
Amazon DynamoDB

Developer Guide

API Version 2012-08-10



Amazon DynamoDB: Developer Guide

Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

| | |
|--|----|
| What Is Amazon DynamoDB | 1 |
| High Availability and Durability | 1 |
| Getting Started with DynamoDB | 1 |
| How It Works | 2 |
| Core Components | 2 |
| DynamoDB API | 10 |
| Naming Rules and Data Types | 12 |
| Read Consistency | 16 |
| Read/write Capacity Mode | 17 |
| Partitions and Data Distribution | 21 |
| From SQL to NoSQL | 23 |
| Relational or NoSQL? | 24 |
| Characteristics of Databases | 25 |
| Creating a Table | 27 |
| Getting Information About a Table | 29 |
| Writing Data to a Table | 30 |
| Reading Data from a Table | 32 |
| Managing Indexes | 37 |
| Modifying Data in a Table | 40 |
| Deleting Data from a Table | 42 |
| Removing a Table | 43 |
| Additional Amazon DynamoDB Resources | 44 |
| Blog Posts, Repositories, and Guides | 44 |
| Data Modeling and Design Patterns | 44 |
| Advanced Design Patterns with Rick Houlihan | 44 |
| Training Courses | 44 |
| Tools for Coding and Visualization | 45 |
| Setting Up DynamoDB | 46 |
| Setting Up DynamoDB Local (Downloadable Version) | 46 |
| Deploying DynamoDB on Your Computer | 46 |
| Deploying as an Apache Maven Repository | 48 |
| Install the Official Docker Image | 48 |
| Usage Notes | 49 |
| Setting Up DynamoDB (Web Service) | 51 |
| Signing Up for AWS | 51 |
| Getting an AWS Access Key | 52 |
| Configuring Your Credentials | 53 |
| Accessing DynamoDB | 54 |
| Using the Console | 54 |
| Working with User Preferences | 55 |
| Using the AWS CLI | 56 |
| Downloading and Configuring the AWS CLI | 56 |
| Using the AWS CLI with DynamoDB | 56 |
| Using the AWS CLI with Downloadable DynamoDB | 57 |
| Using the API | 58 |
| Using the NoSQL Workbench | 58 |
| IP Address Ranges | 58 |
| Getting Started with DynamoDB | 59 |
| Basic Concepts | 59 |
| Prerequisites | 59 |
| Step 1: Create a Table | 59 |
| Step 2: Write Data | 62 |
| Step 3: Read Data | 64 |
| Step 4: Update Data | 65 |

| | |
|--|-----|
| Step 5: Query Data | 67 |
| Step 6: Create a Global Secondary Index | 69 |
| Step 7: Query the Global Secondary Index | 72 |
| Step 8: (Optional) Clean Up | 74 |
| Next Steps | 74 |
| Getting Started with the AWS SDKs | 76 |
| Java and DynamoDB | 76 |
| Tutorial Prerequisites | 76 |
| Step 1: Create a Table | 77 |
| Step 2: Load Sample Data | 78 |
| Step 3: Create, Read, Update, and Delete an Item | 81 |
| Step 4: Query and Scan the Data | 89 |
| Step 5: (Optional) Delete the Table | 94 |
| Summary | 95 |
| JavaScript and DynamoDB | 95 |
| Tutorial Prerequisites | 96 |
| Step 1: Create a Table | 96 |
| Step 2: Load Sample Data | 98 |
| Step 3: Create, Read, Update, and Delete an Item | 101 |
| Step 4: Query and Scan the Data | 111 |
| Step 5: (Optional): Delete the Table | 116 |
| Summary | 117 |
| Node.js and DynamoDB | 118 |
| Tutorial Prerequisites | 118 |
| Step 1: Create a Table | 118 |
| Step 2: Load Sample Data | 119 |
| Step 3: Create, Read, Update, and Delete an Item | 122 |
| Step 4: Query and Scan Data | 130 |
| Step 5: (Optional): Delete the Table | 134 |
| Summary | 135 |
| .NET and DynamoDB | 135 |
| Tutorial Prerequisites | 136 |
| Step 1: Create a Client | 137 |
| Step 2: Create a Table | 139 |
| Step 3: Load Data | 142 |
| Step 4: Add a Movie | 145 |
| Step 5: Read a Movie | 146 |
| Step 6: Update the Movie | 147 |
| Step 7: Delete an Item with Conditions | 150 |
| Step 8: Query a Table | 152 |
| Step 9: Scan a Table | 155 |
| Step 10: Delete the Table | 156 |
| PHP and DynamoDB | 157 |
| Tutorial Prerequisites | 157 |
| Step 1: Create a Table | 158 |
| Step 2: Load Sample Data | 159 |
| Step 3: Create, Read, Update, and Delete an Item | 162 |
| Step 4: Query and Scan the Data | 171 |
| Step 5: (Optional) Delete the Table | 176 |
| Summary | 177 |
| Python and DynamoDB | 178 |
| Tutorial Prerequisites | 178 |
| Step 1: Create a Table | 178 |
| Step 2: Load Sample Data | 179 |
| Step 3: Crud Operations | 181 |
| Step 4: Query and Scan the Data | 187 |
| Step 5: (Optional) Delete the Table | 190 |

| | |
|---|-----|
| Summary | 191 |
| Ruby and DynamoDB | 191 |
| Tutorial Prerequisites | 192 |
| Step 1: Create a Table | 192 |
| Step 2: Load Sample Data | 193 |
| Step 3: Create, Read, Update, and Delete an Item | 195 |
| Step 4: Query and Scan the Data | 203 |
| Step 5: (Optional) Delete the Table | 208 |
| Summary | 208 |
| Programming with DynamoDB | 210 |
| Overview of AWS SDK Support for DynamoDB | 210 |
| Programmatic Interfaces | 212 |
| Low-Level Interfaces | 213 |
| Document Interfaces | 214 |
| Object Persistence Interface | 214 |
| Low-Level API | 216 |
| Request Format | 217 |
| Response Format | 218 |
| Data Type Descriptors | 218 |
| Numeric Data | 219 |
| Binary Data | 219 |
| Error Handling | 219 |
| Error Components | 220 |
| Error Messages and Codes | 220 |
| Error Handling in Your Application | 223 |
| Error Retries and Exponential Backoff | 224 |
| Batch Operations and Error Handling | 224 |
| Higher-Level Programming Interfaces for DynamoDB | 225 |
| Java: DynamoDBMapper | 225 |
| .NET: Document Model | 273 |
| .NET: Object Persistence Model | 295 |
| Running the Code Examples | 324 |
| Load Sample Data | 325 |
| Java Code Examples | 330 |
| .NET Code Examples | 332 |
| Working with DynamoDB | 335 |
| Working with Tables | 335 |
| Basic Operations on Tables | 335 |
| Considerations When Changing Read/Write Capacity Mode | 340 |
| Provisioned Capacity Tables | 341 |
| Item Sizes and Formats | 345 |
| Managing Throughput Capacity with Auto Scaling | 345 |
| Tagging Resources | 359 |
| Working with Tables: Java | 362 |
| Working with Tables: .NET | 367 |
| Working with Items | 374 |
| Reading an Item | 375 |
| Writing an Item | 375 |
| Return Values | 377 |
| Batch Operations | 378 |
| Atomic Counters | 380 |
| Conditional Writes | 380 |
| Using Expressions | 385 |
| Expiring Items with Time to Live | 408 |
| Working with Items: Java | 415 |
| Working with Items: .NET | 435 |
| Working with Queries | 458 |

| | |
|--|-----|
| Key Condition Expression | 458 |
| Filter Expressions for Query | 460 |
| Limiting the Number of Items in the Result Set | 461 |
| Paginating Query Results | 461 |
| Counting the Items in the Results | 462 |
| Capacity Units Consumed by Query | 463 |
| Read Consistency for Query | 463 |
| Querying: Java | 463 |
| Querying: .NET | 469 |
| Working with Scans | 475 |
| Filter Expressions for Scan | 476 |
| Limiting the Number of Items in the Result Set | 476 |
| Paginating the Results | 477 |
| Counting the Items in the Results | 478 |
| Capacity Units Consumed by Scan | 478 |
| Read Consistency for Scan | 479 |
| Parallel Scan | 479 |
| Scanning: Java | 481 |
| Scanning: .NET | 488 |
| Working with Indexes | 496 |
| Global Secondary Indexes | 499 |
| Local Secondary Indexes | 536 |
| Working with Streams | 573 |
| Endpoints for DynamoDB Streams | 574 |
| Enabling a Stream | 575 |
| Reading and Processing a Stream | 576 |
| DynamoDB Streams and TTL | 578 |
| Using the DynamoDB Streams Kinesis Adapter to Process Stream Records | 578 |
| DynamoDB Streams Low-Level API: Java Example | 590 |
| Cross-Region Replication | 594 |
| DynamoDB Streams and AWS Lambda Triggers | 594 |
| On-Demand Backup and Restore | 603 |
| How It Works | 603 |
| Backups | 603 |
| Restores | 604 |
| Backing Up a Table | 605 |
| Creating a Table Backup (Console) | 606 |
| Creating a Table Backup (AWS CLI) | 606 |
| Restoring a Table from a Backup | 607 |
| Restoring a Table from a Backup (Console) | 607 |
| Restoring a Table from a Backup (AWS CLI) | 610 |
| Deleting a Table Backup | 612 |
| Deleting a Table Backup (Console) | 612 |
| Deleting a Table Backup (AWS CLI) | 613 |
| Using IAM | 614 |
| Example 1: Allow the CreateBackup and RestoreTableFromBackup Actions | 614 |
| Example 2: Allow CreateBackup and Deny RestoreTableFromBackup | 614 |
| Example 3: Allow ListBackups and Deny CreateBackup and RestoreTableFromBackup | 615 |
| Example 4: Allow ListBackups and Deny DeleteBackup | 615 |
| Example 5: Allow RestoreTableFromBackup and DescribeBackup for All Resources and Deny DeleteBackup for a Specific Backup | 616 |
| Example 6: Allow CreateBackup for a Specific Table | 616 |
| Example 7: Allow ListBackups | 617 |
| Point-in-Time Recovery | 618 |
| How It Works | 618 |
| Before You Begin | 619 |
| Restoring a Table To a Point in Time | 620 |

| | |
|--|-----|
| Restoring a Table to a Point in Time (Console) | 620 |
| Restoring a Table to a Point in Time (AWS CLI) | 621 |
| Global Tables | 624 |
| Determine Version | 625 |
| Version 2019.11.21 (Current) | 625 |
| How It Works | 626 |
| Best Practices and Requirements | 627 |
| Tutorial: Creating a Global Table | 627 |
| Monitoring Global Tables | 632 |
| Using IAM with Global Tables | 633 |
| Updating | 635 |
| Version 2017.11.29 | 638 |
| How It Works | 638 |
| Best Practices and Requirements | 640 |
| Creating a Global Table | 642 |
| Monitoring Global Tables | 644 |
| Using IAM with Global Tables | 645 |
| DynamoDB Transactions | 648 |
| How It Works | 648 |
| TransactWriteItems | 649 |
| TransactGetItems | 650 |
| Isolation Levels | 650 |
| Transaction Conflict Handling | 652 |
| Transactions in DAX | 652 |
| Capacity Management | 652 |
| Best Practices | 653 |
| Integration with Other Features | 653 |
| Transactions vs. the Client Library | 653 |
| Using IAM with Transactions | 654 |
| Example 1: Allow Transactional Operations | 654 |
| Example 2: Allow Only Transactional Operations | 654 |
| Example 3: Allow Nontransactional Reads and Writes, and Block Transactional Reads and Writes | 655 |
| Example 4: Prevent Information from Being Returned on a ConditionCheck Failure | 655 |
| Example Code | 656 |
| Making an Order | 656 |
| Reading the Order Details | 658 |
| Additional Examples | 658 |
| In-Memory Acceleration with DAX | 659 |
| Use Cases for DAX | 659 |
| DAX Usage Notes | 660 |
| How It Works | 661 |
| How DAX Processes Requests | 662 |
| Item Cache | 663 |
| Query Cache | 663 |
| DAX Cluster Components | 664 |
| Nodes | 664 |
| Clusters | 664 |
| Regions and Availability Zones | 665 |
| Parameter Groups | 666 |
| Security Groups | 666 |
| Cluster ARN | 666 |
| Cluster Endpoint | 666 |
| Node Endpoints | 666 |
| Subnet Groups | 667 |
| Events | 667 |
| Maintenance Window | 667 |

| | |
|--|-----|
| Creating a DAX Cluster | 668 |
| Creating an IAM Service Role for DAX to Access DynamoDB | 668 |
| Using the AWS CLI | 669 |
| Using the Console | 673 |
| Consistency Models | 676 |
| Consistency Among DAX Cluster Nodes | 676 |
| DAX Item Cache Behavior | 676 |
| DAX Query Cache Behavior | 678 |
| Strongly Consistent and Transactional Reads | 679 |
| Negative Caching | 679 |
| Strategies for Writes | 680 |
| Developing with the DAX Client | 682 |
| Tutorial: Running a Sample Application | 682 |
| Modifying an Existing Application to Use DAX | 720 |
| Querying Global Secondary Indexes | 723 |
| Managing DAX Clusters | 726 |
| IAM Permissions for Managing a DAX Cluster | 726 |
| Scaling a DAX Cluster | 728 |
| Customizing DAX Cluster Settings | 729 |
| Configuring TTL Settings | 729 |
| Tagging Support for DAX | 730 |
| AWS CloudTrail Integration | 731 |
| Deleting a DAX Cluster | 731 |
| Monitoring DAX | 732 |
| Monitoring Tools | 732 |
| Monitoring with CloudWatch | 733 |
| Logging DAX Operations Using AWS CloudTrail | 745 |
| DAX Access Control | 745 |
| IAM Service Role for DAX | 745 |
| IAM Policy to Allow DAX Cluster Access | 746 |
| Case Study: Accessing DynamoDB and DAX | 747 |
| Access to DynamoDB, But No Access with DAX | 748 |
| Access to DynamoDB and to DAX | 749 |
| Access to DynamoDB Via DAX, But No Direct Access to DynamoDB | 753 |
| DAX Encryption at Rest | 755 |
| Enabling Encryption at Rest Using the AWS Management Console | 756 |
| Using Service-Linked Roles for DAX | 756 |
| Service-Linked Role Permissions for DAX | 757 |
| Creating a Service-Linked Role for DAX | 758 |
| Editing a Service-Linked Role for DAX | 758 |
| Deleting a Service-Linked Role for DAX | 758 |
| Accessing DAX Across AWS Accounts | 759 |
| Set Up IAM | 760 |
| Set Up a VPC | 761 |
| Modify the DAX Client to Allow Cross-account Access | 763 |
| DAX Cluster Sizing Guide | 766 |
| Overview | 766 |
| Estimating Traffic | 767 |
| Load Testing | 767 |
| API Reference | 768 |
| NoSQL Workbench | 769 |
| Download | 769 |
| Data Modeler | 770 |
| Creating a New Model | 770 |
| Importing an Existing Model | 774 |
| Exporting a Model | 775 |
| Editing an Existing Model | 776 |

| | |
|---|-----|
| Data Visualizer | 779 |
| Adding Sample Data | 779 |
| Facets | 780 |
| Aggregate View | 781 |
| Committing a Data Model | 782 |
| Operation Builder | 785 |
| Exploring Datasets | 785 |
| Building Operations | 789 |
| Sample Data Models | 798 |
| Employee Data Model | 798 |
| Discussion Forum Data Model | 799 |
| Music Library Data Model | 799 |
| Ski Resort Data Model | 799 |
| Credit Card Offers Data Model | 799 |
| Bookmarks Data Model | 800 |
| Release History | 800 |
| Contributor Insights | 802 |
| How It Works | 802 |
| CloudWatch Contributor Insights for DynamoDB Rules | 802 |
| Understanding CloudWatch Contributor Insights for DynamoDB Graphs | 803 |
| Interactions with Other DynamoDB Features | 805 |
| CloudWatch Contributor Insights for DynamoDB Billing | 806 |
| Getting Started | 806 |
| Using Contributor Insights (Console) | 807 |
| Using Contributor Insights (AWS CLI) | 810 |
| Using IAM | 810 |
| | 810 |
| Example: Enable or Disable CloudWatch Contributor Insights for DynamoDB | 811 |
| Example: Retrieve CloudWatch Contributor Insights Rule Report | 811 |
| Example: Selectively Apply CloudWatch Contributor Insights for DynamoDB Permissions Based on Resource | 811 |
| Using Service-Linked Roles | 812 |
| Security | 814 |
| Data Protection | 814 |
| Encryption at Rest | 815 |
| Data Protection in DAX | 823 |
| Internetwork Traffic Privacy | 823 |
| Identity and Access Management | 823 |
| Identity and Access Management | 824 |
| Identity and Access Management in DAX | 855 |
| Logging and Monitoring | 855 |
| Logging and Monitoring in DynamoDB | 855 |
| Logging and Monitoring in DAX | 885 |
| Compliance Validation | 885 |
| Resilience | 886 |
| Infrastructure Security | 886 |
| Using VPC Endpoints | 887 |
| Configuration and Vulnerability Analysis | 893 |
| Security Best Practices | 893 |
| Preventative Security Best Practices | 893 |
| Detective Security Best Practices | 895 |
| Best Practices | 897 |
| NoSQL Design | 898 |
| NoSQL vs. RDBMS | 898 |
| Two Key Concepts | 899 |
| General Approach | 899 |
| Partition Key Design | 900 |

| | |
|--|-----|
| Burst Capacity | 900 |
| Adaptive Capacity | 900 |
| Distributing Workloads | 902 |
| Write Sharding | 903 |
| Uploading Data Efficiently | 904 |
| Sort Key Design | 905 |
| Version Control | 905 |
| Secondary Indexes | 906 |
| General Guidelines | 907 |
| Sparse Indexes | 908 |
| Aggregation | 910 |
| GSI Overloading | 910 |
| GSI Sharding | 911 |
| Creating a Replica | 912 |
| Large Items | 913 |
| Compression | 913 |
| Using Amazon S3 | 913 |
| Time Series Data | 914 |
| Design Pattern for Time Series Data | 914 |
| Time Series Table Examples | 914 |
| Many-to-Many Relationships | 915 |
| Adjacency Lists | 915 |
| Materialized Graphs | 916 |
| Hybrid DynamoDB-RDBMS | 919 |
| Not Migrating | 919 |
| Hybrid System Implementation | 920 |
| Relational Modeling | 920 |
| First Steps | 922 |
| Example | 923 |
| Querying and Scanning | 925 |
| Scan Performance | 925 |
| Avoid Spikes | 926 |
| Parallel Scans | 928 |
| Global Tables | 929 |
| Integrating with Other AWS Services | 930 |
| Integrating with Amazon Cognito | 930 |
| Integrating with Amazon Redshift | 932 |
| Integrating with Amazon EMR | 933 |
| Overview | 933 |
| Tutorial: Working with Amazon DynamoDB and Apache Hive | 934 |
| Creating an External Table in Hive | 940 |
| Processing HiveQL Statements | 942 |
| Querying Data in DynamoDB | 943 |
| Copying Data to and from Amazon DynamoDB | 945 |
| Performance Tuning | 956 |
| Limits | 960 |
| Read/Write Capacity Mode and Throughput | 960 |
| Capacity Unit Sizes (for Provisioned Tables) | 960 |
| Request Unit Sizes (for On-Demand Tables) | 960 |
| Throughput Default Limits | 961 |
| Increasing or Decreasing Throughput (for Provisioned Tables) | 961 |
| Tables | 962 |
| Table Size | 962 |
| Tables Per Account | 962 |
| Global Tables | 962 |
| Secondary Indexes | 963 |
| Secondary Indexes Per Table | 963 |

| | |
|---|------|
| Projected Secondary Index Attributes Per Table | 963 |
| Partition Keys and Sort Keys | 963 |
| Partition Key Length | 963 |
| Partition Key Values | 963 |
| Sort Key Length | 963 |
| Sort Key Values | 963 |
| Naming Rules | 964 |
| Table Names and Secondary Index Names | 964 |
| Attribute Names | 964 |
| Data Types | 964 |
| String | 964 |
| Number | 964 |
| Binary | 965 |
| Items | 965 |
| Item Size | 965 |
| Item Size for Tables with Local Secondary Indexes | 965 |
| Attributes | 965 |
| Attribute Name-Value Pairs Per Item | 965 |
| Number of Values in List, Map, or Set | 965 |
| Attribute Values | 965 |
| Nested Attribute Depth | 966 |
| Expression Parameters | 966 |
| Lengths | 966 |
| Operators and Operands | 966 |
| Reserved Words | 966 |
| DynamoDB Transactions | 966 |
| DynamoDB Streams | 967 |
| Simultaneous Readers of a Shard in DynamoDB Streams | 967 |
| Maximum Write Capacity for a Table with a Stream Enabled | 967 |
| DynamoDB Accelerator (DAX) | 967 |
| AWS Region Availability | 967 |
| Nodes | 967 |
| Parameter Groups | 967 |
| Subnet Groups | 968 |
| API-Specific Limits | 968 |
| DynamoDB Encryption at Rest | 969 |
| API Reference | 970 |
| Appendix | 971 |
| Troubleshooting SSL/TLS connection establishment issues | 971 |
| Testing your application or service | 971 |
| Testing your client browser | 972 |
| Updating your software application client | 972 |
| Updating your client browser | 972 |
| Manually updating your certificate bundle | 972 |
| Example Tables and Data | 973 |
| Sample Data Files | 973 |
| Creating Example Tables and Uploading Data | 982 |
| Creating Example Tables and Uploading Data - Java | 982 |
| Creating Example Tables and Uploading Data - .NET | 989 |
| Example Application Using AWS SDK for Python (Boto3) | 998 |
| Step 1: Deploy and Test Locally | 998 |
| Step 2: Examine the Data Model and Implementation Details | 1002 |
| Step 3: Deploy in Production | 1010 |
| Step 4: Clean Up Resources | 1016 |
| Integrating with Amazon Data Pipeline | 1017 |
| Prerequisites to Export and Import Data | 1019 |
| Exporting Data From DynamoDB to Amazon S3 | 1022 |

| | |
|---|------|
| Importing Data From Amazon S3 to DynamoDB | 1023 |
| Troubleshooting | 1024 |
| Predefined Templates for AWS Data Pipeline and DynamoDB | 1025 |
| Amazon DynamoDB Storage Backend for Titan | 1025 |
| Reserved Words in DynamoDB | 1026 |
| Legacy Conditional Parameters | 1035 |
| AttributesToGet | 1036 |
| AttributeUpdates | 1036 |
| ConditionalOperator | 1038 |
| Expected | 1038 |
| KeyConditions | 1041 |
| QueryFilter | 1043 |
| ScanFilter | 1045 |
| Writing Conditions With Legacy Parameters | 1046 |
| Previous Low-Level API Version (2011-12-05) | 1051 |
| Batch.GetItem | 1052 |
| BatchWriteItem | 1057 |
| CreateTable | 1061 |
| DeleteItem | 1066 |
| DeleteTable | 1070 |
| DescribeTables | 1073 |
| GetItem | 1076 |
| ListTables | 1078 |
| PutItem | 1080 |
| Query | 1085 |
| Scan | 1094 |
| UpdateItem | 1104 |
| UpdateTable | 1110 |
| Document History | 1114 |
| Earlier Updates | 1116 |

What Is Amazon DynamoDB?

Welcome to the Amazon DynamoDB Developer Guide.

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data. For more information, see [DynamoDB Encryption at Rest \(p. 815\)](#).

With DynamoDB, you can create database tables that can store and retrieve any amount of data and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation. You can use the AWS Management Console to monitor resource utilization and performance metrics.

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival for regulatory compliance needs. For more information, see [On-Demand Backup and Restore for DynamoDB \(p. 603\)](#).

You can create on-demand backups and enable point-in-time recovery for your Amazon DynamoDB tables. Point-in-time recovery helps protect your tables from accidental write or delete operations. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. For more information, see [Point-in-Time Recovery: How It Works \(p. 618\)](#).

DynamoDB allows you to delete expired items from tables automatically to help you reduce storage usage and the cost of storing data that is no longer relevant. For more information, see [Expiring Items By Using DynamoDB Time to Live \(TTL\) \(p. 408\)](#).

High Availability and Durability

DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state disks (SSDs) and is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions. For more information, see [Global Tables: Multi-Region Replication with DynamoDB \(p. 624\)](#).

Getting Started with DynamoDB

We recommend that you begin by reading the following sections:

- [Amazon DynamoDB: How It Works \(p. 2\)](#)—To learn essential DynamoDB concepts.
- [Setting Up DynamoDB \(p. 46\)](#)—To learn how to set up DynamoDB (the downloadable version or the web service).
- [Accessing DynamoDB \(p. 54\)](#)—To learn how to access DynamoDB using the console, AWS CLI, or API.

To get started quickly with DynamoDB, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

To learn more about application development, see the following:

- [Programming with DynamoDB and the AWS SDKs \(p. 210\)](#)
- [Working with Tables, Items, Queries, Scans, and Indexes \(p. 335\)](#)

To quickly find recommendations for maximizing performance and minimizing throughput costs, see [Best Practices for Designing and Architecting with DynamoDB \(p. 897\)](#). To learn how to tag DynamoDB resources, see [Adding Tags and Labels to Resources \(p. 359\)](#).

For best practices, how-to guides, and tools, see [Amazon DynamoDB resources](#).

You can use AWS Database Migration Service (AWS DMS) to migrate data from a relational database or MongoDB to a DynamoDB table. For more information, see the [AWS Database Migration Service User Guide](#).

To learn how to use MongoDB as a migration source, see [Using MongoDB as a Source for AWS Database Migration Service](#). To learn how to use DynamoDB as a migration target, see [Using an Amazon DynamoDB Database as a Target for AWS Database Migration Service](#).

Amazon DynamoDB: How It Works

The following sections provide an overview of Amazon DynamoDB service components and how they interact.

After you read this introduction, try working through the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section, which walks you through the process of creating sample tables, uploading data, and performing some basic database operations.

For language-specific tutorials with sample code, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Topics

- [Core Components of Amazon DynamoDB \(p. 2\)](#)
- [DynamoDB API \(p. 10\)](#)
- [Naming Rules and Data Types \(p. 12\)](#)
- [Read Consistency \(p. 16\)](#)
- [Read/Write Capacity Mode \(p. 17\)](#)
- [Partitions and Data Distribution \(p. 21\)](#)

Core Components of Amazon DynamoDB

In DynamoDB, tables, items, and attributes are the core components that you work with. A *table* is a collection of *items*, and each item is a collection of *attributes*. DynamoDB uses primary keys to uniquely identify each item in a table and secondary indexes to provide more querying flexibility. You can use DynamoDB Streams to capture data modification events in DynamoDB tables.

There are limits in DynamoDB. For more information, see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

Topics

- [Tables, Items, and Attributes \(p. 3\)](#)
- [Primary Key \(p. 6\)](#)
- [Secondary Indexes \(p. 7\)](#)
- [DynamoDB Streams \(p. 9\)](#)

Tables, Items, and Attributes

The following are the basic DynamoDB components:

- **Tables** – Similar to other database systems, DynamoDB stores data in tables. A *table* is a collection of data. For example, see the example table called *People* that you could use to store personal contact information about friends, family, or anyone else of interest. You could also have a *Cars* table to store information about vehicles that people drive.
- **Items** – Each table contains zero or more items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar in many ways to rows, records, or tuples in other database systems. In DynamoDB, there is no limit to the number of items you can store in a table.
- **Attributes** – Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

The following diagram shows a table named *People* with some example items and attributes.

People

```
{  
    "PersonID": 101,  
    "LastName": "Smith",  
    "FirstName": "Fred",  
    "Phone": "555-4321"  
}
```

```
{  
    "PersonID": 102,  
    "LastName": "Jones",  
    "FirstName": "Mary",  
    "Address": {  
        "Street": "123 Main",  
        "City": "Anytown",  
        "State": "OH",  
        "ZIPCode": 12345  
    }  
}
```

```
{  
    "PersonID": 103,  
    "LastName": "Stephens",  
    "FirstName": "Howard",  
    "Address": {  
        "Street": "123 Main",  
        "City": "London",  
        "PostalCode": "ER3 5K8"  
    },  
    "FavoriteColor": "Blue"  
}
```

Note the following about the *People* table:

- Each item in the table has a unique identifier, or primary key, that distinguishes the item from all of the others in the table. In the *People* table, the primary key consists of one attribute (*PersonID*).
- Other than the primary key, the *People* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.
- Most of the attributes are *scalar*, which means that they can have only one value. Strings and numbers are common examples of scalars.

- Some of the items have a nested attribute (*Address*). DynamoDB supports nested attributes up to 32 levels deep.

The following is another example table named *Music* that you could use to keep track of your music collection.

Music

| |
|---|
| <pre>{ "Artist": "No One You Know", "SongTitle": "My Dog Spot", "AlbumTitle": "Hey Now", "Price": 1.98, "Genre": "Country", "CriticRating": 8.4 }</pre> |
| <pre>{ "Artist": "No One You Know", "SongTitle": "Somewhere Down The Road", "AlbumTitle": "Somewhat Famous", "Genre": "Country", "CriticRating": 8.4, "Year": 1984 }</pre> |
| <pre>{ "Artist": "The Acme Band", "SongTitle": "Still in Love", "AlbumTitle": "The Buck Starts Here", "Price": 2.47, "Genre": "Rock", "PromotionInfo": { "RadioStationsPlaying": ["KHCN", "KQBX", "WTNR", "WJZH"], "TourDates": { "Seattle": "20150625", "Cleveland": "20150630" }, "Rotation": "Heavy" } }</pre> |
| <pre>{ "Artist": "The Acme Band", "SongTitle": "Look Out, World", "AlbumTitle": "The Buck Starts Here", "Price": 0.99, "Genre": "Rock" }</pre> |

Note the following about the *Music* table:

- The primary key for *Music* consists of two attributes (*Artist* and *SongTitle*). Each item in the table must have these two attributes. The combination of *Artist* and *SongTitle* distinguishes each item in the table from all of the others.

- Other than the primary key, the *Music* table is schemaless, which means that neither the attributes nor their data types need to be defined beforehand. Each item can have its own distinct attributes.
- One of the items has a nested attribute (*PromotionInfo*), which contains other nested attributes. DynamoDB supports nested attributes up to 32 levels deep.

For more information, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. The primary key uniquely identifies each item in the table, so that no two items can have the same key.

DynamoDB supports two different kinds of primary keys:

- **Partition key** – A simple primary key, composed of one attribute known as the *partition key*.

DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.

In a table that has only a partition key, no two items can have the same partition key value.

The *People* table described in [Tables, Items, and Attributes \(p. 3\)](#) is an example of a table with a simple primary key (*PersonID*). You can access any item in the *People* table directly by providing the *PersonId* value for that item.

- **Partition key and sort key** – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.

DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored. All items with the same partition key value are stored together, in sorted order by sort key value.

In a table that has a partition key and a sort key, it's possible for two items to have the same partition key value. However, those two items must have different sort key values.

The *Music* table described in [Tables, Items, and Attributes \(p. 3\)](#) is an example of a table with a composite primary key (*Artist* and *SongTitle*). You can access any item in the *Music* table directly, if you provide the *Artist* and *SongTitle* values for that item.

A composite primary key gives you additional flexibility when querying data. For example, if you provide only the value for *Artist*, DynamoDB retrieves all of the songs by that artist. To retrieve only a subset of songs by a particular artist, you can provide a value for *Artist* along with a range of values for *SongTitle*.

Note

The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions, based on their partition key values.

The sort key of an item is also known as its *range attribute*. The term *range attribute* derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

Each primary key attribute must be a scalar (meaning that it can hold only a single value). The only data types allowed for primary key attributes are string, number, or binary. There are no such restrictions for other, non-key attributes.

Secondary Indexes

You can create one or more secondary indexes on a table. A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB doesn't require that you use indexes, but they give your applications more flexibility when querying your data. After you create a secondary index on a table, you can read data from the index in much the same way as you do from the table.

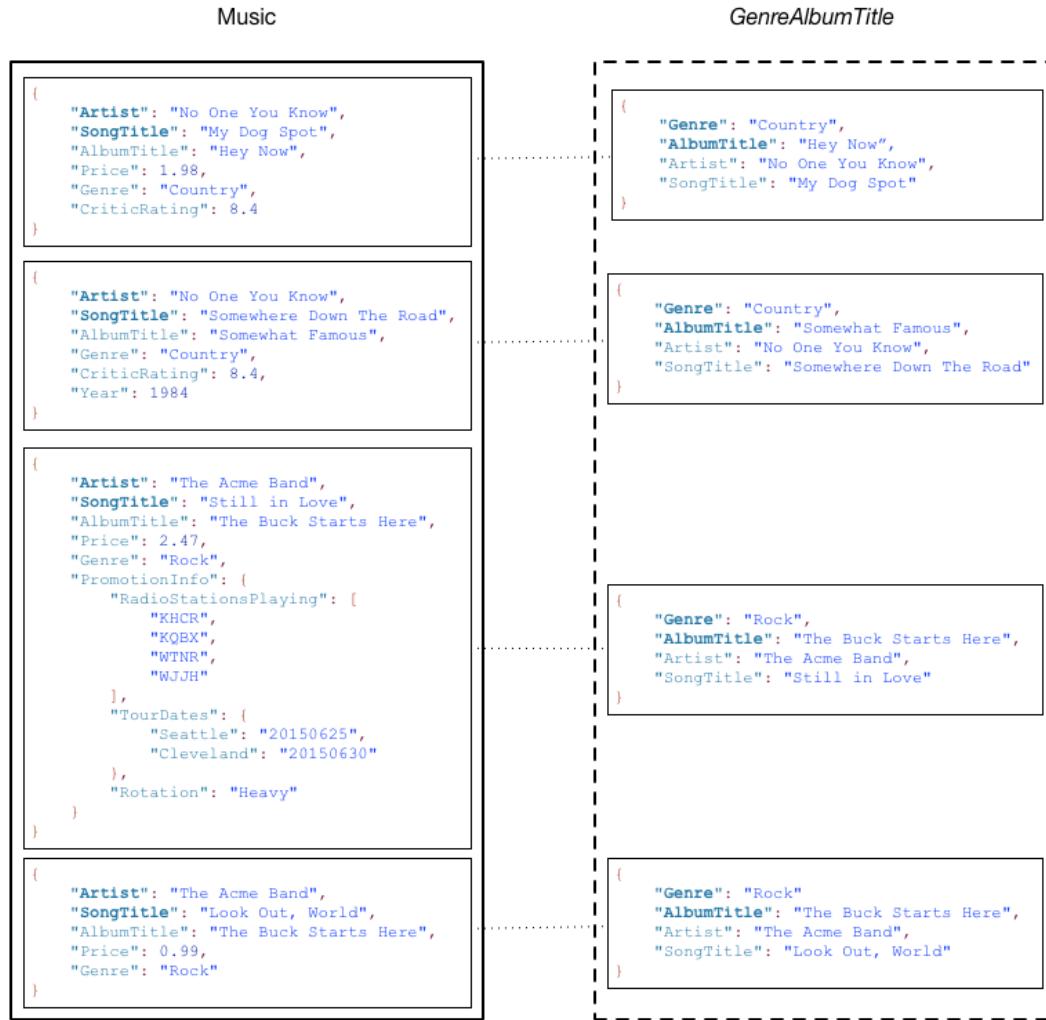
DynamoDB supports two kinds of indexes:

- Global secondary index – An index with a partition key and sort key that can be different from those on the table.
- Local secondary index – An index that has the same partition key as the table, but a different sort key.

Each table in DynamoDB has a limit of 20 global secondary indexes (default limit) and 5 local secondary indexes per table.

In the example *Music* table shown previously, you can query data items by *Artist* (partition key) or by *Artist* and *SongTitle* (partition key and sort key). What if you also wanted to query the data by *Genre* and *AlbumTitle*? To do this, you could create an index on *Genre* and *AlbumTitle*, and then query the index in much the same way as you'd query the *Music* table.

The following diagram shows the example *Music* table, with a new index called *GenreAlbumTitle*. In the index, *Genre* is the partition key and *AlbumTitle* is the sort key.



Note the following about the `GenreAlbumTitle` index:

- Every index belongs to a table, which is called the *base table* for the index. In the preceding example, `Music` is the base table for the `GenreAlbumTitle` index.
- DynamoDB maintains indexes automatically. When you add, update, or delete an item in the base table, DynamoDB adds, updates, or deletes the corresponding item in any indexes that belong to that table.
- When you create an index, you specify which attributes will be copied, or *projected*, from the base table to the index. At a minimum, DynamoDB projects the key attributes from the base table into the index. This is the case with `GenreAlbumTitle`, where only the key attributes from the `Music` table are projected into the index.

You can query the *GenreAlbumTitle* index to find all albums of a particular genre (for example, all *Rock* albums). You can also query the index to find all albums within a particular genre that have certain album titles (for example, all *Country* albums with titles that start with the letter H).

