。

- 時折のスロットリングは正常であり、適切に最適化された DynamoDB アプリケーションで予期されます。スロットリングとは、プロビジョニングした内容の 100% を消費していることを意味します。アプリケーションが再試行でスロットリングを適切に処理し、全体的なパフォーマンスが要件を満たしている場合、スロットリングは即時のアクションを必要としない場合があります。

- ただし、スロットリングによってクライアント側の許容できないレイテンシーが発生している場合、ユーザーエクスペリエンスが低下している場合、または重要なオペレーションがタイムリーに完了しない場合は、以下の緩和オプションに進みます。

スロットリングの問題に対処する必要がある場合は、まずスロットリングの原因が以下であるかどうかを判断します。

- 一時的なトラフィックのスパイク: プロビジョニングされたキャパシティを超えるが持続しないトラフィックの短期間の増加。これには、継続的な高トラフィックとは異なる戦略が必要です。
- 継続的な高トラフィック: プロビジョニングされたキャパシティを一貫して超える持続的なワークロード。

トラフィックのスパイクについては、the section called “その他のリソース” の「Handle traffic spikes with Amazon DynamoDB provisioned capacity」ブログの戦略を検討してください。

継続的な高トラフィックの場合は、以下のキャパシティ調整オプションを検討します。

- the section called “TableReadProvisionedThroughputExceeded”
- the section called “TableWriteProvisionedThroughputExceeded”
- the section called “IndexReadProvisionedThroughputExceeded”
- the section called “IndexWriteProvisionedThroughputExceeded”

TableReadProvisionedThroughputExceeded

この問題が発生した場合

アプリケーションの読み込み消費レートが、テーブルに設定されたプロビジョニングされた読み込みキャパシティユニット (RCU) を超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

読み取りキャパシティスロットリングを解決するには、次の戦略を検討します。

- オンデマンドキャパシティモードに切り替える: トラフィックの急増によるスロットリングが頻繁に発生する場合は、テーブルをオンデマンドに切り替えることを検討します。オンデマンドにより、プロビジョニングの懸念がなくなり、ワークロードに合わせて自動的にスケーリングされます。

- プロビジョンドモードのままで、Auto Scaling が有効になっていない場合:
  - テーブルの読み取りキャパシティを増やすことを検討します。
  - テーブルの読み取りキャパシティに対して Auto Scaling を有効にします。
- Auto Scaling が有効になっている場合 (コンソールで作成されたテーブルのデフォルト):
  - テーブルの読み取り Auto Scaling パラメータを最適化します。

TableWriteProvisionedThroughputExceeded

この問題が発生した場合

アプリケーションの書き込み消費量が、テーブルに設定されたプロビジョニングされた書き込みキャパシティユニット (WCU) を超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

書き込みキャパシティスロットリングを解決するには、次の戦略を検討します。

- オンデマンドキャパシティモードに切り替える: トラフィックの急増によるスロットリングが頻繁に発生する場合は、テーブルをオンデマンドに切り替えることを検討します。オンデマンドにより、プロビジョニングの懸念がなくなり、ワークロードに合わせて自動的にスケーリングされます。
- プロビジョンドモードのままで、Auto Scaling が有効になっていない場合:
  - テーブルの書き込みキャパシティを増やすことを検討します。
  - テーブルの書き込みキャパシティに対して Auto Scaling を有効にします。
- Auto Scaling が有効になっている場合 (コンソールで作成されたテーブルのデフォルト):
  - テーブルの書き込み Auto Scaling パラメータを最適化します。

IndexReadProvisionedThroughputExceeded

この問題が発生した場合

グローバルセカンダリインデックス (GSI) の読み取り消費量が GSI のプロビジョニングされた読み込みキャパシティユニット (RCU) を超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

GSI 読み取りキャパシティスロットリングを解決するには、次の戦略を検討します。

- オンデマンドキャパシティモードに切り替える: トラフィックの急増によるスロットリングが頻繁に発生する場合は、基本テーブルをオンデマンドに切り替えることを検討します。オンデマンドにより、プロビジョニングの懸念がなくなり、ワークロードに合わせて自動的にスケーリングされます。
- プロビジョンドモードのままで、Auto Scaling が有効になっていない場合:
  - GSI 読み取りキャパシティを増やすことを検討します。
  - GSI の読み取りキャパシティに対して Auto Scaling を有効にします。
- Auto Scaling が有効になっている場合 (コンソールで作成されたテーブルのデフォルト):
  - GSI の読み取り Auto Scaling パラメータを最適化します。

IndexWriteProvisionedThroughputExceeded

この問題が発生した場合

基本テーブルの項目の更新により、GSI のプロビジョニングされた書き込みキャパシティを超える GSI への書き込みがトリガーされます。これにより、基本テーブルの書き込み時にバックプレッシャースロットリングが発生します。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

GSI 書き込みキャパシティスロットリングを解決するには、次の戦略を検討します。

- オンデマンドキャパシティモードに切り替える: トラフィックの急増によるスロットリングが頻繁に発生する場合は、基本テーブルをオンデマンドに切り替えることを検討します。オンデマンドにより、プロビジョニングの懸念がなくなり、ワークロードに合わせて自動的にスケーリングされます。
- プロビジョンドモードのままで、Auto Scaling が有効になっていない場合:
  - GSI 書き込みキャパシティを増やすことを検討します。
  - GSI の書き込みキャパシティに対して Auto Scaling を有効にします。
- Auto Scaling が有効になっている場合 (コンソールで作成されたテーブルのデフォルト):
  - GSI の書き込み Auto Scaling パラメータを最適化します。

一般的な診断とモニタリング

スループットエラーをトラブルシューティングする場合、いくつかの CloudWatch メトリクスが根本原因の特定に役立ちます。

重要な CloudWatch メトリクス

これらの主要なメトリクスをモニタリングして、プロビジョンドキャパシティのスロットリングを診断します。

- スロットリングイベント: `ReadProvisionedThroughputThrottleEvents` と `WriteProvisionedThroughputThrottleEvents` は、この理由でリクエストがスロットリングされたタイミングを追跡します。`ReadThrottleEvents` と `WriteThrottleEvents` は、読み取りまたは書き込みリクエストがプロビジョンドキャパシティを超えたタイミングを追跡します。
- キャパシティの消費: `ConsumedReadCapacityUnits` と `ConsumedWriteCapacityUnits` は実際の使用量を表示します。
- プロビジョンドキャパシティ: `ProvisionedReadCapacityUnits` および `ProvisionedWriteCapacityUnits` は、設定された制限を表示します。

解決手順

テーブルスループットキャパシティの増加

Auto Scaling が有効になっておらず、すぐにキャパシティを増やす必要がある場合は、この手順を使用します。

1. DynamoDB コンソール、AWS CLI、または SDK を使用して、テーブルのプロビジョンドキャパシティを更新します。
   - 読み込みキャパシティの場合: DynamoDB がリクエストを調整する前に 1 秒あたりに消費される強力な整合性のある読み込みの最大数を指定する `ReadCapacityUnits` パラメータを増やします。
   - 書き込みキャパシティの場合: DynamoDB がリクエストを調整する前に 1 秒あたりに消費される書き込みの最大数を指定する `WriteCapacityUnits` パラメータを増やします。
2. 新しいキャパシティ設定がテーブルごとのスループットクォータを超えていないこと、およびアカウントの総消費量がリージョンのアカウントごとのスループットクォータを下回っていることを確認します。これらの制限に近づいている場合は、代わりにオンデマンドキャパシティモードに切り替えることを検討します。

テーブルまたは GSI の読み取りまたは書き込みキャパシティを調整するようにテーブル Auto Scaling を設定する

トラフィックパターンに基づいて読み取りまたは書き込みキャパシティを自動的に調整するように DynamoDB Auto Scaling を設定します。Auto Scaling は、テーブルと GSI の両方で個別に設定でき、読み取りキャパシティユニットと書き込みキャパシティユニットを個別に制御できます。

1. テーブルまたは GSI の読み取りキャパシティ、書き込みキャパシティ、またはその両方に対して Auto Scaling を有効にします。
2. トラフィックのスパイクのヘッドルームでターゲット使用率を設定します。

> **Note**
>
> ターゲット使用率が低いほど、コストとスケーリング頻度が増加します。ターゲットが 40% 未満の場合、オーバープロビジョニングが発生する可能性があります。使用パターンとコストをモニタリングして、パフォーマンスと効率のバランスを取ります。

3. キャパシティの境界を設定します。
   - 最小 RCU/WCU: トラフィックが少ない期間に十分なキャパシティを維持します。
   - 最大 RCU/WCU: ピークトラフィックの需要に対応し、暴走スケーリングイベントから保護します。

DynamoDB Auto Scaling の設定と管理に関するガイダンスについては、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

> **Note**
>
> Auto Scaling は通常、トラフィックの変化に対応するために数分かかります。突然トラフィックが急増した場合、テーブルのバーストキャパシティは Auto Scaling が調整している間にすぐに保護されます。適切なヘッドルームを使用してターゲット使用率を設定し、スケーリングオペレーションの時間を確保し、予期しない需要に備えてバーストキャパシティを維持します。

テーブルまたはインデックスの読み取りまたは書き込み Auto Scaling 設定の最適化

Auto Scaling が有効になっていてもスロットリングがまだ発生する場合は、この手順を使用します。Auto Scaling は、テーブルとグローバルセカンダリインデックス (GSI) の両方で個別に調整でき、読み取りキャパシティユニットと書き込みキャパシティユニットを個別に制御できます。

- ターゲット使用率の調整: スロットリングが発生する前にスケーリングをトリガーするために、テーブルまたは GSI のターゲット使用率を下げることを検討します。これらの調整を行った後は、トラフィックを必ずモニタリングしてください。キャパシティの消費とコストへの影響の詳細については「the section called “テーブルまたは GSI の読み取りまたは書き込みキャパシティを調整するようにテーブル Auto Scaling を設定する”」を参照してください。
- キャパシティの境界を確認する: 最小キャパシティ設定と最大キャパシティ設定が実際のワークロードパターンと一致していることを確認します。

オンデマンドキャパシティモードに切り替える

キャパシティモードの切り替えに関する一般的な情報については「the section called “キャパシティモードの切り替え”」を参照してください。モードを切り替える際の特定の制約については、Service Quotas を参照してください。

GSI スループットキャパシティの増加

GSI で Auto Scaling が有効になっていない場合、またはすぐにキャパシティを増やす必要がある場合は、この手順を使用します。

1. DynamoDB コンソール、AWS CLI、または SDK を使用して GSI プロビジョンドキャパシティを更新します。
   - 読み取りキャパシティの場合: DynamoDB がリクエストを調整する前に GSI が 1 秒あたりに消費できる読み取りの最大数を指定する、特定の GSI の `ReadCapacityUnits` パラメータを増やします。GSI は結果整合性のある読み込みのみをサポートすることに注意してください。
   - 書き込みキャパシティの場合: DynamoDB がリクエストを調整する前に GSI が 1 秒あたりに消費できる書き込みの最大数を指定する、特定の GSI の `WriteCapacityUnits` パラメータを増やします。
2. GSI のプロビジョンドスループットキャパシティが、アカウントごとおよびテーブルごとのスループットクォータ内に留まっていることを確認します。

その他のリソース

- Auto Scaling とバーストキャパシティの活用から戦略的スロットル管理まで、さまざまな戦略を含む、DynamoDB プロビジョンドキャパシティテーブルのトラフィックのスパイクの処理の詳細については、「Handle traffic spikes with Amazon DynamoDB provisioned capacity」を参照してください。
- cron 式を使用してスケーリングポリシーをスケジュールする方法については、「Optimize costs by scheduling provisioned capacity for DynamoDB」を参照してください。
- プロビジョンドキャパシティモードでの DynamoDB テーブルのスループット使用率パターンのモニタリングと分析に関する実践的な情報については、「How to evaluate throughput utilization for Amazon DynamoDB tables in provisioned mode」を参照してください。

3- アカウント制限の超過

オンデマンドテーブルには管理するプロビジョンドキャパシティレベルはありませんが、DynamoDB はアカウントレベルのスループット制限を適用して、実行の暴走を防ぎ、すべてのお客様に公平なリソース使用量を確保します。これらのテーブルごとのアカウント制限は、アカウントとリージョンの組み合わせごとに設定された調整可能な保護手段として機能します。読み取りまたは書き込みの消費量がこれらの制限を超えると、DynamoDB は `AccountLimitExceeded` スロットリング例外でスロットリングの理由タイプを返します。テーブルごとのデフォルトのアカウント制限は、テーブルにカスタム最大スループット設定が設定されていない場合に自動的に適用されます。オプションで、コスト管理と予測可能性を高めるために最大スループット設定を構成することも、アプリケーション要件がデフォルトの制限を超えた場合は the section called “クォータ” コンソールからクォータの引き上げをリクエストすることもできます。

アカウント制限緩和策

このセクションでは、アカウント制限スロットリングシナリオの解決ガイダンスを提供します。このガイドを使用する前に、アプリケーションの例外処理から特定のスロットリングの理由を特定し、影響を受けるリソースの Amazon リソースネーム (ARN) を決定していることを確認します。スロットリングの理由の取得とスロットリングされたリソースの識別については、「the section called “診断フレームワーク”」を参照してください。

特定のスロットリングシナリオに進む前に、まずアクションが実際に必要かどうかを判断します。

- パフォーマンスへの影響を評価する: スロットリングが適用されているにもかかわらず、アプリケーションが依然としてパフォーマンス要件を満たしているかどうかを確認します。多くのアプリ

ケーションは、特に一括操作またはデータ移行中に、アカウントの制限またはその近くで正常に動作します。

- スロットリングパターンを確認する: スロットリングが断続的で、アプリケーションが再試行を効果的に処理する場合、ワークロードには現在の制限で十分である可能性があります。

一時的にアカウント制限に達した場合でもアプリケーションが正常に動作する場合は、即時の変更を実装するのではなく、単に状況をモニタリングすることを選択できます。

スロットリングが許容できないパフォーマンスの問題や信頼性の懸念を引き起こしていると判断した場合は、以下の特定のスロットリングの理由を選択して、推奨される緩和オプションを見つけます。

- the section called “TableReadAccountLimitExceeded”
- the section called “TableWriteAccountLimitExceeded”
- the section called “IndexReadAccountLimitExceeded”
- the section called “IndexWriteAccountLimitExceeded”

TableReadAccountLimitExceeded

この問題が発生した場合:

テーブルの読み取り消費量が、リージョンのテーブルごとのアカウントレベルの読み取りスループットクォータを超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

このスロットリングを解決するには、次の手順に従います。

- クォータの引き上げのリクエスト:

  テーブルごとの読み取りスループット制限 (クォータコード L-CF0CBE56) の引き上げをリクエストします。リクエストの送信方法の詳細については、「テーブルあたりのクォータの引き上げのリクエスト」を参照してください。

TableWriteAccountLimitExceeded

この問題が発生した場合:

テーブルの書き込み消費量が、リージョンのアカウントレベルのテーブルあたりの書き込みスループットクォータを超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

このスロットリングを解決するには、次の手順に従います。

- クォータの引き上げのリクエスト: テーブルあたりの書き込みスループット制限 (クォータコード L-AB614373) の引き上げをリクエストします。リクエストの送信方法の詳細については、「テーブルあたりのクォータの引き上げのリクエスト」を参照してください。

IndexReadAccountLimitExceeded

この問題が発生した場合:

グローバルセカンダリインデックス (GSI) に向けられた読み取りオペレーションは、現在の AWS リージョンでアカウントのテーブルごとの読み取りクォータで許可されているよりも多くのスループットを消費します。テーブルごとのアカウントレベルの読み取りスループットクォータは、テーブルとそのすべての GSI の組み合わせにまとめて適用されます。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

解決方法

アカウントのキャパシティ分布に基づいて適切な解決策を選択します。

- クォータの引き上げのリクエスト: テーブルあたりの読み取りスループット制限 (クォータコード L-CF0CBE56) の引き上げをリクエストします。リクエストの送信方法の詳細については、「テーブルあたりのクォータの引き上げのリクエスト」を参照してください。
- GSI の使用を最適化する: GSI 設計とクエリパターンを確認して、不要な読み込みキャパシティの消費量を減らします。

IndexWriteAccountLimitExceeded

この問題が発生した場合:

基本テーブルへの書き込みオペレーションにより、GSI に対応する更新を生成し、AWS リージョンのアカウントレベルのテーブルあたりの書き込みスループットクォータを合計で超過します。GSI に

よってインデックス付けされた属性を含む基本テーブル項目へのすべての書き込みは、その GSI への対応する書き込みオペレーションをトリガーします。これらの結合書き込みオペレーションは、テーブルあたりの書き込みスループットクォータにカウントされます。CloudWatch メトリクスを the section called “一般的な診断とモニタリング” でモニタリングして、これらのスロットリングイベントのパターンとタイミングを分析し、過剰な GSI 書き込みアクティビティの原因となっているオペレーションを特定できます。

解決方法

アカウントのキャパシティ分布に基づいて適切な解決策を選択します。

- クォータの引き上げのリクエスト: 基本テーブルオペレーションからの GSI 書き込みトラフィックの増加に対応するため、テーブルあたりの書き込みスループット制限 (クォータコード L-AB614373) の引き上げをリクエストします。テーブルごとの書き込みスループットクォータは、すべての GSI を含むテーブル全体に適用されます。リクエストの送信方法の詳細については、「テーブルあたりのクォータの引き上げのリクエスト」を参照してください。
- GSI 射影の最適化: GSI 射影と設計を確認して GSI への書き込み量を減らします。

一般的な診断とモニタリング

アカウント制限がスロットリングイベントを超えたトラブルシューティングを行う場合、いくつかの CloudWatch メトリクスは、テーブル単位またはアカウント全体の制限に達しているかどうかを特定し、キャパシティ分散パターンを理解するのに役立ちます。

重要な CloudWatch メトリクス

これらの主要なメトリクスをモニタリングして、アカウント制限スロットリングを診断します。

- アカウント制限スロットリングイベント: `ReadAccountLimitThrottleEvents` と `WriteAccountLimitThrottleEvents` は、アカウントレベルの制限によりリクエストがスロットリングされたタイミングを追跡します。`ReadThrottleEvents` および `WriteThrottleEvents` は、読み取りまたは書き込みリクエストがプロビジョニングされたキャパシティを超えたタイミングを追跡します。
- リソースごとのキャパシティ消費量: 各テーブルと GSI の `ConsumedReadCapacityUnits` と `ConsumedWriteCapacityUnits` は、個々のリソースの使用状況を表示します。
- アカウント全体の消費: CloudWatch メトリクス数式を使用して、すべてのテーブルと GSI の消費キャパシティを合計して、アカウントの合計使用量をモニタリングします。

解決手順

テーブルあたりのクォータの引き上げのリクエスト

アプリケーションが現在のテーブルごとのスループット制限を超えて動作する必要がある場合は、次の手順を使用してクォータ引き上げリクエストを送信する必要があります。AWS アカウント内の各 DynamoDB テーブル (および関連付けられたすべての GSI) は、特定のリージョン内のこれらのスループットクォータの対象となります。これらのクォータは、個々のテーブルとその GSI がまとめて消費できる読み取りまたは書き込みの最大キャパシティを表します。これらは、アカウント内のすべてのテーブルへの集約としてではなく、各テーブルに個別に適用されます。

オプションで、オンデマンドスループットの最大設定を構成することで、テーブルごとまたは GSI ごとにより低い制限を設定することもできます。

1. 引き上げる必要がある特定のクォータを特定します。
   - テーブルあたりの読み取りスループット制限 (クォータコード L-CF0CBE56): テーブルあたりのデフォルト 40,000 RCU
   - テーブルあたりの書き込みスループット制限 (クォータコード L-AB614373): テーブルあたりのデフォルト 40,000 WCU
2. 引き上げをリクエストする際は、AWS Service Quotas コンソールを使用します。
   - Service Quotas で DynamoDB サービスに移動する
   - クォータコードを使用して適切なクォータを検索する
   - 予測されるピーク使用量に基づいて引き上げをリクエストする
3. 以下を含め、引き上げの根拠を記載します。
   - 現在の使用パターンとピークトラフィック要件
   - キャパシティの増加に関するビジネス上の根拠
   - キャパシティの増加が必要な場合のタイムライン

> **Note**
>
> 通常、クォータの増加の処理には 24 ~ 48 時間かかります。それに応じてリクエストを計画し、承認を待っている間に一時的な緩和戦略を検討します。

GSI 射影と設計の最適化

グローバルセカンダリインデックス (GSI) の射影と設計を最適化して、キャパシティの消費量を減らし、パフォーマンスを向上させます。

選択的射影戦略

クエリがいくつかの属性にのみアクセスする必要がある場合、それらの属性のみを射影すると、基本テーブル項目が変更されたときに GSI に書き込まれるデータの量が減少します。射影タイプの詳細については、「グローバルセカンダリインデックスの射影」を参照してください。

1. クエリパターンを分析する: アプリケーションのクエリパターンを確認して、GSI を介して実際にアクセスされる属性を特定します。
2. 選択的射影を使用する: 書き込み量を減らすためにクエリで実際に必要な属性のみを射影します。
3. KEYS_ONLY の検討: クエリでキー属性のみが必要な場合は、KEYS_ONLY 射影を使用して書き込み量を最小限に抑えます。
4. 読み取りと書き込みのトレードオフのバランスを取る: 属性の射影数を減らすと、書き込みキャパシティの消費量は減りますが、基本テーブルの読み取り回数が増える場合があります。

スパース GSI 実装

スパース GSI には、基本テーブルのすべての項目ではなく、インデックス付き属性を持つ項目のみが含まれます。これにより、データの特定のサブセットを頻繁にクエリする場合に、パーティション密度が低下し、パフォーマンスが向上します。

1. 特定の属性値を持つ項目のみを含む GSI を設計します。
2. インデックスを作成する必要がある項目にのみ GSI パーティションキー属性を設定して、条件付きインデックスを作成します。
3. スパース GSI の複合キー (status#timestamp など) を使用して、インデックス化された項目のサブセット内にトラフィックをさらに分散します。

これらの戦略の実装の詳細については、「DynamoDB でセカンダリインデックスを使用するためのベストプラクティス」を参照してください。

4- オンデマンドの最大スループットの超過

オンデマンドテーブルまたは GSI を設定する場合、オプションでテーブルまたはインデックスレベルで最大スループット制限 (`MaxReadRequestUnits` および `MaxWriteRequestUnits`) を設定して、コストの跳ね上がりを防止したり、ダウンストリームシステムが過負荷にならないようにしたりできます。最大スループットの詳細については、「the section called “オンデマンドテーブルの DynamoDB 最大スループット”」を参照してください。

読み取りまたは書き込みの消費量がこれらの自己適用制限を超えると、制限を超える追加のリクエストはクイックスロットリングレスポンスを受け取ります。DynamoDB は、どのリソースがスループット境界に達したかを示す `MaxOnDemandThroughputExceeded` スロットリングの理由タイプの例外を返します。

オンデマンドの最大スループットがスロットリングを超過

このセクションでは、オンデマンドの最大スループット超過スロットリングシナリオの解決ガイダンスを提供します。このガイドを使用する前に、アプリケーションの例外処理から特定のスロットリングの理由を特定し、影響を受けるリソースの Amazon リソースネーム (ARN) を決定していることを確認します。スロットリングの理由の取得とスロットリングされたリソースの識別については、「the section called “診断フレームワーク”」を参照してください。

特定のスロットリングシナリオに進む前に、まずアクションが実際に必要かどうかを検討します。

- 最大スループット設定を評価する: これらの制限は、コストを制御したり、ダウンストリームシステムを保護するように意図的に設定されました。`MaxOnDemandThroughputExceeded` スロットリングイベントを受信している場合、制限は設計どおりに機能しています。これらの制限の引き上げが、元のコストコントロールまたはシステム保護の目標と一致するかどうかを検討します。
- アプリケーションの影響を評価する: スロットリングがアプリケーションまたはユーザーに実際に問題を引き起こしているかどうかを確認します。アプリケーションが再試行を効果的に処理し、時折スロットリングしてもパフォーマンス要件を満たしている場合は、現在の制限を維持することが適切な選択である可能性があります。
- トラフィックパターンを確認する: スロットリングが予想されるトラフィックパターンを表すか、異常なスパイクを表すかを分析します。一貫して制限を超える予測可能で定期的なトラフィックパターンについては、最大スループット設定の調整が必要になる場合があります。一時的なスパイクの場合、制限を引き上げるよりも、より良いリクエスト分散戦略を実装する方が適切な場合があります。

考慮後、最大スループット設定を調整する必要があると判断した場合は、ターゲットを絞った修復オプションについて、以下の特定のスロットリングシナリオを参照してください。

- the section called “TableReadMaxOnDemandThroughputExceeded”
- the section called “TableWriteMaxOnDemandThroughputExceeded”
- the section called “IndexReadMaxOnDemandThroughputExceeded”
- the section called “IndexWriteMaxOnDemandThroughputExceeded”

TableReadMaxOnDemandThroughputExceeded

この問題が発生した場合

オンデマンドテーブルが、設定された最大読み取りスループットキャパシティを超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- 最大スループット制限を引き上げる: DynamoDB コンソール、AWS CLI、または DynamoDB `UpdateTable` API を使用して、影響を受けるテーブルの `MaxReadRequestUnits` の値を増やし、モニタリングして調整します。これにより、スロットリングが発生する前にテーブルでより高い読み取りスループットを処理できます。
- 最大制限を削除する: `MaxReadRequestUnits` を `-1` に設定すると上限が削除され、アカウントレベルのスループットクォータまで需要に応じてスケーリングできます。これにより、カスタム制限は削除されますが、AWS のアカウントレベルの保護は維持されます。ただし、テーブルがアカウントレベルのクォータに達する前に大幅に多くのキャパシティを消費するようになったため、この制限を削除した後は支出を注意深くモニタリングすることが重要です。

TableWriteMaxOnDemandThroughputExceeded

この問題が発生した場合

オンデマンドテーブルが、設定された最大書き込みスループットキャパシティを超えています。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- 最大スループット制限を引き上げる: DynamoDB コンソール、AWS CLI、または DynamoDB `UpdateTable` API を使用して、影響を受けるテーブルの `MaxWriteRequestUnits` の値を増やし、モニタリングして調整します。
- 最大制限を削除する: `MaxWriteRequestUnits` を `-1` に設定すると上限が削除され、アカウントレベルのスループットクォータまで需要に応じてスケーリングできます。これにより、カスタム制限は削除されますが、AWS のアカウントレベルの保護は維持されます。ただし、テーブルがアカウントレベルのクォータに達する前に大幅に多くのキャパシティを消費するようになったため、この制限を削除した後は支出を注意深くモニタリングすることが重要です。

IndexReadMaxOnDemandThroughputExceeded

この問題が発生した場合

オンデマンドモードでの GSI への読み取りリクエストが、GSI で設定された最大読み取りスループットキャパシティを超えました。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- GSI 最大スループット制限を引き上げる: DynamoDB コンソール、AWS CLI、または DynamoDB `UpdateTable` API を使用して、影響を受ける GSI の `MaxReadRequestUnits` の値を増やし、モニタリングして調整します。
- GSI の最大制限を削除する: GSI の `MaxReadRequestUnits` を `-1` に設定すると上限が削除され、アカウントレベルのスループットクォータまで需要に応じてスケーリングできます。これにより、カスタム制限は削除されますが、AWS のアカウントレベルの保護は維持されます。ただし、この制限を削除した後は、支出を注意深くモニタリングすることが重要です。

IndexWriteMaxOnDemandThroughputExceeded

この問題が発生した場合

基本テーブルの項目を更新すると、GSI が設定した最大書き込みスループットキャパシティを超えるオンデマンドモードで GSI に書き込みが行われ、バックプレッシャースロットリングが発生しま

す。the section called “一般的な診断とモニタリング” で CloudWatch メトリクスをモニタリングして、スロットリングイベントを分析できます。

修復オプション

スロットリングイベントに対処するには、以下の手順を検討します。

- GSI 最大スループット制限を引き上げる: DynamoDB コンソール、AWS CLI、または DynamoDB UpdateTable API を使用して、影響を受ける GSI の MaxWriteRequestUnits の値を増やし、モニタリングして調整します。
- GSI の最大制限を削除する: GSI の MaxWriteRequestUnits を -1 に設定すると上限が削除され、アカウントレベルのスループットクォータまで需要に応じてスケーリングできます。これにより、カスタム制限は削除されますが、AWS のアカウントレベルの保護は維持されます。ただし、この制限を削除した後は、支出を注意深くモニタリングすることが重要です。

一般的な診断とモニタリング

オンデマンドの最大スループットがスロットリングイベントを超えた場合、いくつかの CloudWatch メトリクスが根本原因とスケーリングパターンの特定に役立ちます。

重要な CloudWatch メトリクス

これらの主要なメトリクスをモニタリングして、オンデマンドの最大スループットがスロットリングを超えたことを診断します。

- 最大スループットスロットリングイベント: ReadMaxOnDemandThroughputThrottleEvents と WriteMaxOnDemandThroughputThrottleEvents は、最大制限を超えたためにリクエストがスロットリングされたタイミングを追跡します。ReadThrottleEvents と WriteThrottleEvents は、読み取りまたは書き込みリクエストがプロビジョニングされたキャパシティを超えたタイミングを追跡します。
- テーブルまたはグローバルセカンダリインデックスに設定された現在の最大スループット: OnDemandMaxReadRequestUnits および OnDemandMaxWriteRequestUnits には、現在の最大キャパシティ制限が表示されます。
- 実際のキャパシティ消費量: ConsumedReadCapacityUnits および ConsumedWriteCapacityUnits は実際の使用パターンを表示します。

分析アプローチ

以下の手順に従って、オンデマンドの最大スループットが診断を超過したことを確認します。

1. 消費されたキャパシティと最大キャパシティ制限を比較する - 消費が一貫して最大制限に近づいているか、または超えているかを確認します。
2. スロットリングイベントの頻度とタイミングを確認して、パターンを特定します。スロットリングイベントと一致する消費キャパシティの突然の増加を探します。
3. CloudWatch Contributor Insights を使用して、キャパシティを最も多く消費する項目またはパーティションキーを特定します。

DynamoDB のグローバルセカンダリインデックス (GSI) の書き込みスロットリングとバックプレッシャーについて

GSI バックプレッシャースロットリングは、書き込みオペレーションとスロットリングの間に間接的な関係が生じるため、DynamoDB の最も複雑なスロットリングシナリオの 1 つです。アプリケーションは基本テーブルに書き込みますが、1 つ以上のインデックスでキャパシティの制約によりスロットリングが発生します。

GSI バックプレッシャースロットリングについて

DynamoDB テーブルに書き込むと、そのテーブルのすべてのグローバルセカンダリインデックス (GSI) は、結果整合性モデルを使用して、非同期で更新されます。GSI にこれらの更新を処理するのに十分なキャパシティがない場合、DynamoDB はデータ整合性を維持するために基本テーブルへの書き込みを調整します。これは GSI バックプレッシャーと呼ばれます。GSI の仕組みの詳細については、「DynamoDB のグローバルセカンダリインデックス」を参照してください。

アクセスされるリソースがスロットリングの原因となるリソースでもあるダイレクトテーブルスロットリングとは異なり、GSI バックプレッシャーは基本テーブルとそのインデックス間に依存関係を作成します。基本テーブルに十分なキャパシティがある場合でも、関連付けられた GSI が更新ボリュームを処理できない場合、書き込みはスロットリングされます。この関係は、パーティションレベルの制約が基本テーブルと各 GSI に個別に適用されるため、理解することが特に重要です。それぞれに独自のパーティション構造と対応するスループット制限があります。

GSI パーティショニングは GSI のパーティションキーに基づいています。これは、基本テーブルのパーティションキーとは異なることがよくあります。基本テーブルへのアクセスがパーティション全体に完全に分散されていても、GSI 更新は特定のパーティションに集中し、GSI にホットスポットを作成する可能性があります。テーブルと GSI のパーティションキー設計に関する一般的なベストプラクティスについては、「DynamoDB パーティションキーの設計」を参照してください。

例えば、基本テーブルがパーティションキーとして customerId (均等に分散) を使用し、GSI がパーティションキーとして status (「アクティブ」、「保留中」、「クローズ」などの制限された値) を使用する場合、一般的なステータス値を持つ項目を更新すると、基本テーブルアクセスのバランスが取れていても GSI ホットパーティションを作成できます。これにより、特に困難なシナリオが発生します。基本テーブルと GSI の両方に十分な全体的なキャパシティがあり、基本テーブルのアクセスパターンが適切に分散されているように見える場合でも、GSI ホットパーティションが原因でアプリケーションのスロットリングが発生する可能性があります。

スロットリング例外は GSI を指していますが (ResourceArn 経由)、スロットリングされる実際のオペレーションは基本テーブルへの書き込みです。アプリケーションが基本テーブルに書き込んでいるが、GSI に関する例外を受信しているため、これは混乱を招く可能性があります。

GSI スロットリングのタイプ

GSI バックプレッシャースロットリングは、特定のキャパシティ制約に応じて異なる例外タイプを通じて現れます。

- GSI プロビジョンドキャパシティの超過: GSI に基本テーブルオペレーションからの更新を処理するのに十分な書き込みキャパシティユニットがない場合に発生します。これにより、理由が IndexWriteProvisionedThroughputExceeded の ProvisionedThroughputExceededException が生成され、ResourceArn はキャパシティ制約が発生している特定の GSI を直接指します。
- GSI オンデマンド最大スループット超過: GSI 書き込みオペレーションがオンデマンドテーブルで設定された最大制限を超えた場合に発生します。これにより、理由が IndexWriteMaxOnDemandThroughputExceeded の ThrottlingException が生成され、スループット制限が設定された特定の GSI が識別されます。
- GSI パーティションの制限を超過: 全体的な GSI キャパシティが十分であると思われる場合でも、個々の GSI パーティションがスループット制限 (ホットパーティション) を超えたときに発生します。これにより、IndexWriteKeyRangeThroughputExceeded を理由とする ThrottlingException が生成され、ResourceArn で識別された特定の GSI でホットパーティションの問題が示されます。GSI パーティションディストリビューションは基本テーブルのパーティションディストリビューションと大きく異なり、基本テーブルアクセスが均等に分散されている場合でも GSI にホットスポットが発生する可能性があるため、これは特に重要です。
- GSI アカウント制限の超過: 特定の GSI への書き込みオペレーションが、アカウントレベルで設定されたテーブルごと (またはそのテーブル内の個々の GSI) のリージョンスループット境界を超えるとトリガーされます。DynamoDB は、アカウント制限を超えて使用量をプッシュした GSI を指す IndexWriteAccountLimitExceeded を理由とする ThrottlingException を返します。

このスロットリングは、制限を超える GSI ごとに個別に発生します。テーブルごと、アカウントごと、リージョンごとのサービスクォータの詳細については、「the section called “クォータ”」を参照してください。

CloudWatch スロットリングメトリクス

このページでは、DynamoDB テーブルとインデックスのスロットリングの問題の特定、診断、解決に役立つように特別に設計された CloudWatch メトリクスに関する包括的なガイドを提供します。

一般的なスロットリングメトリクス

- `ThrottledRequests`
  - リクエスト内の個々のイベントがスロットリングされた数に関係なく、リクエスト内のイベントがスロットリングされたときに 1 つ増加します。例えば、グローバルセカンダリインデックス (GSI) を持つテーブル内の項目を更新する場合、基本テーブルへの書き込みオペレーションと各インデックスへの書き込みオペレーションという複数のイベントが発生します。これらの個々のイベントのいずれかがスロットリングされた場合、`ThrottledRequests` メトリクスは 1 回だけ増分されます。

    スロットリングの実際の範囲が隠される可能性があるため、DynamoDB のパフォーマンスをモニタリングおよびトラブルシューティングするときには、この動作を理解することが重要です。より包括的なインサイトを得るには、`ThrottledRequests` メトリクスを `ReadThrottleEvents`、`WriteThrottleEvents` などの特定のイベントレベルのメトリクス、および `ReadKeyRangeThroughputThrottleEvents` などのターゲットメトリクスと比較します。これらの原因固有のメトリクスの完全なリストは、このページで入手できます。各メトリクスは、スロットリング例外内でキャプチャされる特定のスロットリングの理由に対応します。スロットリングイベント中にこれらの理由を取得して解釈するガイダンスについては、スロットリングの問題の根本原因を特定して解決する手順を提供する「the section called “スロットリングの診断”」セクションを参照してください。
- `ReadThrottleEvents`
  - 1 つのテーブルまたは GSI のプロビジョニングされた RCU を超えるリクエストを監視します。
- `WriteThrottleEvents`
  - 1 つのテーブルまたは GSI のプロビジョニングされた WCU を超えるリクエストを監視します。

原因別の詳細なスロットリングメトリクス

オンデマンドスループットスロットリング

- ReadMaxOnDemandThroughputThrottleEvents
  - オンデマンドの最大スループットによってスロットリングされた読み取りリクエストの数。
- WriteMaxOnDemandThroughputThrottleEvents
  - オンデマンドの最大スループットによってスロットリングされた書き込みリクエストの数。

アカウントレベルのスロットリング

- ReadAccountLimitThrottleEvents
  - アカウント制限によりスロットリングされた読み取りリクエストの数。
- WriteAccountLimitThrottleEvents
  - アカウント制限によりスロットリングされた書き込みリクエストの数。

パーティションレベルのスロットリング

- ReadKeyRangeThroughputThrottleEvents
  - パーティションの制限によりスロットリングされた読み取りリクエストの数。
- WriteKeyRangeThroughputThrottleEvents
  - パーティションの制限によりスロットリングされた書き込みリクエストの数。

キャパシティ分析メトリクス

- OnlineIndexConsumedWriteCapacity
  - 既存のテーブルに新しい GSI を追加すると、DynamoDB は基本テーブルから新しいインデックスにデータをコピーするバックフィルオペレーションを実行します。このプロセスでは、書き込みキャパシティーユニットが消費されます。OnlineIndexConsumedWriteCapacity メトリクスは、この特定の消費量を追跡します。

    この消費は、ConsumedWriteCapacityUnits によって追跡される通常の書き込みオペレーションとは別に追加されます。GSI の通常の ConsumedWriteCapacityUnits メトリクスには、インデックスの作成中に消費される書き込みスループットは含まれません。
- ProvisionedReadCapacityUnits および ProvisionedWriteCapacityUnits
  - テーブルまたは指定したグローバルセカンダリインデックスについて、指定した期間に消費された書き込み容量ユニットのプロビジョンド読み取りまたは書き込み容量ユニットの数。

    - デフォルトでは、TableName ディメンションはテーブルのみの ProvisionedReadCapacityUnits を返します。グローバルセカンダリインデックスの読み込みまたは書き込みのプロビジョンドキャパシティユニットの数を確認するには、TableName と GlobalSecondaryIndexName の両方を指定する必要があります。
- ConsumedReadCapacityUnits および ConsumedWriteCapacityUnits
    - 指定された期間に消費された読み取りまたは書き込みキャパシティユニットの数を表示します。ConsumedWriteCapacityUnits には、最初のインデックス作成プロセス中に消費された書き込みキャパシティは含まれません。

DynamoDB CloudWatch のメトリクスの詳細については、「DynamoDB のメトリクスとディメンション」を参照してください。

DynamoDB 付録

トピック

- DynamoDB で SSL/TLS 接続の確立に関する問題をトラブルシューティングする
- DynamoDB で使用するテーブルとデータの例
- DynamoDB でサンプルテーブルを作成してデータをアップロードする
- AWS SDK for Python (Boto) を使用した DynamoDB のサンプルアプリケーション: Tic-tac-toe
- DynamoDB の予約語
- AWS SDK for Java 1.x の例
- AWS SDK for Go 1.x の例
- AWS SDK for Node.js 2.x の例

DynamoDB で SSL/TLS 接続の確立に関する問題をトラブルシューティングする

Amazon DynamoDB は、サードパーティの認証機関ではなく Amazon Trust Services (ATS) 認証機関によって署名された安全な証明書へのエンドポイントの移動を行っています。2017 年 12 月、Amazon Trust Services が発行する安全な証明書を使用して EU-WEST-3 (パリ) リージョンを立ち上げました。2017 年 12 月以降に提供が開始されたすべての新しいリージョンには、Amazon Trust Services 発行の証明書を持つエンドポイントがあります。このガイドでは、SSL/TLS 接続の問題を検証し、トラブルシューティングする方法を示します。

アプリケーションまたはサービスをテストする

ほとんどの AWS SDK とコマンドラインインターフェイス (CLI) は、Amazon Trust Services 証明書機関をサポートしています。2013 年 10 月 29 日以前にリリースされた AWS SDK for Python (Boto) または CLI のバージョンを使用している場合は、アップグレードする必要があります。.NET、Java、PHP、Go、JavaScript、C ++、SDK、CLI には証明書がバンドルされていません。証明書は、基盤となるオペレーティングシステムから取得されます。2015 年 6 月 10 日以降、Ruby SDK には、少なくとも 1 つの必要な CA が含まれています。この日付以前の Ruby V2 SDK には証明書はバンドルされていませんでした。サポートされていない、カスタム、または変更バージョンの AWS SDK を使用している場合、またはカスタム信頼ストアを使用している場合は、Amazon Trust Services 証明書機関のサポートがない可能性があります。

DynamoDB エンドポイントへのアクセスを検証するには、EU-WEST-3 リージョンの DynamoDB API または DynamoDB Streams API するアクセスするテストを開発し、TLS ハンドシェイクが成功することを検証する必要があります。このテストでアクセスする必要がある特定のエンドポイントは次のとおりです。

- DynamoDB: https://dynamodb.eu-west-3.amazonaws.com
- DynamoDB Streams: https://streams.dynamodb.eu-west-3.amazonaws.com

アプリケーションが Amazon Trust Services 認証機関をサポートしていない場合は、次のいずれかのエラーが表示されます。

- SSL/TLS ネゴシエーションエラー
- ソフトウェアが SSL/TLS ネゴシエーションの障害を示すエラーを受信するまでに長い遅延が発生します。遅延時間は、クライアントの再試行戦略とタイムアウト設定に応じて異なります。

クライアントブラウザのテスト

ブラウザが Amazon DynamoDB に接続できることを確認するには、https://dynamodb.eu-west-3.amazonaws.com を開きます。テストが成功すると、次のようなメッセージが表示されます。

```
healthy: dynamodb.eu-west-3.amazonaws.com
```

テストが失敗した場合は、https://untrusted-root.badssl.com/ に見られるようなエラーが表示されます。

ソフトウェアアプリケーションクライアントの更新中

DynamoDB または DynamoDB Streams API エンドポイントに (ブラウザまたはプログラムで) アクセスするアプリケーションが以下のいずれかの CA をサポートしていない場合、クライアントマシン上の信頼できる CA リストを更新する必要があります。

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

クライアントが上記 3 つの CA のいずれかを既に信頼している場合、これらは DynamoDB で使用される証明書を信頼するため、アクションは必要ありません。ただし、クライアントが上記の CA を

信頼していない場合、DynamoDB API または DynamoDB Streams API への HTTPS 接続が失敗します。詳細については、https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/ のブログ記事を参照してください。

クライアントブラウザを更新する

ブラウザの証明書バンドルは、ブラウザを更新するだけで更新できます。最も一般的なブラウザの手順は、それぞれのブラウザのウェブサイトに記載されています。

- Chrome: https://support.google.com/chrome/answer/95414?hl=en
- Firefox: https://support.mozilla.org/en-US/kb/update-firefox-latest-version
- Safari: https://support.apple.com/en-us/HT204416
- Internet Explorer: https://support.microsoft.com/en-us/help/17295/windows-internet-explorer-which-version#ie=other

手動で証明書バンドルを更新する

DynamoDB API または DynamoDB Streams API にアクセスできない場合は、証明書バンドルを更新する必要があります。そのためには、必要な CA の少なくとも 1 つをインポートする必要があります。これらは、https://www.amazontrust.com/repository/ から入手できます。

以下のオペレーティングシステムとプログラミング言語は、Amazon Trust Services 証明書をサポートしています。

- 2005 年 1 月以降の更新プログラムがインストールされている Microsoft Windows のバージョン、Windows Vista、Windows 7、Windows Server 2008、およびそれ以降のバージョン。
- MacOS X 10.4 Release 5 用 の MacOS X 10.4 with Java、MacOS X 10.5 およびそれ以降のバージョン。
- Red Hat Enterprise Linux 5 (2007 年 3 月)、Linux 6、Linux 7、CentOS 5、CentOS 6、CentOS 7
- Ubuntu 8.10
- Debian 5.0
- Amazon Linux (すべてのバージョン)
- Java 1.4.2_12、Java 5 update 2、およびそれ以降のバージョン (Java 6、Java 7、Java 8)

それでも接続できない場合は、該当するソフトウェアのドキュメントを参照するか、OS ベンダーまたは AWS サポート (https://aws.amazon.com/support) にお問い合わせください。

DynamoDB で使用するテーブルとデータの例

Amazon DynamoDB デベロッパーガイドでは、サンプルテーブルを使用して DynamoDB のさまざまな機能について説明しています。

テーブル名	プライマリキー
ProductCatalog	シンプルなプライマリキー: • `Id` (数値)
Forum	シンプルなプライマリキー: • `Name` (文字列)
Thread	複合プライマリキー: • `ForumName` (文字列) • `Subject` (文字列)
Reply	複合プライマリキー: • `Id` (文字列) • `ReplyDateTime` (文字列)

Reply テーブルには、PostedBy-Message-Index というグローバルセカンダリインデックスがあります。このインデックスは、Reply テーブルの 2 つの非キー属性でのクエリを容易にします。

インデックス名	プライマリキー
PostedBy-Message-Index	複合プライマリキー: • `PostedBy` (文字列) • `Message` (文字列)

これらのテーブルの詳細については、「ステップ 1: DynamoDB にテーブルを作成する」および「ステップ 2: DynamoDB テーブルにデータを書き込む」を参照してください。

サンプルデータファイル

トピック

- ProductCatalog サンプルデータ
- Forum サンプルデータ
- Thread サンプルデータ
- Reply サンプルデータ

このセクションでは、ProductCatalog、Forum、Thread、Reply テーブルのロードに使用されるサンプルデータファイルを示します。

各データファイルには複数の `PutRequest` 要素が含まれます。各要素には 1 つの項目が含まれます。これらの `PutRequest` 要素は、AWS Command Line Interface (AWS CLI) と共に `BatchWriteItem` オペレーションへの入力として使用されます。

ProductCatalog サンプルデータ

```
{
    "ProductCatalog": [
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "101"
                    },
                    "Title": {
                        "S": "Book 101 Title"
                    },
                    "ISBN": {
                        "S": "111-1111111111"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            }
                        ]
                    },
                    "Price": {
                        "N": "2"
```

```
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 0.5"
                    },
                    "PageCount": {
                        "N": "500"
                    },
                    "InPublication": {
                        "BOOL": true
                    },
                    "ProductCategory": {
                        "S": "Book"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "102"
                    },
                    "Title": {
                        "S": "Book 102 Title"
                    },
                    "ISBN": {
                        "S": "222-2222222222"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            },
                            {
                                "S": "Author2"
                            }
                        ]
                    },
                    "Price": {
                        "N": "20"
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 0.8"
                    },
```

```
                    "PageCount": {
                        "N": "600"
                    },
                    "InPublication": {
                        "BOOL": true
                    },
                    "ProductCategory": {
                        "S": "Book"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "103"
                    },
                    "Title": {
                        "S": "Book 103 Title"
                    },
                    "ISBN": {
                        "S": "333-3333333333"
                    },
                    "Authors": {
                        "L": [
                            {
                                "S": "Author1"
                            },
                            {
                                "S": "Author2"
                            }
                        ]
                    },
                    "Price": {
                        "N": "2000"
                    },
                    "Dimensions": {
                        "S": "8.5 x 11.0 x 1.5"
                    },
                    "PageCount": {
                        "N": "600"
                    },
                    "InPublication": {
```

```
                    "BOOL": false
                },
                "ProductCategory": {
                    "S": "Book"
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "201"
                    },
                    "Title": {
                        "S": "18-Bike-201"
                    },
                    "Description": {
                        "S": "201 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Mountain A"
                    },
                    "Price": {
                        "N": "100"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
```

```
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "202"
                    },
                    "Title": {
                        "S": "21-Bike-202"
                    },
                    "Description": {
                        "S": "202 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Brand-Company A"
                    },
                    "Price": {
                        "N": "200"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Green"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "203"
                    },
```

```
                    "Title": {
                        "S": "19-Bike-203"
                    },
                    "Description": {
                        "S": "203 Description"
                    },
                    "BicycleType": {
                        "S": "Road"
                    },
                    "Brand": {
                        "S": "Brand-Company B"
                    },
                    "Price": {
                        "N": "300"
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Green"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "N": "204"
                    },
                    "Title": {
                        "S": "18-Bike-204"
                    },
                    "Description": {
```

```
                    "S": "204 Description"
                },
                "BicycleType": {
                    "S": "Mountain"
                },
                "Brand": {
                    "S": "Brand-Company B"
                },
                "Price": {
                    "N": "400"
                },
                "Color": {
                    "L": [
                        {
                            "S": "Red"
                        }
                    ]
                },
                "ProductCategory": {
                    "S": "Bicycle"
                }
            }
        }
    },
    {
        "PutRequest": {
            "Item": {
                "Id": {
                    "N": "205"
                },
                "Title": {
                    "S": "18-Bike-204"
                },
                "Description": {
                    "S": "205 Description"
                },
                "BicycleType": {
                    "S": "Hybrid"
                },
                "Brand": {
                    "S": "Brand-Company C"
                },
                "Price": {
                    "N": "500"
```

```
                    },
                    "Color": {
                        "L": [
                            {
                                "S": "Red"
                            },
                            {
                                "S": "Black"
                            }
                        ]
                    },
                    "ProductCategory": {
                        "S": "Bicycle"
                    }
                }
            }
        }
    ]
}
```