For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

DynamoDB Streams

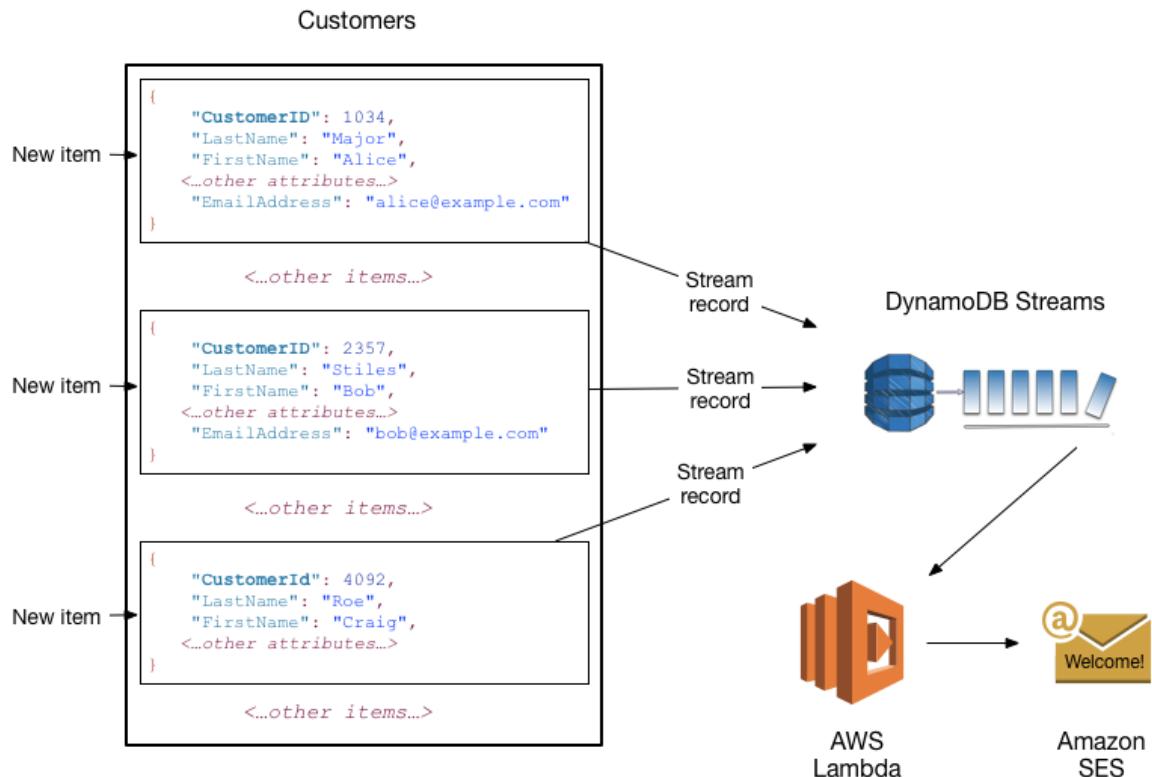
DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near-real time, and in the order that the events occurred.

Each event is represented by a *stream record*. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

- A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
- An item is updated: The stream captures the "before" and "after" image of any attributes that were modified in the item.
- An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create a *trigger*—code that executes automatically whenever an event of interest appears in a stream. For example, consider a *Customers* table that contains customer information for a company. Suppose that you want to send a "welcome" email to each new customer. You could enable a stream on that table, and then associate the stream with a Lambda function. The Lambda function would execute whenever a new stream record appears, but only process new items added to the *Customers* table. For any item that has an `EmailAddress` attribute, the Lambda function would invoke Amazon Simple Email Service (Amazon SES) to send an email to that address.



Note

In this example, the last customer, Craig Roe, will not receive an email because he doesn't have an `EmailAddress`.

In addition to triggers, DynamoDB Streams enables powerful solutions such as data replication within and across AWS Regions, materialized views of data in DynamoDB tables, data analysis using Kinesis materialized views, and much more.

For more information, see [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#).

DynamoDB API

To work with Amazon DynamoDB, your application must use a few simple API operations. The following is a summary of these operations, organized by category.

Topics

- [Control Plane \(p. 10\)](#)
- [Data Plane \(p. 11\)](#)
- [DynamoDB Streams \(p. 12\)](#)
- [Transactions \(p. 12\)](#)

Control Plane

Control plane operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables.

- `CreateTable` – Creates a new table. Optionally, you can create one or more secondary indexes, and enable DynamoDB Streams for the table.

- `DescribeTable` – Returns information about a table, such as its primary key schema, throughput settings, and index information.
- `ListTables` – Returns the names of all of your tables in a list.
- `UpdateTable` – Modifies the settings of a table or its indexes, creates or removes new indexes on a table, or modifies DynamoDB Streams settings for a table.
- `DeleteTable` – Removes a table and all of its dependent objects from DynamoDB.

Data Plane

Data plane operations let you perform create, read, update, and delete (also called *CRUD*) actions on data in a table. Some of the data plane operations also let you read data from a secondary index.

Creating Data

- `PutItem` – Writes a single item to a table. You must specify the primary key attributes, but you don't have to specify other attributes.
- `BatchWriteItem` – Writes up to 25 items to a table. This is more efficient than calling `PutItem` multiple times because your application only needs a single network round trip to write the items. You can also use `BatchWriteItem` for deleting multiple items from one or more tables.

Reading Data

- `GetItem` – Retrieves a single item from a table. You must specify the primary key for the item that you want. You can retrieve the entire item, or just a subset of its attributes.
- `BatchGetItem` – Retrieves up to 100 items from one or more tables. This is more efficient than calling `GetItem` multiple times because your application only needs a single network round trip to read the items.
- `Query` – Retrieves all items that have a specific partition key. You must specify the partition key value. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply a condition to the sort key values so that you only retrieve a subset of the data that has the same partition key. You can use this operation on a table, provided that the table has both a partition key and a sort key. You can also use this operation on an index, provided that the index has both a partition key and a sort key.
- `Scan` – Retrieves all items in the specified table or index. You can retrieve entire items, or just a subset of their attributes. Optionally, you can apply a filtering condition to return only the values that you are interested in and discard the rest.

Updating Data

- `UpdateItem` – Modifies one or more attributes in an item. You must specify the primary key for the item that you want to modify. You can add new attributes and modify or remove existing attributes. You can also perform conditional updates, so that the update is only successful when a user-defined condition is met. Optionally, you can implement an atomic counter, which increments or decrements a numeric attribute without interfering with other write requests.

Deleting Data

- `DeleteItem` – Deletes a single item from a table. You must specify the primary key for the item that you want to delete.
- `BatchWriteItem` – Deletes up to 25 items from one or more tables. This is more efficient than calling `DeleteItem` multiple times because your application only needs a single network round trip to delete the items. You can also use `BatchWriteItem` for adding multiple items to one or more tables.

DynamoDB Streams

DynamoDB Streams operations let you enable or disable a stream on a table, and allow access to the data modification records contained in a stream.

- `ListStreams` – Returns a list of all your streams, or just the stream for a specific table.
- `DescribeStream` – Returns information about a stream, such as its Amazon Resource Name (ARN) and where your application can begin reading the first few stream records.
- `GetShardIterator` – Returns a *shard iterator*, which is a data structure that your application uses to retrieve the records from the stream.
- `GetRecords` – Retrieves one or more stream records, using a given shard iterator.

Transactions

Transactions provide atomicity, consistency, isolation, and durability (ACID) enabling you to maintain data correctness in your applications more easily.

- `TransactWriteItems` – A batch operation that allows `Put`, `Update`, and `Delete` operations to multiple items both within and across tables with a guaranteed all-or-nothing result.
- `TransactGetItems` – A batch operation that allows `Get` operations to retrieve multiple items from one or more tables.

Naming Rules and Data Types

This section describes the Amazon DynamoDB naming rules and the various data types that DynamoDB supports. There are limits that apply to data types. For more information, see [Data Types \(p. 964\)](#).

Topics

- [Naming Rules \(p. 12\)](#)
- [Data Types \(p. 13\)](#)

Naming Rules

Tables, attributes, and other objects in DynamoDB must have names. Names should be meaningful and concise—for example, names such as *Products*, *Books*, and *Authors* are self-explanatory.

The following are the naming rules for DynamoDB:

- All names must be encoded using UTF-8, and are case-sensitive.
- Table names and index names must be between 3 and 255 characters long, and can contain only the following characters:
 - a–z
 - A–Z
 - 0–9
 - _ (underscore)
 - – (dash)
 - . (dot)
- Attribute names must be between 1 and 255 characters long.

Reserved Words and Special Characters

DynamoDB has a list of reserved words and special characters. For a complete list of reserved words in DynamoDB, see [Reserved Words in DynamoDB \(p. 1026\)](#). Also, the following characters have special meaning in DynamoDB: # (hash) and : (colon).

Although DynamoDB allows you to use these reserved words and special characters for names, we recommend that you avoid doing so because you have to define placeholder variables whenever you use these names in an expression. For more information, see [Expression Attribute Names in DynamoDB](#) (p. 389).

Data Types

DynamoDB supports many different data types for attributes within a table. They can be categorized as follows:

- **Scalar Types** – A scalar type can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.
 - **Document Types** – A document type can represent a complex structure with nested attributes, such as you would find in a JSON document. The document types are list and map.
 - **Set Types** – A set type can represent multiple scalar values. The set types are string set, number set, and binary set.

When you create a table or a secondary index, you must specify the names and data types of each primary key attribute (partition key and sort key). Furthermore, each primary key attribute must be defined as type string, number, or binary.

DynamoDB is a NoSQL database and is *schemaless*. This means that, other than the primary key attributes, you don't have to define any attributes or data types when you create tables. By comparison, relational databases require you to define the names and data types of each column when you create a table.

The following are descriptions of each data type, along with examples in JSON format.

Scalar Types

The scalar types are number, string, binary, Boolean, and null.

Number

Numbers can be positive, negative, or zero. Numbers can have up to 38 digits of precision. Exceeding this results in an exception.

In DynamoDB, numbers are represented as variable length. Leading and trailing zeroes are trimmed.

All numbers are sent across the network to DynamoDB as strings, to maximize compatibility across languages and libraries. However, DynamoDB treats them as number type attributes for mathematical operations.

Note

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from the number type.

You can use the number data type to represent a date or a timestamp. One way to do this is by using epoch time—the number of seconds since 00:00:00 UTC on 1 January 1970. For example, the epoch time 1437136300 represents 12:31:40 PM UTC on 17 July 2015.

For more information, see http://en.wikipedia.org/wiki/Unix_time.

String

Strings are Unicode with UTF-8 binary encoding. The minimum length of a string can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

The following additional constraints apply to primary key attributes that are defined as type string:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

DynamoDB collates and compares strings using the bytes of the underlying UTF-8 string encoding. For example, "a" (0x61) is greater than "A" (0x41), and "ż" (0xC2BF) is greater than "z" (0x7A).

You can use the string data type to represent a date or a timestamp. One way to do this is by using ISO 8601 strings, as shown in these examples:

- 2016-02-15
- 2015-12-21T17:42:34Z
- 20150311T122706Z

For more information, see http://en.wikipedia.org/wiki/ISO_8601.

Binary

Binary type attributes can store any binary data, such as compressed text, encrypted data, or images. Whenever DynamoDB compares binary values, it treats each byte of the binary data as unsigned.

The length of a binary attribute can be zero, if the attribute is not used as a key for an index or table, and is constrained by the maximum DynamoDB item size limit of 400 KB.

If you define a primary key attribute as a binary type attribute, the following additional constraints apply:

- For a simple primary key, the maximum length of the first attribute value (the partition key) is 2048 bytes.
- For a composite primary key, the maximum length of the second attribute value (the sort key) is 1024 bytes.

Your applications must encode binary values in base64-encoded format before sending them to DynamoDB. Upon receipt of these values, DynamoDB decodes the data into an unsigned byte array and uses that as the length of the binary attribute.

The following example is a binary attribute, using base64-encoded text.

```
dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk
```

Boolean

A Boolean type attribute can store either `true` or `false`.

Null

Null represents an attribute with an unknown or undefined state.

Document Types

The document types are list and map. These data types can be nested within each other, to represent complex data structures up to 32 levels deep.

There is no limit on the number of values in a list or a map, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

An attribute value can be an empty string or empty binary value if the attribute is not used for a table or index key. An attribute value cannot be an empty set (string set, number set, or binary set), however, empty lists and maps are allowed. Empty string and binary values are allowed within lists and maps. For more information, see [Attributes \(p. 965\)](#).

List

A list type attribute can store an ordered collection of values. Lists are enclosed in square brackets:

[...]

A list is similar to a JSON array. There are no restrictions on the data types that can be stored in a list element, and the elements in a list element do not have to be of the same type.

The following example shows a list that contains two strings and a number.

```
FavoriteThings: ["Cookies", "Coffee", 3.14159]
```

Note

DynamoDB lets you work with individual elements within lists, even if those elements are deeply nested. For more information, see [Using Expressions in DynamoDB \(p. 385\)](#).

Map

A map type attribute can store an unordered collection of name-value pairs. Maps are enclosed in curly braces: { ... }

A map is similar to a JSON object. There are no restrictions on the data types that can be stored in a map element, and the elements in a map do not have to be of the same type.

Maps are ideal for storing JSON documents in DynamoDB. The following example shows a map that contains a string, a number, and a nested list that contains another map.

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity : 3},
            Pencils: { Quantity : 2},
            Erasers: { Quantity : 1}
        }
    ]
}
```

```
    ]  
}
```

Note

DynamoDB lets you work with individual elements within maps, even if those elements are deeply nested. For more information, see [Using Expressions in DynamoDB \(p. 385\)](#).

Sets

DynamoDB supports types that represent sets of number, string, or binary values. All the elements within a set must be of the same type. For example, an attribute of type Number Set can only contain numbers; String Set can only contain strings; and so on.

There is no limit on the number of values in a set, as long as the item containing the values fits within the DynamoDB item size limit (400 KB).

Each value within a set must be unique. The order of the values within a set is not preserved. Therefore, your applications must not rely on any particular order of elements within the set. DynamoDB does not support empty sets, however, empty string and binary values are allowed within a set.

The following example shows a string set, a number set, and a binary set:

```
["Black", "Green", "Red"]  
[42.2, -19, 7.5, 3.14]  
[U3Vubnk=, UmFpbnk=, U25vd3k=]
```

Read Consistency

Amazon DynamoDB is available in multiple AWS Regions around the world. Each Region is independent and isolated from other AWS Regions. For example, if you have a table called *People* in the *us-east-2* Region and another table named *People* in the *us-west-2* Region, these are considered two entirely separate tables. For a list of all the AWS Regions in which DynamoDB is available, see [AWS Regions and Endpoints](#) in the *Amazon Web Services General Reference*.

Every AWS Region consists of multiple distinct locations called Availability Zones. Each Availability Zone is isolated from failures in other Availability Zones, and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region. This allows rapid replication of your data among multiple Availability Zones in a Region.

When your application writes data to a DynamoDB table and receives an HTTP 200 response (OK), the write has occurred and is durable. The data is eventually consistent across all storage locations, usually within one second or less.

DynamoDB supports *eventually consistent* and *strongly consistent* reads.

Eventually Consistent Reads

When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data.

Strongly Consistent Reads

When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. However, this consistency comes with some disadvantages:

- A strongly consistent read might not be available if there is a network delay or outage. In this case, DynamoDB may return a server error (HTTP 500).
- Strongly consistent reads may have higher latency than eventually consistent reads.
- Strongly consistent reads are not supported on global secondary indexes.
- Strongly consistent reads use more throughput capacity than eventually consistent reads. For details, see [Read/Write Capacity Mode \(p. 17\)](#)

Note

DynamoDB uses eventually consistent reads, unless you specify otherwise. Read operations (such as `GetItem`, `Query`, and `Scan`) provide a `ConsistentRead` parameter. If you set this parameter to true, DynamoDB uses strongly consistent reads during the operation.

Read/Write Capacity Mode

Amazon DynamoDB has two read/write capacity modes for processing reads and writes on your tables:

- On-demand
- Provisioned (default, free-tier eligible)

The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity. You can set the read/write capacity mode when creating a table or you can change it later.

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations When Changing Read/Write Capacity Mode \(p. 340\)](#).

Topics

- [On-Demand Mode \(p. 17\)](#)
- [Provisioned Mode \(p. 19\)](#)

On-Demand Mode

Amazon DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use.

When you choose on-demand mode, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload. Tables that use on-demand mode deliver the same single-digit millisecond latency, service-level agreement (SLA) commitment, and security that DynamoDB already offers. You can choose on-demand for both new and existing tables and you can continue using the existing DynamoDB APIs without changing code.

On-demand mode is a good option if any of the following are true:

- You create new tables with unknown workloads.
- You have unpredictable application traffic.
- You prefer the ease of paying for only what you use.

The request rate is only limited by the DynamoDB throughput default table limits, but it can be raised upon request. For more information, see [Throughput Default Limits \(p. 961\)](#).

To get started with on-demand, you can create or update a table to use on-demand mode. For more information, see [Basic Operations on DynamoDB Tables \(p. 335\)](#).

You can switch between read/write capacity modes once every 24 hours. For issues you should consider when switching read/write capacity modes, see [Considerations When Changing Read/Write Capacity Mode \(p. 340\)](#).

Note

On-demand is currently not supported by the DynamoDB import/export tool.

Topics

- [Read Request Units and Write Request Units \(p. 18\)](#)
- [Peak Traffic and Scaling Properties \(p. 18\)](#)
- [Initial Throughput for On-Demand Capacity Mode \(p. 19\)](#)
- [Table Behavior while Switching Read/Write Capacity Mode \(p. 19\)](#)

Read Request Units and Write Request Units

For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB charges you for the reads and writes that your application performs on your tables in terms of read request units and write request units.

- One *read request unit* represents one strongly consistent read request, or two eventually consistent read requests, for an item up to 4 KB in size. Transactional read requests require 2 read request units to perform one read for items up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB needs additional read request units. The total number of read request units required depends on the item size, and whether you want an eventually consistent or strongly consistent read. For example, if your item size is 8 KB, you require 2 read request units to sustain one strongly consistent read, 1 read request unit if you choose eventually consistent reads, or 4 read request units for a transactional read request.

Note

To learn more about DynamoDB read consistency models, see [Read Consistency \(p. 16\)](#).

- One *write request unit* represents one write for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB needs to consume additional write request units. Transactional write requests require 2 write request units to perform one write for items up to 1 KB. The total number of write request units required depends on the item size. For example, if your item size is 2 KB, you require 2 write request units to sustain one write request or 4 write request units for a transactional write request.

For a list of AWS Regions where DynamoDB on-demand is available, see [Amazon DynamoDB Pricing](#).

Peak Traffic and Scaling Properties

DynamoDB tables using on-demand capacity mode automatically adapt to your application's traffic volume. On-demand capacity mode instantly accommodates up to double the previous peak traffic on a table. For example, if your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second where 50,000 reads per second is the previous traffic peak, on-demand capacity mode instantly accommodates sustained traffic of up to 100,000 reads per second. If your application sustains traffic of 100,000 reads per second, that peak becomes your new previous peak, enabling subsequent traffic to reach up to 200,000 reads per second.

If you need more than double your previous peak on table, DynamoDB automatically allocates more capacity as your traffic volume increases to help ensure that your workload does not experience throttling. However, throttling can occur if you exceed double your previous peak within 30 minutes. For example, if your application's traffic pattern varies between 25,000 and 50,000 strongly consistent reads per second where 50,000 reads per second is the previously reached traffic peak, DynamoDB

recommends spacing your traffic growth over at least 30 minutes before driving more than 100,000 reads per second.

Initial Throughput for On-Demand Capacity Mode

If you recently switched an existing table to on-demand capacity mode for the first time, or if you created a new table with on-demand capacity mode enabled, the table has the following previous peak settings, even though the table has not served traffic previously using on-demand capacity mode:

- **Newly created table with on-demand capacity mode:** The previous peak is 2,000 write request units or 6,000 read request units. You can drive up to double the previous peak immediately, which enables newly created on-demand tables to serve up to 4,000 write request units or 12,000 read request units, or any linear combination of the two.
- **Existing table switched to on-demand capacity mode:** The previous peak is half the maximum write capacity units and read capacity units provisioned since the table was created, or the settings for a newly created table with on-demand capacity mode, whichever is higher. In other words, your table will deliver at least as much throughput as it did prior to switching to on-demand capacity mode.

Table Behavior while Switching Read/Write Capacity Mode

When you switch a table from provisioned capacity mode to on-demand capacity mode, DynamoDB makes several changes to the structure of your table and partitions. This process can take several minutes. During the switching period, your table delivers throughput that is consistent with the previously provisioned write capacity unit and read capacity unit amounts. When switching from on-demand capacity mode back to provisioned capacity mode, your table delivers throughput consistent with the previous peak reached when the table was set to on-demand capacity mode.

Provisioned Mode

If you choose provisioned mode, you specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes. This helps you govern your DynamoDB use to stay at or below a defined request rate in order to obtain cost predictability.

Provisioned mode is a good option if any of the following are true:

- You have predictable application traffic.
- You run applications whose traffic is consistent or ramps gradually.
- You can forecast capacity requirements to control costs.

Read Capacity Units and Write Capacity Units

For provisioned mode tables, you specify throughput capacity in terms of read capacity units (RCUs) and write capacity units (WCUs):

- One *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. Transactional read requests require two read capacity units to perform one read per second for items up to 4 KB. If you need to read an item that is larger than 4 KB, DynamoDB must consume additional read capacity units. The total number of read capacity units required depends on the item size, and whether you want an eventually consistent or strongly consistent read. For example, if your item size is 8 KB, you require 2 read capacity units to sustain one strongly consistent read per second, 1 read capacity unit if you choose eventually consistent reads, or 4 read capacity units for a transactional read request. For more information, see [Capacity Unit Consumption for Reads \(p. 342\)](#).

Note

To learn more about DynamoDB read consistency models, see [Read Consistency \(p. 16\)](#).

- One *write capacity unit* represents one write per second for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB must consume additional write capacity units. Transactional write requests require 2 write capacity units to perform one write per second for items up to 1 KB. The total number of write capacity units required depends on the item size. For example, if your item size is 2 KB, you require 2 write capacity units to sustain one write request per second or 4 write capacity units for a transactional write request. For more information, see [Capacity Unit Consumption for Writes \(p. 342\)](#).

Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

If your application reads or writes larger items (up to the DynamoDB maximum item size of 400 KB), it will consume more capacity units.

For example, suppose that you create a provisioned table with 6 read capacity units and 6 write capacity units. With these settings, your application could do the following:

- Perform strongly consistent reads of up to 24 KB per second ($4\text{ KB} \times 6$ read capacity units).
- Perform eventually consistent reads of up to 48 KB per second (twice as much read throughput).
- Perform transactional read requests of up to 12 KB per second.
- Write up to 6 KB per second ($1\text{ KB} \times 6$ write capacity units).
- Perform transactional write requests of up to 3 KB per second.

For more information, see [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#).

Provisioned throughput is the maximum amount of capacity that an application can consume from a table or index. If your application exceeds your provisioned throughput capacity on a table or index, it is subject to request throttling.

Throttling prevents your application from consuming too many capacity units.

When a request is throttled, it fails with an HTTP 400 code (`Bad Request`) and a `ProvisionedThroughputExceededException`. The AWS SDKs have built-in support for retrying throttled requests (see [Error Retries and Exponential Backoff \(p. 224\)](#)), so you do not need to write this logic yourself.

You can use the AWS Management Console to monitor your provisioned and actual throughput, and to modify your throughput settings if necessary.

DynamoDB Auto Scaling

DynamoDB auto scaling actively manages throughput capacity for tables and global secondary indexes. With auto scaling, you define a range (upper and lower limits) for read and write capacity units. You also define a target utilization percentage within that range. DynamoDB auto scaling seeks to maintain your target utilization, even as your application workload increases or decreases.

With DynamoDB auto scaling, a table or a global secondary index can increase its provisioned read and write capacity to handle sudden increases in traffic, without request throttling. When the workload decreases, DynamoDB auto scaling can decrease the throughput so that you don't pay for unused provisioned capacity.

Note

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default.

You can manage auto scaling settings at any time by using the console, the AWS CLI, or one of the AWS SDKs.

For more information, see [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#).

Reserved Capacity

As a DynamoDB customer, you can purchase *reserved capacity* in advance, as described at [Amazon DynamoDB Pricing](#). With reserved capacity, you pay a one-time upfront fee and commit to a minimum provisioned usage level over a period of time. Your reserved capacity is billed at the hourly reserved capacity rate. By reserving your read and write capacity units ahead of time, you realize significant cost savings compared to on-demand provisioned throughput settings. Any capacity that you provision in excess of your reserved capacity is billed at standard provisioned capacity rates.

Note

Reserved capacity is not available in on-demand mode.

To manage reserved capacity, go to the [DynamoDB console](#) and choose **Reserved Capacity**.

Note

You can prevent users from viewing or purchasing reserved capacity, while still allowing them to access the rest of the console. For more information, see "Grant Permissions to Prevent Purchasing of Reserved Capacity Offerings" in [Identity and Access Management in Amazon DynamoDB \(p. 824\)](#).

Partitions and Data Distribution

Amazon DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB—you never have to manage partitions yourself.

When you create a table, the initial status of the table is `CREATING`. During this phase, DynamoDB allocates sufficient partitions to the table so that it can handle your provisioned throughput requirements. You can begin writing and reading table data after the table status changes to `ACTIVE`.

DynamoDB allocates additional partitions to a table in the following situations:

- If you increase the table's provisioned throughput settings beyond what the existing partitions can support.
- If an existing partition fills to capacity and more storage space is required.

Partition management occurs automatically in the background and is transparent to your applications. Your table remains available throughout and fully supports your provisioned throughput requirements.

For more details, see [Partition Key Design \(p. 900\)](#).

Global secondary indexes in DynamoDB are also composed of partitions. The data in a global secondary index is stored separately from the data in its base table, but index partitions behave in much the same way as table partitions.

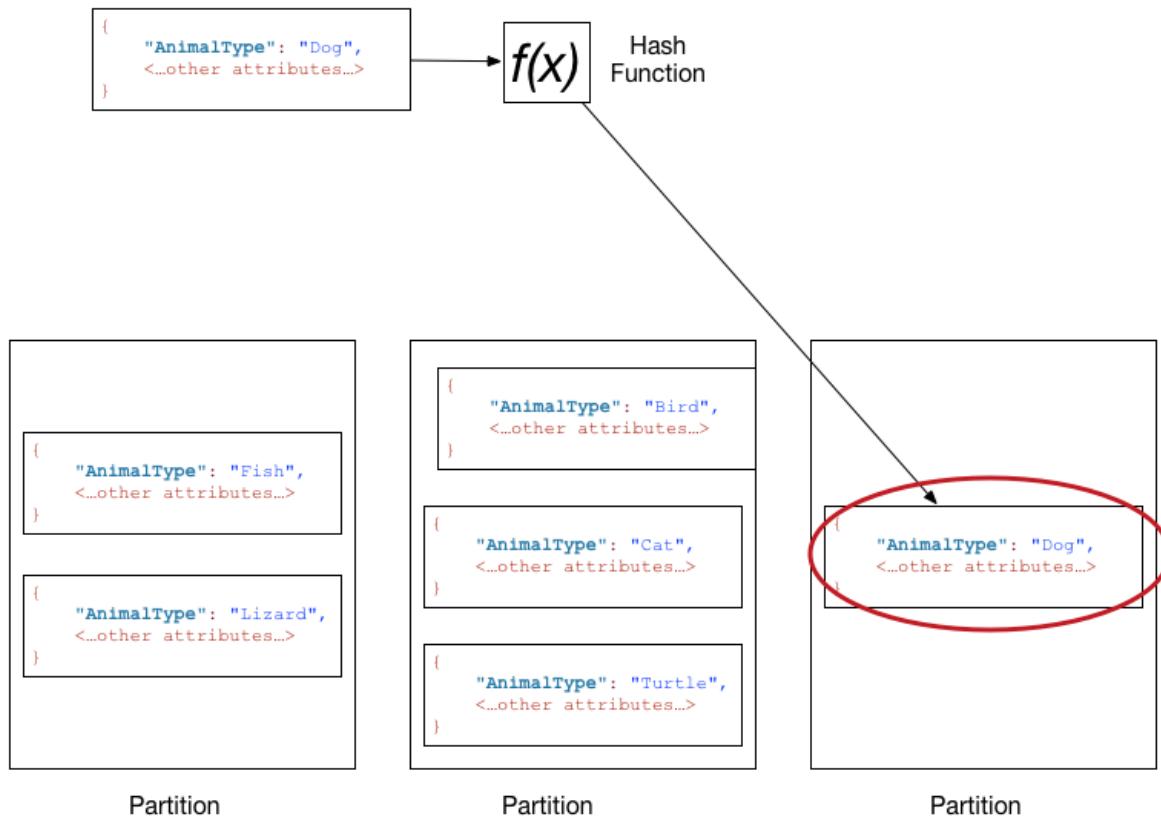
Data Distribution: Partition Key

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value.

To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.

To read an item from the table, you must specify the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.

The following diagram shows a table named *Pets*, which spans multiple partitions. The table's primary key is *AnimalType* (only this key attribute is shown). DynamoDB uses its hash function to determine where to store a new item, in this case based on the hash value of the string *Dog*. Note that the items are not stored in sorted order. Each item's location is determined by the hash value of its partition key.



Note

DynamoDB is optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be. We recommend that you choose a partition key that can have a large number of distinct values relative to the number of items in the table.

Data Distribution: Partition Key and Sort Key

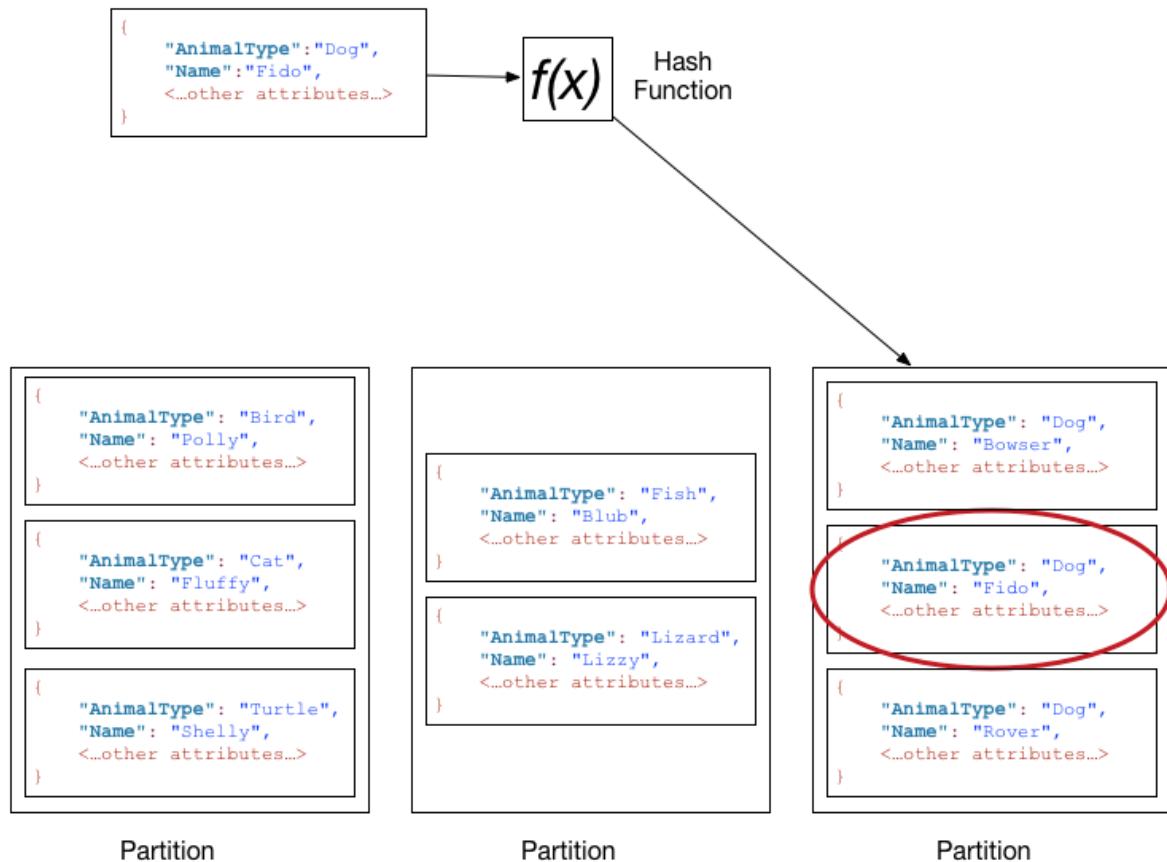
If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way as described in [Data Distribution: Partition Key \(p. 21\)](#). However, it stores all the items with the same partition key value physically close together, ordered by sort key value.

To write an item to the table, DynamoDB calculates the hash value of the partition key to determine which partition should contain the item. In that partition, several items could have the same partition key value. So DynamoDB stores the item among the others with the same partition key, in ascending order by sort key.

To read an item from the table, you must specify its partition key value and sort key value. DynamoDB calculates the partition key's hash value, yielding the partition in which the item can be found.

You can read multiple items from the table in a single operation (*Query*) if the items you want have the same partition key value. DynamoDB returns all of the items with that partition key value. Optionally, you can apply a condition to the sort key so that it returns only the items within a certain range of values.

Suppose that the *Pets* table has a composite primary key consisting of *AnimalType* (partition key) and *Name* (sort key). The following diagram shows DynamoDB writing an item with a partition key value of *Dog* and a sort key value of *Fido*.



To read that same item from the *Pets* table, DynamoDB calculates the hash value of *Dog*, yielding the partition in which these items are stored. DynamoDB then scans the sort key attribute values until it finds *Fido*.

To read all of the items with an *AnimalType* of *Dog*, you can issue a `Query` operation without specifying a sort key condition. By default, the items are returned in the order that they are stored (that is, in ascending order by sort key). Optionally, you can request descending order instead.

To query only some of the *Dog* items, you can apply a condition to the sort key (for example, only the *Dog* items where *Name* begins with a letter that is within the range A through K).

Note

In a DynamoDB table, there is no upper limit on the number of distinct sort key values per partition key value. If you needed to store many billions of *Dog* items in the *Pets* table, DynamoDB would allocate enough storage to handle this requirement automatically.

From SQL to NoSQL

If you are an application developer, you might have some experience using a relational database management system (RDBMS) and Structured Query Language (SQL). As you begin working with Amazon DynamoDB, you will encounter many similarities, but also many things that are different. This section describes common database tasks, comparing and contrasting SQL statements with their equivalent DynamoDB operations.

NoSQL is a term used to describe nonrelational database systems that are highly available, scalable, and optimized for high performance. Instead of the relational model, NoSQL databases (like DynamoDB) use alternate models for data management, such as key-value pairs or document storage. For more information, see <http://aws.amazon.com/nosql>.

Note

The SQL examples in this section are compatible with the MySQL RDBMS.

The DynamoDB examples in this section show the name of the DynamoDB operation, along with the parameters for that operation in JSON format. For code examples that use these operations, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Topics

- [Relational \(SQL\) or NoSQL? \(p. 24\)](#)
- [Characteristics of Databases \(p. 25\)](#)
- [Creating a Table \(p. 27\)](#)
- [Getting Information About a Table \(p. 29\)](#)
- [Writing Data to a Table \(p. 30\)](#)
- [Key Differences When Reading Data from a Table \(p. 32\)](#)
- [Managing Indexes \(p. 37\)](#)
- [Modifying Data in a Table \(p. 40\)](#)
- [Deleting Data from a Table \(p. 42\)](#)
- [Removing a Table \(p. 43\)](#)

Relational (SQL) or NoSQL?

Today's applications have more demanding requirements than ever before. For example, an online game might start out with just a few users and a very small amount of data. However, if the game becomes successful, it can easily outstrip the resources of the underlying database management system. It is common for web-based applications to have hundreds, thousands, or millions of concurrent users, with terabytes or more of new data generated per day. Databases for such applications must handle tens (or hundreds) of thousands of reads and writes per second.

Amazon DynamoDB is well-suited for these kinds of workloads. As a developer, you can start small and gradually increase your utilization as your application becomes more popular. DynamoDB scales seamlessly to handle very large amounts of data and very large numbers of users.

The following table shows some high-level differences between an RDBMS and DynamoDB.

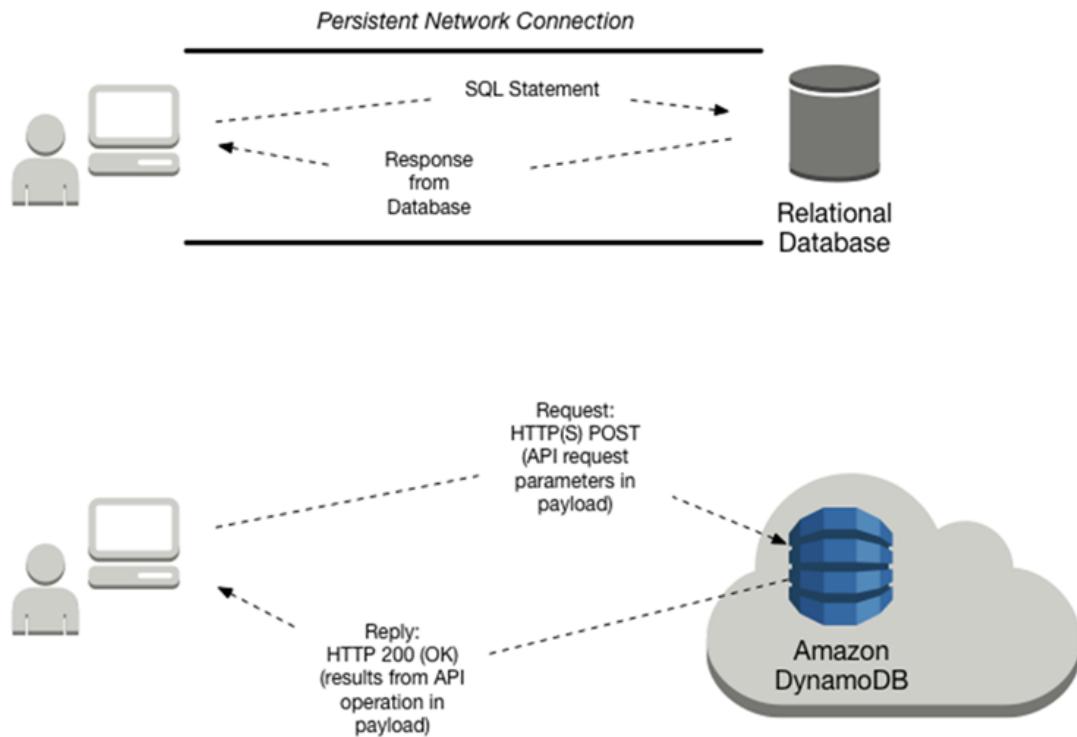
| Characteristic | Relational Database Management System (RDBMS) | Amazon DynamoDB |
|--------------------------|--|--|
| Optimal Workloads | Ad hoc queries; data warehousing; OLAP (online analytical processing). | Web-scale applications, including social networks, gaming, media sharing, and Internet of Things (IoT). |
| Data Model | The relational model requires a well-defined schema, where data is normalized into tables, rows, and columns. In addition, all of the relationships are defined among tables, columns, indexes, and other database elements. | DynamoDB is schemaless. Every table must have a primary key to uniquely identify each data item, but there are no similar constraints on other non-key attributes. DynamoDB can manage structured or |

| Characteristic | Relational Database Management System (RDBMS) | Amazon DynamoDB |
|--------------------|--|---|
| | | semistructured data, including JSON documents. |
| Data Access | SQL is the standard for storing and retrieving data. Relational databases offer a rich set of tools for simplifying the development of database-driven applications, but all of these tools use SQL. | You can use the AWS Management Console or the AWS CLI to work with DynamoDB and perform ad hoc tasks. Applications can use the AWS software development kits (SDKs) to work with DynamoDB using object-based, document-centric, or low-level interfaces. |
| Performance | Relational databases are optimized for storage, so performance generally depends on the disk subsystem. Developers and database administrators must optimize queries, indexes, and table structures in order to achieve peak performance. | DynamoDB is optimized for compute, so performance is mainly a function of the underlying hardware and network latency. As a managed service, DynamoDB insulates you and your applications from these implementation details, so that you can focus on designing and building robust, high-performance applications. |
| Scaling | It is easiest to scale up with faster hardware. It is also possible for database tables to span across multiple hosts in a distributed system, but this requires additional investment. Relational databases have maximum sizes for the number and size of files, which imposes upper limits on scalability. | DynamoDB is designed to scale out using distributed clusters of hardware. This design allows increased throughput without increased latency. Customers specify their throughput requirements, and DynamoDB allocates sufficient resources to meet those requirements. There are no upper limits on the number of items per table, nor the total size of that table. |

Characteristics of Databases

Before your application can access a database, it must be *authenticated* to ensure that the application is allowed to use the database. It must be *authorized* so that the application can perform only the actions for which it has permissions.

The following diagram shows a client's interaction with a relational database and with Amazon DynamoDB.



The following table has more details about client interaction tasks.

| Characteristic | Relational Database Management System (RDBMS) | Amazon DynamoDB |
|---|--|--|
| Tools for Accessing the Database | Most relational databases provide a command line interface (CLI) so that you can enter ad hoc SQL statements and see the results immediately. | In most cases, you write application code. You can also use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to send ad hoc requests to DynamoDB and view the results. |
| Connecting to the Database | An application program establishes and maintains a network connection with the database. When the application is finished, it terminates the connection. | DynamoDB is a web service, and interactions with it are stateless. Applications do not need to maintain persistent network connections. Instead, interaction with DynamoDB occurs using HTTP(S) requests and responses. |
| Authentication | An application cannot connect to the database until it is authenticated. The RDBMS can perform the authentication itself, or it can offload this task to the host operating system or a directory service. | Every request to DynamoDB must be accompanied by a cryptographic signature, which authenticates that particular request. The AWS SDKs provide all of the logic necessary for creating signatures and signing requests. For more information, |

| Characteristic | Relational Database Management System (RDBMS) | Amazon DynamoDB |
|-----------------------------|--|--|
| | | see Signing AWS API Requests in the AWS General Reference . |
| Authorization | Applications can perform only the actions for which they have been authorized. Database administrators or application owners can use the SQL GRANT and REVOKE statements to control access to database objects (such as tables), data (such as rows within a table), or the ability to issue certain SQL statements. | In DynamoDB, authorization is handled by AWS Identity and Access Management (IAM). You can write an IAM policy to grant permissions on a DynamoDB resource (such as a table), and then allow IAM users and roles to use that policy. IAM also features fine-grained access control for individual data items in DynamoDB tables. For more information, see Identity and Access Management in Amazon DynamoDB (p. 824). |
| Sending a Request | The application issues a SQL statement for every database operation that it wants to perform. Upon receipt of the SQL statement, the RDBMS checks its syntax, creates a plan for performing the operation, and then executes the plan. | The application sends HTTP(S) requests to DynamoDB. The requests contain the name of the DynamoDB operation to perform, along with parameters. DynamoDB executes the request immediately. |
| Receiving a Response | The RDBMS returns the results from the SQL statement. If there is an error, the RDBMS returns an error status and message. | DynamoDB returns an HTTP(S) response containing the results of the operation. If there is an error, DynamoDB returns an HTTP error status and messages. |

Creating a Table

Tables are the fundamental data structures in relational databases and in Amazon DynamoDB. A relational database management system (RDBMS) requires you to define the table's schema when you create it. In contrast, DynamoDB tables are schemaless—other than the primary key, you do not need to define any extra attributes or data types when you create a table.

Topics

- [SQL \(p. 27\)](#)
- [DynamoDB \(p. 28\)](#)

SQL

Use the `CREATE TABLE` statement to create a table, as shown in the following example.