Forum サンプルデータ

```
{
    "Forum": [
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S":"Amazon DynamoDB"},
                    "Category": {"S":"Amazon Web Services"},
                    "Threads": {"N":"2"},
                    "Messages": {"N":"4"},
                    "Views": {"N":"1000"}
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Name": {"S":"Amazon S3"},
                    "Category": {"S":"Amazon Web Services"}
                }
            }
        }
```

```
    ]
}
```

Thread サンプルデータ

```
{
    "Thread": [
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 1"
                    },
                    "Message": {
                        "S": "DynamoDB thread 1 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-22T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "0"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "index"
                            },
                            {
                                "S": "primarykey"
                            },
                            {
```

```
                                "S": "table"
                            }
                        ]
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon DynamoDB"
                    },
                    "Subject": {
                        "S": "DynamoDB Thread 2"
                    },
                    "Message": {
                        "S": "DynamoDB thread 2 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-15T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "3"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "items"
                            },
                            {
                                "S": "attributes"
                            },
                            {
                                "S": "throughput"
```

```
                        }
                    ]
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "ForumName": {
                        "S": "Amazon S3"
                    },
                    "Subject": {
                        "S": "S3 Thread 1"
                    },
                    "Message": {
                        "S": "S3 thread 1 message"
                    },
                    "LastPostedBy": {
                        "S": "User A"
                    },
                    "LastPostedDateTime": {
                        "S": "2015-09-29T19:58:22.514Z"
                    },
                    "Views": {
                        "N": "0"
                    },
                    "Replies": {
                        "N": "0"
                    },
                    "Answered": {
                        "N": "0"
                    },
                    "Tags": {
                        "L": [
                            {
                                "S": "largeobjects"
                            },
                            {
                                "S": "multipart upload"
                            }
                        ]
                    }
                }
```

```
            }
        }
    ]
}
```

Reply サンプルデータ

```
{
    "Reply": [
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-15T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 1 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-22T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 2 text"
                    },
                    "PostedBy": {
                        "S": "User B"
                    }
                }
```

```
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 2"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-29T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 2 Reply 1 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 2"
                    },
                    "ReplyDateTime": {
                        "S": "2015-10-05T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 2 Reply 2 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        }
    ]
}
```

DynamoDB でサンプルテーブルを作成してデータをアップロードする

この付録では、テーブルの作成とプログラムによるデータの追加の両方を行うコードについて説明します。

トピック

- AWS SDK for Java を使用してサンプルテーブルを作成してデータをアップロードする
- AWS SDK for .NET を使用してサンプルテーブルを作成してデータをアップロードする

AWS SDK for Java を使用してサンプルテーブルを作成してデータをアップロードする

次の Java コード例は、テーブルを作成し、テーブルにデータをアップロードします。Eclipse を使用してこのコードを実行するための詳しい手順については、「Java コードの例」を参照してください。

```
package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.HashSet;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
```

```
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class CreateTablesLoadData {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    static String productCatalogTableName = "ProductCatalog";
    static String forumTableName = "Forum";
    static String threadTableName = "Thread";
    static String replyTableName = "Reply";

    public static void main(String[] args) throws Exception {

        try {

            deleteTable(productCatalogTableName);
            deleteTable(forumTableName);
            deleteTable(threadTableName);
            deleteTable(replyTableName);

            // Parameter1: table name
            // Parameter2: reads per second
            // Parameter3: writes per second
            // Parameter4/5: partition key and data type
            // Parameter6/7: sort key and data type (if applicable)

            createTable(productCatalogTableName, 10L, 5L, "Id", "N");
            createTable(forumTableName, 10L, 5L, "Name", "S");
            createTable(threadTableName, 10L, 5L, "ForumName", "S", "Subject", "S");
            createTable(replyTableName, 10L, 5L, "Id", "S", "ReplyDateTime", "S");

            loadSampleProducts(productCatalogTableName);
            loadSampleForums(forumTableName);
            loadSampleThreads(threadTableName);
            loadSampleReplies(replyTableName);

        } catch (Exception e) {
            System.err.println("Program failed:");
            System.err.println(e.getMessage());
        }
```

```
        System.out.println("Success.");
    }

    private static void deleteTable(String tableName) {
        Table table = dynamoDB.getTable(tableName);
        try {
            System.out.println("Issuing DeleteTable request for " + tableName);
            table.delete();
            System.out.println("Waiting for " + tableName + " to be deleted...this may
take a while...");
            table.waitForDelete();

        } catch (Exception e) {
            System.err.println("DeleteTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
            String partitionKeyName, String partitionKeyType) {

        createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
    }

    private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
            String partitionKeyName, String partitionKeyType, String sortKeyName,
String sortKeyType) {

        try {

            ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
            keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); //
Partition

                            // key

            ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
            attributeDefinitions
                    .add(new AttributeDefinition().withAttributeName(partitionKeyName)
```

```
                            .withAttributeType(partitionKeyType));

            if (sortKeyName != null) {
                keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

                              // key
                attributeDefinitions
                        .add(new
AttributeDefinition().withAttributeName(sortKeyName).withAttributeType(sortKeyType));
            }

            CreateTableRequest request = new
CreateTableRequest().withTableName(tableName).withKeySchema(keySchema)
                    .withProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits(readCapacityUnits)
                            .withWriteCapacityUnits(writeCapacityUnits));

            // If this is the Reply table, define a local secondary index
            if (replyTableName.equals(tableName)) {

                attributeDefinitions
                        .add(new
AttributeDefinition().withAttributeName("PostedBy").withAttributeType("S"));

                ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
ArrayList<LocalSecondaryIndex>();
                localSecondaryIndexes.add(new
LocalSecondaryIndex().withIndexName("PostedBy-Index")
                        .withKeySchema(
                                new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH), //
Partition

                               // key
                                new
KeySchemaElement().withAttributeName("PostedBy").withKeyType(KeyType.RANGE)) // Sort

                              // key
                        .withProjection(new
Projection().withProjectionType(ProjectionType.KEYS_ONLY)));

                request.setLocalSecondaryIndexes(localSecondaryIndexes);
            }
```

```
            request.setAttributeDefinitions(attributeDefinitions);

            System.out.println("Issuing CreateTable request for " + tableName);
            Table table = dynamoDB.createTable(request);
            System.out.println("Waiting for " + tableName + " to be created...this may
take a while...");
            table.waitForActive();

        } catch (Exception e) {
            System.err.println("CreateTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void loadSampleProducts(String tableName) {

        Table table = dynamoDB.getTable(tableName);

        try {

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("Id", 101).withString("Title", "Book
101 Title")
                    .withString("ISBN", "111-1111111111")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1"))).withNumber("Price", 2)
                    .withString("Dimensions", "8.5 x 11.0 x
0.5").withNumber("PageCount", 500)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 102).withString("Title", "Book 102
Title")
                    .withString("ISBN", "222-2222222222")
                    .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1", "Author2")))
                    .withNumber("Price", 20).withString("Dimensions", "8.5 x 11.0 x
0.8").withNumber("PageCount", 600)
                    .withBoolean("InPublication", true).withString("ProductCategory",
"Book");
            table.putItem(item);
```

```
            item = new Item().withPrimaryKey("Id", 103).withString("Title", "Book 103
 Title")
                    .withString("ISBN", "333-3333333333")
                    .withStringSet("Authors", new
 HashSet<String>(Arrays.asList("Author1", "Author2")))
                    // Intentional. Later we'll run Scan to find price error. Find
                    // items > 1000 in price.
                    .withNumber("Price", 2000).withString("Dimensions", "8.5 x 11.0 x
 1.5").withNumber("PageCount", 600)
                    .withBoolean("InPublication", false).withString("ProductCategory",
 "Book");
            table.putItem(item);

            // Add bikes.

            item = new Item().withPrimaryKey("Id", 201).withString("Title", "18-
Bike-201")
                    // Size, followed by some title.
                    .withString("Description", "201
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Mountain A")
                    // Trek, Specialized.
                    .withNumber("Price", 100).withStringSet("Color", new
 HashSet<String>(Arrays.asList("Red", "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 202).withString("Title", "21-
Bike-202")
                    .withString("Description", "202
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Brand-Company A").withNumber("Price", 200)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Green",
 "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 203).withString("Title", "19-
Bike-203")
                    .withString("Description", "203
 Description").withString("BicycleType", "Road")
                    .withString("Brand", "Brand-Company B").withNumber("Price", 300)
```

```
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
 "Green", "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 204).withString("Title", "18-
Bike-204")
                    .withString("Description", "204
 Description").withString("BicycleType", "Mountain")
                    .withString("Brand", "Brand-Company B").withNumber("Price", 400)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", 205).withString("Title", "20-
Bike-205")
                    .withString("Description", "205
 Description").withString("BicycleType", "Hybrid")
                    .withString("Brand", "Brand-Company C").withNumber("Price", 500)
                    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red",
 "Black")))
                    .withString("ProductCategory", "Bicycle");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }

    }

    private static void loadSampleForums(String tableName) {

        Table table = dynamoDB.getTable(tableName);

        try {

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("Name", "Amazon DynamoDB")
                    .withString("Category", "Amazon Web
 Services").withNumber("Threads", 2).withNumber("Messages", 4)
                    .withNumber("Views", 1000);
            table.putItem(item);
```

```
            item = new Item().withPrimaryKey("Name", "Amazon
S3").withString("Category", "Amazon Web Services")
                    .withNumber("Threads", 0);
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }
    }

    private static void loadSampleThreads(String tableName) {
        try {
            long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000); // 7
            // days
            // ago
            long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000); // 14
            // days
            // ago
            long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000); // 21
            // days
            // ago

            Date date1 = new Date();
            date1.setTime(time1);

            Date date2 = new Date();
            date2.setTime(time2);

            Date date3 = new Date();
            date3.setTime(time3);

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

            Table table = dynamoDB.getTable(tableName);

            System.out.println("Adding data to " + tableName);

            Item item = new Item().withPrimaryKey("ForumName", "Amazon DynamoDB")
                    .withString("Subject", "DynamoDB Thread 1").withString("Message",
"DynamoDB thread 1 message")
                    .withString("LastPostedBy", "User
A").withString("LastPostedDateTime", dateFormatter.format(date2))
```

```
                    .withNumber("Views", 0).withNumber("Replies",
0).withNumber("Answered", 0)
                    .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"primarykey", "table")));
            table.putItem(item);

            item = new Item().withPrimaryKey("ForumName", "Amazon
DynamoDB").withString("Subject", "DynamoDB Thread 2")
                    .withString("Message", "DynamoDB thread 2
message").withString("LastPostedBy", "User A")
                    .withString("LastPostedDateTime",
dateFormatter.format(date3)).withNumber("Views", 0)
                    .withNumber("Replies", 0).withNumber("Answered", 0)
                    .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"partitionkey", "sortkey")));
            table.putItem(item);

            item = new Item().withPrimaryKey("ForumName", "Amazon
S3").withString("Subject", "S3 Thread 1")
                    .withString("Message", "S3 Thread 3
message").withString("LastPostedBy", "User A")
                    .withString("LastPostedDateTime",
dateFormatter.format(date1)).withNumber("Views", 0)
                    .withNumber("Replies", 0).withNumber("Answered", 0)
                    .withStringSet("Tags", new
HashSet<String>(Arrays.asList("largeobjects", "multipart upload")));
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());
        }

    }

    private static void loadSampleReplies(String tableName) {
        try {
            // 1 day ago
            long time0 = (new Date()).getTime() - (1 * 24 * 60 * 60 * 1000);
            // 7 days ago
            long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000);
            // 14 days ago
            long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000);
            // 21 days ago
```

```
            long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000);

            Date date0 = new Date();
            date0.setTime(time0);

            Date date1 = new Date();
            date1.setTime(time1);

            Date date2 = new Date();
            date2.setTime(time2);

            Date date3 = new Date();
            date3.setTime(time3);

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

            Table table = dynamoDB.getTable(tableName);

            System.out.println("Adding data to " + tableName);

            // Add threads.

            Item item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB
Thread 1")
                    .withString("ReplyDateTime", (dateFormatter.format(date3)))
                    .withString("Message", "DynamoDB Thread 1 Reply 1
text").withString("PostedBy", "User A");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 1")
                    .withString("ReplyDateTime", dateFormatter.format(date2))
                    .withString("Message", "DynamoDB Thread 1 Reply 2
text").withString("PostedBy", "User B");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
                    .withString("ReplyDateTime", dateFormatter.format(date1))
                    .withString("Message", "DynamoDB Thread 2 Reply 1
text").withString("PostedBy", "User A");
            table.putItem(item);

            item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
                    .withString("ReplyDateTime", dateFormatter.format(date0))
```

```
                    .withString("Message", "DynamoDB Thread 2 Reply 2
 text").withString("PostedBy", "User A");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Failed to create item in " + tableName);
            System.err.println(e.getMessage());

        }
    }

}
```

AWS SDK for .NET を使用してサンプルテーブルを作成してデータをアップロードする

次の C# コード例は、テーブルを作成し、テーブルにデータをアップロードします。Visual Studio でこのコードを実行するための詳しい手順については、「.NET コード例」を参照してください。

```
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class CreateTablesLoadData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                //DeleteAllTables(client);
                DeleteTable("ProductCatalog");
                DeleteTable("Forum");
                DeleteTable("Thread");
```

```
            DeleteTable("Reply");

            // Create tables (using the AWS SDK for .NET low-level API).
            CreateTableProductCatalog();
            CreateTableForum();
            CreateTableThread(); // ForumTitle, Subject */
            CreateTableReply();

            // Load data (using the .NET SDK document API)
            LoadSampleProducts();
            LoadSampleForums();
            LoadSampleThreads();
            LoadSampleReplies();
            Console.WriteLine("Sample complete!");
            Console.WriteLine("Press ENTER to continue");
            Console.ReadLine();
        }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }

    private static void DeleteTable(string tableName)
    {
        try
        {
            var deleteTableResponse = client.DeleteTable(new DeleteTableRequest()
            {
                TableName = tableName
            });
            WaitTillTableDeleted(client, tableName, deleteTableResponse);
        }
        catch (ResourceNotFoundException)
        {
            // There is no such table.
        }
    }

    private static void CreateTableProductCatalog()
    {
        string tableName = "ProductCatalog";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
```

```
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Id",
                                AttributeType = "N"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement
                            {
                                AttributeName = "Id",
                                KeyType = "HASH"
                            }
                        },
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableForum()
    {
        string tableName = "Forum";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Name",
                                AttributeType = "S"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement
```

```
                                {
                                    AttributeName = "Name", // forum Title
                                    KeyType = "HASH"
                                }
                            },
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableThread()
    {
        string tableName = "Thread";

        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                            {
                                new AttributeDefinition
                                {
                                    AttributeName = "ForumName", // Hash attribute
                                    AttributeType = "S"
                                },
                                new AttributeDefinition
                                {
                                    AttributeName = "Subject",
                                    AttributeType = "S"
                                }
                            },
            KeySchema = new List<KeySchemaElement>()
                            {
                                new KeySchemaElement
                                {
                                    AttributeName = "ForumName", // Hash attribute
                                    KeyType = "HASH"
                                },
                                new KeySchemaElement
                                {
```

```
                                AttributeName = "Subject", // Range attribute
                                KeyType = "RANGE"
                            }
                        },
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 5
            }
        });

        WaitTillTableCreated(client, tableName, response);
    }

    private static void CreateTableReply()
    {
        string tableName = "Reply";
        var response = client.CreateTable(new CreateTableRequest
        {
            TableName = tableName,
            AttributeDefinitions = new List<AttributeDefinition>()
                        {
                            new AttributeDefinition
                            {
                                AttributeName = "Id",
                                AttributeType = "S"
                            },
                            new AttributeDefinition
                            {
                                AttributeName = "ReplyDateTime",
                                AttributeType = "S"
                            },
                            new AttributeDefinition
                            {
                                AttributeName = "PostedBy",
                                AttributeType = "S"
                            }
                        },
            KeySchema = new List<KeySchemaElement>()
                        {
                            new KeySchemaElement()
                            {
                                AttributeName = "Id",
                                KeyType = "HASH"
```

```
                                },
                                new KeySchemaElement()
                                {
                                    AttributeName = "ReplyDateTime",
                                    KeyType = "RANGE"
                                }
                            },
                LocalSecondaryIndexes = new List<LocalSecondaryIndex>()
                            {
                                new LocalSecondaryIndex()
                                {
                                    IndexName = "PostedBy_index",


                                    KeySchema = new List<KeySchemaElement>() {
                                        new KeySchemaElement() {
                                            AttributeName = "Id", KeyType = "HASH"
                                        },
                                        new KeySchemaElement() {
                                            AttributeName = "PostedBy", KeyType =
"RANGE"
                                        }
                                    },
                                    Projection = new Projection() {
                                        ProjectionType = ProjectionType.KEYS_ONLY
                                    }
                                }
                            },
                ProvisionedThroughput = new ProvisionedThroughput
                {
                    ReadCapacityUnits = 10,
                    WriteCapacityUnits = 5
                }
            });

            WaitTillTableCreated(client, tableName, response);
        }

        private static void WaitTillTableCreated(AmazonDynamoDBClient client, string
tableName,
                                 CreateTableResponse response)
        {
            var tableDescription = response.TableDescription;
```

```
            string status = tableDescription.TableStatus;

            Console.WriteLine(tableName + " - " + status);

            // Let us wait until table is created. Call DescribeTable.
            while (status != "ACTIVE")
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });
                    Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                              res.Table.TableStatus);
                    status = res.Table.TableStatus;
                }
                // Try-catch to handle potential eventual-consistency issue.
                catch (ResourceNotFoundException)
                { }
            }
        }

        private static void WaitTillTableDeleted(AmazonDynamoDBClient client, string
tableName,
                                 DeleteTableResponse response)
        {
            var tableDescription = response.TableDescription;

            string status = tableDescription.TableStatus;

            Console.WriteLine(tableName + " - " + status);

            // Let us wait until table is created. Call DescribeTable
            try
            {
                while (status == "DELETING")
                {
                    System.Threading.Thread.Sleep(5000); // wait 5 seconds

                    var res = client.DescribeTable(new DescribeTableRequest
                    {
```

```
                        TableName = tableName
                    });
                    Console.WriteLine("Table name: {0}, status: {1}",
res.Table.TableName,
                              res.Table.TableStatus);
                    status = res.Table.TableStatus;
                }
            }
            catch (ResourceNotFoundException)
            {
                // Table deleted.
            }
        }

        private static void LoadSampleProducts()
        {
            Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
            // ********** Add Books ********************
            var book1 = new Document();
            book1["Id"] = 101;
            book1["Title"] = "Book 101 Title";
            book1["ISBN"] = "111-1111111111";
            book1["Authors"] = new List<string> { "Author 1" };
            book1["Price"] = -2; // *** Intentional value. Later used to illustrate
scan.
            book1["Dimensions"] = "8.5 x 11.0 x 0.5";
            book1["PageCount"] = 500;
            book1["InPublication"] = true;
            book1["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book1);

            var book2 = new Document();

            book2["Id"] = 102;
            book2["Title"] = "Book 102 Title";
            book2["ISBN"] = "222-2222222222";
            book2["Authors"] = new List<string> { "Author 1", "Author 2" }; ;
            book2["Price"] = 20;
            book2["Dimensions"] = "8.5 x 11.0 x 0.8";
            book2["PageCount"] = 600;
            book2["InPublication"] = true;
            book2["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book2);
```

```
            var book3 = new Document();
            book3["Id"] = 103;
            book3["Title"] = "Book 103 Title";
            book3["ISBN"] = "333-3333333333";
            book3["Authors"] = new List<string> { "Author 1", "Author2", "Author
3" }; ;
            book3["Price"] = 2000;
            book3["Dimensions"] = "8.5 x 11.0 x 1.5";
            book3["PageCount"] = 700;
            book3["InPublication"] = false;
            book3["ProductCategory"] = "Book";
            productCatalogTable.PutItem(book3);

            // ************ Add bikes. ******************
            var bicycle1 = new Document();
            bicycle1["Id"] = 201;
            bicycle1["Title"] = "18-Bike 201"; // size, followed by some title.
            bicycle1["Description"] = "201 description";
            bicycle1["BicycleType"] = "Road";
            bicycle1["Brand"] = "Brand-Company A"; // Trek, Specialized.
            bicycle1["Price"] = 100;
            bicycle1["Color"] = new List<string> { "Red", "Black" };
            bicycle1["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle1);

            var bicycle2 = new Document();
            bicycle2["Id"] = 202;
            bicycle2["Title"] = "21-Bike 202Brand-Company A";
            bicycle2["Description"] = "202 description";
            bicycle2["BicycleType"] = "Road";
            bicycle2["Brand"] = "";
            bicycle2["Price"] = 200;
            bicycle2["Color"] = new List<string> { "Green", "Black" };
            bicycle2["ProductCategory"] = "Bicycle";
            productCatalogTable.PutItem(bicycle2);

            var bicycle3 = new Document();
            bicycle3["Id"] = 203;
            bicycle3["Title"] = "19-Bike 203";
            bicycle3["Description"] = "203 description";
            bicycle3["BicycleType"] = "Road";
            bicycle3["Brand"] = "Brand-Company B";
            bicycle3["Price"] = 300;
            bicycle3["Color"] = new List<string> { "Red", "Green", "Black" };
```

```
            bicycle3["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle3);

            var bicycle4 = new Document();
            bicycle4["Id"] = 204;
            bicycle4["Title"] = "18-Bike 204";
            bicycle4["Description"] = "204 description";
            bicycle4["BicycleType"] = "Mountain";
            bicycle4["Brand"] = "Brand-Company B";
            bicycle4["Price"] = 400;
            bicycle4["Color"] = new List<string> { "Red" };
            bicycle4["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle4);

            var bicycle5 = new Document();
            bicycle5["Id"] = 205;
            bicycle5["Title"] = "20-Title 205";
            bicycle4["Description"] = "205 description";
            bicycle5["BicycleType"] = "Hybrid";
            bicycle5["Brand"] = "Brand-Company C";
            bicycle5["Price"] = 500;
            bicycle5["Color"] = new List<string> { "Red", "Black" };
            bicycle5["ProductCategory"] = "Bike";
            productCatalogTable.PutItem(bicycle5);
        }

        private static void LoadSampleForums()
        {
            Table forumTable = Table.LoadTable(client, "Forum");

            var forum1 = new Document();
            forum1["Name"] = "Amazon DynamoDB"; // PK
            forum1["Category"] = "Amazon Web Services";
            forum1["Threads"] = 2;
            forum1["Messages"] = 4;
            forum1["Views"] = 1000;

            forumTable.PutItem(forum1);

            var forum2 = new Document();
            forum2["Name"] = "Amazon S3"; // PK
            forum2["Category"] = "Amazon Web Services";
            forum2["Threads"] = 1;
```

```
            forumTable.PutItem(forum2);
        }

        private static void LoadSampleThreads()
        {
            Table threadTable = Table.LoadTable(client, "Thread");

            // Thread 1.
            var thread1 = new Document();
            thread1["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
            thread1["Subject"] = "DynamoDB Thread 1"; // Range attribute.
            thread1["Message"] = "DynamoDB thread 1 message text";
            thread1["LastPostedBy"] = "User A";
            thread1["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14,
0, 0, 0));
            thread1["Views"] = 0;
            thread1["Replies"] = 0;
            thread1["Answered"] = false;
            thread1["Tags"] = new List<string> { "index", "primarykey", "table" };

            threadTable.PutItem(thread1);

            // Thread 2.
            var thread2 = new Document();
            thread2["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
            thread2["Subject"] = "DynamoDB Thread 2"; // Range attribute.
            thread2["Message"] = "DynamoDB thread 2 message text";
            thread2["LastPostedBy"] = "User A";
            thread2["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21,
0, 0, 0));
            thread2["Views"] = 0;
            thread2["Replies"] = 0;
            thread2["Answered"] = false;
            thread2["Tags"] = new List<string> { "index", "primarykey", "rangekey" };

            threadTable.PutItem(thread2);

            // Thread 3.
            var thread3 = new Document();
            thread3["ForumName"] = "Amazon S3"; // Hash attribute.
            thread3["Subject"] = "S3 Thread 1"; // Range attribute.
            thread3["Message"] = "S3 thread 3 message text";
            thread3["LastPostedBy"] = "User A";
```

```
            thread3["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0,
0, 0));
            thread3["Views"] = 0;
            thread3["Replies"] = 0;
            thread3["Answered"] = false;
            thread3["Tags"] = new List<string> { "largeobjects", "multipart upload" };
            threadTable.PutItem(thread3);
        }

        private static void LoadSampleReplies()
        {
            Table replyTable = Table.LoadTable(client, "Reply");

            // Reply 1 - thread 1.
            var thread1Reply1 = new Document();
            thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21,
0, 0, 0)); // Range attribute.
            thread1Reply1["Message"] = "DynamoDB Thread 1 Reply 1 text";
            thread1Reply1["PostedBy"] = "User A";

            replyTable.PutItem(thread1Reply1);

            // Reply 2 - thread 1.
            var thread1reply2 = new Document();
            thread1reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14,
0, 0, 0)); // Range attribute.
            thread1reply2["Message"] = "DynamoDB Thread 1 Reply 2 text";
            thread1reply2["PostedBy"] = "User B";

            replyTable.PutItem(thread1reply2);

            // Reply 3 - thread 1.
            var thread1Reply3 = new Document();
            thread1Reply3["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash
attribute.
            thread1Reply3["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7,
0, 0, 0)); // Range attribute.
            thread1Reply3["Message"] = "DynamoDB Thread 1 Reply 3 text";
            thread1Reply3["PostedBy"] = "User B";
```

```
            replyTable.PutItem(thread1Reply3);

            // Reply 1 - thread 2.
            var thread2Reply1 = new Document();
            thread2Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
 attribute.
            thread2Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7,
 0, 0, 0)); // Range attribute.
            thread2Reply1["Message"] = "DynamoDB Thread 2 Reply 1 text";
            thread2Reply1["PostedBy"] = "User A";


            replyTable.PutItem(thread2Reply1);

            // Reply 2 - thread 2.
            var thread2Reply2 = new Document();
            thread2Reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash
 attribute.
            thread2Reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(1,
 0, 0, 0)); // Range attribute.
            thread2Reply2["Message"] = "DynamoDB Thread 2 Reply 2 text";
            thread2Reply2["PostedBy"] = "User A";

            replyTable.PutItem(thread2Reply2);
        }
    }
}
```

AWS SDK for Python (Boto) を使用した DynamoDB のサンプルアプリケーション: Tic-tac-toe

この Tic-Tac-Toe ゲームのサンプルウェブアプリケーションは、Amazon DynamoDB 上に構築されます。このアプリケーションでは AWS SDK for Python (Boto) を使用して、DynamoDB テーブルにゲームデータを保存するのに必要な DynamoDB 呼び出しを行います。また、Python ウェブフレームワーク Flask を使用して、データのモデル化の方法を含む DynamoDB でのアプリケーション開発について、包括的に見ることができます。また、DynamoDB でのデータのモデル化に関するベストプラクティスを示します。これには、ゲームアプリケーション用に作成するテーブル、ユーザーが定義するプライマリキー、クエリ要件に基づいて必要になる追加のインデックス、および連結された属性値の使用などが含まれます。

Tic-Tac-Toe アプリケーションは、次のようにしてウェブ上でプレイします。

1. アプリケーションのホームページにログインします。
2. 次に、他のユーザーを対戦相手としてゲームに招待します。

   他のユーザーが招待を受け入れるまで、ゲームのステータスは PENDING となります。対戦相手が招待を受け入れると、ステータスは IN_PROGRESS に変わります。
3. ゲームは、対戦相手がログインして招待を受け入れてから開始されます。
4. このアプリケーションでは、ゲームのすべての動きやステータス情報は、DynamoDB テーブルに格納されます。
5. ゲームは優勝または引き分けで終わり、これによりゲームのステータスは FINISHED に設定されます。

アプリケーション構築についてすべて網羅した演習を各ステップで示します。

- ステップ 1: ローカルにデプロイおよびテストを実行します – このセクションでは、アプリケーションをローカルコンピュータにダウンロード、デプロイし、そのテストを行います。DynamoDB のダウンロード可能バージョンで、必要なテーブルを作成していきます。
- ステップ 2: データモデルと実装の詳細を調べます – このセクションでは、最初にインデックスや連結された属性値の使用を含め、データモデルの詳細について説明します。次に、アプリケーションの動作について説明します。
- ステップ 3: DynamoDB サービスを使用して実稼働環境のデプロイを行います – このセクションでは、実稼働環境でのデプロイの考慮事項について説明します。このステップでは、Amazon DynamoDB サービスを使用してテーブルを作成し、AWS Elastic Beanstalk を使用してアプリケーションをデプロイします。実稼働状態のアプリケーションがある場合は、アプリケーションが DynamoDB テーブルにアクセスできるようにするため、適切なアクセス許可を付与します。このセクションの手順では、本稼働デプロイについて詳細に説明します。
- ステップ 4: リソースをクリーンアップする – このセクションでは、このサンプルアプリケーションでカバーしていない領域に目を向けます。また、料金が発生しないようにするため、前のステップで作成した AWS リソースを削除するためのステップについても説明します。

ステップ 1: ローカルにデプロイおよびテストを実行します

トピック

- 1.1: 必要なパッケージのダウンロードとインストール
- 1.2: ゲームアプリケーションをテストします

このステップでは、Tic-Tac-Toe ゲームアプリケーションをローカルコンピュータにダウンロード、デプロイ、およびテストします。Amazon DynamoDB ウェブサービスを使用する代わりに、DynamoDB をコンピュータにダウンロードして必要なテーブルを作成します。

1.1: 必要なパッケージのダウンロードとインストール

このアプリケーションをローカルでテストするには、次のものが必要です。

- Python
- Flask (Python 用のマイクロフレームワーク)
- AWS SDK for Python (Boto)
- ローカルコンピュータ上で実行される DynamoDB
- Git

これらのツールを入手するには、以下の作業を実行します。

1. Python をインストールします。詳しい手順については、「Python のダウンロード」を参照してください。

   Tic-Tac-Toe アプリケーションは、Python バージョン 2.7 を使用してテスト済みです。
2. Python パッケージインストーラ (PIP) を使用して Flask および AWS SDK for Python (Boto) をインストールします。
   - PIP をインストールします。

     手順については、「PIP のインストール」を参照してください。インストールページで、get-pip.py リンクを選択して、ファイルを保存します。次に、管理者としてコマンドターミナルを開き、コマンドプロンプトで以下のように入力します。

     ```
     python.exe get-pip.py
     ```

     Linux では、`.exe` 拡張子を指定しません。`python get-pip.py` のみを指定します。
   - PIP で、次に示すコードを使用して Flask と Boto パッケージをインストールします。

     ```
     pip install Flask
     pip install boto
     pip install configparser
     ```

3. DynamoDB をローカルのコンピュータにダウンロードします。その実行方法については、「DynamoDB local (ダウンロード可能バージョン) のセットアップ」を参照してください。
4. Tic-Tac-Toe アプリケーションをダウンロードします。
   a. Git をインストールします。手順については、「Git のダウンロード」を参照してください。
   b. アプリケーションをダウンロードするには、次のコードを実行します。

```
git clone https://github.com/awslabs/dynamodb-tictactoe-example-app.git
```

1.2: ゲームアプリケーションをテストします

Tic-Tac-Toe アプリケーションをテストするには、ローカルコンピュータで DynamoDB を実行する必要があります。

tic-tac-toe アプリケーションを実行するには

1. DynamoDB を起動します。
2. Tic-Tac-Toe アプリケーション用のウェブサーバーを起動します。

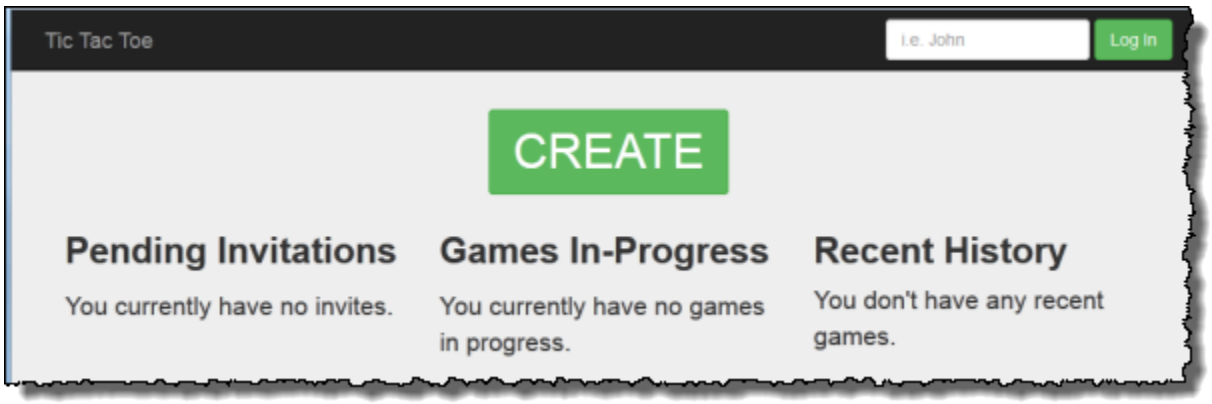
   そのためには、コマンドターミナルを開き、Tic-Tac-Toe アプリケーションをダウンロードしたフォルダに移動し、次に示すコードを使用してアプリケーションをローカルに実行します。

```
python.exe application.py --mode local --serverPort 5000 --port 8000
```

   Linux では、`.exe` 拡張子を指定しません。
3. ウェブブラウザを開き、次のように入力します。

```
http://localhost:5000/
```

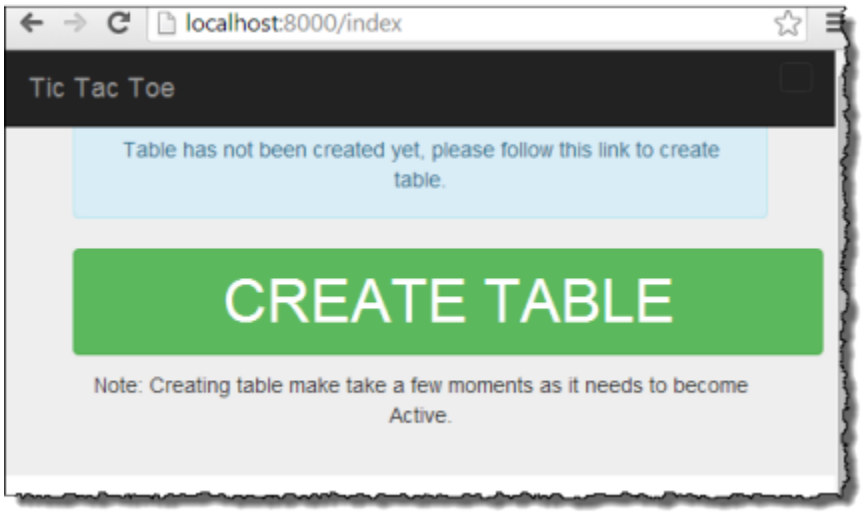
   ブラウザにホームページが表示されます。



4. [Log in] (ログイン) ボックスに **`user1`** と入力し、user1 としてログインします。

> **Note**
>
> このサンプルアプリケーションは、ユーザー認証を実行しません。ユーザー ID のみがプレーヤーを識別するために使用されます。2 人のプレーヤーが同じエイリアスでログインすると、アプリケーションは 2 つの別のブラウザでプレイしているかのように動作します。

5. 初めてゲームを実行する場合は、DynamoDB で必要なテーブル (`Games`) を作成するように促すページが表示されます。[テーブルの作成] を選択します。



6. [作成] を選択して、最初の Tic-Tac-Toe ゲームを作成します。
7. [Choose an Opponent] (対戦相手の選択) ボックスに **`user2`** と入力し、[Create Game!] (ゲームを作成する!) を選択します。



この操作を行うと、Games テーブルに項目を追加してゲームが作成されます。これにより、ゲームのステータスが PENDING に設定されます。

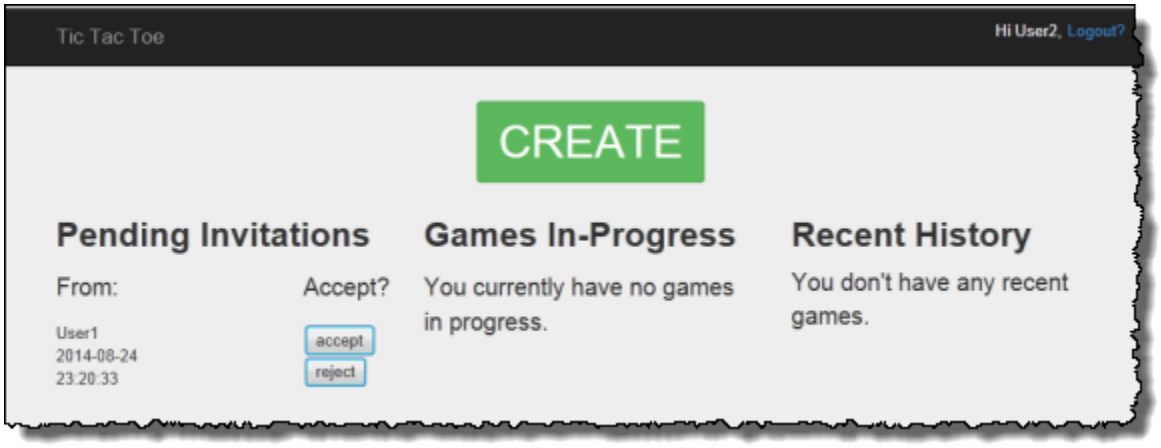
8. 別のブラウザウィンドウを開き、次のように入力します。

```
http://localhost:5000/
```

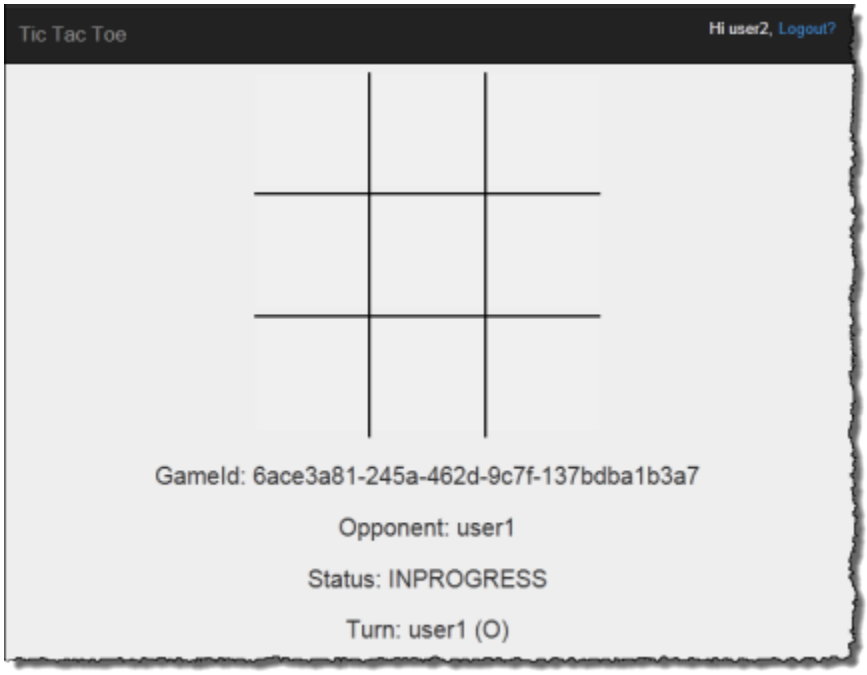
ブラウザは Cookie を通じて情報を渡すので、Cookie を引き継がないように、匿名モードまたはプライベートブラウジングを使用します。

9. user2 としてログインします。

user1 からの保留中の招待を示すページが表示されます。



10. [accept (受け入れる)] を選択して招待を受け入れます。



ゲームページが、空の Tic-Tac-Toe グリッドとともに表示されます。このページには、ゲーム ID、誰の番か、ゲームのステータスなど、関連するゲーム情報も表示されます。

11. ゲームをプレイします。

ユーザーが動くたびに、ウェブサービスは、Games テーブルのゲーム項目に条件付き更新を実行するためのリクエストを DynamoDB に送信します。例えば、条件により、動きが有効で、ユーザーが選択したマス目が利用可能で、動きを行ったユーザーの番であったことを確認できます。動きが有効な場合、更新操作により、ボードでの選択に対応する新しい属性が追加されます。また、既存の属性値が、次に動くことができるユーザーに設定されます。

アプリケーションは、ゲームページから非同期の JavaScript 呼び出しを毎秒、最大 5 分にわたって実行し、DynamoDB 内のゲームの状態に変化があったかどうかを確認します。変わった場合、アプリケーションは新しい情報でページを更新します。5 分後に、アプリケーションはリクエストを中止します。ユーザーはページを更新して、更新された情報を取得する必要があります。

ステップ 2: データモデルと実装の詳細を調べます

トピック

- 2.1: 基本的なデータモデル
- 2.2: 実行中のアプリケーション (コードのウォークスルー)

2.1: 基本的なデータモデル

このサンプルアプリケーションでは、次の DynamoDB データモデルの概念を説明します。

- テーブル – DynamoDB では、テーブルは項目 (つまりレコード) の集合であり、各項目には、属性と呼ばれる名前と値のペアが集約されています。

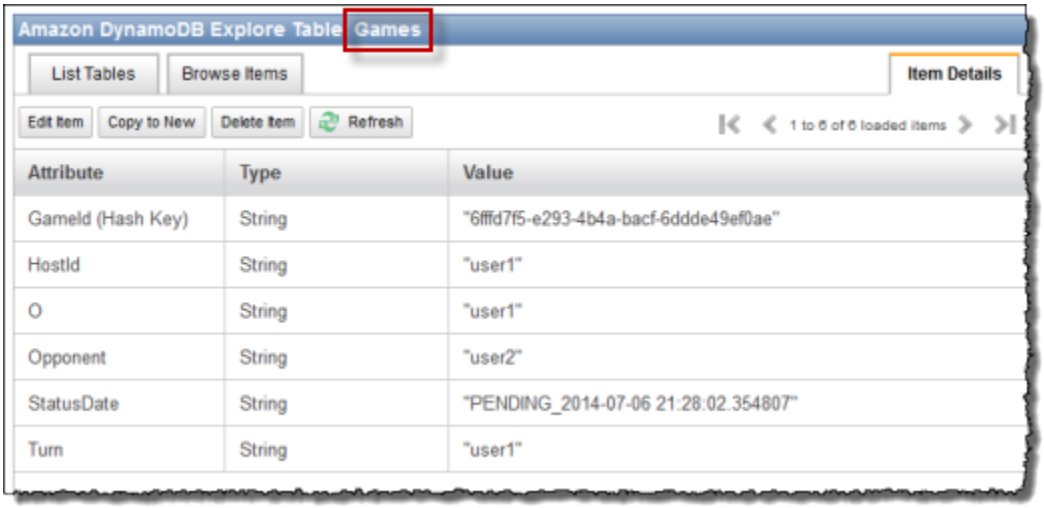
  この Tic-Tac-Toe の例では、アプリケーションは、Games テーブル内のすべてのゲームデータを保存します。アプリケーションはゲームごとにテーブルで 1 つの項目を作成し、すべてのゲームデータを属性として格納します。Tic-Tac-Toe ゲームでは最大 9 回の移動が可能です。DynamoDB テーブルは、プライマリキーのみが必須の属性である場合はスキーマを持たないので、アプリケーションは、ゲーム項目 ごとに異なる数の属性を格納できます。

  Games テーブルには、文字列型の 1 つの属性 GameIdで構成されるシンプルなプライマリキーがあります。アプリケーションは各ゲームに一意の ID を割り当てます。DynamoDB でのプライマリキーの詳細については、「プライマリキー」を参照してください。

  ユーザーが他のユーザーを招待して Tic-Tac-Toe ゲームを開始すると、アプリケーションは、次のようなゲームメタデータを保存する属性で、Games テーブルに新しい項目を作成します。

  - HostIdゲームを開始したユーザーである 。
  - Opponentゲームに招待されたユーザーである 。
  - プレイする番のユーザー。最初にゲームを開始したユーザー。
  - ボードで O 記号を使用するユーザー。ゲームを開始するユーザーは O 記号を使用します。

  さらに、アプリケーションは StatusDate 連結属性を作成し、ゲームの初期状態を PENDING としてマークします。次のスクリーンショットに、DynamoDB コンソールに表示される項目の例を示します。



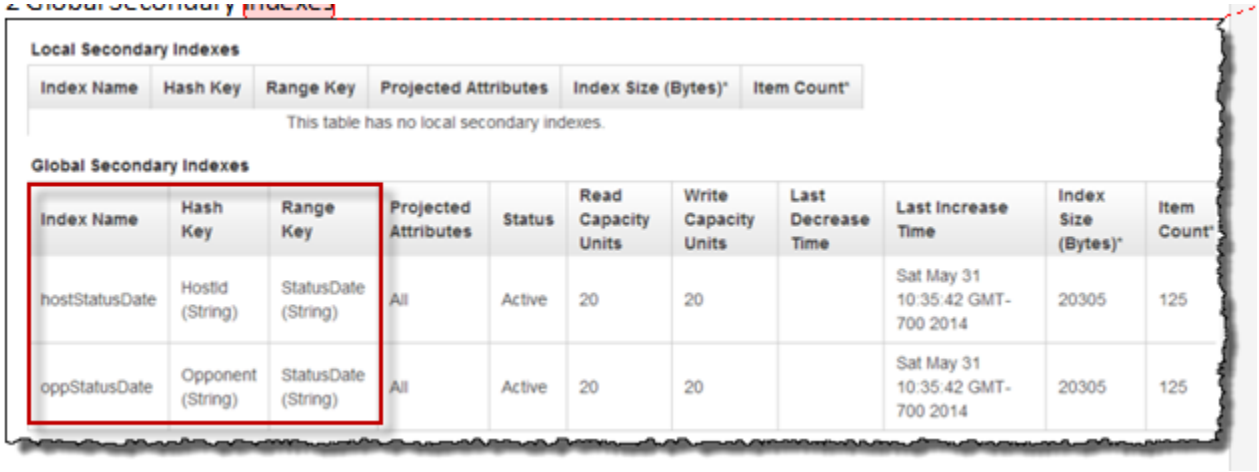
アプリケーションは、ゲームの進行に合わせて、ゲームで動きがあるたびに 1 つの属性をテーブルに追加します。属性名はボードの位置 (TopLeft、BottomRight など) です。たとえば、動きには値 TopLeftの 0 属性、値 TopRight の 0 属性、および値 BottomRightの X 属性があるなどです。属性値は、動いたユーザーに応じて、0 または X になります。たとえば、次のボードを考えてみます。



- 連結属性値 – StatusDate 属性が、連結値属性を示します。この手法では、ゲームのステータス (PENDING、IN_PROGRESS、FINISHED) および日付 (最後の動き) を格納する個別の属性を作成する代わりに、IN_PROGRESS_2014-04-30 10:20:32 などの単一の属性としてそれらを組み合わせます。

  次に、アプリケーションは、インデックスのソートキーとして StatusDate を指定して、セカンダリインデックスの作成で StatusDate 属性を使用します。StatusDate 連結値属性を使用する利点について、さらに次のインデックスの説明で示します。

- グローバルセカンダリインデックス – テーブルのプライマリキー、GameId を使用すると、テーブルを効率的に照会してゲーム項目を検索できます。プライマリキー属性以外の属性でテーブルをクエリするため、DynamoDB ではセカンダリインデックスの作成をサポートしています。このサンプルアプリケーションでは、次の 2 つのセカンダリインデックスを構築します。



Local Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes	Index Size (Bytes)*	Item Count*

This table has no local secondary indexes.