```
CREATE TABLE Music (
    Artist VARCHAR(20) NOT NULL,
    SongTitle VARCHAR(30) NOT NULL,
    AlbumTitle VARCHAR(25),
```

```

    Year INT,
    Price FLOAT,
    Genre VARCHAR(10),
    Tags TEXT,
    PRIMARY KEY(Artist, SongTitle)
);

```

The primary key for this table consists of *Artist* and *SongTitle*.

You must define all of the table's columns and data types, and the table's primary key. (You can use the `ALTER TABLE` statement to change these definitions later, if necessary.)

Many SQL implementations let you define storage specifications for your table, as part of the `CREATE TABLE` statement. Unless you indicate otherwise, the table is created with default storage settings. In a production environment, a database administrator can help determine the optimal storage parameters.

DynamoDB

Use the `CreateTable` action to create a provisioned mode table, specifying parameters as shown following:

```

{
    TableName : "Music",
    KeySchema: [
        {
            AttributeName: "Artist",
            KeyType: "HASH", //Partition key
        },
        {
            AttributeName: "SongTitle",
            KeyType: "RANGE" //Sort key
        }
    ],
    AttributeDefinitions: [
        {
            AttributeName: "Artist",
            AttributeType: "S"
        },
        {
            AttributeName: "SongTitle",
            AttributeType: "S"
        }
    ],
    ProvisionedThroughput: {           // Only specified if using provisioned mode
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
}

```

The primary key for this table consists of *Artist* (partition key) and *SongTitle* (sort key).

You must provide the following parameters to `CreateTable`:

- `TableName` – Name of the table.
- `KeySchema` – Attributes that are used for the primary key. For more information, see [Tables, Items, and Attributes \(p. 3\)](#) and [Primary Key \(p. 6\)](#).
- `AttributeDefinitions` – Data types for the key schema attributes.
- `ProvisionedThroughput` (for provisioned tables) – Number of reads and writes per second that you need for this table. DynamoDB reserves sufficient storage and system resources so that your throughput requirements are always met. You can use the `UpdateTable` action to change these later,

if necessary. You do not need to specify a table's storage requirements because storage allocation is managed entirely by DynamoDB.

Note

For code examples that use `CreateTable`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Getting Information About a Table

You can verify that a table has been created according to your specifications. In a relational database, all of the table's schema is shown. Amazon DynamoDB tables are schemaless, so only the primary key attributes are shown.

Topics

- [SQL \(p. 29\)](#)
- [DynamoDB \(p. 29\)](#)

SQL

Most relational database management systems (RDBMS) allow you to describe a table's structure—columns, data types, primary key definition, and so on. There is no standard way to do this in SQL. However, many database systems provide a `DESCRIBE` command. The following is an example from MySQL.

```
DESCRIBE Music;
```

This returns the structure of your table, with all of the column names, data types, and sizes.

| Field | Type | Null | Key | Default | Extra |
|------------|-------------|------|-----|---------|-------|
| Artist | varchar(20) | NO | PRI | NULL | |
| SongTitle | varchar(30) | NO | PRI | NULL | |
| AlbumTitle | varchar(25) | YES | | NULL | |
| Year | int(11) | YES | | NULL | |
| Price | float | YES | | NULL | |
| Genre | varchar(10) | YES | | NULL | |
| Tags | text | YES | | NULL | |

The primary key for this table consists of *Artist* and *SongTitle*.

DynamoDB

DynamoDB has a `DescribeTable` action, which is similar. The only parameter is the table name.

```
{  
    TableName : "Music"  
}
```

The reply from `DescribeTable` looks like the following.

```
{  
    "Table": {  
        "AttributeDefinitions": [  
            {
```

```
        "AttributeName": "Artist",
        "AttributeType": "S"
    },
    {
        "AttributeName": "SongTitle",
        "AttributeType": "S"
    }
],
"TableName": "Music",
"KeySchema": [
    {
        "AttributeName": "Artist",
        "KeyType": "HASH"    //Partition key
    },
    {
        "AttributeName": "SongTitle",
        "KeyType": "RANGE"   //Sort key
    }
],
...
...
```

DescribeTable also returns information about indexes on the table, provisioned throughput settings, an approximate item count, and other metadata.

Writing Data to a Table

Relational database tables contain *rows* of data. Rows are composed of *columns*. Amazon DynamoDB tables contain *items*. Items are composed of *attributes*.

This section describes how to write one row (or item) to a table.

Topics

- [SQL \(p. 30\)](#)
- [DynamoDB \(p. 31\)](#)

SQL

A table in a relational database is a two-dimensional data structure composed of rows and columns. Some database management systems also provide support for semistructured data, usually with native JSON or XML data types. However, the implementation details vary among vendors.

In SQL, you use the `INSERT` statement to add a row to a table.

```
INSERT INTO Music
  (Artist, SongTitle, AlbumTitle,
  Year, Price, Genre,
  Tags)
VALUES(
  'No One You Know', 'Call Me Today', 'Somewhat Famous',
  2015, 2.14, 'Country',
  '{"Composers": ["Smith", "Jones", "Davis"], "LengthInSeconds": 214}')
);
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these columns.

Note

This example uses the *Tags* column to store semistructured data about the songs in the *Music* table. The *Tags* column is defined as type TEXT, which can store up to 65,535 characters in MySQL.

DynamoDB

In Amazon DynamoDB, you use the `PutItem` action to add an item to a table.

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Today",  
        "AlbumTitle": "Somewhat Famous",  
        "Year": 2015,  
        "Price": 2.14,  
        "Genre": "Country",  
        "Tags": {  
            "Composers": [  
                "Smith",  
                "Jones",  
                "Davis"  
            ],  
            "LengthInSeconds": 214  
        }  
    }  
}
```

The primary key for this table consists of *Artist* and *SongTitle*. You must specify values for these attributes.

Here are some key things to know about this `PutItem` example:

- DynamoDB provides native support for documents, using JSON. This makes DynamoDB ideal for storing semistructured data, such as *Tags*. You can also retrieve and manipulate data from within JSON documents.
- The *Music* table does not have any predefined attributes, other than the primary key (*Artist* and *SongTitle*).
- Most SQL databases are transaction oriented. When you issue an `INSERT` statement, the data modifications are not permanent until you issue a `COMMIT` statement. With Amazon DynamoDB, the effects of a `PutItem` action are permanent when DynamoDB replies with an HTTP 200 status code (OK).

Note

For code examples using `PutItem`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

The following are some other `PutItem` examples.

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "No One You Know",  
        "SongTitle": "My Dog Spot",  
        "AlbumTitle": "Hey Now",  
        "Price": 1.98,  
        "Genre": "Country",  
        "CriticRating": 8.4  
    }  
}
```

```
{  
    TableName: "Music",  
}
```

```
Item: {  
    "Artist": "No One You Know",  
    "SongTitle": "Somewhere Down The Road",  
    "AlbumTitle": "Somewhat Famous",  
    "Genre": "Country",  
    "CriticRating": 8.4,  
    "Year": 1984  
}  
}
```

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "The Acme Band",  
        "SongTitle": "Still In Love",  
        "AlbumTitle": "The Buck Starts Here",  
        "Price": 2.47,  
        "Genre": "Rock",  
        "PromotionInfo": {  
            "RadioStationsPlaying": [  
                "KHCR", "KBQX", "WTNR", "WJJH"  
            ],  
            "TourDates": {  
                "Seattle": "20150625",  
                "Cleveland": "20150630"  
            },  
            "Rotation": "Heavy"  
        }  
    }  
}
```

```
{  
    TableName: "Music",  
    Item: {  
        "Artist": "The Acme Band",  
        "SongTitle": "Look Out, World",  
        "AlbumTitle": "The Buck Starts Here",  
        "Price": 0.99,  
        "Genre": "Rock"  
    }  
}
```

Note

In addition to `PutItem`, DynamoDB supports a `BatchWriteItem` action for writing multiple items at the same time.

Key Differences When Reading Data from a Table

With SQL, you use the `SELECT` statement to retrieve one or more rows from a table. You use the `WHERE` clause to determine the data that is returned to you.

Amazon DynamoDB provides the following operations for reading data:

- `GetItem` – Retrieves a single item from a table. This is the most efficient way to read a single item because it provides direct access to the physical location of the item. (DynamoDB also provides the `BatchGetItem` operation, allowing you to perform up to 100 `GetItem` calls in a single operation.)
- `Query` – Retrieves all of the items that have a specific partition key. Within those items, you can apply a condition to the sort key and retrieve only a subset of the data. `Query` provides quick, efficient access to the partitions where the data is stored. (For more information, see [Partitions and Data Distribution \(p. 21\)](#).)

- Scan – Retrieves all of the items in the specified table. (This operation should not be used with large tables because it can consume large amounts of system resources.)

Note

With a relational database, you can use the `SELECT` statement to join data from multiple tables and return the results. Joins are fundamental to the relational model. To ensure that joins execute efficiently, the database and its applications should be performance-tuned on an ongoing basis. DynamoDB is a non-relational NoSQL database that does not support table joins. Instead, applications read data from one table at a time.

The following sections describe different use cases for reading data, and how to perform these tasks with a relational database and with DynamoDB.

Topics

- [Reading an Item Using Its Primary Key \(p. 33\)](#)
- [Querying a Table \(p. 34\)](#)
- [Scanning a Table \(p. 36\)](#)

Reading an Item Using Its Primary Key

One common access pattern for databases is to read a single item from a table. You have to specify the primary key of the item you want.

Topics

- [SQL \(p. 33\)](#)
- [DynamoDB \(p. 33\)](#)

SQL

In SQL, you use the `SELECT` statement to retrieve data from a table. You can request one or more columns in the result (or all of them, if you use the `*` operator). The `WHERE` clause determines which rows to return.

The following is a `SELECT` statement to retrieve a single row from the *Music* table. The `WHERE` clause specifies the primary key values.

```
SELECT *
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

You can modify this query to retrieve only a subset of the columns.

```
SELECT AlbumTitle, Year, Price
FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today'
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

DynamoDB

DynamoDB provides the `GetItem` action for retrieving an item by its primary key. `GetItem` is highly efficient because it provides direct access to the physical location of the item. (For more information, see [Partitions and Data Distribution \(p. 21\)](#).)

By default, `GetItem` returns the entire item with all of its attributes.

```
{  
    TableName: "Music",  
    Key: {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Today"  
    }  
}
```

You can add a `ProjectionExpression` parameter to return only some of the attributes.

```
{  
    TableName: "Music",  
    Key: {  
        "Artist": "No One You Know",  
        "SongTitle": "Call Me Today"  
    },  
    "ProjectionExpression": "AlbumTitle, Year, Price"  
}
```

Note that the primary key for this table consists of `Artist` and `SongTitle`.

The DynamoDB `GetItem` action is very efficient: It uses the primary key values to determine the exact storage location of the item in question, and retrieves it directly from there. The SQL `SELECT` statement is similarly efficient, in the case of retrieving items by primary key values.

The SQL `SELECT` statement supports many kinds of queries and table scans. DynamoDB provides similar functionality with its `Query` and `Scan` actions, which are described in [Querying a Table \(p. 34\)](#) and [Scanning a Table \(p. 36\)](#).

The SQL `SELECT` statement can perform table joins, allowing you to retrieve data from multiple tables at the same time. Joins are most effective where the database tables are normalized and the relationships among the tables are clear. However, if you join too many tables in one `SELECT` statement application performance can be affected. You can work around such issues by using database replication, materialized views, or query rewrites.

DynamoDB is a nonrelational database and doesn't support table joins. If you are migrating an existing application from a relational database to DynamoDB, you need to denormalize your data model to eliminate the need for joins.

Note

For code examples that use `GetItem`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Querying a Table

Another common access pattern is reading multiple items from a table, based on your query criteria.

Topics

- [SQL \(p. 34\)](#)
- [DynamoDB \(p. 35\)](#)

SQL

The SQL `SELECT` statement lets you query on key columns, non-key columns, or any combination. The `WHERE` clause determines which rows are returned, as shown in the following examples.

```
/* Return a single song, by primary key */
```

```
SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today';
```

```
/* Return all of the songs by an artist */

SELECT * FROM Music
WHERE Artist='No One You Know';
```

```
/* Return all of the songs by an artist, matching first part of title */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle LIKE 'Call%';
```

```
/* Return all of the songs by an artist, with a particular word in the title...
...but only if the price is less than 1.00 */

SELECT * FROM Music
WHERE Artist='No One You Know' AND SongTitle LIKE '%Today%'
AND Price < 1.00;
```

Note that the primary key for this table consists of *Artist* and *SongTitle*.

DynamoDB

The Amazon DynamoDB Query action lets you retrieve data in a similar fashion. The Query action provides quick, efficient access to the physical locations where the data is stored. For more information, see [Partitions and Data Distribution \(p. 21\)](#).

You can use Query with any table that has a composite primary key (partition key and sort key). You must specify an equality condition for the partition key, and you can optionally provide another condition for the sort key.

The KeyConditionExpression parameter specifies the key values that you want to query. You can use an optional FilterExpression to remove certain items from the results before they are returned to you.

In DynamoDB, you must use ExpressionAttributeValues as placeholders in expression parameters (such as KeyConditionExpression and FilterExpression). This is analogous to the use of *bind variables* in relational databases, where you substitute the actual values into the SELECT statement at runtime.

Note that the primary key for this table consists of *Artist* and *SongTitle*.

The following are some DynamoDB Query examples.

```
// Return a single song, by primary key

{
    TableName: "Music",
    KeyConditionExpression: "Artist = :a and SongTitle = :t",
    ExpressionAttributeValues: {
        ":a": "No One You Know",
        ":t": "Call Me Today"
    }
}
```

```
// Return all of the songs by an artist
```

```
{  
    TableName: "Music",  
    KeyConditionExpression: "Artist = :a",  
    ExpressionAttributeValues: {  
        ":a": "No One You Know"  
    }  
}
```

```
// Return all of the songs by an artist, matching first part of title  
  
{  
    TableName: "Music",  
    KeyConditionExpression: "Artist = :a and begins_with(SongTitle, :t)",  
    ExpressionAttributeValues: {  
        ":a": "No One You Know",  
        ":t": "Call"  
    }  
}
```

Note

For code examples that use `Query`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Scanning a Table

In SQL, a `SELECT` statement without a `WHERE` clause will return every row in a table. In Amazon DynamoDB, the `Scan` operation does the same thing. In both cases, you can retrieve all of the items or just some of them.

Whether you are using a SQL or a NoSQL database, scans should be used sparingly because they can consume large amounts of system resources. Sometimes a scan is appropriate (such as scanning a small table) or unavoidable (such as performing a bulk export of data). However, as a general rule, you should design your applications to avoid performing scans.

Topics

- [SQL \(p. 36\)](#)
- [DynamoDB \(p. 36\)](#)

SQL

In SQL, you can scan a table and retrieve all of its data by using a `SELECT` statement without specifying a `WHERE` clause. You can request one or more columns in the result. Or you can request all of them if you use the wildcard character (*).

The following are examples of using a `SELECT` statement.

```
/* Return all of the data in the table */  
SELECT * FROM Music;
```

```
/* Return all of the values for Artist and Title */  
SELECT Artist, Title FROM Music;
```

DynamoDB

DynamoDB provides a `Scan` action that works in a similar way. The following are some examples.

```
// Return all of the data in the table
{
    TableName: "Music"
}
```

```
// Return all of the values for Artist and Title
{
    TableName: "Music",
    ProjectionExpression: "Artist, Title"
}
```

The `Scan` action also provides a `FilterExpression` parameter, which you can use to discard items that you do not want to appear in the results. A `FilterExpression` is applied after the entire table is scanned, but before the results are returned to you. (This is not recommended with large tables. You are still charged for the entire `Scan`, even if only a few matching items are returned.)

Note

For code examples that use `Scan`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Managing Indexes

Indexes give you access to alternate query patterns, and can speed up queries. This section compares and contrasts index creation and usage in SQL and Amazon DynamoDB.

Whether you are using a relational database or DynamoDB, you should be judicious with index creation. Whenever a write occurs on a table, all of the table's indexes must be updated. In a write-heavy environment with large tables, this can consume large amounts of system resources. In a read-only or read-mostly environment, this is not as much of a concern. However, you should ensure that the indexes are actually being used by your application, and not simply taking up space.

Topics

- [Creating an Index \(p. 37\)](#)
- [Querying and Scanning an Index \(p. 39\)](#)

Creating an Index

Compare the `CREATE INDEX` statement in SQL with the `UpdateTable` operation in Amazon DynamoDB.

Topics

- [SQL \(p. 37\)](#)
- [DynamoDB \(p. 38\)](#)

SQL

In a relational database, an index is a data structure that lets you perform fast queries on different columns in a table. You can use the `CREATE INDEX` SQL statement to add an index to an existing table, specifying the columns to be indexed. After the index has been created, you can query the data in the table as usual, but now the database can use the index to quickly find the specified rows in the table instead of scanning the entire table.

After you create an index, the database maintains it for you. Whenever you modify data in the table, the index is automatically modified to reflect changes in the table.

In MySQL, you can create an index like the following.

```
CREATE INDEX GenreAndPriceIndex
ON Music (genre, price);
```

DynamoDB

In DynamoDB, you can create and use a *secondary index* for similar purposes.

Indexes in DynamoDB are different from their relational counterparts. When you create a secondary index, you must specify its key attributes—a partition key and a sort key. After you create the secondary index, you can *Query* it or *Scan* it just as you would with a table. DynamoDB does not have a query optimizer, so a secondary index is only used when you *Query* it or *Scan* it.

DynamoDB supports two different kinds of indexes:

- Global secondary indexes – The primary key of the index can be any two attributes from its table.
- Local secondary indexes – The partition key of the index must be the same as the partition key of its table. However, the sort key can be any other attribute.

DynamoDB ensures that the data in a secondary index is eventually consistent with its table. You can request strongly consistent *Query* or *Scan* actions on a table or a local secondary index. However, global secondary indexes support only eventual consistency.

You can add a global secondary index to an existing table, using the `UpdateTable` action and specifying `GlobalSecondaryIndexUpdates`.

```
{
    TableName: "Music",
    AttributeDefinitions:[
        {AttributeName: "Genre", AttributeType: "S"},
        {AttributeName: "Price", AttributeType: "N"}
    ],
    GlobalSecondaryIndexUpdates: [
        {
            Create: {
                IndexName: "GenreAndPriceIndex",
                KeySchema: [
                    {AttributeName: "Genre", KeyType: "HASH"}, //Partition key
                    {AttributeName: "Price", KeyType: "RANGE"} //Sort key
                ],
                Projection: {
                    "ProjectionType": "ALL"
                },
                ProvisionedThroughput: { // Only specified
                    if using provisioned mode
                        "ReadCapacityUnits": 1,"WriteCapacityUnits": 1
                }
            }
        }
    ]
}
```

You must provide the following parameters to `UpdateTable`:

- `TableName` – The table that the index will be associated with.
- `AttributeDefinitions` – The data types for the key schema attributes of the index.
- `GlobalSecondaryIndexUpdates` – Details about the index you want to create:
 - `IndexName` – A name for the index.
 - `KeySchema` – The attributes that are used for the index's primary key.

- **Projection** – Attributes from the table that are copied to the index. In this case, `ALL` means that all of the attributes are copied.
- **ProvisionedThroughput** (for provisioned tables) – The number of reads and writes per second that you need for this index. (This is separate from the provisioned throughput settings of the table.)

Part of this operation involves backfilling data from the table into the new index. During backfilling, the table remains available. However, the index is not ready until its `Backfilling` attribute changes from true to false. You can use the `DescribeTable` action to view this attribute.

Note

For code examples that use `UpdateTable`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Querying and Scanning an Index

Compare querying and scanning an index using the `SELECT` statement in SQL with the `Query` and `Scan` operations in Amazon DynamoDB.

Topics

- [SQL \(p. 39\)](#)
- [DynamoDB \(p. 39\)](#)

SQL

In a relational database, you do not work directly with indexes. Instead, you query tables by issuing `SELECT` statements, and the query optimizer can make use of any indexes.

A *query optimizer* is a relational database management system (RDBMS) component that evaluates the available indexes and determines whether they can be used to speed up a query. If the indexes can be used to speed up a query, the RDBMS accesses the index first and then uses it to locate the data in the table.

Here are some SQL statements that can use `GenreAndPriceIndex` to improve performance. We assume that the `Music` table has enough data in it that the query optimizer decides to use this index, rather than simply scanning the entire table.

```
/* All of the rock songs */  
  
SELECT * FROM Music  
WHERE Genre = 'Rock';
```

```
/* All of the cheap country songs */  
  
SELECT Artist, SongTitle, Price FROM Music  
WHERE Genre = 'Country' AND Price < 0.50;
```

DynamoDB

In DynamoDB, you perform `Query` operations directly on the index, in the same way that you would on a table. You must specify both `TableName` and `IndexName`.

The following are some queries on `GenreAndPriceIndex` in DynamoDB. (The key schema for this index consists of `Genre` and `Price`.)

```
// All of the rock songs
{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre",
    ExpressionAttributeValues: {
        ":genre": "Rock"
    },
}
```

```
// All of the cheap country songs
{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex",
    KeyConditionExpression: "Genre = :genre and Price < :price",
    ExpressionAttributeValues: {
        ":genre": "Country",
        ":price": 0.50
    },
    ProjectionExpression: "Artist, SongTitle, Price"
};
```

This example uses a `ProjectionExpression` to indicate that you only want some of the attributes, rather than all of them, to appear in the results.

You can also perform Scan operations on a secondary index, in the same way that you would on a table. The following is a scan on `GenreAndPriceIndex`.

```
// Return all of the data in the index
{
    TableName: "Music",
    IndexName: "GenreAndPriceIndex"
}
```

Modifying Data in a Table

The SQL language provides the UPDATE statement for modifying data. Amazon DynamoDB uses the `UpdateItem` operation to accomplish similar tasks.

Topics

- [SQL \(p. 40\)](#)
- [DynamoDB \(p. 41\)](#)

SQL

In SQL, you use the UPDATE statement to modify one or more rows. The SET clause specifies new values for one or more columns, and the WHERE clause determines which rows are modified. The following is an example.

```
UPDATE Music
SET RecordLabel = 'Global Records'
WHERE Artist = 'No One You Know' AND SongTitle = 'Call Me Today';
```

If no rows match the `WHERE` clause, the `UPDATE` statement has no effect.

DynamoDB

In DynamoDB, you use the `UpdateItem` action to modify a single item. (If you want to modify multiple items, you must use multiple `UpdateItem` operations.)

The following is an example.

```
{
  TableName: "Music",
  Key: {
    "Artist": "No One You Know",
    "SongTitle": "Call Me Today"
  },
  UpdateExpression: "SET RecordLabel = :label",
  ExpressionAttributeValues: {
    ":label": "Global Records"
  }
}
```

You must specify the `Key` attributes of the item to be modified and an `UpdateExpression` to specify attribute values. `UpdateItem` behaves like an "upsert" operation: The item is updated if it exists in the table, but if not, a new item is added (inserted).

`UpdateItem` supports *conditional writes*, where the operation succeeds only if a specific `ConditionExpression` evaluates to true. For example, the following `UpdateItem` action does not perform the update unless the price of the song is greater than or equal to 2.00.

```
{
  TableName: "Music",
  Key: {
    "Artist": "No One You Know",
    "SongTitle": "Call Me Today"
  },
  UpdateExpression: "SET RecordLabel = :label",
  ConditionExpression: "Price >= :p",
  ExpressionAttributeValues: {
    ":label": "Global Records",
    ":p": 2.00
  }
}
```

`UpdateItem` also supports *atomic counters*, or attributes of type `Number` that can be incremented or decremented. Atomic counters are similar in many ways to sequence generators, identity columns, or autoincrement fields in SQL databases.

The following is an example of an `UpdateItem` action to initialize a new attribute (`Plays`) to keep track of the number of times a song has been played.

```
{
  TableName: "Music",
  Key: {
    "Artist": "No One You Know",
    "SongTitle": "Call Me Today"
  },
  UpdateExpression: "SET Plays = :val",
  ExpressionAttributeValues: {
    ":val": 0
  },
}
```

```
    ReturnValues: "UPDATED_NEW"
}
```

The `ReturnValues` parameter is set to `UPDATED_NEW`, which returns the new values of any attributes that were updated. In this case, it returns 0 (zero).

Whenever someone plays this song, we can use the following `UpdateItem` action to increment `Plays` by one.

```
{
  TableName: "Music",
  Key: {
    "Artist": "No One You Know",
    "SongTitle": "Call Me Today"
  },
  UpdateExpression: "SET Plays = Plays + :incr",
  ExpressionAttributeValues: {
    ":incr": 1
  },
  ReturnValues: "UPDATED_NEW"
}
```

Note

For code examples that use `UpdateItem`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Deleting Data from a Table

In SQL, the `DELETE` statement removes one or more rows from a table. Amazon DynamoDB uses the `DeleteItem` operation to delete one item at a time.

Topics

- [SQL \(p. 42\)](#)
- [DynamoDB \(p. 42\)](#)

SQL

In SQL, you use the `DELETE` statement to delete one or more rows. The `WHERE` clause determines the rows that you want to modify. The following is an example.

```
DELETE FROM Music
WHERE Artist = 'The Acme Band' AND SongTitle = 'Look Out, World';
```

You can modify the `WHERE` clause to delete multiple rows. For example, you could delete all of the songs by a particular artist, as shown in the following example.

```
DELETE FROM Music WHERE Artist = 'The Acme Band'
```

Note

If you omit the `WHERE` clause, the database attempts to delete all of the rows from the table.

DynamoDB

In DynamoDB, you use the `DeleteItem` action to delete data from a table, one item at a time. You must specify the item's primary key values.

```
{  
    TableName: "Music",  
    Key: {  
        Artist: "The Acme Band",  
        SongTitle: "Look Out, World"  
    }  
}
```

Note

In addition to `DeleteItem`, Amazon DynamoDB supports a `BatchWriteItem` action for deleting multiple items at the same time.

`DeleteItem` supports *conditional writes*, where the operation succeeds only if a specific `ConditionExpression` evaluates to true. For example, the following `DeleteItem` action deletes the item only if it has a `RecordLabel` attribute.

```
{  
    TableName: "Music",  
    Key: {  
        Artist: "The Acme Band",  
        SongTitle: "Look Out, World"  
    },  
    ConditionExpression: "attribute_exists(RecordLabel)"  
}
```

Note

For code examples that use `DeleteItem`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Removing a Table

In SQL, you use the `DROP TABLE` statement to remove a table. In Amazon DynamoDB, you use the `DeleteTable` operation.

Topics

- [SQL \(p. 43\)](#)
- [DynamoDB \(p. 43\)](#)

SQL

When you no longer need a table and want to discard it permanently, you use the `DROP TABLE` statement in SQL.

```
DROP TABLE Music;
```

After a table is dropped, it cannot be recovered. (Some relational databases do allow you to undo a `DROP TABLE` operation, but this is vendor-specific functionality and it is not widely implemented.)

DynamoDB

DynamoDB has a similar action: `DeleteTable`. In the following example, the table is permanently deleted.

```
{  
    TableName: "Music"
```

}

Note

For code examples that use `DeleteTable`, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Additional Amazon DynamoDB Resources

Topics

- [Blog Posts, Repositories, and Guides \(p. 44\)](#)
- [Data Modeling and Design Patterns \(p. 44\)](#)
- [Advanced Design Patterns with Rick Houlihan \(p. 44\)](#)
- [Training Courses \(p. 44\)](#)
- [Tools for Coding and Visualization \(p. 45\)](#)

Blog Posts, Repositories, and Guides

- [How to switch from RDBMS to DynamoDB in 20 easy steps](#) — An irreverent list from [Jeremy Daly](#) of useful steps for learning data modeling.
- [DynamoDB JavaScript DocumentClient cheat sheet](#) — A cheat sheet to help you get started building applications with DynamoDB in a Node.js or JavaScript environment.
- [From relational database to single DynamoDB table: A step-by-step exploration](#) — An in-depth post from [Forrest Brazeal](#) that includes a step-by-step approach for moving to a DynamoDB-style, one-table design.

Data Modeling and Design Patterns

- [AWS re:Invent 2019: Data modeling with DynamoDB](#) — A talk by [Alex DeBrie](#) that gets you started on the principles of DynamoDB data modeling.

Advanced Design Patterns with Rick Houlihan

- [AWS re:Invent 2019: Advanced design patterns](#)
 - Jeremy Daly shares his [12 key takeaways](#) from this session.
- [AWS re:Invent 2018: Advanced design patterns](#)
- [AWS re:Invent 2017: Advanced design patterns](#)

Note

Each session covers different use cases and examples.

Training Courses

- [DynamoDB Deep-Dive Course](#) — A course from Linux Academy with help from the Amazon DynamoDB team.
- [Amazon DynamoDB: Building NoSQL Database-Driven Applications](#) — A course from the AWS Training and Certification team hosted on edX.

- [AWS DynamoDB — From Beginner to Pro](#) — A course from A Cloud Guru.

Tools for Coding and Visualization

- [NoSQL Workbench for Amazon DynamoDB](#) — A unified, visual tool that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables.
- [DynamoDB Toolbox](#) — A project from Jeremy Daly that provides helpful utilities for working with data modeling and in JavaScript and Node.js.
- [Dynobase](#) — A desktop tool that makes it easy to see and work with your DynamoDB tables, create app code, and edit records with real-time validation.
- [DynamoDB Streams Processor](#) — A simple tool that makes working with [DynamoDB Streams](#) super easy.

Setting Up DynamoDB

In addition to the Amazon DynamoDB web service, AWS provides a downloadable version of DynamoDB that you can run on your computer and is perfect for development and testing of your code. The downloadable version lets you write and test applications locally without accessing the DynamoDB web service.

The topics in this section describe how to set up DynamoDB (downloadable version) and the DynamoDB web service.

Topics

- [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#)
- [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#)

Setting Up DynamoDB Local (Downloadable Version)

With the downloadable version of Amazon DynamoDB, you can develop and test applications without accessing the DynamoDB web service. Instead, the database is self-contained on your computer. When you're ready to deploy your application in production, you remove the local endpoint in the code, and then it points to the DynamoDB web service.

Having this local version helps you save on throughput, data storage, and data transfer fees. In addition, you don't need an internet connection while you develop your application.

DynamoDB Local is available as a download (requires JRE), as an Apache Maven dependency, or as a Docker image.

If you prefer to use the Amazon DynamoDB web service instead, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).

Topics

- [Deploying DynamoDB Locally on Your Computer \(p. 46\)](#)
- [Deploying DynamoDB by adding an Apache Maven Repository \(p. 48\)](#)
- [Install the DynamoDB Docker Image \(p. 48\)](#)
- [DynamoDB Usage Notes \(p. 49\)](#)

Deploying DynamoDB Locally on Your Computer

The downloadable version of Amazon DynamoDB is provided as an executable .jar file. The application runs on Windows, Linux, macOS, and other platforms that support Java.

Follow these steps to set up and run DynamoDB on your computer.

To set up DynamoDB on your computer

1. Download DynamoDB for free using one of the following links.

| Region | Download Links | Checksums |
|----------------------------------|--|--|
| Asia Pacific (Mumbai) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |
| Asia Pacific (Singapore) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |
| Asia Pacific (Tokyo) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |
| Europe (Frankfurt) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |
| South America (São Paulo) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |
| US West (Oregon) Region | .tar.gz .zip | .tar.gz.sha256 .zip.sha256 |

Downloadable DynamoDB is available on Apache Maven. For more information, see [Deploying DynamoDB by adding an Apache Maven Repository \(p. 48\)](#). DynamoDB is also available as part of the AWS Toolkit for Eclipse. For more information, see [AWS Toolkit For Eclipse](#).

Important

To run DynamoDB on your computer, you must have the Java Runtime Environment (JRE) version 6.x or newer. The application doesn't run on earlier JRE versions.

2. After you download the archive, extract the contents and copy the extracted directory to a location of your choice.
3. To start DynamoDB on your computer, open a command prompt window, navigate to the directory where you extracted `DynamoDBLocal.jar`, and enter the following command.

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

Note

If you're using Windows PowerShell, be sure to enclose the parameter name or the entire name and value like this:

```
java -D"java.library.path=./DynamoDBLocal_lib" -jar DynamoDBLocal.jar
```

DynamoDB processes incoming requests until you stop it. To stop DynamoDB, press Ctrl+C at the command prompt.

DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. For a complete list of DynamoDB runtime options, including `-port`, enter this command.

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

4. Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must configure your credentials to enable authorization for your applications. Downloadable DynamoDB requires any credentials to work, as shown in the following example.

```
AWS Access Key ID: "fakeMyKeyId"
AWS Secret Access Key: "fakeSecretAccessKey"
```

You can use the `aws configure` command of the AWS CLI to set up credentials. For more information, see [Using the AWS CLI \(p. 56\)](#).

5. Start writing applications. To access DynamoDB running locally with the AWS CLI, use the `--endpoint-url` parameter. For example, use the following command to list DynamoDB tables.

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

Deploying DynamoDB by adding an Apache Maven Repository

Follow these steps to use Amazon DynamoDB in your application as a dependency.

To deploy DynamoDB as an Apache Maven repository

1. Download and install Apache Maven. For more information, see [Downloading Apache Maven](#) and [Installing Apache Maven](#).
2. Add the DynamoDB Maven repository to your application's Project Object Model (POM) file.