Global Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes	Status	Read Capacity Units	Write Capacity Units	Last Decrease Time	Last Increase Time	Index Size (Bytes)*	Item Count*
hostStatusDate	HostId (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125
oppStatusDate	Opponent (String)	StatusDate (String)	All	Active	20	20		Sat May 31 10:35:42 GMT-700 2014	20305	125

  - HostId-StatusDate-index。このインデックスはパーティションキーとして HostId を、ソートキーとして StatusDate を持ちます。このインデックスを使用して、たとえば特定のユーザーによってホストされたゲームを検索するなど、HostId でクエリを実行できます。
  - OpponentId-StatusDate-index。このインデックスはパーティションキーとして OpponentId を、ソートキーとして StatusDate を持ちます。たとえば、特定のユーザーが対戦相手であるゲームを検索するなど、このインデックスを使用して Opponent でクエリを実行できます。

これらのインデックスは、グローバルセカンダリインデックスと呼ばれます。これは、インデックスのパーティションキーが、テーブルのプライマリキーで使用されるパーティションキー (GameId) とは同じでないためです。

両方のインデックスがソートキーとして StatusDate を指定することに注意してください。これを行うと、次のことが可能になります。

- BEGINS_WITH 比較演算子を使用してクエリを実行できます。たとえば、特定のユーザーによってホストされた IN_PROGRESS 属性を持つすべてのゲームを検索できます。この場合、BEGINS_WITH 演算子は、StatusDate で始まる IN_PROGRESS 値を確認します。
- DynamoDB は、項目をソートキー値によって整列させ、インデックスに保存します。したがって、すべてのステータスのプレフィックスは同じ (たとえば、IN_PROGRESS) になり、日付部分に使用される ISO 形式には、最も古いものから最も新しいものの順にソートされた項目が含まれます。この手法により、たとえば次のように特定のクエリを効率的に実行できるようになります。

- ログインしているユーザーによってホストされている最新の IN_PROGRESS のゲームを最大 10 個取得します。このクエリでは、HostId-StatusDate-index インデックスを指定します。
- ログインしているユーザーが対戦相手である最新の IN_PROGRESS のゲームを最大 10 個取得します。このクエリでは、OpponentId-StatusDate-index インデックスを指定します。

セカンダリインデックスの詳細については、「DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上」を参照してください。

2.2: 実行中のアプリケーション (コードのウォークスルー)

このアプリケーションには 2 つのメインページがあります。

- ホームページ – このページには、簡単なログイン、新しい tic-tac-toe ゲームを作成する [CREATE] (作成) ボタン、進行中のゲームのリスト、ゲームの履歴、およびアクティブな保留中のゲームの招待が表示されます。

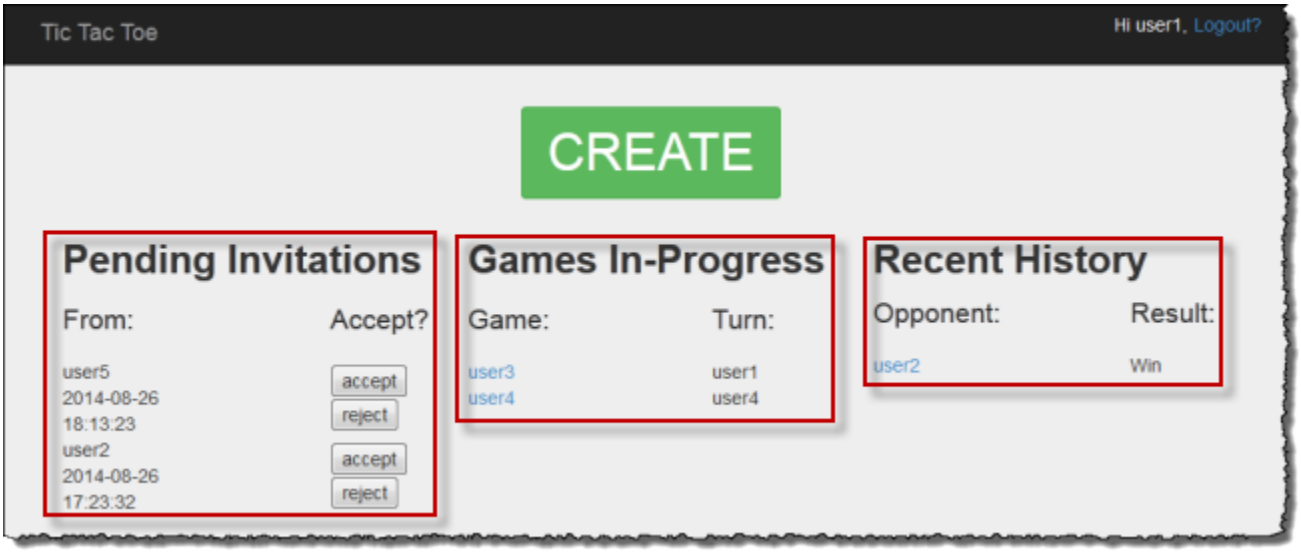
  ホームページは自動的に更新されません。リストを更新するには、ユーザーがページを更新する必要があります。
- ゲームページ – このページには、ユーザーがプレイするための、tic-tac-toe グリッドが表示されます。

  アプリケーションは、毎秒ゲームページを自動的に更新します。ブラウザの JavaScript が毎秒 Python ウェブサーバーを呼び出して、テーブルのゲーム項目が変更されたかどうかを Games テーブルに照会します。変更された場合、JavaScript はページ更新をトリガーし、更新されたボードがユーザーに表示されるようにします。

アプリケーションの動作について詳細に説明します。

ホームページ

ユーザーがログインすると、アプリケーションには、以下の 3 つの情報のリストが表示されます。



- 招待 – このリストには、ログインしているユーザーが受け入れを保留中の、他のユーザーからの最新の招待が最大 10 個表示されます。前のスクリーンショットで、user1 は user5 および user2 からの招待を保留中です。
- 進行中のゲーム – このリストには、進行中の最新のゲームが最大 10 個表示されます。これらは、ユーザーがアクティブに実行中のゲームで、そのステータスは IN_PROGRESS です。スクリーンショットでは、user1 は user3 および user4 と Tic-Tac-Toe ゲームをアクティブにプレイ中です。
- 最近の履歴 – このリストには、ユーザーが終了した最近のゲームが最大 10 個表示されます。そのステータスは FINISHED です。スクリーンショットに示したゲームで、user1 は以前に user2 とプレイしています。完了した各ゲームについて、ゲームの結果がリストに表示されます。

コードで、index 関数は (application.py で) 次の 3 つの呼び出しを行い、ゲームのステータス情報を取得します。

```
inviteGames     = controller.getGameInvites(session["username"])
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames   = controller.getGamesWithStatus(session["username"], "FINISHED")
```

これらの呼び出しはそれぞれ、Game オブジェクトでラップされた、DynamoDB からの項目のリストを返します。ビューでこれらのオブジェクトからデータを抽出することは簡単です。インデックス関数は、HTML を表示するため、これらのオブジェクトリストをビューに渡します。

```
return render_template("index.html",
                       user=session["username"],
                       invites=inviteGames,
```

```
                    inprogress=inProgressGames,
                    finished=finishedGames)
```

Tic-Tac-Toe アプリケーションは、主に DynamoDB から取得したゲームデータを格納するため、Game クラスを定義します。これらの関数は、Amazon DynamoDB の項目に関連するコードからアプリケーションの他の部分を分離できるようにするために、Game オブジェクトのリストを返します。これらの関数により、このようにしてデータストア層の詳細からアプリケーションコードを切り離すことができます。

ここで説明するアーキテクチャーパターンは、model-view-controller (MVC) UI パターンとも呼ばれます。この場合、Game オブジェクトのインスタンス (データを表す) はモデルで、HTML ページはビューです。コントローラーは 2 つのファイルに分割されます。application.py ファイルには Flask フレームワーク用のコントローラーロジックがあり、ビジネスロジックは gameController.py ファイルに分離されます。つまり、このアプリケーションでは、DynamoDB SDK に関連のあるすべてが、dynamodb フォルダ内にある独自の個別ファイルに格納されます。

3 つの関数と、それらが該当データを取得するためにグローバルセカンダリインデックスを使用して Games テーブルを照会する方法について説明します。

getGameInvites を使用して保留中のゲームの招待リストを取得します

getGameInvites 関数は保留中の最新の 10 個の招待リストを取得します。これらのゲームはユーザーによって作成されましたが、対戦相手はゲームの招待を受け入れていません。これらのゲームの場合、対戦相手が招待を受け入れるまでステータスは PENDING のままになります。対戦相手が招待を辞退した場合、アプリケーションは、対応する項目をテーブルから削除します。

関数は、以下のようなクエリを指定します。

- 関数は、以下のキー値および比較演算子とともに使用する OpponentId-StatusDate-index インデックスを指定します。
  - パーティションキーは OpponentId で、インデックスキー *user ID* を受け取ります。
  - ソートキーは StatusDate で、比較演算子およびインデックスキー値 beginswith="PENDING_" を受け取ります。

  OpponentId-StatusDate-index インデックスを使用して、ログインしているユーザーが招待されているゲーム、つまりログインしているユーザーが対戦相手であるゲームを取得します。
- クエリは結果を 10 項目に制限します。

```
gameInvitesIndex = self.cm.getGamesTable().query(
                                                Opponent__eq=user,
                                                StatusDate__beginswith="PENDING_",
                                                index="OpponentId-StatusDate-index",
                                                limit=10)
```

このインデックスでは、OpponentId (パーティションキー) ごとに、DynamoDB が項目を StatusDate (ソートキー) により整列させて保持しています。そのため、クエリが返すゲームは最新の 10 個のゲームになります。

getGamesWithStatus を使用して特定のステータスのゲームリストの取得します

対戦相手がゲームの招待を受け入れると、ゲームのステータスは IN_PROGRESS に変わります。ゲームが完了すると、ステータスは FINISHED に変わります。

進行中または終了済みのゲームを検索するクエリは、ステータス値が異なる場合を除いて同じです。したがって、アプリケーションは getGamesWithStatus 関数を定義し、この関数はステータス値をパラメータとして受け取ります。

```
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames   = controller.getGamesWithStatus(session["username"], "FINISHED")
```

次のセクションでは進行中のゲームについて説明しますが、終了済みのゲームにも同じ説明が当てはまります。

特定のユーザーの進行中のゲームのリストには、次の両方が含まれます。

- ユーザーによってホストされた進行中のゲーム
- ユーザーが対戦相手となっている進行中のゲーム

getGamesWithStatus 関数は、毎回適切なセカンダリインデックスを使用して次の 2 つのクエリを実行します。

- この関数は Games インデックスを使用して HostId-StatusDate-index テーブルをクエリします。このインデックスのためにクエリは、プライマリキー値つまりパーティションキー (HostId) およびソートキー (StatusDate) の値と、比較演算子を指定します。

  ```
  hostGamesInProgress = self.cm.getGamesTable ().query(HostId__eq=user,
  ```

```
                                                StatusDate__beginswith=status,
                                                index="HostId-StatusDate-index",
                                                limit=10)
```

比較演算子の Python の構文に注意してください。

- `HostId__eq=user` は等価比較演算子を指定します。
- `StatusDate__beginswith=status` は `BEGINS_WITH` 比較演算子を指定します。

- この関数は `Games` インデックスを使用して `OpponentId-StatusDate-index` テーブルをクエリします。

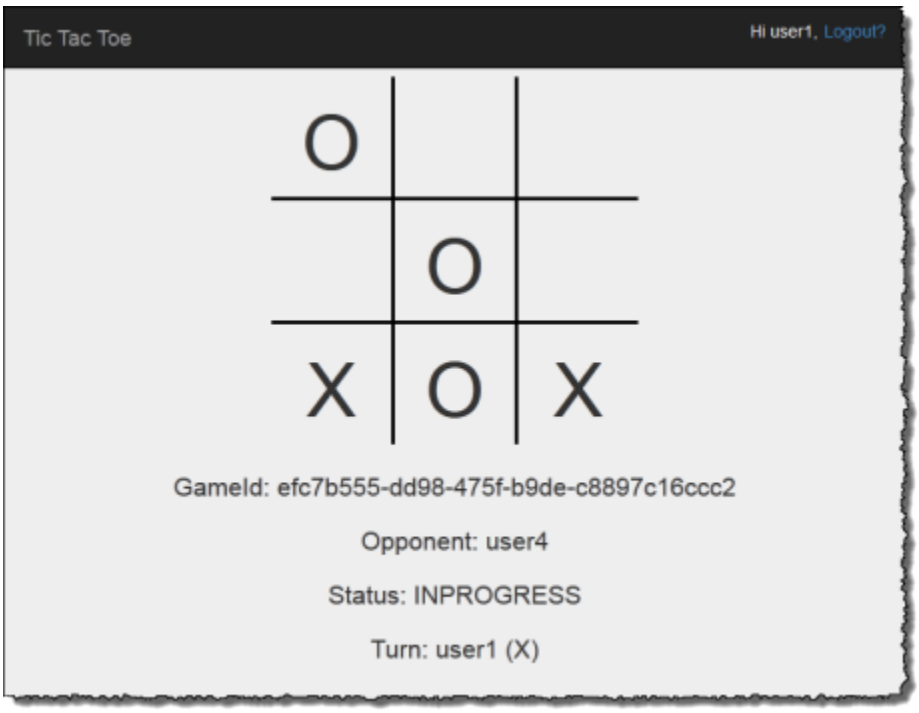
```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,
                                                   StatusDate__beginswith=status,
                                                   index="OpponentId-StatusDate-index",
                                                   limit=10)
```

- 次に、この関数は 2 つのリストを組み合わせ、ソートし、最初の 0 ~ 10 項目について `Game` オブジェクトのリストを作成して、呼び出し元関数 (インデックス) にリストを返します。

```
games = self.mergeQueries(hostGamesInProgress,
                          oppGamesInProgress)
return games
```

ゲームページ

ゲームページは、ユーザーが Tic-Tac-Toe ゲームをプレイする場所です。このページには、ゲームの関連情報とともに、ゲームグリッドが表示されます。次のスクリーンショットは、進行中のサンプルゲームを示しています。



アプリケーションは次の状況でゲームページを表示します。

- ユーザーは他のユーザーを招待してゲームを作成します。

  この場合、ページはホストやゲームのステータスを PENDING としてユーザーに表示し、対戦相手が受け入れるのを待ちます。
- ユーザーは、ホームページで保留中の招待の 1 つを受け入れます。

  この場合、ページではユーザーは対戦相手として、ゲームのステータスは IN_PROGRESS として表示されます。

ボード上でのユーザーによる選択操作により、アプリケーションへのフォーム POST リクエストが生成されます。つまり、Flask は HTML フォームデータとともに selectSquare 関数を (application.py で) 呼び出します。次に、この関数は updateBoardAndTurn 関数を (gameController.py で) 呼び出して、次のようにゲーム項目を更新します。

- これにより、動きに固有の新しい属性が追加されます。
- Turn 属性が、次の番のユーザーに更新されます。

```
controller.updateBoardAndTurn(item, value, session["username"])
```

項目の更新が成功した場合、関数は true を返します。それ以外の場合は、false を返します。updateBoardAndTurn 関数について、以下の点に注意してください。

- この関数は、既存の項目に対する一定範囲の更新を実行するため、SDK for Python (Boto) の update_item 関数を呼び出します。この関数は、DynamoDB の UpdateItem オペレーションにマッピングされます。詳細については、「UpdateItem」を参照してください。

> **Note**
>
> UpdateItem オペレーションと PutItem オペレーションの違いは、PutItem が項目全体を置き換えることです。詳細については、「PutItem」を参照してください。

update_item 呼び出しでは、コードは以下を識別します。

- Games テーブルのプライマリキー (ItemId)。

```
key = { "GameId" : { "S" : gameId } }
```

- 現在のユーザーの動きに固有の、追加する新しい属性とその値 (例: TopLeft="X") 。

```
attributeUpdates = {
    position : {
        "Action" : "PUT",
        "Value" : { "S" : representation }
    }
}
```

- 更新が実行されるために満たされる必要がある条件
  - ゲームは進行中である必要があります。つまり、StatusDate 属性値は IN_PROGRESS で始まる必要があります。
  - 現在の番は、Turn 属性で指定されている有効なユーザーの番である必要があります。
  - ユーザーが選択した四角形は使用可能である必要があります。つまり、四角形に対応する属性は存在していてはなりません。

```
expectations = {"StatusDate" : {"AttributeValueList": [{"S" : "IN_PROGRESS_"}],
    "ComparisonOperator": "BEGINS_WITH"},
    "Turn" : {"Value" : {"S" : current_player}},
    position : {"Exists" : False}}
```

ここで、関数は update_item を呼び出して項目を更新します。

```
self.cm.db.update_item("Games", key=key,
    attribute_updates=attributeUpdates,
    expected=expectations)
```

関数に戻ると、selectSquare 関数呼び出しは次の例に示すようにリダイレクトされます。

```
redirect("/game="+gameId)
```

この呼び出しにより、ブラウザが更新されます。この更新の一環として、アプリケーションはゲームが優勝または引き分けで終了したかどうかを確認します。終了した場合、アプリケーションはそれに応じてゲーム項目を更新します。

ステップ 3: DynamoDB サービスを使用して実稼働環境のデプロイを行います

トピック

- 3.1: Amazon EC2 向けの IAM ロールを作成します
- 3.2: Amazon DynamoDB でゲームテーブルを作成します
- 3.3: tic-tac-toe アプリケーションコードのバンドルとデプロイ
- 3.4: AWS Elastic Beanstalk 環境の設定

前のセクションでは、DynamoDB local を使用して、ローカルコンピュータ上で Tic-Tac-Toe アプリケーションをデプロイし、テストを行いました。ここでは、次のようにして本稼働環境でアプリケーションをデプロイします。

- ウェブアプリケーションやサービスをデプロイ、スケーリングするための使いやすいサービスである AWS Elastic Beanstalk を使用して、アプリケーションをデプロイします。詳細については、「AWS Elastic Beanstalk への Flask アプリケーションのデプロイ」を参照してください。

  Elastic Beanstalk により、1 つ以上の Amazon Elastic Compute Cloud (Amazon EC2) インスタンスが起動されます。Tic-Tac-Toe の設定は、そのアプリケーションが実行される Elastic Beanstalk を通じて行われます。
- Amazon DynamoDB サービスを使用して、ローカルのコンピュータ上ではなく、AWS 上に Games テーブルを作成します。

さらに、アクセス許可も設定する必要があります。DynamoDB の Games テーブルなど、ここで作成する AWS リソースは、デフォルトでプライベートになります。リソース所有者 (Games テーブルを作成した AWS アカウント) のみが、このテーブルにアクセスできます。したがって、デフォルトでは Tic-Tac-Toe アプリケーションは Games テーブルを更新することはできません。

必要なアクセス許可を付与するには、AWS Identity and Access Management (IAM) ロールを作成し、このロールに Games テーブルにアクセスするアクセス許可を付与します。Amazon EC2 インスタンスが、最初にこのロールを引き受けます。その後 AWS は、Amazon EC2 インスタンスが Tic-Tac-Toe アプリケーションに代わって Games テーブルを更新するために使用できる、一時的なセキュリティ認証情報を返します。ユーザーが Elastic Beanstalk アプリケーションを設定する際には、Amazon EC2 インスタンス、もしくは他のインスタンスが引き受けることができるIAM ロールを指定します。IAM ロールの詳細については、「Amazon EC2 ユーザーガイド」の「Amazon EC2 の IAM ロール」を参照してください。

> **Note**
>
> Tic-Tac-Toe アプリケーション用の Amazon EC2 インスタンスを作成する前に、Elastic Beanstalk がインスタンスを作成する AWS リージョンを、最初に決定する必要があります。Elastic Beanstalk アプリケーションの作成が完了したら、その作成先と同じリージョン名とエンドポイントを設定ファイルで指定します。Tic-Tac-Toe アプリケーションはこのファイルの情報を使用して Games テーブルを作成し、それ以降のリクエストを特定の AWS リージョンに送信します。Elastic Beanstalk が起動する DynamoDB Games テーブルと Amazon EC2 インスタンスの両方が同じリージョンにある必要があります。　利用可能なリージョンのリストについては、「Amazon Web Services 全般のリファレンス」の「Amazon DynamoDB」を参照してください。

要約すると、Tic-Tac-Toe アプリケーションを本稼働環境にデプロイするには、以下の作業を行います。

1. IAM サービスを使用して、IAM ロールを作成します。DynamoDB アクションが Games テーブルにアクセスするためのアクセス許可を付与するポリシーを、このロールにアタッチします。
2. Tic-Tac-Toe アプリケーションコードおよび設定ファイルをバンドルし、.zip ファイルを作成します。この .zip ファイルを使用して、サーバーに配置する Tic-Tac-Toe アプリケーションコードを Elastic Beanstalk に渡します。バンドルの作成に関する詳細については、「AWS Elastic Beanstalk デベロッパーガイド」の「アプリケーションソースバンドルを作成する」を参照してください。

設定ファイル (beanstalk.config) で、AWS リージョンおよびエンドポイント情報を指定します。Tic-Tac-Toe アプリケーションは、この情報を使用して、通信の対象となる DynamoDB リージョンを決定します。

3. Elastic Beanstalk 環境をセットアップします。Elastic Beanstalk により、Amazon EC2 インスタンス、もしくは他のインスタンスが起動され、それらの上で Tic-Tac-Toe アプリケーションバンドルがデプロイされます。Elastic Beanstalk 環境の準備が整ったら、CONFIG_FILE 環境変数を追加して設定ファイル名を指定します。
4. DynamoDB テーブルを作成します。Amazon DynamoDB サービスを使用して、ローカルコンピュータ上にではなく Games 上に AWS テーブルを作成します。このテーブルは、文字列型の GameId パーティションキーで構成されたシンプルなプライマリキーを持ちます。
5. 本稼働環境でゲームをテストします。

3.1: Amazon EC2 向けの IAM ロールを作成します

Amazon EC2 型の IAM ロールを作成すると、Tic-Tac-Toe アプリケーションを実行している Amazon EC2 インスタンスが適切なロールを引き受け、Games テーブルへのアクセスをアプリケーションにリクエストできるようになります。ロールを作成するときに、[Custom Policy (カスタムポリシー)] オプションを選択し、次のポリシーをコピーして貼り付けます。

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Action":[
            "dynamodb:ListTables"
         ],
         "Effect":"Allow",
         "Resource":"*"
      },
      {
         "Action":[
            "dynamodb:*"
         ],
         "Effect":"Allow",
         "Resource":[
            "arn:aws:dynamodb:us-west-2:922852403271:table/Games",
```

```
            "arn:aws:dynamodb:us-west-2:922852403271:table/Games/index/*"
         ]
      }
   ]
}
```

詳しい手順については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成 (AWS マネジメントコンソール)」を参照してください。

3.2: Amazon DynamoDB でゲームテーブルを作成します

ゲームのデータは、DynamoDB の Games テーブルに保存されます。テーブルが存在しない場合、アプリケーションによって自動的にテーブルが作成されます。この例では、アプリケーションで Games テーブルを作成します。

3.3: tic-tac-toe アプリケーションコードのバンドルとデプロイ

この例のステップを実行した場合、既に Tic-Tac-Toe アプリケーションをダウンロードしています。そうでない場合は、アプリケーションをダウンロードし、すべてのファイルをローカルコンピュータ上のフォルダに展開します。手順については、「ステップ 1: ローカルにデプロイおよびテストを実行します」を参照してください。

すべてのファイルを展開すると、code フォルダが作成されます。このフォルダを Elastic Beanstalk に引き渡すには、このフォルダのコンテンツを .zip ファイルとしてバンドルします。最初に、そのフォルダに設定ファイルを追加します。アプリケーションは、リージョンとエンドポイント情報を使用して、指定されたリージョンで DynamoDB テーブルを作成します。さらに、指定されたエンドポイントを使用しながら、それ以降のテーブルオペレーションをリクエストします。

1. Tic-Tac-Toe アプリケーションをダウンロードしたフォルダに切り替えます。
2. アプリケーションのルートフォルダで、次のコンテンツを使用して beanstalk.config という名前のテキストファイルを作成します。

   ```
   [dynamodb]
   region=<AWS region>
   endpoint=<DynamoDB endpoint>
   ```

   たとえば、次のコンテンツを使用します。

   ```
   [dynamodb]
   ```

```
region=us-west-2
endpoint=dynamodb.us-west-2.amazonaws.com
```

使用可能なリージョンのリストについては、Amazon Web Services 全般リファレンスの「Amazon DynamoDB」を参照してください

> **Important**
>
> 設定ファイルで指定されたリージョンは、Tic-Tac-Toe アプリケーションが DynamoDB の Games テーブルを作成する場所です。次のセクションで説明する Elastic Beanstalk アプリケーションを、同じリージョンで作成する必要があります。

> **Note**
>
> Elastic Beanstalk アプリケーションを作成する際には、環境タイプを選択しながら、その環境を起動するリクエストを行います。Tic-Tac-Toe サンプルアプリケーションをテストするには、[Single Instance (単一インスタンス)] 環境タイプを選択し、それ以降をスキップして、次のステップに進みます。
> ただし、[Load balancing, autoscaling (ロードバランシングでオートスケーリング)] 環境タイプでは、高可用性でスケーラブルな環境が提供されます。これは、他のアプリケーションを作成、デプロイする場合に検討してください。この環境タイプを選択する場合、UUID を生成し、次に示すように設定ファイルに追加する必要があります。
>
> ```
> [dynamodb]
> region=us-west-2
> endpoint=dynamodb.us-west-2.amazonaws.com
> [flask]
> secret_key= 284e784d-1a25-4a19-92bf-8eeb7a9example
> ```
>
> クライアント/サーバー通信でサーバーが応答を送信するときは、セキュリティのため、サーバーは、クライアントが次のリクエストでサーバーに送り返す署名済み Cookie を送信します。サーバーが 1 台のみの場合、サーバーは、起動時にローカルに暗号化キーを生成できます。多くのサーバーがある場合、それらのサーバーはすべて同じ暗号化キーを知る必要があります。そうしない場合、ピアサーバーによって設定された Cookie を読み取ることができません。secret_key を設定ファイルに追加することで、この暗号化キーを使用するようにすべてのサーバーに伝えます。

3. アプリケーションのルートフォルダのコンテンツを、(beanstalk.config ファイルも含めながら) Zip で圧縮します。たとえば TicTacToe.zip などとします。
4. Amazon Simple Storage Service (Amazon S3) バケットに、.zip ファイルをアップロードします。次のセクションで、この .zip ファイルを 1 つ以上のサーバーにアップロードするために Elastic Beanstalk に渡します。

   Amazon S3 バケットにアップロードする方法については、「Amazon Simple Storage Service ユーザーガイド」の「バケットの作成」および「バケットへのオブジェクトの追加」を参照してください。

3.4: AWS Elastic Beanstalk 環境の設定

このステップでは、環境など各コンポーネントの集合である Elastic Beanstalk アプリケーションを作成します。この例では、1 つの Amazon EC2 インスタンスを起動して、Tic-Tac-Toe アプリケーションをデプロイし実行します。

1. 環境をセットアップするには、次のカスタム URL を入力して Elastic Beanstalk コンソールをセットアップします。

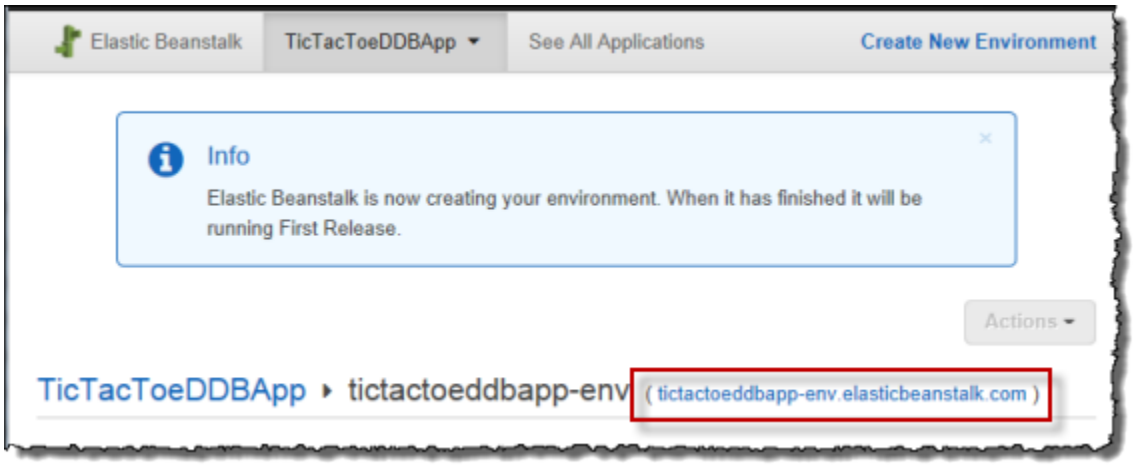
   ```
   https://console.aws.amazon.com/elasticbeanstalk/?region=<AWS-Region>#/
   newApplication
   ?applicationName=TicTacToeyour-name
   &solutionStackName=Python
   &sourceBundleUrl=https://s3.amazonaws.com/<bucket-name>/TicTacToe.zip
   &environmentType=SingleInstance
   &instanceType=t1.micro
   ```

   カスタム URL の詳細については、AWS Elastic Beanstalk デベロッパーガイドの「Launch Now URL の作成」を参照してください URL については、次の点に注意してください。

   - AWS リージョンの(設定ファイルで指定したものと同じ)名前、Amazon S3 バケット名、およびオブジェクト名を指定する必要があります。
   - テストでは、URL は SingleInstance 環境タイプ、および t1.micro をインスタンスタイプとしてリクエストします。
   - アプリケーション名は一意である必要があります。そのため、前の URL では、applicationName の前に名前を追加することをお勧めします。

これにより、Elastic Beanstalk コンソールが開きます。場合によっては、サインインが必要にあることがあります。

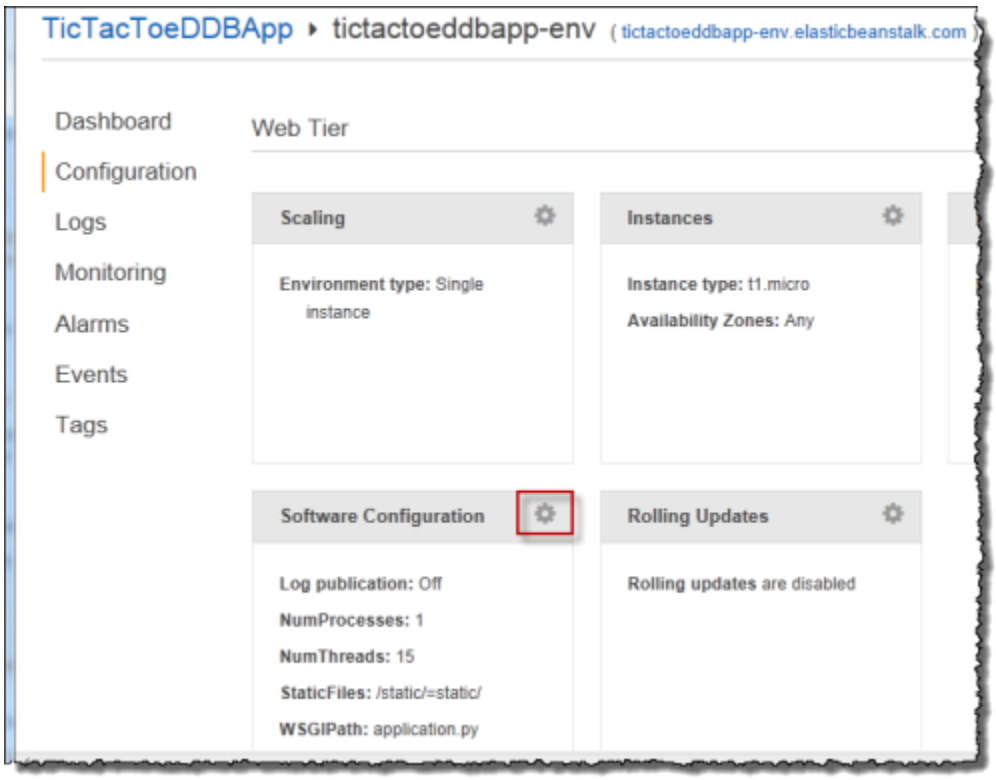
2. Elastic Beanstalk コンソールで、[Review and Launch] (確認して起動) を選択し、[Launch] (起動) を選択します。
3. 今後の参照用に URL を書き留めてください。この URL により、Tic-Tac-Toe アプリケーションのホームページが開きます。



4. Tic-Tac-Toe アプリケーションを設定し、設定ファイルの場所を指定します。

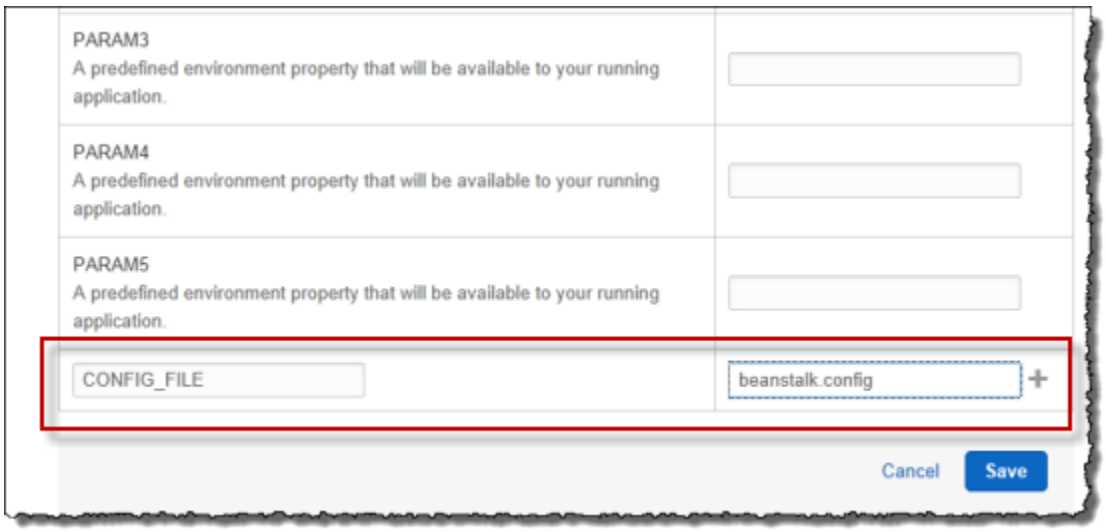
   Elastic Beanstalk によりアプリケーションが作成されたら、[Configuration] (設定) を選択します。

   a. 次のスクリーンショットに示すように、[Software Configuration (ソフトウェア設定)] の横にある歯車のアイコンを選択します。



b. [Environment Properties (環境プロパティ)] セクションの最後で、**CONFIG_FILE** とその値 **beanstalk.config** を入力し、[Save (保存)] を選択します。

この環境の更新が完了するには数分かかる場合があります。

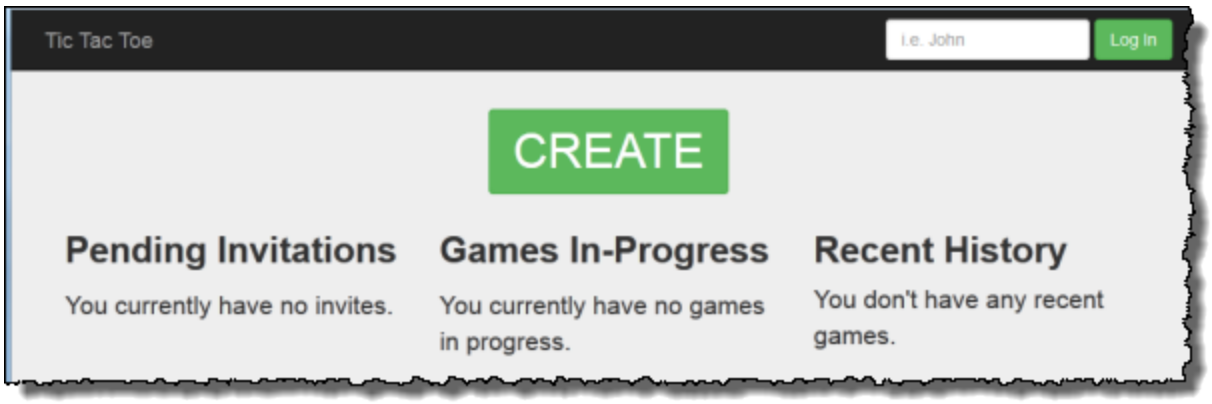


更新が完了したら、ゲームをプレイできます。

5. ブラウザで、前のステップでコピーした URL を、以下の例に示すように入力します。

```
http://<pen-name>.elasticbeanstalk.com
```

これにより、アプリケーションのホームページが開きます。



6. testuser1 としてログインし、[作成] を選択して新しい Tic-Tac-Toe ゲームを開始します。
7. [Choose an Opponent] (対戦相手を選択する) ボックスに **testuser2** を入力します。



8. 別のブラウザウィンドウを開きます。

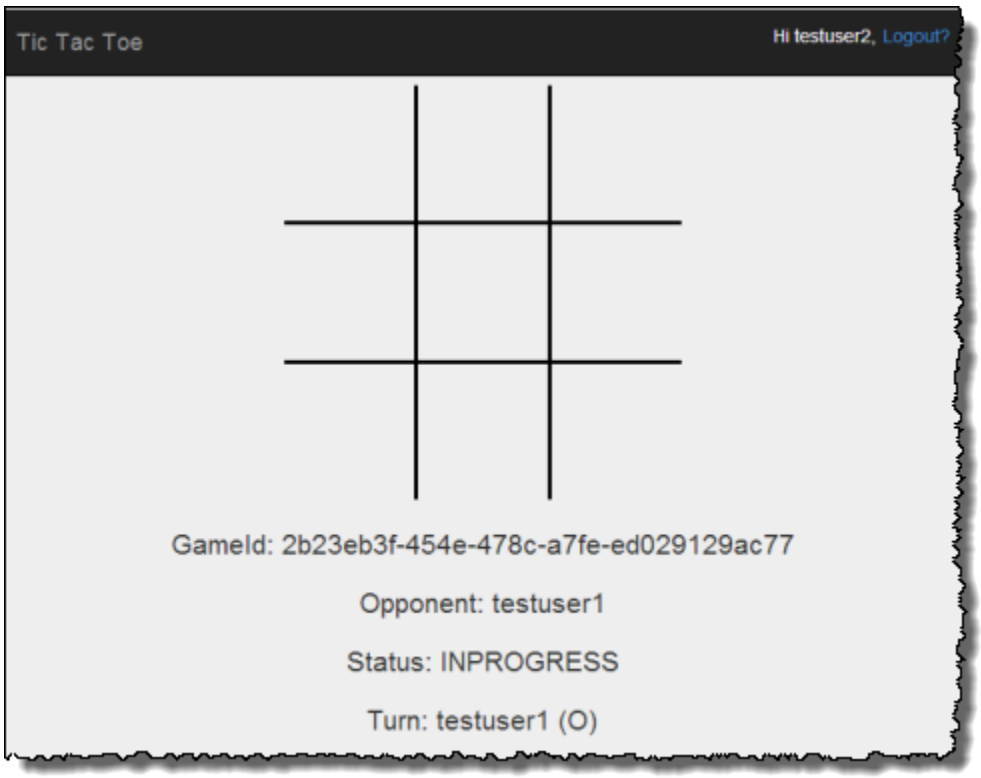
   ブラウザウィンドウのすべての Cookie を消去し、同じユーザーとしてログインしないようにします。
9. 以下の例に示すように、同じ URL を入力してアプリケーションのホームページを開きます。

```
http://<env-name>.elasticbeanstalk.com
```

10. testuser2 としてログインします。
11. 保留中の招待のリストで、testuser1 からの招待に対して [accept (受け入れる)] を選択します。



12. これで、ゲームページが表示されます。



testuser1 と testuser2 の両者がゲームをプレイできます。動きがあるごとに、アプリケーションは Games テーブルの対応する項目に動きを保存します。

ステップ 4: リソースをクリーンアップする

これで Tic-Tac-Toe アプリケーションのデプロイとテストを完了しました。このアプリケーションには、ユーザー認証を除き、Amazon DynamoDB でのウェブアプリケーション開発プロセスがすべて網羅されています。このアプリケーションは、ゲームの作成時にプレーヤーの名前を追加するためにのみ、ホームページのログイン情報を使用します。本稼働アプリケーションでは、ユーザーログインおよび認証を実行するために必要なコードを追加します。

テストを終了したら、料金が発生しないようにするため、Tic-Tac-Toe アプリケーションのテスト用に作成したリソースを削除できます。

作成したリソースを削除するには

1. DynamoDB で作成した Games テーブルを削除します。
2. Elastic Beanstalk 環境を終了し、Amazon EC2 インスタンスを解放します。
3. 作成した IAM ロールを削除します。
4. Amazon S3 で作成したオブジェクトを削除します。

DynamoDB の予約語

次のキーワードは DynamoDB によって予約されています。これらの単語を式の属性名として使用しないでください。このリストでは大文字と小文字が区別されません。

DynamoDB の予約語と競合する属性名を含む式を記述する必要が生じた場合は、予約語の代わりに使用する式属性名を定義できます。詳細については、「DynamoDB の式の属性名 (エイリアス)」を参照してください。

```
ABORT
ABSOLUTE
ACTION
ADD
AFTER
AGENT
AGGREGATE
ALL
ALLOCATE
ALTER
ANALYZE
```

```
AND
ANY
ARCHIVE
ARE
ARRAY
AS
ASC
ASCII
ASENSITIVE
ASSERTION
ASYMMETRIC
AT
ATOMIC
ATTACH
ATTRIBUTE
AUTH
AUTHORIZATION
AUTHORIZE
AUTO
AVG
BACK
BACKUP
BASE
BATCH
BEFORE
BEGIN
BETWEEN
BIGINT
BINARY
BIT
BLOB
BLOCK
BOOLEAN
BOTH
BREADTH
BUCKET
BULK
BY
BYTE
CALL
CALLED
CALLING
CAPACITY
CASCADE
```

```
CASCADED
CASE
CAST
CATALOG
CHAR
CHARACTER
CHECK
CLASS
CLOB
CLOSE
CLUSTER
CLUSTERED
CLUSTERING
CLUSTERS
COALESCE
COLLATE
COLLATION
COLLECTION
COLUMN
COLUMNS
COMBINE
COMMENT
COMMIT
COMPACT
COMPILE
COMPRESS
CONDITION
CONFLICT
CONNECT
CONNECTION
CONSISTENCY
CONSISTENT
CONSTRAINT
CONSTRAINTS
CONSTRUCTOR
CONSUMED
CONTINUE
CONVERT
COPY
CORRESPONDING
COUNT
COUNTER
CREATE
CROSS
```

```
CUBE
CURRENT
CURSOR
CYCLE
DATA
DATABASE
DATE
DATETIME
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DEFINE
DEFINED
DEFINITION
DELETE
DELIMITED
DEPTH
DEREF
DESC
DESCRIBE
DESCRIPTOR
DETACH
DETERMINISTIC
DIAGNOSTICS
DIRECTORIES
DISABLE
DISCONNECT
DISTINCT
DISTRIBUTE
DO
DOMAIN
DOUBLE
DROP
DUMP
DURATION
DYNAMIC
EACH
ELEMENT
ELSE
```

```
ELSEIF
EMPTY
ENABLE
END
EQUAL
EQUALS
ERROR
ESCAPE
ESCAPED
EVAL
EVALUATE
EXCEEDED
EXCEPT
EXCEPTION
EXCEPTIONS
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXIT
EXPLAIN
EXPLODE
EXPORT
EXPRESSION
EXTENDED
EXTERNAL
EXTRACT
FAIL
FALSE
FAMILY
FETCH
FIELDS
FILE
FILTER
FILTERING
FINAL
FINISH
FIRST
FIXED
FLATTERN
FLOAT
FOR
FORCE
FOREIGN
```

```
FORMAT
FORWARD
FOUND
FREE
FROM
FULL
FUNCTION
FUNCTIONS
GENERAL
GENERATE
GET
GLOB
GLOBAL
GO
GOTO
GRANT
GREATER
GROUP
GROUPING
HANDLER
HASH
HAVE
HAVING
HEAP
HIDDEN
HOLD
HOUR
IDENTIFIED
IDENTITY
IF
IGNORE
IMMEDIATE
IMPORT
IN
INCLUDING
INCLUSIVE
INCREMENT
INCREMENTAL
INDEX
INDEXED
INDEXES
INDICATOR
INFINITE
INITIALLY
```

```
INLINE
INNER
INNTER
INOUT
INPUT
INSENSITIVE
INSERT
INSTEAD
INT
INTEGER
INTERSECT
INTERVAL
INTO
INVALIDATE
IS
ISOLATION
ITEM
ITEMS
ITERATE
JOIN
KEY
KEYS
LAG
LANGUAGE
LARGE
LAST
LATERAL
LEAD
LEADING
LEAVE
LEFT
LENGTH
LESS
LEVEL
LIKE
LIMIT
LIMITED
LINES
LIST
LOAD
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOCATION
```

```
LOCATOR
LOCK
LOCKS
LOG
LOGED
LONG
LOOP
LOWER
MAP
MATCH
MATERIALIZED
MAX
MAXLEN
MEMBER
MERGE
METHOD
METRICS
MIN
MINUS
MINUTE
MISSING
MOD
MODE
MODIFIES
MODIFY
MODULE
MONTH
MULTI
MULTISET
NAME
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMBER
NUMERIC
```

```
OBJECT
OF
OFFLINE
OFFSET
OLD
ON
ONLINE
ONLY
OPAQUE
OPEN
OPERATOR
OPTION
OR
ORDER
ORDINALITY
OTHER
OTHERS
OUT
OUTER
OUTPUT
OVER
OVERLAPS
OVERRIDE
OWNER
PAD
PARALLEL
PARAMETER
PARAMETERS
PARTIAL
PARTITION
PARTITIONED
PARTITIONS
PATH
PERCENT
PERCENTILE
PERMISSION
PERMISSIONS
PIPE
PIPELINED
PLAN
POOL
POSITION
PRECISION
PREPARE
```

```
PRESERVE
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROCEDURE
PROCESSED
PROJECT
PROJECTION
PROPERTY
PROVISIONING
PUBLIC
PUT
QUERY
QUIT
QUORUM
RAISE
RANDOM
RANGE
RANK
RAW
READ
READS
REAL
REBUILD
RECORD
RECURSIVE
REDUCE
REF
REFERENCE
REFERENCES
REFERENCING
REGEXP
REGION
REINDEX
RELATIVE
RELEASE
REMAINDER
RENAME
REPEAT
REPLACE
REQUEST
RESET
RESIGNAL
```

```
RESOURCE
RESPONSE
RESTORE
RESTRICT
RESULT
RETURN
RETURNING
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLES
ROLLBACK
ROLLUP
ROUTINE
ROW
ROWS
RULE
RULES
SAMPLE
SATISFIES
SAVE
SAVEPOINT
SCAN
SCHEMA
SCOPE
SCROLL
SEARCH
SECOND
SECTION
SEGMENT
SEGMENTS
SELECT
SELF
SEMI
SENSITIVE
SEPARATE
SEQUENCE
SERIALIZABLE
SESSION
SET
SETS
SHARD
```

```
SHARE
SHARED
SHORT
SHOW
SIGNAL
SIMILAR
SIZE
SKEWED
SMALLINT
SNAPSHOT
SOME
SOURCE
SPACE
SPACES
SPARSE
SPECIFIC
SPECIFICTYPE
SPLIT
SQL
SQLCODE
SQLERROR
SQLEXCEPTION
SQLSTATE
SQLWARNING
START
STATE
STATIC
STATUS
STORAGE
STORE
STORED
STREAM
STRING
STRUCT
STYLE
SUB
SUBMULTISET
SUBPARTITION
SUBSTRING
SUBTYPE
SUM
SUPER
SYMMETRIC
SYNONYM
```

```
SYSTEM
TABLE
TABLESAMPLE
TEMP
TEMPORARY
TERMINATED
TEXT
THAN
THEN
THROUGHPUT
TIME
TIMESTAMP
TIMEZONE
TINYINT
TO
TOKEN
TOTAL
TOUCH
TRAILING
TRANSACTION
TRANSFORM
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
TRUNCATE
TTL
TUPLE
TYPE
UNDER
UNDO
UNION
UNIQUE
UNIT
UNKNOWN
UNLOGGED
UNNEST
UNPROCESSED
UNSIGNED
UNTIL
UPDATE
UPPER
```

```
URL
USAGE
USE
USER
USERS
USING
UUID
VACUUM
VALUE
VALUED
VALUES
VARCHAR
VARIABLE
VARIANCE
VARINT
VARYING
VIEW
VIEWS
VIRTUAL
VOID
WAIT
WHEN
WHENEVER
WHERE
WHILE
WINDOW
WITH
WITHIN
WITHOUT
WORK
WRAPPED
WRITE
YEAR
ZONE
```

AWS SDK for Java 1.x の例

このセクションでは、SDK for Java 1.x を使用した DAX アプリケーションのコード例について説明します。

トピック

- DAX を AWS SDK for Java 1.x で使用する

- 既存の SDK for Java 1.x アプリケーションを DAX を使用するように変更する
- SDK for Java 1.x を使用したグローバルセカンダリインデックスのクエリ

DAX を AWS SDK for Java 1.x で使用する

この手順に従って、Amazon EC2 インスタンスで Amazon DynamoDB Accelerator (DAX) の Java サンプルを実行します。

> **Note**
>
> これらの手順は、AWS SDK for Java 1.x を使用するアプリケーション向けです。AWS SDK for Java 2.x を使用するアプリケーションの場合は、「Java および DAX」を参照してください。

DAX の Java サンプルを実行するには

1. Java 開発キット (JDK) をインストールします。

   ```
   sudo yum install -y java-devel
   ```

2. AWS SDK for Java (`.zip` ファイル) をダウンロードして解凍します。

   ```
   wget http://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip

   unzip aws-java-sdk.zip
   ```

3. DAX Java クライアント (`.jar` ファイル) の最新バージョンをダウンロードします。

   ```
   wget http://dax-sdk.s3-website-us-west-2.amazonaws.com/java/DaxJavaClient-
   latest.jar
   ```

   > **Note**
   >
   > DAX SDK for Java のクライアントは、Apache Maven で利用可能です。詳細については、「クライアントを Apache Maven 依存関係として使用する」を参照してください。

4. CLASSPATH 変数を設定します。この例では、*`sdkVersion`* を AWS SDK for Java の実際のバージョン番号に置き換えます (例: `1.11.112`)。

```
export SDKVERSION=sdkVersion

export CLASSPATH=$(pwd)/TryDax/java:$(pwd)/DaxJavaClient-latest.jar:$(pwd)/
aws-java-sdk-$SDKVERSION/lib/aws-java-sdk-$SDKVERSION.jar:$(pwd)/aws-java-sdk-
$SDKVERSION/third-party/lib/*
```

5. サンプルプログラムソースコード (.zip ファイル) をダウンロードします。

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

ダウンロードが完了したら、ソースファイルを解凍します。

```
unzip TryDax.zip
```

6. Java コードディレクトリに移動し、次のようにコードをコンパイルします。

```
cd TryDax/java/
javac TryDax*.java
```

7. プログラムを実行します。

```
java TryDax
```

次のような出力が表示されます。

```
Creating a DynamoDB client

Attempting to create table; please wait...
Successfully created table.  Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
Writing 10 items for partition key: 4
Writing 10 items for partition key: 5
Writing 10 items for partition key: 6
Writing 10 items for partition key: 7
Writing 10 items for partition key: 8
Writing 10 items for partition key: 9
```

```
Writing 10 items for partition key: 10

Running GetItem, Scan, and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits

GetItem test - partition key 1 and sort keys 1-10
 Total time: 136.681 ms - Avg time: 13.668 ms
 Total time: 122.632 ms - Avg time: 12.263 ms
 Total time: 167.762 ms - Avg time: 16.776 ms
 Total time: 108.130 ms - Avg time: 10.813 ms
 Total time: 137.890 ms - Avg time: 13.789 ms
Query test - partition key 5 and sort keys between 2 and 9
 Total time: 13.560 ms - Avg time: 2.712 ms
 Total time: 11.339 ms - Avg time: 2.268 ms
 Total time: 7.809 ms - Avg time: 1.562 ms
 Total time: 10.736 ms - Avg time: 2.147 ms
 Total time: 12.122 ms - Avg time: 2.424 ms
Scan test - all items in the table
 Total time: 58.952 ms - Avg time: 11.790 ms
 Total time: 25.507 ms - Avg time: 5.101 ms
 Total time: 37.660 ms - Avg time: 7.532 ms
 Total time: 26.781 ms - Avg time: 5.356 ms
 Total time: 46.076 ms - Avg time: 9.215 ms

Attempting to delete table; please wait...
Successfully deleted table.
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

8. 前のステップで、DynamoDB エンドポイントに対してプログラムを実行しました。ここでプログラムを再度実行しますが、今度は GetItem、Query、Scan オペレーションが DAX クラスターによって処理されます。

   DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

   - DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

     ```
     dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
     ```

   - AWS CLI の使用 — 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

次の例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

ここでプログラムを再度実行しますが、今度はクラスターエンドポイントをコマンドラインパラメータとして指定します。

```
java TryDax dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

出力の残りの部分を見て、タイミング情報を書き留めます。`GetItem`、`Query` および `Scan` の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

このプログラムの詳細については、以下のセクションを参照してください。

- TryDax.java
- TryDaxHelper.java
- TryDaxTests.java

クライアントを Apache Maven 依存関係として使用する

これらのステップに従って、アプリケーションで DAX SDK for Java のクライアントを依存関係として使用します。

クライアントを Maven 依存関係として使用するには

1. Apache Maven をダウンロードし、インストールします。詳細については、「Apache Maven のダウンロード」および「Apache Maven のインストール」を参照してください。
2. Maven 依存関係クライアントをアプリケーションのプロジェクトオブジェクトモデル (POM) ファイルに追加します。この例では、`x.x.x.x` をクライアントの実際のバージョン番号に置き換えます (例: `1.0.200704.0`)。

```
<!--Dependency:-->
<dependencies>
    <dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>amazon-dax-client</artifactId>
     <version>x.x.x.x</version>
    </dependency>
</dependencies>
```

TryDax.java

`TryDax.java` ファイルには、`main` メソッドが含まれています。コマンドラインパラメータを指定しないでプログラムを実行すると、Amazon DynamoDB クライアントが作成され、すべての API オペレーションでそのクライアントが使用されます。コマンドラインで DynamoDB Accelerator (DAX) クラスターエンドポイントを指定すると、プログラムは DAX クライアントも作成し、`GetItem`、`Query`、`Scan` オペレーションでこれを使用します。

さまざまな方法でプログラムを変更できます。

- DynamoDB クライアントではなく、DAX クライアントを使用します。詳細については、「Java および DAX」を参照してください。
- テストテーブルに別の名前を付けます。
- `helper.writeData` パラメータを変更して書き込まれる項目の数を変更する。2 番目のパラメータはパーティションキーの数、3 番目のパラメータはソートキーの数です。デフォルトでは、プログラムはパーティションキーの値に 1 ~ 10 を使用し、ソートキーの値に 1 ~ 10 を使用し、合計で 100 個の項目がテーブルに書き込まれます。詳細については、「TryDaxHelper.java」を参照してください。
- `GetItem`、`Query` および `Scan` テストの数を変更し、そのパラメータを変更する。
- `helper.createTable` および `helper.deleteTable` を含むラインをコメントアウトする (プログラムを実行するたびにテーブルを作成および削除しない場合)。

> **Note**
>
> このプログラムを実行するには、DAX SDK for Java のクライアント、および AWS SDK for Java を依存関係として使用するように Maven を設定できます。詳細については、「クライアントを Apache Maven 依存関係として使用する」を参照してください。

また、DAX Java クライアントと AWS SDK for Java の両方をダウンロードして、クラスパスに含めることもできます。CLASSPATH 変数の設定例については、「Java および DAX」を参照してください。

```
public class TryDax {

    public static void main(String[] args) throws Exception {

        TryDaxHelper helper = new TryDaxHelper();
        TryDaxTests tests = new TryDaxTests();

        DynamoDB ddbClient = helper.getDynamoDBClient();
        DynamoDB daxClient = null;
        if (args.length >= 1) {
            daxClient = helper.getDaxClient(args[0]);
        }

        String tableName = "TryDaxTable";

        System.out.println("Creating table...");
        helper.createTable(tableName, ddbClient);
        System.out.println("Populating table...");
        helper.writeData(tableName, ddbClient, 10, 10);

        DynamoDB testClient = null;
        if (daxClient != null) {
            testClient = daxClient;
        } else {
            testClient = ddbClient;
        }

        System.out.println("Running GetItem, Scan, and Query tests...");
        System.out.println("First iteration of each test will result in cache misses");
        System.out.println("Next iterations are cache hits\n");

        // GetItem
        tests.getItemTest(tableName, testClient, 1, 10, 5);

        // Query
        tests.queryTest(tableName, testClient, 5, 2, 9, 5);
```

```
        // Scan
        tests.scanTest(tableName, testClient, 5);

        helper.deleteTable(tableName, ddbClient);
    }

}
```

TryDaxHelper.java

TryDaxHelper.java ファイルにはユーティリティメソッドが含まれています。

getDynamoDBClient および getDaxClient メソッドは、Amazon DynamoDB および Amazon DynamoDB Accelerator (DAX) クライアントを提供します。コントロールプレーンオペレーション (CreateTable、DeleteTable) および書き込みオペレーションでは、プログラムは DynamoDB クライアントを使用します。DAX クラスターエンドポイントを指定すると、メインプログラムは読み込みオペレーション (GetItem、Query、Scan) 用に DAX クライアントを作成します。

他の TryDaxHelper メソッド (createTable、writeData、deleteTable) は、DynamoDB テーブルとそのデータのセットアップおよび解体用です。

さまざまな方法でプログラムを変更できます。

- テーブルで別のプロビジョニングされたスループット設定を使用する。
- 書き込まれた各項目のサイズを変更する (writeData メソッドの stringSize 変数を参照してください)。
- GetItem、Query、および Scan テストの数とそのパラメータを変更する。
- helper.CreateTable および helper.DeleteTable を含むラインをコメントアウトする (プログラムを実行するたびにテーブルを作成および削除しない場合)。

> **Note**
>
> このプログラムを実行するには、DAX SDK for Java のクライアント、および AWS SDK for Java を依存関係として使用するように Maven を設定できます。詳細については、「クライアントを Apache Maven 依存関係として使用する」を参照してください。

または、DAX Java クライアントと AWS SDK for Java の両方をダウンロードして、クラスパスに含めることもできます。CLASSPATH 変数の設定例については、「Java および DAX」を参照してください。

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.util.EC2MetadataUtils;

public class TryDaxHelper {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDynamoDBClient() {
        System.out.println("Creating a DynamoDB client");
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
                .withRegion(region)
                .build();
        return new DynamoDB(client);
    }

    DynamoDB getDaxClient(String daxEndpoint) {
        System.out.println("Creating a DAX client with cluster endpoint " +
 daxEndpoint);
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
        AmazonDynamoDB client = daxClientBuilder.build();
        return new DynamoDB(client);
    }

    void createTable(String tableName, DynamoDB client) {
        Table table = client.getTable(tableName);
```

```
        try {
            System.out.println("Attempting to create table; please wait...");

            table = client.createTable(tableName,
                    Arrays.asList(
                            new KeySchemaElement("pk", KeyType.HASH), // Partition key
                            new KeySchemaElement("sk", KeyType.RANGE)), // Sort key
                    Arrays.asList(
                            new AttributeDefinition("pk", ScalarAttributeType.N),
                            new AttributeDefinition("sk", ScalarAttributeType.N)),
                    new ProvisionedThroughput(10L, 10L));
            table.waitForActive();
            System.out.println("Successfully created table.  Table status: " +
                    table.getDescription().getTableStatus());

        } catch (Exception e) {
            System.err.println("Unable to create table: ");
            e.printStackTrace();
        }
    }

    void writeData(String tableName, DynamoDB client, int pkmax, int skmax) {
        Table table = client.getTable(tableName);
        System.out.println("Writing data to the table...");

        int stringSize = 1000;
        StringBuilder sb = new StringBuilder(stringSize);
        for (int i = 0; i < stringSize; i++) {
            sb.append('X');
        }
        String someData = sb.toString();

        try {
            for (Integer ipk = 1; ipk <= pkmax; ipk++) {
                System.out.println(("Writing " + skmax + " items for partition key: " +
ipk));
                for (Integer isk = 1; isk <= skmax; isk++) {
                    table.putItem(new Item()
                            .withPrimaryKey("pk", ipk, "sk", isk)
                            .withString("someData", someData));
                }
            }
        } catch (Exception e) {
            System.err.println("Unable to write item:");
```

```
            e.printStackTrace();
        }
    }

    void deleteTable(String tableName, DynamoDB client) {
        Table table = client.getTable(tableName);
        try {
            System.out.println("\nAttempting to delete table; please wait...");
            table.delete();
            table.waitForDelete();
            System.out.println("Successfully deleted table.");

        } catch (Exception e) {
            System.err.println("Unable to delete table: ");
            e.printStackTrace();
        }
    }

}
```

TryDaxTests.java

`TryDaxTests.java` ファイルには、Amazon DynamoDB のテストテーブルに対して読み込みオペレーションを実行するメソッドが含まれています。これらのメソッドはデータへのアクセス方法 (DynamoDB クライアントを使用するか DAX クライアントを使用するか) は問わないため、アプリケーションロジックを変更する必要はありません。

さまざまな方法でプログラムを変更できます。

- `queryTest` メソッドを変更して異なる `KeyConditionExpression` を使用する。
- `ScanFilter` を `scanTest` メソッドに追加し、項目の一部のみが返されるようにします。

> **Note**
>
> このプログラムを実行するには、DAX SDK for Java のクライアント、および AWS SDK for Java を依存関係として使用するように Maven を設定できます。詳細については、「クライアントを Apache Maven 依存関係として使用する」を参照してください。

または、DAX Java クライアントと AWS SDK for Java の両方をダウンロードして、クラスパスに含めることもできます。CLASSPATH 変数の設定例については、「Java および DAX」を参照してください。

```
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;

public class TryDaxTests {

    void getItemTest(String tableName, DynamoDB client, int pk, int sk, int iterations)
 {
        long startTime, endTime;
        System.out.println("GetItem test - partition key " + pk + " and sort keys 1-" +
 sk);
        Table table = client.getTable(tableName);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            try {
                for (Integer ipk = 1; ipk <= pk; ipk++) {
                    for (Integer isk = 1; isk <= sk; isk++) {
                        table.getItem("pk", ipk, "sk", isk);
                    }
                }
            } catch (Exception e) {
                System.err.println("Unable to get item:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, pk * sk);
        }
    }
```

```
    void queryTest(String tableName, DynamoDB client, int pk, int sk1, int sk2, int
iterations) {
        long startTime, endTime;
        System.out.println("Query test - partition key " + pk + " and sort keys between
" + sk1 + " and " + sk2);
        Table table = client.getTable(tableName);

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":pkval", pk);
        valueMap.put(":skval1", sk1);
        valueMap.put(":skval2", sk2);

        QuerySpec spec = new QuerySpec()
                .withKeyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
                .withValueMap(valueMap);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            ItemCollection<QueryOutcome> items = table.query(spec);

            try {
                Iterator<Item> iter = items.iterator();
                while (iter.hasNext()) {
                    iter.next();
                }
            } catch (Exception e) {
                System.err.println("Unable to query table:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, iterations);
        }
    }

    void scanTest(String tableName, DynamoDB client, int iterations) {
        long startTime, endTime;
        System.out.println("Scan test - all items in the table");
        Table table = client.getTable(tableName);

        for (int i = 0; i < iterations; i++) {
            startTime = System.nanoTime();
            ItemCollection<ScanOutcome> items = table.scan();
            try {
```

```
                Iterator<Item> iter = items.iterator();
                while (iter.hasNext()) {
                    iter.next();
                }
            } catch (Exception e) {
                System.err.println("Unable to scan table:");
                e.printStackTrace();
            }
            endTime = System.nanoTime();
            printTime(startTime, endTime, iterations);
        }
    }

    public void printTime(long startTime, long endTime, int iterations) {
        System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) /
 (1000000.0));
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *
 1000000.0));
    }
}
```

既存の SDK for Java 1.x アプリケーションを DAX を使用するように変更する

Amazon DynamoDB を使用している Java アプリケーションがすでにある場合、DynamoDB Accelerator (DAX) クラスターにアクセスできるように変更する必要があります。DAX Java クライアントは、AWS SDK for Java に含まれる DynamoDB 低レベルクライアントとよく似ているため、アプリケーション全体を書き換える必要はありません。

> **Note**
>
> これらの手順は、AWS SDK for Java 1.x を使用するアプリケーション向けです。AWS SDK for Java 2.x を使用するアプリケーションの場合は、「既存のアプリケーションを DAX を使用するように変更する」を参照してください。

`Music` という名前の DynamoDB テーブルがあるとします。テーブルのパーティションキーは `Artist` で、ソートキーは `SongTitle` です。次のプログラムは、`Music` テーブルから直接項目を読み込みます。

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {

    public static void main(String[] args) throws Exception {

        // Create a DynamoDB client
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        try {
            System.out.println("Attempting to read the item...");
            GetItemResult result = client.getItem(request);
            System.out.println("GetItem succeeded: " + result);

        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }
    }
}
```

プログラムを変更するには、DynamoDB クライアントを DAX クライアントに置き換えます。

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;
```

```
public class GetMusicItem {

    public static void main(String[] args) throws Exception {

    //Create a DAX client

    AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
    daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com
    AmazonDynamoDB client = daxClientBuilder.build();


       /*
       ** ...
       ** Remaining code omitted (it is identical)
       ** ...
       */

    }
}
```

DynamoDB ドキュメント API を使用する

AWS SDK for Java は DynamoDB 向けのドキュメントインターフェイスを提供します。ドキュメント API は、低レベル DynamoDB クライアントのラッパーとして機能します。詳細については、「ドキュメントインターフェイス」を参照してください。

ドキュメントインターフェイスは、次の例に示すように、低レベル DAX クライアントとともに使用することもできます。

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class GetMusicItemWithDocumentApi {

    public static void main(String[] args) throws Exception {

        //Create a DAX client
```

```
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com
        AmazonDynamoDB client = daxClientBuilder.build();

        // Document client wrapper
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");

        try {
            System.out.println("Attempting to read the item...");
            GetItemOutcome outcome = table.tgetItemOutcome(
                "Artist", "No One You Know",
                "SongTitle", "Scared of My Shadow");
            System.out.println(outcome.getItem());
            System.out.println("GetItem succeeded: " + outcome);
        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }

    }
}
```

DAX 非同期クライアント

AmazonDaxClient は同期します。大きいテーブルの Scan のように長期間実行する DAX API オペレーションでは、そのオペレーションが完了するまでプログラムの実行がブロックされることがあります。DAX API オペレーションの処理中にプログラムで他の作業を実行する必要がある場合は、代わりに ClusterDaxAsyncClient を使用できます。

次のプログラムは、ClusterDaxAsyncClient を Java Future と併せて使用し、ノンブロッキングソリューションを実装する方法を示します。

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import com.amazon.dax.client.dynamodbv2.ClientConfig;
import com.amazon.dax.client.dynamodbv2.ClusterDaxAsyncClient;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.handlers.AsyncHandler;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBAsync;
```

```
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class DaxAsyncClientDemo {
 public static void main(String[] args) throws Exception {

  ClientConfig daxConfig = new ClientConfig().withCredentialsProvider(new
 ProfileCredentialsProvider())
    .withEndpoints("mydaxcluster.2cmrwl.clustercfg.dax.use1.cache.amazonaws.com:8111");

  AmazonDynamoDBAsync client = new ClusterDaxAsyncClient(daxConfig);

  HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
  key.put("Artist", new AttributeValue().withS("No One You Know"));
  key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

  GetItemRequest request = new GetItemRequest()
    .withTableName("Music").withKey(key);

  // Java Futures
  Future<GetItemResult> call = client.getItemAsync(request);
  while (!call.isDone()) {
   // Do other processing while you're waiting for the response
   System.out.println("Doing something else for a few seconds...");
   Thread.sleep(3000);
  }
  // The results should be ready by now

  try {
   call.get();

  } catch (ExecutionException ee) {
   // Futures always wrap errors as an ExecutionException.
   // The *real* exception is stored as the cause of the
   // ExecutionException
   Throwable exception = ee.getCause();
   System.out.println("Error getting item: " + exception.getMessage());
  }

  // Async callbacks
  call = client.getItemAsync(request, new AsyncHandler<GetItemRequest, GetItemResult>()
 {
```

```
    @Override
    public void onSuccess(GetItemRequest request, GetItemResult getItemResult) {
     System.out.println("Result: " + getItemResult);
    }

    @Override
    public void onError(Exception e) {
     System.out.println("Unable to read item");
     System.err.println(e.getMessage());
     // Callers can also test if exception is an instance of
     // AmazonServiceException or AmazonClientException and cast
     // it to get additional information
    }

   });
   call.get();

  }
}
```

SDK for Java 1.x を使用したグローバルセカンダリインデックスのクエリ

Amazon DynamoDB Accelerator (DAX) を使用して、DynamoDB プログラムインターフェイスを使用してグローバルセカンダリインデックスをクエリできます。

次の例は、DAX を使用して、「例: AWS SDK for Java ドキュメント API を使用したグローバルセカンダリインデックス」で作成した `CreateDateIndex` グローバルセカンダリインデックスをクエリする方法を示します。

`DAXClient` クラスは、DynamoDB プログラミングインターフェイスを操作するために必要なクライアントオブジェクトをインスタンス化します。

```
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import software.amazon.dynamodb.datamodeling.DynamoDBMapper;
import software.amazon.dynamodb.document.DynamoDB;
import com.amazonaws.util.EC2MetadataUtils;
import software.amazon.dynamodb.AmazonDynamoDB;

public class DaxClient {

 private static final String region = EC2MetadataUtils.getEC2InstanceRegion();
```

```
 DynamoDB getDaxDocClient(String daxEndpoint) {
  System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
  AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

  daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
  AmazonDynamoDB client = daxClientBuilder.build();

  return new DynamoDB(client);
 }

 DynamoDBMapper getDaxMapperClient(String daxEndpoint) {
  System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
  AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

  daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
  AmazonDynamoDB client = daxClientBuilder.build();

  return new DynamoDBMapper(client);
 }
}
```

グローバルセカンダリインデックスは、以下の方法でクエリできます。

- 以下の例で定義された queryIndex クラスの QueryIndexDax メソッドを使用します。QueryIndexDax は、getDaxDocClient クラスの DaxClient メソッドによって返されるクライアントオブジェクトをパラメータとして取ります。
- オブジェクト永続性インターフェイスを使用する場合は、以下の例で定義された queryIndexMapper クラスで QueryIndexDax メソッドを使用します。queryIndexMapper は、getDaxMapperClient クラスで定義された DaxClient メソッドによって返されるクライアントオブジェクトをパラメータとして取ります。

```
import java.util.Iterator;
import software.amazon.dynamodb.datamodeling.DynamoDBMapper;
import java.util.List;
import software.amazon.dynamodb.datamodeling.DynamoDBQueryExpression;
import software.amazon.dynamodb.model.AttributeValue;
import java.util.HashMap;
import software.amazon.dynamodb.document.Item;
import software.amazon.dynamodb.document.utils.ValueMap;
import software.amazon.dynamodb.document.spec.QuerySpec;
import software.amazon.dynamodb.document.QueryOutcome;
```

```
import software.amazon.dynamodb.document.ItemCollection;
import software.amazon.dynamodb.document.Index;
import software.amazon.dynamodb.document.Table;
import software.amazon.dynamodb.document.DynamoDB;

public class QueryIndexDax {

 //This is used to query Index using the low-level interface.
 public static void queryIndex(DynamoDB client, String tableName, String indexName) {
  Table table = client.getTable(tableName);

  System.out.println("\n***********************************************************
\n");
  System.out.print("Querying index " + indexName + "...");

  Index index = table.getIndex(indexName);

  ItemCollection<QueryOutcome> items = null;

  QuerySpec querySpec = new QuerySpec();

  if (indexName == "CreateDateIndex") {
   System.out.println("Issues filed on 2013-11-01");
   querySpec.withKeyConditionExpression("CreateDate = :v_date and
 begins_with(IssueId, :v_issue)")
     .withValueMap(new ValueMap().withString(":v_date",
 "2013-11-01").withString(":v_issue", "A-"));
   items = index.query(querySpec);
  } else {
   System.out.println("\nNo valid index name provided");
   return;
  }

  Iterator<Item> iterator = items.iterator();

  System.out.println("Query: printing results...");

  while (iterator.hasNext()) {
   System.out.println(iterator.next().toJSONPretty());
  }

 }

 //This is used to query Index using the high-level mapper interface.
```

```
 public static void queryIndexMapper(DynamoDBMapper mapper, String tableName, String
 indexName) {
  HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
  eav.put(":v_date", new AttributeValue().withS("2013-11-01"));
  eav.put(":v_issue", new AttributeValue().withS("A-"));
  DynamoDBQueryExpression<CreateDate> queryExpression = new
 DynamoDBQueryExpression<CreateDate>()
     .withIndexName("CreateDateIndex").withConsistentRead(false)
     .withKeyConditionExpression("CreateDate = :v_date and
 begins_with(IssueId, :v_issue)")
     .withExpressionAttributeValues(eav);

  List<CreateDate> items = mapper.query(CreateDate.class, queryExpression);
  Iterator<CreateDate> iterator = items.iterator();

  System.out.println("Query: printing results...");

  while (iterator.hasNext()) {
   CreateDate iterObj = iterator.next();
   System.out.println(iterObj.getCreateDate());
   System.out.println(iterObj.getIssueId());
  }
 }
}
```

以下の定義は、「Issues」テーブルを表しており、queryIndexMapper メソッドで使用されます。

```
import software.amazon.dynamodb.datamodeling.DynamoDBTable;
import software.amazon.dynamodb.datamodeling.DynamoDBIndexHashKey;
import software.amazon.dynamodb.datamodeling.DynamoDBIndexRangeKey;
import software.amazon.dynamodb.datamodeling.DynamoDBHashKey;

@DynamoDBTable(tableName = "Issues")
public class CreateDate {
 private String createDate;
 @DynamoDBHashKey(attributeName = "IssueId")
 private String issueId;

 @DynamoDBIndexHashKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
 "CreateDate")
 public String getCreateDate() {
  return createDate;
 }
```

```
 public void setCreateDate(String createDate) {
  this.createDate = createDate;
 }

 @DynamoDBIndexRangeKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
 "IssueId")
 public String getIssueId() {
  return issueId;
 }

 public void setIssueId(String issueId) {
  this.issueId = issueId;
 }
}
```

AWS SDK for Go 1.x の例

このセクションでは、Go 1.x を使用した DAX アプリケーションのコード例について説明します。

トピック

- DAX SDK for Go

DAX SDK for Go

この手順に従って、Amazon EC2 インスタンスで Go 対応 Amazon DynamoDB Accelerator (DAX) SDK for Go サンプルアプリケーションを実行します。

DAX の SDK for Go サンプルを実行するには

1. Amazon EC2 インスタンスで SDK for Go をセットアップします。

   a. Go プログラミング言語をインストール (Golang) します。

   ```
   sudo yum install -y golang
   ```

   b. Golang が正しくインストールおよび実行されているかテストします。

   ```
   go version
   ```

次のようなメッセージが表示されます。

```
go version go1.15.5 linux/amd64
```

残りの手順は、Go バージョン 1.13 のデフォルトとなったモジュールのサポートに基づいています。

2. サンプル Golang アプリケーションをインストールします。

```
go get github.com/aws-samples/aws-dax-go-sample
```

3. 以下の Golang プログラムを実行します。最初のプログラムは、TryDaxGoTable という DynamoDB テーブルを作成します。2 番目のプログラムは、テーブルにデータを書き込みます。

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command create-table
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command put-item
```

4. 以下の Golang プログラムを実行します。

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command get-item
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command query
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command scan
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

5. 前のステップで、DynamoDB エンドポイントに対してプログラムを実行しました。ここでプログラムを再度実行しますが、今度は GetItem、Query、Scan オペレーションが DAX クラスターによって処理されます。

DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- DynamoDB コンソールの使用 — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- AWS CLI の使用 — 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

次の例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

ここでプログラムを再度実行しますが、今度はクラスターエンドポイントをコマンドラインパラメータとして指定します。

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
 -service dax -command get-item -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
 -service dax -command query -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go
 -service dax -command scan -endpoint my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com:8111
```

出力の残りの部分を見て、タイミング情報を書き留めます。`GetItem`、`Query` および `Scan` の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

6. 次の Golang プログラムを実行して `TryDaxGoTable` を削除します。

```
go run ~/go/pkg/mod/github.com/aws-samples/aws-dax-go-sample@v1.0.2/try_dax.go -
service dynamodb -command delete-table
```

AWS SDK for Node.js 2.x の例

このセクションでは、AWS SDK for Node.js 2.x を使用した DAX アプリケーションのコード例について説明します。

トピック

- Node.js および DAX

Node.js および DAX

このステップに従って、Amazon EC2 インスタンスで Node.js サンプルアプリケーションを実行します。

DAX の Node.js サンプルを実行するには

1. 以下のように、Amazon EC2 インスタンスで Node.js を設定します。
   a. ノードバージョンマネージャー (nvm) をインストールします。

      ```
      curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
      ```

   b. nvm を使用して Node.js をインストールします。

      ```
      nvm install 12.16.3
      ```

   c. Node.js が正しくインストールおよび実行されているかテストします。

      ```
      node -e "console.log('Running Node.js ' + process.version)"
      ```

      これにより次のメッセージが表示されるはずです。

      ```
      Running Node.js v12.16.3
      ```

2. ノードパッケージマネージャー (npm) を使用して、DAX Node.js クライアントをインストールします。

```
npm install amazon-dax-client
```

3. サンプルプログラムソースコード (.zip ファイル) をダウンロードします。

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/
TryDax.zip
```

ダウンロードが完了したら、ソースファイルを解凍します。

```
unzip TryDax.zip
```

4. 次の Node.js プログラムを実行します。最初のプログラムは、TryDaxTable という Amazon DynamoDB テーブルを作成します。2 番目のプログラムは、テーブルにデータを書き込みます。

```
node 01-create-table.js
node 02-write-data.js
```

5. 次の Node.js プログラムを実行します。

```
node 03-getitem-test.js
node 04-query-test.js
node 05-scan-test.js
```

タイミング情報を書き留めます。これは GetItem、Query、Scan テストに必要なミリ秒の数字です。

6. 前のステップで、DynamoDB エンドポイントに対してプログラムを実行しました。プログラムを再度実行しますが、今度は GetItem、Query、Scan オペレーションが DAX クラスターによって処理されます。

DAX クラスターのエンドポイントを確認するには、次のいずれかを選択します。

- [Using the DynamoDB console] (DynamoDB コンソールの使用) — DAX クラスターを選択します。次の例のように、クラスターエンドポイントがコンソールに表示されます。

```
dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

- AWS CLI の使用 - 次のコマンドを入力します。

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

次の例のように、クラスターエンドポイントが出力に表示されます。

```
{
    "Address": "my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com",
    "Port": 8111,
    "URL": "dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com"
}
```

ここでプログラムを再度実行しますが、今度はクラスターエンドポイントをコマンドラインパラメータとして指定します。

```
node 03-getitem-test.js dax://my-cluster.l6fzcv.dax-clusters.us-
east-1.amazonaws.com
node 04-query-test.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
node 05-scan-test.js dax://my-cluster.l6fzcv.dax-clusters.us-east-1.amazonaws.com
```

出力の残りの部分を見て、タイミング情報を書き留めます。GetItem、Query および Scan の経過時間は、DynamoDB を使用した場合より DAX を使用した方が大幅に低いはずです。

7. 次の Node.js プログラムを実行して TryDaxTable を削除します。

```
node 06-delete-table
```

これらのプログラムの詳細については、以下のセクションを参照してください。

- 01-create-table.js
- 02-write-data.js
- 03-getitem-test.js
- 04-query-test.js
- 05-scan-test.js
- 06-delete-table.js

01-create-table.js

`01-create-table.js` プログラムはテーブル (`TryDaxTable`) を作成します。このセクションの残りの Node.js プログラムは、このテーブルによって異なります。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
  KeySchema: [
    { AttributeName: "pk", KeyType: "HASH" }, //Partition key
    { AttributeName: "sk", KeyType: "RANGE" }, //Sort key
  ],
  AttributeDefinitions: [
    { AttributeName: "pk", AttributeType: "N" },
    { AttributeName: "sk", AttributeType: "N" },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 10,
    WriteCapacityUnits: 10,
  },
};

dynamodb.createTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to create table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
    console.log(
      "Created table. Table description JSON:",
```

```
      JSON.stringify(data, null, 2)
    );
  }
});
```

02-write-data.js

02-write-data.js プログラムはテストデータを TryDaxTable に書き込みます。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();

var tableName = "TryDaxTable";

var someData = "X".repeat(1000);
var pkmax = 10;
var skmax = 10;

for (var ipk = 1; ipk <= pkmax; ipk++) {
  for (var isk = 1; isk <= skmax; isk++) {
    var params = {
      TableName: tableName,
      Item: {
        pk: ipk,
        sk: isk,
        someData: someData,
      },
    };

    //
    //put item

    ddbClient.put(params, function (err, data) {
      if (err) {
```

```
        console.error("Unable to write data: ", JSON.stringify(err, null, 2));
      } else {
        console.log("PutItem succeeded");
      }
    });
  }
}
```

03-getitem-test.js

`03-getitem-test.js` プログラムは、`GetItem` で `TryDaxTable` オペレーションを実行します。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 1;
var sk = 10;
var iterations = 5;

for (var i = 0; i < iterations; i++) {
  var startTime = new Date().getTime();
```

```
  for (var ipk = 1; ipk <= pk; ipk++) {
    for (var isk = 1; isk <= sk; isk++) {
      var params = {
        TableName: tableName,
        Key: {
          pk: ipk,
          sk: isk,
        },
      };

      client.get(params, function (err, data) {
        if (err) {
          console.error(
            "Unable to read item. Error JSON:",
            JSON.stringify(err, null, 2)
          );
        } else {
          // GetItem succeeded
        }
      });
    }
  }

  var endTime = new Date().getTime();
  console.log(
    "\tTotal time: ",
    endTime - startTime,
    "ms - Avg time: ",
    (endTime - startTime) / iterations,
    "ms"
  );
}
```

04-query-test.js

`04-query-test.js` プログラムは、Query で TryDaxTable オペレーションを実行します。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";
```

```
AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 5;
var sk1 = 2;
var sk2 = 9;
var iterations = 5;

var params = {
  TableName: tableName,
  KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",
  ExpressionAttributeValues: {
    ":pkval": pk,
    ":skval1": sk1,
    ":skval2": sk2,
  },
};

for (var i = 0; i < iterations; i++) {
  var startTime = new Date().getTime();

  client.query(params, function (err, data) {
    if (err) {
      console.error(
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
      );
    } else {
      // Query succeeded
```

```
    }
  });

  var endTime = new Date().getTime();
  console.log(
    "\tTotal time: ",
    endTime - startTime,
    "ms - Avg time: ",
    (endTime - startTime) / iterations,
    "ms"
  );
}
```

05-scan-test.js

`05-scan-test.js` プログラムは、Scan で `TryDaxTable` オペレーションを実行します。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var ddbClient = new AWS.DynamoDB.DocumentClient();
var daxClient = null;

if (process.argv.length > 2) {
  var dax = new AmazonDaxClient({
    endpoints: [process.argv[2]],
    region: region,
  });
  daxClient = new AWS.DynamoDB.DocumentClient({ service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var iterations = 5;
```

```
var params = {
  TableName: tableName,
};
var startTime = new Date().getTime();
for (var i = 0; i < iterations; i++) {
  client.scan(params, function (err, data) {
    if (err) {
      console.error(
        "Unable to read item. Error JSON:",
        JSON.stringify(err, null, 2)
      );
    } else {
      // Scan succeeded
    }
  });
}

var endTime = new Date().getTime();
console.log(
  "\tTotal time: ",
  endTime - startTime,
  "ms - Avg time: ",
  (endTime - startTime) / iterations,
  "ms"
);
```

06-delete-table.js

06-delete-table.js プログラムは TryDaxTable を削除します。テストが完了してから、このプログラムを実行します。

```
const AmazonDaxClient = require("amazon-dax-client");
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region,
});

var dynamodb = new AWS.DynamoDB(); //low-level client
```

```
var tableName = "TryDaxTable";

var params = {
  TableName: tableName,
};