```
<!--Dependency:-->
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>DynamoDBLocal</artifactId>
        <version>[1.12, 2.0)</version>
    </dependency>
</dependencies>
<!--Custom repository:-->
<repositories>
    <repository>
        <id>dynamodb-local-oregon</id>
        <name>DynamoDB Local Release Repository</name>
        <url>https://s3-us-west-2.amazonaws.com/dynamodb-local/release</url>
    </repository>
</repositories>
```

Note

You can also use one of the following repository URLs, depending on your AWS Region.

| id | Repository URL |
|--------------------------|---|
| dynamodb-local-mumbai | https://s3.ap-south-1.amazonaws.com/dynamodb-local-mumbai/release |
| dynamodb-local-singapore | https://s3.ap-southeast-1.amazonaws.com/dynamodb-local-singapore/release |
| dynamodb-local-tokyo | https://s3.ap-northeast-1.amazonaws.com/dynamodb-local-tokyo/release |
| dynamodb-local-frankfurt | https://s3.eu-central-1.amazonaws.com/dynamodb-local-frankfurt/release |
| dynamodb-local-sao-paulo | https://s3.sa-east-1.amazonaws.com/dynamodb-local-sao-paulo/release |

The `aws-dynamodb-examples` repository in GitHub contains examples for [starting and stopping DynamoDB local](#) inside a Java program and [using DynamoDB local in JUnit tests](#).

Install the DynamoDB Docker Image

The downloadable version of Amazon DynamoDB is available as a Docker image. For more information, see [dynamodb-local](#).

For an example of using DynamoDB local as part of a REST application built on the AWS Serverless Application Model (AWS SAM), see [SAM DynamoDB application for managing orders](#). This sample application demonstrates how to use DynamoDB local for testing.

DynamoDB Usage Notes

Except for the endpoint, applications that run with the downloadable version of Amazon DynamoDB should also work with the DynamoDB web service. However, when using DynamoDB locally, you should be aware of the following:

- If you use the `-sharedDb` option, DynamoDB creates a single database file named *shared-local-instance.db*. Every program that connects to DynamoDB accesses this file. If you delete the file, you lose any data that you have stored in it.
- If you omit `-sharedDb`, the database file is named *myaccesskeyid_region.db*, with the AWS access key ID and AWS Region as they appear in your application configuration. If you delete the file, you lose any data that you have stored in it.
- If you use the `-inMemory` option, DynamoDB doesn't write any database files at all. Instead, all data is written to memory, and the data is not saved when you terminate DynamoDB.
- If you use the `-optimizeDbBeforeStartup` option, you must also specify the `-dbPath` parameter so that DynamoDB can find its database file.
- The AWS SDKs for DynamoDB require that your application configuration specify an access key value and an AWS Region value. Unless you're using the `-sharedDb` or the `-inMemory` option, DynamoDB uses these values to name the local database file. These values don't have to be valid AWS values to run locally. However, you might find it convenient to use valid values so that you can run your code in the cloud later by changing the endpoint you're using.

Topics

- [Command Line Options \(p. 49\)](#)
- [Setting the Local Endpoint \(p. 50\)](#)
- [Differences Between Downloadable DynamoDB and the DynamoDB Web Service \(p. 50\)](#)

Command Line Options

You can use the following command line options with the downloadable version of DynamoDB:

- `-cors value` — Enables support for cross-origin resource sharing (CORS) for JavaScript. You must provide a comma-separated "allow" list of specific domains. The default setting for `-cors` is an asterisk (*), which allows public access.
- `-dbPath value` — The directory where DynamoDB writes its database file. If you don't specify this option, the file is written to the current directory. You can't specify both `-dbPath` and `-inMemory` at once.
- `-delayTransientStatuses` — Causes DynamoDB to introduce delays for certain operations. DynamoDB (downloadable version) can perform some tasks almost instantaneously, such as create/update/delete operations on tables and indexes. However, the DynamoDB service requires more time for these tasks. Setting this parameter helps DynamoDB running on your computer simulate the behavior of the DynamoDB web service more closely. (Currently, this parameter introduces delays only for global secondary indexes that are in either *CREATING* or *DELETING* status.)
- `-help` — Prints a usage summary and options.
- `-inMemory` — DynamoDB runs in memory instead of using a database file. When you stop DynamoDB, none of the data is saved. You can't specify both `-dbPath` and `-inMemory` at once.
- `-optimizeDbBeforeStartup` — Optimizes the underlying database tables before starting DynamoDB on your computer. You also must specify `-dbPath` when you use this parameter.

- **-port value** — The port number that DynamoDB uses to communicate with your application. If you don't specify this option, the default port is 8000.

Note

DynamoDB uses port 8000 by default. If port 8000 is unavailable, this command throws an exception. You can use the **-port** option to specify a different port number. For a complete list of DynamoDB runtime options, including **-port**, type this command:

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

- **-sharedDb** — If you specify **-sharedDb**, DynamoDB uses a single database file instead of separate files for each credential and Region.

Setting the Local Endpoint

By default, the AWS SDKs and tools use endpoints for the Amazon DynamoDB web service. To use the SDKs and tools with the downloadable version of DynamoDB, you must specify the local endpoint:

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

AWS Command Line Interface

You can use the AWS Command Line Interface (AWS CLI) to interact with downloadable DynamoDB. For example, you can use it to perform all the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

To access DynamoDB running locally, use the **--endpoint-url** parameter. The following is an example of using the AWS CLI to list the tables in DynamoDB on your computer.

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

Note

The AWS CLI can't use the downloadable version of DynamoDB as a default endpoint. Therefore, you must specify **--endpoint-url** with each AWS CLI command.

AWS SDKs

The way you specify an endpoint depends on the programming language and AWS SDK you're using. The following sections describe how to do this:

- [Java: Setting the AWS Region and Endpoint \(p. 331\)](#)
- [.NET: Setting the AWS Region and Endpoint \(p. 333\)](#)

Note

For examples in other programming languages, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Differences Between Downloadable DynamoDB and the DynamoDB Web Service

The downloadable version of DynamoDB is intended for development and testing purposes only. By comparison, the DynamoDB web service is a managed service with scalability, availability, and durability features that make it ideal for production use.

The downloadable version of DynamoDB differs from the web service in the following ways:

- AWS Regions and distinct AWS accounts are not supported at the client level.
- Provisioned throughput settings are ignored in downloadable DynamoDB, even though the `CreateTable` operation requires them. For `CreateTable`, you can specify any numbers you want for provisioned read and write throughput, even though these numbers are not used. You can call `UpdateTable` as many times as you want per day. However, any changes to provisioned throughput values are ignored.
- Scan operations are performed sequentially. Parallel scans are not supported. The `Segment` and `TotalSegments` parameters of the `Scan` operation are ignored.
- The speed of read and write operations on table data is limited only by the speed of your computer. `CreateTable`, `UpdateTable`, and `DeleteTable` operations occur immediately, and table state is always ACTIVE. `UpdateTable` operations that change only the provisioned throughput settings on tables or global secondary indexes occur immediately. If an `UpdateTable` operation creates or deletes any global secondary indexes, then those indexes transition through normal states (such as CREATING and DELETING, respectively) before they become an ACTIVE state. The table remains ACTIVE during this time.
- Read operations are eventually consistent. However, due to the speed of DynamoDB running on your computer, most reads appear to be strongly consistent.
- Item collection metrics and item collection sizes are not tracked. In operation responses, nulls are returned instead of item collection metrics.
- In DynamoDB, there is a 1 MB limit on data returned per result set. Both the DynamoDB web service and the downloadable version enforce this limit. However, when querying an index, the DynamoDB service calculates only the size of the projected key and attributes. By contrast, the downloadable version of DynamoDB calculates the size of the entire item.
- If you're using DynamoDB Streams, the rate at which shards are created might differ. In the DynamoDB web service, shard-creation behavior is partially influenced by table partition activity. When you run DynamoDB locally, there is no table partitioning. In either case, shards are ephemeral, so your application should not be dependent on shard behavior.
- `TransactionConflictExceptions` are not thrown by downloadable DynamoDB for transactional APIs. We recommend that you use a Java mocking framework to simulate `TransactionConflictExceptions` in the DynamoDB handler to test how your application responds to conflicting transactions.
- In the DynamoDB web service, table names are case sensitive. A table named `Authors` and one named `authors` can both exist as separate tables. In the downloadable version, table names are case insensitive, and attempting to create these two tables would result in an error.

Setting Up DynamoDB (Web Service)

To use the Amazon DynamoDB web service:

1. [Sign up for AWS. \(p. 51\)](#)
2. [Get an AWS access key \(p. 52\)](#) (used to access DynamoDB programmatically).

Note

If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key, and you can skip ahead to [Using the Console \(p. 54\)](#).

3. [Configure your credentials \(p. 53\)](#) (used to access DynamoDB programmatically).

Signing Up for AWS

To use the DynamoDB service, you must have an AWS account. If you don't already have an account, you are prompted to create one when you sign up. You're not charged for any AWS services that you sign up for unless you use them.

To sign up for AWS

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Getting an AWS Access Key

Before you can access DynamoDB programmatically or through the AWS Command Line Interface (AWS CLI), you must have an AWS access key. You don't need an access key if you plan to use the DynamoDB console only.

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. As a best practice, do not use the AWS account root user access keys for any task where it's not required. Instead, [create a new administrator IAM user](#) with access keys for yourself.

The only time that you can view or download the secret access key is when you create the keys. You cannot recover them later. However, you can create new access keys at any time. You must also have permissions to perform the required IAM actions. For more information, see [Permissions Required to Access IAM Resources](#) in the *IAM User Guide*.

To create access keys for an IAM user

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, choose **Users**.
3. Choose the name of the user whose access keys you want to create, and then choose the **Security credentials** tab.
4. In the **Access keys** section, choose **Create access key**.
5. To view the new access key pair, choose **Show**. You will not have access to the secret access key again after this dialog box closes. Your credentials will look something like this:
 - Access key ID: AKIAIOSFODNN7EXAMPLE
 - Secret access key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
6. To download the key pair, choose **Download .csv file**. Store the keys in a secure location. You will not have access to the secret access key again after this dialog box closes.

Keep the keys confidential in order to protect your AWS account and never email them. Do not share them outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.

7. After you download the .csv file, choose **Close**. When you create an access key, the key pair is active by default, and you can use the pair right away.

Related topics

- [What Is IAM?](#) in the *IAM User Guide*
- [AWS Security Credentials](#) in *AWS General Reference*

Configuring Your Credentials

Before you can access DynamoDB programmatically or through the AWS CLI, you must configure your credentials to enable authorization for your applications.

There are several ways to do this. For example, you can manually create the credentials file to store your access key ID and secret access key. You also can use the `aws configure` command of the AWS CLI to automatically create the file. Alternatively, you can use environment variables. For more information about configuring your credentials, see the programming-specific AWS SDK developer guide.

To install and configure the AWS CLI, see [Using the AWS CLI \(p. 56\)](#).

Accessing DynamoDB

You can access Amazon DynamoDB using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API.

Topics

- [Using the Console \(p. 54\)](#)
- [Using the AWS CLI \(p. 56\)](#)
- [Using the API \(p. 58\)](#)
- [Using the NoSQL Workbench for DynamoDB \(p. 58\)](#)
- [IP Address Ranges \(p. 58\)](#)

Using the Console

You can access the AWS Management Console for Amazon DynamoDB at <https://console.aws.amazon.com/dynamodb/home>.

You can use the console to do the following in DynamoDB:

- Monitor recent alerts, total capacity, service health, and the latest DynamoDB news on the DynamoDB dashboard.
- Create, update, and delete tables. The capacity calculator provides estimates of how many capacity units to request based on the usage information you provide.
- Manage streams.
- View, add, update, and delete items that are stored in tables. Manage Time to Live (TTL) to define when items in a table expire so that they can be automatically deleted from the database.
- Query and scan a table.
- Set up and view alarms to monitor your table's capacity usage. View your table's top monitoring metrics on real-time graphs from CloudWatch.
- Modify a table's provisioned capacity.
- Create and delete global secondary indexes.
- Create triggers to connect DynamoDB streams to AWS Lambda functions.
- Apply tags to your resources to help organize and identify them.
- Purchase reserved capacity.

The console displays an introductory screen that prompts you to create your first table. To view your tables, in the navigation pane on the left side of the console, choose **Tables**.

Here's a high-level overview of the actions available per table within each navigation tab:

- **Overview** – View stream and table details, and manage streams and Time to Live (TTL).
- **Items** – Manage items and perform queries and scans.
- **Metrics** – Monitor Amazon CloudWatch metrics.
- **Alarms** – Manage CloudWatch alarms.
- **Capacity** – Modify a table's provisioned capacity.
- **Indexes** – Manage global secondary indexes.
- **Triggers** – Manage triggers to connect DynamoDB streams to Lambda functions.

- **Access control** – Set up fine-grained access control with web identity federation.
- **Tags** – Apply tags to your resources to help organize and identify them.

Working with User Preferences

You can configure some of the default settings in the Amazon DynamoDB console. For example, you can change the default query type that is used when you access the **Items** tab. If you're signed in to the console as an IAM user, you can store information about how you prefer to use the console. This information, also known as your *user preferences*, is stored and applied every time you use the console. Any time you access the DynamoDB console, these preferences are applied to all tables in all AWS Regions for your IAM user. They are not table-specific or Region-specific. They don't affect your interactions with the AWS CLI, DynamoDB API, or other services that interact with DynamoDB.

You can still change individual settings on console pages without having saved any user preferences. Those choices persist until you close the console window. When you return to the console, any saved user preferences are applied.

Note

User preferences are available only for IAM users. You can't set preferences if you use federated access, temporary access, or an AWS account root user to access the console.

User preferences include the following:

- **Table detail view mode:** View all the table-specific information vertically, horizontally, or covering the full screen (if enabled, the navigation bar still appears).
- **Show navigation bar:** Enable this option to show the navigation bar on the left side (expanded). Disable it to automatically collapse the navigation bar (you can expand it using the right chevron).
- **Default entry page (Dashboard or Tables):** Choose the page that loads when you access DynamoDB. This option automatically loads the Dashboard or the Tables page, respectively.
- **Items editor mode (Tree or Text):** Choose the default editor mode to use when you create or edit an item.
- **Items default query type (Scan or Query):** Choose the default query type to use when you access the **Items** tab. Choose **Scan** if you want to either enable or disable the automatic scan operation that occurs when accessing the **Items** tab.
- **Automatic scan operation when accessing the items tab:** If **Scan** is the default query type for items and you enable this setting, an automatic scan operation occurs when you access the **Items** tab. If you disable this setting, you can perform a scan by choosing **Start search** on the **Items** tab.

View and Save User Preferences

You can view and change your user preferences for the DynamoDB console. These settings apply only to your IAM user in the DynamoDB console. They don't affect other IAM users in your AWS account.

To view and save preferences on the DynamoDB console for your IAM user

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
Sign in as an IAM user. You can't configure user preferences for other user types.
2. In the title bar navigation, choose **Preferences**.
3. In **Preferences**, configure your preferences.

Do one of the following:

- To save and apply your changes, choose **Save**.

- To view the DynamoDB console default settings, choose **Restore**. These defaults are applied if you choose **Save**.

The default settings are as follows:

- **Table detail view mode:** Vertical
- **Show navigation bar:** Yes
- **Default entry page:** Dashboard
- **Items editor mode:** Tree
- **Items default query type:** Scan
- **Automatic scan operation when accessing the items tab:** Yes
- To return to the console page that you were on previously, choose **Back**. Or, choose **Dashboard** to go to the DynamoDB Dashboard page.

Using the AWS CLI

You can use the AWS Command Line Interface (AWS CLI) to control multiple AWS services from the command line and automate them through scripts. You can use the AWS CLI for ad hoc operations, such as creating a table. You can also use it to embed Amazon DynamoDB operations within utility scripts.

Before you can use the AWS CLI with DynamoDB, you must get an access key ID and secret access key. For more information, see [Getting an AWS Access Key \(p. 52\)](#).

For a complete listing of all the commands available for DynamoDB in the AWS CLI, see the [AWS CLI Command Reference](#).

Topics

- [Downloading and Configuring the AWS CLI \(p. 56\)](#)
- [Using the AWS CLI with DynamoDB \(p. 56\)](#)
- [Using the AWS CLI with Downloadable DynamoDB \(p. 57\)](#)

Downloading and Configuring the AWS CLI

The AWS CLI is available at <http://aws.amazon.com/cli>. It runs on Windows, macOS, or Linux. After you download the AWS CLI, follow these steps to install and configure it:

1. Go to the [AWS Command Line Interface User Guide](#).
2. Follow the instructions for [Installing the AWS CLI](#) and [Configuring the AWS CLI](#).

Using the AWS CLI with DynamoDB

The command line format consists of a DynamoDB operation name followed by the parameters for that operation. The AWS CLI supports a shorthand syntax for the parameter values, as well as JSON.

For example, the following command creates a table named *Music*. The partition key is *Artist*, and the sort key is *SongTitle*. (For easier readability, long commands in this section are broken into separate lines.)

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
```

```
AttributeName=Artist,AttributeType=S \
AttributeName=SongTitle,AttributeType=S \
--key-schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1
```

The following commands add new items to the table. These examples use a combination of shorthand syntax and JSON.

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

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

On the command line, it can be difficult to compose valid JSON. However, the AWS CLI can read JSON files. For example, consider the following JSON code snippet, which is stored in a file named *key-conditions.json*.

```
{
    "Artist": {
        "AttributeValueList": [
            {
                "S": "No One You Know"
            }
        ],
        "ComparisonOperator": "EQ"
    },
    "SongTitle": {
        "AttributeValueList": [
            {
                "S": "Call Me Today"
            }
        ],
        "ComparisonOperator": "EQ"
    }
}
```

You can now issue a Query request using the AWS CLI. In this example, the contents of the *key-conditions.json* file are used for the --key-conditions parameter.

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

Using the AWS CLI with Downloadable DynamoDB

The AWS CLI can also interact with DynamoDB (Downloadable Version) that runs on your computer. To enable this, add the following parameter to each command:

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

The following example uses the AWS CLI to list the tables in a local database.

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

If DynamoDB is using a port number other than the default (8000), modify the `--endpoint-url` value accordingly.

Note

The AWS CLI can't use the downloadable version of DynamoDB as a default endpoint. Therefore, you must specify `--endpoint-url` with each command.

Using the API

You can use the AWS Management Console and the AWS Command Line Interface to work interactively with Amazon DynamoDB. However, to get the most out of DynamoDB, you can write application code using the AWS SDKs.

The AWS SDKs provide broad support for DynamoDB in [Java](#), [JavaScript in the browser](#), [.NET](#), [Node.js](#), [PHP](#), [Python](#), [Ruby](#), [C++](#), [Go](#), [Android](#), and [iOS](#). To get started quickly with these languages, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Before you can use the AWS SDKs with DynamoDB, you must get an AWS access key ID and secret access key. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).

For a high-level overview of DynamoDB application programming with the AWS SDKs, see [Programming with DynamoDB and the AWS SDKs \(p. 210\)](#).

Using the NoSQL Workbench for DynamoDB

You can also access DynamoDB, by [downloading](#) and using the [NoSQL Workbench for Amazon DynamoDB \(p. 769\)](#).

IP Address Ranges

Amazon Web Services (AWS) publishes its current IP address ranges in JSON format. To view the current ranges, download [ip-ranges.json](#). For more information, see [AWS IP Address Ranges](#) in the AWS General Reference.

To find the IP address ranges that you can use to [access to DynamoDB tables and indexes](#), search the `ip-ranges.json` file for the following string: `"service": "DYNAMODB"`.

Note

The IP address ranges do not apply to DynamoDB Streams or DynamoDB Accelerator (DAX).

Getting Started with DynamoDB

Use the hands-on tutorials in this section to help you get started and learn more about Amazon DynamoDB.

Topics

- [Basic Concepts in DynamoDB \(p. 59\)](#)
- [Prerequisites - Getting Started Tutorial \(p. 59\)](#)
- [Step 1: Create a Table \(p. 59\)](#)
- [Step 2: Write Data to a Table Using the Console or AWS CLI \(p. 62\)](#)
- [Step 3: Read Data from a Table \(p. 64\)](#)
- [Step 4: Update Data in a Table \(p. 65\)](#)
- [Step 5: Query Data in a Table \(p. 67\)](#)
- [Step 6: Create a Global Secondary Index \(p. 69\)](#)
- [Step 7: Query the Global Secondary Index \(p. 72\)](#)
- [Step 8: \(Optional\) Clean Up Resources \(p. 74\)](#)
- [Getting Started with DynamoDB: Next Steps \(p. 74\)](#)

Basic Concepts in DynamoDB

Before you begin, you should familiarize yourself with the basic concepts in Amazon DynamoDB. For more information, see [DynamoDB Core Components](#).

Then continue on to [Prerequisites](#) to learn about setting up DynamoDB.

Prerequisites - Getting Started Tutorial

Before starting the Amazon DynamoDB tutorial, follow the steps in [Setting Up DynamoDB](#). Then continue on to [Step 1: Create a Table \(p. 59\)](#).

Note

- If you plan to interact with DynamoDB only through the AWS Management Console, you don't need an AWS access key. Complete the steps in [Signing Up for AWS](#), and then continue on to [Step 1: Create a Table \(p. 59\)](#).
- If you don't want to sign up for a free tier account, you can set up [DynamoDB Local \(Downloadable Version\)](#). Then continue on to [Step 1: Create a Table \(p. 59\)](#).

Step 1: Create a Table

In this step, you create a `Music` table in Amazon DynamoDB. The table has the following details:

- Partition key — `Artist`
- Sort key — `SongTitle`

For more information about table operations, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

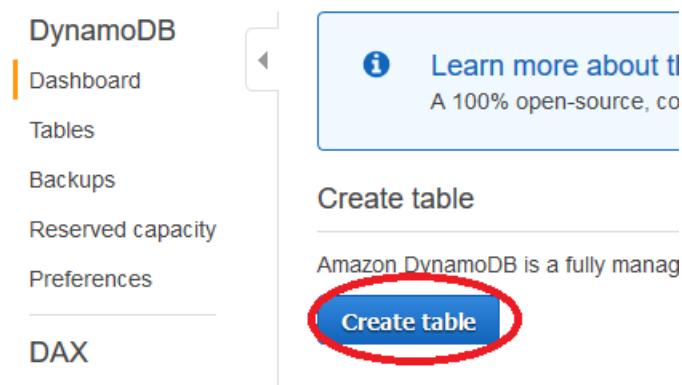
Note

Before you begin, make sure that you followed the steps in [Prerequisites - Getting Started Tutorial \(p. 59\)](#).

AWS Management Console

To create a new **Music** table using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Dashboard**.
3. On the right side of the console, choose **Create Table**.



4. Enter the table details as follows:
 - a. For the table name, enter **Music**.
 - b. For the partition key, enter **Artist**.
 - c. Choose **Add sort key**.
 - d. Enter **SongTitle** as the sort key.
5. Choose **Create** to create the table.

The screenshot shows the 'Create DynamoDB table' wizard. At the top, it says 'Create DynamoDB table' and has a 'Tutorial' link. Below that is a note about DynamoDB being schema-less. The main form has fields for 'Table name*' (set to 'Music'), 'Primary Key*' (set to 'Artist' under 'Partition key'), and 'SongTitle' (under 'Add sort key'). Under 'Table settings', there's a note about default settings, a checkbox for 'Use default settings', and a list of three items: 'No secondary indexes.', 'Auto Scaling capacity set to 70% target utilization, at minimum capacity of 5 reads and 5 writes.', and 'Encryption at Rest with DEFAULT encryption type.' At the bottom, there's a '+ Add tags [NEW]' button, a note about additional charges, and a 'Create' button.

AWS CLI

The following AWS CLI example creates a new Music table using `create-table`.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

Using `create-table` returns the following sample result.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-west-2:522194210714:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "TableStatus": "CREATING",
        "TableId": "d04c7240-0e46-435d-b231-d54091fe1017",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1558028402.69
    }
}
```

Note that the value of the `TableStatus` field is set to `CREATING`.

To verify that DynamoDB has finished creating the `Music` table, use the `describe-table` command.

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

This command returns the following result. When DynamoDB finishes creating the table, the value of the `TableStatus` field is set to `ACTIVE`.

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

After creating the new table, proceed to [Step 2: Write Data to a Table Using the Console or AWS CLI \(p. 62\)](#).

Step 2: Write Data to a Table Using the Console or AWS CLI

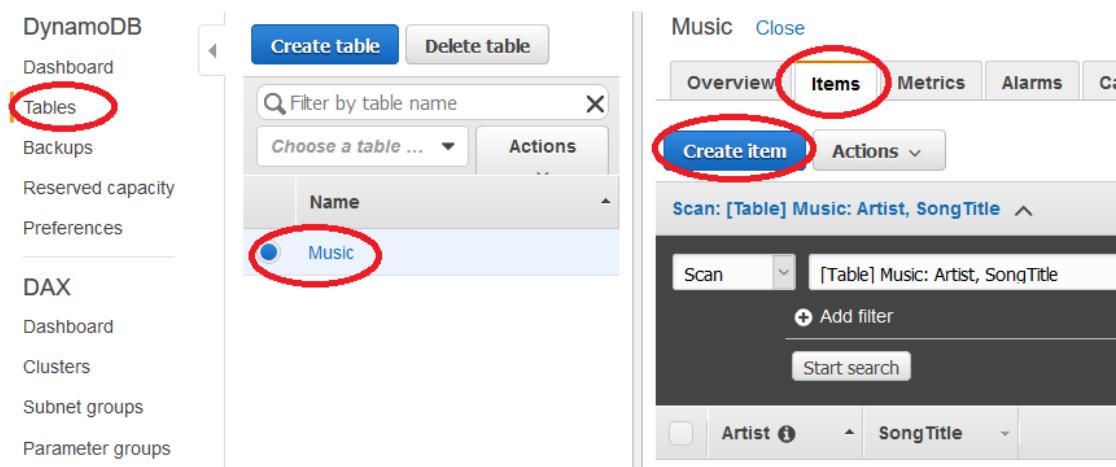
In this step, you insert two items into the `Music` table that you created in [Step 1: Create a Table \(p. 59\)](#).

For more information about write operations, see [Writing an Item \(p. 375\)](#).

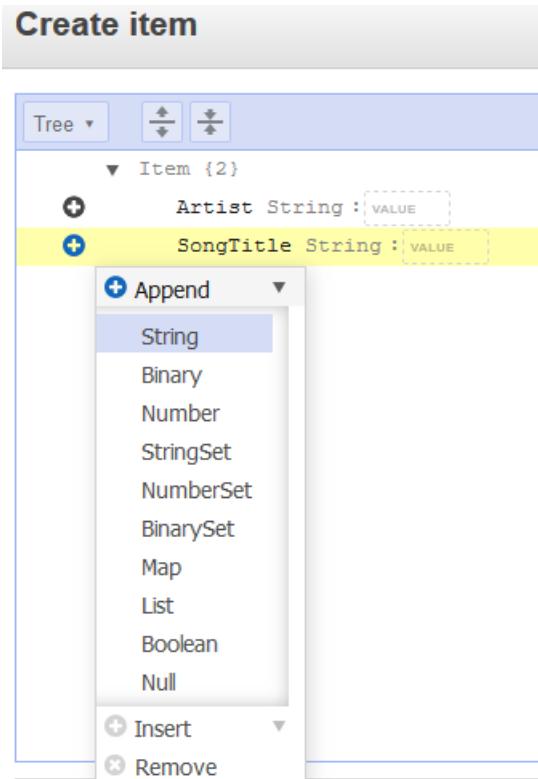
AWS Management Console

Follow these steps to write data to the `Music` table using the DynamoDB console.

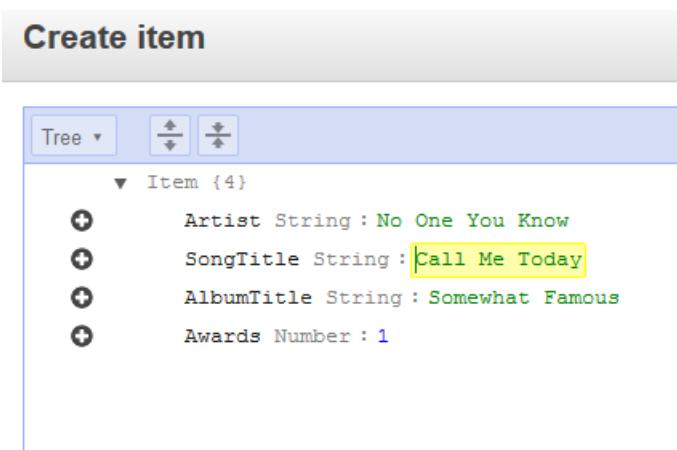
1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. In the table list, choose the **Music** table.
4. Choose the **Items** tab for the **Music** table.
5. On the **Items** tab, choose **Create item**.



6. Choose the plus sign (+) symbol next to **SongTitle**.
7. Choose **Append**, and then choose **Number**. Name the field **Awards**.



8. Repeat this process to create an **AlbumTitle** of type **String**.
9. Choose the following values for your item:
 - a. For **Artist**, enter **No One You Know** as the value.
 - b. For **SongTitle**, enter **Call Me Today**.
 - c. For **AlbumTitle**, enter **Somewhat Famous**.
 - d. For **Awards**, enter **1**.
10. Choose **Save**.



11. Repeat this process and create another item with the following values:
 - a. For **Artist**, enter **Acme Band**.

- b. For **SongTitle** enter **Happy Day**.
- c. For **AlbumTitle**, enter **Songs About Life**.
- d. For **Awards**, enter **10**.

AWS CLI

The following AWS CLI example creates two new items in the **Music** table using `put-item`.

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

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

For more information about supported data types in DynamoDB, see [Data Types](#).

For more information about how to represent DynamoDB data types in JSON, see [Attribute Values](#).

After writing data to your table, proceed to [Step 3: Read Data from a Table \(p. 64\)](#).

Step 3: Read Data from a Table

In this step, you will read back an item that was created in [Step 2: Write Data to a Table Using the Console or AWS CLI \(p. 62\)](#). You can use the DynamoDB console or the AWS CLI to read an item from the **Music** table by specifying **Artist** and **SongTitle**.

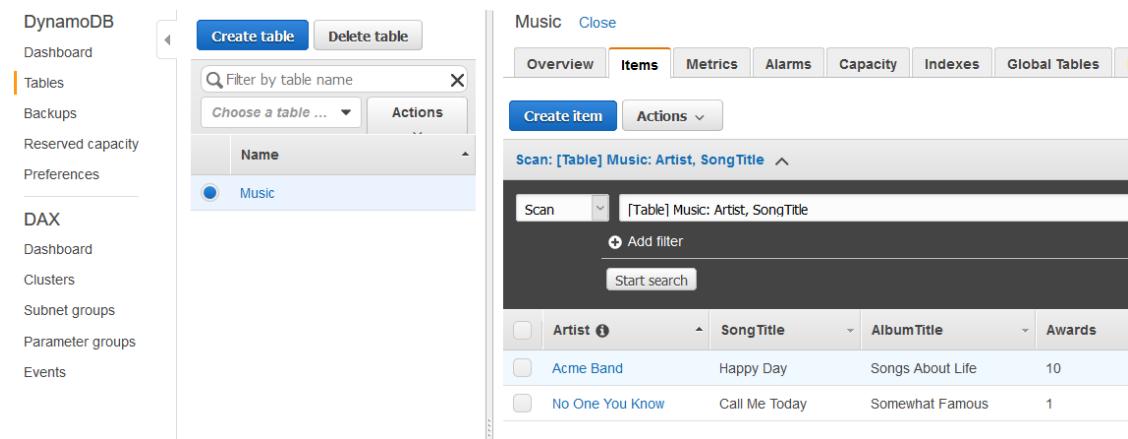
For more information about read operations in DynamoDB, see [Reading an Item \(p. 375\)](#).

AWS Management Console

Follow these steps to read data from the **Music** table using the DynamoDB console.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Items** tab for the Music table.
5. On the **Items** tab, view the list of items stored in the table, sorted by **Artist** and **SongTitle**.

The first item in the list is the one with the **Artist Acme Band** and the **SongTitle Happy Day**.



AWS CLI

The following AWS CLI example reads an item from the `Music` table using `get-item`.

Note

The default behavior for DynamoDB is eventually consistent reads. The `consistent-read` parameter is used below to demonstrate strongly consistent reads.

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

Using `get-item` returns the following sample result.

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

To update the data in your table, proceed to [Step 4: Update Data in a Table \(p. 65\)](#).

Step 4: Update Data in a Table

In this step, you update an item that you created in [Step 2: Write Data to a Table Using the Console or AWS CLI \(p. 62\)](#). You can use the DynamoDB console or the AWS CLI to update the `AlbumTitle` of an item in the `Music` table by specifying `Artist`, `SongTitle`, and the updated `AlbumTitle`.

For more information about write operations, see [Writing an Item \(p. 375\)](#).

AWS Management Console

You can use the DynamoDB console to update data in the **Music** table.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Items** tab for the Music table.
5. Choose the item whose **Artist** value is **Acme Band** and **SongTitle** value is **Happy Day**.
6. Update the **AlbumTitle** value to **Updated Album Title**, and then choose **Save**.

The following image shows the updated item on the console.

The screenshot shows the AWS Management Console interface for the Music table. On the left, the navigation pane shows the table list with 'Music' selected. The main area has tabs for Overview, Items, Metrics, Alarms, Capacity, Indexes, Global Tables, Backups, and Triggers, with 'Items' selected. Below the tabs, there's a search bar with 'Scan: [Table] Music: Artist, SongTitle'. A dropdown menu shows 'Scan' and '[Table] Music: Artist, SongTitle'. There's also an 'Add filter' button and a 'Start search' button. The results table shows two items: 'Acme Band' and 'No One You Know'. The 'Acme Band' row is currently selected. A modal window titled 'Edit item' is open over the table, showing a tree view of the item structure. The tree view shows 'Item (4)' expanded, with four attributes listed: 'AlbumTitle String : Updated Album Title', 'Artist String : Acme Band', 'Awards Number : 10', and 'SongTitle String : Happy Day'. The 'AlbumTitle' value is highlighted in green.

AWS CLI

The following AWS CLI example updates an item in the **Music** table using `update-item`.

```
aws dynamodb update-item \
--table-name Music \
--key '{ "Artist": { "S": "Acme Band"}, "SongTitle": { "S": "Happy Day"} }' \
--update-expression "SET AlbumTitle = :newval" \
--expression-attribute-values '{":newval":{ "S": "Updated Album Title"} }' \
--return-values ALL_NEW
```

Using `update-item` returns the following sample result.

```
{
```

```
"Attributes": {  
    "AlbumTitle": {  
        "S": "Updated Album Title"  
    },  
    "Awards": {  
        "N": "10"  
    },  
    "SongTitle": {  
        "S": "Happy Day"  
    },  
    "Artist": {  
        "S": "Acme Band"  
    }  
}
```

To query the data in the **Music** table, proceed to [Step 5: Query Data in a Table \(p. 67\)](#).

Step 5: Query Data in a Table

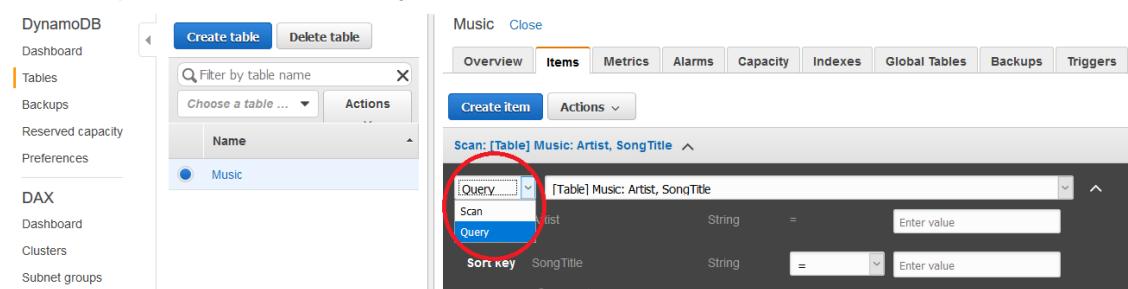
In this step, you query the data that you wrote to the **Music** table in [the section called "Step 2: Write Data" \(p. 62\)](#) by specifying **Artist**.

For more information about query operations, see [Working with Queries in DynamoDB \(p. 458\)](#).

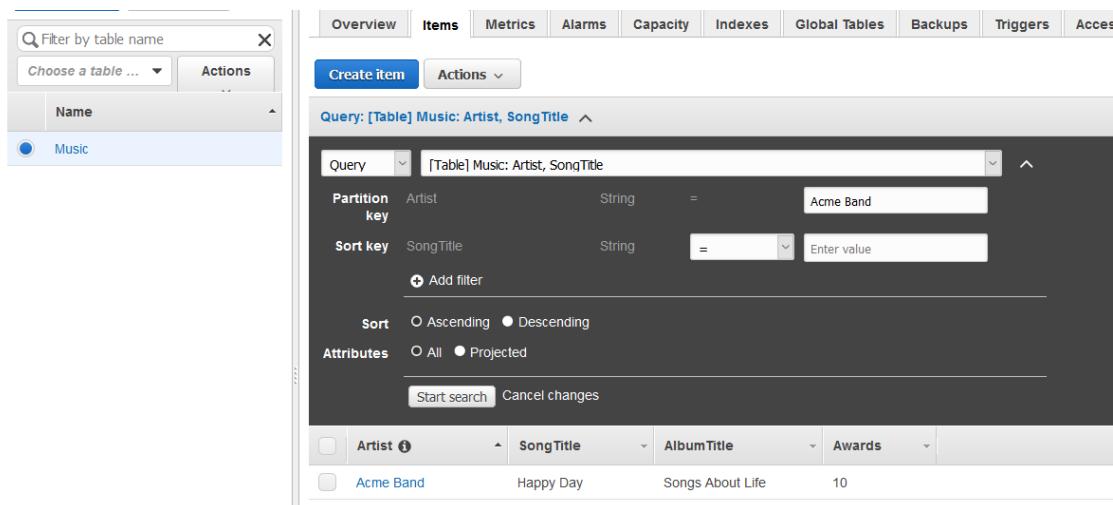
AWS Management Console

Follow these steps to use the DynamoDB console to query data in the **Music** table.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Items** tab for the **Music** table.
5. In the drop-down list, choose **Query**.



6. For **Partition key**, enter **Acme Band**, and then choose **Start search**.



AWS CLI

The following AWS CLI example queries an item in the `Music` table using `query`.

```
aws dynamodb query \
--table-name Music \
--key-condition-expression "Artist = :name" \
--expression-attribute-values '{":name":{"S":"Acme Band"}}'
```

Using `query` returns the following sample result.