dynamodb.deleteTable(params, function (err, data) {
  if (err) {
    console.error(
      "Unable to delete table. Error JSON:",
      JSON.stringify(err, null, 2)
    );
  } else {
    console.log(
      "Deleted table. Table description JSON:",
      JSON.stringify(data, null, 2)
    );
  }
});
```

DynamoDB のドキュメント履歴

2018 年 7 月 3 日以降の DynamoDB デベロッパーガイドの各リリースにおける重要な変更点を以下の表に示します。このドキュメントの更新に関する通知については、RSS フィード (このページの左上) にサブスクライブできます。

変更	説明	日付
Amazon S3 への DynamoDB エクスポートがフィルタリングされたエクスポートをサポートするようになりました	DynamoDB テーブルデータについて、テーブル全体ではなくサブセットを Amazon S3 にエクスポートできるようになりました。これを行うには、エクスポートリクエストでキー条件式、フィルター式、プロジェクション式を指定します。フィルタリングされたエクスポートは、フルエクスポートと増分エクスポートの両方で機能します。詳細については、「Amazon S3 への DynamoDB データのエクスポート: 仕組み」を参照してください。	2026 年 10 月 1 日
ベクトル検索ガイダンスと AI フレームワークコネクタによるエージェンティック AI ドキュメントの拡充	RAG とエージェントメモリのベクトル検索ガイダンスを追加してエージェンティック AI の章を更新し、セッションの永続性、チャット履歴、長期エージェントメモリ、セマンティック検索を網羅する Strands Agents、LangChain、LangGraph コネクタの詳細トピックを追加しました。詳細については、	2026 年 8 月 31 日

	「DynamoDB でのエージェンティック AI の使用」を参照してください。	
DAX が商用リージョンと中国リージョンで AWS PrivateLink をサポートするようになりました	DynamoDB Accelerator (DAX) で AWS PrivateLink を使用して、同じ AWS リージョン内で DAX コントロールプレーンエンドポイントへのプライベートで安全な接続を確立するのに役立つガイダンスを追加しました。詳細については、「Console-to-Code を使用して Amazon DynamoDB のインフラストラクチャコードを生成する」を参照してください。	2025 年 10 月 30 日
DynamoDB がインターネットプロトコルのバージョン 6 (IPv6) をサポートするようになりました	DynamoDB は、AWS PrivateLink ゲートウェイおよびインターフェイスエンドポイントを含む DynamoDB テーブル、ストリーム、および DAX に接続するときに、Amazon VPC のインターネットプロトコルバージョン 6 (IPv6) アドレスをサポートするようになりました。IPv6 を使用すると、ネットワークスタックを簡素化し、IPv4 と IPv6 の両方をサポートするネットワークでコンプライアンス要件を満たすことができます。	2025 年 10 月 3 日

DynamoDB の新しい Console-to-Code 機能のドキュメントを追加	Console-to-Code は、AWS マネジメントコンソールでの手動の DynamoDB テーブル作成ステップを、AWS CDK や CloudFormation などの複数の形式で自動化され、再現可能なインフラストラクチャコードに変換します。詳細については、「Console-to-Code を使用して Amazon DynamoDB のインフラストラクチャコードを生成する」を参照してください。	2025 年 9 月 23 日
DynamoDB Local バージョン 3.1.0 をリリース	この更新により、Joda 時間の依存関係を含む PartiQL クエリのパフォーマンスが向上します。	2025 年 9 月 14 日

DynamoDB は、新しい CloudWatch Contributor Insights モード、スロットリング例外の詳細、およびターゲット CloudWatch メトリクスを使用してスロットリング診断機能を強化しました	DynamoDB は、CloudWatch Contributor Insights の新しい `Throttled keys` 専用モードで拡張スロットリング診断を提供します。これにより、最もスロットリングされたパーティションキー、構造化 `ThrottlingReason` フィールドを含むより詳細なスロットリング例外、スロットリングされるリソースと理由を正確に特定するための特定の理由コードとリソース ARN、および特定のスロットリングイベントを正確にモニタリングするためのさまざまなスロットリングシナリオに対応する 16 の新しいターゲット CloudWatch メトリクスが具体的に追跡されます。詳細については、「DynamoDB でのスロットリングのトラブルシューティング」を参照してください。	2025 年 8 月 14 日

キャパシティモードを切り替えるための制限の引き上げ	テーブルをプロビジョンドキャパシティモードからオンデマンドキャパシティモードに切り替える制限を 24 時間に 1 回から 24 時間に 4 回に引き上げました。この変更により、1 日に複数回大量のデータをロードする必要があるワークロードや、予期しないトラフィックの増加に対応する必要があるワークロードの柔軟性が向上します。詳細については、「DynamoDB でキャパシティモードを切り替える際の考慮事項」を参照してください。	2025 年 8 月 12 日
DynamoDB Local バージョン 3.0 がリリースされ、AWS SDK for Java V1 から V2 に移行しました。	この更新により、V1 の依存関係が削除され、パフォーマンスが向上し、AWS SDK V2 パッケージ構造と一致します。	2025 年 7 月 17 日
DynamoDB Streams で最適化されたシャード検出のサポートを追加	新しいシャード検出メカニズムは、初期検出時間を短縮し、定常状態の処理パフォーマンスを向上させ、処理ステータスを可視化し、大規模なオペレーション全体で一貫した p100 レイテンシーを維持します。詳細については、「AWS PrivateLink for Amazon DynamoDB Streams」を参照してください。	2025 年 7 月 17 日

DynamoDB グローバルテーブルに対する、マルチリージョンの強力な整合性 (MRSC) のサポートを追加	DynamoDB は、オプションの監視リージョンを含むグローバルテーブルのマルチリージョンの強力な整合性 (MRSC) をサポートするようになりました。MRSC は、リージョンの障害時に目標復旧時点 (RPO) が 0 のリージョン間で強力な整合性のある読み込みを許可します。MRSC グローバルテーブルは、3 つのレプリカまたは 2 つのレプリカと 1 つの監視を含む、厳密に 3 つのリージョンにデプロイする必要があります。詳細と利用可能なリージョンについては、「マルチリージョンの強力な整合性 (MRSC)」を参照してください。	2025 年 6 月 30 日
Amazon DynamoDB Streams Kinesis Client Library (KCL) 2.x アダプター	DynamoDB は、KCL 2.x および AWS SDK v2 と互換性のあるアップグレードされた DynamoDB Streams アダプターをサポートするようになりました。詳細については、「KCL1.x から KCL 3.x への移行」を参照してください。	2025 年 6 月 11 日

DynamoDB local を AWS CloudShell で実行するためのサポートを追加	DynamoDB local は、ソフトウェアをダウンロードまたはインストールすることなく、AWS マネジメントコンソール を介してブラウザで直接実行できるようになりました。詳細については、「コンピュータ上で DynamoDB をローカルでデプロイする」を参照してください。	2025 年 5 月 19 日
DAX での r7i インスタンスタイプのサポートを追加	DAX のインスタンスタイプオプションを拡張し、パフォーマンスとキャパシティの柔軟性を向上させました。新しい r7i インスタンスタイプとその機能の詳細については、「保管時の DAX 暗号化」を参照してください。	2025 年 4 月 29 日
新しいスコープダウンフルアクセスポリシー AmazonDynamoDBFullAccess_v2 で AmazonDynamoDBFullAccess を置き換えます。	2025 年 4 月 25 日をもって、DynamoDB は AmazonDynamoDBFullAccess ポリシーを廃止し、AmazonDynamoDBFullAccess_v2 という名前のスコープダウンポリシーに置き換えました。詳細については、「AWS マネージドポリシー: AmazonDynamoDBFullAccess_v2」を参照してください。	2025 年 4 月 28 日

DynamoDB に DynamoDB Streams の AWS PrivateLink サポートを導入	AWS に DynamoDB Streams の AWS PrivateLink サポートが追加されました。これにより、お客様はプライベートネットワーク接続を使用して Amazon Virtual Private Cloud (VPC) 内から DynamoDB Streams API を呼び出すことができます。この機能を使用すると、パブリックインターネットを経由することなく、簡素化されたプライベートネットワークアクセスが可能になり、DynamoDB ワークロードのコンプライアンスとセキュリティ要件に準拠できます。詳細については、「AWS PrivateLink for Amazon DynamoDB Streams」を参照してください。	2025 年 3 月 26 日
DAX SDK For JS 3.x および Go 2.x が利用可能になりました	DynamoDB Accelerator (DAX) SDK for JS 3.x が利用可能になりました。これは、AWS SDK for Java 3.x と互換性があります。	2025 年 3 月 17 日
デフォルトテーブル設定の NoSQL Workbench 3.13.5 オンデマンドキャパシティモードのリリース	デフォルト設定でテーブルを作成すると、DynamoDB はプロビジョニングされたキャパシティモードの代わりにオンデマンドキャパシティモードを使用するテーブルを作成します。	2025 年 2 月 24 日

DAX クライアントを設定するための新しいベストプラクティス	DAX クライアントを設定するための新しい DAX 規範ガイダンスのベストプラクティストピックを公開しました。詳細については、「Configuring your DAX client」を参照してください。	2025 年 2 月 17 日
バルクデータオペレーションに関する新しいベストプラクティス	DynamoDB で複雑なデータモデルを操作するための新しいベストプラクティストピックを公開しました。詳細については、「Best practices for using bulk data operations in DynamoDB」を参照してください。	2025 年 1 月 9 日
Amazon DynamoDB が設定可能なポイントインタイムリカバリ (PITR) をサポートするようになりました	DynamoDB は PITR の設定可能なリカバリ期間をサポートするようになりました。各テーブルの PITR 期間を 1 ~ 35 日間の任意の値に設定できるようになりました。詳細については、「DynamoDB のポイントインタイムバックアップ」を参照してください。	2025 年 1 月 7 日

Amazon Managed Streaming for Apache Kafka と Amazon DynamoDB の統合に関する新しいトピックを公開	Amazon Managed Streaming for Apache Kafka が Apache Kafka トピックからデータを読み取って DynamoDB に保存することで Amazon DynamoDB と統合する方法について説明します。詳細は、「Integrating DynamoDB with Amazon Managed Streaming for Apache Kafka」を参照してください。	2024 年 12 月 26 日
DynamoDB が SageMaker AI Lakehouse とのゼロ ETL 統合のサポートを導入	DynamoDB では、Amazon DynamoDB から顧客のデータレイクへのデータの抽出とロードを自動化するゼロ ETL 統合が導入されています。詳細については、「SageMaker AI Lakehouse との DynamoDB ゼロ ETL 統合」を参照してください。	2024 年 12 月 3 日
DynamoDB グローバルテーブルがマルチリージョンの強力な整合性をサポート	マルチリージョンの強力な整合性により、目標復旧時点 (RPO) を 0 にして高可用性のマルチリージョンアプリケーションを構築し、最高レベルの耐障害性を実現できます。詳細については、「Multi-Region strong consistency」を参照してください。	2024 年 12 月 3 日

DynamoDB マネージドポリシーのアップデート	AmazonDynamoDBReadOnlyAccess 管理ポリシーに dynamodb:GetAbacStatus と dynamodb:UpdateAbacStatus の 2 つの新しいアクセス許可が追加されました。これらのアクセス許可により、現在のリージョンで AWS アカウントの ABAC ステータスを表示し、ABAC を有効にすることができます。詳細については、「AWS 管理ポリシー: AmazonDynamoDBReadOnlyAccess」を参照してください。	2024 年 11 月 18 日
DynamoDB が属性ベースのアクセス制御 (ABAC) のサポートを導入	ABAC は、ユーザー、ロール、AWS リソースにアタッチされたタグに基づいてアクセス許可を定義できる認可戦略です。ABAC は、AWS Identity and Access Management (IAM) ポリシーまたは他のポリシーでタグベースの条件を使用して、IAM プリンシパルのタグがテーブルのタグと一致する場合に、テーブルまたはインデックスに対する特定のアクションを許可または拒否します。詳細については、「Using attribute-based access control with DynamoDB」を参照してください。	2024 年 11 月 18 日

DynamoDB がオンデマンドテーブルとプロビジョンドテーブルにウォームスループットを導入	DynamoDB はウォームスループットをサポートするようになりました。ウォームスループットは、DynamoDB テーブルが瞬時にサポートできる読み取りおよび書き込みオペレーションの数と、DynamoDB テーブルを事前ウォーミングする機能を可視化します。詳細については、「Understanding DynamoDB warm throughput」を参照してください。	2024 年 11 月 13 日
Apache Flink で Amazon DynamoDB Streams レコードを使用する機能	Apache Flink で Amazon DynamoDB Streams レコードを使用し、Amazon Managed Service for Apache Flink を活用して、エンドツーエンドのストリーム処理アプリケーションをすばやく構築および管理できるようになりました。詳細については、「What is Amazon DynamoDB?」を参照してください。	2024 年 11 月 12 日
グローバルテーブルとバックアップの請求に関する新しいトピックを公開	グローバルテーブルの請求とバックアップの請求に関する 2 つの新しいトピックを公開しました。詳細については、「Amazon DynamoDB のグローバルテーブルの請求について」および「Amazon DynamoDB のバックアップの請求について」を参照してください。	2024 年 10 月 16 日

Amazon DynamoDB の Amazon Redshift とのゼロ ETL 統合	Amazon DynamoDB の Amazon Redshift とのゼロ ETL 統合は、DynamoDB から Amazon Redshift へのレプリケーションを備えた、コード不要で完全マネージド型の ETL パイプラインを提供します。詳細については、「Amazon DynamoDB の Amazon Redshift とのゼロ ETL 統合」を参照してください。	2024 年 10 月 15 日
DynamoDB での生成 AI の使用に関するトピックを追加するための、ドキュメントのみの更新	DynamoDB での生成 AI の使用に関する情報を提供する新しいトピックを公開しました。これには、DynamoDB での生成 AI のユースケースも含まれています。詳細については、DynamoDB での生成 AI の使用」を参照してください。	2024 年 10 月 11 日
SDK が AWS アカウントベースのエンドポイントをサポート開始	アカウントベースのエンドポイントと SDK クライアントの `ACCOUNT_ID_ENDPOINT_MODE` 設定に関するドキュメントを追加しました。詳細については、「AWS アカウントベースのエンドポイントの SDK サポート」を参照してください。	2024 年 9 月 3 日

開始方法エクスペリエンスの再設計	開始方法エクスペリエンスを再設計し、情報を統合してオンボーディングの開始を迅速化しました。詳細については、「DynamoDB の開始方法」を参照してください。	2024 年 8 月 1 日
スペインとスウェーデンの新しいリージョンへの DAX の拡張	DAX がスペインおよびスウェーデンのリージョンで利用可能になりました。詳細については、「DAX クラスターのコンポーネント」を参照してください。	2024 年 7 月 30 日
DynamoDB のバックアップと復元のドキュメントを再構成、統合	DynamoDB デベロッパーガイドには、バックアップと復元のための新しい構造があります。詳細については、「DynamoDB のバックアップと復元」を参照してください。	2024 年 7 月 2 日
Amazon DynamoDB とは」のトピックを更新	「Amazon DynamoDB とは」のトピックの改訂 / 更新版を公開しました。詳細については、「Amazon DynamoDB とは」を参照してください。	2024 年 6 月 21 日
DynamoDB Streams と EventBridge の統合	DynamoDB Streams と EventBridge の統合に関する新しいトピックを公開しました。詳細については、「Integrating with EventBridge」を参照してください。	2024 年 6 月 21 日

DAX の規範ガイダンス	DynamoDB Accelerator の効果的な使用のための包括的なインサイトを提供する新しいベストプラクティストピックを公開しました。このトピックでは、パフォーマンスの最適化、コスト管理、運用のベストプラクティスについて説明します。詳細については、「DAX の規範ガイダンス」を参照してください。	2024 年 6 月 3 日
DynamoDB テーブルをアカウントから別のアカウントに移行する	あるアカウントから別のアカウントへの DynamoDB テーブルの移行に関する新しいトピックを追加しました。詳細については、「DynamoDB テーブルをアカウントから別のアカウントに移行する」を参照してください。	2024 年 5 月 29 日
DynamoDB のモニタリングとログ記録のドキュメントを再構成、統合	DynamoDB でのモニタリングとログ記録の新しい構造には、メトリクス、ログ記録オペレーション、投稿者のインサイトに関する 3 つの簡潔な章が含まれています。	2024 年 5 月 3 日

DynamoDB キャパシティモードのドキュメントを再構成、統合	DynamoDB ガイドに、DynamoDB キャパシティモード (オンデマンドとプロビジョンド) に関するすべての情報を示す新しい章が追加されました。この更新では、「読み込み/書き込みキャパシティモードを変更する際の考慮事項」トピックが、「ベストプラクティス」章に移動されました。このトピックは、「キャパシティモードを切り替える際の考慮事項」に名前が変更され、キャパシティモードを切り替える際のベストプラクティスに関する詳細な情報が追加されました。さらに、このガイドには、DynamoDB の読み込み/書き込みと、読み込み/書き込みオペレーションのキャパシティユニットの消費量に関するすべての情報を示す新しい章が追加されました。詳細については、「DynamoDB のスループットキャパシティ」、「キャパシティモードを切り替える際の考慮事項」、「DynamoDB の読み込みと書き込み」を参照してください。	2024 年 5 月 1 日

オンデマンドリクエストの最大数	個別のテーブル、インデックス、またはその両方で実行できるオンデマンドリクエストの最大数を指定できるようになりました。最大のオンデマンドスループットを指定すると、テーブルレベルの使用量とコストを制限し、リソース消費量の意図しない急増を防止できます。詳細については、「オンデマンドテーブルの最大スループット」を参照してください。	2024 年 5 月 1 日
NoSQL Workbench オペレーションビルダーの改善点	NoSQL Workbench には、ダークモードのネイティブサポートが含まれるようになりました。オペレーションビルダーのテーブルと項目のオペレーションを改善しました。項目の結果とオペレーションビルダーのリクエスト情報は JSON 形式で利用できます。詳細については、「NoSQL Workbench オペレーションビルダー」を参照してください。	2024 年 4 月 24 日

Amazon DynamoDB リソース向けのリソースベースのポリシー	テーブル、インデックス、ストリーム用のリソースベースのポリシーが DynamoDB でサポートされるようになりました。リソースベースのポリシーでは、各リソースにアクセスできるユーザーと、各リソースに対して実行できるアクションを指定することで、アクセス許可を定義できます。詳細については、「DynamoDB 用のリソースベースのポリシーを使用する」を参照してください。	2024 年 3 月 20 日
DynamoDB マネージドポリシーのアップデート	AmazonDynamoDBReadOnlyAccess マネージドポリシーに新しいアクセス許可 dynamodb:GetResourcePolicy を追加しました。このアクセス許可は、DynamoDB リソースにアタッチされたリソースベースのポリシーの読み取りアクセスを提供します。詳細については、「AWS 管理ポリシー: AmazonDynamoDBReadOnlyAccess」を参照してください。	2024 年 3 月 20 日

AWS PrivateLinkAmazon DynamoDB 向けの	AWS PrivateLink が Amazon DynamoDB でサポートされるようになりました。AWS PrivateLink では、インターフェイス VPC エンドポイントおよびプライベート IP アドレスを使用して、仮想プライベートクラウド (VPC)、DynamoDB、およびオンプレミスデータセンター間のプライベートネットワーク接続を簡素化できます。詳細については、「AWS PrivateLink for DynamoDB」を参照してください。	2024 年 3 月 19 日
JavaScript を使ったプログラミングのためのガイド	Amazon DynamoDB は、AWS SDK for JavaScript プログラミングガイドを提供しています。AWS SDK for JavaScript、抽象化レイヤー、接続設定、エラー処理、再試行ポリシーの定義、キープアライブの管理などについて学ぶことができます。詳細については、「JavaScript による Amazon DynamoDB のプログラミング」を参照してください。	2024 年 3 月 6 日

AWS SDK for Java 2.x を使ったプログラミングのためのガイド	高レベル、低レベル、ドキュメントインタフェース、HTTP クライアントとその構成、エラー処理の詳細、SDK for Java 2.x を使用する際に考慮すべき最も一般的な構成設定について説明する新しいプログラミングガイドを作成しました。詳細については、「AWS SDK for Java 2.x を使用した Amazon DynamoDB のプログラミング」を参照してください。	2024 年 3 月 5 日
NoSQL Workbench を使用したテーブルのクローン作成	開発者は NoSQL Workbench を使用して、開発環境とリージョン (DynamoDB ローカルと DynamoDB ウェブ) 間でテーブルのコピーやクローンを作成することができます。詳細については、「NoSQL Workbench を使用したテーブルのクローン作成」を参照してください。	2024 年 2 月 26 日
Python を使ったプログラミングのためのガイド	高レベルライブラリ、低レベルライブラリの詳細、Python SDK を使用する際に考慮すべき最も一般的な設定について説明する新しいガイドを作成しました。詳細については、「Python と Boto3 による Amazon DynamoDB のプログラミング」を参照してください。	2024 年 1 月 5 日

Time to live (TTL) トピックのリライト	ガイド内の TTL セクションをすべてリライトしました。更新されたセクションにはすぐに使用できるコードスニペットが用意されており、TTL の使用を開始する際に役立ちます。現在提供されているコードスニペットは Python と Javascript で記述されています。詳細については、「TTL」を参照してください。	2023 年 12 月 20 日
AWS の請求および使用状況レポートを理解するためのベストプラクティス	DynamoDB のさまざまな使用タイプとそれらの使用タイプの料金を明確にする新しいセクションを追加しました。詳細については、「AWS の請求および使用状況レポートを理解するためのベストプラクティス」を参照してください。	2023 年 12 月 15 日
Amazon OpenSearch Service と Amazon DynamoDB のゼロ ETL 統合	Amazon OpenSearch Service とのゼロ ETL 統合が Amazon DynamoDB でサポートされるようになりました。これにより、カスタムコードやインフラストラクチャを使用せずに、DynamoDB データを自動的に複製および変換することで検索を実行できます。詳細については、「DynamoDB と Amazon OpenSearch Service のゼロ ETL 統合」を参照してください。	2023 年 11 月 28 日

リレーショナルデータベースから DynamoDB への移行	リレーショナルデータベースから DynamoDB への移行方法をユーザーが理解できるように、移行ガイドを作成しました。	2023 年 11 月 27 日
NoSQL Workbench でサンプルデータを生成する	NoSQL Workbench for Amazon DynamoDB で、サンプルデータ型テンプレートから直接データ型を作成できるようになったため、ワークロードのデータスキーマを設計しやすくなりました。この機能を使用して、DynamoDB でアプリケーションを構築するときに NoSQL データモデリングのベストプラクティスに慣れることができます。	2023 年 9 月 28 日
S3 への増分エクスポート	挿入、更新、削除されたデータをわずかな増分ずつエクスポートできるようになりました。増分エクスポートでは、AWS 管理コンソール、API 呼び出し、または AWS コマンドラインインターフェイスで数回クリックするだけで、数メガバイトからテラバイト単位までの変更データをエクスポートできます。	2023 年 9 月 26 日

DynamoDB のデータモデリング	特定のユースケース、そのアクセスパターン、およびそれらのアクセスパターンを実現するためのステップバイステップのガイダンスにに焦点を当てた DynamoDB の例を使用することで、データモデリングについて詳しく学習できるようになりました。	2023 年 7 月 14 日
トラブルシューティング」セクション	DynamoDB テーブルで発生する可能性のあるレイテンシーとスロットリングの問題について、トラブルシューティングのコンテンツが見つかるようになりました。	2023 年 3 月 13 日
Amazon DynamoDB の削除保護	すべての AWS リージョンの Amazon DynamoDB テーブルで削除保護を利用できるようになりました。DynamoDB では、通常のテーブル管理オペレーションを実行する際に、テーブルが誤って削除されないように保護できるようになりました。	2023 年 3 月 8 日

グローバルテーブルでの KDSD に対する CloudFormation のサポート	Amazon Kinesis Data Streams for DynamoDB で DynamoDB グローバルテーブルに対して CloudFormation がサポートされるようになりました。これにより、CloudFormation テンプレートを使用して DynamoDB グローバルテーブル上の Amazon Kinesis Data Streams へのストリーミングを有効化できるようになりました。	2023 年 2 月 15 日
DynamoDB ローカルは 1 トランザクションあたり 100 アクションをサポート	DynamoDB ローカルでは、1 つのトランザクションで最大 100 のアクションを実行できるようになりました。	2023 年 2 月 9 日
DynamoDB の Well-Architected レンズを使用して DynamoDB ワークロードを最適化する	優れたアーキテクチャの DynamoDB ワークロードを設計するための一連の設計原則とガイダンス、DynamoDB Well-Architected レンズを使用できるようになりました。	2023 年 2 月 3 日
PartiQL GovGloud のサポート	AWS GovCloud (米国東部) および AWS GovCloud (米国西部) で PartiQL (Amazon DynamoDB の SQL 互換のクエリ言語) がサポートされるようになりました。	2022 年 12 月 21 日

NoSQL ワークベンチおよび DynamoDB ローカルのシングルインストールスイート	NoSQL Workbench for DynamoDB に、ガイド付きの DynamoDB ローカルインストールプロセスが含まれるようになり、DynamoDB ローカル開発環境の設定が効率化されました。	2022 年 12 月 6 日
S3 からの一括インポート	Amazon DynamoDB では、Amazon S3 からの一括データインポートをサポートすることで、新しい DynamoDB テーブルへのデータの移行とロードが簡単になりました。	2022 年 8 月 18 日
Service Quotas との高度な統合	Service Quotas では、アカウントとテーブルクォータをプロアクティブに管理できるようになりました。現在の値を表示したり、クォータの使用率が設定可能なしきい値を超えた場合のアラームを設定したりできます。	2022 年 6 月 15 日
NoSQL Workbench がテーブルと GSI のサポートを追加	NoSQL Workbench をテーブルとグローバルセカンダリインデックス (GSI) のコントロールプレーンオペレーション (CreateTable、UpdateTable、DeleteTable など) に使用できるようになりました。	2022 年 6 月 2 日

Standard-Infrequent Access テーブルクラスが中国で利用可能に	Amazon DynamoDB Standard-Infrequent Access テーブルクラスが中国リージョンで利用可能です。この新しいテーブルクラスをアクセス頻度の低いデータを格納するテーブルで使用することで、DynamoDB のコストが最大 60% 削減されます。	2022 年 4 月 18 日
デフォルトの Service Quotas とテーブル管理オペレーションの増加	DynamoDB では、アカウントおよびリージョンあたりのテーブル数のデフォルトクォータが 256 テーブルから 2,500 テーブルに増加し、同時実行テーブルの管理オペレーションの数が 50 から 500 に増加しました。	2022 年 3 月 9 日
DynamoDB 用の PartiQL を使用して項目を制限するオプション	DynamoDB では、各リクエストのオプションパラメータとして、DynamoDB オペレーションの PartiQL で処理するアイテム数を制限することができます。	2022 年 3 月 8 日
AWS Backup 統合が北京および寧夏リージョンで利用可能に	AWS Backup が、中国 (北京および寧夏) リージョンで DynamoDB に統合されるようになりました。アカウント間、リージョン間のバックアップなど、AWS Backup でのバックアップ機能の強化により、コンプライアンスやビジネス継続性の要件をより容易に満たすことができます。	2022 年 1 月 26 日

PartiQL API コールによるスループットキャパシティに関する情報	DynamoDB は、PartiQL API コールによって消費されたスループット容量を返すことができ、クエリとスループットコストの最適化を支援します。	2022 年 1 月 18 日
AWS Backup の統合	DynamoDB では、アカウント間、リージョン間のバックアップなど、AWS Backup でのバックアップ機能の強化により、コンプライアンスやビジネス継続性の要件をより簡単に満たすことができるようになりました。	2021 年 11 月 24 日
CSV での NoSQL Workbench データセットの読み込み/書き出し	Amazon DynamoDB 用 NoSQL Workbench では、データモデルの構築と可視化に役立つサンプルデータのインポートと自動投入が可能になりました。	2021 年 10 月 11 日
Amazon DynamoDB Streams データプレーンアクティビティを AWS CloudTrail でフィルタリングして取得する	Amazon DynamoDB では、Streams のデータプレーン API アクティビティを AWS CloudTrail でフィルタリングできるようになり、監査ログをより詳細に制御できるようになりました。	2021 年 9 月 22 日

コンソールの更新	DynamoDB コンソールがデフォルトのコンソールになり、データ管理を容易にし、一般的なタスクを簡素化し、リソースや機能へより迅速にアクセスできるようになりました。	2021 年 8 月 25 日
DAX SDK For Java 2.x が利用可能に	DynamoDB Accelerator (DAX) SDK for Java 2.x が利用可能になりました。これは、AWS SDK for Java 2.x と互換性があります。ノンブロック I/O を含む最新機能のメリットを得ることができます。	2021 年 7 月 29 日
コントロールプレーンオペレーションを含む NoSQL Workbench 機能の更新	Amazon DynamoDB 用 NoSQL Workbench では、テーブルのデータを変更したりアクセスしたりするための頻繁な操作を、より簡単に実行できるようになりました。	2021 年 7 月 28 日
DynamoDB グローバルテーブルがアジアパシフィックリージョンで利用可能に	DynamoDB グローバルテーブルがアジアパシフィックリージョンで利用可能になりました。選択する 22 の AWS リージョン全体で DynamoDB テーブルを自動的に複製します。	2021 年 7 月 28 日
DAX が中国で利用可能に	DynamoDB Accelerator (DAX) が、Sinnet が運営する中国 (北京) リージョンで利用可能になりました。	2021 年 7 月 28 日

転送時の DAX 暗号化	DynamoDB Accelerator (DAX) では、アプリケーションと DAX クラスター間、および DAX クラスター内のノード間のデータ転送時の暗号化がサポートされるようになりました。	2021 年 7 月 24 日
CloudFormation と CloudTrail の統合	AWS CloudFormation との統合 および CloudFormation データプレーンのログ記録によるセキュリティの機能強化。	2021 年 6 月 18 日
グローバルテーブルが CloudFormation をサポート	Amazon DynamoDB グローバルテーブルで AWS CloudFormation がサポートされるようになり、CloudFormation テンプレートを使用してグローバルテーブルを作成し、その設定を管理できます。	2021 年 5 月 14 日
Amazon DynamoDB Local が Java 2.x をサポート	Amazon DynamoDB のダウンロード版である DynamoDB ローカル で、AWS SDK for Java 2.x を使用できるようになりました。DynamoDB local を使用すると、ローカル開発環境で実行されている DynamoDB のバージョンを使用して、追加コストをかけずにアプリケーションを開発およびテストできます。	2021 年 5 月 3 日

NoSQL Workbench が AWS CloudFormation をサポート	Amazon DynamoDB 用 NoSQL Workbench で AWS CloudFormation がサポートされるようになり、CloudFormation テンプレートで DynamoDB のデータモデルを管理および変更できるようになりました。さらに、NoSQL Workbench でテーブルの容量設定を構成できるようになりました。	2021 年 4 月 22 日
DynamoDB と AWS Amplify の機能が統合	AWS Amplify では、1 つのデプロイで複数の DynamoDB グローバルセカンダリインデックスの更新をオーケストレートできるようになりました。	2021 年 4 月 20 日
AWS CloudTrail で Amazon DynamoDB Streams のデータプレーン API を口グに記録する	AWS CloudTrail を使用して Amazon DynamoDB Streams のデータプレーン API アクティビティを口グに記録し、DynamoDB テーブルの項目レベルの変更をモニタリングおよび調査することができるようになりました。	2021 年 4 月 20 日

Amazon DynamoDB 用 Amazon Kinesis Data Streams で AWS CloudFormation がサポートされるようになりました	Amazon DynamoDB 用 Amazon Kinesis Data Streams で AWS CloudFormation がサポートされるようになり、CloudFormation テンプレートを使用して DynamoDB テーブル上の Amazon Kinesis Data Streams へのストリーミングを有効化できるようになりました。DynamoDB データの変更を Kinesis Data Streams にストリーミングすることで、Amazon Kinesis のサービスを使用して高度なストリーミングアプリケーションを構築できます。	2021 年 4 月 12 日
Amazon Keyspaces が FIPS 140-2 準拠のエンドポイントを提供	Amazon Keyspaces (Apache Cassandra 用) が、連邦情報処理規格 (FIPS) 140-2 準拠のエンドポイントを提供し、高度に規制されたワークロードをより簡単に実行できるようになりました。FIPS 140-2 は、機密情報を保護する暗号モジュールのセキュリティ要件を規定する米国およびカナダ政府の規格です。	2021 年 4 月 8 日

DAX 用の Amazon EC2 T3 インスタンス	DAX で Amazon EC2 T3 インスタンスタイプがサポートされるようになりました。このインスタンスタイプには、CPU パフォーマンスのベースラインレベルを設定でき、必要な場合にそのレベルを超えてバーストする機能を備えています。	2021 年 2 月 15 日
NoSQL Workbench for Amazon DynamoDB の PartiQL サポート	Amazon DynamoDB 用 NoSQL Workbench を使用して、DynamoDB 用に PartiQLステートメントを作成できるようになりました。	2020 年 12 月 4 日
DynamoDB 用 PartiQL	DynamoDB 用 PartiQL (SQL 互換クエリ言語) を使用して、DynamoDB テーブルを操作できるようになりました。また、PartiQL 用に AWS マネジメントコンソール、AWS Command Line Interface、もしくは DynamoDB API を使用して、アドホックのクエリを実行することもできます。	2020 年 11 月 23 日
Amazon DynamoDB 用 Amazon Kinesis Data Streams	Amazon DynamoDB 用 Amazon Kinesis Data Streams と DynamoDB テーブルを使用して、項目レベルの変更をキャプチャし、その結果を Kinesis Data Streams にレプリケートできるようになりました。	2020 年 11 月 23 日

DynamoDB テーブルのエクスポート	DynamoDB テーブルを Amazon S3 にエクスポートできるようになりました。この機能により、Athena、AWS Glue、Lake Formation などのサービスを使用しながら、データに対する分析や複雑なクエリを実行できます。	2020 年 11 月 9 日
空の値のサポート	DynamoDB テーブルにおいて、非キー文字列およびバイナリ属性の空の値を使用できるようになりました。空の値のサポートにより、DynamoDB に送信する前に属性を変換しなくてもよくなり、より広範なユースケースで属性を使用するための柔軟性が高まります。空の文字列とバイナリ値では、リスト作成、マッピング、およびデータ型の設定も行えます。	2020 年 5 月 18 日
NoSQL Workbench for Amazon DynamoDB での Linux サポート	NoSQL Workbench for Amazon DynamoDB で、Linux- Ubuntu、Fedora および Debian のサポートが開始されました。	2020 年 5 月 4 日

CloudWatch Contributor Insights for DynamoDB – 一般提供	DynamoDB 用 CloudWatch Contributor Insights の一般提供が開始されました。DynamoDB 用 CloudWatch Contributor Insights は、DynamoDB テーブルにおけるトラフィックの傾向を一目で把握できる診断ツールです。テーブルで最も頻繁にアクセスされるキー (ホットキーとも呼ばれます) を特定するのに役立ちます。	2020 年 4 月 2 日
グローバルテーブルのアップグレード	グローバルテーブルのバージョン 2017.11.29 からの、最新のバージョン (2019.11.21) への更新が、DynamoDB コンソールで数回クリックするだけで行えるようになりました。グローバルテーブルのバージョンをアップグレードすることで、既存のテーブルを他の AWS リージョンに拡張する際にもテーブルの再構築は必要なくなり、DynamoDB テーブルの可用性を簡単に向上させることができます。	2020 年 3 月 16 日
Amazon DynamoDB 用 NoSQL Workbench – 一般提供	Amazon DynamoDB 用 NoSQL Workbench の一般提供が開始されました。DynamoDB テーブルの設計、作成、クエリ、および管理に、NoSQL Workbench をご利用いただけます。	2020 年 3 月 2 日

DAX キャッシュクラスターのメトリクス	DAX で新たな CloudWatch メトリクスがサポートされ、DAX クラスターのパフォーマンスをよりよく把握できるようになりました。	2020 年 2 月 6 日
DynamoDB 用 CloudWatch Contributor Insights – プレビュー	DynamoDB 用 CloudWatch Contributor Insights は、DynamoDB テーブルにおけるトラフィックの傾向を一目で把握できる診断ツールです。テーブルで最も頻繁にアクセスされるキー (ホットキーとも呼ばれます) を特定するのに役立ちます。	2019 年 11 月 26 日
不均衡なワークロードのための適応型キャパシティのサポート	Amazon DynamoDB の適応型キャパシティーにより、頻繁にアクセスされるアイテムを自動的に分離することで、不均衡なワークロードを処理できるようになりました。アプリケーションが、1 つ以上のアイテムに不均等な高トラフィックを送っている場合、DynamoDB がパーティションのバランスを再調整することで、アクセス頻度の高いアイテムが同一パーティションに偏ることを防ぎます。	2019 年 11 月 26 日

カスタマーマネージドキーのサポート	DynamoDB でカスタマーマネージドキーのサポートが開始されました。つまり、DynamoDB データの暗号化やセキュリティ管理を完全に制御できます。	2019 年 11 月 25 日
NoSQL Workbench による DynamoDB local (ダウンロード可能バージョン) のサポート	NoSQL Workbench では、DynamoDB local (ダウンロード可能バージョン) に接続して、DynamoDB テーブルを設計、作成、クエリ、管理できるようになりました。	2019 年 11 月 8 日
NoSQL Workbench – プレビュー	これは DynamoDB 用 NoSQL Workbench の初版リリースです。NoSQL Workbench を使用して、DynamoDB テーブルを設計、作成、クエリ、および管理します。詳細については、「Amazon DynamoDB 用 NoSQL Workbench (プレビュー)」を参照してください。	2019 年 9 月 16 日
DAX が Python と .NET を使用したトランザクションオペレーションのサポートを追加	DAX は、Go、Java、.NET、Node.js、および Python で記述されたアプリケーション用の `TransactWriteItems` と `TransactGetItems` API をサポートしています。詳細については、「インメモリアクセラレーションと DAX」を参照してください。	2019 年 2 月 14 日

Amazon DynamoDB local (ダウンロード可能バージョン) の更新	DynamoDB local (ダウンロード可能バージョン) がトランザクション API、オンデマンドの読み取り/書き込みキャパシティー、読み取りおよび書き込みオペレーションのキャパシティー報告、20 個のグローバルセカンダリインデックスをサポートするようになりました。詳細については、「ダウンロード可能な DynamoDB と DynamoDB ウェブサービスの違い」を参照してください。	2019 年 2 月 4 日
Amazon DynamoDB オンデマンド	DynamoDB オンデマンドは、容量計画なしで 1 秒あたりに数千ものリクエストを処理できる柔軟な請求オプションです。DynamoDB オンデマンドには、読み取りおよび書き込みリクエストのリクエストごとの支払い料金が用意されているため、使用した分だけ課金されます。詳細については、「DynamoDB スループットキャパシティ」を参照してください。	2018 年 11 月 28 日

Amazon DynamoDB Transactions	DynamoDB のトランザクションは、テーブル内およびテーブル間の複数の項目に調整されたオールオアナッシングの変更を行うことで、DynamoDB にアトミック性、整合性、分離、および堅牢性 (ACID) をもたらします。詳細については、「Amazon DynamoDB Transactions」をご覧ください。	2018 年 11 月 27 日
Amazon DynamoDB により、お客様の保管時のデータをすべて暗号化	DynamoDB の保管時の暗号化は、そのプライマリキー、ローカルおよびグローバルセカンダリインデックス、ストリーム、グローバルテーブル、バックアップ、DAX クラスター (データが堅牢なメディアに保存される場合) を含む、暗号化されたテーブルでデータを保護することにより、追加のデータ保護レイヤーを提供します。詳細については、「保管時の Amazon DynamoDB 暗号化」を参照してください。	2018 年 11 月 15 日

新しいドッカーイメージで Amazon DynamoDB ローカルをさらに容易に	DynamoDB のダウンロード可能なバージョンである、DynamoDB local の使用がさらに容易になりました。新たな DynamoDB local Docker イメージによる、DynamoDB アプリケーションの開発とテストが行えます。詳細およびダウンロード手順については、「DynamoDB (Downloadable Version) and Docker」を参照してください。	2018 年 8 月 22 日
DynamoDB Accelerator (DAX) に保管時の暗号化のサポートが追加されました	DynamoDB Accelerator (DAX) で、新しい DAX クラスターの保管時の暗号化が行えます。セキュリティが重要で、厳格なコンプライアンスと規制要件の対象となるアプリケーションでの、 Amazon DynamoDB テーブルからの読み込みが高速化されます。詳細については、「DAX Encryption at Rest」を参照してください。	2018 年 8 月 9 日
DynamoDB ポイントインタイムリカバリ (PITR) で削除されたテーブル復元のサポートが追加されました	ポイントインタイムリカバリが有効になっているテーブルを削除すると、システムバックアップが自動的に作成され、35 日間 (追加料金なしで) 保持されます。詳細については、「ポイントインタイムリカバリを使用して開始する前に」を参照してください。	2018 年 8 月 7 日

更新を RSS で今すぐ入手可能	RSS フィード (このページの左上) にサブスクライブして、Amazon DynamoDB 開発者ガイドの更新についての通知を受信できるようになりました。	2018 年 7 月 3 日

以前の更新

以下の表に、2018 年 7 月 3 日以前の DynamoDB デベロッパーガイドにおける重要な変更点を示します。

変更	説明	変更日
DAX での Go 言語のサポート	DynamoDB Accelerator (DAX) SDK for Go を使用して Go プログラム言語で記述されたアプリケーションによる、Amazon DynamoDB テーブルのマイクロ秒レベルでの高速読み込みができるようになりました。詳細については、「DAX SDK for Go」を参照してください。	2018 年 26 月 6 日
DynamoDB から SLA がリリース	DynamoDB は、パブリックアベイラビリティの SLA をリリースしました。詳細については、「Amazon DynamoDB サービスレベルアグリーメント」を参照してください。	2018 年 6 月 19 日
DynamoDB での継続的なバックアップとポイントインタイムリカバリ (PITR)	ポイントインタイムリカバリを使用することで、オペレーションによって Amazon DynamoDB テーブルが誤っ	2018 年 4 月 25 日

変更	説明	変更日
	て上書きされたり削除されたりしないようにできます。ポイントインタイムリカバリを有効化すれば、オンデマンドバックアップの作成、維持、スケジュールを心配する必要はありません。たとえば、テストスクリプトで、誤って本稼働環境の DynamoDB テーブルに書き込みを行ったとします。ポイントインタイムリカバリを使用すれば、過去 35 日間の任意の時点にテーブルを復元することができます。DynamoDB では、テーブルの増分バックアップが維持されます。詳細については、「DynamoDB のポイントインタイムバックアップ」を参照してください。	
DynamoDB での保管時の暗号化	新しい DynamoDB テーブルで利用できる DynamoDB の保管時の暗号化は、AWS Key Management Service に保管された AWS マネージド暗号キーを使用して、アプリケーションデータを Amazon DynamoDB テーブルに保護するために便利です。詳細については、「保管時の DynamoDB 暗号化」を参照してください。	2018 年 2 月 8 日

変更	説明	変更日
DynamoDB でのバックアップと復元	オンデマンドバックアップで、DynamoDB テーブルのデータの完全なバックアップを作成してアーカイブできます。これを利用して企業および行政の既成要件を満たすことができます。数メガバイトから数百テラバイトまでのデータを持つテーブルを、本稼働アプリケーションのパフォーマンスや可用性に影響を与えずにバックアップできます。詳細については、「DynamoDB のバックアップと復元」を参照してください。	2017 年 11 月 29 日

変更	説明	変更日
DynamoDB グローバルテーブル	DynamoDB のグローバルフットプリント上に構築されたグローバルテーブルにより、完全マネージド型で、マルチリージョン、マルチアクティブなデータベースが提供されます。このデータベースでは、大規模に拡張されたグローバルアプリケーションからの読み込み/書き込みのパフォーマンスが高速に、かつローカルに実現されます。Global Tables は、選択する AWS リージョン全体で Amazon DynamoDB テーブルを自動的に複製します。詳細については、「グローバルテーブル - マルチアクティブ、マルチリージョンレプリケーション」を参照してください。	2017 年 11 月 29 日
DAX の Node.js のサポート	Node.js のデベロッパーは、Node.js 向けの DAX クライアントを使用することで、DynamoDB Accelerator (DAX) を活用できます。詳細については、「DynamoDB Accelerator (DAX) とインメモリアクセラレーション」を参照してください。	2017 年 10 月 5 日

変更	説明	変更日
DynamoDB の VPC エンドポイント	DynamoDB エンドポイントにより、Amazon VPC 内にある Amazon EC2 インスタンスは、パブリックインターネットに公開されることなく DynamoDB にアクセスできるようになります。VPC と DynamoDB 間のネットワークトラフィックは、Amazon ネットワーク内のみに置かれます。詳細については、「Amazon VPC エンドポイントを使用して DynamoDB にアクセスする」を参照してください。	2017 年 8 月 16 日

変更	説明	変更日
DynamoDB での Auto Scaling	DynamoDB での Auto Scaling では、プロビジョニングされたスループットの設定を手動で定義または調整する必要はありません。代わりに、DynamoDB の Auto Scaling では、実際のトラフィックパターンに応じて、読み込み/書き込み容量を動的に調整します。これにより、テーブルまたはグローバルセカンダリインデックスで、プロビジョニングされた読み込み/書き込みキャパシティーが拡張され、トラフィックの急激な増加をスロットリングなしに処理できるようになります。ワークロードが減少した場合には、DynamoDB の Auto Scaling はプロビジョニングされているキャパシティーを削減します。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。	2017 年 6 月 14 日

変更	説明	変更日
DynamoDB Accelerator (DAX)	DynamoDB Accelerator (DAX) は、フルマネージド型で可用性の高い、DynamoDB 用のインメモリキャッシュです。1 秒あたりのリクエスト数が数百万におよぶ場合であっても、最大 10 倍のパフォーマンスの (ミリセカンドからマイクロセカンドの範囲への) 向上を実現します。詳細については、「DynamoDB Accelerator (DAX) とインメモリアクセラレーション」を参照してください。	2017 年 4 月 19 日
DynamoDB が項目の有効期限 (TTL) による自動的な停止をサポート	Amazon DynamoDB の Time-to-Live (TTL) では、追加料金なしに、有効期限が切れた項目をテーブルから自動的に削除することができます。詳細については、「DynamoDB での Time to Live (TTL) の使用」を参照してください。	2017 年 2 月 27 日
DynamoDB がコスト配分タグをサポート	使用量の類別と細分化したコストレポートの改善に、Amazon DynamoDB にタグを追加できるようになりました。詳細については、「DynamoDB でのリソースへのタグとラベルの追加」を参照してください。	2017 年 1 月 19 日

変更	説明	変更日
新しい DynamoDB `DescribeLimits` API	`DescribeLimits` API は、リージョンの AWS アカウント用に現在プロビジョンされた容量の上限を返します。この API はリージョン全体と、そこで作成した任意の DynamoDB テーブルの両方について値を返します。これにより、現在のアカウントレベルの制限を判断し、現在使用中のプロビジョニングされた容量と比較できるようになります。また、制限に達する前に引き上げを申請するために十分な時間が得られます。詳細については、Amazon DynamoDB API リファレンスの「Amazon DynamoDB のクォータ」、および「DescribeLimits」を参照してください。	2016 年 3 月 1 日

変更	説明	変更日
DynamoDB コンソールの更新とプライマリキー属性に関する新しい用語	DynamoDB マネジメントコンソールが、より直感的に、より使いやすくするために設計し直されました。この更新の一部であるプライマリキー属性の新しい用語について説明します。 • [Partition Key (パーティションキー)] – ハッシュ属性とも呼ばれます。 • [Sort Key (ソートキー)] – 範囲属性とも呼ばれます。 名前のみが変更され、機能は同じです。 テーブル、またはセカンダリインデックスを作成する際には、(パーティションキーのみの) シンプルなプライマリキーを選択することも、(パーティションキーおよびソートキーによる) 複合的なプライマリキーを選択することも可能です。これらの変更を反映するために、DynamoDB ドキュメントが更新されています。	2015 年 12 月 11 日

変更	説明	変更日
Titan 向け Amazon DynamoDB Storage Backend	Titan 向け Amazon DynamoDB Storage Backend は、Amazon DynamoDB 上に実装される Titan グラフデータベース用のストレージバックエンドです。Titan 向け Amazon DynamoDB Storage Backend を使用する場合には、Amazon の高い可用性を備えたデータセンター全体で実行される DynamoDB による保護が、データで活用できるようになります。プラグインは、Titan バージョン 0.4.4 (主に既存のアプリケーションに対応) と Titan バージョン 0.5.4 (新しいアプリケーションに推奨) で使用できます。このプラグインは、Titan 用の他のストレージバックエンドのように、Blueprints API および Gremlin シェルを含めて、Tinkerpop スタック (バージョン 2.4 および 2.5) をサポートします。	2015 年 8 月 20 日

変更	説明	変更日
DynamoDB Streams、クロスリージョンレプリケーション、および強力な整合性のある読み込みによるスキャニング	DynamoDB Streams は、DynamoDB テーブル内の項目レベルの変更に関するシーケンスを時間順にキャプチャし、その情報を最大 24 時間ログに保存します。アプリケーションは、このログにアクセスし、データ項目の変更前および変更後の内容をほぼリアルタイムで参照できます。詳細については、「DynamoDB Streams の変更データキャプチャ」および DynamoDB Streams API リファレンスを参照してください。 DynamoDB クロスリージョンレプリケーションは、異なる AWS リージョン間で、DynamoDB テーブルの完全なコピーをほぼリアルタイムで維持するための、クライアント側ソリューションです。クロスリージョンレプリケーションを使用することで、DynamoDB テーブルをバックアップしたり、ユーザーが地理的に分散されたケースで、データへの低レイテンシーなアクセスを実現したりできます。 DynamoDB Scan オペレーションは、デフォルトで	2015 年 7 月 16 日

変更	説明	変更日
	結果整合性のある読み込みを使用します。代わりに、`ConsistentRead` パラメータを true に設定することによって、強い整合性のある読み込みを使用できます。詳細については、Amazon DynamoDB API リファレンスの「スキャンの読み込み整合性」および「Scan」を参照してください。	
AWS CloudTrailAmazon DynamoDB の サポート	DynamoDB が CloudTrail に統合されました。CloudTrail は、DynamoDB コンソールまたは DynamoDB API で行われた API 呼び出しをキャプチャーし、ログファイルに記録します。詳細については、AWS CloudTrail を使用して DynamoDB オペレーションをログに記録する および AWS CloudTrail ユーザーガイドを参照してください。	2015 年 5 月 28 日

変更	説明	変更日
クエリ式のサポートの向上	このリリースでは、KeyConditionExpression API に新しい Query パラメータが追加されました。Query は、プライマリキー値を使用してテーブルまたはインデックスから項目を読み込みます。KeyConditionExpression パラメータは、プライマリキーの名前、さらにキー値に適用される条件を特定する文字列です。Query は、式を満たすこれらの項目のみを取り出します。KeyConditionExpression の構文は、DynamoDB で使用する他のパラメータ表現に似ています。この構文により、表現内の名前や値に置換する変数を定義できます。詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。	2015 年 4 月 27 日

変更	説明	変更日
条件付き書き込みの新しい比較関数	DynamoDB では、ConditionExpression パラメータにより PutItem、UpdateItem 、または DeleteItem の成否が判断されます。項目は、条件が true に評価された場合にのみ書き込まれます。このリリースでは、attribute_type で使用するための 2 つの新しい機能、size と ConditionExpression が追加されました。これらの機能により、テーブルの属性のデータの型やサイズに基づいて条件付き書き込みを実行できます。詳細については、「DynamoDB 条件式 CLI の例」を参照してください。	2015 年 4 月 27 日

変更	説明	変更日
セカンダリインデックスのスキャン API	DynamoDB では、テーブル内のすべての項目は Scan オペレーションで読み込みまれます。これらの項目には、ユーザー定義のフィルター基準が適用され、そこで選択されたデータ項目がアプリケーションに返されます。この同じ機能は、セカンダリインデックスでも使用できるようになりました。ローカルセカンダリインデックスまたはグローバルセカンダリインデックスをスキャンするには、インデックス名と親テーブルの名前を指定します。デフォルトでは、インデックス Scan はインデックス内のすべてのデータを返します。フィルタ式を使用して、アプリケーションに返される結果を絞り込むことができます。詳細については、「DynamoDB でのテーブルのスキャン」を参照してください。	2015 年 2 月 10 日

変更	説明	変更日
グローバルセカンダリインデックスのオンラインオペレーション	オンラインインデックスでは、既存のテーブルにグローバルセカンダリインデックスを追加または削除することができます。オンラインインデックスを使用すれば、テーブルの作成時にテーブルのすべてのインデックスを定義する必要はありません。代わりに、いつでも新しいインデックスを追加できます。同様に、インデックスが不要であると判断した場合は、いつでもそのインデックスを削除できます。オンラインインデックスオペレーションはノンブロッキングであるため、インデックスの追加または削除中に、テーブルは読み取りまたは書き込みアクティビティに利用できる状態になります。詳細については、「DynamoDB のグローバルセカンダリインデックスの管理」を参照してください。	2015 年 1 月 27 日

変更	説明	変更日
JSON によるドキュメントモデルのサポート	DynamoDB は、すべてのドキュメントモデルをサポートしており、それを保存および利用することができます。新しいデータ型は、JSON 標準と完全互換で、ドキュメント要素を互いに入れ子にすることができます。ドキュメントパスの逆参照演算子を使用して、ドキュメント全体を取得しなくても個々の要素を読み書きすることができます。今回のリリースでは、データ項目の読み書き時に射影、条件、更新アクションを指定する新しい式パラメータも導入されました。JSON によるドキュメントモデルのサポートについて詳しくは、「データ型」と「DynamoDB での式の使用」を参照してください。	2014 年 10 月 7 日
柔軟なスケーリング	テーブルおよびグローバルセカンダリインデックスについては、テーブルごと、および、アカウントごとの制限内であれば、プロビジョニングされた読み書きスループットキャパシティを任意の量だけ増やすことができます。詳細については、「Amazon DynamoDB のクォータ」を参照してください。	2014 年 10 月 7 日

変更	説明	変更日
項目サイズの拡大	DynamoDB の最大項目サイズが 64 KB から 400 KB に拡大されました。詳細については、「Amazon DynamoDB のクォータ」を参照してください。	2014 年 10 月 7 日

変更	説明	変更日
強化された条件式	DynamoDB で条件式に使用できる演算子が強化され、条件付きの入力、更新、削除をより柔軟に行えるようになりました。新たに導入された演算子では、ある属性が存在するかしないかや、特定の値より大きいか小さいかを調べることができます。また、2 つ値の範囲内にあるかや、特定の文字で開始されるものであるかどうかを含め、その他多くの条件を判断することが可能です。DynamoDB では、オプションで OR 演算子も用意されており、多種の条件を評価するために使用できます デフォルトでは、式内の複数の条件は AND 演算が行われるため、それらのすべての条件が true になる場合にのみ、式は true と評価されます。代わりに OR を指定すると、少なくとも 1 つ の条件が true になる場合に、式は true と評価されます。詳細については、「DynamoDB での項目と属性の操作」を参照してください。	2014 年 4 月 24 日

変更	説明	変更日
クエリフィルター	DynamoDB の Query API で、新しい QueryFilter オプションがサポートされました。デフォルトでは、Query は、特定のパーティションキー値とオプションのソートキー条件 (省略可能) に一致する項目を検索します。Query フィルターはキー以外の属性 (非キー属性) に条件式を適用します。Query フィルターがある場合は、フィルター条件に一致しない項目が除外されてから、Query の結果がアプリケーションに返されます。詳細については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。	2014 年 4 月 24 日

変更	説明	変更日
AWS マネジメントコンソール を使用したデータのエクスポートとインポート	DynamoDB コンソールが改良され、DynamoDB テーブルのデータのエクスポートとインポートが簡略化されました。数回だけのクリックで AWS Data Pipeline を設定することで、ワークフローのオーケストレーションが可能です。また、Amazon Elastic MapReduce クラスターを設定することで、DynamoDB テーブルから Amazon S3 バケットへの (またはその逆方向での) データのコピーも実行できます。エクスポートまたはインポートを 1 回のみ実行するか、日次エクスポートジョブを設定することができます。さらに、クロスリージョンのエクスポートとインポートを実行して、DynamoDB データを 1 つの AWS リージョン内のテーブルから 別の AWS リージョン内のテーブルにコピーすることもできます。	2014 年 3 月 6 日

変更	説明	変更日
高レベル API のドキュメントの再構成	次の API に関する情報が見つけやすくなりました。 • Java: DynamoDBMappper • .NET ドキュメントモデルとオブジェクト永続性モデル この高レベル API は、「DynamoDB の上位レベルのプログラミングインターフェイス」に記載されています。	2014 年 1 月 20 日

変更	説明	変更日
グローバルセカンダリインデックス	DynamoDB に、グローバルセカンダリインデックスのサポートが追加されました。ローカルセカンダリインデックスと同様、テーブルからの代替キーを使用して、インデックスに対してクエリリクエストを発行することで、グローバルセカンダリインデックスを定義します。ローカルセカンダリインデックスとは異なり、グローバルセカンダリインデックスのパーティションキーは、テーブルのパーティションキーと同じである必要がありません。テーブルからの任意のスカラー属性にすることができます。ソートキーはオプションです。このキーも任意のスカラーテーブル属性にすることができます。グローバルセカンダリインデックスにも独自のプロビジョニングされたスループット設定があり、親テーブルの設定から独立しています。詳細については、「DynamoDB でのセカンダリインデックスを使用したデータアクセス性の向上」および「DynamoDB のグローバルセカンダリインデックスの使用」を参照してください。	2013 年 12 月 12 日

変更	説明	変更日
きめ細かなアクセス制御	DynamoDB に、きめ細かなアクセス制御のサポートが追加されました。この機能を使用するお客様は、DynamoDB テーブルまたはセカンダリインデックスに置かれた、個別の項目と属性にアクセスできるプリンシパル (ユーザー、グループ、またはロール) を指定できます。また、アプリケーションでウェブアイデンティティフェデレーションを利用することで、Facebook、Google、Login with Amazon などサードパーティのアイデンティティプロバイダーに、ユーザー認証に関するタスクを任せることができます。こうすることで、アプリケーション (モバイルアプリケーションを含む) は、非常に多くのユーザーを処理できると同時に、権限のないユーザーによる DynamoDB のデータ項目へのアクセスを防ぐことができます。詳細については、「詳細に設定されたアクセスコントロールのための IAM ポリシー条件の使用」を参照してください。	2013 年 10 月 29 日

変更	説明	変更日
4 KB の読み込みキャパシティーユニットのサイズ	読み込み用のキャパシティーユニットのサイズが 1 KB から 4 KB に増加しました。この拡張により、多くのアプリケーションで必要な、プロビジョニングされる読み込みキャパシティーユニットの数が少なくなります。たとえば、これより前のリリースでは、10 KB の項目を読み取る場合に 10 個の読み込みキャパシティーユニットが消費されていました。現在のリリースでは、同じ 10 KB の読み込みでも消費されるのは 3 ユニット（10 KB / 4 KB、次の 4 KB 境界に切り上げ）だけです。詳細については、「DynamoDB のスループットキャパシティ」を参照してください。	2013 年 5 月 14 日

変更	説明	変更日
並列スキャン	DynamoDB に、並列スキャンオペレーションのサポートが追加されました。各アプリケーションでは、1 つのテーブルを論理的なセグメントに分割して、すべてのセグメントを同時にスキャンすることができます。この機能によって、スキャンが完了するまでの時間が短縮され、テーブルにプロビジョニングされた読み込みキャパシティーが完全に利用されます。詳細については、「DynamoDB でのテーブルのスキャン」を参照してください。	2013 年 5 月 14 日
ローカルセカンダリインデックス	DynamoDB に、ローカルセカンダリインデックスのサポートが追加されました。非キー属性にソートキーインデックスを定義し、それらのインデックスをクエリリクエストで使用することができます。アプリケーションがローカルセカンダリインデックスを使って、複数のディメンション全体でデータの項目を効率的に取得することが可能になります。詳細については、「ローカルセカンダリインデックス」を参照してください。	2013 年 4 月 18 日

変更	説明	変更日
API の新バージョン	このリリースでは、新しいバージョンの API (2012-08-10) が DynamoDB に導入されました。以前のバージョンの API (2011-12-05) も、既存のアプリケーションとの下位互換性のために引き続きサポートされています。新しいアプリケーションでは、最新の API バージョン 2012-08-10 を使用してください。DynamoDB の新機能 (ローカルセカンダリインデックスなど) は以前の API バージョンにバックポートされないため、既存のアプリケーションでご使用の API バージョンを、2012-08-10 に移行することをお勧めします。API バージョン 2012-08-10 の詳細については、Amazon DynamoDB API リファレンスを参照してください。	2013 年 4 月 18 日

変更	説明	変更日
IAM ポリシー変数のサポート	IAM アクセスポリシー言語では、変数がサポートされるようになりました。ポリシーが評価されると、認証ユーザーのセッションから得たコンテキストベースの情報に基づく値にポリシー変数が置き換えられます。ポリシー変数を使用すれば、ポリシーのすべての構成要素を明示的にリストしなくても、汎用的なポリシーを定義できます。ポリシー変数の詳細については、AWS Identity and Access Management での IAM の使用ガイドで「ポリシー変数」を参照してください。 DynamoDB のポリシー変数の例については、「Amazon DynamoDB の Identity and Access Management」でご確認ください。	2013 年 4 月 4 日
AWS SDK for PHP バージョン 2 用に更新された PHP コード例	AWS SDK for PHP のバージョン 2 が使用可能となりました。Amazon DynamoDB デベロッパーガイドに示した PHP のコード例も、この新しい SDK に対応するように更新されています。SDK バージョン 2 の詳細については、「AWS SDK for PHP」を参照してください。	2013 年 1 月 23 日

変更	説明	変更日
新しいエンドポイント	DynamoDB が AWS GovCloud (米国西部) リージョンに拡張されました。現在のサービスエンドポイントとプロトコルのリストについては、リージョンとエンドポイントに関する章を参照してください。	2012 年 12 月 3 日
新しいエンドポイント	DynamoDB が南米 (サンパウロ) リージョンに拡張されました。現在サポートされているエンドポイントのリストについては、リージョンとエンドポイントに関する章を参照してください。	2012 年 12 月 3 日
新しいエンドポイント	DynamoDB がアジアパシフィック (シドニー) リージョンに拡張されました。現在サポートされているエンドポイントのリストについては、リージョンとエンドポイントに関する章を参照してください。	2012 年 11 月 13 日

変更	説明	変更日
DynamoDB で CRC32 チェックサムのサポートが実装され、強力な整合性のあるバッチ取得のサポート導入と、テーブルの同時更新に対する制限の廃止が行われました。	• HTTP ペイロードの CRC32 チェックサムが、DynamoDB により計算され、このチェックサムが新しいヘッダー `x-amz-crc32` に返されます。詳細については、「DynamoDB 低レベル API」を参照してください。 • デフォルトでは、`BatchGetItem` API が実行する読み込みオペレーションは、結果整合性がとられます。`ConsistentRead` の新しい `BatchGetItem` パラメータを使用すれば、リクエスト内の任意のテーブルに対して、強力な読み込み整合性を選択できます。詳細については、「説明」を参照してください。 • このリリースでは、多数のテーブルを同時に更新する場合の制限がいくつか廃止されています。一度に更新できるテーブルの合計数は 10 のままですが、これらのテーブルのステータスについては、`CREATING`、`UPDATING`、または `DELETING` を任意に組み合わせることが可能になりました。さらに、1 つ	2012 年 11 月 2 日

変更	説明	変更日
	のテーブルの ReadCapacityUnits または WriteCapacityUnits を増減できる最小値の制限もなくなりました。詳細について、Amazon DynamoDB のクォータ を参照してください。	
ベストプラクティスに関するドキュメント	Amazon DynamoDB デベロッパーガイド では、テーブルや項目の使用に関するベストプラクティスを紹介しています。加えて、クエリおよびスキャンオペレーションに関する推奨事項も示しています。	2012 年 9 月 28 日

変更	説明	変更日
バイナリデータ型のサポート	Number および String 型に加えて、DynamoDB で バイナリデータ型がサポートされるようになりました。 これより前のリリースでは、バイナリデータを DynamoDB に格納する際には、そのデータを文字列 (String) 形式に変換していました。クライアント側で変換作業が必要なだけでなく、変換によってデータ項目のサイズが大きくなり、より多くのストレージが必要になり、スループットキャパシティを追加でプロビジョニングしなければならない場合もありました。 バイナリタイプの属性を使用することで、圧縮データ、暗号化データ、イメージなど、任意のバイナリデータを格納できるようになりました。詳細については、「データ型」を参照してください。AWS SDK を使用してバイナリタイプのデータを処理する実例については、以下のセクションを参照してください。 • 例: AWS SDK for Java ドキュメント API を使用したバイナリタイプ属性の処理	2012 年 8 月 21 日

変更	説明	変更日
	• 例: AWS SDK for .NET 低レベル API を使用したバイナリタイプ属性の処理 AWS SDK でサポートされるバイナリデータのタイプを追加するには、最新の SDK をダウンロードする必要があります。既存のアプリケーションの更新が必要になる場合もあります。AWS SDK のダウンロードの詳細については、「.NET コード例」を参照してください。	
DynamoDB テーブルの項目を DynamoDB コンソールを使用して、更新およびコピーできるようになりました。	DynamoDB のユーザーは、テーブル項目の追加と削除に加えて、その更新とコピーも DynamoDB コンソールから行えるようになりました。この新機能によって、コンソールを使用した個々の項目の変更が容易になりました。	2012 年 8 月 14 日

変更	説明	変更日
DynamoDB で最小テーブルスループットの要件が引き下げられました。	DynamoDB の最小テーブルスループットの要件が引き下げられ、書き込みキャパシティーユニットおよび読み込みキャパシティーユニットでは、最低 1 個までがサポートされるようになりました。詳細については、Amazon DynamoDB デベロッパーガイドの「Amazon DynamoDB のクォータ」トピックを参照してください。	2012 年 8 月 9 日
Signature Version 4 のサポート	DynamoDB で、リクエストの認証用に署名バージョン 4 がサポートされるようになりました。	2012 年 7 月 5 日
DynamoDB コンソールでテーブルエクスプローラーが使用可能になりました。	DynamoDB コンソールで、テーブル内のデータの参照とクエリが可能な、テーブルエクスプローラーがサポートされるようになりました。新しい項目の挿入や既存の項目の削除も可能です。SampleData および コンソールを使用する セクションは、これらの機能を反映して更新されています。	2012 年 5 月 22 日

変更	説明	変更日
新しいエンドポイント	DynamoDB の可用性が、米国西部（北カリフォルニア）リージョン、米国西部（オレゴン）リージョン、アジアパシフィック（シンガポール）リージョンの新しいエンドポイントに拡張されました。 現在サポートされているエンドポイントのリストについては、リージョンとエンドポイントに関する章を参照してください。	2012 年 4 月 24 日
BatchWriteItem API のサポート	DynamoDB でバッチ書き込み API がサポートされ、1 回の API 呼び出しによって、1 つ以上のテーブルでの複数の項目の入力と削除ができるようにました。DynamoDB のバッチ書き込み API の詳細については、「BatchWriteItem」を参照してください。 AWS SDK による項目の操作とバッチ書き込み機能の詳細については、「DynamoDB での項目と属性の操作」および「.NET コード例」を参照してください。	2012 年 4 月 19 日
エラーコードの追加	詳細については、「DynamoDB でのエラー処理」を参照してください。	2012 年 4 月 5 日

変更	説明	変更日
新しいエンドポイント	DynamoDB がアジアパシフィック (東京) リージョンに拡張されました。現在サポートされているエンドポイントのリストについては、リージョンとエンドポイントに関する章を参照してください。	2012 年 2 月 29 日
`ReturnedItemCount` メトリクスの追加	新しいメトリクス `ReturnedItemCount` により、クエリからいくつかの項目を返せるようになり、同時に、DynamoDB に対するスキャンオペレーションを CloudWatch を通じてモニタリングできるようになりました。	2012 年 2 月 24 日
値を増加させる例を追加	DynamoDB で、既存の数値の増減が可能になりました。以下のドキュメントの「項目の更新」セクションでは、既存の値を増加させる例を示しています。 項目の操作: Java. 項目の操作: .NET.	2012 年 1 月 25 日
初回製品リリース	DynamoDB が、新たにベータリリースのサービスとして導入されました。	2012 年 1 月 18 日

DynamoDB のレガシー機能

以下のトピックは、DynamoDB が現在もサポートしているレガシー機能です。これらの機能については活発な開発は行われていません。

トピック

- グローバルテーブルバージョン 2017.11.29 (レガシー)
- 以前の低レベル DynamoDB API バージョン (2011-12-05)
- レガシー DynamoDB 条件パラメータ

グローバルテーブルバージョン 2017.11.29 (レガシー)

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

トピック

- グローバルテーブル: 仕組み
- グローバルテーブルを管理するためのベストプラクティスと要件
- グローバルテーブルの作成 (バージョン 2017.11.29)
- グローバルテーブルのモニタリング
- グローバルテーブルで IAM を使用します

グローバルテーブル: 仕組み

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

以下のセクションでは、Amazon DynamoDB におけるグローバルテーブルの概念および動作を理解することができます。

バージョン 2017.11.29 (レガシー) のグローバルテーブルの概念

グローバルテーブルは 1 つ以上のレプリカテーブルの集合体であり、すべて単一の AWS アカウントが所有します。

レプリカテーブル (または、略してレプリカ) は、グローバルテーブルの一環として機能する単一の DynamoDB テーブルです。各レプリカには、同じデータ項目のセットが保存されます。指定のグローバルテーブルは、AWS リージョンごとに 1 つのレプリカテーブルのみを持つことができます。

以下は、グローバルテーブルの作成方法における概念的な概要です。

1. AWS リージョンで、DynamoDB Streams を有効にしてある通常の DynamoDB テーブルを作成します。
2. データをレプリケートする他のすべてのリージョンに対して、ステップ 1 を繰り返します。
3. 作成したテーブルに基づいて DynamoDB グローバルテーブルを定義します。

AWS マネジメントコンソール はこれらのタスクを自動化するため、グローバルテーブルをより迅速かつ簡単に作成できます。詳細については、「」を参照してくださいグローバルテーブルの作成 (バージョン 2017.11.29)

結果の DynamoDB グローバルテーブルは、DynamoDB が単一の単位として扱う複数のレプリカテーブル (リージョンごとに 1 つ) で構成されます。すべてのレプリカは、同じテーブル名と同じプ

ライマリキーのスキーマを持っています。アプリケーションが 1 つのリージョンのレプリカテーブルにデータを書き込むと、DynamoDB はその書き込みを他の AWS リージョンの他のレプリカテーブルに自動的に伝搬します。

> **Important**
>
> テーブルデータの同期を維持するために、グローバルテーブルでは、すべての項目に対して次の属性を自動的に作成します。
>
> - `aws:rep:deleting`
> - `aws:rep:updatetime`
> - `aws:rep:updateregion`
>
> これらの属性を変更したり、同じ名前の属性を作成したりしないでください。

レプリカテーブルをグローバルテーブルに追加して、追加のリージョンで使用できるようにすることができます。(これを行うには、グローバルテーブルが空である必要があります。つまり、どのレプリカテーブルにもデータを含めることはできません。)

グローバルテーブルからレプリカテーブルを削除することもできます。これを行うと、テーブルはグローバルテーブルから完全に関連付けが解除されます。この新しく独立したテーブルは、グローバルテーブルと相互作用しなくなり、データはグローバルテーブルとの間で伝播されなくなります。

> **Warning**
>
> レプリカの削除はアトミックプロセスではないことに注意してください。動作と既知の状態に確実に一貫させるために、削除されるレプリカからアプリケーションの書き込みトラフィックを事前に切り離しておくことをお勧めします。削除後、すべてのレプリカリージョンのエンドポイントでレプリカが関連付け解除されたものとして表示されるのを待ってから、レプリカへの以降の書き込みを独自の分離リージョンテーブルとして実行してください。

一般的なタスク

グローバルテーブルの一般的なタスクは以下のように機能します。

グローバルテーブルのレプリカテーブルは、通常のテーブルと同じように削除できます。これにより、そのリージョンへのレプリケーションが停止し、そのリージョンに保存されているテーブルコピーが削除されます。レプリケーションを切断して、テーブルのコピーを独立したエンティティとすることはできません。

> **Note**
>
> ソーステーブルは、新しいリージョンの作成に使用されてから 24 時間以上経たないと削除できません。削除を試みるのが早すぎると、エラーが発生します。

アプリケーションが異なるリージョンにある同一項目をほぼ同時に更新すると、競合が発生する可能性があります。結果整合性を確保するために、DynamoDB グローバルテーブルでは同時更新間の調整のために、「最新書き込み優先」方法が使用されます。すべてのレプリカが最新の更新に合意し、すべてのレプリカが同一のデータを持つ状態に収束します。

> **Note**
>
> 競合を回避する方法は、以下を含めていくつかあります。
>
> - IAM ポリシーを使用して、1 つのリージョンのテーブルへの書き込みのみを許可する。
> - IAM ポリシーを使用してユーザーを 1 つのリージョンのみにルーティングし、別のリージョンはアイドル状態のスタンバイのままにするか、奇数のユーザーを 1 つのリージョンに、偶数のユーザーを別のリージョンにルーティングする。
> - Bookmark = Bookmark + 1 のような非冪等の更新は避け、Bookmark=25 のような静的な更新を優先する。

グローバルテーブルのモニタリング

CloudWatch を使用して、`ReplicationLatency` メトリクスをモニタリングできます。このメトリクスは、更新された項目が 1 つのレプリカテーブルの DynamoDB ストリームに表示されてから、その項目がグローバルテーブルの別のレプリカに表示されるまでの経過時間を追跡します。`ReplicationLatency` はミリ秒単位で表され、すべての送信元リージョンと送信先リージョンのペアに対して出力されます。これは、グローバルテーブル v2 が提供する唯一の CloudWatch メトリクスです。

表示されるレイテンシーは、選択したリージョン間の距離やその他の変動要素によって異なります。リージョンのレイテンシーが 0.5 ~ 2.5 秒の範囲で発生するのは、同じ地域内でもよくあることです。

有効期限 (TTL)

Time To Live (TTL) を使用して、項目の有効期限を示す値を含む属性名を指定できます。この値は、Unix エポックの開始からの秒数として指定されます。

グローバルテーブルのレガシーバージョンでは、TTL 削除は他のレプリカに自動的に複製されません。TTL ルールによって項目が削除された場合、削除は書き込みユニットを消費することなく実行されます。

ソーステーブルとターゲットテーブルのプロビジョニングされた書き込みキャパシティーが非常に少ない場合、TTL 削除には書き込みキャパシティーが必要となるため、スロットリングが発生する可能性があります。

グローバルテーブルでのストリームとトランザクション

各グローバルテーブルは、書き込みの開始点に関係なく、すべての書き込みに基づいて独立したストリームを生成します。この DynamoDB ストリームを 1 つのリージョンで使用するか、すべてのリージョンで個別に使用するかを選択できます。

レプリケートされた書き込みではなくローカル書き込みを処理する場合は、各項目に独自のリージョン属性を追加できます。次に、Lambda イベントフィルターを使用して、ローカルリージョンへの書き込みに対して Lambda のみを呼び出すことができます。

トランザクションオペレーションは、書き込みが最初に行われたリージョン内でのみ、ACID (不可分性、一貫性、分離性、耐久性) を保証します。グローバルテーブルのリージョン間では、トランザクションはサポートされていません。

例えば、米国東部 (オハイオ) および米国西部 (オレゴン) リージョンにレプリカを持つグローバルテーブルがあり、米国東部 (オハイオ) リージョンで TransactWriteItems オペレーションを実行する場合、変更がレプリケートされると米国西部 (オレゴン) ではトランザクションが部分的に完了される場合があります。変更は、ソースリージョンでコミットされると、他のリージョンにのみレプリケートされます。

> **Note**
>
> - グローバルテーブルは DynamoDB を直接更新することで、DynamoDB Accelerator を「書き込み迂回」します。その結果、DAX は古いデータを保持していることを認識しません。DAX キャッシュは、キャッシュの TTL が期限切れになったときにのみ更新されます。
> - グローバルテーブルのタグは自動的には伝播されません。

読み取りおよび書き込みスループット

グローバルテーブルは、次の方法で読み取りと書き込みのスループットを管理します。

- 書き込みキャパシティーは、リージョン間のすべてのテーブルインスタンスで同じでなければなりません。
- バージョン 2019.11.21 (現行) では、テーブルが、自動スケーリングをサポートするように設定されているか、オンデマンドモードになっている場合、書き込みキャパシティは自動的に同期されます。各リージョンでプロビジョニングされている書き込みキャパシティの現在の量は、同期された自動スケーリング設定の範囲内で個別に増減します。テーブルをオンデマンドモードにすると、そのモードは他のレプリカと同期します。
- 読み取りキャパシティーはリージョンによって異なる場合があります。これは、読み取りキャパシティーが等しくない場合があるためです。グローバルレプリカをテーブルに追加すると、ソースリージョンのキャパシティーが反映されます。作成後、1 つのレプリカの読み込みキャパシティを調整できますが、この新しい設定は他方には転送されません。

整合性と競合の解決

レプリカテーブル内の項目に加えられた変更は、同じグローバルテーブル内の他のすべてのレプリカにレプリケートされます。グローバルテーブルでは、新しく書き込まれた項目は通常、数秒以内にすべてのレプリカテーブルに伝播されます。

グローバルテーブルでは、各レプリカテーブルには同じデータ項目のセットが保存されます。DynamoDB では、一部の項目のみの部分的なレプリケーションはサポートされていません。

アプリケーションは、任意のレプリカテーブルとの間でデータを読み込みおよび書き込みできます。DynamoDB は、リージョン間での結果整合性にある読み込みに対応していますが、リージョン間での強力な整合性のある読み込みには対応していません。アプリケーションが結果整合性のある読

み込みのみを使用し、1 つの AWS リージョンに対して読み込みのみを発行する場合、変更を加えずに動作します。ただし、アプリケーションで強力な整合性のある読み込みが必要な場合は、同じリージョンですべての強力な整合性のある読み込みと書き込みを実行する必要があります。そうではなく、あるリージョンに書き込んで別のリージョンから読み込む場合、読み込みのレスポンスには、他のリージョンで最近完了した書き込みの結果を反映していない古いデータが含まれる可能性があります。

アプリケーションが異なるリージョンにある同一項目をほぼ同時に更新すると、競合が発生する可能性があります。結果整合性を確保するために、DynamoDB グローバルテーブルでは最終書き込み者優先を使用して、同時更新間の調整を行い、DynamoDB は最終書き込み者を判断するために最善を尽くします。この競合解決メカニズムでは、すべてのレプリカが最新の更新に合意し、すべてのレプリカが同一のデータを持つ状態に収束します。

可用性と耐久性

単一の AWS リージョンが分離または低下した場合、アプリケーションは別のリージョンにリダイレクトし、別のレプリカテーブルに対して読み込みと書き込みを実行できます。カスタムビジネスロジックを適用して、リクエストを他のリージョンにリダイレクトするタイミングを決定できます。

リージョンが分離または縮退した場合、DynamoDB は、実行されたが、まだすべてのレプリカテーブルに反映されていない書き込みを追跡します。リージョンがオンラインに戻ると、DynamoDB はそのリージョンから他のリージョンのレプリカテーブルへの保留中の書き込みの伝播を再開します。また、他のレプリカテーブルから現在オンラインに戻っているリージョンへの書き込みの伝播も再開します。リージョンの分離期間がどれほど長くても、それまでに成功した書き込みはすべて最終的に反映されます。

グローバルテーブルを管理するためのベストプラクティスと要件

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

Amazon DynamoDB グローバルテーブルを使用すると、テーブルデータを AWS リージョン間でレプリケートできます。データが適切にレプリケートされるように、グローバルテーブル内のレプリカテーブルとセカンダリインデックスにも、書き込みキャパシティーが同じように設定されていることが重要です。

トピック

- グローバルテーブルのバージョン
- 新しいレプリカテーブルを追加するための要件
- キャパシティを管理するためのベストプラクティスと要件

グローバルテーブルのバージョン

DynamoDB グローバルテーブルには、グローバルテーブルバージョン 2019.11.21 (現行) と グローバルテーブルバージョン 2017.11.29 (レガシー) の 2 つのバージョンがあります。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。

ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「グローバルテーブルのアップグレード」を参照してください。

新しいレプリカテーブルを追加するための要件

新しいレプリカテーブルをグローバルテーブルに追加するには、以下のすべての条件が満たされている必要があります。

- その他すべてのレプリカと同じパーティションキーがテーブルに含まれている。
- テーブルは、同じ指定した書き込みキャパシティー管理設定である必要があります。
- テーブルの名前が、その他すべてのレプリカと同じ名前である。
- 項目の新しいイメージと古いイメージの両方を含むストリーミングで、テーブルの DynamoDB Streams が有効になっている。
- このグローバルテーブルの新規または既存のレプリカテーブルにはデータを含むことはできません。

グローバルセカンダリインデックスが指定されている場合は、次の条件も満たす必要があります。

- グローバルセカンダリインデックスにも同じ名前が設定されている。

- グローバルセカンダリインデックスのパーティションキーとソートキー (存在する場合) にも同じ名前が設定されている。

> **Important**
>
> 書き込みキャパシティーの設定は、すべてのグローバルテーブルのレプリカテーブルおよび一致するセカンダリインデックスに対して一貫性があるように設定する必要があります。グローバルテーブルの書き込み容量を更新するには、DynamoDB コンソールまたは UpdateGlobalTableSettings API UpdateGlobalTableSettingsオペレーションを使用することを強くお勧めします。[585] は、自動的にグローバルテーブル内のすべてのレプリカテーブルおよび一致するセカンダリインデックスに書き込み容量の設定の変更を適用します。UpdateTable、RegisterScalableTarget、あるいは PutScalingPolicy オペレーションを使用する場合には、各レプリカテーブルおよび一致するセカンダリインデックスに個別に変更を適用する必要があります。詳細については、「Amazon DynamoDB API リファレンス」の「UpdateGlobalTableSettings」を参照してください。
> プロビジョニングされた書き込みキャパシティー設定を管理するために、Auto Scaling を有効にすることが強く推奨されます。書き込みキャパシティー設定を手動で管理する場合は、すべてのレプリカテーブルと同等のレプリケートされた書き込みキャパシティーユニットをプロビジョニングする必要があります。また、グローバルテーブル全体で一致するセカンダリインデックスには、レプリケートされた同等の書き込みキャパシティーユニットをプロビジョニングします。
> また、適切な AWS Identity and Access Management (IAM) 権限も必要です。詳細については、「グローバルテーブルで IAM を使用します」を参照してください。

キャパシティを管理するためのベストプラクティスと要件

DynamoDB でレプリカテーブルの容量設定を管理するときは、以下の点を考慮に入れてください。

DynamoDB Auto Scaling の使用

プロビジョンされたモードを使用するレプリカテーブルのスループット容量設定を管理するには、DynamoDB Auto Scaling の使用をお勧めします。DynamoDB Auto Scaling は各レプリカテーブルの読み込み容量単位 (RCU) および書き込み容量単位 (WCU) を、実際のアプリケーションワークロードに基づいて、自動的に調整します。詳細については、「DynamoDB Auto Scaling によるスループットキャパシティの自動管理」を参照してください。

AWS マネジメントコンソール を使用してレプリカテーブルを作成すると、Auto Scaling が各レプリカテーブルに対してデフォルトで有効になります。この際、読み込みキャパシティーユニットおよび書き込みキャパシティーユニットを管理するデフォルトの Auto Scaling 設定が使用されます。

DynamoDB コンソールを介して、あるいは `UpdateGlobalTableSettings` コールを使用して、レプリカテーブルあるいはセカンダリインデックスの Auto Scaling 設定に行う変更は、グローバルテーブルのすべてのレプリカテーブルおよび一致するセカンダリインデックスに対して自動的に適用されます。これらの変更により、すべての既存の Auto Scaling 設定が上書きされます。これによって、プロビジョニングされた書き込みキャパシティーは、グローバルテーブルのすべてのレプリカテーブルおよび一致するセカンダリインデックスを通して一貫性があるようになります。`UpdateTable`、`RegisterScalableTarget`、あるいは `PutScalingPolicy` 呼び出しを使用する場合には、各レプリカテーブルおよび一致するセカンダリインデックスに個別に変更を適用する必要があります。

> **Note**
>
> Auto Scaling がアプリケーションの容量の変更を満たさない場合 (予測不可能なワークロード) や、その設定を行わない場合 (最小、最大、または使用率しきい値のターゲット設定)、オンデマンドモードを使用してグローバルテーブルの容量を管理できます。詳細については、「オンデマンドモード」を参照してください。
> グローバルテーブルでオンデマンドモードを有効にした場合、レプリケートされた書き込みリクエストユニット (rWCU) の消費は、rWCU のプロビジョニング方法と整合します。たとえば、2 つの追加リージョンにレプリケートされるローカルテーブルに対して書き込みを 10 回実行した場合、書き込みリクエストユニットが 60 個消費されます (10 + 10 + 10 = 30、30 x 2 = 60)。消費される 60 個の書き込みリクエストユニットには、`aws:rep:deleting`、`aws:rep:updatetime`、および `aws:rep:updateregion` 属性を更新するためにグローバルテーブルバージョン 2017.11.29 (レガシー) で消費される追加の書き込みが含まれます。

手動によるキャパシティ管理

DynamoDB Auto Scaling を使用しない場合は、各レプリカテーブルおよびセカンダリインデックスの読み込み容量および書き込み容量を手動で設定する必要があります。

すべてのレプリカテーブルでレプリケートされたプロビジョンド書き込みキャパシティーユニット (rWCU) は、全リージョンにおけるアプリケーションの書き込みに必要な rWCU 合計数の 2 倍に設定する必要があります。これにより、ローカルリージョンで発生するアプリケーションの書き込み

と、他のリージョンからのレプリケートされたアプリケーションの書き込みに対応できます。たとえば、オハイオ州のレプリカテーブルに 1 秒間に 5 回の書き込みを、バージニア北部のレプリカテーブルに 1 秒間に 5 回の書き込みを行うとします。この場合、各レプリカテーブルに 20 個の rWCU をプロビジョニングする必要があります (5 + 5 = 10、10×2 = 20)。

グローバルテーブルの書き込み容量を更新するには、DynamoDB コンソールまたは UpdateGlobalTableSettings API UpdateGlobalTableSettingsオペレーションを使用することを強くお勧めします。[585] は、自動的にグローバルテーブル内のすべてのレプリカテーブルおよび一致するセカンダリインデックスに書き込み容量の設定の変更を適用します。UpdateTable、RegisterScalableTarget、あるいは PutScalingPolicy オペレーションを使用する場合には、各レプリカテーブルおよび一致するセカンダリインデックスに個別に変更を適用する必要があります。詳細については、Amazon DynamoDB API リファレンスを参照してください。

> **Note**
>
> DynamoDB のグローバルテーブルの設定 (UpdateGlobalTableSettings) を更新するには、dynamodb:UpdateGlobalTable、dynamodb:DescribeLimits、application-autoscaling:DeleteScalingPolicy および application-autoscaling:DeregisterScalableTarget の許可が必要です。詳細については、「グローバルテーブルで IAM を使用します」を参照してください。

グローバルテーブルの作成 (バージョン 2017.11.29)

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

このセクションでは、Amazon DynamoDB コンソールまたは AWS Command Line Interface (AWS CLI) を使用してグローバルテーブルを作成する方法について説明します。

トピック

- グローバルテーブルの作成 (コンソール)
- グローバルテーブル (AWS CLI) の作成

グローバルテーブルの作成 (コンソール)

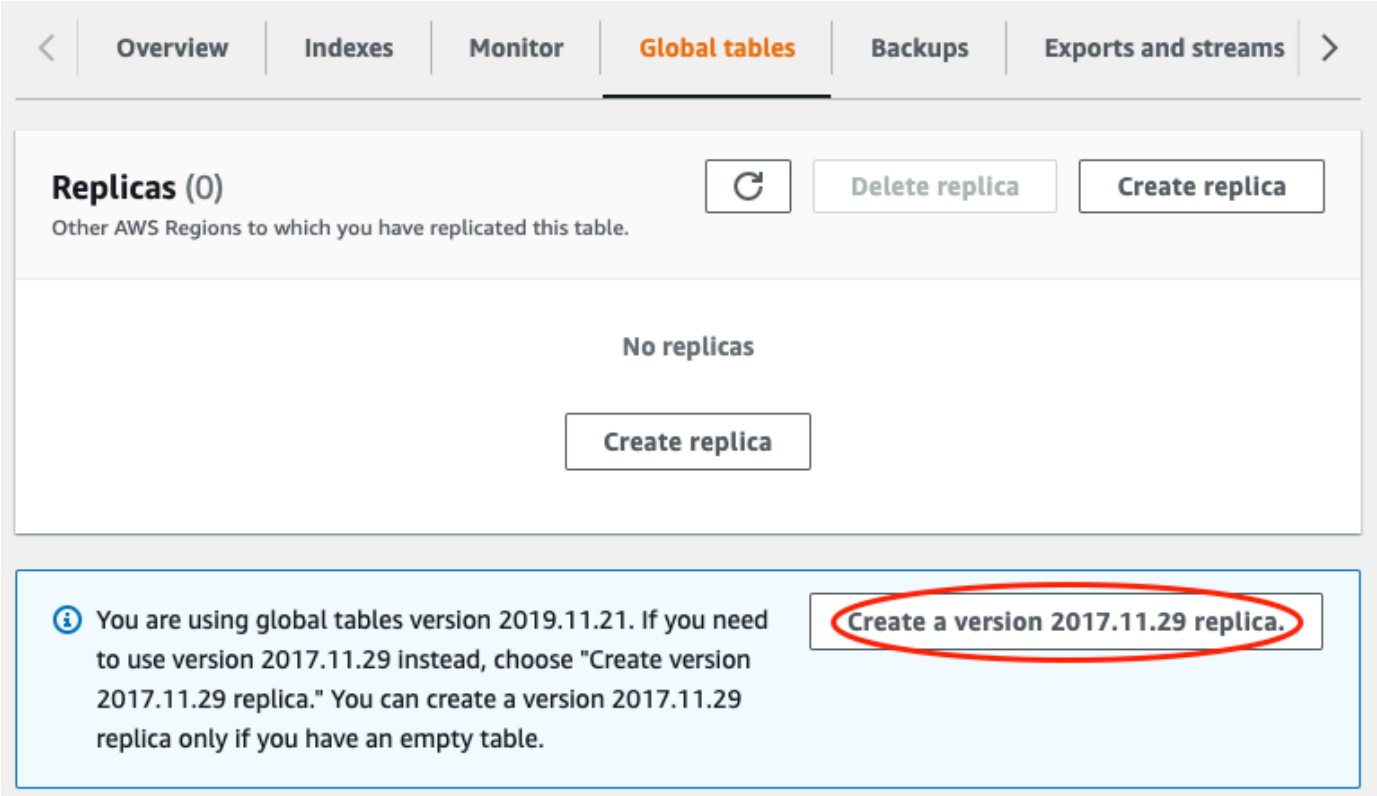
コンソールを使用してグローバルテーブルを作成するには、次の手順に従います。以下の例では、レプリカテーブルを持つグローバルテーブルを米国およびヨーロッパに作成します。

1. DynamoDB コンソール (https://console.aws.amazon.com/dynamodb/home) を開きます。この例では、us-east-2 (米国東部オハイオ) リージョンを選択します。
2. コンソールの左側のナビゲーションペインで、[テーブル] を選択します。
3. [テーブルの作成] を選択します。

   [テーブル名] に「**Music**」と入力します。

   [プライマリキー] に **Artist** と入力します。[Add sort key (ソートキーの追加)] を選択し、**SongTitle** と入力します (**Artist** と **SongTitle** はいずれも文字列で入力します)。

   テーブルを作成するには、[Create (作成)] を選択します。このテーブルは、新しいグローバルテーブルの最初のレプリカテーブルとして機能します。これは、後で追加する他のレプリカテーブルのプロトタイプです。
4. [グローバルテーブル] タブを選択後、[バージョン 2017.11.29 (レガシー) レプリカを作成] を選択します。



5. [利用できるレプリケーションリージョン] ドロップダウンで、米国西部 (オレゴン) を選択します。

   コンソールは、選択したリージョンに同一名のテーブルが存在しないことを確認します。同一名のテーブルが存在する場合は、そのリージョンで新しいレプリカテーブルを作成する前に既存のテーブルを削除する必要があります。

6. [レプリカを作成] を選択します。これにより、米国西部 (オレゴン) でテーブル作成プロセスが開始されます。

   選択したテーブル (およびその他すべてのレプリカテーブル) の [グローバルテーブル] タブに、そのテーブルが複数のリージョンでレプリケートされたことが示されます。

7. ここで、別のリージョンを追加して、グローバルテーブルがレプリケートされ、米国およびヨーロッパに同期されるようにします。これを行うには、ステップ 5 を繰り返しますが、今回は米国西部 (オレゴン) の代わりに欧州 (フランクフルト) を指定します。

8. 引き続き、米国東部 (オハイオ)リージョンで AWS マネジメントコンソール を使用する必要があります。左のナビゲーションメニューで [項目] を選択し、[音楽] テーブルを選択してから、[項目を作成] を選択します。

   a. [Artist] に **`item_1`** と入力します。

b. [SongTitle] に **Song Value 1** と入力します。

c. 項目を書き込むには、[項目を作成] を選択します。

9. 間もなくすると、この項目は、グローバルテーブルの 3 つのすべてのリージョンにレプリケートされます。これを確認するには、コンソールの右上隅にあるリージョンセレクターで、[Europe (Frankfurt) (欧州 (フランクフルト))] を選択します。欧州 (フランクフルト) の Music テーブルに新しい項目が追加されます。
10. ステップ 9 を繰り返し、米国西部 (オレゴン) を選択して、そのリージョンでのレプリケーションを確認します。

グローバルテーブル (AWS CLI) の作成

Music を使用してグローバルテーブル AWS CLI を作成するには、次の手順に従います。以下の例では、米国およびヨーロッパのレプリカテーブルを使用して、グローバルテーブルを作成します。

1. DynamoDB Streams を有効にして (Music)、米国東部 (オハイオ) で新規テーブル (NEW_AND_OLD_IMAGES) DynamoDB Streams を作成します。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2
```

2. 米国東部 (バージニア北部) に同じ Music テーブルを作成します。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
```

```
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-1
```

3. us-east-2 リージョンと us-east-1 リージョンのレプリカテーブルで構成されるグローバルテーブル (Music) を作成します。

```
aws dynamodb create-global-table \
    --global-table-name Music \
    --replication-group RegionName=us-east-2 RegionName=us-east-1 \
    --region us-east-2
```

> **Note**
>
> グローバルテーブル名 (Music) は、各レプリカテーブルの名前 (Music) と一致する必要があります。詳細については、「グローバルテーブルのベストプラクティス」を参照してください。

4. ステップ 1 とステップ 2 で作成した設定と同じ設定で、欧州 (アイルランド) に別のテーブルを作成します。

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region eu-west-1
```

このステップを実行した後、新しいテーブルを Music グローバルテーブルに追加します。

```
aws dynamodb update-global-table \
    --global-table-name Music \
    --replica-updates 'Create={RegionName=eu-west-1}' \
```

```
    --region us-east-2
```

5. レプリケーションが機能していることを確認するには、新しい項目を米国東部 (オハイオ) の Music テーブルに追加します。

```
aws dynamodb put-item \
    --table-name Music \
    --item '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

6. 数秒間待ってから、項目が米国東部 (バージニア北部) および欧州 (アイルランド) に正常にレプリケートされたかどうかを確認します。

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1
```

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"},"SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

グローバルテーブルのモニタリング

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

Amazon CloudWatch を使用して、グローバルテーブルの動作とパフォーマンスをモニタリングできます。Amazon DynamoDB は、グローバルテーブルの各レプリカの ReplicationLatency および PendingReplicationCount メトリクスを公開します。

- **ReplicationLatency** - 更新された項目が 1 つのレプリカテーブルの DynamoDB ストリームに表示されてから、その項目がグローバルテーブルの別のレプリカに表示されるまでの経過時間。ReplicationLatency はミリ秒単位で表し、すべての送信元リージョンと送信先リージョンのペアに対して出力されます。

  通常オペレーション中、ReplicationLatency は完全に一定にする必要があります。ReplicationLatency で評価された値は、1 つのレプリカを更新しても、他のレプリカテーブルにタイムリーに伝達されません。これが原因で、アップデートは一貫して受け取ることができないため、時間の経過とともに、他のレプリカテーブルは遅延します。この場合、各レプリカテーブルの読み込みキャパシティーユニット (RCU) と書き込みキャパシティーユニット (WCU) が同一であることを確認する必要があります。また、WCU 設定を選択する場合は、「グローバルテーブルのバージョン」の推奨事項に従います。

  ReplicationLatency は、AWS リージョンのパフォーマンスが低下し、そのリージョンにレプリカテーブルが含まれる場合は増加することがあります。この場合、アプリケーションの読み込みおよび書き込みアクティビティを別の AWS リージョンに一時的にリダイレクトすることができます。

- **PendingReplicationCount** — 1 つのレプリカテーブルに書き込まれているが、グローバルテーブル内の別のレプリカにはまだ書き込まれていない項目の更新数。PendingReplicationCount は項目数で表され、すべての送信元と送信先のリージョンのペアに対して発行されます。

  通常のオペレーションでは、PendingReplicationCount は非常に低いです。PendingReplicationCount が長期間増加する場合は、レプリカテーブルのプロビジョンされた書き込み容量設定が現在のワークロードに対して十分であるかどうかを調査します。

  PendingReplicationCount は、AWS リージョンのパフォーマンスが低下し、そのリージョンにレプリカテーブルが含まれる場合は増加することがあります。この場合、アプリケーションの読み込みおよび書き込みアクティビティを別の AWS リージョンに一時的にリダイレクトすることができます。

詳細については、「DynamoDB のメトリクスとディメンション」を参照してください。

グローバルテーブルで IAM を使用します

> **Important**
>
> このドキュメントはグローバルテーブルのバージョン 2017.11.29 (レガシー) を対象としています。新しいグローバルテーブルでは使用しないでください。可能な限り、グローバルテーブルバージョン 2019.11.21 (現行) を使用してください。2017.11.29 (レガシー) よりも柔軟性と効率が高く、消費する書き込みキャパシティが少ないためです。
> ご使用のバージョンを確認するには、「グローバルテーブルのバージョンを確認する」を参照してください。既存のグローバルテーブルを 2017.11.29 (レガシー) からバージョン 2019.11.21 (現行) に更新する方法については、「DynamoDB グローバルテーブルのバージョン」を参照してください。

グローバルテーブルを初めて作成する場合は、Amazon DynamoDB によって、AWS Identity and Access Management (IAM) サービスにリンクされたロールが自動的に作成されます。このロールの名前は です。このロールを使用して、お客様に代わって DynamoDB でグローバルテーブルのクロスリージョンレプリケーションを管理できます。このサービスにリンクされたロールは削除しないでください。削除すると、グローバルテーブルはすべて機能しなくなります。

サービスリンクロールの詳細については、「IAM ユーザーガイド」の「サービスリンクロールの使用」を参照してください。

DynamoDB でグローバルテーブルを作成および管理するには、次のそれぞれにアクセスするための `dynamodb:CreateGlobalTable` 許可が必要です。

- 追加するレプリカテーブル。
- すでにグローバルテーブルの一部になっている既存の各レプリカ。
- グローバルテーブル自体。

DynamoDB のグローバルテーブルの設定 (`UpdateGlobalTableSettings`) を更新するには、`dynamodb:UpdateGlobalTable`、`dynamodb:DescribeLimits`、`application-autoscaling:DeleteScalingPolicy` および `application-autoscaling:DeregisterScalableTarget` の許可が必要です。

`application-autoscaling:DeleteScalingPolicy` および `application-autoscaling:DeregisterScalableTarget` 許可は、既存のスケーリングポリシーを更新すると

きに必要です。これは、新しいポリシーをテーブルまたはセカンダリインデックスに添付する前に、グローバルテーブルサービスが古いスケーリングポリシーを削除できるようにするためです。

IAM ポリシーを使用して 1 つのレプリカテーブルへのアクセスを管理する場合は、そのグローバルテーブル内の他のすべてのレプリカに同じポリシーを適用する必要があります。この方法により、すべてのレプリカテーブルで一貫した許可モデルを維持できます。

グローバルテーブル内のすべてのレプリカで同じ IAM ポリシーを使用することにより、グローバルテーブルデータに対する意図しない読み込みおよび書き込みアクセスを許可しないようにすることもできます。たとえば、グローバルテーブル内の 1 つのレプリカにのみアクセスできるユーザーについて考えてみます。そのユーザーがこのレプリカに書き込むことができる場合、DynamoDB はその書き込みを他のすべてのレプリカテーブルに伝播します。実際には、ユーザーは、グローバルテーブル内の他のすべてのレプリカに (間接的に) 書き込むことができます。このシナリオは、すべてのレプリカテーブルで一貫した IAM ポリシーを使用することで回避できます。

例: CreateGlobalTable アクションを許可します

グローバルテーブルにレプリカを追加するには、その前にグローバルテーブルとその各レプリカテーブルに対して `dynamodb:CreateGlobalTable` 許可を付与します。

以下の IAM ポリシーは、`CreateGlobalTable` アクションをすべてのテーブルに適用する許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["dynamodb:CreateGlobalTable"],
            "Resource": "*"
        }
    ]
}
```

例: UpdateGlobalTable、DescribeLimits、application-autoscaling:DeleteScalingPolicy、および application-autoscaling:DeregisterScalableTarget アクションを許可します

DynamoDB のグローバルテーブルの設定 (UpdateGlobalTableSettings) を更新するには、dynamodb:UpdateGlobalTable、dynamodb:DescribeLimits、application-autoscaling:DeleteScalingPolicy および application-autoscaling:DeregisterScalableTarget の許可が必要です。

以下の IAM ポリシーは、UpdateGlobalTableSettings アクションをすべてのテーブルに適用する許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateGlobalTable",
                "dynamodb:DescribeLimits",
                "application-autoscaling:DeleteScalingPolicy",
                "application-autoscaling:DeregisterScalableTarget"
            ],
            "Resource": "*"
        }
    ]
}
```

例: 特定のリージョンでのみレプリカが許可されている特定のグローバルテーブル名に対して CreateGlobalTable アクションを許可します

以下の IAM ポリシーは、CreateGlobalTable アクションが 2 つのリージョンにレプリカを持つ Customers という名前のグローバルテーブルを作成できるようにする許可を付与します。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb:CreateGlobalTable",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Customers",
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers"
            ]
        }
    ]
}
```

以前の低レベル DynamoDB API バージョン (2011-12-05)

このセクションでは、以前の DynamoDB 低レベル API (2011-12-05) で利用可能なオペレーションについて説明します。このバージョンの低レベル API は、既存のアプリケーションとの下位互換性のために維持されています。

新しいアプリケーションでは、最新の API バージョン (2012-08-10) を使用してください。詳細については、「DynamoDB API リファレンス」を参照してください。

> **Note**
>
> DynamoDB の新機能は以前の API バージョンにバックポートされないため、既存のアプリケーションを新しい API バージョン (2012-08-10) に移行することをお勧めします。

トピック

- BatchGetItem
- BatchWriteItem
- CreateTable
- DeleteItem

- DeleteTable
- DescribeTables
- GetItem
- ListTables
- PutItem
- Query
- Scan
- UpdateItem
- UpdateTable

BatchGetItem

> **Important**
>
> **`##########API ##### 2011-12-05 ########################################## #######`**
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

`BatchGetItem` オペレーションは、プライマリキーを使用して、複数のテーブルの複数の項目の属性を返します。1 回のオペレーションで取得できる項目の最大数は 100 です。また、取得される項目の数は、1 MB のサイズ制限によって制限されます。テーブルのプロビジョニングされたスループットを超えたことや内部処理が失敗したことが原因でレスポンスサイズの制限を超えた場合または結果の一部だけが返された場合、DynamoDB は `UnprocessedKeys` 値を返すので、取得する次の項目からオペレーションを再試行できます。DynamoDB は、この制限を適用するために、ページごとに返される項目の数を自動的に調整します。例えば、100 個の項目を取得するようにリクエストしても、個々の項目のサイズが 50 KB の場合、返されるのは 20 個の項目だけなので、適切な `UnprocessedKeys` 値を設定して結果の次のページを取得できます。必要に応じて、アプリケーションに独自のロジックを組み込んで、結果のページを 1 つのセットにまとめることができます。

リクエストに含まれる各テーブルのプロビジョニングされたスループットが
不十分だったことが原因で処理される項目がなかった場合、DynamoDB は
`ProvisionedThroughputExceededException` エラーを返します。

> **Note**
>
> デフォルトでは、BatchGetItem は、リクエスト内の各テーブルに対して結果整合性のある読み込みを実行します。整合性のある読み込みが必要な場合は、テーブルごとに ConsistentRead パラメータを true に設定できます。
> BatchGetItem は、レスポンスのレイテンシーを最小限に抑えるために項目を並列で取得します。
> アプリケーションを設計する際は、DynamoDB では、返されるレスポンスで属性の順序が保証されないことに注意してください。リクエスト内の項目の AttributesToGet にプライマリキーバリューを含めると、項目ごとにレスポンスを解析するのに役立ちます。
> リクエストされた項目が存在しない場合、その項目のレスポンスでは何も返されません。存在しない項目に対するリクエストは、読み込みのタイプに応じた最小読み込み容量ユニットを消費します。詳細については、「DynamoDB 項目のサイズと形式」を参照してください。

リクエスト

構文

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{"RequestItems":
    {"Table1":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement":
{"N":"KeyValue2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":
{"N":"KeyValue6"}}],
        "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
    "Table2":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}}],
        "AttributesToGet": ["AttributeName4", "AttributeName5", "AttributeName6"]
        }
    }
```

```
}
```

名前	説明	必須
RequestItems	プライマリキーで取得するテーブル名および対応する項目のコンテナ。項目をリクエストしている間、各テーブル名はオペレーションごとに 1 回だけ呼び出すことができます。 型: 文字列 デフォルト: なし	はい
Table	取得する項目を含んでいるテーブルの名前。エントリは、ラベルのない既存のテーブルを指定する文字列です。 型: 文字列 デフォルト: なし	はい
Table:Keys	指定されたテーブルの項目を定義するプライマリのキーバリュー。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: キー	はい
Table:AttributesToGet	指定したテーブル内の属性名の配列。属性名が指定されていない場合、すべての属性が返されます。見つからなかった属性は結果に表示されません。	いいえ

名前	説明	必須
	型: 配列	
Table:ConsistentRead	true に設定されている場合、整合性のある読み込みが発行されます。それ以外の場合、結果整合性が使用されます。 型: ブール値	不可

レスポンス

構文

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 855

{"Responses":
    {"Table1":
        {"Items":
        [{"AttributeName1": {"S":"AttributeValue"},
        "AttributeName2": {"N":"AttributeValue"},
        "AttributeName3": {"SS":["AttributeValue", "AttributeValue", "AttributeValue"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
        "AttributeName3": {"NS": ["AttributeValue", "AttributeValue",
 "AttributeValue"]}
        }],
    "ConsumedCapacityUnits":1},
    "Table2":
        {"Items":
        [{"AttributeName1": {"S":"AttributeValue"},
        "AttributeName2": {"N":"AttributeValue"},
        "AttributeName3": {"SS":["AttributeValue", "AttributeValue", "AttributeValue"]}
        },
        {"AttributeName1": {"S": "AttributeValue"},
        "AttributeName2": {"S": "AttributeValue"},
        "AttributeName3": {"NS": ["AttributeValue", "AttributeValue","AttributeValue"]}
```

```
        }],
    "ConsumedCapacityUnits":1}
    },
    "UnprocessedKeys":
        {"Table3":
        {"Keys":
            [{"HashKeyElement": {"S":"KeyValue1"}, "RangeKeyElement":
{"N":"KeyValue2"}},
            {"HashKeyElement": {"S":"KeyValue3"}, "RangeKeyElement":{"N":"KeyValue4"}},
            {"HashKeyElement": {"S":"KeyValue5"}, "RangeKeyElement":
{"N":"KeyValue6"}}],
        "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]}
        }
}
```

名前	説明
Responses	テーブル名とテーブルのそれぞれの項目属性。 型: マップ
Table	項目を含んでいるテーブルの名前。エントリは、ラベルのないテーブルを指定する文字列です。 型: 文字列
Items	オペレーションパラメータを満たす属性名と値のコンテナ。 型: 属性名およびそのデータ型と値のマップ。
ConsumedCapacityUnits	各テーブルで消費された読み込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。存在しない項目に対するリクエストは、読み込みのタイプに応じた最小読み込み容量ユニットを消費します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。

名前	説明
	型: 数値
UnprocessedKeys	レスポンスサイズの制限に達したなどの理由で、現在のレスポンスで処理されなかったテーブルの配列とそれぞれのキーが含まれます。UnprocessedKeys 値は RequestItems パラメータと同じ形式です (値は後続の BatchGetItem オペレーションに直接渡すことができます)。詳細については、上記の RequestItems パラメータを参照してください。 型: 配列
UnprocessedKeys : Table: Keys	項目と、項目に関連付けられた属性を定義するプライマリキー属性値。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: 属性の名前と値のペアの配列。
UnprocessedKeys : Table: AttributesToGet	指定したテーブル内の属性名。属性名が指定されていない場合、すべての属性が返されます。見つからなかった属性は結果に表示されません。 型: 属性名の配列。
UnprocessedKeys : Table: ConsistentRead	true に設定されている場合、指定されたテーブルに対して整合性のある読み込みが使用されます。それ以外の場合は、結果整合性のある読み込みが使用されます。 型: ブール。

特殊なエラー

エラー	説明
ProvisionedThroughputExceededException	許可されているプロビジョニングスループットの最大値を超えました。

例

次の例は、BatchGetItem オペレーションを使用した HTTP POST リクエストとレスポンスを示しています。AWS SDK を使用した例については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト例

次の例では、2 つの異なるテーブルの属性をリクエストします。

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0
content-length: 409

{"RequestItems":
    {"comp1":
        {"Keys":
            [{"HashKeyElement":{"S":"Casey"},"RangeKeyElement":{"N":"1319509152"}},
            {"HashKeyElement":{"S":"Dave"},"RangeKeyElement":{"N":"1319509155"}},
            {"HashKeyElement":{"S":"Riley"},"RangeKeyElement":{"N":"1319509158"}}],
        "AttributesToGet":["user","status"]},
    "comp2":
        {"Keys":
            [{"HashKeyElement":{"S":"Julie"}},{"HashKeyElement":{"S":"Mingus"}}],
        "AttributesToGet":["user","friends"]}
    }
}
```

レスポンス例

以下の例がレスポンスです。

```
HTTP/1.1 200 OK
x-amzn-RequestId: GTPQVRM4VJS792J1UFJTKUBVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 373
Date: Fri, 02 Sep 2011 23:07:39 GMT

{"Responses":
    {"comp1":
        {"Items":
            [{"status":{"S":"online"},"user":{"S":"Casey"}},
            {"status":{"S":"working"},"user":{"S":"Riley"}},
            {"status":{"S":"running"},"user":{"S":"Dave"}}],
        "ConsumedCapacityUnits":1.5},
    "comp2":
        {"Items":
            [{"friends":{"SS":["Elisabeth", "Peter"]},"user":{"S":"Mingus"}},
            {"friends":{"SS":["Dave", "Peter"]},"user":{"S":"Julie"}}],
        "ConsumedCapacityUnits":1}
    },
    "UnprocessedKeys":{}
}
```

BatchWriteItem

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

このオペレーションでは、単一の呼び出しで、複数のテーブルの複数の項目を配置または削除できます。

1 つの項目をアップロードするには、`PutItem` を使用できます。1 つの項目を削除するには、`DeleteItem` を使用できます。ただし、Amazon EMR (Amazon EMR) から大量のデータをアップロードする場合や別のデータベースのデータを DynamoDB に移行する場合など、大量のデータをアップロードまたは削除する場合、`BatchWriteItem` では効率的な方法が用意されています。

Java などの言語を使用する場合は、スレッドを使用して項目を並列でアップロードできます。これにより、スレッドを処理するアプリケーションの複雑さが増します。他の言語はスレッドの使用をサポートしていません。例えば、PHP を使用している場合は、項目を 1 つずつアップロードまたは削除する必要があります。どちらの状況でも、BatchWriteItem は、指定されたアップロードおよび削除オペレーションを並列で処理する代替手段を提供し、アプリケーションを複雑にすることなくスレッドプールのアプローチの威力を発揮します。

BatchWriteItem オペレーションで指定された個々のアップロードと削除のコストは、消費される容量単位と同じです。ただし、指定したオペレーションが BatchWriteItem によって並列で実行されると、レイテンシーが小さくなります。存在しない項目に対する削除オペレーションは、書き込み容量ユニットを 1 つ消費します。消費される容量ユニットの詳細については、「DynamoDB でのテーブルとデータの操作」を参照してください。

BatchWriteItem を使用する場合、以下の制限に注意してください。

- 1 つのリクエストにおける最大オペレーション数 - 最大 25 個のアップロードまたは削除オペレーションを指定できます。ただし、リクエストの合計サイズは 1 MB (HTTP ペイロード) を超えることはできません。
- BatchWriteItem オペレーションは、項目のアップロードと削除行うためにのみ使用できます。既存の項目を更新するためには使用できません。
- アトミック操作ではありません - 1 つの BatchWriteItem で指定されている個々の操作はアトミックです。ただし、BatchWriteItem 全体が「ベストエフォート」操作であり、アトミック操作ではありません。したがって、BatchWriteItem リクエストでは、一部のオペレーションは成功し、他のオペレーションは失敗する可能性があります。失敗したオペレーションは、レスポンスの UnprocessedItems フィールドに返されます。これらの失敗の原因としては、テーブルに設定されているプロビジョニングされたスループットを超えたか、ネットワークエラーなどの一時的な障害が考えられます。リクエストを調査し、必要に応じてリクエストを再送信できます。通常、ループ内で BatchWriteItem を呼び出し、各反復で未処理の項目をチェックして、未処理の項目とともに新しい BatchWriteItem リクエストを送信します。
- 項目は返されません - BatchWriteItem は大量のデータを効率的にアップロードするように設計されています。PutItem および DeleteItem で提供されている高度な機能の一部は提供されていません。例えば、DeleteItem は、レスポンス内の削除済み項目をリクエストするためにリクエストボディ内の ReturnValues フィールドをサポートします。BatchWriteItem オペレーションはレスポンスで項目を返しません。
- PutItem や DeleteItem とは異なり、BatchWriteItem では、オペレーション内の個々の書き込みリクエストに対して条件を指定することはできません。

- 属性値は NULL であってはなりません。文字列型およびバイナリ型属性の長さは 0 より大きくなければなりません。また、セット型属性は空であってはなりません。空の値を持つリクエストは、ValidationException で拒否されます。