```
{
    "Count": 1,
    "Items": [
        {
            "AlbumTitle": {
                "S": "Updated Album Title"
            },
            "Awards": {
                "N": "10"
            },
            "SongTitle": {
                "S": "Happy Day"
            },
            "Artist": {
                "S": "Acme Band"
            }
        }
    ],
    "ScannedCount": 1,
    "ConsumedCapacity": null
}
```

To create a global secondary index for your table, proceed to [Step 6: Create a Global Secondary Index \(p. 69\)](#).

Step 6: Create a Global Secondary Index

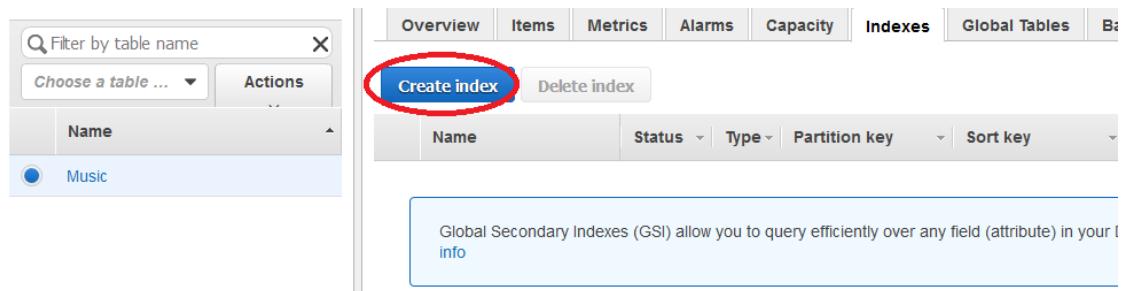
In this step, you create a global secondary index for the **Music** table that you created in [Step 1: Create a Table \(p. 59\)](#).

For more information about global secondary indexes, see [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#).

AWS Management Console

To use the Amazon DynamoDB console to create a global secondary index `AlbumTitle-index` for the **Music** table:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Indexes** tab for the Music table.
5. Choose **Create index**.



6. For the **Partition key**, enter `AlbumTitle`, and then choose **Create index**.

Create index

Primary key* Partition key

AlbumTitle String i

Add sort key

Index name* AlbumTitle-index i

Projected attributes All i

Read capacity units Write capacity units

10 5

Estimated cost \$3.39 / month ([Capacity calculator](#))

Approximate creation time is 5 minutes. Additional write capacity may decrease creation time. A notification will be sent to the SNS topic dynamodb once the index creation is complete. Basic Alarms with 80% upper threshold using SNS topic 'dynamodb' will be automatically created. Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced configuration for alarms can be done in the alarms tab.

[Cancel](#) [Create index](#)

AWS CLI

The following AWS CLI example creates a global secondary index `AlbumTitle-index` for the `Music` table using `update-table`.

```
aws dynamodb update-table \
--table-name Music \
--attribute-definitions AttributeName=AlbumTitle,AttributeType=S \
--global-secondary-index-updates \
"[{\\"Create\\":{\\\"IndexName\\\": \\"AlbumTitle-index\\\", \\\"KeySchema\\\": [{\\\"AttributeName\\\": \\\"AlbumTitle\\\", \\\"KeyType\\\": \\\"HASH\\\"}], \
\\\"ProvisionedThroughput\\\": {\\\"ReadCapacityUnits\\\": 10, \\\"WriteCapacityUnits\\\": 5 \
}, \\\"Projection\\\":{\\\"ProjectionType\\\": \\\"ALL\\\"}}}]"
```

Using `update-table` returns the following sample result.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-west-2:522194210714:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "AlbumTitle",
                "AttributeType": "S"
            },
        ]
    }
}
```

```
{
    "AttributeName": "Artist",
    "AttributeType": "S"
},
{
    "AttributeName": "SongTitle",
    "AttributeType": "S"
}
],
"GlobalSecondaryIndexes": [
    {
        "IndexSizeBytes": 0,
        "IndexName": "AlbumTitle-index",
        "Projection": {
            "ProjectionType": "ALL"
        },
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
        "IndexStatus": "CREATING",
        "Backfilling": false,
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "AlbumTitle"
            }
        ],
        "IndexArn": "arn:aws:dynamodb:us-west-2:522194210714:table/Music/index/AlbumTitle-index",
        "ItemCount": 0
    }
],
"ProvisionedThroughput": {
    "NumberOfDecreasesToday": 0,
    "WriteCapacityUnits": 5,
    "ReadCapacityUnits": 10
},
"TableSizeBytes": 0,
"TableName": "Music",
"TableStatus": "UPDATING",
"TableId": "d04c7240-0e46-435d-b231-d54091fe1017",
"KeySchema": [
    {
        "KeyType": "HASH",
        "AttributeName": "Artist"
    },
    {
        "KeyType": "RANGE",
        "AttributeName": "SongTitle"
    }
],
"ItemCount": 0,
"CreationDateTime": 1558028402.69
}
}
```

Note that the value of the `IndexStatus` field is set to `CREATING`.

To verify that DynamoDB has finished creating the `AlbumTitle-index` global secondary index, use the `describe-table` command.

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

This command returns the following result. The index is ready for use when the value of the `IndexStatus` field returned is set to `ACTIVE`.

```
"IndexStatus": "ACTIVE",
```

Next, you can query the global secondary index. For details, see [Step 7: Query the Global Secondary Index \(p. 72\)](#).

Step 7: Query the Global Secondary Index

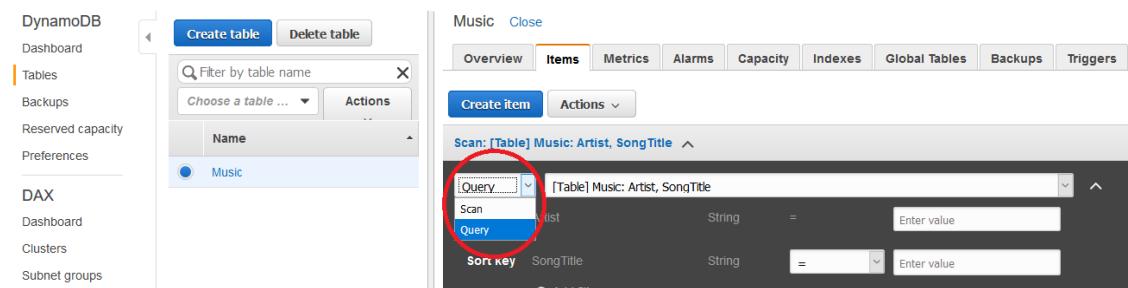
In this step, you query a global secondary index on the `Music` table using the Amazon DynamoDB console or the AWS CLI.

For more information about global secondary indexes, see [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#).

AWS Management Console

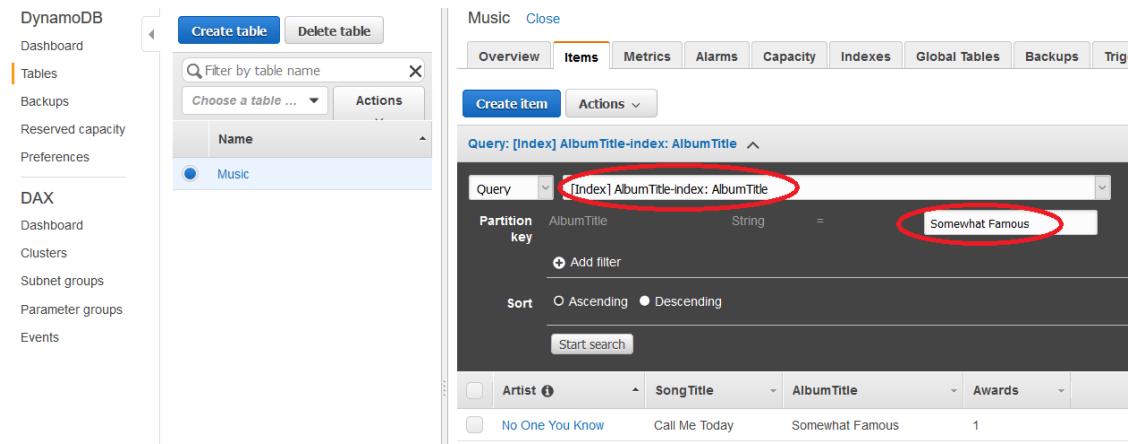
Follow these steps to use the DynamoDB console to query data through the `AlbumTitle-index` global secondary index.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Items** tab for the Music table.
5. In the drop-down list, choose **Query**.



6. In the drop-down list next to **Query**, choose **[Index] AlbumTitle-index: AlbumTitle**.

For **AlbumTitle**, enter **Somewhat Famous**, and then choose **Start search**.



AWS CLI

The following AWS CLI example queries a global secondary index `AlbumTitle-index` on the `Music` table.

```
aws dynamodb query \
--table-name Music \
--index-name AlbumTitle-index \
--key-condition-expression "AlbumTitle = :name" \
--expression-attribute-values '{":name":{"S":"Somewhat Famous"}}'
```

Using query returns the following sample result.

```
{
  "Count": 1,
  "Items": [
    {
      "AlbumTitle": {
        "S": "Somewhat Famous"
      },
      "Awards": {
        "N": "1"
      },
      "SongTitle": {
        "S": "Call Me Today"
      },
      "Artist": {
        "S": "No One You Know"
      }
    }
  ],
  "ScannedCount": 1,
  "ConsumedCapacity": null
}
```

Step 8: (Optional) Clean Up Resources

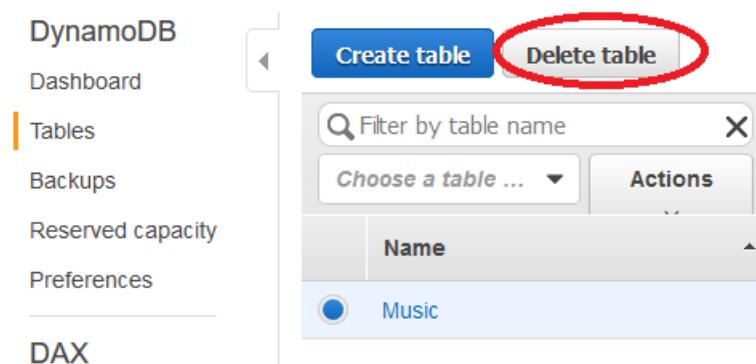
If you no longer need the Amazon DynamoDB table that you created for the tutorial, you can delete it. This step helps ensure that you aren't charged for resources that you aren't using. You can use the DynamoDB console or the AWS CLI to delete the **Music** table that you created in [Step 1: Create a Table \(p. 59\)](#).

For more information about table operations in DynamoDB, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

AWS Management Console

To delete the **Music** table using the console:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table from the table list.
4. Choose the **Indexes** tab for the **Music** table.
5. Choose **Delete table**.



AWS CLI

The following AWS CLI example deletes the **Music** table using `delete-table`.

```
aws dynamodb delete-table --table-name Music
```

Getting Started with DynamoDB: Next Steps

For more information about using Amazon DynamoDB, see the following topics:

- [Working with Tables and Data in DynamoDB \(p. 335\)](#)
- [Working with Items and Attributes \(p. 374\)](#)
- [Working with Queries in DynamoDB \(p. 458\)](#)
- [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#)
- [DynamoDB Transactions \(p. 648\)](#)
- [In-Memory Acceleration with DynamoDB Accelerator \(DAX\) \(p. 659\)](#)

- [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#)
- [Programming with DynamoDB and the AWS SDKs \(p. 210\)](#)

Getting Started with DynamoDB and AWS SDKs

This section contains hands-on tutorials to help you learn about Amazon DynamoDB. We encourage you to work through one of the language-specific tutorials. The code examples in these tutorials can run against either the downloadable version of DynamoDB or the DynamoDB web service.

Note

AWS SDKs are available for a wide variety of languages. For a complete list, see [Tools for Amazon Web Services](#).

Topics

- [Getting Started with Java and DynamoDB \(p. 76\)](#)
- [Getting Started with JavaScript and DynamoDB \(p. 95\)](#)
- [Getting Started with Node.js and DynamoDB \(p. 118\)](#)
- [Getting Started with .NET and DynamoDB \(p. 135\)](#)
- [PHP and DynamoDB \(p. 157\)](#)
- [Getting Started Developing with Python and DynamoDB \(p. 178\)](#)
- [Ruby and DynamoDB \(p. 191\)](#)

Getting Started with Java and DynamoDB

In this tutorial, you use the AWS SDK for Java to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

The SDK for Java offers several programming models for different use cases. In this exercise, the Java code uses the document model that provides a level of abstraction that makes it easier for you to work with JSON documents.

As you work through this tutorial, you can refer to the [AWS SDK for Java Documentation](#).

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

DynamoDB (downloadable version) is also available as part of the AWS Toolkit for Eclipse. For more information, see [AWS Toolkit For Eclipse](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 95\)](#).

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up the AWS SDK for Java:

- Install a Java development environment. If you are using the Eclipse IDE, install the AWS Toolkit for Eclipse.
- Install the AWS SDK for Java.
- Set up your AWS security credentials for use with the SDK for Java.

For instructions, see [Getting Started](#) in the *AWS SDK for Java Developer Guide*.

Step 1: Create a Table using Java and DynamoDB

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `ScalarAttributeType` is `N` for number.
- `title` – The sort key. The `ScalarAttributeType` is `S` for string.

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.Arrays;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;  
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;  
import com.amazonaws.services.dynamodbv2.model.KeyType;  
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;  
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;  
  
public class MoviesCreateTable {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        String tableName = "Movies";
```

```
try {
    System.out.println("Attempting to create table; please wait...");
    Table table = dynamoDB.createTable(tableName,
        Arrays.asList(new KeySchemaElement("year", KeyType.HASH), // Partition
                      // key
                      new KeySchemaElement("title", KeyType.RANGE)), // Sort key
        Arrays.asList(new AttributeDefinition("year", ScalarAttributeType.N),
                      new AttributeDefinition("title", ScalarAttributeType.S)),
        new ProvisionedThroughput(10L, 10L));
    table.waitForActive();
    System.out.println("Success. Table status: " +
    table.getDescription().getTableStatus());

}
catch (Exception e) {
    System.err.println("Unable to create table: ");
    System.err.println(e.getMessage());
}
}

}
```

Note

- You set the endpoint to indicate that you are creating the table in DynamoDB on your computer.
 - In the `createTable` call, you specify table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
2. Compile and run the program.

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data with Java and DynamoDB

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 79\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 79\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ...,  
    "title" : ....,  
    "info" : { ... }  
  },  
]
```

```
[ ... ]
```

In the JSON data, note the following:

- You use the `year` and `title` as the primary key attribute values for the `Movies` table.
- You store the rest of the `info` values in a single attribute called `info`. This program illustrates how you can store JSON in an Amazon DynamoDB attribute.

The following is an example of movie data.

```
{  
    "year" : 2013,  
    "title" : "Turn It Down, Or Else!",  
    "info" : {  
        "directors" : [  
            "Alice Smith",  
            "Bob Jones"  
        ],  
        "release_date" : "2013-01-18T00:00:00Z",  
        "rating" : 6.2,  
        "genres" : [  
            "Comedy",  
            "Drama"  
        ],  
        "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
        "rank" : 11,  
        "running_time_secs" : 5215,  
        "actors" : [  
            "David Matthewman",  
            "Ann Thomas",  
            "Jonathan G. Neff"  
        ]  
    }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").
```

```
* You may not use this file except in compliance with the License. A copy of
* the License is located at
*/
* http://aws.amazon.com/apache2.0/
*
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
* CONDITIONS OF ANY KIND, either express or implied. See the License for the
* specific language governing permissions and limitations under the License.
*/



package com.amazonaws.codesamples.gsg;

import java.io.File;
import java.util.Iterator;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class MoviesLoadData {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        JsonParser parser = new JsonFactory().createParser(new File("moviedata.json"));

        JsonNode rootNode = new ObjectMapper().readTree(parser);
        Iterator<JsonNode> iter = rootNode.iterator();

        ObjectNode currentNode;

        while (iter.hasNext()) {
            currentNode = (ObjectNode) iter.next();

            int year = currentNode.path("year").asInt();
            String title = currentNode.path("title").asText();

            try {
                table.putItem(new Item().withPrimaryKey("year", year, "title",
                    title).withJSON("info",
                        currentNode.path("info").toString()));
                System.out.println("PutItem succeeded: " + year + " " + title);
            }

            catch (Exception e) {
                System.err.println("Unable to add movie: " + year + " " + title);
                System.err.println(e.getMessage());
                break;
            }
        }
    }
}
```

```
        }
    }
}
```

This program uses the open source Jackson library to process JSON. Jackson is included in the AWS SDK for Java. You don't have to install it separately.

2. Compile and run the program.

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 81\)](#)
- [Step 3.2: Read an Item \(p. 82\)](#)
- [Step 3.3: Update an Item \(p. 84\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 85\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 87\)](#)
- [Step 3.6: Delete an Item \(p. 88\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy the following program and paste it into your Java development environment.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.gsg;

import java.util.HashMap;
import java.util.Map;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.PutItemOutcome;
```

```

import com.amazonaws.services.dynamodbv2.document.Table;

public class MoviesItemOps01 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        final Map<String, Object> infoMap = new HashMap<String, Object>();
        infoMap.put("plot", "Nothing happens at all.");
        infoMap.put("rating", 0);

        try {
            System.out.println("Adding a new item...");
            PutItemOutcome outcome = table
                .putItem(new Item().withPrimaryKey("year", year, "title",
                    title).withMap("info", infoMap));

            System.out.println("PutItem succeeded:\n" + outcome.getPutItemResult());

        }
        catch (Exception e) {
            System.err.println("Unable to add item: " + year + " " + title);
            System.err.println(e.getMessage());
        }

    }
}

```

Note

The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores JSON example code that provides more information about the movie.

2. Compile and run the program.

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}
```

You can use the `getItem` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Item;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;  
  
public class MoviesItemOps02 {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        int year = 2015;  
        String title = "The Big New Movie";  
  
        GetItemSpec spec = new GetItemSpec().withPrimaryKey("year", year, "title",  
            title);  
  
        try {  
            System.out.println("Attempting to read the item...");  
            Item outcome = table.getItem(spec);  
            System.out.println("GetItem succeeded: " + outcome);  
        }  
        catch (Exception e) {  
            System.err.println("Unable to read item: " + year + " " + title);  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

2. Compile and run the program.

Step 3.3: Update an Item

You can use the `updateItem` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from the following:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

To this:

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Everything happens all at once.",  
        rating: 5.5,  
        actors: ["Larry", "Moe", "Curly"]  
    }  
}
```

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.Arrays;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
```

```

import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps03 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("year",
year, "title", title)
            .withUpdateExpression("set info.rating = :r, info.plot=:p, info.actors=:a")
            .WithValueMap(new ValueMap().withNumber(":r", 5.5).withString(":p",
"Everything happens all at once."))
            .withList(":a", Arrays.asList("Larry", "Moe", "Curly")))
            .withReturnValues(ReturnValue.UPDATED_NEW);

        try {
            System.out.println("Updating the item...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
outcome.getItem().toJSONPretty());
        }
        catch (Exception e) {
            System.err.println("Unable to update item: " + year + " " + title);
            System.err.println(e.getMessage());
        }
    }
}

```

Note

This program uses an `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs Amazon DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. Compile and run the program.

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters. Use the `updateItem` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy the following program and paste it into your Java development environment.

```
/**
```

```

* Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
*
* This file is licensed under the Apache License, Version 2.0 (the "License").
* You may not use this file except in compliance with the License. A copy of
* the License is located at
*
* http://aws.amazon.com/apache2.0/
*
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
* CONDITIONS OF ANY KIND, either express or implied. See the License for the
* specific language governing permissions and limitations under the License.
*/



package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps04 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("year",
year, "title", title)
            .withUpdateExpression("set info.rating = info.rating + :val")
            .WithValueMap(new ValueMap().withNumber(":val",
1)).withReturnValues(ReturnValue.UPDATED_NEW);

        try {
            System.out.println("Incrementing an atomic counter...");
            UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
            System.out.println("UpdateItem succeeded:\n" +
outcome.getItem().toJSONPretty());
        }
        catch (Exception e) {
            System.err.println("Unable to update item: " + year + " " + title);
            System.err.println(e.getMessage());
        }
    }
}

```

2. Compile and run the program.

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the movie item is updated only if there are more than three actors.

1. Copy the following program and paste it into your Java development environment.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class MoviesItemOps05 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey(new PrimaryKey("year", year, "title",
                title)).withUpdateExpression("remove info.actors[0]")
            .withConditionExpression("size(info.actors) > :num").withValueMap(new
                ValueMap().withNumber(":num", 3))
            .withReturnValues(ReturnValue.UPDATED_NEW);

        // Conditional update (we expect this to fail)
        try {
            System.out.println("Attempting a conditional update...");

```

```
        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
        System.out.println("UpdateItem succeeded:\n" +
outcome.getItem().toJSONPretty());

    }
    catch (Exception e) {
        System.err.println("Unable to update item: " + year + " " + title);
        System.err.println(e.getMessage());
    }
}
```

- ## 2. Compile and run the program.

The program should fail with the following message:

The conditional request failed

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like the following.

```
.withConditionExpression("size(info.actors) >= :num")
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

Step 3.6: Delete an Item

You can use the `deleteItem` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */
```

```
package com.amazonaws.codesamples.gsg;

import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.PrimaryKey;
```

```

import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesItemOps06 {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        int year = 2015;
        String title = "The Big New Movie";

        DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
            .withPrimaryKey(new PrimaryKey("year", year, "title",
        title)).withConditionExpression("info.rating <= :val")
            .WithValueMap(new ValueMap().withNumber(":val", 5.0));

        // Conditional delete (we expect this to fail)

        try {
            System.out.println("Attempting a conditional delete...");
            table.deleteItem(deleteItemSpec);
            System.out.println("DeleteItem succeeded");
        }
        catch (Exception e) {
            System.err.println("Unable to delete item: " + year + " " + title);
            System.err.println(e.getMessage());
        }
    }
}

```

2. Compile and run the program.

The program should fail with the following message:

The conditional request failed

This is because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition in DeleteItemSpec.

```

DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
    .withPrimaryKey(new PrimaryKey("year", 2015, "title", "The Big New Movie"));

```

4. Compile and run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.

- **title** – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the **year**. You can also provide the **title** to retrieve a subset of movies based on some condition (on the sort key). For example, you can find movies released in 2014 that have a title starting with the letter "A".

In addition to the **query** method, there is also a **scan** method that can retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query \(p. 90\)](#)
- [Step 4.2: Scan \(p. 92\)](#)

Step 4.1: Query

The code included in this step performs the following queries:

- Retrieve all movies released in the **year 1985**.
- Retrieve all movies released in the **year 1992**, with a **title** beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.HashMap;  
import java.util.Iterator;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Item;  
import com.amazonaws.services.dynamodbv2.document.ItemCollection;  
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;  
  
public class MoviesQuery {  
  
    public static void main(String[] args) throws Exception {
```

```

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        HashMap<String, String> nameMap = new HashMap<String, String>();
        nameMap.put("#yr", "year");

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":yyyy", 1985);

        QuerySpec querySpec = new QuerySpec().withKeyConditionExpression("#yr
= :yyyy").withNameMap(nameMap)
            .withValueMap(valueMap);

        ItemCollection<QueryOutcome> items = null;
        Iterator<Item> iterator = null;
        Item item = null;

        try {
            System.out.println("Movies from 1985");
            items = table.query(querySpec);

            iterator = items.iterator();
            while (iterator.hasNext()) {
                item = iterator.next();
                System.out.println(item.getNumber("year") + ": " +
item.getString("title"));
            }
        }
        catch (Exception e) {
            System.err.println("Unable to query movies from 1985");
            System.err.println(e.getMessage());
        }

        valueMap.put(":yyyy", 1992);
        valueMap.put(":letter1", "A");
        valueMap.put(":letter2", "L");

        querySpec.withProjectionExpression("#yr, title, info.genres, info.actors[0]")
            .withKeyConditionExpression("#yr = :yyyy and title between :letter1
and :letter2").withNameMap(nameMap)
            .withValueMap(valueMap);

        try {
            System.out.println("Movies from 1992 - titles A-L, with genres and lead
actor");
            items = table.query(querySpec);

            iterator = items.iterator();
            while (iterator.hasNext()) {
                item = iterator.next();
                System.out.println(item.getNumber("year") + ": " +
item.getString("title") + " " + item.getMap("info"));
            }
        }
        catch (Exception e) {
            System.err.println("Unable to query movies from 1992:");
            System.err.println(e.getMessage());
        }
    }
}

```

```
        }  
    }  
}
```

Note

- **nameMap** provides name substitution. This is used because `year` is a reserved word in Amazon DynamoDB. You can't use it directly in any expression, including `KeyConditionExpression`. You use the expression attribute name `#yr` to address this.
- **valueMap** provides value substitution. This is used because you can't use literals in any expression, including `KeyConditionExpression`. You use the expression attribute value `:yyyy` to address this.

First, you create the `querySpec` object, which describes the query parameters, and then you pass the object to the `query` method.

2. Compile and run the program.

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Scan

The `scan` method reads every item in the entire table and returns all the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The `scan` specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items) and discard all the others.

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import java.util.Iterator;  
  
import com.amazonaws.client.builder.AwsClientBuilder;
```

```

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.ScanSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class MoviesScan {

    public static void main(String[] args) throws Exception {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withEndpointConfiguration(new
        AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
            .build();

        DynamoDB dynamoDB = new DynamoDB(client);

        Table table = dynamoDB.getTable("Movies");

        ScanSpec scanSpec = new ScanSpec().withProjectionExpression("#yr, title,
info.rating")
            .withFilterExpression("#yr between :start_yr and :end_yr").withNameMap(new
        NameMap().with("#yr", "year"))
            .withValueMap(new ValueMap().withNumber(":start_yr",
1950).withNumber(":end_yr", 1959));

        try {
            ItemCollection<ScanOutcome> items = table.scan(scanSpec);

            Iterator<Item> iter = items.iterator();
            while (iter.hasNext()) {
                Item item = iter.next();
                System.out.println(item.toString());
            }

        }
        catch (Exception e) {
            System.err.println("Unable to scan the table:");
            System.err.println(e.getMessage());
        }
    }
}

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.

2. Compile and run the program.

Note

You can also use the `Scan` operation with any secondary indexes that you created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program and paste it into your Java development environment.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.gsg;  
  
import com.amazonaws.client.builder.AwsClientBuilder;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Table;  
  
public class MoviesDeleteTable {  
  
    public static void main(String[] args) throws Exception {  
  
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()  
            .withEndpointConfiguration(new  
                AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))  
            .build();  
  
        DynamoDB dynamoDB = new DynamoDB(client);  
  
        Table table = dynamoDB.getTable("Movies");  
  
        try {  
            System.out.println("Attempting to delete table; please wait...");  
            table.delete();  
            table.waitForDelete();  
            System.out.print("Success.");  
  
        }  
        catch (Exception e) {  
            System.err.println("Unable to delete table: ");  
            System.err.println(e.getMessage());  
        }  
    }  
}
```

2. Compile and run the program.

Summary

In this tutorial, you created the `Movies` table in Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application.

1. Remove the following import.

```
import com.amazonaws.client.builder.AwsClientBuilder;
```

2. Next, go to `AmazonDynamoDB` in the code.

```
AmazonDynamoDB client =
    AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(
        new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2"))
    .build();
```

3. Now modify the client so that it accesses an AWS Region instead of a specific endpoint.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.REGION)
    .build();
```

For example, if you want to access the `us-west-2` region, you would do the following.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.US_WEST_2)
    .build();
```

Instead of using DynamoDB on your computer, the program now uses the DynamoDB web service endpoint in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting Regions and endpoints in your code, see [AWS Region Selection](#) in the [AWS SDK for Java Developer Guide](#).

Getting Started with JavaScript and DynamoDB

In this tutorial, you use JavaScript to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

As you work through this tutorial, you can refer to the [AWS SDK for JavaScript API Reference](#).

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary and Review of JavaScript and DynamoDB Tutorial \(p. 117\)](#).

- Set up an AWS access key to use AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up the AWS SDK for JavaScript. To do this, add or modify the following script tag to your HTML pages:

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>
```

Note

The version of AWS SDK for JavaScript might have been updated. For the latest version, see the [AWS SDK for JavaScript API Reference](#).

- Enable [cross-origin resource sharing \(CORS\)](#) so that your computer's browser can communicate with the downloadable version of DynamoDB.

To enable CORS

1. Download the free ModHeader Chrome browser extension (or any other browser extension that allows you to modify HTTP response headers).
2. Run the ModHeader Chrome browser extension, and add an HTTP response header with the name set to "Access-Control-Allow-Origin" and a value of "null" or "*".

Important

This configuration is required only while running this tutorial program for JavaScript on your computer. After you finish the tutorial, you should disable or remove this configuration.

3. You can now run the JavaScript tutorial program files.

If you prefer to run a complete version of the JavaScript tutorial program instead of performing step-by-step instructions, do the following:

1. Download the following file: [MoviesJavaScript.zip](#).
2. Extract the file `MoviesJavaScript.html` from the archive.
3. Modify the `MoviesJavaScript.html` file to use your endpoint.
4. Run the `MoviesJavaScript.html` file.

Step 1: Create a DynamoDB Table with JavaScript

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy the following program and paste it into a file named `MoviesCreateTable.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var dynamodb = new AWS.DynamoDB();

function createMovies() {
    var params = {
        TableName : "Movies",
        KeySchema: [
            { AttributeName: "year", KeyType: "HASH" },
            { AttributeName: "title", KeyType: "RANGE" }
        ],
        AttributeDefinitions: [
            { AttributeName: "year", AttributeType: "N" },
            { AttributeName: "title", AttributeType: "S" }
        ],
        ProvisionedThroughput: {
            ReadCapacityUnits: 5,
            WriteCapacityUnits: 5
        }
    };
    dynamodb.createTable(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to create table: " +
+ "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "Created table: " + "\n" +
JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>
```

```
<body>
<input id="createTableButton" type="button" value="Create Table"
    onclick="createMovies();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>

</body>
</html>
```

Note

- You set the endpoint to indicate that you are creating the table in Amazon DynamoDB on your computer.
 - In the `createMovies` function, you specify the table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this tutorial.)
2. Open the `MoviesCreateTable.html` file in your browser.
 3. Choose **Create Table**.

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data with JavaScript into DynamoDB

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 99\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 99\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ... ,  
    "title" : ....,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.

- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in an Amazon DynamoDB attribute.

The following is an example of movie data.

```
{  
    "year" : 2013,  
    "title" : "Turn It Down, Or Else!",  
    "info" : {  
        "directors" : [  
            "Alice Smith",  
            "Bob Jones"  
        ],  
        "release_date" : "2013-01-18T00:00:00Z",  
        "rating" : 6.2,  
        "genres" : [  
            "Comedy",  
            "Drama"  
        ],  
        "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
        "rank" : 11,  
        "running_time_secs" : 5215,  
        "actors" : [  
            "David Matthewman",  
            "Ann Thomas",  
            "Jonathan G. Neff"  
        ]  
    }  
}
```

Step 2.1: Download the Sample Data File

- Download the sample data archive: [moviedata.zip](#)
- Extract the data file (`moviedata.json`) from the archive.
- Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

- Copy the following program and paste it into a file named `MoviesLoadData.html`.

```
<!--  
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
  
This file is licensed under the Apache License, Version 2.0 (the "License").  
You may not use this file except in compliance with the License. A copy of  
the License is located at  
  
http://aws.amazon.com/apache2.0/  
  
This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
CONDITIONS OF ANY KIND, either express or implied. See the License for the  
specific language governing permissions and limitations under the License.  
-->  
<html>
```

```

<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script type="text/javascript">
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function processFile(evt) {
    document.getElementById('textarea').innerHTML = "";
    document.getElementById('textarea').innerHTML += "Importing movies into DynamoDB.
Please wait..." + "\n";
    var file = evt.target.files[0];
    if (file) {
        var r = new FileReader();
        r.onload = function(e) {
            var contents = e.target.result;
            var allMovies = JSON.parse(contents);

            allMovies.forEach(function (movie) {
                document.getElementById('textarea').innerHTML += "Processing: " +
movie.title + "\n";
                var params = {
                    TableName: "Movies",
                    Item: {
                        "year": movie.year,
                        "title": movie.title,
                        "info": movie.info
                    }
                };
                docClient.put(params, function (err, data) {
                    if (err) {
                        document.getElementById('textarea').innerHTML += "Unable to add
movie: " + count + movie.title + "\n";
                        document.getElementById('textarea').innerHTML += "Error JSON: " +
JSON.stringify(err) + "\n";
                    } else {
                        document.getElementById('textarea').innerHTML += "PutItem
succeeded: " + movie.title + "\n";
                        textarea.scrollTop = textarea.scrollHeight;
                    }
                });
            });
            r.readAsText(file);
        } else {
            alert("Could not read movie data file");
        }
    }
}

</script>
</head>

<body>

```

```
<input type="file" id="fileinput" accept='application/json' />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

<script>
    document.getElementById('fileinput').addEventListener('change', processFile,
    false);
</script>
</body>
</html>
```

2. Open the `MoviesLoadData.html` file in your browser.
3. Choose **Browse**, and load the `moviedata.json` file.

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 101\)](#)
- [Step 3.2: Read an Item \(p. 102\)](#)
- [Step 3.3: Update an Item \(p. 104\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 106\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 107\)](#)
- [Step 3.6: Delete an Item \(p. 109\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesItemOps01.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
```

```

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
accessKeyId: "fakeMyKeyId",
// secretAccessKey default can be used while using the downloadable version of
DynamoDB.
// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function createItem() {
    var params = {
        TableName :"Movies",
        Item:{ 
            "year": 2015,
            "title": "The Big New Movie",
            "info":{ 
                "plot": "Nothing happens at all.",
                "rating": 0
            }
        }
    };
    docClient.put(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to add item: " +
"\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "PutItem succeeded: " +
"\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="createItem" type="button" value="Create Item" onclick="createItem();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

Note

The primary key is required. This code adds an item that has a primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

2. Open the `MoviesItemOps01.html` file in your browser.
3. Choose **Create Item**.

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
```

```
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the get method to read the item from the `Movies` table. You must specify the primary key values, so you can read any item from `Movies` if you know its year and title.

1. Copy the following program and paste it into a file named `MoviesItemOps02.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function readItem() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    var params = {
        TableName: table,
        Key:{ 
            "year": year,
            "title": title
        }
    };
    docClient.get(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to read item: " +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "GetItem succeeded: " +
            "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}
```

```

        });
    }

</script>
</head>

<body>
<input id="readItem" type="button" value="Read Item" onclick="readItem();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesItemOps02.html` file in your browser.
3. Choose **Read Item**.

Step 3.3: Update an Item

You can use the `update` method to modify an item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The item changes from the following:

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}
```

To this:

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Everything happens all at once.",
    rating: 5.5,
    actors: ["Larry", "Moe", "Curly"]
  }
}
```

1. Copy the following program and paste it into a file named `MoviesItemOps03.html`.

```

<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

```

```

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function updateItem() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    var params = {
        TableName:table,
        Key:{
            "year": year,
            "title": title
        },
        UpdateExpression: "set info.rating = :r, info.plot=:p, info.actors=:a",
        ExpressionAttributeValues:{
            ":r":5.5,
            ":p":"Everything happens all at once.",
            ":a":["Larry", "Moe", "Curly"]
        },
        ReturnValues:"UPDATED_NEW"
    };

    docClient.update(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to update item: " +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "UpdateItem succeeded: " +
            "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="updateItem" type="button" value="Update Item" onclick="updateItem();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

```

```
</body>
</html>
```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs Amazon DynamoDB to return only the updated attributes ("`UPDATED_NEW`").

2. Open the `MoviesItemOps03.html` file in your browser.
3. Choose **Update Item**.

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update` method to increment or decrement the value of an attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run it, the program increments this attribute by one.

1. Copy the following program and paste it into a file named `MoviesItemOps04.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  accessKeyId: "fakeMyKeyId",
  // secretAccessKey default can be used while using the downloadable version of
  DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function increaseRating() {
  var table = "Movies";
  var year = 2015;
  var title = "The Big New Movie";
```

```

var params = {
    TableName:table,
    Key:{
        "year": year,
        "title": title
    },
    UpdateExpression: "set info.rating = info.rating + :val",
    ExpressionAttributeValues:{
        ":val":1
    },
    ReturnValues:"UPDATED_NEW"
};

docClient.update(params, function(err, data) {
    if (err) {
        document.getElementById('textarea').innerHTML = "Unable to update rating: " +
        "\n" + JSON.stringify(err, undefined, 2);
    } else {
        document.getElementById('textarea').innerHTML = "Increase Rating succeeded: " +
        "\n" + JSON.stringify(data, undefined, 2);
    }
});
}

</script>
</head>

<body>
<input id="increaseRating" type="button" value="Increase Rating"
    onclick="increaseRating();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

2. Open the `MoviesItemOps04.html` file in your browser.
3. Choose **Increase Rating**.

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy the following program and paste it into a file named `MoviesItemOps05.html`.

```

<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>

```

```

<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function conditionalUpdate() {
    var table = "Movies";
    var year = 2015;
    var title = "The Big New Movie";

    // Conditional update (will fail)
    var params = {
        TableName:table,
        Key:{
            "year": year,
            "title": title
        },
        UpdateExpression: "remove info.actors[0]",
        ConditionExpression: "size(info.actors) > :num",
        ExpressionAttributeValues:{
            ":num":3
        },
        ReturnValues:"UPDATED_NEW"
    };

    docClient.update(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "The conditional update
failed: " + "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "The conditional update
succeeded: " + "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="conditionalUpdate" type="button" value="Conditional Update"
onclick="conditionalUpdate();;" />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesItemOps05.html` file in your browser.
3. Choose **Conditional Update**.

The program should fail with the following message:

The conditional update failed

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

4. Modify the program so that the ConditionExpression looks like the following.

```
ConditionExpression: "size(info.actors) >= :num",
```

The condition is now *greater than or equal to* 3 instead of *greater than* 3.