DynamoDB は、次のいずれかが真である (true) 場合バッチ書き込みオペレーション全体を拒否します。

- BatchWriteItem リクエスト内で指定したテーブルが存在しない。
- リクエスト内の項目で指定されたプライマリキー属性が、対応するテーブルのプライマリキースキーマと一致しない。
- 同じ BatchWriteItem リクエストで同じ項目に複数のオペレーションを実行する。例えば、同じ項目を同一の BatchWriteItem リクエストでアップロードおよび削除することはできません。
- 合計リクエストサイズが 1 MB のリクエストサイズ (HTTP ペイロード) 制限を超過する。
- バッチ内の個々の項目が 64 KB の項目のサイズ制限を超えている。

リクエスト

構文

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
  "RequestItems" :  RequestItems
}

RequestItems
{
     "TableName1" :  [ Request, Request, ... ],
     "TableName2" :  [ Request, Request, ... ],
     ...
}

Request ::=
  PutRequest | DeleteRequest
```

```
PutRequest ::=
{
  "PutRequest" : {
     "Item" : {
        "Attribute-Name1" : Attribute-Value,
        "Attribute-Name2" : Attribute-Value,
         ...
     }
  }
}

DeleteRequest ::=
{
   "DeleteRequest" : {
      "Key" : PrimaryKey-Value
   }
}


PrimaryKey-Value ::= HashTypePK | HashAndRangeTypePK

HashTypePK ::=
{
   "HashKeyElement" : Attribute-Value
}

HashAndRangeTypePK
{
   "HashKeyElement" : Attribute-Value,
   "RangeKeyElement" : Attribute-Value,
}

Attribute-Value ::= String | Numeric| Binary | StringSet | NumericSet | BinarySet

Numeric ::=
{
   "N": "Number"
}

String ::=
{
   "S": "String"
}
```

```
Binary ::=
{
    "B": "Base64 encoded binary data"
}

StringSet ::=
{
   "SS": [ "String1", "String2", ... ]
}

NumberSet ::=
{
   "NS": [ "Number1", "Number2", ... ]
}

BinarySet ::=
{
   "BS": [ "Binary1", "Binary2", ... ]
}
```

リクエストボディでは、RequestItems JSON オブジェクトは、実行するオペレーションを記述します。オペレーションはテーブルごとにグループ化されます。BatchWriteItem を使用して、複数のテーブル間で複数の項目を更新または削除できます。書き込みリクエストごとに、リクエストのタイプ (PutItem、DeleteItem) を指定した後にオペレーションに関する詳細情報を指定する必要があります。

- PutRequest では、項目 (属性とその値のリスト) を指定します。
- DeleteRequest では、プライマリキーの名前と値を指定します。

レスポンス

構文

レスポンスで返される JSON 本文の構文を以下に示します。

```
{
  "Responses" :         ConsumedCapacityUnitsByTable
  "UnprocessedItems" :  RequestItems
}

ConsumedCapacityUnitsByTable
```

```
{
    "TableName1" : { "ConsumedCapacityUnits", : NumericValue },
    "TableName2" : { "ConsumedCapacityUnits", : NumericValue },
     ...
}
```

RequestItems
This syntax is identical to the one described in the JSON syntax in the request.

特殊なエラー

このオペレーションに固有のエラーはありません。

例

次の例は、HTTP POST リクエストと BatchWriteItem オペレーションのレスポンスを示しています。リクエストは、Reply テーブルと Thread テーブルに対して以下のオペレーションを指定します。

- Reply テーブルに対して 1 つの項目の配置と削除を行う
- Thread テーブルに項目を配置する

AWS SDK を使用した例については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
  "RequestItems":{
    "Reply":[
      {
        "PutRequest":{
          "Item":{
            "ReplyDateTime":{
              "S":"2012-04-03T11:04:47.034Z"
            },
```

```
            "Id":{
              "S":"DynamoDB#DynamoDB Thread 5"
            }
          }
        }
      },
      {
        "DeleteRequest":{
          "Key":{
            "HashKeyElement":{
              "S":"DynamoDB#DynamoDB Thread 4"
            },
            "RangeKeyElement":{
              "S":"oops - accidental row"
            }
          }
        }
      }
    ],
    "Thread":[
      {
        "PutRequest":{
          "Item":{
            "ForumName":{
              "S":"DynamoDB"
            },
            "Subject":{
              "S":"DynamoDB Thread 5"
            }
          }
        }
      }
    ]
  }
}
```

レスポンス例

次のレスポンスの例は、Thread テーブルと Reply テーブルの両方に対する配置オペレーションが成功し、Reply テーブルに対する削除オペレーションが (テーブルでプロビジョニングされたスループットを超えた場合に発生するスロットリングなどの理由で) 失敗したことを示しています。JSON レスポンスでは、以下の点に注意してください。

- Responses オブジェクトは、Thread テーブルと Reply テーブルのそれぞれで成功した配置オペレーションの結果として両方のテーブルで 1 つの容量ユニットが消費されます。
- UnprocessedItems オブジェクトには、Reply テーブルで正常に完了しなかった削除オペレーションが示されます。その後、新しい BatchWriteItem 呼び出しを発行して、これらの未処理のリクエストに対処できます。

```
HTTP/1.1 200 OK
x-amzn-RequestId: G8M9ANLOE5QA26AEUHJKJE0ASBVV4KQNSO5AEMVJF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 536
Date: Thu, 05 Apr 2012 18:22:09 GMT

{
   "Responses":{
      "Thread":{
         "ConsumedCapacityUnits":1.0
      },
      "Reply":{
         "ConsumedCapacityUnits":1.0
      }
   },
   "UnprocessedItems":{
      "Reply":[
         {
            "DeleteRequest":{
               "Key":{
                  "HashKeyElement":{
                     "S":"DynamoDB#DynamoDB Thread 4"
                  },
                  "RangeKeyElement":{
                     "S":"oops - accidental row"
                  }
               }
            }
         }
      ]
   }
}
```

CreateTable

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

`CreateTable` オペレーションは新しいテーブルをアカウントに追加します。

テーブル名は、リクエストを発行する AWS アカウントに関連付けられている名前と、リクエストを受信する AWS リージョン (dynamodb.us-west-2.amazonaws.com など) の中で一意である必要があります。各 DynamoDB エンドポイントは完全に独立しています。例えば、"MyTable" というテーブルが dynamodb.us-west-2.amazonaws.com と dynamodb.us-west-1.amazonaws.com にある場合、この 2 つの "MyTable" テーブルは相互に独立していてデータを共有しません。

`CreateTable` オペレーションは、テーブルの作成を開始する非同期ワークフローをトリガーします。DynamoDB は、テーブルが `ACTIVE` 状態になるまで直ちにテーブルの状態 (`CREATING`) を返します。テーブルが `ACTIVE` 状態になると、データプレーンオペレーションを実行できます。

テーブルのステータスを確認するには、DescribeTables オペレーションを使用します。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}
```

```
}
```

名前	説明	必須
TableName	作成するテーブルの名前。 使用できる文字は、a ~ z、A ~ Z、0 ~ 9、'_' (アンダースコア)、'-' (ダッシュ)、'.' (ドット) です。名前は 3 ~ 255 文字の長さです。 型: 文字列	はい
KeySchema	テーブルのプライマリキー (シンプルまたは複合) 構造。HashKeyElement の名前と値のペアは必須です。RangeKeyElement の名前と値のペアはオプションです (複合プライマリキーにのみ必要)。プライマリキーの詳細については、「プライマリキー」を参照してください。 プライマリキー要素の名前は 1 ~ 255 文字の長さで、使用できる文字の制限はありません。 AttributeType に指定できる値は、「S」(文字列)、「N」(数値)、または「B」(バイナリ) です。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および	はい

名前	説明	必須
	RangeKeyElement のマップ。	
ProvisionedThroughput	指定したテーブルの新しいスループット (ReadCapacityUnits と WriteCapacityUnits の値で構成されます)。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 **Note** 現在の最大値/最小値については、「Amazon DynamoDB のクォータ」を参照してください。 型: 配列	はい

名前	説明	必須
ProvisionedThroughput : ReadCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される整合性のある ReadCapacityUnits の最小数を設定します。 結果整合性のある読み込みオペレーションに必要な労力は、整合性のある読み込みオペレーションより少ないので、1 秒あたり 50 の整合性のある ReadCapacityUnits の設定では、1 秒あたり 100 の結果整合性のある ReadCapacityUnits が実現します。 型: 数値	はい
ProvisionedThroughput : WriteCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される整合性のある WriteCapacityUnits の最小数を設定します 型: 数値	可能

レスポンス

構文

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.310506263362E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"CREATING"
    }
}
```

名前	説明
TableDescription	テーブルプロパティのコンテナ。
CreationDateTime	テーブルの作成日 (UNIX エポック時間)。 型: 数値
KeySchema	テーブルのプライマリキー (シンプルまたは複合) 構造。HashKeyElement の名前と値のペアは必須です。RangeKeyElement の名前と値のペアはオプションです (複合プライマリキーにのみ必要)。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement のマップ。

名前	説明
ProvisionedThroughput	指定したテーブルのスループット (ReadCapacityUnits と WriteCapacityUnits の値で構成されます)。「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 配列
ProvisionedThroughput :ReadCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に 1 秒あたりに消費される ReadCapacityUnits の最小数。 型: 数値
ProvisionedThroughput :WriteCapacityUnits	WriteCapacityUnits がロードとその他のオペレーションのバランスを取る前に 1 秒あたりに消費される ReadCapacityUnits の最小数。 型: 数値
TableName	作成されたテーブルの名前。 型: 文字列
TableStatus	テーブルの現在の状態 (CREATING)。テーブルが ACTIVE 状態になったらデータを配置できます。 テーブルのステータスを確認するには、DescribeTables API を使用します。 型: 文字列

特殊なエラー

エラー	説明
ResourceInUseException	既存のテーブルを再度作成しようとしました。
LimitExceededException	同時テーブルリクエストの数 (CREATING、DELETING、または UPDATING の状態にある累積的なテーブル数) が許容最大値を超えています。 **Note** 現在の最大値/最小値については、「Amazon DynamoDB のクォータ」を参照してください。 .

例

次の例は、文字列と数値を含む複合プライマリキーを持つテーブルを作成します。AWS SDK を使用した例については、「DynamoDB でのテーブルとデータの操作」を参照してください。

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0


{"TableName":"comp-table",
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10}
```

```
}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.310506263362E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
    "TableName":"comp-table",
    "TableStatus":"CREATING"
    }
}
```

関連アクション

- DescribeTables
- DeleteTable

DeleteItem

> **Important**
>
> ***##########API ##### 2011-12-05 ## #######***
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

プライマリキーを使用してテーブルから 1 つの項目を削除します。項目が存在する場合、または予期される属性値が項目にある場合は、項目を削除する条件付き削除オペレーションを実行できます。

> **Note**
>
> 属性または値なしで DeleteItem を指定すると、項目のすべての属性が削除されます。
> 条件を指定した場合を除き、DeleteItem は、べき等性の操作です。同じ項目または属性に対して複数回実行してもエラーレスポンスが返されません。
> 条件付き削除は、特定の条件が満たされた場合にのみ項目と属性を削除する場合に便利です。条件が満たされると、DynamoDB は削除を実行します。それ以外の場合、項目は削除されません。
> 1 度のオペレーションにつき 1 つの属性に対して、予期される条件チェックを実行できます。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Key":
        {"HashKeyElement":{"S":"AttributeValue1"},"RangeKeyElement":
{"N":"AttributeValue2"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3"}}},
    "ReturnValues":"ALL_OLD"}
}
```

名前	説明	必須
TableName	削除する項目を含むテーブルの名前。 型: 文字列	はい
Key	項目を定義するプライマリキー。プライマリキーの詳	はい

名前	説明	必須
	細については、「プライマリキー」を参照してください。 型: HashKeyElement の値に対するマップと RangeKeyElement の値に対するマップ。	
Expected	条件付き削除の属性を指定します。Expected パラメータを使用すると、属性名を指定できます。また、DynamoDB が属性を削除する前に特定の値があるかどうかを確認するかどうかも指定できます。 型: 属性名のマップ。	いいえ
Expected:AttributeName	条件付き配置の属性の名前。 型: 文字列	いいえ

名前	説明	必須

Expected:AttributeName: ExpectedAttributeValue	このパラメータを使用して、属性名と値のペアに値が既に存在するかどうかを指定します。
	
	次の JSON 表記法では、「Color」属性が存在しない項目が削除されます。
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	次の JSON 表記法では、項目を削除する前に、「Color」という名前の属性の既存の値が「Yellow」であるかどうかがチェックされます。
	
	`"Expected" : {"Color":{"Exists":true},{"Value":{"S":"Yellow"}}}`
	
	デフォルトでは、Expected パラメータを使用し、Value を指定した場合、DynamoDB は属性が存在し、置換する現在の値があると仮定します。したがって、{"Exists":true} は暗示されているので指定する必要はありません。リクエストを短縮するには、次のようにします。
	
	`"Expected" :`
		いいえ

名前	説明	必須
	{"Color":{"Value":{"S":"Yellow"}}} **Note** チェックする属性値なしで {"Exists":true} を指定すると、DynamoDB はエラーを返します。	
ReturnValues	削除する前に属性名と値のペアを取得する場合、このパラメータを使用します。使用できるパラメータの値は、NONE (デフォルト) または ALL_OLD です。ALL_OLD を指定した場合、古い項目の内容が返されます。このパラメータを指定しない場合、または NONE の場合、何も返されません。 型: 文字列	不可

レスポンス

構文

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"Attributes":
    {"AttributeName3":{"SS":["AttributeValue3","AttributeValue4","AttributeValue5"]},
```

```
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName1":{"N":"AttributeValue1"}
    },
"ConsumedCapacityUnits":1
}
```

名前	説明
Attributes	リクエストで ReturnValues パラメータが ALL_OLD として指定されている場合、DynamoDB は属性名と値のペア (実質的に、削除された項目) の配列を返します。それ以外の場合、レスポンスには空のセットが含まれます。 型: 属性の名前と値のペアの配列。
ConsumedCapacityUnits	オペレーションによって消費される書き込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。存在しない項目に対する削除リクエストは、書き込み容量ユニットを 1 つ消費します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

エラー	説明
ConditionalCheckFailedException	条件チェックに失敗しました。予期された属性値が見つかりませんでした。

例

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteItem
content-type: application/x-amz-json-1.0

{"TableName":"comp-table",
    "Key":
        {"HashKeyElement":{"S":"Mingus"},"RangeKeyElement":{"N":"200"}},
    "Expected":
        {"status":{"Value":{"S":"shopping"}}},
    "ReturnValues":"ALL_OLD"
}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: U9809LI6BBFJA5N2R0TB0P017JVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 22:31:23 GMT

{"Attributes":
    {"friends":{"SS":["Dooley","Ben","Daisy"]},
    "status":{"S":"shopping"},
    "time":{"N":"200"},
    "user":{"S":"Mingus"}
    },
"ConsumedCapacityUnits":1
}
```

関連アクション

- PutItem

DeleteTable

> **Important**
>
> ***##########API ##### 2011-12-05 ##################################### #######***
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

DeleteTable オペレーションは、テーブルとそのすべての項目を削除します。DeleteTable リクエストの後、指定されたテーブルは、DynamoDB が削除を完了するまで DELETING 状態になります。ACTIVE 状態になったらテーブルを削除できます。テーブルが CREATING または UPDATING 状態の場合、DynamoDB は ResourceInUseException エラーを返します。指定されたテーブルが存在しない場合、DynamoDB は、ResourceNotFoundException を返します。テーブルが既に DELETING 状態である場合、エラーは返されません。

> **Note**
>
> DynamoDB は、テーブルの削除が完了するまで、DELETING 状態のテーブルで引き続きデータプレーンオペレーションのリクエスト (GetItem および PutItem など) を受け入れることがあります。

テーブルは、リクエストを発行する AWS アカウントに関連付けられているテーブルと、リクエストを受信する AWS リージョン(dynamodb.us-west-1.amazonaws.com など) の中で一意です。各 DynamoDB エンドポイントは完全に独立しています。例えば、"MyTable" というテーブルが dynamodb.us-west-2.amazonaws.com と dynamodb.us-west-1.amazonaws.com にある場合、この 2 つの "MyTable" テーブルは相互に独立していてデータを共有しません。一方を削除しても、もう一方は削除されません。

テーブルのステータスを確認するには、DescribeTables オペレーションを使用します。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

名前	説明	必須
TableName	削除するテーブルの名前。 型: 文字列	可能

レスポンス

構文

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Sun, 14 Aug 2011 22:56:22 GMT

{"TableDescription":
    {"CreationDateTime":1.313362508446E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"DELETING"
    }
}
```

名前	説明
TableDescription	テーブルプロパティのコンテナ。
CreationDateTime	テーブルの作成日。 型: 数値
KeySchema	テーブルのプライマリキー (シンプルまたは複合) 構造。HashKeyElement の名前と値のペアは必須です。RangeKeyElement の名前と値のペアはオプションです (複合プライマリキーにのみ必要)。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement のマップ。
ProvisionedThroughput	指定したテーブルのスループット (ReadCapacityUnits と WriteCapacityUnits の値で構成されます)。「DynamoDB プロビジョンドキャパシティモード」を参照してください。
ProvisionedThroughput : ReadCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される ReadCapacityUnits の最小数。 型: 数値
ProvisionedThroughput : WriteCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される WriteCapacityUnits の最小数。

名前	説明
	型: 数値
TableName	削除されたテーブルの名前。 型: 文字列
TableStatus	テーブルの現在の状態 (DELETING)。テーブルが削除されると、テーブルに対する後続のリクエストは resource not found を返します。 テーブルのステータスを確認するには、DescribeTables オペレーションを使用します。 型: 文字列

特殊なエラー

エラー	説明
ResourceInUseException	CREATING または UPDATING の状態のテーブルは削除できません。

例

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0
content-length: 40

{"TableName":"favorite-movies-table"}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 160
Date: Sun, 14 Aug 2011 17:20:03 GMT

{"TableDescription":
    {"CreationDateTime":1.313362508446E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"name","AttributeType":"S"}},
    "TableName":"favorite-movies-table",
    "TableStatus":"DELETING"
}
```

関連アクション

- CreateTable
- DescribeTables

DescribeTables

> **Important**
>
> ***##########API ##### 2011-12-05 ## #######***
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

テーブルに関する情報 (テーブルの現在のステータス、プライマリキーのスキーマ、テーブルが作成された日時など) を返します。DescribeTable の結果は結果整合性です。テーブル作成のプロセスで DescribeTable を使用するタイミングが早すぎる場合、DynamoDB は `ResourceNotFoundException` を返します。テーブル更新のプロセスで DescribeTable を使用するタイミングが早すぎる場合、新しい値がすぐに使用できないことがあります。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1"}
```

名前	説明	必須
TableName	説明するテーブルの名前。 型: 文字列	可能

レスポンス

構文

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
Content-Length: 543


{"Table":
    {"CreationDateTime":1.309988345372E9,
    ItemCount:1,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},
    "ProvisionedThroughput":{"LastIncreaseDateTime": Date, "LastDecreaseDateTime":
 Date, "ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableSizeBytes":1,
    "TableStatus":"ACTIVE"
    }
```

```
}
```

名前	説明
Table	記述されているテーブルのコンテナ。 型: 文字列
CreationDateTime	テーブルの作成日 (UNIX エポック時間)。
ItemCount	指定されたテーブルに含まれる項目の数。DynamoDB は、この値を約 6 時間ごとに更新します。最近の変更は、この値に反映されないことがあります。 型: 数値
KeySchema	テーブルのプライマリキー (シンプルまたは複合) 構造。HashKeyElement の名前と値のペアは必須です。RangeKeyElement の名前と値のペアはオプションです (複合プライマリキーにのみ必要)。ハッシュキーの最大サイズは 2048 バイトです。範囲キーの最大サイズは 1024 バイトです。両方の制限は別々に強制されます (ハッシュと範囲の合計で 2048 + 1024 キーを組み合わせることができます)。プライマリキーの詳細については、「プライマリキー」を参照してください。
ProvisionedThroughput	指定されたテーブルのスループット。LastIncreaseDateTime (該当する場合)、LastDecreaseDateTime (該当する場合)、ReadCapacityUnits 、および WriteCapacityUnits で構成されます。テーブルのスループットが増減したことがない場合、DynamoDB はこれらの要素の値を返しません。「DynamoDB プロビジョンドキャパシティモード」を参照してください。

名前	説明
	型: 配列
TableName	リクエストされたテーブルの名前。 型: 文字列
TableSizeBytes	指定されたテーブルの合計サイズ (バイト単位)。DynamoDB は、この値を約 6 時間ごとに更新します。最近の変更は、この値に反映されないことがあります。 型: 数値
TableStatus	テーブルの現在の状態 (CREATING、ACTIVE、DELETING、または UPDATING)。テーブルが ACTIVE の状態になったらデータを追加できます。

特殊なエラー

このオペレーションに固有のエラーはありません。

例

次の例は、「comp-table」という名前のテーブルに対する DescribeTable オペレーションを使用した HTTP POST リクエストとレスポンスを示しています。テーブルには複合プライマリキーがあります。

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName":"users"}
```

レスポンス例

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 543

{"Table":
    {"CreationDateTime":1.309988345372E9,
    "ItemCount":23,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":{"LastIncreaseDateTime": 1.309988345384E9,
 "ReadCapacityUnits":10,"WriteCapacityUnits":10},
    "TableName":"users",
    "TableSizeBytes":949,
    "TableStatus":"ACTIVE"
    }
}
```

関連アクション

- CreateTable
- DeleteTable
- ListTables

GetItem

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

GetItem オペレーションは、プライマリキーに一致する項目の Attributes のセットを返します。一致する項目がない場合、GetItem はデータを返しません。

デフォルトでは、GetItem オペレーションは、結果整合性のある読み込みを提供します。結果整合性のある読み込みがアプリケーションで受け入れられない場合は、ConsistentRead を使用します。このオペレーションは標準の読み込みよりも時間がかかることがありますが、常に最後に更新された値を返します。詳細については、「DynamoDB の読み取り整合性」を参照してください。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
  "Key":
  {"HashKeyElement": {"S":"AttributeValue1"},
  "RangeKeyElement": {"N":"AttributeValue2"}
 },
 "AttributesToGet":["AttributeName3","AttributeName4"],
 "ConsistentRead":Boolean
}
```

名前	説明	必須
TableName	リクエストされた項目を含むテーブルの名前。 型: 文字列	はい
Key	項目を定義するプライマリキーバリュー。プライマリキーの詳細については、「プライマリキー」を参照してください。	はい

名前	説明	必須
	型: HashKeyElement の値に対するマップと RangeKeyElement の値に対するマップ。	
AttributesToGet	属性名の配列。属性名が指定されていない場合、すべての属性が返されます。見つからなかった属性は結果に表示されません。 型: 配列	いいえ
ConsistentRead	true に設定されている場合、整合性のある読み込みが発行されます。それ以外の場合、結果整合性が使用されます。 型: ブール値	不可

レスポンス

構文

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 144

{"Item":{
 "AttributeName3":{"S":"AttributeValue3"},
 "AttributeName4":{"N":"AttributeValue4"},
 "AttributeName5":{"B":"dmFsdWU="}
 },
"ConsumedCapacityUnits": 0.5
}
```

名前	説明
Item	リクエストされた属性が含まれます。 型: 属性の名前と値のペアのマップ。
ConsumedCapacityUnits	オペレーションによって消費される読み込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。存在しない項目に対するリクエストは、読み込みのタイプに応じた最小読み込み容量ユニットを消費します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

このオペレーションに固有のエラーはありません。

例

AWS SDK を使用した例については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"comptable",
 "Key":
  {"HashKeyElement":{"S":"Julie"},
  "RangeKeyElement":{"N":"1307654345"}},
 "AttributesToGet":["status","friends"],
 "ConsistentRead":true
}
```

レスポンス例

ConsumedCapacityUnits の値が 1 であることに注意してください。これは、オプションのパラメータ ConsistentRead が true に設定されているからです。同じリクエストで ConsistentRead が false に設定されている場合 (または指定されていない場合)、レスポンスは結果整合性で、ConsumedCapacityUnits の値は 0.5 になります。

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 72

{"Item":
 {"friends":{"SS":["Lynda, Aaron"]},
 "status":{"S":"online"}
 },
"ConsumedCapacityUnits": 1
}
```

ListTables

> **Important**
>
> ***##########API ##### 2011-12-05 ## #######***
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

現在のアカウントおよびエンドポイントに関連付けられたテーブルの配列を返します。

各 DynamoDB エンドポイントは完全に独立しています。例えば、"MyTable" というテーブルが dynamodb.us-west-2.amazonaws.com と dynamodb.us-east-1.amazonaws.com にある場合、この 2 つの "MyTable" テーブルは相互に独立していてデータを共有しません。ListTables オペレーションは、リクエストを受信するエンドポイントについて、リクエストを作成するアカウントに関連付けられたすべてのテーブル名を返します。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0

{"ExclusiveStartTableName":"Table1","Limit":3}
```

デフォルトでは、ListTables オペレーションは、リクエストを受信するエンドポイントについて、リクエストを作成するアカウントに関連付けられたすべてのテーブル名をリクエストします。

名前	説明	必須
Limit	返されるテーブル名の最大数。 型: 整数	いいえ
ExclusiveStartTableName	リストの最初のテーブルの名前。すでに ListTables オペレーションを実行し、レスポンスで LastEvaluatedTableName 値を受け取っている場合、その値をここで使用してリストを続行します。 型: 文字列	不可

レスポンス

構文

```
HTTP/1.1 200 OK
```

```
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"TableNames":["Table1","Table2","Table3"], "LastEvaluatedTableName":"Table3"}
```

名前	説明
TableNames	現在のエンドポイントの現在のアカウントに関連付けられているテーブルの名前。 型: 配列
LastEvaluatedTableName	現在のリストの最後のテーブルの名前 (アカウントとエンドポイントの一部のテーブルが返されていない場合のみ)。すべてのテーブル名がすでに返されている場合、この値はレスポンスに存在しません。すべてのテーブル名が返されるまでリストを続行するには、この値を新しいリクエスト内で ExclusiveStartTableName として使用します。 型: 文字列

特殊なエラー

このオペレーションに固有のエラーはありません。

例

次の例は、ListTables オペレーションを使用した HTTP POST リクエストとレスポンスを示しています。

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
```

```
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0

{"ExclusiveStartTableName":"comp2","Limit":3}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"LastEvaluatedTableName":"comp5","TableNames":["comp3","comp4","comp5"]}
```

関連アクション

- DescribeTables
- CreateTable
- DeleteTable

PutItem

> **Important**
>
> ***##########API ##### 2011-12-05 ## #######***
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

新しい項目を作成するか、古い項目を新しい項目で置き換えます (すべての属性を含む)。指定されたテーブルに同じプライマリキーを持つ項目がすでに存在する場合、新しい項目によって既存の項目が完全に置き換えられます。条件付き配置 (指定されたプライマリキーを持つ項目が存在しない場合に新しい項目を挿入) を実行することや、特定の属性値を持つ既存の項目を置き換えることができます。

属性値は NULL であってはなりません。文字列型およびバイナリ型属性の長さは 0 より大きくなければなりません。また、セット型属性は空であってはなりません。空の値を持つリクエストは、ValidationException で拒否されます。

> **Note**
>
> 新しい項目によって既存の項目が置き換えられないようにするには、プライマリキー属性に対して Exists を false に設定した条件付き配置オペレーション、または属性を使用します。

PutItem の使用の詳細については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Item":{
        "AttributeName1":{"S":"AttributeValue1"},
        "AttributeName2":{"N":"AttributeValue2"},
        "AttributeName5":{"B":"dmFsdWU="}
    },
    "Expected":{"AttributeName3":{"Value": {"S":"AttributeValue"}, "Exists":Boolean}},
    "ReturnValues":"ReturnValuesConstant"}
```

名前	説明	必須
TableName	項目を含めるテーブルの名前。 型: 文字列	はい
Item	項目の属性のマップ。項目を定義するプライマリキーバ	はい

名前	説明	必須
	リューを含む必要があります。項目には、他の属性名と値のペアを指定できます。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: 属性名から属性値へのマップ。	
Expected	条件付き配置の属性を指定します。Expected パラメータを使用すると、属性名を指定できます。また、DynamoDB で更新を行う前に、属性値が既に存在するかどうか、属性値が存在する場合は特定の値があるかどうかの確認の有無も指定できます。 型: 属性名の属性値へのマップ、および属性値が存在するかどうか。	いいえ
Expected:AttributeName	条件付き配置の属性の名前。 型: 文字列	いいえ

名前	説明	必須

Expected:AttributeName: ExpectedAttributeValue	このパラメータを使用して、属性名と値のペアに値が既に存在するかどうかを指定します。
	
	次の JSON 表記法では、項目に「Color」属性がまだ存在しない場合、その項目が置き換えられます。
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	次の JSON 表記法では、項目を置き換える前に、「Color」という名前の属性の既存の値が「Yellow」であるかどうかがチェックされます。
	
	`"Expected" : {"Color":{"Exists":true,{"Value":{"S":"Yellow"}}}}`
	
	デフォルトでは、Expected パラメータを使用し、Value を指定した場合、DynamoDB は属性が存在し、置換する現在の値があると仮定します。したがって、{"Exists":true} は暗示されているので指定する必要はありません。リクエストを短縮するには、次のようにします。
		いいえ

名前	説明	必須
	`"Expected" : {"Color":{"Value": {"S":"Yellow"}}}` **Note** チェックする属性値なしで `{"Exists" :true}` を指定すると、DynamoDB はエラーを返します。	
`ReturnValues`	`PutItem` リクエストで属性名と値のペアを更新する前に属性名と値のペアを取得する場合、このパラメータを使用します。使用できるパラメータの値は、`NONE` (デフォルト) または `ALL_OLD` です。`ALL_OLD` を指定し、`PutItem` によって属性名と値のペアが上書きされた場合、古い項目の内容が返されます。このパラメータを指定しない場合、または `NONE` の場合、何も返されません。 型: 文字列	不可

レスポンス

構文

次の構文例では、`ALL_OLD` の `ReturnValues` パラメータが指定されたリクエストを引き受けます。指定されていない場合、レスポンスには `ConsumedCapacityUnits` 要素のみが含まれます。

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 85

{"Attributes":
 {"AttributeName3":{"S":"AttributeValue3"},
 "AttributeName2":{"SS":"AttributeValue2"},
 "AttributeName1":{"SS":"AttributeValue1"},
 },
"ConsumedCapacityUnits":1
}
```

名前	説明
Attributes	配置オペレーションの前の属性値 (リクエストで ReturnValues パラメータが ALL_OLD として指定されている場合のみ)。 型: 属性の名前と値のペアのマップ。
ConsumedCapacityUnits	オペレーションによって消費される書き込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

エラー	説明
ConditionalCheckFailedException	条件チェックに失敗しました。予期された属性値が見つかりませんでした。

エラー	説明
ResourceNotFoundException	指定された項目または属性が見つかりませんでした。

例

AWS SDK を使用した例については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Item":
  {"time":{"N":"300"},
  "feeling":{"S":"not surprised"},
  "user":{"S":"Riley"}
  },
 "Expected":
  {"feeling":{"Value":{"S":"surprised"},"Exists":true}}
 "ReturnValues":"ALL_OLD"
}
```

レスポンス例

```
HTTP/1.1 200
x-amzn-RequestId: 8952fa74-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 84

{"Attributes":
 {"feeling":{"S":"surprised"},
 "time":{"N":"300"},
 "user":{"S":"Riley"}},
"ConsumedCapacityUnits":1
}
```

関連アクション

- UpdateItem
- DeleteItem
- GetItem
- BatchGetItem

Query

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

Query オペレーションオペレーションは、1 つ以上の項目の値とその属性をプライマリキーごとに取得します (Query はハッシュと範囲のプライマリキーテーブルでのみ使用できます)。特定の HashKeyValue を指定する必要があります。プライマリキーの RangeKeyValue で比較オペレーションを使用してクエリの範囲を絞り込むことができます。範囲キーごとに正順または逆順で結果を取得するには、ScanIndexForward パラメータを使用します。

結果を返さないクエリは、読み込みのタイプに応じて最小読み込み容量ユニットを消費します。

> **Note**
>
> クエリパラメータを満たす項目の合計数が 1 MB の制限を超えるとクエリが停止し、結果が後続のオペレーションでクエリを続行するための LastEvaluatedKey とともにユーザーに返されます。スキャン操作とは異なり、クエリ操作では、空の結果セットおよび LastEvaluatedKey が返されません。結果が 1 MB を超える場合、または Limit パラメータを使用した場合、LastEvaluatedKey のみが返されます。
> ConsistentRead パラメータを使用して、結果を整合性のある読み込みに設定できます。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
 "Limit":2,
 "ConsistentRead":true,
 "HashKeyValue":{"S":"AttributeValue1":},
 "RangeKeyCondition": {"AttributeValueList":
[{"N":"AttributeValue2"}],"ComparisonOperator":"GT"}
 "ScanIndexForward":true,
 "ExclusiveStartKey":{
  "HashKeyElement":{"S":"AttributeName1"},
  "RangeKeyElement":{"N":"AttributeName2"}
 },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

名前	説明	必須
TableName	リクエストされた項目を含むテーブルの名前。 型: 文字列	はい
AttributesToGet	属性名の配列。属性名が指定されていない場合、すべての属性が返されます。見つからなかった属性は結果に表示されません。 型: 配列	いいえ
Limit	返される項目の最大数 (一致する項目の数であるとは限り	いいえ

名前	説明	必須
	ません)。テーブルをクエリする間に制限までの項目数を処理した場合、DynamoDB はクエリを停止し、その時点までの一致する値とともに、後続のオペレーションでクエリを続行するために適用する LastEvaluatedKey を返します。また、この制限に達する前に結果セットのサイズが 1 MB を超えた場合、DynamoDB はクエリを停止し、一致する値とともに、後続のオペレーションでクエリをクリックするために適用する LastEvaluatedKey を返します。 型: 数値	
ConsistentRead	true に設定されている場合、整合性のある読み込みが発行されます。それ以外の場合、結果整合性が使用されます。 型: ブール値	いいえ

名前	説明	必須
Count	true に設定されている場合、DynamoDB は、一致する項目とその属性のリストではなく、クエリパラメータに一致する項目の合計数を返します。Limit パラメータをカウントのみのクエリに適用できます。 AttributesToGet のリストを提供する場合、Count を true に設定しないでください。そうでないと、DynamoDB は検証エラーを返します。詳細については、「結果での項目のカウント」を参照してください。 型: ブール値	いいえ
HashKeyValue	複合プライマリキーのハッシュコンポーネントの属性値。 型: 文字列、数値、またはバイナリ	はい

名前	説明	必須
RangeKeyCondition	クエリに使用する属性値および比較演算子のコンテナ。クエリリクエストには RangeKeyCondition は必要ありません。HashKeyValue だけを指定した場合、DynamoDB は、指定されたハッシュキー要素値を持つすべての項目を返します。 型: マップ	いいえ
RangeKeyCondition : AttributeValueList	クエリパラメータについて評価する属性値。BETWEEN 比較を指定した場合を除き、AttributeValueList には属性値が 1 つ含まれます。BETWEEN 比較では、AttributeValueList には、2 つの属性値が含まれます。 型: ComparisonOperator への AttributeValue のマップ。	いいえ

名前	説明	必須
RangeKeyCondition : ComparisonOperator	指定された属性 (等しい、より大きい、など) を評価するための基準。クエリオペレーションで有効な比較演算子は次の通りです。 **Note** より大きい、等しい、より小さいに関する文字列値の比較は、ASCII 文字コード値に基づきます。例えば、a は A より大きく、aa は B より大きいと評価されます。コードの値のリストについては、http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters を参照してください。バイナリの場合、DynamoDB は、バイナリ値を比較する際 (クエリ表現の評価など)、バイナリデータの各バイトを符号なしとしてて扱います。 型: 文字列またはバイナリ	いいえ

名前	説明	必須
	EQ: 等しい。 EQ の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。	

名前	説明	必須
	LE: より小さい、または等しい。 LE の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	LT: より小さい。 LT の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	GE: より大きい、または等しい。 GE の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	GT: より大きい。 GT の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	
	BEGINS_WITH : プレフィックスを確認します。 BEGINS_WITH の場合、AttributeValueList は、文字列またはバイナリの 1 つの AttributeValue のみを含むことができます (数値やセットではありません)。比較のターゲット属性は、(数値またはセットではなく) 文字列またはバイナリである必要があります。	

名前	説明	必須
	BETWEEN: 最初の値より大きいまたは等しい、および 2 番目の値より小さいまたは等しい。 BETWEEN の場合、AttributeValueList は、文字列、数値、またはバイナリのいずれかの同じ型の 2 つの AttributeValue 要素を含む必要があります (セットではありません)。ターゲット属性は、ターゲット値が最初の要素より大きいか等しく、2番目の要素より小さいか等しい場合に一致します。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と比較されません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
ScanIndexForward	インデックスの昇順または降順トラバーサルを指定します。DynamoDB は、範囲キーによって決定されたリクエストされた順序を反映した結果を返します。データ型が数値の場合、結果は数値順に返されます。それ以外の場合は、トラバーサルは ASCII 文字コード値に基づきます。 型: ブール値 デフォルトは true (昇順) です。	いいえ
ExclusiveStartKey	以前のクエリを続行する項目のプライマリキー。クエリが完了する前に結果セットサイズまたは Limit パラメータのいずれかが原因でクエリオペレーションが中断した場合、そのクエリで、この値が LastEvaluatedKey として提供されることがあります。新しいクエリリクエストで LastEvaluatedKey を渡し、その時点からオペレーションを続行できます。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement 。	不可

レスポンス

構文

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":[{
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"N":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
    },{
    "AttributeName1":{"S":"AttributeValue3"},
    "AttributeName2":{"N":"AttributeValue4"},
    "AttributeName3":{"S":"AttributeValue3"},
    "AttributeName5":{"B":"dmFsdWU="}
}],
    "LastEvaluatedKey":{"HashKeyElement":{"AttributeValue3":"S"},
                        "RangeKeyElement":{"AttributeValue4":"N"}
     },
     "ConsumedCapacityUnits":1
}
```

名前	説明
Items	クエリパラメータを満たす項目属性。 型: 属性名およびそのデータ型と値のマップ。
Count	レスポンス内の項目数。詳細については、「結果での項目のカウント」を参照してください。 型: 数値
LastEvaluatedKey	クエリオペレーションが停止した項目のプライマリキー (以前の結果セットを含みます)。この値を使用して、新しいリクエストでこの値を除く新しいオペレーションをスタートします。

名前	説明
	クエリ結果セット全体が完了したとき (オペレーションが「最後のページ」を処理したとき)、LastEvaluatedKey は null です。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement 。
ConsumedCapacityUnits	オペレーションによって消費される読み込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

エラー	説明
ResourceNotFoundException	指定されたターゲットが見つかりませんでした。

例

AWS SDK を使用した例については、「DynamoDB のテーブルに対するクエリの実行」を参照してください。

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
```

```
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"John"},
 "ScanIndexForward":false,
 "ExclusiveStartKey":{
  "HashKeyElement":{"S":"John"},
  "RangeKeyElement":{"S":"The Matrix"}
 }
}
```

レスポンス例

```
HTTP/1.1 200
x-amzn-RequestId: 3647e778-71eb-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":[{
 "fans":{"SS":["Jody","Jake"]},
 "name":{"S":"John"},
 "rating":{"S":"***"},
 "title":{"S":"The End"}
 },{
 "fans":{"SS":["Jody","Jake"]},
 "name":{"S":"John"},
 "rating":{"S":"***"},
 "title":{"S":"The Beatles"}
 }],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"John"},"RangeKeyElement":{"S":"The
 Beatles"}},
"ConsumedCapacityUnits":1
}
```

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0
```

```
{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"Airplane"},
 "RangeKeyCondition":{"AttributeValueList":[{"N":"1980"}],"ComparisonOperator":"EQ"},
 "ScanIndexForward":false}
```

レスポンス例

```
HTTP/1.1 200
x-amzn-RequestId: 8b9ee1ad-774c-11e0-9172-d954e38f553a
content-type: application/x-amz-json-1.0
content-length: 119

{"Count":1,"Items":[{
 "fans":{"SS":["Dave","Aaron"]},
 "name":{"S":"Airplane"},
 "rating":{"S":"***"},
 "year":{"N":"1980"}
 }],
"ConsumedCapacityUnits":1
}
```

関連アクション

- Scan

Scan

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
>
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

Scan オペレーションは、テーブルのフルスキャンを実行することにより、1 つ以上の項目とその属性を返します。より詳細な結果を取得するには、ScanFilter を使用します。

> **Note**
>
> スキャンされた項目の合計数が 1 MB の制限を超えると、スキャンが停止し、結果が後続のオペレーションでスキャンを続行するための LastEvaluatedKey とともにユーザーに返されます。結果には、制限を超えた項目数も含まれます。スキャンの結果には、フィルタ基準を満たすテーブルデータがないことがあります。
> 結果セットは結果整合性です。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Limit": 2,
    "ScanFilter":{
        "AttributeName1":{"AttributeValueList":
[{"S":"AttributeValue"}],"ComparisonOperator":"EQ"}
    },
    "ExclusiveStartKey":{
        "HashKeyElement":{"S":"AttributeName1"},
        "RangeKeyElement":{"N":"AttributeName2"}
    },
    "AttributesToGet":["AttributeName1", "AttributeName2", "AttributeName3"]},
}
```

名前	説明	必須
TableName	リクエストされた項目を含むテーブルの名前。 型: 文字列	はい
AttributesToGet	属性名の配列。属性名が指定されていない場合、すべての属性が返されます。見つからなかった属性は結果に表示されません。 型: 配列	いいえ
Limit	評価する項目の最大数 (一致する項目の数であるとは限りません)。結果を処理する間に制限までの項目数を処理した場合、DynamoDB は処理を停止し、その時点までの一致する値とともに、後続のオペレーションで項目の取得を続行するために適用する LastEvaluatedKey を返します。また、この制限に達する前に、スキャンされたデータセットのサイズが 1MB を超えた場合、DynamoDB はスキャンを停止し、制限までの一致値とともに、後続のオペレーションでスキャンを続行するために適用する LastEvaluatedKey を返します。 型: 数値	いいえ

名前	説明	必須
Count	true に設定されている場合、DynamoDB は、割り当てられたフィルターに一致する項目がなくても、スキャンオペレーションの項目の合計数を返します。Limit パラメータは、カウントのみのスキャンに適用できます。 AttributesToGet のリストを提供する場合、Count を true に設定しないでください。そうでないと、DynamoDB は検証エラーを返します。詳細については、「結果での項目のカウント」を参照してください。 型: ブール値	いいえ
ScanFilter	スキャン結果を評価し、目的の値のみを返します。複数の条件は「AND」オペレーションとして扱われます。すべての条件を満たす値だけが結果に含まれます。 型: 比較演算子を含む値への属性名のマップ。	いいえ
ScanFilter :AttributeValueList	フィルターのスキャン結果を評価するための値と条件。 型: Condition への AttributeValue のマップ。	いいえ

名前	説明	必須
`ScanFilter : ComparisonOperator`	指定された属性 (等しい、より大きい、など) を評価するための基準。スキャンオペレーションで有効な比較演算子は次の通りです。 **Note** より大きい、等しい、より小さいに関する文字列値の比較は、ASCII 文字コード値に基づきます。例えば、`a` は `A` より大きく、`aa` は `B` より大きいと評価されます。コードの値のリストについては、http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters を参照してください。バイナリの場合、DynamoDB は、バイナリ値を比較する際 (クエリ表現の評価など)、バイナリデータの各バイトを符号なしとしてて扱います。 型: 文字列またはバイナリ	いいえ

名前	説明	必須
	EQ: 等しい。 EQ の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。	
	NE: 等しくない。 NE の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。	

名前	説明	必須
	LE: より小さい、または等しい。 LE の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	LT: より小さい。 LT の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	GE: より大きい、または等しい。 GE の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
	GT: より大きい。 GT の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	
	NOT_NULL : 属性が存在します。	
	NULL: 属性は存在しません。	

名前	説明	必須
	CONTAINS: サブシーケンス、またはセット内の値をチェックします。 CONTAINS の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。比較のターゲット属性が文字列の場合、オペレーションは部分文字列の一致をチェックします。比較のターゲット属性がバイナリの場合、オペレーションは入力に一致するターゲットのサブシーケンスを検索します。比較のターゲット属性がセット (「SS」、「NS」、または「BS」) である場合、オペレーションは、(部分文字列としてではなく) セットの要素をチェックします。	

名前	説明	必須
	NOT_CONTAINS ：サブシーケンスの欠如、またはセット内の値の欠如をチェックします。 NOT_CONTAINS の場合、AttributeValueList は、文字列、数値、またはバイナリのうち、1 つの AttributeValue のみを含むことができます (セットではありません)。比較のターゲット属性が文字列の場合、オペレーションは部分文字列の欠如がないかどうかをチェックします。比較のターゲット属性がバイナリである場合、オペレーションは入力に一致するターゲットのサブシーケンスの欠如がないかどうかをチェックします。比較のターゲット属性がセット (「SS」、「NS」、または「BS」) である場合、オペレーションは、(部分文字列としてではなく) セットの要素の欠如ないかどうかをチェックします。	

名前	説明	必須
	BEGINS_WITH : プレフィックスを確認します。 BEGINS_WITH の場合、AttributeValueList は、文字列またはバイナリの 1 つの AttributeValue のみを含むことができます (数値やセットではありません)。比較のターゲット属性は、(数値またはセットではなく) 文字列またはバイナリである必要があります。	
	IN: 完全一致をチェックします。 IN の場合、AttributeValueList は、文字列、数値、またはバイナリの 1 つ以上の AttributeValue を含むことができます (セットではありません)。比較のターゲット属性は、一致対象と同じ型および同じ値である必要があります。文字列は、文字列セットと一致しません。	

名前	説明	必須
	BETWEEN: 最初の値より大きいまたは等しい、および 2 番目の値より小さいまたは等しい。 BETWEEN の場合、AttributeValueList は、文字列、数値、またはバイナリのいずれかの同じ型の 2 つの AttributeValue 要素を含む必要があります (セットではありません)。ターゲット属性は、ターゲット値が最初の要素より大きいか等しく、2番目の要素より小さいか等しい場合に一致します。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と比較されません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。	

名前	説明	必須
ExclusiveStartKey	以前のスキャンを続行する項目のプライマリキー。以前のスキャンでは、テーブル全体をスキャンする前にスキャンオペレーションが中断された場合、結果セットのサイズまたは Limit パラメータのいずれかにより、この値が提供されることがあります。新しいスキャンリクエストで LastEvaluatedKey を渡し、その時点からオペレーションを続行できます。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement 。	不可

レスポンス

構文

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 229

{"Count":2,"Items":[{
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
    },{
    "AttributeName1":{"S":"AttributeValue4"},
    "AttributeName2":{"S":"AttributeValue5"},
    "AttributeName3":{"S":"AttributeValue6"},
    "AttributeName5":{"B":"dmFsdWU="}
```

```
    }],
    "LastEvaluatedKey":
        {"HashKeyElement":{"S":"AttributeName1"},
        "RangeKeyElement":{"N":"AttributeName2"},
    "ConsumedCapacityUnits":1,
    "ScannedCount":2}
}
```

名前	説明
Items	オペレーションパラメータを満たす属性のコンテナ。 型: 属性名およびそのデータ型と値のマップ。
Count	レスポンス内の項目数。詳細については、「結果での項目のカウント」を参照してください。 型: 数値
ScannedCount	フィルターが適用されるまでの完全なスキャン内の項目の数。大きい ScannedCount 値がほとんどない (またはない) 場合、Count の結果は、不十分なスキャンオペレーションを示します。詳細については、「結果での項目のカウント」を参照してください。 型: 数値
LastEvaluatedKey	スキャンオペレーションが停止した項目のプライマリキー。その時点からオペレーションを続行するには、後続のスキャンオペレーションでこの値を指定します。 スキャン結果セット全体が完了したとき (オペレーションが「最後のページ」を処理したとき)、LastEvaluatedKey は null です。
ConsumedCapacityUnits	オペレーションによって消費される読み込み容量ユニットの数。この値は、プロビジョニング

名前	説明
	されたスループットに適用される数を示します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

エラー	説明
ResourceNotFoundException	指定されたターゲットが見つかりませんでした。

例

AWS SDK を使用した例については、「DynamoDB でのテーブルのスキャン」を参照してください。

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable","ScanFilter":{}}
```

レスポンス例

```
HTTP/1.1 200
x-amzn-RequestId: 4e8a5fa9-71e7-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 465

{"Count":4,"Items":[{
```

```
 "date":{"S":"1980"},
 "fans":{"SS":["Dave","Aaron"]},
 "name":{"S":"Airplane"},
 "rating":{"S":"***"}
 },{
 "date":{"S":"1999"},
 "fans":{"SS":["Ziggy","Laura","Dean"]},
 "name":{"S":"Matrix"},
 "rating":{"S":"*****"}
 },{
 "date":{"S":"1976"},
 "fans":{"SS":["Riley"]},"
 name":{"S":"The Shaggy D.A."},
 "rating":{"S":"**"}
 },{
 "date":{"S":"1985"},
 "fans":{"SS":["Fox","Lloyd"]},
 "name":{"S":"Back To The Future"},
 "rating":{"S":"****"}
 }],
    "ConsumedCapacityUnits":0.5
 "ScannedCount":4}
```

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0
content-length: 125

{"TableName":"comp5",
 "ScanFilter":
  {"time":
   {"AttributeValueList":[{"N":"400"}],
   "ComparisonOperator":"GT"}
 }
}
```