5. Run the program again. The updateItem operation should now succeed.

Step 3.6: Delete an Item

You can use the delete method to delete one item by specifying its primary key. You can optionally provide a ConditionExpression to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into a file named MoviesItemOps06.html.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  accessKeyId: "fakeMyKeyId",
  // secretAccessKey default can be used while using the downloadable version of
  DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function conditionalDelete() {
  var table = "Movies";
  var year = 2015;
```

```

var title = "The Big New Movie";

var params = {
    TableName:table,
    Key:{
        "year":year,
        "title":title
    },
    ConditionExpression:"info.rating <= :val",
    ExpressionAttributeValues: {
        ":val": 5.0
    }
};

docClient.delete(params, function(err, data) {
    if (err) {
        document.getElementById('textarea').innerHTML = "The conditional delete failed: " + "\n" + JSON.stringify(err, undefined, 2);
    } else {
        document.getElementById('textarea').innerHTML = "The conditional delete succeeded: " + "\n" + JSON.stringify(data, undefined, 2);
    }
});
}

</script>
</head>

<body>
<input id="conditionalDelete" type="button" value="Conditional Delete"
    onclick="conditionalDelete();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

2. Open the `MoviesItemOps06.html` file in your browser.
3. Choose **Conditional Delete**.

The program should fail with the following message:

The conditional delete failed

This is because the rating for this particular movie is greater than 5.

4. Modify the program to remove the condition from `params`.

```

var params = {
    TableName:table,
    Key:{
        "title":title,
        "year":year
    }
};

```

5. Run the program again. The delete succeeds because you removed the condition.

Step 4: Query and Scan the Data with JavaScript and DynamoDB

You can use the `query` method to retrieve data from a table. You must specify a partition key value; the sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key); for example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to the `query` method, you can use the `scan` method to retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 111\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 113\)](#)
- [Step 4.3: Scan \(p. 114\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the year 1985.

1. Copy the following program and paste it into a file named `MoviesQuery01.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  accessKeyId: "fakeMyKeyId",
})
```

```

// secretAccessKey default can be used while using the downloadable version of
DynamoDB.
// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function queryData() {
    document.getElementById('textarea').innerHTML += "Querying for movies from 1985.';

    var params = {
        TableName : "Movies",
        KeyConditionExpression: "#yr = :yyyy",
        ExpressionAttributeNames:{
            "#yr": "year"
        },
        ExpressionAttributeValues: {
            ":yyyy":1985
        }
    };

    docClient.query(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML += "Unable to query. Error: " +
+ "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML += "Querying for movies from
1985: " + "\n" + JSON.stringify(data, undefined, 2);
        }
    });
}

</script>
</head>

<body>
<input id="queryData" type="button" value="Query" onclick="queryData();" />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

Note

`ExpressionAttributeNames` provides name substitution. This is used because `year` is a reserved word in Amazon DynamoDB. You can't use it directly in any expression, including `KeyConditionExpression`. For this reason, you use the expression attribute name `#yr`. `ExpressionAttributeValues` provides value substitution. This is used because you can't use literals in any expression, including `KeyConditionExpression`. For this reason, you use the expression attribute value `:yyyy`.

2. Open the `MoviesQuery01.html` file in your browser.
3. Choose **Query**.

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in year 1992, with title beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into a file named `MoviesQuery02.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
  region: "us-west-2",
  endpoint: 'http://localhost:8000',
  // accessKeyId default can be used while using the downloadable version of DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  accessKeyId: "fakeMyKeyId",
  // secretAccessKey default can be used while using the downloadable version of
  DynamoDB.
  // For security reasons, do not store AWS Credentials in your files. Use Amazon
  Cognito instead.
  secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function queryData() {
  document.getElementById('textarea').innerHTML += "Querying for movies from 1985.";

  var params = {
    TableName : "Movies",
    ProjectionExpression:"#yr, title, info.genres, info.actors[0]",
    KeyConditionExpression: "#yr = :yyyy and title between :letter1 and :letter2",
    ExpressionAttributeNames:{
      "#yr": "year"
    },
    ExpressionAttributeValues: {
      ":yyyy":1992,
      ":letter1": "A",
      ":letter2": "L"
    }
  };

  docClient.query(params, function(err, data) {
    if (err) {
      document.getElementById('textarea').innerHTML += "Unable to query. Error: "
      + "\n" + JSON.stringify(err, undefined, 2);
    }
  });
}
```

```

        } else {
            document.getElementById('textarea').innerHTML += "Querying for movies
from 1992 - titles A-L, with genres and lead actor: " + "\n" + JSON.stringify(data,
undefined, 2);
        }
    });

</script>
</head>

<body>
<input id="queryData" type="button" value="Query" onclick="queryData();" />
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>
</body>
</html>

```

2. Open the `MoviesQuery02.html` file in your browser.
3. Choose **Query**.

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The `scan` specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all the others.

1. Copy the following program and paste it into a file named `MoviesScan.html`.

```

<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.

```

```

// For security reasons, do not store AWS Credentials in your files. Use Amazon
Cognito instead.
secretAccessKey: "fakeSecretAccessKey"
});

var docClient = new AWS.DynamoDB.DocumentClient();

function scanData() {
    document.getElementById('textarea').innerHTML += "Scanning Movies table." + "\n";

    var params = {
        TableName: "Movies",
        ProjectionExpression: "#yr, title, info.rating",
        FilterExpression: "#yr between :start_yr and :end_yr",
        ExpressionAttributeNames: {
            "#yr": "year",
        },
        ExpressionAttributeValues: {
            ":start_yr": 1950,
            ":end_yr": 1959
        }
    };

    docClient.scan(params, onScan);

    function onScan(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML += "Unable to scan the table:" +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            // Print all the movies
            document.getElementById('textarea').innerHTML += "Scan succeeded. " + "\n";
            data.Items.forEach(function(movie) {
                document.getElementById('textarea').innerHTML += movie.year + ": " +
                movie.title + " - rating: " + movie.info.rating + "\n";
            });

            // Continue scanning if we have more movies (per scan 1MB limitation)
            document.getElementById('textarea').innerHTML += "Scanning for more..." +
            "\n";
            params.ExclusiveStartKey = data.LastEvaluatedKey;
            docClient.scan(params, onScan);
        }
    }
}

</script>
</head>

<body>
<input id="scanData" type="button" value="Scan" onclick="scanData();"/>
<br><br>
<textarea readonly id= "textarea" style="width:400px; height:800px"></textarea>

</body>
</html>

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.

2. Open the `MoviesScan.html` file in your browser.

3. Choose Scan.

Note

You also can use the Scan operation with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: Delete the Table with JavaScript

To delete the Movies table:

1. Copy the following program and paste it into a file named `MoviesDeleteTable.html`.

```
<!--
Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

This file is licensed under the Apache License, Version 2.0 (the "License").
You may not use this file except in compliance with the License. A copy of
the License is located at

http://aws.amazon.com/apache2.0/

This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
-->
<html>
<head>
<script src="https://sdk.amazonaws.com/js/aws-sdk-2.7.16.min.js"></script>

<script>
AWS.config.update({
    region: "us-west-2",
    endpoint: 'http://localhost:8000',
    // accessKeyId default can be used while using the downloadable version of DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    accessKeyId: "fakeMyKeyId",
    // secretAccessKey default can be used while using the downloadable version of
    DynamoDB.
    // For security reasons, do not store AWS Credentials in your files. Use Amazon
    Cognito instead.
    secretAccessKey: "fakeSecretAccessKey"
});

var dynamodb = new AWS.DynamoDB();

function deleteMovies() {
    var params = {
        TableName : "Movies"
    };

    dynamodb.deleteTable(params, function(err, data) {
        if (err) {
            document.getElementById('textarea').innerHTML = "Unable to delete table: " +
            "\n" + JSON.stringify(err, undefined, 2);
        } else {
            document.getElementById('textarea').innerHTML = "Table deleted.";
        }
    });
}

</script>
```

```
</head>

<body>
<input id="deleteTableButton" type="button" value="Delete Table"
      onclick="deleteMovies();"/>
<br><br>
<textarea readonly id="textarea" style="width:400px; height:800px"></textarea>

</body>
</html>
```

2. Open the `MoviesDeleteTable.html` file in your browser.
3. Choose **Delete Table**.

Summary and Review of JavaScript and DynamoDB Tutorial

In this tutorial, you created the `Movies` table in Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the DynamoDB web service.

Updating the AWS Configuration Region to Use the DynamoDB Web Service

To use the DynamoDB web service, you must update the AWS Region in your application. You also have to make sure that Amazon Cognito is available in that same Region so that your browser scripts can authenticate successfully.

```
AWS.config.update({region: "aws-region"});
```

For example, if you want to use the us-west-2 Region, set the following Region.

```
AWS.config.update({region: "us-west-2"});
```

The program now uses the Amazon DynamoDB web service in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the *AWS General Reference*. For more information about setting Regions and endpoints in your code, see [Setting the Region](#) in the *AWS SDK for JavaScript Getting Started Guide*.

Configuring AWS Credentials in Your Files Using Amazon Cognito

The recommended way to obtain AWS credentials for your web and mobile applications is to use Amazon Cognito. Amazon Cognito helps you avoid hardcoding your AWS credentials in your files. Amazon Cognito uses AWS Identity and Access Management (IAM) roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For more information, see [Configuring AWS Credentials in Your Files Using Amazon Cognito \(p. 930\)](#).

Getting Started with Node.js and DynamoDB

In this tutorial, you use the AWS SDK for JavaScript to write simple applications to perform the following Amazon DynamoDB operations:

- Create a table named `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

As you work through this tutorial, you can refer to the [AWS SDK for JavaScript API Reference](#).

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 135\)](#).

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up the AWS SDK for JavaScript:
 - Install [Node.js](#).
 - Install the [AWS SDK for JavaScript](#).

For more information, see the [AWS SDK for JavaScript Getting Started Guide](#).

Step 1: Create a Table with in DynamoDB with AWS SDK for JavaScript

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy the following program and paste it into a file named `MoviesCreateTable.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */
```

```
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var dynamodb = new AWS.DynamoDB();

var params = {
    TableName : "Movies",
    KeySchema: [
        { AttributeName: "year", KeyType: "HASH"}, //Partition key
        { AttributeName: "title", KeyType: "RANGE" } //Sort key
    ],
    AttributeDefinitions: [
        { AttributeName: "year", AttributeType: "N" },
        { AttributeName: "title", AttributeType: "S" }
    ],
    ProvisionedThroughput: {
        ReadCapacityUnits: 10,
        WriteCapacityUnits: 10
    }
};

dynamodb.createTable(params, function(err, data) {
    if (err) {
        console.error("Unable to create table. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("Created table. Table description JSON:", JSON.stringify(data, null, 2));
    }
});
```

Note

- You set the endpoint to indicate that you are creating the table in Amazon DynamoDB on your computer.
 - In the `createTable` call, you specify table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this tutorial.)
2. To run the program, enter the following command.

```
node MoviesCreateTable.js
```

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data in DynamoDB with AWS SDK for JavaScript

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 120\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 121\)](#)

You use a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a year, a title, and a JSON map named info.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The year and title are used as the primary key attribute values for the Movies table.
- The rest of the info values are stored in a single attribute called info. This program illustrates how you can store JSON in an Amazon DynamoDB attribute.

The following is an example of movie data.

```
{  
  "year" : 2013,  
  "title" : "Turn It Down, Or Else!",  
  "info" : {  
    "directors" : [  
      "Alice Smith",  
      "Bob Jones"  
    ],  
    "release_date" : "2013-01-18T00:00:00Z",  
    "rating" : 6.2,  
    "genres" : [  
      "Comedy",  
      "Drama"  
    ],  
    "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
    "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
    "rank" : 11,  
    "running_time_secs" : 5215,  
    "actors" : [  
      "David Matthewman",  
      "Ann Thomas",  
      "Jonathan G. Neff"  
    ]  
  }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (moviedata.json) from the archive.

3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesLoadData.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
var fs = require('fs');  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();  
  
console.log("Importing movies into DynamoDB. Please wait.");  
  
var allMovies = JSON.parse(fs.readFileSync('moviedata.json', 'utf8'));  
allMovies.forEach(function(movie) {  
    var params = {  
        TableName: "Movies",  
        Item: {  
            "year": movie.year,  
            "title": movie.title,  
            "info": movie.info  
        }  
    };  
  
    docClient.put(params, function(err, data) {  
        if (err) {  
            console.error("Unable to add movie", movie.title, ". Error JSON:",  
JSON.stringify(err, null, 2));  
        } else {  
            console.log("PutItem succeeded:", movie.title);  
        }  
    });  
});
```

2. To run the program, enter the following command.

```
node MoviesLoadData.js
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 122\)](#)
- [Step 3.2: Read an Item \(p. 123\)](#)
- [Step 3.3: Update an Item \(p. 124\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 126\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 127\)](#)
- [Step 3.6: Delete an Item \(p. 128\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesItemOps01.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();  
  
var table = "Movies";  
  
var year = 2015;  
var title = "The Big New Movie";  
  
var params = {  
    TableName:table,  
    Item:{  
        "year": year,  
        "title": title,  
        "info":{  
            "plot": "Nothing happens at all.",  
            "rating": 0  
        }  
    }  
}
```

```

};

console.log("Adding a new item...");
docClient.put(params, function(err, data) {
    if (err) {
        console.error("Unable to add item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("Added item:", JSON.stringify(data, null, 2));
    }
});

```

Note

The primary key is required. This code adds an item that has a primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.

2. To run the program, enter the following command.

```
node MoviesItemOps01.js
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the `get` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy the following program and paste it into a file named `MoviesItemOps02.js`.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});
```

```

var docClient = new AWS.DynamoDB.DocumentClient();

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

var params = {
    TableName: table,
    Key:{ 
        "year": year,
        "title": title
    }
};

docClient.get(params, function(err, data) {
    if (err) {
        console.error("Unable to read item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("GetItem succeeded:", JSON.stringify(data, null, 2));
    }
});

```

2. To run the program, enter the following command.

```
node MoviesItemOps02.js
```

Step 3.3: Update an Item

You can use the `update` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

Previously, you added the following item to the table.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

The item is updated as follows.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
```

```
    }  
}
```

1. Copy the following program and paste it into a file named `MoviesItemOps03.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient()  
  
var table = "Movies";  
  
var year = 2015;  
var title = "The Big New Movie";  
  
// Update the item, unconditionally,  
  
var params = {  
    TableName:table,  
    Key:{  
        "year": year,  
        "title": title  
    },  
    UpdateExpression: "set info.rating = :r, info.plot=:p, info.actors=:a",  
    ExpressionAttributeValues:{  
        ":r":5.5,  
        ":p":"Everything happens all at once.",  
        ":a":["Larry", "Moe", "Curly"]  
    },  
    ReturnValues:"UPDATED_NEW"  
};  
  
console.log("Updating the item...");  
docClient.update(params, function(err, data) {  
    if (err) {  
        console.error("Unable to update item. Error JSON:", JSON.stringify(err, null, 2));  
    } else {  
        console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));  
    }  
});
```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes ("`UPDATED_NEW`").

2. To run the program, enter the following command.

```
node MoviesItemOps03.js
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, where you use the `update` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run the program, it increments this attribute by one.

1. Copy the following program and paste it into a file named `MoviesItemOps04.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient()  
  
var table = "Movies";  
  
var year = 2015;  
var title = "The Big New Movie";  
  
// Increment an atomic counter  
  
var params = {  
    TableName:table,  
    Key:{  
        "year": year,  
        "title": title  
    },  
    UpdateExpression: "set info.rating = info.rating + :val",  
    ExpressionAttributeValues:{  
        ":val": 1  
    },
```

```
        ReturnValues:"UPDATED_NEW"
    };

    console.log("Updating the item...");
    docClient.update(params, function(err, data) {
        if (err) {
            console.error("Unable to update item. Error JSON:", JSON.stringify(err, null,
2));
        } else {
            console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));
        }
    });
}
```

2. To run the program, enter the following command.

```
node MoviesItemOps04.js
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy the following program and paste it into a file named `MoviesItemOps05.js`.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient()

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

// Conditional update (will fail)

var params = {
    TableName:table,
    Key:{
        "year": year,
```

```

        "title": title
    },
    UpdateExpression: "remove info.actors[0]",
    ConditionExpression: "size(info.actors) > :num",
    ExpressionAttributeValues:{ 
        ":num": 3
    },
    ReturnValues:"UPDATED_NEW"
};

console.log("Attempting a conditional update...");
docClient.update(params, function(err, data) {
    if (err) {
        console.error("Unable to update item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("UpdateItem succeeded:", JSON.stringify(data, null, 2));
    }
});

```

2. To run the program, enter the following command.

```
node MoviesItemOps05.js
```

The program should fail with the following message.

The conditional request failed

This is because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like the following.

```
ConditionExpression: "size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The `updateItem` operation should now succeed.

Step 3.6: Delete an Item

You can use the `delete` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into a file named `MoviesItemOps06.js`.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 *
```

```

 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
var AWS = require("aws-sdk");

AWS.config.update({
    region: "us-west-2",
    endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

var table = "Movies";

var year = 2015;
var title = "The Big New Movie";

var params = {
    TableName:table,
    Key:{
        "year": year,
        "title": title
    },
    ConditionExpression:"info.rating <= :val",
    ExpressionAttributeValues: {
        ":val": 5.0
    }
};

console.log("Attempting a conditional delete...");
docClient.delete(params, function(err, data) {
    if (err) {
        console.error("Unable to delete item. Error JSON:", JSON.stringify(err, null, 2));
    } else {
        console.log("DeleteItem succeeded:", JSON.stringify(data, null, 2));
    }
});

```

2. To run the program, enter the following command.

```
node MoviesItemOps06.js
```

The program should fail with the following message.

The conditional request failed

This is because the rating for this particular movie is greater than 5.

3. Modify the program to remove the condition from `params`.

```

var params = {
    TableName:table,
    Key:{
        "title":title,
        "year":year
    }
};

```

4. Run the program again. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan Data with AWS SDK for JavaScript in DynamoDB

You can use the `query` method to retrieve data from a table. You must specify a partition key value; the sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is number.
- `title` – The sort key. The attribute type is string.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key). For example, you can find movies released in 2014 that have a title starting with the letter "A".

In addition to the `query` method, you also can use the `scan` method, which can retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 130\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 131\)](#)
- [Step 4.3: Scan \(p. 132\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the year 1985.

1. Copy the following program and paste it into a file named `MoviesQuery01.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();
```

```
console.log("Querying for movies from 1985.");

var params = {
    TableName : "Movies",
    KeyConditionExpression: "#yr = :yyyy",
    ExpressionAttributeNames:{
        "#yr": "year"
    },
    ExpressionAttributeValues: {
        ":yyyy": 1985
    }
};

docClient.query(params, function(err, data) {
    if (err) {
        console.error("Unable to query. Error:", JSON.stringify(err, null, 2));
    } else {
        console.log("Query succeeded.");
        data.Items.forEach(function(item) {
            console.log(" - ", item.year + ": " + item.title);
        });
    }
});
```

Note

`ExpressionAttributeNames` provides name substitution. You use this because `year` is a reserved word in Amazon DynamoDB. You can't use it directly in any expression, including `KeyConditionExpression`. You use the expression attribute name `#yr` to address this. `ExpressionAttributeValues` provides value substitution. You use this because you cannot use literals in any expression, including `KeyConditionExpression`. You use the expression attribute value `:yyyy` to address this.

2. To run the program, enter the following command.

```
node MoviesQuery01.js
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in year 1992, with title beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into a file named `MoviesQuery02.js`.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
```

```

    * the License is located at
    *
    * http://aws.amazon.com/apache2.0/
    *
    * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
    * CONDITIONS OF ANY KIND, either express or implied. See the License for the
    * specific language governing permissions and limitations under the License.
    */
var AWS = require("aws-sdk");

AWS.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
});

var docClient = new AWS.DynamoDB.DocumentClient();

console.log("Querying for movies from 1992 - titles A-L, with genres and lead actor");

var params = {
  TableName : "Movies",
  ProjectionExpression:"#yr, title, info.genres, info.actors[0]",
  KeyConditionExpression: "#yr = :yyyy and title between :letter1 and :letter2",
  ExpressionAttributeNames:{
    "#yr": "year"
  },
  ExpressionAttributeValues: {
    ":yyyy": 1992,
    ":letter1": "A",
    ":letter2": "L"
  }
};

docClient.query(params, function(err, data) {
  if (err) {
    console.log("Unable to query. Error:", JSON.stringify(err, null, 2));
  } else {
    console.log("Query succeeded.");
    data.Items.forEach(function(item) {
      console.log(" -", item.year + ": " + item.title
      + " ... " + item.info.genres
      + " ... " + item.info.actors[0]);
    });
  }
});

```

2. To run the program, enter the following command.

```
node MoviesQuery02.js
```

Step 4.3: Scan

The `scan` method reads every item in the table and returns all the data in the table. You can provide an optional `filter_expression`, so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all of the others.

1. Copy the following program and paste it into a file named `MoviesScan.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var docClient = new AWS.DynamoDB.DocumentClient();  
  
var params = {  
    TableName: "Movies",  
    ProjectionExpression: "#yr, title, info.rating",  
    FilterExpression: "#yr between :start_yr and :end_yr",  
    ExpressionAttributeNames: {  
        "#yr": "year",  
    },  
    ExpressionAttributeValues: {  
        ":start_yr": 1950,  
        ":end_yr": 1959  
    }  
};  
  
console.log("Scanning Movies table.");  
docClient.scan(params, onScan);  
  
function onScan(err, data) {  
    if (err) {  
        console.error("Unable to scan the table. Error JSON:", JSON.stringify(err,  
null, 2));  
    } else {  
        // print all the movies  
        console.log("Scan succeeded.");  
        data.Items.forEach(function(movie) {  
            console.log(  
                movie.year + ":",  
                movie.title, "- rating:", movie.info.rating);  
        });  
  
        // continue scanning if we have more movies, because  
        // scan can retrieve a maximum of 1MB of data  
        if (typeof data.LastEvaluatedKey != "undefined") {  
            console.log("Scanning for more...");  
            params.ExclusiveStartKey = data.LastEvaluatedKey;  
            docClient.scan(params, onScan);  
        }  
    }  
}
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
 - `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. To run the program, enter the following command.

```
node MoviesScan.js
```

Note

You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: (Optional): Delete the Table using AWS SDK for JavaScript

To delete the `Movies` table:

1. Copy the following program and paste it into a file named `MoviesDeleteTable.js`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
var AWS = require("aws-sdk");  
  
AWS.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
});  
  
var dynamodb = new AWS.DynamoDB();  
  
var params = {  
    TableName : "Movies"  
};  
  
dynamodb.deleteTable(params, function(err, data) {  
    if (err) {  
        console.error("Unable to delete table. Error JSON:", JSON.stringify(err, null,  
        2));  
    } else {  
        console.log("Deleted table. Table description JSON:", JSON.stringify(data,  
        null, 2));  
    }  
});
```

2. To run the program, enter the following command.

```
node MoviesDeleteTable.js
```

Summary

In this tutorial, you created the `Movies` table in Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application. To do this, modify the following code.

```
AWS.config.update({endpoint: "https://dynamodb.aws-region.amazonaws.com"});
```

For example, if you want to use the us-west-2 Region, set the following endpoint.

```
AWS.config.update({endpoint: "https://dynamodb.us-west-2.amazonaws.com"});
```

Instead of using DynamoDB on your computer, the program now uses the DynamoDB web service endpoint in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting Regions and endpoints in your code, see [Setting the Region](#) in the [AWS SDK for JavaScript Developer Guide](#).

Getting Started with .NET and DynamoDB

In this tutorial, you use the AWS SDK for .NET to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table named `Movies` using a utility program written in C#, and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

The DynamoDB module of the AWS SDK for .NET offers several programming models for different use cases. In this exercise, the C# code uses the document model, which provides a level of abstraction that is often convenient. It also uses the low-level API, which handles nested attributes more effectively.

For information about the document model API, see [.NET: Document Model \(p. 273\)](#). For information about the low-level API, see [Working with DynamoDB Tables in .NET \(p. 367\)](#).

Topics

- [.NET and DynamoDB Tutorial Prerequisites \(p. 136\)](#)
- [Step 1: Create a DynamoDB Client \(p. 137\)](#)
- [Step 2: Create a DynamoDB Table Using the Low-Level API \(p. 139\)](#)
- [Step 3: Load Sample Data into the DynamoDB Table \(p. 142\)](#)

- Step 4: Add a Movie to the DynamoDB Table (p. 145)
- Step 5: Read and Display a Record from the DynamoDB Table (p. 146)
- Step 6: Update the New Movie Record in the DynamoDB Table (p. 147)
- Step 7: Conditionally Delete (p. 150)
- Step 8: Query a DynamoDB Table with .NET (p. 152)
- Step 9: Scan the Movies Table with .NET (p. 155)
- Step 10: Delete the Movies Table with .NET (p. 156)

.NET and DynamoDB Tutorial Prerequisites

The [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#) describes how to use the AWS SDK for .NET to create simple programs for performing Amazon DynamoDB operations.

Before you begin, follow these steps to ensure that you have all the prerequisites needed to complete the tutorial:

- Use a computer that is running a recent version of Windows and a current version of Microsoft Visual Studio. If you don't have Visual Studio installed, you can download a free copy of the Community edition from the [Microsoft Visual Studio website](#).
- Download and run DynamoDB (downloadable version). For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For more information about how to run the same code against the DynamoDB web service, see [Step 1: Create a DynamoDB Client \(p. 137\)](#).

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up a security profile for DynamoDB in Visual Studio. For step-by-step instructions, see [.NET Code Examples \(p. 332\)](#).
- Open the getting started demo solution that is used in this tutorial in Visual Studio:
 1. Download a .zip archive containing the solution from [DynamoDB_intro.zip](#).
 2. Save the archive to a convenient place on your computer, and extract (unzip) the files in it.
 3. In Visual Studio, press **Ctrl+Shift+O**, or choose **Open** on the **File** menu and choose **Project/Solution**.
 4. Navigate to `DynamoDB_intro.sln` in the `DynamoDB_intro` directory that you unzipped, and choose **Open**.
- Build the `DynamoDB_intro` solution, and then open the `00_Main.cs` file in Visual Studio.
 1. On the **Build** menu, choose **Build Solution** (or press **Ctrl+Shift+B**). The solution should build successfully.
 2. Make sure that the **Solution Explorer** pane is being displayed and pinned in Visual Studio. If it isn't, you can find it in the **View** menu, or by pressing **Ctrl+Alt+L**.
 3. In **Solution Explorer**, open the `00_Main.cs` file. This is the file that controls the execution of the demo program that is used in this tutorial.

Note

This tutorial shows how to use asynchronous methods rather than synchronous methods. This is because .NET core supports only asynchronous methods, and also because the asynchronous model is generally preferable when performance is crucial. For more information, see [AWS Asynchronous APIs for .NET](#).

Installing the External Dependencies of the DynamoDB_intro Solution

The `DynamoDB_intro` solution already has the Amazon DynamoDB SDK installed in it. It also has the open-source `Newtonsoft.Json`.NET library for deserializing JSON data, licensed under the MIT License (MIT) (see <https://github.com/JamesNK/Newtonsoft.Json/blob/master/LICENSE.md>).

To install the NuGet package for the DynamoDB module of the AWS SDK for .NET version 3 in your own programs, open the **NuGet Package Manager Console** on the **Tools** menu in Visual Studio. Then enter the following command at the `PM>` prompt.

```
PM> Install-Package AWSSDK.DynamoDBv2
```

In a similar way, you can use the **NuGet Package Manager Console** to load the `Json`.NET library into your own projects in Visual Studio. At the `PM>` prompt, enter the following command.

```
PM> Install-Package Newtonsoft.Json
```

Next Step

[Step 1: Create a DynamoDB Client \(p. 137\)](#)

Step 1: Create a DynamoDB Client

The first step in the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#) is to create a client that gives you access to the Amazon DynamoDB API. The `Main` function in `DynamoDB_intro` does this by calling a `createClient` function implemented in the `01_CreateClient.cs` file.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Net.Sockets;  
using Amazon.DynamoDBv2;  
  
namespace DynamoDB_intro  
{  
    public static partial class Ddb_Intro  
    {  
        /*-----  
         * If you are creating a client for the DynamoDB service, make sure your credentials  
         * are set up first, as explained in:  
         * https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/  
SettingUp.DynamoWebService.html,  
         *  
         * If you are creating a client for DynamoDBLocal (for testing purposes),  
         * DynamoDB-Local should be started first. For most simple testing, you can keep
```

```

*   data in memory only, without writing anything to disk. To do this, use the
*   following command line:
*
*       java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -inMemory
*
*   For information about DynamoDBLocal, see:
*   https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
DynamoDBLocal.html.
*-----
*/
/*-----
*      createClient
*-----*/
public static bool createClient( bool useDynamoDBLocal )
{
    if( useDynamoDBLocal )
    {
        operationSucceeded = false;
        operationFailed = false;

        // First, check to see whether anyone is listening on the DynamoDB local port
        // (by default, this is port 8000, so if you are using a different port, modify
this accordingly)
        bool localFound = false;
        try
        {
            using (var tcp_client = new TcpClient())
            {
                var result = tcp_client.BeginConnect("localhost", 8000, null, null);
                localFound = result.AsyncWaitHandle.WaitOne(3000); // Wait 3 seconds
                tcp_client.EndConnect(result);
            }
        }
        catch
        {
            localFound = false;
        }
        if( !localFound )
        {
            Console.WriteLine("\n      ERROR: DynamoDB Local does not appear to have been
started..." +
                           "\n              (checked port 8000)");
            operationFailed = true;
            return (false);
        }
    }

    // If DynamoDB-Local does seem to be running, so create a client
    Console.WriteLine( "  -- Setting up a DynamoDB-Local client (DynamoDB Local seems
to be running) );
    AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig();
    ddbConfig.ServiceURL = "http://localhost:8000";
    try { client = new AmazonDynamoDBClient( ddbConfig ); }
    catch( Exception ex )
    {
        Console.WriteLine( "      FAILED to create a DynamoDBLocal client; " +
ex.Message );
        operationFailed = true;
        return false;
    }
}

else
{
    try { client = new AmazonDynamoDBClient( ); }
    catch( Exception ex )
{

```

```
        Console.WriteLine("      FAILED to create a DynamoDB client; " + ex.Message);
        operationFailed = true;
    }
    operationSucceeded = true;
    return true;
}
}
```

Main calls this function with the `useDynamoDBLocal` parameter set to `true`. Therefore the local test version of DynamoDB must already be running on your computer using the default port (8000), or the call fails. If you do not have it installed yet, see [Running DynamoDB on Your Computer \(p. 46\)](#).

Setting the `useDynamoDBLocal` parameter to `false` creates a client for the DynamoDB service itself rather than the local test program.

Next Step

[Step 2: Create a DynamoDB Table Using the Low-Level API \(p. 139\)](#)

Step 2: Create a DynamoDB Table Using the Low-Level API

The document model in the AWS SDK for .NET doesn't provide for creating tables, so you have to use the low-level APIs. For more information, see [Working with DynamoDB Tables in .NET \(p. 367\)](#).

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you create a table named `Movies` in Amazon DynamoDB. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.

Note

Because "year" is a reserved word in DynamoDB, you need to create an alias for it (such as `#yr`) using an `ExpressionAttributeNames` object when referring to it in a low-level expression.

- `title` – The sort key. The `AttributeType` is `S` for string.

The `Main` function in `DynamoDB_intro` does this by waiting on an asynchronous `CreatingTable_async` function implemented in the `02_CreatingTable.cs` file.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
```

```

using System.Threading.Tasks;
using Amazon.DynamoDBv2.Model;

namespace DynamoDB_intro
{
    public static partial class Ddb_Intro
    {
        /*-----
         *          CreatingTable_async
         *-----*/
        public static async Task CreatingTable_async( string new_table_name,
                                                    List<AttributeDefinition> table_attributes,
                                                    List<KeySchemaElement> table_key_schema,
                                                    ProvisionedThroughput provisionedThroughput )
        {
            Console.WriteLine( " -- Creating a new table named {0}...", new_table_name );
            if( await checkingTableExistence_async( new_table_name ) )
            {
                Console.WriteLine( " -- No need to create a new table..." );
                return;
            }
            if( operationFailed )
                return;

            operationSucceeded = false;
            Task<bool> newTbl = CreateNewTable_async( new_table_name,
                                                       table_attributes,
                                                       table_key_schema,
                                                       provisionedThroughput );
            await newTbl;
        }

        /*-----
         *          checkingTableExistence_async
         *-----*/
        static async Task<bool> checkingTableExistence_async( string tblNm )
        {
            DescribeTableResponse descResponse;

            operationSucceeded = false;
            operationFailed = false;
            ListTablesResponse tblResponse = await Ddb_Intro.client.ListTablesAsync();
            if (tblResponse.TableNames.Contains(tblNm))
            {
                Console.WriteLine(" A table named {0} already exists in DynamoDB!", tblNm);

                // If the table exists, get its description
                try
                {
                    descResponse = await
Ddb_Intro.client.DescribeTableAsync(Ddb_Intro.movies_table_name);
                    operationSucceeded = true;
                }
                catch (Exception ex)
                {
                    Console.WriteLine(" However, its description is not available ({0})",
ex.Message);
                    Ddb_Intro.moviesTableDescription = null;
                    operationFailed = true;
                    return ( true );
                }
                Ddb_Intro.moviesTableDescription = descResponse.Table;
                return ( true );
            }
            return ( false );
        }
    }
}

```

```

        }

    /*-----
     *          CreateNewTable_async
     *-----*/
    public static async Task<bool> CreateNewTable_async( string table_name,
                                                       List<AttributeDefinition>
table_attributes,
                                                       List<KeySchemaElement>
table_key_schema,
                                                       ProvisionedThroughput
provisioned_throughput )
    {
        CreateTableRequest request;
        CreateTableResponse response;

        // Build the 'CreateTableRequest' structure for the new table
        request = new CreateTableRequest
        {
            TableName           = table_name,
            AttributeDefinitions = table_attributes,
            KeySchema           = table_key_schema,
            // Provisioned-throughput settings are always required,
            // although the local test version of DynamoDB ignores them.
            ProvisionedThroughput = provisioned_throughput
        };

        operationSucceeded = false;
        operationFailed = false;
        try
        {
            Task<CreateTableResponse> makeTbl = Ddb_Intro.client.CreateTableAsync( request );
            response = await makeTbl;
            Console.WriteLine( "      -- Created the \"{0}\" table successfully!", table_name );
            operationSucceeded = true;
        }
        catch( Exception ex )
        {
            Console.WriteLine( "      FAILED to create the new table, because: {0}.",
ex.Message );
            operationFailed = true;
            return( false );
        }

        // Report the status of the new table...
        Console.WriteLine( "      Status of the new table: '{0}'.",
response.TableDescription.TableStatus );
        Ddb_Intro.moviesTableDescription = response.TableDescription;
        return ( true );
    }
}

```

The `CreatingTable_async` function starts by waiting on an asynchronous `checkingTableExistence_async` function to determine whether a table named "Movies" already exists. If the table exists, `checkingTableExistence_async` retrieves the `TableDescription` for the existing Table.

If the table doesn't already exist, `CreatingTable_async` waits on `CreateNewTable_async` to create the `Movies` table using the DynamoDB client API `CreateTableAsync`.

The DynamoDB_intro sample uses asynchronous methods rather than synchronous methods wherever possible. This is because .NET core supports only asynchronous methods, and the asynchronous model

is generally preferable when performance is crucial. For more information, see [AWS Asynchronous APIs for .NET](#).

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Next Step

[Step 3: Load Sample Data into the DynamoDB Table \(p. 142\)](#)

Step 3: Load Sample Data into the DynamoDB Table

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you populate the new `Movies` table in Amazon DynamoDB with sample data from the Internet Movie Database (IMDb). This data is stored in JSON format in a local text file named `moviedata.json`.

For each movie, `moviedata.json` defines a `year` name-value pair, a `title` name-value pair, and a complex `info` object, as illustrated by the following example.

```
{  
    "year" : 2013,  
    "title" : "Turn It Down, Or Else!",  
    "info" : {  
        "directors" : [  
            "Alice Smith",  
            "Bob Jones"  
        ],  
        "release_date" : "2013-01-18T00:00:00Z",  
        "rating" : 6.2,  
        "genres" : [  
            "Comedy",  
            "Drama"  
        ],  
        "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
        "rank" : 11,  
        "running_time_secs" : 5215,  
        "actors" : [  
            "David Matthewman",  
            "Ann Thomas",  
            "Jonathan G. Neff"  
        ]  
    }  
}
```

Before loading the `moviedata.json` file, the `Main` function in `DynamoDB_intro` checks to determine whether the `Movies` table exists and is still empty. If so, it waits on an asynchronous `LoadingData_async` function that is implemented in the `03_LoadingData.cs` file.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 */
```

```

/*
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.IO;
using System.Threading.Tasks;
using Amazon.DynamoDBv2.DocumentModel;

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace DynamoDB_intro
{
    public static partial class Ddb_Intro
    {

        /*-----
         *      LoadingData_async
         *-----*/
        public static async Task LoadingData_async( Table table, string filePath )
        {
            JArray movieArray;

            movieArray = await ReadJsonMovieFile_async( filePath );
            if( movieArray != null )
                await LoadJsonMovieData_async( table, movieArray );
        }

        /*-----
         *          ReadJsonMovieFile_async
         *-----*/
        public static async Task<JArray> ReadJsonMovieFile_async( string JsonMovieFilePath )
        {
            StreamReader sr = null;
            JsonTextReader jtr = null;
            JArray movieArray = null;

            Console.WriteLine( " -- Reading the movies data from a JSON file..." );
            operationSucceeded = false;
            operationFailed = false;
            try
            {
                sr = new StreamReader( JsonMovieFilePath );
                jtr = new JsonTextReader( sr );
                movieArray = (JArray) await JToken.ReadFromAsync( jtr );
                operationSucceeded = true;
            }
            catch( Exception ex )
            {
                Console.WriteLine( "     ERROR: could not read the file!\n" + Reason: {0}., ex.Message );
                operationFailed = true;
            }
            finally
            {
                if( jtr != null )
                    jtr.Close( );
                if( sr != null )
                    sr.Close( );
            }
            if( operationSucceeded )
            {
                Console.WriteLine( " -- Succeeded in reading the JSON file!" );
                return ( movieArray );
            }
        }
    }
}

```

```

        return ( null );
    }

/*
 *          LoadJsonMovieData_async
 */
public static async Task LoadJsonMovieData_async( Table moviesTable, JArray
moviesArray )
{
    operationSucceeded = false;
    operationFailed = false;

    int n = moviesArray.Count;
    Console.WriteLine( "      -- Starting to load {0:#,##0} movie records into the Movies
table asynchronously...\n" + "" +
                      "      Wrote: ", n );
    for( int i = 0, j = 99; i < n; i++ )
    {
        try
        {
            string itemJson = moviesArray[i].ToString();
            Document doc = Document.FromJson(itemJson);
            Task putItem = moviesTable.PutItemAsync(doc);
            if( i >= j )
            {
                j++;
                Console.WriteLine( "{0,5:#,##0}, ", j );
                if( j % 1000 == 0 )
                    Console.WriteLine( "\n" );
                j += 99;
            }
            await putItem;
        }
        catch( Exception ex )
        {
            Console.WriteLine( "\n      ERROR: Could not write the movie record #{0:#,##0},
because:\n          {1}", i, ex.Message );
            operationFailed = true;
            break;
        }
    }
    if( !operationFailed )
    {
        operationSucceeded = true;
        Console.WriteLine( "\n      -- Finished writing all movie records to DynamoDB!" );
    }
}
}

```

LoadingData_async begins by waiting on ReadJsonMovieFile_async. This function reads the moviedata.json file using the open-source [Newtonsoft Json.NET library](#), which is licensed under the [MIT License](#).

When the data has been read successfully, LoadingData_async waits on LoadJsonMovieData_async to load the movie records into the Movies table using the DynamoDB document-model Table.PutItemAsync API. For information about the document model API, see [.NET: Document Model \(p. 273\)](#).

Next Step

[Step 4: Add a Movie to the DynamoDB Table \(p. 145\)](#)

Step 4: Add a Movie to the DynamoDB Table

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you add a new movie record to the `Movies` table in Amazon DynamoDB. The `Main` function in `DynamoDB_intro` starts by creating a DynamoDB document model `Document` and then waits on `writingNewMovie_async`, which is implemented in the `04_WritingNewItem.cs` file.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Threading.Tasks;  
using Amazon.DynamoDBv2.DocumentModel;  
  
namespace DynamoDB_intro  
{  
    public static partial class Ddb_Intro  
    {  
        /*-----  
         *      WritingNewMovie  
         *-----*/  
        public static async Task WritingNewMovie_async( Document newItem )  
        {  
            operationSucceeded = false;  
            operationFailed = false;  
  
            int year = (int) newItem["year"];  
            string name = newItem["title"];  
  
            if( await ReadingMovie_async( year, name, false ) )  
                Console.WriteLine( " The {0} movie \"\{1\}\" is already in the Movies table...\n" +  
                    " -- No need to add it again... its info is as follows:\n{2}",  
                    year, name, movie_record.ToStringPretty() );  
            else  
            {  
                try  
                {  
                    Task<Document> writeNew = moviesTable.PutItemAsync(newItem, token);  
                    Console.WriteLine(" -- Writing a new movie to the Movies table...");  
                    await writeNew;  
                    Console.WriteLine(" -- Wrote the item successfully!");  
                    operationSucceeded = true;  
                }  
                catch (Exception ex)  
                {  
                    Console.WriteLine(" FAILED to write the new movie, because:\n {0}.",  
                        ex.Message);  
                    operationFailed = true;  
                }  
            }  
        }  
    }  
}
```

`WritingNewMovie_async` begins by checking to determine whether the new movie has already been added to the `Movies` table. If it has not, it waits for the `DynamoDB.Table.PutItemAsyn` method to add the new movie record.

For More Information

- To learn more about reading and writing data in DynamoDB tables, see [Working with Items and Attributes \(p. 374\)](#).
- For more information about the DynamoDB document model API, see [.NET: Document Model \(p. 273\)](#).
- For more information about asynchronous methods, see [AWS Asynchronous APIs for .NET](#).

Next Step

[Step 5: Read and Display a Record from the DynamoDB Table \(p. 146\)](#)

Step 5: Read and Display a Record from the DynamoDB Table

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you retrieve and display the new movie record that you added in [Step 4 \(p. 145\)](#). The `Main` function in `DynamoDB_intro` does this by waiting on `ReadingMovie_async`, which is implemented in the `05_ReadingItem.cs` file.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Threading.Tasks;  
using Amazon.DynamoDbv2.DocumentModel;  
  
namespace DynamoDB_intro  
{  
    public static partial class Ddb_Intro  
    {  
        /*-----  
         *                               ReadingMovie_async  
         *-----*/  
        public static async Task<bool> ReadingMovie_async( int year, string title, bool report )  
        {  
            // Create Primitives for the HASH and RANGE portions of the primary key  
            Primitive hash = new Primitive(year.ToString(), true);  
            Primitive range = new Primitive(title, false);  
  
            operationSucceeded = false;  
            operationFailed = false;  
            try
```

```
{  
    Task<Document> readMovie = moviesTable.GetItemAsync(hash, range, token);  
    if( report )  
        Console.WriteLine( " -- Reading the {0} movie \"{1}\" from the Movies table...",  
year, title );  
    movie_record = await readMovie;  
    if( movie_record == null )  
    {  
        if( report )  
            Console.WriteLine( " -- Sorry, that movie isn't in the Movies table." );  
        return ( false );  
    }  
    else  
    {  
        if( report )  
            Console.WriteLine( " -- Found it! The movie record looks like this:\n" +  
                movie_record.ToJsonPretty( ) );  
        operationSucceeded = true;  
        return ( true );  
    }  
}  
catch( Exception ex )  
{  
    Console.WriteLine( " FAILED to get the movie, because: {0}.", ex.Message );  
    operationFailed = true;  
}  
return ( false );  
}
```

`ReadingMovie_async` in turn waits on the `DynamoDB Table.GetItemAsyn` method to retrieve the new movie record as a `Document`. `ReadingMovie_async` then displays the movie as JSON text using the `Document.ToJsonPretty` method.

For More Information

- To learn more about reading and writing data in DynamoDB tables, see [Working with Items and Attributes \(p. 374\)](#).
- For more information about the DynamoDB document model API, see [.NET: Document Model \(p. 273\)](#).
- For more information about asynchronous methods, see [AWS Asynchronous APIs for .NET](#).

Next Step

[Step 6: Update the New Movie Record in the DynamoDB Table \(p. 147\)](#)

Step 6: Update the New Movie Record in the DynamoDB Table

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you update the new movie record in several different ways.

Topics

- [Change Plot and Rating, and Add Actors \(p. 148\)](#)
- [Increment the Movie Rating Atomically \(p. 149\)](#)
- [Try to Update Using a Condition That Fails \(p. 150\)](#)
- [For More Information \(p. 150\)](#)

- [Next Step \(p. 150\)](#)

Change Plot and Rating, and Add Actors

The `Main` function in `DynamoDB_intro` changes the plot and rating of the movie record that you added in [Step 4 \(p. 145\)](#), and it also adds a list of actors to it. The document model in the AWS SDK for .NET doesn't support updating nested attributes such as the items under the `info` attribute. Because of this, `Main` uses the low-level client API rather than a document-model Table method.

It starts by creating a low-level `UpdateItemRequest` to make this change.

```
UpdateItemRequest updateRequest = new UpdateItemRequest()
{
    TableName = movies_table_name,
    Key = new Dictionary<string, AttributeValue>
    {
        { partition_key_name, new AttributeValue { N = "2018" } },
        { sort_key_name, new AttributeValue { S = "The Big New Movie" } }
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>
    {
        { ":r", new AttributeValue { N = "5.5" } },
        { ":p", new AttributeValue { S = "Everything happens all at once!" } },
        { ":a", new AttributeValue { L = new List<AttributeValue>
            {
                new AttributeValue { S = "Larry" },
                new AttributeValue { S = "Moe" },
                new AttributeValue { S = "Curly" }
            }
        } }
    },
    UpdateExpression = "SET info.rating = :r, info.plot = :p, info.actors = :a",
    ReturnValue = "NONE"
};
```

Setting `ReturnValue` to `NONE` specifies that no update information should be returned. However, when `Main` then waits on `UpdatingMovie_async`, it sets the `report` parameter to `true`. This causes `UpdatingMovie_async` to change `ReturnValue` to `ALL_NEW`, meaning that the updated item should be returned in its entirety.

The `UpdatingMovie_async` is implemented in the `06_UpdatingItem.cs` file.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Text;
using Amazon.DynamoDBv2.Model;
```

```

namespace DynamoDB_intro
{
    public static partial class Ddb_Intro
    {
        /*
         *                               UpdatingMovie_async
         */
        public static async Task<bool> UpdatingMovie_async( UpdateItemRequest updateRequest,
        bool report )
        {
            UpdateItemResponse updateResponse = null;

            operationSucceeded = false;
            operationFailed = false;
            if( report )
            {
                Console.WriteLine( " -- Trying to update a movie item..." );
                updateRequest.ReturnValues = "ALL_NEW";
            }

            try
            {
                updateResponse = await client.UpdateItemAsync( updateRequest );
                Console.WriteLine( "      -- SUCCEEDED in updating the movie item!" );
            }
            catch( Exception ex )
            {
                Console.WriteLine( "      -- FAILED to update the movie item, because:\n{0}.", ex.Message );
                if( updateResponse != null )
                    Console.WriteLine( "          -- The status code was " +
updateResponse.HttpStatusCode.ToString( ) );
                operationFailed = true;return ( false );
            }
            if( report )
            {
                Console.WriteLine( "      Here is the updated movie information:" );
                Console.WriteLine( movieAttributesToJson( updateResponse.Attributes ) );
            }
            operationSucceeded = true;
            return ( true );
        }
    }
}

```

`UpdatingMovie_async` waits on the low-level `DynamoDB.Client.UpdateItemAsync` method to update the movie record. If the update is successful, and if the `report` parameter is `true`, `UpdatingMovie_async` displays the updated movie record.

Where the document model has a handy `Document.ToJsonPretty()` method for displaying document content, working with low-level attribute values is a little more complicated. The `00b_DDB_Attributes.cs` file can provide some examples of how to access and work with `AttributeValue` objects.

Increment the Movie Rating Atomically

DynamoDB supports the atomic update of counters, where you use a low-level update method to increment or decrement the value of an existing attribute without interference from other write requests. (All write requests in DynamoDB are applied in the order in which they are received.)

To increment the rating value in the movie that you just created, the `Main` function makes the following changes in the `UpdateItemRequest` that it used in the previous update.

```
updateRequest.ExpressionAttributeValues = new Dictionary<string, AttributeValue>
{
    { ":inc", new AttributeValue { N = "1" } }
};
updateRequest.UpdateExpression = "SET info.rating = info.rating + :inc";
```

Then, once again, it waits on `UpdatingMovie_async` to make the change.

Try to Update Using a Condition That Fails

You can also add a condition to an update request, so that if the condition is not met, the update does not occur.

To demonstrate this, the `Main` function makes the following changes to the `UpdateItemRequest` that it just used to increment the movie rating.

```
updateRequest.ExpressionAttributeValues.Add( ":n", new AttributeValue { N = "3" } );
updateRequest.ConditionExpression = "size(info.actors) > :n";
```

The update can now occur only if there are more than three actors in the movie record being updated. Because there are only three actors listed, the condition fails when `Main` waits on `UpdatingMovie_async`, and the update does not occur.

For More Information

- To learn more about reading and writing data in DynamoDB tables, see [Working with Items and Attributes \(p. 374\)](#).
- For more information about the DynamoDB document model API, see [.NET: Document Model \(p. 273\)](#).
- For more information about asynchronous methods, see [AWS Asynchronous APIs for .NET](#).

Next Step

[Step 7: Conditionally Delete \(p. 150\)](#)

Step 7: Conditionally Delete

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you try to delete a movie record with a condition that is not met, and the deletion fails. Then, when the condition is changed so that it is met, the deletion succeeds.

The `Main` function in `DynamoDB_intro` starts by creating a condition as follows.

```
Expression condition = new Expression();
condition.ExpressionAttributeValues[":val"] = 5.0;
condition.ExpressionStatement = "info.rating <= :val";
```

The `Main` function then passes the `Expression` as one of the parameters of `DeletingItem_async` and waits on it. `DeletingItem_async` is implemented in the `07_DeletingItem.cs` file.

```
/**
```

```

/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Threading.Tasks;
using Amazon.DynamoDbv2.DocumentModel;

namespace DynamoDB_intro
{
    public static partial class Ddb_Intro
    {

        /*
         *           DeletingItem_async
         */
        public static async Task<bool> DeletingItem_async( Table table, int year, string title,
            Expression condition=null )
        {
            Document deletedItem = null;
            operationSucceeded = false;
            operationFailed = false;

            // Create Primitives for the HASH and RANGE portions of the primary key
            Primitive hash = new Primitive(year.ToString(), true);
            Primitive range = new Primitive(title, false);
            DeleteItemOperationConfig deleteConfig = new DeleteItemOperationConfig( );
            deleteConfig.ConditionalExpression = condition;
            deleteConfig.ReturnValues = ReturnValues.AllOldAttributes;

            Console.WriteLine( " -- Trying to delete the {0} movie \"{1}\"...", year, title );
            try
            {
                Task<Document> delItem = table.DeleteItemAsync( hash, range, deleteConfig );
                deletedItem = await delItem;
            }
            catch( Exception ex )
            {
                Console.WriteLine( "      FAILED to delete the movie item, for this reason:\n{0}\n", ex.Message );
                operationFailed = true;
                return ( false );
            }
            Console.WriteLine( "      -- SUCCEEDED in deleting the movie record that looks like
this:\n" +
                deletedItem.ToJsonPretty( ) );
            operationSucceeded = true;
            return ( true );
        }
    }
}

```

DeletingItem_async in turn includes the condition Expression in a DeleteItemOperationConfig object that it passes to the DynamoDB Table.DeleteItemAsync method when it waits on it.

Because the movie's rating is 6.5, which is higher than 5.0, the condition is not met, and the deletion fails.

Then, when the `Main` function changes the rating threshold in the condition to 7.0 instead of 5.0, the deletion succeeds.

Next Step

[Step 8: Query a DynamoDB Table with .NET \(p. 152\)](#)

Step 8: Query a DynamoDB Table with .NET

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you query the `Movies` table in three different ways.

Topics

- [Use a Simple Document Model Search to Query for 1985 Movies \(p. 152\)](#)
- [Use a QueryOperationConfig to Create a More Complex Query Search \(p. 154\)](#)
- [Use a Low-Level Query to Find 1992 Movies with Titles Between 'M...' and 'Tzz...' \(p. 154\)](#)
- [Next Step \(p. 155\)](#)

Use a Simple Document Model Search to Query for 1985 Movies

To set up a simple document-model query, the `Main` function in `DynamoDB_intro` creates a `Search` object by calling the `Table.Query` API with 1985 as the *partition key* (also known as the *hash key*), and an empty filter `Expression`.

```
try { search = moviesTable.Query( 1985, new Expression( ) ); }
```

It then waits on `SearchListing_async`, which is implemented in `08_Querying.cs` to retrieve and display the query results.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Text;  
using System.Threading.Tasks;  
using Amazon.DynamoDBv2.Model;  
using Amazon.DynamoDBv2.DocumentModel;  
using System.Collections.Generic;  
  
namespace DynamoDB_intro  
{  
    public static partial class Ddb_Intro  
    {  
        /*-----  
         *-----  
        -----*/  
        public static async Task<SearchResponse> SearchListing_async  
        {  
            var search = moviesTable.Query( 1985, new Expression( ) );  
            return await search.GetResultAsync();  
        }  
    }  
}
```

```

*-----*/
public static async Task<bool> SearchListing_async( Search search )
{
    int i = 0;
    List<Document> docList = new List<Document>();
    Console.WriteLine( "           Here are the movies retrieved:\n" +
                      "           " );
    Task<List<Document>> getNextBatch;
    operationSucceeded = false;
    operationFailed = false;

    do
    {
        try
        {
            getNextBatch = search.GetNextSetAsync( );
            docList = await getNextBatch;
        }
        catch( Exception ex )
        {
            Console.WriteLine( "           FAILED to get the next batch of movies from Search!
Reason:\n           " + ex.Message );
            operationFailed = true;
            return ( false );
        }

        foreach( Document doc in docList )
        {
            i++;
            showMovieDocShort( doc );
        }
    } while( !search.IsDone );
    Console.WriteLine( "           -- Retrieved {0} movies.", i );
    operationSucceeded = true;
    return ( true );
}

/*-----*/
*                         ClientQuerying_async
*-----*/
public static async Task<bool> ClientQuerying_async( QueryRequest qRequest )
{
    operationSucceeded = false;
    operationFailed = false;

    QueryResponse qResponse;
    try
    {
        Task<QueryResponse> clientQueryTask = client.QueryAsync( qRequest );
        qResponse = await clientQueryTask;
    }
    catch( Exception ex )
    {
        Console.WriteLine( "           The low-level query FAILED, because:\n           {0}.",
ex.Message );
        operationFailed = true;
        return ( false );
    }
    Console.WriteLine( "           -- The low-level query succeeded, and returned {0} movies!",
qResponse.Items.Count );
    if( !pause( ) )
    {
}

```

```

        operationFailed = true;
        return ( false );
    }
    Console.WriteLine( "           Here are the movies retrieved:" +
                       "-----" );
    foreach( Dictionary<string, AttributeValue> item in qResponse.Items )
        showMovieAttrsShort( item );

    Console.WriteLine( "      -- Retrieved {0} movies.", qResponse.Items.Count );
    operationSucceeded = true;
    return ( true );
}
}

```

Use a QueryOperationConfig to Create a More Complex Query Search

To query for 1992 movies with titles from "B..." to "Hzz...", the Main function creates a QueryOperationConfig object with a QueryFilter and various other fields.

```

QueryOperationConfig config = new QueryOperationConfig();
config.Filter = new QueryFilter();
config.Filter.AddCondition( "year", QueryOperator.Equal, new DynamoDBEntry[ ]
{ 1992 } );
config.Filter.AddCondition( "title", QueryOperator.Between, new DynamoDBEntry[ ]
{ "B", "Hzz" } );
config.AttributesToGet = new List<string> { "year", "title", "info" };
config.Select = SelectValues.SpecificAttributes;

```

Once again, it then creates a Search object by calling the Table.Query API, this time with the QueryOperationConfig object as the only parameter.

And again, it waits on SearchListing_async to retrieve and display the query results.

Use a Low-Level Query to Find 1992 Movies with Titles Between 'M...' and 'Tzz...'

To use a low-level query to retrieve 1992 movies with titles from "M..." to "Tzz...", the Main function creates a QueryRequest object.

```

QueryRequest qRequest= new QueryRequest
{
    TableName = "Movies",
    ExpressionAttributeNames = new Dictionary<string, string>
    {
        { "#yr", "year" }
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>
    {
        { ":qYr",   new AttributeValue { N = "1992" } },
        { ":tSt",   new AttributeValue { S = "M" } },
        { ":tEn",   new AttributeValue { S = "Tzz" } }
    },
    KeyConditionExpression = "#yr = :qYr and title between :tSt and :tEn",
    ProjectionExpression = "#yr, title, info.actors[0], info.genres,
    info.running_time_secs"
};

```

It then waits on the `ClientQuerying_async` function implemented in the `08_Querying.cs` file. `ClientQuerying_async` waits in turn on the low-level DynamoDB method `AmazonDynamoDBClient.QueryAsync` to retrieve the query results.

Note

Because "year" is a reserved word in DynamoDB, you need to create an alias for it (here `#yr`) using `ExpressionAttributeNames` in order to use it in a low-level expression.

Next Step

[Step 9: Scan the Movies Table with .NET \(p. 155\)](#)

Step 9: Scan the Movies Table with .NET

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you scan the `Movies` table in two different ways: by using a document model scan and a low-level scan.

Topics

- [Use a Document Model Search to Scan for 1950s Movies \(p. 155\)](#)
- [Use a Low-Level Scan to Retrieve 1960s Movies \(p. 155\)](#)
- [Next Step \(p. 156\)](#)

Use a Document Model Search to Scan for 1950s Movies

To set up a document model scan for 1950s movies, the `Main` function in `DynamoDB_intro` creates a `ScanOperationConfig` object with a `ScanFilter`:

```
ScanFilter filter = new ScanFilter( );
filter.AddCondition( "year", ScanOperator.Between, new DynamoDBEntry[ ] { 1950,
1959 } );
ScanOperationConfig scanConfig = new ScanOperationConfig
{
    AttributesToGet = new List<string> { "year, title, info" },
    Filter = filter
};
```

To obtain a `Search` object for the scan, it passes the `ScanOperationConfig` object to `Table.Scan`. Using the `Search` object, it then waits on `SearchListing_async` (implemented in `08_Querying.cs`) to retrieve and display the scan results.

Use a Low-Level Scan to Retrieve 1960s Movies

To set up a low-level scan for 1960s movies, the `Main` function in `DynamoDB_intro` creates a `ScanRequest` object with various fields:

```
ScanRequest sRequest = new ScanRequest
{
    TableName = "Movies",
    ExpressionAttributeNames = new Dictionary<string, string>
    {
        { "#yr", "year" }
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>
    {
        { ":y_a", new AttributeValue { N = "1960" } },
        { ":y_z", new AttributeValue { N = "1969" } },
    }
};
```

```
        },
        FilterExpression = "#yr between :y_a and :y_z",
        ProjectionExpression = "#yr, title, info.actors[0], info.directors,
info.running_time_secs"
    };
```

It then waits on the `ClientScanning_async` function implemented in the `09_Scanning.cs` file. `ClientScanning_async` waits in turn on the low-level DynamoDB method `AmazonDynamoDBClient.ScanAsync` to retrieve the query results.

Note

Because "year" is a reserved word in DynamoDB, you must create an alias for it (here `#yr`) using `ExpressionAttributeNames` in order to use it in a low-level expression.

Next Step

[Step 10: Delete the Movies Table with .NET \(p. 156\)](#)

Step 10: Delete the Movies Table with .NET

In this step of the [Microsoft .NET and DynamoDB Tutorial \(p. 135\)](#), you delete the `Movies` table in Amazon DynamoDB.

The `Main` function in `DynamoDB_intro` waits on `DeletingTable_async`, which is implemented in the `10_DeletingTable.cs` file.

```
/***
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Threading.Tasks;

namespace DynamoDB_intro
{
    public static partial class Ddb_Intro
    {
        //-------------------------------------
        *             DeletingTable_async
        *-----*/
        public static async Task<bool> DeletingTable_async( string tableName )
        {
            operationSucceeded = false;
            operationFailed = false;

            Console.WriteLine( " -- Trying to delete the table named \"{0}\", {1} ", tableName );
            pause( );
            Task tblDelete = client.DeleteTableAsync( tableName );
            try
            {
                await tblDelete;
            }
        }
    }
}
```

```
        catch( Exception ex )
    {
        Console.WriteLine( "      ERROR: Failed to delete the table, because:\n" +
+ ex.Message );
        operationFailed = true;
        return ( false );
    }
    Console.WriteLine( "      -- Successfully deleted the table!" );
    operationSucceeded = true;
    pause( );
    return ( true );
}
```

`DeletingTable_async` waits on the low-level DynamoDB method `AmazonDynamoDBClient.DeleteTableAsync` to delete the table.

For More Information

- To learn more about reading and writing data in DynamoDB tables, see [Working with Items and Attributes \(p. 374\)](#).
- For more information about the DynamoDB document model API, see [.NET: Document Model \(p. 273\)](#).
- For more information about asynchronous methods, see [AWS Asynchronous APIs for .NET](#).

PHP and DynamoDB

In this tutorial, you use the AWS SDK for PHP to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

As you work through this tutorial, you can refer to the [AWS SDK for PHP Developer Guide](#). The [Amazon DynamoDB section](#) in the [AWS SDK for PHP API Reference](#) describes the parameters and results for DynamoDB operations.

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB service, see the [Summary \(p. 177\)](#).

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up the AWS SDK for PHP:
 - [Install PHP](#).
 - [Install the SDK for PHP](#).

For more information, see [Getting Started](#) in the [AWS SDK for PHP Getting Started Guide](#).

Step 1: Create a Table

In this step, you create a table named `Movies`. The primary key for the table is composed of the following two attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy the following program and paste it into a file named `MoviesCreateTable.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
  
$params = [  
    'TableName' => 'Movies',  
    'KeySchema' => [  
        [  
            'AttributeName' => 'year',  
            'KeyType' => 'HASH' //Partition key  
        ],  
        [  
            'AttributeName' => 'title',  
            'KeyType' => 'RANGE' //Sort key  
        ]  
    ],  
    'AttributeDefinitions' => [  
        [  
            'AttributeName' => 'year',  
            'AttributeType' => 'N'  
        ],  
        [  
            'AttributeName' => 'title',  
            'AttributeType' => 'S'  
        ],  
    ],  
];
```

```
'ProvisionedThroughput' => [
    'ReadCapacityUnits' => 10,
    'WriteCapacityUnits' => 10
]
];

try {
    $result = $dynamodb->createTable($params);
    echo 'Created table. Status: ' .
        $result['TableDescription']['TableStatus'] . "\n";
} catch (DynamoDbException $e) {
    echo "Unable to create table:\n";
    echo $e->getMessage() . "\n";
}
```

Note

- You set the endpoint to indicate that you are creating the table in Amazon DynamoDB on your computer.
 - In the `createTable` call, you specify the table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required, but the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
2. To run the program, enter the following command.

```
php MoviesCreateTable.php
```

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 160\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 160\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a year, a title, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ...,  
    "title" : ....,  
    "info" : { ... }  
  },  
  ...
```

]

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in an Amazon DynamoDB attribute.

The following is an example of movie data.

```
{  
    "year" : 2013,  
    "title" : "Turn It Down, Or Else!",  
    "info" : {  
        "directors" : [  
            "Alice Smith",  
            "Bob Jones"  
        ],  
        "release_date" : "2013-01-18T00:00:00Z",  
        "rating" : 6.2,  
        "genres" : [  
            "Comedy",  
            "Drama"  
        ],  
        "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
        "rank" : 11,  
        "running_time_secs" : 5215,  
        "actors" : [  
            "David Matthewman",  
            "Ann Thomas",  
            "Jonathan G. Neff"  
        ]  
    }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy and paste the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesLoadData.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 */
```

```
* http://aws.amazon.com/apache2.0/
*
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
* CONDITIONS OF ANY KIND, either express or implied. See the License for the
* specific language governing permissions and limitations under the License.
*/
require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region' => 'us-west-2',
    'version' => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaller = new Marshaler();

$tableName = 'Movies';

$movies = json_decode(file_get_contents('moviedata.json'), true);

foreach ($movies as $movie) {

    $year = $movie['year'];
    $title = $movie['title'];
    $info = $movie['info'];

    $json = json_encode([
        'year' => $year,
        'title' => $title,
        'info' => $info
    ]);

    $params = [
        'TableName' => $tableName,
        'Item' => $marshaller->marshalJson($json)
    ];

    try {
        $result = $dynamodb->putItem($params);
        echo "Added movie: " . $movie['year'] . " " . $movie['title'] . "\n";
    } catch (DynamoDbException $e) {
        echo "Unable to add movie:\n";
        echo $e->getMessage() . "\n";
        break;
    }
}
```

Note

The [DynamoDB Marshaler class](#) has methods for converting JSON documents and PHP arrays to the DynamoDB format. In this program, `$marshaller->marshalJson($json)` takes a JSON document and converts it into a DynamoDB item.

2. To run the program, enter the following command.

```
php MoviesLoadData.php
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 162\)](#)
- [Step 3.2: Read an Item \(p. 163\)](#)
- [Step 3.3: Update an Item \(p. 165\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 167\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 168\)](#)
- [Step 3.6: Delete an Item \(p. 169\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesItemOps01.php`.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region' => 'us-west-2',
    'version' => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaller = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';
```

```

$item = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . '' ,
    "info": {
        "plot": "Nothing happens at all.",
        "rating": 0
    }
});
$params = [
    'TableName' => 'Movies',
    'Item' => $item
];

try {
    $result = $dynamodb->putItem($params);
    echo "Added item: $year - $title\n";

} catch (DynamoDbException $e) {
    echo "Unable to add item:\n";
    echo $e->getMessage() . "\n";
}

```

Note

The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores a map that provides more information about the movie.

2. To run the program, enter the following command.

```
php MoviesItemOps01.php
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

You can use the `getItem` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy the following program and paste it into a file named `MoviesItemOps02.php`.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
```

```
* This file is licensed under the Apache License, Version 2.0 (the "License").  
* You may not use this file except in compliance with the License. A copy of  
* the License is located at  
*  
* http://aws.amazon.com/apache2.0/  
*  
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
* CONDITIONS OF ANY KIND, either express or implied. See the License for the  
* specific language governing permissions and limitations under the License.  
*/  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaler = new Marshaler();  
  
$tableName = 'Movies';  
  
$year = 2015;  
$title = 'The Big New Movie';  
  
$key = $marshaler->marshalJson(''  
{  
    "year": ' . $year . ',  
    "title": "' . $title . '"  
}  
' );  
  
$params = [  
    'TableName' => $tableName,  
    'Key' => $key  
];  
  
try {  
    $result = $dynamodb->getItem($params);  
    print_r($result["Item"]);  
}  
} catch (DynamoDbException $e) {  
    echo "Unable to get item:\n";  
    echo $e->getMessage() . "\n";  
}
```

-
2. To run the program, enter the following command.

```
php MoviesItemOps02.php
```

Step 3.3: Update an Item

You can use the `updateItem` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The following shows the existing item.

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Nothing happens at all.",  
        rating: 0  
    }  
}
```

The item is updated as follows.

```
{  
    year: 2015,  
    title: "The Big New Movie",  
    info: {  
        plot: "Everything happens all at once.",  
        rating: 5.5,  
        actors: ["Larry", "Moe", "Curly"]  
    }  
}
```

1. Copy the following program and paste it into a file named `MoviesItemOps03.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
]);
```

```
        'version' => 'latest'
    ]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": '$year',
    "title": "' . $title . '"
}
');

$eav = $marshaler->marshalJson(
{
    ":r": 5.5 ,
    ":p": "Everything happens all at once.",
    ":a": [ "Larry", "Moe", "Curly" ]
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' =>
        'set info.rating = :r, info.plot=:p, info.actors=:a',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];
try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item.\n";
    print_r($result['Attributes']);
} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}
```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs Amazon DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, enter the following command.

```
php MoviesItemOps03.php
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, which use the `updateItem` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run the program, it increments this attribute by one.

1. Copy the following program and paste it into a file named `MoviesItemOps04.php`.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaler = new Marshaler();

$tableName = 'Movies';

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . '"
}
');

$eav = $marshaler->marshalJson(
{
    ":val": 1
}
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'set info.rating = info.rating + :val',
]
```

```

    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];

try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);
}

} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}

```

2. To run the program, enter the following command.

```
php MoviesItemOps04.php
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors in the movie.

1. Copy the following program and paste it into a file named `MoviesItemOps05.php`.

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

$ sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
$marshaller = new Marshaler();

$tableName = 'Movies';

```

```

$year = 2015;
$title = 'The Big New Movie';

$key = $marshaler->marshalJson(
{
    "year": ' . $year . ',
    "title": "' . $title . ''"
});
');

$eav = $marshaler->marshalJson(
{
    ":num": 3
});
');

$params = [
    'TableName' => $tableName,
    'Key' => $key,
    'UpdateExpression' => 'remove info.actors[0]',
    'ConditionExpression' => 'size(info.actors) > :num',
    'ExpressionAttributeValues'=> $eav,
    'ReturnValues' => 'UPDATED_NEW'
];
try {
    $result = $dynamodb->updateItem($params);
    echo "Updated item. ReturnValues are:\n";
    print_r($result['Attributes']);
} catch (DynamoDbException $e) {
    echo "Unable to update item:\n";
    echo $e->getMessage() . "\n";
}

```

2. To run the program, enter the following command.

```
php MoviesItemOps05.php
```

The program should fail with the following message.

The conditional request failed

The program fails because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the ConditionExpression looks like the following.

```
ConditionExpression="size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The UpdateItem operation should now succeed.

Step 3.6: Delete an Item

You can use the `deleteItem` method to delete one item by specifying its primary key. You can optionally provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into a file named `MoviesItemOps06.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaler = new Marshaler();  
  
$tableName = 'Movies';  
  
$year = 2015;  
$title = 'The Big New Movie';  
  
$key = $marshaler->marshalJson(''  
    {  
        "year": ' . $year . ',  
        "title": "' . $title . '"  
    }  
' );  
  
$eav = $marshaler->marshalJson(''  
    {  
        ":val": 5  
    }  
' );  
  
$params = [  
    'TableName' => $tableName,  
    'Key' => $key,  
    'ConditionExpression' => 'info.rating <= :val',  
    'ExpressionAttributeValues'=> $eav  
];  
  
try {  
    $result = $dynamodb->deleteItem($params);  
    echo "Deleted item.\n";  
}  
} catch (DynamoDbException $e) {
```

```
        echo "Unable to delete item:\n";
        echo $e->getMessage() . "\n";
    }
```

2. To run the program, enter the following command.

```
php MoviesItemOps06.php
```

The program should fail with the following message.

The conditional request failed

The program fails because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition.

```
$params = [
    'TableName' => $tableName,
    'Key' => $key
];
```

4. Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is `number`.
- `title` – The sort key. The attribute type is `string`.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key). For example, to find movies released in 2014 that have a title starting with the letter "A".

In addition to the `query` method, you can use the `scan` method to retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 171\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 173\)](#)
- [Step 4.3: Scan \(p. 174\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the year 1985.

1. Copy the following program and paste it into a file named `MoviesQuery01.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region' => 'us-west-2',  
    'version' => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaler = new Marshaler();  
  
$tableName = 'Movies';  
  
$eav = $marshaler->marshalJson(''  
    ':yyyy": 1985  
'');  
  
$params = [  
    'TableName' => $tableName,  
    'KeyConditionExpression' => '#yr = :yyyy',  
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],  
    'ExpressionAttributeValues'=> $eav  
];  
  
echo "Querying for movies from 1985.\n";  
  
try {  
    $result = $dynamodb->query($params);  
  
    echo "Query succeeded.\n";  
  
    foreach ($result['Items'] as $movie) {  
        echo $marshaler->unmarshalValue($movie['year']) . ':' .  
            $marshaler->unmarshalValue($movie['title']) . "\n";  
    }  
  
} catch (DynamoDbException $e) {  
    echo "Unable to query:\n";  
    echo $e->getMessage() . "\n";  
}
```

Note

- `ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
 - `ExpressionAttributeValues` provides value substitution. You use this because you can't use literals in any expression, including `KeyConditionExpression`. You can use the expression attribute value `:yyyy` to address this.
2. To run the program, enter the following command:

```
php MoviesItemQuery01.php
```

Note

The preceding program shows how to query a table by its primary key attributes. In Amazon DynamoDB, you can optionally create one or more secondary indexes on a table, and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in year 1992 with title beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into a file named `MoviesQuery02.php`.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
require 'vendor/autoload.php';  
  
date_default_timezone_set('UTC');  
  
use Aws\DynamoDb\Exception\DynamoDbException;  
use Aws\DynamoDb\Marshaler;  
  
$sdk = new Aws\Sdk([  
    'endpoint' => 'http://localhost:8000',  
    'region'   => 'us-west-2',  
    'version'  => 'latest'  
]);  
  
$dynamodb = $sdk->createDynamoDb();  
$marshaller = new Marshaler();
```

```


|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$tableName = 'Movies';  \$eav = \$marshaler->marshalJson( {     ":yyyy":1992,     ":letter1": "A",     ":letter2": "L" } ');  \$params = [     'TableName' => \$tableName,     'ProjectionExpression' => '#yr, title, info.genres, info.actors[0]',     'KeyConditionExpression' =>         '#yr = :yyyy and title between :letter1 and :letter2',     'ExpressionAttributeNames'=> [ '#yr' => 'year' ],     'ExpressionAttributeValues'=> \$eav ];  echo "Querying for movies from 1992 - titles A-L, with genres and lead actor\n";  try {     \$result = \$dynamodb->query(\$params);      echo "Query succeeded.\n";      foreach (\$result['Items'] as \$i) {         \$movie = \$marshaler->unmarshalItem(\$i);         print \$movie['year'] . ' ' . \$movie['title'] . ' ... ';          foreach (\$movie['info']['genres'] as \$gen) {             print \$gen . ' ';         }          echo ' ... ' . \$movie['info']['actors'][0] . "\n";     } }  } catch (DynamoDbException \$e) {     echo "Unable to query:\n";     echo \$e->getMessage() . "\n"; } |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|


```

2. To run the program, enter the following command.

```
php MoviesQuery02.php
```

Step 4.3: Scan

The `scan` method reads every item in the entire table, and returns all of the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items), and discard all of the others.

1. Copy the following program and paste it into a file named `MoviesScan.php`.

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;
use Aws\DynamoDb\Marshaler;

sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();

$marshaller = new Marshaler();

//Expression attribute values
$eav = $marshaller->marshalJson(
{
    ":start_yr": 1950,
    ":end_yr": 1959
}
');

$params = [
    'TableName' => 'Movies',
    'ProjectionExpression' => '#yr, title, info.rating',
    'FilterExpression' => '#yr between :start_yr and :end_yr',
    'ExpressionAttributeNames'=> [ '#yr' => 'year' ],
    'ExpressionAttributeValues'=> $eav
];
echo "Scanning Movies table.\n";

try {
    while (true) {
        $result = $dynamodb->scan($params);

        foreach ($result['Items'] as $i) {
            $movie = $marshaller->unmarshalItem($i);
            echo $movie['year'] . ':' . $movie['title'];
            echo ' ... ' . $movie['info']['rating']
            . "\n";
        }

        if (isset($result['LastEvaluatedKey'])) {
            $params['ExclusiveStartKey'] = $result['LastEvaluatedKey'];
        } else {

```

```
        break;
    }
}

} catch (DynamoDbException $e) {
    echo "Unable to scan:\n";
    echo $e->getMessage() . "\n";
}
```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
 - `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. To run the program, enter the following command.