レスポンス例

```
HTTP/1.1 200 OK
```

```
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 262
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":2,
 "Items":[
  {"friends":{"SS":["Dave","Ziggy","Barrie"]},
  "status":{"S":"chatting"},
  "time":{"N":"2000"},
  "user":{"S":"Casey"}},
  {"friends":{"SS":["Dave","Ziggy","Barrie"]},
  "status":{"S":"chatting"},
  "time":{"N":"2000"},
  "user":{"S":"Fredy"}
  }],
"ConsumedCapacityUnits":0.5
"ScannedCount":4
}
```

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB ####
 API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Limit":2,
 "ScanFilter":
  {"time":
   {"AttributeValueList":[{"N":"400"}],
   "ComparisonOperator":"GT"}
 },
 "ExclusiveStartKey":
  {"HashKeyElement":{"S":"Fredy"},"RangeKeyElement":{"N":"2000"}}
}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
```

```
content-type: application/x-amz-json-1.0
content-length: 232
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":1,
 "Items":[
  {"friends":{"SS":["Jane","James","John"]},
  "status":{"S":"exercising"},
  "time":{"N":"2200"},
  "user":{"S":"Roger"}}
 ],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"Riley"},"RangeKeyElement":{"N":"250"}},
"ConsumedCapacityUnits":0.5
"ScannedCount":2
}
```

関連アクション

- Query
- BatchGetItem

UpdateItem

> **Important**
>
> **##########API ##### 2011-12-05 ##**
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

既存の項目の属性を編集します。条件付き更新を実行できます (属性名と値のペアが存在しない場合は新しい属性名と値のペアを挿入し、予期される属性値が存在する場合は既存の名前と値のペアを置き換えます)。

> **Note**
>
> UpdateItem を使用してプライマリキー属性を更新することはできません。その場合、項目を削除し、PutItem を使用して新しい属性で新しい項目を作成します。

UpdateItem オペレーションには、更新の実行方法を定義する `Action` パラメータが含まれます。属性値を配置、削除、または追加できます。

属性値は NULL であってはなりません。文字列型およびバイナリ型属性の長さは 0 より大きくなければなりません。また、セット型属性は空であってはなりません。空の値を持つリクエストは、`ValidationException` で拒否されます。

指定したプライマリキーが既存の項目に存在する場合:

- PUT - 指定した属性を追加します。属性が存在する場合は、新しい値で置き換えられます。
- DELETE - 値を指定しない場合、属性とその値が削除されます。値のセットが指定されている場合、指定されたセット内の値が古いセットから削除されます。したがって、属性値に [a,b,c] が含まれ、削除アクションに [a,c] が含まれる場合、最終的な属性値は [b] です。指定した値の型は、既存の値の型と一致する必要があります。空のセットの指定は無効です。
- ADD — 数値の追加アクションの場合、またはターゲット属性がセット (文字列セットを含む) の場合にのみ使用します。ターゲット属性が単一の文字列値またはスカラーバイナリ値の場合、ADD は機能しません。指定された値は、数値に追加されるか (既存の数値のインクリメントまたはデクリメント)、文字列セット内の追加値として追加されます。値のセットが指定されている場合、値は既存のセットに追加されます。例えば、元のセットが [1,2] で、指定された値が [3] の場合、追加オペレーション後のセットは [4,5] ではなく [1,2,3] になります。セット属性に対して Add アクションが指定されていて、指定した属性の型が既存のセットの型と一致しない場合、エラーが発生します。

  存在しない属性に対して ADD を使用すると、属性とその値が項目に追加されます。

指定したプライマリキーに一致する項目がない場合:

- PUT - 指定されたプライマリキーで新しい項目を作成します。その後、指定された属性を追加します。
- DELETE - 何の処理も行われません。

- ADD — 属性値に対して指定されたプライマリキーと数値 (または数値のセット) で項目を作成します。文字列またはバイナリ型の場合は無効です。

> **Note**
>
> ADD を使用して、更新の前に存在しない項目の数値をインクリメントまたはデクリメントした場合、DynamoDB は初期値として 0 を使用します。また、ADD を使用して項目を更新し、更新の前に (項目は存在しても) 存在しない属性の数値をインクリメントまたはデクリメントした場合、DynamoDB は初期値として 0 を使用します。例えば、ADD を使用して、更新の前に存在しなかった属性に +3 を追加するとします。DynamoDB は初期値に 0 を使用します。更新後の値は 3 です。

このオペレーションの使用に関する詳細については、DynamoDB での項目と属性の操作 を参照してください。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "Key":
        {"HashKeyElement":{"S":"AttributeValue1"},
        "RangeKeyElement":{"N":"AttributeValue2"}},
    "AttributeUpdates":{"AttributeName3":{"Value":
{"S":"AttributeValue3_New"},"Action":"PUT"}},
    "Expected":{"AttributeName3":{"Value":{"S":"AttributeValue3_Current"}}},
    "ReturnValues":"ReturnValuesConstant"
}
```

名前	説明	必須
TableName	更新する項目を含めるテーブルの名前。 型: 文字列	はい
Key	項目を定義するプライマリキー。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: HashKeyElement の値に対するマップと RangeKeyElement の値に対するマップ。	はい
AttributeUpdates	更新のための値、およびアクションに対する属性名マップ。属性名は変更する属性を指定します。プライマリキー属性を含めることはできません。 型: 属性更新のための属性名、値、およびアクション。	
Attribute Updates :Action	更新の実行方法を指定します。使用できる値: PUT (デフォルト)、ADD、または DELETE。セマンティクスについては、UpdateItem の説明を参照してください。 型: 文字列 デフォルト: PUT	いいえ

名前	説明	必須
Expected	条件付き更新の属性を指定します。Expected パラメータを使用すると、属性名を指定できます。また、DynamoDB で更新を行う前に、属性値が既に存在するかどうか、属性値が存在する場合は特定の値があるかどうかの確認の有無も指定できます。 型: 属性名のマップ。	いいえ
Expected:AttributeName	条件付き配置の属性の名前。 型: 文字列	いいえ

名前	説明	必須

Expected:AttributeName: ExpectedAttributeValue	このパラメータを使用して、属性名と値のペアに値が既に存在するかどうかを指定します。
	
	次の JSON 表記法では、項目に「Color」属性がまだ存在しない場合、その項目が更新されます。
	
	`"Expected" : {"Color":{"Exists":false}}`
	
	次の JSON 表記法では、項目を更新する前に、「Color」という名前の属性の既存の値が「Yellow」であるかどうかがチェックされます。
	
	`"Expected" : {"Color":{"Exists":true},{"Value":{"S":"Yellow"}}}`
	
	デフォルトでは、Expected パラメータを使用し、Value を指定した場合、DynamoDB は属性が存在し、置換する現在の値があると仮定します。したがって、{"Exists":true} は暗示されているので指定する必要はありません。リクエストを短縮するには、次のようにします。
		いいえ

名前	説明	必須
	`"Expected" : {"Color":{"Value":{"S":"Yellow"}}}` **Note** チェックする属性値なしで `{"Exists":true}` を指定すると、DynamoDB はエラーを返します。	

名前	説明	必須
ReturnValues	UpdateItem リクエストで属性名と値のペアを更新する前に属性名と値のペアを取得する場合、このパラメータを使用します。使用できるパラメータの値は、NONE (デフォルト) または ALL_OLD、UPDATED_OLD 、ALL_NEW、または UPDATED_NEW です。ALL_OLD を指定し、UpdateItem によって属性名と値のペアが上書きされた場合、古い項目の内容が返されます。このパラメータを指定しない場合、または NONE の場合、何も返されません。ALL_NEW が指定されている場合、新しいバージョンの項目のすべての属性が返されます。UPDATED_NEW が指定されている場合、更新された属性の新しいバージョンのみが返されます。 型: 文字列	不可

レスポンス

構文

次の構文例では、ALL_OLD の ReturnValues パラメータが指定されたリクエストを引き受けます。指定されていない場合、レスポンスには ConsumedCapacityUnits 要素のみが含まれます。

```
HTTP/1.1 200
```

```
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 140

{"Attributes":{
 "AttributeName1":{"S":"AttributeValue1"},
 "AttributeName2":{"S":"AttributeValue2"},
 "AttributeName3":{"S":"AttributeValue3"},
 "AttributeName5":{"B":"dmFsdWU="}
 },
"ConsumedCapacityUnits":1
}
```

名前	説明
Attributes	属性名と値のペアのマップ (ReturnValues パラメータがリクエストの NONE 以外のものとして指定されている場合)。 型: 属性の名前と値のペアのマップ。
ConsumedCapacityUnits	オペレーションによって消費される書き込み容量ユニットの数。この値は、プロビジョニングされたスループットに適用される数を示します。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 数値

特殊なエラー

エラー	説明
ConditionalCheckFailedException	条件チェックに失敗しました。属性 ("+ name +") の値は ("+ name +") ですが、("+ expValue +") が予期されていました。

エラー	説明
ResourceNotFoundExceptions	指定された項目または属性が見つかりませんでした。

例

AWS SDK を使用した例については、「DynamoDB での項目と属性の操作」を参照してください。

リクエスト例

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateItem
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
    "Key":
        {"HashKeyElement":{"S":"Julie"},"RangeKeyElement":{"N":"1307654350"}},
    "AttributeUpdates":
        {"status":{"Value":{"S":"online"},
        "Action":"PUT"}},
    "Expected":{"status":{"Value":{"S":"offline"}}},
    "ReturnValues":"ALL_NEW"
}
```

レスポンス例

```
HTTP/1.1 200 OK
x-amzn-RequestId: 5IMHO7F01Q9P7Q6QMKMMI3R3QRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 121
Date: Fri, 26 Aug 2011 21:05:00 GMT

{"Attributes":
    {"friends":{"SS":["Lynda, Aaron"]},
    "status":{"S":"online"},
    "time":{"N":"1307654350"},
    "user":{"S":"Julie"}},
"ConsumedCapacityUnits":1
```

```
}
```

関連アクション

- PutItem
- DeleteItem

UpdateTable

> **Important**
>
> ***##########API ##### 2011-12-05 ##*******
> 現在の低レベルの API に関するドキュメントについては、Amazon DynamoDB API リファレンスを参照してください。

説明

指定されたテーブルのプロビジョンされたスループットを更新します。テーブルのスループットを設定すると、パフォーマンスの管理に役立ちます。これは、DynamoDB のプロビジョニングされたスループット機能の一部です。詳細については、「DynamoDB プロビジョンドキャパシティモード」を参照してください。

プロビジョニングされたスループット値は、Amazon DynamoDB のクォータ にリストされている最大値と最小値に基づいてアップグレードまたはダウングレードできます。

このオペレーションが正常に完了するには、テーブルが ACTIVE の状態である必要があります。UpdateTable は非同期オペレーションです。オペレーションの実行中、テーブルは UPDATING の状態です。テーブルが UPDATING の状態のとき、テーブルのプロビジョニングされたスループットは呼び出しの前のものです。新しいプロビジョニングされたスループット設定は、UpdateTable オペレーションの後にテーブルが ACTIVE の状態に戻ったときにのみ有効になります。

リクエスト

構文

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
```

```
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

名前	説明	必須
TableName	更新するテーブルの名前です。 型: 文字列	はい
ProvisionedThroughput	指定したテーブルの新しいスループット (ReadCapacityUnits と WriteCapacityUnits の値で構成されます)。「DynamoDB プロビジョンドキャパシティモード」を参照してください。 型: 配列	はい
ProvisionedThroughput :ReadCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される整合性のある ReadCapacityUnits の最小数を設定します。 結果整合性のある読み込みオペレーションに必要な労力は、整合性のある読み込みオペレーションより少ないので、1 秒あたり 50 の	はい

名前	説明	必須
	整合性のある ReadCapacityUnits の設定では、1 秒あたり 100 の結果整合性のある ReadCapacityUnits が実現します。 型: 数値	
ProvisionedThroughput :WriteCapacityUnits	DynamoDB がロードとその他のオペレーションのバランスを取る前に、指定したテーブルで 1 秒あたりに消費される整合性のある WriteCapacityUnits の最小数を設定します 型: 数値	可能

レスポンス

構文

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OOKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
Content-Type: application/json
Content-Length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
    {"CreationDateTime":1.321657838135E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"AttributeValue1","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"AttributeValue2","AttributeType":"N"}},
    "ProvisionedThroughput":
        {"LastDecreaseDateTime":1.321661704489E9,
        "LastIncreaseDateTime":1.321663607695E9,
        "ReadCapacityUnits":5,
```

```
        "WriteCapacityUnits":10},
    "TableName":"Table1",
    "TableStatus":"UPDATING"}}
```

名前	説明
CreationDateTime	テーブルの作成日。 型: 数値
KeySchema	テーブルのプライマリキー (シンプルまたは複合) 構造。HashKeyElement の名前と値のペアは必須です。RangeKeyElement の名前と値のペアはオプションです (複合プライマリキーにのみ必要)。ハッシュキーの最大サイズは 2048 バイトです。範囲キーの最大サイズは 1024 バイトです。両方の制限は別々に強制されます (ハッシュと範囲の合計で 2048 + 1024 キーを組み合わせることができます)。プライマリキーの詳細については、「プライマリキー」を参照してください。 型: 複合プライマリキーの HashKeyElement 、または HashKeyElement および RangeKeyElement のマップ。
ProvisionedThroughput	指定されたテーブルの現在のスループット設定。LastIncreaseDateTime の値 (該当する場合)、LastDecreaseDateTime (該当する場合) を含みます。 型: 配列
TableName	更新されたテーブルの名前。 型: 文字列
TableStatus	テーブルの現在の状態 (CREATING、ACTIVE、DELETING、または

名前	説明
	UPDATING)。UPDATING である必要があります。 テーブルのステータスをチェックするには、DescribeTables オペレーションを使用します。 型: 文字列

特殊なエラー

エラー	説明
ResourceNotFoundException	指定されたターゲットが見つかりませんでした。
ResourceInUseException	テーブルは ACTIVE の状態ではありません。

例

リクエスト例

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB #### API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"comp1",
    "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

レスポンス例

```
HTTP/1.1 200 OK
content-type: application/x-amz-json-1.0
```

```
content-length: 390
Date: Sat, 19 Nov 2011 00:46:47 GMT

{"TableDescription":
    {"CreationDateTime":1.321657838135E9,
    "KeySchema":
        {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
        "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
    "ProvisionedThroughput":
        {"LastDecreaseDateTime":1.321661704489E9,
        "LastIncreaseDateTime":1.321663607695E9,
        "ReadCapacityUnits":5,
        "WriteCapacityUnits":10},
    "TableName":"comp1",
    "TableStatus":"UPDATING"}
}
```

関連アクション

- CreateTable
- DescribeTables
- DeleteTable

レガシー DynamoDB 条件パラメータ

このドキュメントでは、DynamoDB のレガシー条件パラメータの概要を示し、代わりに新しい式パラメータの使用を推奨します。AttributesToGet、AttributeUpdates、ConditionalOperator、Expected、KeyConditions、QueryFilter、Sc などのパラメータの詳細と、新しい式パラメータを代わりに使用する方法の例を示します。

> **Important**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。
> さらに、DynamoDB では、1 つの呼び出しでレガシー条件パラメータと式パラメータを混在させることはできません。例えば、`AttributesToGet` と `ConditionExpression` で `Query` オペレーションを呼び出すとエラーになります。

次の表は、これらのレガシーパラメータを引き続きサポートする DynamoDB API オペレーション、および代替として使用が推奨される式パラメータを示しています。このテーブルは、式パラメータを使用するようにアプリケーションを更新することを検討している場合に役立ちます。

この API オペレーションを使用すると...	これらのレガシーパラメータでは...	代わりにこの式パラメータを使用
BatchGetItem	AttributesToGet	ProjectionExpression
DeleteItem	Expected	ConditionExpression
GetItem	AttributesToGet	ProjectionExpression
PutItem	Expected	ConditionExpression
Query	AttributesToGet	ProjectionExpression
	KeyConditions	KeyConditionExpression
	QueryFilter	FilterExpression
Scan	AttributesToGet	ProjectionExpression
	ScanFilter	FilterExpression
UpdateItem	AttributeUpdates	UpdateExpression
	Expected	ConditionExpression

以下のセクションでは、レガシー条件パラメータについて詳しく説明します。

トピック

- AttributesToGet (レガシー)
- AttributeUpdates (レガシー)
- ConditionalOperator (レガシー)
- Expected (レガシー)
- KeyConditions (レガシー)

- QueryFilter (レガシー)
- ScanFilter (レガシー)
- レガシーパラメータを使用した条件の記述

AttributesToGet (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに ProjectionExpression を使用」を参照してください。

レガシー条件パラメータ `AttributesToGet` は、DynamoDB から取得する 1 つ以上の属性の配列です。属性名が指定されていない場合、すべての属性が返されます。リクエストした属性が見つからない場合、その属性は結果に表示されません。

`AttributesToGet` では、リストまたはマップ型の属性を取得できますが、リストまたはマップ内の個々の要素を取得することはできません。

`AttributesToGet` は、プロビジョニングされたスループットの消費には影響しません。DynamoDB は、アプリケーションに返されるデータ量ではなく、項目のサイズに基づいて、消費される読み込み容量ユニットの数を決定します。

代わりに ProjectionExpression を使用 - 例

Music テーブルから項目を取得する際、一部の属性のみが返されるようにする場合を考えてみます。次の AWS CLI の例に示すように、`AttributesToGet` パラメータを含む `GetItem` リクエストを使用できます。

```
aws dynamodb get-item \
    --table-name Music \
    --attributes-to-get '["Artist", "Genre"]' \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }'
```

代わりに ProjectionExpression を使用できます。

```
aws dynamodb get-item \
    --table-name Music \
    --projection-expression "Artist, Genre" \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }'
```

AttributeUpdates (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに UpdateExpression を使用します。」を参照してください。

UpdateItem オペレーションでは、レガシー条件パラメータ AttributeUpdates に、変更する属性の名前、それぞれに対して実行するアクション、およびそれぞれの新しい値が含まれます。そのテーブル上のインデックスのインデックスキー属性である属性を更新する場合、属性の型は、テーブルの説明の AttributesDefinition で定義されたインデックスキーの型に一致する必要があります。UpdateItem を使用して非キー属性を更新できます。

属性値を null にすることはできません。文字列型およびバイナリ型属性の長さは、ゼロより大きくなければなりません。セット型の属性を空にすることはできません。空の値を持つリクエストは、ValidationException 例外で拒否されます。

各 AttributeUpdates 要素は、変更する属性名と次の属性で構成されます。

- Value - この属性の新しい値 (該当する場合)。
- Action - 更新の実行方法を指定する値。このアクションは、データ型が数値またはセットである既存の属性に対してのみ有効です。ADD は他のデータ型に使用しないでください。

  指定したプライマリキーを持つ項目がテーブル内に見つかった場合、次の値によって以下のアクションが実行されます。

- PUT - 指定した属性を項目に追加します。属性がすでに存在する場合は、新しい値で置き換えられます。
- DELETE - の値が指定されていない場合、属性とその値を削除します。DELETE指定した値のデータ型は、既存の値のデータ型と一致する必要があります。

  値のセットが指定されている場合、それらの値が古いセットから削除されます。例えば、属性値が [a,b,c] のセットの場合に DELETE アクションで [a,c] を指定すると、最終的な属性値は [b] になります。空のセットを指定するとエラーになります。
- ADD - 属性が存在しない場合、指定された値を項目に追加します。属性が存在する場合、ADD の動作は属性のデータ型によって決まります。
  - 既存の属性が数値で、Value も数値である場合、Value は既存の属性に数学的に追加されます。Value が負の数値である場合は、既存の属性から削除されます。

    > **Note**
    >
    > ADD を使用して、更新の前に存在しない項目の数値をインクリメントまたはデクリメントした場合、DynamoDB は初期値として 0 を使用します。
    > 同様に、更新の前に存在しない属性値をインクリメントまたはデクリメントするために既存の項目に対して ADD を使用した場合、DynamoDB は初期値として 0 を使用します。例えば、itemcount という属性を持たない項目を更新する際に ADD を使用して、この属性に 3 を追加する場合を考えています。この場合、DynamoDB によって itemcount 属性が作成され、その初期値が 0 に設定されます。最後に 3 が追加されます。結果は、3 の値を含む新しい itemcount 属性になります。
  - 既存のデータ型がセットで、Value もセットである場合、Value が既存のセットに追加されます。例えば、属性値が [1,2] のセットで、ADD アクションで [3] が指定されている場合、最終的な属性値は [1,2,3] になります。セット属性に対して ADD アクションが指定されていて、指定した属性の型が既存のセット型に一致しない場合、エラーが発生します。

    両方のセットは同じプリミティブデータ型を持つ必要があります。例えば、既存のデータ型が文字列のセットである場合、Value も文字列のセットである必要があります。

指定されたキーを持つ項目がテーブルに見つからない場合、次の値は次のアクションを実行します。

- PUT - DynamoDB で、指定されたプライマリキーで新しい項目が作成され、属性が追加されます。

- DELETE - 存在しない項目から属性を削除することはできないので何の処理も行われません。オペレーションは成功しますが、DynamoDB は新しい項目を作成しません。
- ADD - DynamoDB で、指定されたプライマリキーと属性値の番号 (または数字のセット) を持つ項目を作成します。使用できるデータ型は、数値型と数値セット型のみです。

インデックスキーの一部である属性を指定する場合、それらの属性のデータ型は、テーブルの属性定義内のスキーマのデータ型と一致する必要があります。

代わりに UpdateExpression を使用 - 例

Music テーブル内の項目を変更する場合を考えてみます。次の AWS CLI の例に示すように、AttributeUpdates パラメータを含む UpdateItem リクエストを使用できます。

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "SongTitle": {"S":"Call Me Today"},
        "Artist": {"S":"No One You Know"}
    }' \
    --attribute-updates '{
        "Genre": {
            "Action": "PUT",
            "Value": {"S":"Rock"}
        }
    }'
```

代わりに UpdateExpression を使用できます。

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "SongTitle": {"S":"Call Me Today"},
        "Artist": {"S":"No One You Know"}
    }' \
    --update-expression 'SET Genre = :g' \
    --expression-attribute-values '{
        ":g": {"S":"Rock"}
    }'
```

ConditionalOperator (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。

レガシー条件パラメータ `ConditionalOperator` は、`Expected`、`QueryFilter`、または `ScanFilter` マップ内の条件に適用するために使用される論理演算子です。

- AND - すべての条件が true に評価された場合、マップ全体が true に評価されます。
- OR - 条件の少なくとも 1 つが true に評価された場合、マップ全体が true に評価されます。

`ConditionalOperator` を省略した場合、`AND` がデフォルトです。

オペレーションは、マップ全体が true と評価された場合にのみ成功します。

> **Note**
>
> このパラメータは、リストおよびマップ型の属性をサポートしません。

Expected (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに ConditionExpression を使用します。」を参照してください。

レガシー条件パラメータ `Expected` は `UpdateItem` オペレーションの条件付きブロックです。`Expected` は属性/条件ペアのマップです。マップの各要素は、属性名、比較演算子、および 1 つ以上の値で構成されます。DynamoDB は、比較演算子を使用して、指定した値と属性を比較します。各 `Expected` 要素に対して、評価の結果は true または false のいずれかです。

Expected マップで複数の要素を指定すると、デフォルトでは、すべての条件が true に評価される必要があります。つまり、条件は AND 演算子を使用して結合されます (ConditionalOperator パラメータを使用して条件を OR に設定できます。その場合、すべての条件ではなく、少なくとも 1 つの条件が true に評価される必要があります)。

Expected マップが true に評価された場合、オペレーションは成功します。それ以外の場合、オペレーションは失敗します。

Expected には以下の要素が含まれます。

- AttributeValueList - 指定された属性に対して評価する 1 つ以上の値。リストの値の数は、使用される ComparisonOperator エンジンによって異なります。

  数値型の場合、値の比較は数値です。

  より大きい、等しい、またはより小さいの文字列値の比較は、UTF-8 バイナリエンコーディングの Unicode に基づきます。例えば、a は A より大きく、a は B より大きいと評価されます。

  バイナリの場合、DynamoDB がバイナリ値を比較する際、バイナリデータの各バイトは符号なしとして扱われます。

- ComparisonOperator - の属性を評価するためのコンパレータ。AttributeValueList比較を実行する際、DynamoDB は強力な整合性のある読み込みを使用します。

  次の比較演算子がサポートされています。

  EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN

  各比較演算子の説明は、以下のとおりです。

  - EQ: 等しい。EQ は、リストやマップなど、すべてのデータ型でサポートされます。

    AttributeValueList には、文字列、数値、バイナリ、文字列セット、数値セット、またはバイナリセットの方の 1 つの AttributeValue 要素のみを含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。

  - NE: 等しくない。NE は、リストやマップを始めとするすべてのデータ型でサポートされています。

AttributeValueList には、文字列、数値、バイナリ、文字列セット、数値セット、またはバイナリセットの型の 1 つの AttributeValue のみを含めることができます。項目に含まれる AttributeValue の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。

- LE: より小さい、または等しい。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- LT: より小さい。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- GE: より大きい、または等しい。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- GT: より大きい。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- NOT_NULL: 属性が存在します。NOT_NULL は、リストやマップを始めとするすべてのデータ型でサポートされています。

> **Note**
>
> この演算子は、データ型ではなく、属性が存在することをテストします。NOT_NULL を使用して属性のデータ型「a」を評価する場合、結果はブール値の true です。この結果理由は、属性「a」が存在するからです。そのデータ型は NOT_NULL 比較演算子に関係ありません。

- NULL: 属性は存在しません。NULL は、リストやマップを始めとするすべてのデータ型でサポートされています。

> **Note**
>
> この演算子は、データ型ではなく、属性が存在しないことをテストします。a を使用して属性のデータ型「NULL」を評価する場合、結果はブール値の false です。この理由は、属性「a」が存在するからです。そのデータ型は NULL 比較演算子に関係ありません。

- CONTAINS: サブシーケンス、またはセット内の値をチェックします。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。比較のターゲット属性が文字列型である場合、演算子は部分文字列の一致をチェックします。比較のターゲット属性が型バイナリの場合、演算子は入力に一致するターゲットのサブシーケンスを検索します。比較のターゲット属性がセット (「SS」、「NS」、または「BS」) の場合、セットのいずれかの要素との完全一致が見つかったときに true に評価されます。

  CONTAINS はリストでサポートされています。「a CONTAINS b」を評価するとき、「a」はリストである可能性があります。しかし、「b」が、セット、マップ、およびリストになることはありません。

- NOT_CONTAINS: サブシーケンスの欠如、またはセット内の値の欠如をチェックします。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。比較のターゲット属性が文字列の場合、演算子は部分文字列の一致の欠如をチェックします。比較のターゲット属性がバイナリである場合、演算子は、入力に一致するターゲットのサブシーケンスの欠如をチェックします。比較のターゲット属性がセット (「SS」、「NS」、または「BS」) のとき、セットのいずれか要素の完全一致が見つからdoes not場合、true に評価されます。

NOT_CONTAINS はリストでサポートされています。「a NOT CONTAINS b」を評価するとき、「a」はリストである可能性があります。しかし、「b」は、セット、マップ、およびリストになることはありません。

- BEGINS_WITH: プレフィックスを確認します。

  AttributeValueList には、文字列またはバイナリ型の 1 つの AttributeValue のみを含めることができます (数値やセットではありません)。比較のターゲット属性は、文字列型またはバイナリ型である必要があります (数値型またはセット型ではありません)。

- IN: 2 つのセット内で一致する要素をチェックします。

  AttributeValueList には、文字列、数値、またはバイナリ型の 1 つ以上の AttributeValue 要素を含むことができます (セット型ではありません)。これらの属性は、項目の既存のセット型属性と比較されます。入力セットのいずれかの要素が項目の属性に存在する場合、式は true に評価されます。

- BETWEEN: 最初の値よりも大きいか等しく、2 番目の値よりも小さいか等しくなります。

  AttributeValueList には、文字列、数値、またはバイナリ (セットではありません) のいずれかの同じ型の 2 つの AttributeValue 要素を含む必要があります。ターゲット属性は、ターゲット値が最初の要素より大きいか等しく、2番目の要素より小さいか等しい場合に一致します。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と比較されません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

次のパラメータは、AttributeValueList および ComparisonOperator の代わりに使用できます。

- Value - DynamoDB が属性と比較する値。
- Exists - 条件付きオペレーションを実行する前に DynamoDB が値を評価するブール値。
  - Exists が true の場合、DynamoDB は、その属性値がテーブル内にすでに存在するかどうかを確認します。見つかった場合、条件は true に評価されます。それ以外の場合、条件は false に評価されます。
  - Exists が false の場合、DynamoDB は属性値がテーブル内に存在notことが予期されます。実際に値が存在しない場合、仮定は有効であり、条件は true に評価されます。存在しないという仮定にもかかわらず値が見つかった場合、条件は false に評価されます。

  Exists のデフォルト値は true です。

`Value` パラメータと `Exists` パラメータは、`AttributeValueList` および `ComparisonOperator` と互換性がありません。両方のパラメータセットを同時に使用すると、DynamoDB は `ValidationException` 例外を返します。

> **Note**
>
> このパラメータは、リストおよびマップ型の属性をサポートしません。

代わりに ConditionExpression を使用 - 例

特定の条件が true の場合のみに Music テーブル内の項目を変更する場合を考えてみます。次の AWS CLI の例に示すように、`Expected` パラメータを含む `UpdateItem` リクエストを使用できます。

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }' \
    --attribute-updates '{
        "Price": {
            "Action": "PUT",
            "Value": {"N":"1.98"}
        }
    }' \
    --expected '{
        "Price": {
            "ComparisonOperator": "LE",
            "AttributeValueList": [ {"N":"2.00"} ]
        }
    }'
```

代わりに `ConditionExpression` を使用できます。

```
aws dynamodb update-item \
    --table-name Music \
    --key '{
        "Artist": {"S":"No One You Know"},
        "SongTitle": {"S":"Call Me Today"}
    }' \
```

```
    --update-expression 'SET Price = :p1' \
    --condition-expression 'Price <= :p2' \
    --expression-attribute-values '{
        ":p1": {"N":"1.98"},
        ":p2": {"N":"2.00"}
    }'
```

KeyConditions (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに KeyConditionExpression を使用します。」を参照してください。

レガシー条件パラメータ KeyConditions には、Query オペレーションの選択基準が含まれています。テーブルに対するクエリでは、テーブルのプライマリキー属性に対してのみ条件を設定できます。EQ 条件としてパーティションキーの名前と値を指定する必要があります。ソートキーを参照する 2 番目の条件を指定できます (オプション)。

> **Note**
>
> ソートキー条件を指定しない場合、パーティションキーに一致するすべての項目が取得されます。FilterExpression または QueryFilter が存在する場合、これは項目が取得された後に適用されます。

インデックスに対するクエリでは、インデックスキー属性に対してのみ条件を設定できます。EQ 条件としてインデックスパーティションキーの名前と値を指定する必要があります。インデックスソートキーを参照する 2 番目の条件を指定できます (オプション)。

各 KeyConditions 要素は、比較する属性名と次の属性で構成されます。

- AttributeValueList - 指定された属性に対して評価する 1 つ以上の値。リストの値の数は、使用される ComparisonOperator エンジンによって異なります。

  数値型の場合、値の比較は数値です。

より大きい、等しい、またはより小さいの文字列値の比較は、UTF-8 バイナリエンコーディングの Unicode に基づきます。例えば、a は A より大きく、a は B より大きいと評価されます。

バイナリの場合、DynamoDB がバイナリ値を比較する際、バイナリデータの各バイトは符号なしとして扱われます。

- ComparisonOperator - 属性を評価するためのコンパレータ。例えば、等しい、より大きい、より小さいなどです。

  KeyConditions の場合、次の比較演算子がサポートされています。

  EQ | LE | LT | GE | GT | BEGINS_WITH | BETWEEN

  これらの比較演算子の説明を次に示します。

  - EQ: 等しい。

    AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue のみ含めることができます。リクエストで指定されているもの以外の型の AttributeValue 要素が項目に含まれる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と等しくありません。

  - LE: より小さい、または等しい。

    AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

  - LT: より小さい。

    AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

  - GE: より大きい、または等しい。

AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- GT: より大きい。

  AttributeValueList には、文字列、数値、またはバイナリの型 (セット型ではありません) の 1 つの AttributeValue 要素のみ含めることができます。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と等しくありません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

- BEGINS_WITH: プレフィックスを確認します。

  AttributeValueList には、文字列またはバイナリ型の 1 つの AttributeValue のみを含めることができます (数値やセットではありません)。比較のターゲット属性は、文字列型またはバイナリ型である必要があります (数値型またはセット型ではありません)。

- BETWEEN: 最初の値よりも大きいか等しく、2 番目の値よりも小さいか等しくなります。

  AttributeValueList には、文字列、数値、またはバイナリ (セットではありません) のいずれかの同じ型の 2 つの AttributeValue 要素を含む必要があります。ターゲット属性は、ターゲット値が最初の要素より大きいか等しく、2番目の要素より小さいか等しい場合に一致します。項目に含まれる AttributeValue 要素の型がリクエストで指定されている型と異なる場合、値は一致しません。例えば、{"S":"6"} は {"N":"6"} と比較されません。また、{"N":"6"} は {"NS":["6", "2", "1"]} と比較されません。

代わりに KeyConditionExpression を使用 - 例

同じパーティションキーを持つ複数の項目を Music テーブルから取得する場合を考えてみます。次の AWS CLI の例に示すように、Query パラメータを含む KeyConditions リクエストを使用できます。

```
aws dynamodb query \
    --table-name Music \
    --key-conditions '{
        "Artist":{
            "ComparisonOperator":"EQ",
```

```
            "AttributeValueList": [ {"S": "No One You Know"} ]
        },
        "SongTitle":{
            "ComparisonOperator":"BETWEEN",
            "AttributeValueList": [ {"S": "A"}, {"S": "M"} ]
        }
    }'
```

代わりに KeyConditionExpression を使用できます。

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression 'Artist = :a AND SongTitle BETWEEN :t1 AND :t2' \
    --expression-attribute-values '{
        ":a": {"S": "No One You Know"},
        ":t1": {"S": "A"},
        ":t2": {"S": "M"}
    }'
```

QueryFilter (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに FilterExpression を使用します。」を参照してください。

Query オペレーションでは、レガシー条件パラメータ QueryFilter は、項目が読み込まれた後にクエリ結果を評価し、目的の値のみを返す条件です。

このパラメータは、リストおよびマップ型の属性をサポートしません。

> **Note**
>
> QueryFilter は、項目が読み込まれた後に適用されます。フィルタリングのプロセスでは、追加の読み込み容量ユニットは消費されません。

QueryFilter マップで複数の条件を指定すると、デフォルトでは、すべての条件が true に評価される必要があります。つまり、条件は AND 演算子を使用して結合されます (ConditionalOperator (レガシー) パラメータを使用して条件を OR に設定できます。その場合、すべての条件ではなく、少なくとも 1 つの条件が true に評価される必要があります)。

QueryFilter ではキー属性を使用できません。パーティションキーまたはソートキーでフィルター条件を定義することはできません。

各 QueryFilter 要素は、比較する属性名と次の属性で構成されます。

- AttributeValueList - 指定された属性に対して評価する 1 つ以上の値。リスト内の値の数は、ComparisonOperator で指定されている演算子により異なります。

  数値型の場合、値の比較は数値です。

  より大きい、等しい、より小さい文字列値の比較は、UTF-8 バイナリエンコーディングに基づいています。例えば、a は A より大きく、a は B より大きいと評価されます。

  バイナリの場合、DynamoDB がバイナリ値を比較する際、バイナリデータの各バイトは符号なしとして扱われます。

  JSON でのデータ型指定については、「DynamoDB 低レベル API」を参照してください。
- ComparisonOperator - 属性を評価するためのコンパレータ。例えば、等しい、より大きい、より小さいなどです。

  次の比較演算子がサポートされています。

  EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN

代わりに FilterExpression を使用 - 例

Music テーブルをクエリし、一致する項目に条件を適用する場合を考えてみます。次の AWS CLI の例に示すように、Query パラメータを含む QueryFilter リクエストを使用できます。

```
aws dynamodb query \
    --table-name Music \
    --key-conditions '{
        "Artist": {
            "ComparisonOperator": "EQ",
            "AttributeValueList": [ {"S": "No One You Know"} ]
```

```
        }
    }' \
    --query-filter '{
        "Price": {
            "ComparisonOperator": "GT",
            "AttributeValueList": [ {"N": "1.00"} ]
        }
    }'
```

代わりに FilterExpression を使用できます。

```
aws dynamodb query \
    --table-name Music \
    --key-condition-expression 'Artist = :a' \
    --filter-expression 'Price > :p' \
    --expression-attribute-values '{
        ":p": {"N":"1.00"},
        ":a": {"S":"No One You Know"}
    }'
```

ScanFilter (レガシー)

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。このパラメータに代わる新しいパラメータの具体的な情報については、「代わりに FilterExpression を使用します。」を参照してください。

Scan オペレーションでは、レガシー条件パラメータ ScanFilter は、スキャン結果を評価し、目的の値のみを返す条件です。

> **Note**
>
> このパラメータは、リストおよびマップ型の属性をサポートしません。

複数の条件を指定すると、ScanFilter マップの場合、デフォルトでは、すべての条件が true に評価される必要があります。つまり、すべての条件が一緒に AND 処理されます。(ConditionalOperator

(レガシー) パラメータを使用して条件を OR に設定できます。その場合、すべての条件ではなく、少なくとも 1 つの条件が true に評価される必要があります)。

各 ScanFilter 要素は、比較する属性名と次の属性で構成されます。

- AttributeValueList - 指定された属性に対して評価する 1 つ以上の値。リスト内の値の数は、ComparisonOperator で指定されている演算子により異なります。

  数値型の場合、値の比較は数値です。

  より大きい、等しい、より小さい文字列値の比較は、UTF-8 バイナリエンコーディングに基づいています。例えば、a は A より大きく、a は B より大きいと評価されます。

  バイナリの場合、DynamoDB がバイナリ値を比較する際、バイナリデータの各バイトは符号なしとして扱われます。

  JSON でのデータ型指定については、「DynamoDB 低レベル API」を参照してください。
- ComparisonOperator - 属性を評価するためのコンパレータ。例えば、等しい、より大きい、より小さいなどです。

  次の比較演算子がサポートされています。

  EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN

代わりに FilterExpression を使用 - 例

Music テーブルをスキャンし、一致する項目に条件を適用する場合を考えてみます。次の AWS CLI の例に示すように、Scan パラメータを含む ScanFilter リクエストを使用できます。

```
aws dynamodb scan \
    --table-name Music \
    --scan-filter '{
        "Genre":{
            "AttributeValueList":[ {"S":"Rock"} ],
            "ComparisonOperator": "EQ"
        }
    }'
```

代わりに FilterExpression を使用できます。

```
aws dynamodb scan \
    --table-name Music \
    --filter-expression 'Genre = :g' \
    --expression-attribute-values '{
        ":g": {"S":"Rock"}
    }'
```

レガシーパラメータを使用した条件の記述

> **Note**
>
> 可能な限り、これらのレガシーパラメータの代わりに新しい式パラメータを使用することをお勧めします。詳細については、「DynamoDB での式の使用」を参照してください。

このセクションでは、Expected、QueryFilter、ScanFilter などのレガシーパラメータで使用するための条件を書き込む方法について説明します。

> **Note**
>
> 新しいアプリケーションでは、式パラメータを使用する必要があります。詳細については、「DynamoDB での式の使用」を参照してください。

シンプルな条件

属性値を使用すると、テーブル属性との比較条件を記述できます。条件は常にtrueまたはfalseに評価され、以下のもので構成されます。

- ComparisonOperator — より大きい、より小さい、等しいなど。
- AttributeValueList (オプション) — 比較対象の属性値。使用する ComparisonOperator に応じて、AttributeValueList には、1 つ、2 つ、またはそれ以上の値が含まれることがあります。まったく存在しないこともあります。

このセクションでは、さまざまな比較演算子と、条件での使用方法の例について説明します。

属性値を含まない比較演算子

- NOT_NULL - 属性が存在する場合は true です。

- NULL - 属性が存在しない場合は true です。

これらの演算子を使用して、属性が存在するか、存在しないかをチェックします。比較する値がないので、AttributeValueList は指定しません。

例

Dimensions 属性が存在する場合、次の式は true と評価されます。

```
...
    "Dimensions": {
         ComparisonOperator: "NOT_NULL"
    }
...
```

属性値を 1 つ含む比較演算子

- EQ - 属性が値と等しい場合、true です。

  AttributeValueList には、文字列、数値、バイナリ、文字列セット、数値セット、またはバイナリセットの型の 1 つの値のみを含めることができます。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。例えば、文字列 "3" は数値 3 と等しくありません。また、数値 3 は数値セット [3, 2, 1] と等しくありません。
- NE - 属性が値と等しくない場合、true です。

  AttributeValueList には、文字列、数値、バイナリ、文字列セット、数値セット、またはバイナリセットの型の 1 つの値のみを含めることができます。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。
- LE - 属性が値より小さいか、等しい場合、true です。

  AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の AttributeValue が項目に含まれる場合、値は一致しません。
- LT - 属性が値より小さい場合、true です。

  AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。
- GE - 属性が値より大きいか、等しい場合、true です。

AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。

- GT - 属性が値より大きい場合、true です。

  AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。

- CONTAINS - 値がセット内に存在する場合、またはある値に別の値が含まれている場合、true です。

  AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。比較のターゲット属性が文字列の場合、演算子は部分文字列の一致をチェックします。比較のターゲット属性がバイナリの場合、演算子は入力に一致するターゲットのサブシーケンスを検索します。比較のターゲット属性がセットの場合、セットのいずれかの要素との完全一致が見つかると、演算子は true と評価されます。

- NOT_CONTAINS - 値がセット内に存在しない場合、またはある値に別の値が含まれている場合、true です。

  AttributeValueList は、文字列、数値、またはバイナリのうち 1 つの値のみを含むことができます (セットではありません)。比較のターゲット属性が文字列の場合、演算子は部分文字列の一致の欠如をチェックします。比較のターゲット属性がバイナリである場合、演算子は、入力に一致するターゲットのサブシーケンスの欠如をチェックします。比較のターゲット属性がセットの場合、セットのいずれかの要素との完全一致が見つからないと、演算子は true と評価されます。

- BEGINS_WITH - 属性の最初の数文字が指定された値と一致する場合、true です。この演算子は数値の比較に使用しないでください。

  AttributeValueList には、文字列またはバイナリ型の 1 つの値のみを含めることができます (数値やセットではありません)。比較のターゲット属性は、(数値またはセットではなく) 文字列またはバイナリである必要があります。

これらの演算子を使用して、属性と値を比較します。1 つの値で構成される AttributeValueList を指定する必要があります。ほとんどの演算子では、この値はスカラーである必要があります。ただし、EQ 演算子と NE 演算子はセットもサポートします。

例

以下の条件が満たされる場合、次の式は true と評価されます。

- 商品の価格が 100 より大きい。

```
...
    "Price": {
        ComparisonOperator: "GT",
        AttributeValueList: [ {"N":"100"} ]
    }
...
```

- 商品カテゴリが「Bo」で始まる。

```
...
    "ProductCategory": {
        ComparisonOperator: "BEGINS_WITH",
        AttributeValueList: [ {"S":"Bo"} ]
    }
...
```

- 製品には、赤、緑、または黒がある。

```
...
    "Color": {
        ComparisonOperator: "EQ",
        AttributeValueList: [
            [ {"S":"Black"}, {"S":"Red"}, {"S":"Green"} ]
        ]
    }
...
```

> **Note**
>
> セットデータ型を比較する場合、要素の順序は関係ありません。DynamoDB は、リクエストで指定した順序に関係なく、同じ値のセットを持つ項目のみを返します。

属性値を 2 つ含む比較演算子

- BETWEEN - 値が下限と上限の間にある場合 (エンドポイントを含む) は true です。

AttributeValueList には、文字列、数値、またはバイナリ (セットではありません) のいずれかの同じ型の 2 つの要素を含む必要があります。ターゲット属性は、ターゲット値が最初の要素より大きいか等しく、2番目の要素より小さいか等しい場合に一致します。リクエストで指定されているもの以外の型の値が項目に含まれる場合、値は一致しません。

この演算子を使用して、属性値が範囲内にあるかどうかを判断します。AttributeValueList は、同じ型 (文字列、数値、またはバイナリ) の 2 つのスカラー要素を含む必要があります。

例

製品の料金が 100 ~ 200 の場合、次の表現は true と評価されます。

```
...
    "Price": {
        ComparisonOperator: "BETWEEN",
        AttributeValueList: [ {"N":"100"}, {"N":"200"} ]
    }
...
```

属性値 N を含む比較演算子

- IN - 値が列挙リスト内のいずれかの値と等しい場合、true です。リストではスカラー値のみがサポートされ、セットはサポートされません。一致するには、ターゲット属性は、同じ型と値である必要があります。

  AttributeValueList は、文字列、数値、またはバイナリの 1 つ以上の要素を含むことができます (セットではありません)。これらの属性は、項目の既存の非セット型属性と比較されます。入力セットのいずれかの要素が項目の属性に存在する場合、式は true に評価されます。

  AttributeValueList は、文字列、数値、またはバイナリの 1 つまたは複数の値を含むことができます (セットではありません)。比較のターゲット属性は、一致対象と同じ型および同じ値である必要があります。文字列は、文字列セットと一致しません。

この演算子を使用して、指定された値が列挙リスト内にあるかどうかを判断します。AttributeValueList ではスカラー値はいくつでも指定できますが、すべて同じデータ型である必要があります。

例

ID の値が 201、203、205 の場合、次の式は true と評価されます。

```
...
    "Id": {
        ComparisonOperator: "IN",
        AttributeValueList: [ {"N":"201"}, {"N":"203"}, {"N":"205"} ]
    }
...
```

複数の条件の使用

DynamoDB では、複数の条件を組み合わせて、複雑な表現を作成できます。これを行うには、オプションの ConditionalOperator (レガシー) を使用して、少なくとも 2 つの式を指定します。

デフォルトでは、複数の条件を指定した場合、式全体が true に評価されるには、すべての条件が true に評価される必要があります。言い換えると、暗黙の AND オペレーションが行われます。

例

商品が 600 ページ以上の本の場合、次の表現は true と評価されます。どちらの条件も暗黙的に AND で処理されるので、true と評価される必要があります。

```
...
    "ProductCategory": {
        ComparisonOperator: "EQ",
        AttributeValueList: [ {"S":"Book"} ]
    },
    "PageCount": {
        ComparisonOperator: "GE",
        AttributeValueList: [ {"N":600} ]
    }
...
```

ConditionalOperator (レガシー) を使用して、AND オペレーションが行われることを明確にすることができます。次の例は、前の例と同様に動作します。

```
...
    "ConditionalOperator" : "AND",
    "ProductCategory": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"N":"Book"} ]
    },
```

```
    "PageCount": {
        "ComparisonOperator": "GE",
        "AttributeValueList": [ {"N":600"} ]
    }
...
```

ConditionalOperator を OR に設定することもできます。その場合、少なくとも 1 つの条件が true に評価される必要があります。

例

製品がマウンテンバイクの場合、特定のブランド名である場合、または価格が 100 より大きい場合、次の表現は true と評価されます。

```
...
    ConditionalOperator : "OR",
    "BicycleType": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S":"Mountain" ]
    },
    "Brand": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S":"Brand-Company A" ]
    },
    "Price": {
        "ComparisonOperator": "GT",
        "AttributeValueList": [ {"N":"100"} ]
    }
...
```

> **Note**
>
> 複雑な表現では、条件は最初の条件から最後の条件まで順番に処理されます。
> 1 つの表現で AND と OR の両方を使用することはできません。

その他の条件演算子

DynamoDB の以前のリリースでは、Expected パラメータの動作は、条件付き書き込みでは異なりました。Expected マップ内の各項目は、以下とともに、チェックする DynamoDB の属性名を表していました。

- Value - 属性に対して比較する値。
- Exists — オペレーションを試みる前に値が存在するかどうかを調べます。

Value と Exists のオプションは、引き続き DynamoDB でサポートされていますが、等価条件または属性が存在するかどうかのテストのみが可能です。ComparisonOperator と AttributeValueList オプションを使用すると、より広い範囲の条件を構築できるので、これらのオプションを使用することをお勧めします。

Example

DeleteItem は、本が出版されていないかどうかを確認し、この条件が true である場合にのみ削除できます。レガシー条件を使用する AWS CLI の例を次に示します。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{
        "Id": {"N":"600"}
    }' \
    --expected '{
        "InPublication": {
            "Exists": true,
            "Value": {"BOOL":false}
        }
    }'
```

次の例では、同じことを実行しますが、レガシー条件は使用しません。

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{
        "Id": {"N":"600"}
    }' \
    --expected '{
        "InPublication": {
            "ComparisonOperator": "EQ",
            "AttributeValueList": [ {"BOOL":false} ]
        }
    }'
```

Example

PutItem オペレーションは、同じプライマリキー属性を持つ既存の項目を上書きしないように保護できます。レガシー条件を使用する例を次に示します。

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item '{
      "Id": {"N":"500"},
        "Title": {"S":"Book 500 Title"}
    }' \
    --expected '{
        "Id": { "Exists": false }
    }'
```

次の例では、同じことを実行しますが、レガシー条件は使用しません。

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item '{
      "Id": {"N":"500"},
        "Title": {"S":"Book 500 Title"}
    }' \
    --expected '{
        "Id": { "ComparisonOperator": "NULL" }
    }'
```

> **Note**
>
> Expected マップ内の条件には、レガシーの Value オプションと Exists オプションを ComparisonOperator と AttributeValueList と一緒に使用しないでください。一緒に使用すると、条件付き書き込みは失敗します。