```
php MoviesScan.php
```

Note

You can also use the `Scan` operation with any secondary indexes that you have created on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program and paste it into a file named `MoviesDeleteTable.php`.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

require 'vendor/autoload.php';

date_default_timezone_set('UTC');

use Aws\DynamoDb\Exception\DynamoDbException;

$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);

$dynamodb = $sdk->createDynamoDb();
```

```
$params = [
    'TableName' => 'Movies'
];

try {
    $result = $dynamodb->deleteTable($params);
    echo "Deleted table.\n";

} catch (DynamoDbException $e) {
    echo "Unable to delete table:\n";
    echo $e->getMessage() . "\n";
}
```

2. To run the program, enter the following command.

```
php MoviesDeleteTable.php
```

Summary

In this tutorial, you created the `Movies` table in Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you need to modify your code so that it uses the DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB web service, you must change the endpoint in your application. To do this, find the following lines in the code.

```
$sdk = new Aws\Sdk([
    'endpoint' => 'http://localhost:8000',
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

Remove the `endpoint` parameter so that the code looks like the following.

```
$sdk = new Aws\Sdk([
    'region'   => 'us-west-2',
    'version'  => 'latest'
]);
```

After you remove this line, the code can access the DynamoDB web service in the AWS Region specified by the `region` config value. For example, the following line specifies that you want to use the US West (Oregon) Region.

```
'region'   => 'us-west-2',
```

Instead of using the downloadable version of DynamoDB on your computer, the program now uses the DynamoDB service endpoint in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting Regions and endpoints in your code, see the [boto: A Python interface to Amazon Web Services](#).

Getting Started Developing with Python and DynamoDB

In this tutorial, you use the AWS SDK for Python (Boto 3) to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

As you work through this tutorial, you can refer to the AWS SDK for Python (Boto) documentation. The following sections are specific to DynamoDB:

- [DynamoDB tutorial](#)
- [DynamoDB low-level client](#)

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. In the [Summary \(p. 191\)](#), we explain how to run the same code against the DynamoDB web service.

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Install Python 2.6 or later. For more information, see <https://www.python.org/downloads>. For instructions, see [Quickstart](#) in the Boto 3 documentation.

Step 1: Create a Table with Python

In this step, you create a table named `Movies`. The primary key for the table is composed of the following attributes:

- `year` – The partition key. The `AttributeType` is `N` for number.
- `title` – The sort key. The `AttributeType` is `S` for string.

1. Copy the following program and paste it into a file named `MoviesCreateTable.py`.

```
import boto3

def create_movie_table(dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.create_table(
        TableName='Movies',
        KeySchema=[
            {
                'AttributeName': 'year',
                'KeyType': 'HASH' # Partition key
            },
            {
                'AttributeName': 'title',
                'KeyType': 'RANGE' # Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'year',
                'AttributeType': 'N'
            },
            {
                'AttributeName': 'title',
                'AttributeType': 'S'
            }
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 1,
            'WriteCapacityUnits': 1
        }
    )
    return table
```

```
        'KeyType': 'HASH' # Partition key
    },
    {
        'AttributeName': 'title',
        'KeyType': 'RANGE' # Sort key
    }
],
AttributeDefinitions=[
    {
        'AttributeName': 'year',
        'AttributeType': 'N'
    },
    {
        'AttributeName': 'title',
        'AttributeType': 'S'
    },
],
ProvisionedThroughput={
    'ReadCapacityUnits': 10,
    'WriteCapacityUnits': 10
}
)
return table

if __name__ == '__main__':
    movie_table = create_movie_table()
    print("Table status:", movie_table.table_status)
```

Note

- You set the endpoint to indicate that you are creating the table in the [downloadable version of DynamoDB](#) on your computer.
 - In the `create_table` call, you specify the table name, primary key attributes, and its data types.
 - The `ProvisionedThroughput` parameter is required. However, the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
 - These examples use the Python 3 style `print` function. The line `from __future__ import print_function` enables Python 3 printing in Python 2.6 and later.
2. To run the program, enter the following command.

```
python MoviesCreateTable.py
```

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 180\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 181\)](#)

This scenario uses a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a year, a title, and a JSON map named `info`.

```
[  
  {  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
  },  
  {  
    "year" : ...,  
    "title" : ...,  
    "info" : { ... }  
  },  
  ...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- The rest of the `info` values are stored in a single attribute called `info`. This program illustrates how you can store JSON in an Amazon DynamoDB attribute.

The following is an example of movie data.

```
{  
  "year" : 2013,  
  "title" : "Turn It Down, Or Else!",  
  "info" : {  
    "directors" : [  
      "Alice Smith",  
      "Bob Jones"  
    ],  
    "release_date" : "2013-01-18T00:00:00Z",  
    "rating" : 6.2,  
    "genres" : [  
      "Comedy",  
      "Drama"  
    ],  
    "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
    "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
    "rank" : 11,  
    "running_time_secs" : 5215,  
    "actors" : [  
      "David Matthewman",  
      "Ann Thomas",  
      "Jonathan G. Neff"  
    ]  
  }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, you can run the following program to populate the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesLoadData.py`.

```
from decimal import Decimal
import json
import boto3

def load_movies(movies, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')
    for movie in movies:
        year = int(movie['year'])
        title = movie['title']
        print("Adding movie:", year, title)
        table.put_item(Item=movie)

if __name__ == '__main__':
    with open("moviedata.json") as json_file:
        movie_list = json.load(json_file, parse_float=Decimal)
    load_movies(movie_list)
```

2. To run the program, enter the following command.

```
python MoviesLoadData.py
```

Step 3: Create, Read, Update, and Delete an Item with Python

In this step, you perform read and write operations on an item in the `Movies` table.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 181\)](#)
- [Step 3.2: Read an Item \(p. 182\)](#)
- [Step 3.3: Update an Item \(p. 183\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 184\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 185\)](#)
- [Step 3.6: Delete an Item \(p. 186\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesItemOps01.py`.

```
from pprint import pprint
import boto3
```

```

def put_movie(title, year, plot, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')
    response = table.put_item(
        Item={
            'year': year,
            'title': title,
            'info': {
                'plot': plot,
                'rating': rating
            }
        }
    )
    return response

if __name__ == '__main__':
    movie_resp = put_movie("The Big New Movie", 2015,
                           "Nothing happens at all.", 0)
    print("Put movie succeeded:")
    pprint(movie_resp, sort_dicts=False)

```

Note

- The primary key is required. This code adds an item that has primary key (`year`, `title`) and `info` attributes. The `info` attribute stores sample JSON that provides more information about the movie.
 - The `DecimalEncoder` class is used to print out numbers stored using the `Decimal` class. The Boto SDK uses the `Decimal` class to hold Amazon DynamoDB number values.
2. To run the program, enter the following command.

```
python MoviesItemOps01.py
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}
```

You can use the `get_item` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy the following program and paste it into a file named `MoviesItemOps02.py`.

```

from pprint import pprint
import boto3
from botocore.exceptions import ClientError

def get_movie(title, year, dynamodb=None):

```

```

if not dynamodb:
    dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

table = dynamodb.Table('Movies')

try:
    response = table.get_item(Key={'year': year, 'title': title})
except ClientError as e:
    print(e.response['Error']['Message'])
else:
    return response['Item']

if __name__ == '__main__':
    movie = get_movie("The Big New Movie", 2015, )
    if movie:
        print("Get movie succeeded:")
        pprint(movie, sort_dicts=False)

```

2. To run the program, enter the following command.

```
python MoviesItemOps02.py
```

Step 3.3: Update an Item

You can use the `update_item` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The following shows the existing item.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

The item is updated as follows.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy the following program and paste it into a file named `MoviesItemOps03.py`.

```
from decimal import Decimal
```

```

from pprint import pprint
import boto3

def update_movie(title, year, rating, plot, actors, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')

    response = table.update_item(
        Key={
            'year': year,
            'title': title
        },
        UpdateExpression="set info.rating=:r, info.plot=:p, info.actors=:a",
        ExpressionAttributeValues={
            ':r': Decimal(rating),
            ':p': plot,
            ':a': actors
        },
        ReturnValues="UPDATED_NEW"
    )
    return response

if __name__ == '__main__':
    update_response = update_movie(
        "The Big New Movie", 2015, 5.5, "Everything happens all at once.",
        ["Larry", "Moe", "Curly"])
    print("Update movie succeeded:")
    pprint(update_response, sort_dicts=False)

```

Note

This program uses `UpdateExpression` to describe all updates you want to perform on the specified item.

The `ReturnValues` parameter instructs DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, enter the following command.

```
python MoviesItemOps03.py
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, which use the `update_item` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run the program, it increments this attribute by one.

1. Copy the following program and paste it into a file named `MoviesItemOps04.py`.

```

from decimal import Decimal
from pprint import pprint
import boto3

def increase_rating(title, year, rating_increase, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

```

```





```

2. To run the program, enter the following command.

```
python MoviesItemOps04.py
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `UpdateItem` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if there are more than three actors.

1. Copy the following program and paste it into a file named `MoviesItemOps05.py`.

```

from pprint import pprint
import boto3
from botocore.exceptions import ClientError

def remove_actors(title, year, actor_count, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')

    try:
        response = table.update_item(
            Key={
                'year': year,
                'title': title
            },
            UpdateExpression="remove info.actors[0]",
            ConditionExpression="size(info.actors) > :num",
            ExpressionAttributeValues={':num': actor_count},
            ReturnValues="UPDATED_NEW"
        )
    except ClientError as e:
        if e.response['Error']['Code'] == "ConditionalCheckFailedException":
            print(e.response['Error']['Message'])
        else:
            raise
    else:
        return response

```

```

        return response

if __name__ == '__main__':
    print("Attempting conditional update (expecting failure)...")
    update_response = remove_actors("The Big New Movie", 2015, 3)
    if update_response:
        print("Update movie succeeded:")
        pprint(update_response, sort_dicts=False)

```

2. To run the program, enter the following command.

```
python MoviesItemOps05.py
```

The program should fail with the following message.

The conditional request failed

The program fails because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the ConditionExpression looks like the following.

```
ConditionExpression="size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The UpdateItem operation should now succeed.

Step 3.6: Delete an Item

You can use the `delete_item` method to delete one item by specifying its primary key. Optionally, you can provide a `ConditionExpression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into a file named `MoviesItemOps06.py`.

```

from decimal import Decimal
from pprint import pprint
import boto3
from botocore.exceptions import ClientError

def delete_underrated_movie(title, year, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')

    try:
        response = table.delete_item(
            Key={
                'year': year,
                'title': title
            },
            ConditionExpression="info.rating <= :val",
            ExpressionAttributeValues={
                ":val": Decimal(rating)
            }
        )
    except ClientError as e:
        print(e.response['Error']['Message'])

```

```
        except ClientError as e:
            if e.response['Error']['Code'] == "ConditionalCheckFailedException":
                print(e.response['Error']['Message'])
            else:
                raise
        else:
            return response

if __name__ == '__main__':
    print("Attempting a conditional delete...")
    delete_response = delete_underrated_movie("The Big New Movie", 2015, 5)
    if delete_response:
        print("Delete movie succeeded:")
        pprint(delete_response, sort_dicts=False)
```

2. To run the program, enter the following command.

```
python MoviesItemOps06.py
```

The program should fail with the following message.

The conditional request failed

The program fails because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition in `table.delete_item`.

```
response = table.delete_item(
    Key={
        'year': year,
        'title': title
    }
)
```

4. Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is `number`.
- `title` – The sort key. The attribute type is `string`.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key). For example, you can find movies released in 2014 that have a title starting with the letter "A".

In addition to the `query` method, you can use the `scan` method to retrieve all the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 188\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 188\)](#)

- [Step 4.3: Scan \(p. 189\)](#)

Step 4.1: Query - All Movies Released in a Year

The program included in this step retrieves all movies released in the year 1985.

1. Copy the following program and paste it into a file named `MoviesQuery01.py`.

```
import boto3
from boto3.dynamodb.conditions import Key

def query_movies(year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')
    response = table.query(
        KeyConditionExpression=Key('year').eq(year)
    )
    return response['Items']

if __name__ == '__main__':
    query_year = 1985
    print(f"Movies from {query_year}")
    movies = query_movies(query_year)
    for movie in movies:
        print(movie['year'], ":", movie['title'])
```

Note

The Boto 3 SDK constructs a `ConditionExpression` for you when you use the `Key` and `Attr` functions imported from `boto3.dynamodb.conditions`. You can also specify a `ConditionExpression` as a string.

For a list of available conditions for Amazon DynamoDB, see [DynamoDB Conditions in AWS SDK for Python \(Boto 3\) Getting Started](#).

For more information, see [Condition Expressions \(p. 392\)](#).

2. To run the program, enter the following command.

```
python MoviesQuery01.py
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The program included in this step retrieves all movies released in year 1992 with title beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into a file named `MoviesQuery02.py`.

```
from pprint import pprint
```

```

import boto3
from boto3.dynamodb.conditions import Key

def query_and_project_movies(year, title_range, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')
    print(f"Get year, title, genres, and lead actor")

    # Expression attribute names can only reference items in the projection expression.
    response = table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=
            Key('year').eq(year) & Key('title').between(title_range[0], title_range[1])
    )
    return response['Items']

if __name__ == '__main__':
    query_year = 1992
    query_range = ('A', 'L')
    print(f"Get movies from {query_year} with titles from "
          f"{query_range[0]} to {query_range[1]}")
    movies = query_and_project_movies(query_year, query_range)
    for movie in movies:
        print(f"\n{movie['year']} : {movie['title']}")
        pprint(movie['info'])

```

2. To run the program, enter the following command.

```
python MoviesQuery02.py
```

Step 4.3: Scan

The `scan` method reads every item in the entire table and returns all the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, the filter is applied only after the entire table has been scanned.

The following program scans the entire `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items) and discard all the others.

1. Copy the following program and paste it into a file named `MoviesScan.py`.

```

from pprint import pprint
import boto3
from boto3.dynamodb.conditions import Key

def scan_movies(year_range, display_movies, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")

    table = dynamodb.Table('Movies')
    scan_kwargs = {
        'FilterExpression': Key('year').between(*year_range),
        'ProjectionExpression': "#yr, title, info.rating",
        'ExpressionAttributeNames': {"#yr": "year"}
    }

```

```

done = False
start_key = None
while not done:
    if start_key:
        scan_kwargs['ExclusiveStartKey'] = start_key
    response = table.scan(**scan_kwargs)
    display_movies(response.get('Items', []))
    start_key = response.get('LastEvaluatedKey', None)
    done = start_key is None

if __name__ == '__main__':
    def print_movies(movies):
        for movie in movies:
            print(f"\n{movie['year']} : {movie['title']}")
            pprint(movie['info'])

    query_range = (1950, 1959)
    print(f"Scanning for movies released from {query_range[0]} to {query_range[1]}...")
    scan_movies(query_range, print_movies)

```

In the code, note the following:

- `ProjectionExpression` specifies the attributes you want in the scan result.
- `FilterExpression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
- The `scan` method returns a subset of the items each time, called a *page*. The `LastEvaluatedKey` value in the response is then passed to the `scan` method via the `ExclusiveStartKey` parameter. When the last page is returned, `LastEvaluatedKey` is not part of the response.

Note

- `ExpressionAttributeNames` provides name substitution. We use this because `year` is a reserved word in DynamoDB—you can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
- `ExpressionAttributeValues` provides value substitution. You use this because you can't use literals in any expression, including `KeyConditionExpression`. You can use the expression attribute value `:yyyy` to address this.

2. To run the program, enter the following command.

```
python MoviesScan.py
```

Note

You can also use the `Scan` operation with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: (Optional) Delete the Table

To delete the `Movies` table:

1. Copy the following program and paste it into a file named `MoviesDeleteTable.py`.

```

import boto3

def delete_movie_table(dynamodb=None):

```

```
if not dynamodb:  
    dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")  
  
table = dynamodb.Table('Movies')  
table.delete()  
  
if __name__ == '__main__':  
    delete_movie_table()  
    print("Movies table deleted.")
```

2. To run the program, enter the following command.

```
python MoviesDeleteTable.py
```

Summary

In this tutorial, you created the `Movies` table in the downloadable version of Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB web service, you must change the endpoint in your application. To do this, modify the following line.

```
dynamodb = boto3.resource('dynamodb', endpoint_url="http://localhost:8000")
```

For example, if you want to use the `us-west-2` Region, change the code to the following.

```
dynamodb = boto3.resource('dynamodb', region_name='us-west-2')
```

Instead of using the downloadable version of DynamoDB on your computer, the program now uses the DynamoDB web service in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information about setting Regions and endpoints in your code, see [AWS Region Selection](#) in the [AWS SDK for Java Developer Guide](#).

Ruby and DynamoDB

In this tutorial, you use the AWS SDK for Ruby to write simple programs to perform the following Amazon DynamoDB operations:

- Create a table called `Movies` and load sample data in JSON format.
- Perform create, read, update, and delete operations on the table.
- Run simple queries.

As you work through this tutorial, you can refer to the [AWS SDK for Ruby API Reference](#). The [DynamoDB section](#) describes the parameters and results for DynamoDB operations.

Tutorial Prerequisites

- Download and run DynamoDB on your computer. For more information, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

Note

You use the downloadable version of DynamoDB in this tutorial. For information about how to run the same code against the DynamoDB web service, see the [Summary \(p. 208\)](#).

- Set up an AWS access key to use the AWS SDKs. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).
- Set up the AWS SDK for Ruby:
 - Install [Ruby](#).
 - Install the [AWS SDK for Ruby](#).

For more information, see [Installation in the AWS SDK for Ruby API Reference](#).

Step 1: Create a Table

In this step, you create a table named `Movies` in Amazon DynamoDB. The primary key for the table is composed of the following two attributes:

- `year` – The partition key. The `attribute_type` is `N` for number.
- `title` – The sort key. The `attribute_type` is `S` for string.

1. Copy the following program and paste it into a file named `MoviesCreateTable.rb`.

```
#  
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
#  
# This file is licensed under the Apache License, Version 2.0 (the "License").  
# You may not use this file except in compliance with the License. A copy of  
# the License is located at  
#  
# http://aws.amazon.com/apache2.0/  
#  
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
# CONDITIONS OF ANY KIND, either express or implied. See the License for the  
# specific language governing permissions and limitations under the License.  
#  
require "aws-sdk"  
  
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})  
  
dynamodb = Aws::DynamoDB::Client.new  
  
params = {  
    table_name: "Movies",  
    key_schema: [  
        {  
            attribute_name: "year",  
            key_type: "HASH" #Partition key  
        },  
        {  
            attribute_name: "title",  
            key_type: "range" #Sort key  
        }  
    ]  
}  
  
response = dynamodb.create_table(params)  
  
puts response.table_description
```

```
        key_type: "RANGE" #Sort key
    }
],
attribute_definitions: [
{
    attribute_name: "year",
    attribute_type: "N"
},
{
    attribute_name: "title",
    attribute_type: "S"
}

],
provisioned_throughput: {
    read_capacity_units: 10,
    write_capacity_units: 10
}
}

begin
    result = dynamodb.create_table(params)
    puts "Created table. Status: " +
        result.table_description.table_status;

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to create table:"
    puts "#{error.message}"
end
```

Note

- You set the endpoint to indicate that you are creating the table in the downloadable version of Amazon DynamoDB on your computer.
 - In the `create_table` call, you specify table name, primary key attributes, and its data types.
 - The `provisioned_throughput` parameter is required. However, the downloadable version of DynamoDB ignores it. (Provisioned throughput is beyond the scope of this exercise.)
2. To run the program, enter the following command.

```
ruby MoviesCreateTable.rb
```

To learn more about managing tables, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Step 2: Load Sample Data

In this step, you populate the `Movies` table in Amazon DynamoDB with sample data.

Topics

- [Step 2.1: Download the Sample Data File \(p. 194\)](#)
- [Step 2.2: Load the Sample Data into the Movies Table \(p. 195\)](#)

You use a sample data file that contains information about a few thousand movies from the Internet Movie Database (IMDb). The movie data is in JSON format, as shown in the following example. For each movie, there is a `year`, a `title`, and a JSON map named `info`.

```
[
```

```
{  
    "year" : ... ,  
    "title" : ... ,  
    "info" : { ... }  
},  
{  
    "year" : ...,  
    "title" : ....,  
    "info" : { ... }  
},  
...  
]
```

In the JSON data, note the following:

- The `year` and `title` are used as the primary key attribute values for the `Movies` table.
- You store the rest of the `info` values in a single attribute called `info`. This program illustrates how you can store JSON in a DynamoDB attribute.

The following is an example of movie data.

```
{  
    "year" : 2013,  
    "title" : "Turn It Down, Or Else!",  
    "info" : {  
        "directors" : [  
            "Alice Smith",  
            "Bob Jones"  
        ],  
        "release_date" : "2013-01-18T00:00:00Z",  
        "rating" : 6.2,  
        "genres" : [  
            "Comedy",  
            "Drama"  
        ],  
        "image_url" : "http://ia.media-imdb.com/images/N/  
09ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@._V1_SX400_.jpg",  
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",  
        "rank" : 11,  
        "running_time_secs" : 5215,  
        "actors" : [  
            "David Matthewman",  
            "Ann Thomas",  
            "Jonathan G. Neff"  
        ]  
    }  
}
```

Step 2.1: Download the Sample Data File

1. Download the sample data archive: [moviedata.zip](#)
2. Extract the data file (`moviedata.json`) from the archive.
3. Copy the `moviedata.json` file and paste it into your current directory.

Step 2.2: Load the Sample Data into the Movies Table

After you download the sample data, run the following program to populate the `Movies` table.

1. Copy the following program and paste it into a file named `MoviesLoadData.rb`.

```
#  
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
#  
# This file is licensed under the Apache License, Version 2.0 (the "License").  
# You may not use this file except in compliance with the License. A copy of  
# the License is located at  
#  
# http://aws.amazon.com/apache2.0/  
#  
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
# CONDITIONS OF ANY KIND, either express or implied. See the License for the  
# specific language governing permissions and limitations under the License.  
#  
require "aws-sdk"  
require "json"  
  
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})  
  
dynamodb = Aws::DynamoDB::Client.new  
  
table_name = 'Movies'  
  
file = File.read('moviedata.json')  
movies = JSON.parse(file)  
movies.each{|movie|  
  
    params = {  
        table_name: table_name,  
        item: movie  
    }  
  
    begin  
        dynamodb.put_item(params)  
        puts "Added movie: #{movie["year"]} #{movie["title"]}"  
  
    rescue Aws::DynamoDB::Errors::ServiceError => error  
        puts "Unable to add movie:"  
        puts "#{error.message}"  
    end  
}
```

2. To run the program, enter the following command.

```
ruby MoviesLoadData.rb
```

Step 3: Create, Read, Update, and Delete an Item

In this step, you perform read and write operations on an item in the `Movies` table in Amazon DynamoDB.

To learn more about reading and writing data, see [Working with Items and Attributes \(p. 374\)](#).

Topics

- [Step 3.1: Create a New Item \(p. 196\)](#)
- [Step 3.2: Read an Item \(p. 197\)](#)
- [Step 3.3: Update an Item \(p. 198\)](#)
- [Step 3.4: Increment an Atomic Counter \(p. 200\)](#)
- [Step 3.5: Update an Item \(Conditionally\) \(p. 201\)](#)
- [Step 3.6: Delete an Item \(p. 202\)](#)

Step 3.1: Create a New Item

In this step, you add a new item to the table.

1. Copy the following program and paste it into a file named `MoviesItemOps01.rb`.

```
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = 'Movies'

year = 2015
title = "The Big New Movie"

item = {
  year: year,
  title: title,
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}

params = {
  table_name: table_name,
  item: item
}

begin
  dynamodb.put_item(params)
  puts "Added item: #{year} - #{title}"
end
```

```
rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to add item:"
    puts "#{$error.message}"
end
```

Note

The primary key is required. This code adds an item that has primary key (`year, title`) and `info` attributes. The `info` attribute stores a map that provides more information about the movie.

2. To run the program, enter the following command.

```
ruby MoviesItemOps01.rb
```

Step 3.2: Read an Item

In the previous program, you added the following item to the table.

```
{
  year: 2015,
  title: "The Big New Movie",
  info: {
    plot: "Nothing happens at all.",
    rating: 0
  }
}
```

You can use the `get_item` method to read the item from the `Movies` table. You must specify the primary key values so that you can read any item from `Movies` if you know its `year` and `title`.

1. Copy the following program and paste it into a file named `MoviesItemOps02.rb`.

```
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
```

```

    table_name: table_name,
    key: {
        year: year,
        title: title
    }
}

begin
    result = dynamodb.get_item(params)
    printf "%i - %s\n%s\n%d\n",
        result.item["year"],
        result.item["title"],
        result.item["info"]["plot"],
        result.item["info"]["rating"]

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to read item:"
    puts "#{$error.message}"
end

```

2. To run the program, enter the following command.

```
ruby MoviesItemOps02.rb
```

Step 3.3: Update an Item

You can use the `update_item` method to modify an existing item. You can update values of existing attributes, add new attributes, or remove attributes.

In this example, you perform the following updates:

- Change the value of the existing attributes (`rating`, `plot`).
- Add a new list attribute (`actors`) to the existing `info` map.

The following shows the existing item.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Nothing happens at all.",
        rating: 0
    }
}
```

The item is updated as follows.

```
{
    year: 2015,
    title: "The Big New Movie",
    info: {
        plot: "Everything happens all at once.",
        rating: 5.5,
        actors: ["Larry", "Moe", "Curly"]
    }
}
```

1. Copy the following program and paste it into a file named `MoviesItemOps03.rb`.

```

#
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
  table_name: table_name,
  key: {
    year: year,
    title: title
  },
  update_expression: "set info.rating = :r, info.plot=:p, info.actors=:a",
  expression_attribute_values: {
    ":r" => 5.5,
    ":p" => "Everything happens all at once.", # value
<Hash,Array,String,Numeric,Boolean,IO,Set,nil>
    ":a" => ["Larry", "Moe", "Curly"]
  },
  return_values: "UPDATED_NEW"
}

begin
  dynamodb.update_item(params)
  puts "Added item: #{year} - #{title}"

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to add item:"
  puts "#{@error.message}"
end

```

Note

This program uses `update_expression` to describe all updates you want to perform on the specified item.

The `return_values` parameter instructs Amazon DynamoDB to return only the updated attributes (`UPDATED_NEW`).

2. To run the program, enter the following command.

```
ruby MoviesItemOps03.rb
```

Step 3.4: Increment an Atomic Counter

DynamoDB supports atomic counters, which use the `update_item` method to increment or decrement the value of an existing attribute without interfering with other write requests. (All write requests are applied in the order in which they are received.)

The following program shows how to increment the `rating` for a movie. Each time you run the program, it increments this attribute by one.

1. Copy the following program and paste it into a file named `MoviesItemOps04.rb`.

```
#  
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
#  
# This file is licensed under the Apache License, Version 2.0 (the "License").  
# You may not use this file except in compliance with the License. A copy of  
# the License is located at  
#  
# http://aws.amazon.com/apache2.0/  
#  
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
# CONDITIONS OF ANY KIND, either express or implied. See the License for the  
# specific language governing permissions and limitations under the License.  
#  
require "aws-sdk"  
  
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})  
  
dynamodb = Aws::DynamoDB::Client.new  
  
table_name = 'Movies'  
  
year = 2015  
title = "The Big New Movie"  
  
params = {  
    table_name: table_name,  
    key: {  
        year: year,  
        title: title  
    },  
    update_expression: "set info.rating = info.rating + :val",  
    expression_attribute_values: {  
        ":val" => 1  
    },  
    return_values: "UPDATED_NEW"  
}  
  
begin  
    result = dynamodb.update_item(params)  
    puts "Updated item. ReturnValues are:"  
    result.attributes["info"].each do |key, value|  
        if key == "rating"  
            puts "#{key}: #{value.to_f}"  
        else  
            puts "#{key}: #{value}"  
        end  
    end  
rescue Aws::DynamoDB::Errors::ServiceError => error  
    puts "Unable to update item:"
```

```
    puts "#{error.message}"
end
```

2. To run the program, enter the following command.

```
ruby MoviesItemOps04.rb
```

Step 3.5: Update an Item (Conditionally)

The following program shows how to use `update_item` with a condition. If the condition evaluates to true, the update succeeds; otherwise, the update is not performed.

In this case, the item is updated only if the number of actors is greater than three.

1. Copy the following program and paste it into a file named `MoviesItemOps05.rb`.

```
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
  table_name: table_name,
  key: {
    year: year,
    title: title
  },
  update_expression: "remove info.actors[0]",
  condition_expression: "size(info.actors) > :num",
  expression_attribute_values: {
    ":num" => 3
  },
  return_values: "UPDATED_NEW"
}

begin
  result = dynamodb.update_item(params)
  puts "Updated item. ReturnValues are:"
  result.attributes["info"].each do |key, value|
    if key == "rating"
```

```

        puts "#{key}: #{value.to_f}"
    else
        puts "#{key}: #{value}"
    end
end

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{@error.message}"
end

```

2. To run the program, enter the following command.

```
ruby MoviesItemOps05.rb
```

The program should fail with the following message.

The conditional request failed

The program fails because the movie has three actors in it, but the condition is checking for *greater than* three actors.

3. Modify the program so that the `ConditionExpression` looks like the following.

```
condition_expression: "size(info.actors) >= :num",
```

The condition is now *greater than or equal to 3* instead of *greater than 3*.

4. Run the program again. The `update_item` method should now succeed.

Step 3.6: Delete an Item

You can use the `delete_item` method to delete one item by specifying its primary key. You can optionally provide a `condition_expression` to prevent the item from being deleted if the condition is not met.

In the following example, you try to delete a specific movie item if its rating is 5 or less.

1. Copy the following program and paste it into a file named `MoviesItemOps06.rb`.

```

#
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

```

```
table_name = 'Movies'

year = 2015
title = "The Big New Movie"

params = {
    table_name: table_name,
    key: {
        year: year,
        title: title
    },
    condition_expression: "info.rating <= :val",
    expression_attribute_values: {
        ":val" => 5
    }
}

begin
    dynamodb.delete_item(params)
    puts "Deleted item."

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to update item:"
    puts "#{@error.message}"
end
```

2. To run the program, enter the following command.

```
ruby MoviesItemOps06.rb
```

The program should fail with the following message.

The conditional request failed

The program fails because the rating for this particular move is greater than 5.

3. Modify the program to remove the condition.

```
params = {
    table_name: "Movies",
    key: {
        year: year,
        title: title
    }
}
```

4. Run the program. Now, the delete succeeds because you removed the condition.

Step 4: Query and Scan the Data

You can use the `query` method to retrieve data from a table. You must specify a partition key value. The sort key is optional.

The primary key for the `Movies` table is composed of the following:

- `year` – The partition key. The attribute type is `number`.
- `title` – The sort key. The attribute type is `string`.

To find all movies released during a year, you need to specify only the `year`. You can also provide the `title` to retrieve a subset of movies based on some condition (on the sort key). For example, you can find movies released in 2014 that have a title starting with the letter "A".

In addition to the `query` method, you can use the `scan` method to retrieve all of the table data.

To learn more about querying and scanning data, see [Working with Queries in DynamoDB \(p. 458\)](#) and [Working with Scans in DynamoDB \(p. 475\)](#), respectively.

Topics

- [Step 4.1: Query - All Movies Released in a Year \(p. 204\)](#)
- [Step 4.2: Query - All Movies Released in a Year with Certain Titles \(p. 205\)](#)
- [Step 4.3: Scan \(p. 206\)](#)

Step 4.1: Query - All Movies Released in a Year

The following program retrieves all movies released in the year 1985.

1. Copy the following program and paste it into a file named `MoviesQuery01.rb`.

```
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = "Movies"

params = {
  table_name: table_name,
  key_condition_expression: "#yr = :yyyy",
  expression_attribute_names: {
    "#yr" => "year"
  },
  expression_attribute_values: {
    ":yyyy" => 1985
  }
}

puts "Querying for movies from 1985."

begin
  result = dynamodb.query(params)
  puts "Query succeeded."
```

```
    result.items.each{|movie|
        puts "#{movie["year"].to_i} #{movie["title"]}"
    }

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to query table:"
    puts "#{error.message}"
end
```

Note

- `expression_attribute_names` provides name substitution. We use this because `year` is a reserved word in Amazon DynamoDB. You can't use it directly in any expression, including `KeyConditionExpression`. You can use the expression attribute name `#yr` to address this.
- `expression_attribute_values` provides value substitution. You use this because you can't use literals in any expression, including `key_condition_expression`. You can use the expression attribute value `:yyyy` to address this.

2. To run the program, enter the following command.

```
ruby MoviesItemQuery01.rb
```

Note

The preceding program shows how to query a table by its primary key attributes. In DynamoDB, you can optionally create one or more secondary indexes on a table and query those indexes in the same way that you query a table. Secondary indexes give your applications additional flexibility by allowing queries on non-key attributes. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 4.2: Query - All Movies Released in a Year with Certain Titles

The following program retrieves all movies released in year 1992 with a title beginning with the letter "A" through the letter "L".

1. Copy the following program and paste it into a file named `MoviesQuery02.rb`.

```
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})
```

```

dynamodb = Aws::DynamoDB::Client.new

table_name = "Movies"

params = {
    table_name: table_name,
    projection_expression: "#yr, title, info.genres, info.actors[0]",
    key_condition_expression:
        "#yr = :yyyy and title between :letter1 and :letter2",
    expression_attribute_names: {
        "#yr" => "year"
    },
    expression_attribute_values: {
        ":yyyy" => 1992,
        ":letter1" => "A",
        ":letter2" => "L"
    }
}

puts "Querying for movies from 1992 - titles A-L, with genres and lead actor";

begin
    result = dynamodb.query(params)
    puts "Query succeeded."

    result.items.each{|movie|
        print "#{movie["year"]}.to_i}: #{movie["title"]} ... "

        movie['info']['genres'].each{|gen|
            print gen + " "
        }

        print "... #{movie["info"]["actors"][0]}\n"
    }

rescue Aws::DynamoDB::Errors::ServiceError => error
    puts "Unable to query table:"
    puts "#{error.message}"
end

```

2. To run the program, enter the following command.

```
ruby MoviesQuery02.rb
```

Step 4.3: Scan

The `scan` method reads every item in the entire table and returns all the data in the table. You can provide an optional `filter_expression` so that only the items matching your criteria are returned. However, note that the filter is only applied after the entire table has been scanned.

The following program scans the `Movies` table, which contains approximately 5,000 items. The scan specifies the optional filter to retrieve only the movies from the 1950s (approximately 100 items) and discard all the others.

1. Copy the following program and paste it into a file named `MoviesScan.rb`.

```

#
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of

```

```

# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
# CONDITIONS OF ANY KIND, either express or implied. See the License for the
# specific language governing permissions and limitations under the License.
#
require "aws-sdk"

Aws.config.update({
  region: "us-west-2",
  endpoint: "http://localhost:8000"
})

dynamodb = Aws::DynamoDB::Client.new

table_name = "Movies"

params = {
  table_name: table_name,
  projection_expression: "#yr, title, info.rating",
  filter_expression: "#yr between :start_yr and :end_yr",
  expression_attribute_names: {"#yr"=> "year"},
  expression_attribute_values: {
    ":start_yr" => 1950,
    ":end_yr" => 1959
  }
}

puts "Scanning Movies table."

begin
  loop do
    result = dynamodb.scan(params)

    result.items.each{|movie|
      puts "#{movie["year"].to_i}: " +
        "#{movie["title"]} ... " +
        "#{movie["info"]["rating"].to_f}"
    }

    break if result.last_evaluated_key.nil?

    puts "Scanning for more..."
    params[:exclusive_start_key] = result.last_evaluated_key
  end

rescue Aws::DynamoDB::Errors::ServiceError => error
  puts "Unable to scan:"
  puts "#{error.message}"
end

```

In the code, note the following:

- `projection_expression` specifies the attributes you want in the scan result.
 - `filter_expression` specifies a condition that returns only items that satisfy the condition. All other items are discarded.
2. To run the program, enter the following command.

```
ruby MoviesScan.rb
```

Note

You can also use the scan method with any secondary indexes that you create on the table. For more information, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

Step 5: (Optional) Delete the Table

Follow these steps to delete the Movies table.

1. Copy the following program and paste it into a file named `MoviesDeleteTable.rb`.

```
#  
# Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
#  
# This file is licensed under the Apache License, Version 2.0 (the "License").  
# You may not use this file except in compliance with the License. A copy of  
# the License is located at  
#  
# http://aws.amazon.com/apache2.0/  
#  
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
# CONDITIONS OF ANY KIND, either express or implied. See the License for the  
# specific language governing permissions and limitations under the License.  
#  
require "aws-sdk"  
  
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})  
  
dynamodb = Aws::DynamoDB::Client.new  
  
params = {  
    table_name: "Movies"  
}  
  
begin  
    dynamodb.delete_table(params)  
    puts "Deleted table."  
  
rescue Aws::DynamoDB::Errors::ServiceError => error  
    puts "Unable to delete table:"  
    puts "#{error.message}"  
end
```

2. To run the program, enter the following command.

```
ruby MoviesDeleteTable.rb
```

Summary

In this tutorial, you created the `Movies` table in the downloadable version of Amazon DynamoDB on your computer and performed basic operations. The downloadable version of DynamoDB is useful during application development and testing. However, when you're ready to run your application in a production environment, you must modify your code so that it uses the DynamoDB web service.

Modifying the Code to Use the DynamoDB Service

To use the DynamoDB service, you must change the endpoint in your application. To do this, find the following lines in the code.

```
Aws.config.update({  
    region: "us-west-2",  
    endpoint: "http://localhost:8000"  
})
```

Remove the `endpoint` parameter so that the code looks like the following.

```
Aws.config.update({  
    region: "us-west-2"  
});
```

After you remove this line, the code can access the DynamoDB service in the AWS Region specified by the `region` config value.

Instead of using the version of DynamoDB on your computer, the program uses the DynamoDB web service endpoint in the US West (Oregon) Region.

DynamoDB is available in AWS Regions worldwide. For the complete list, see [Regions and Endpoints](#) in the [AWS General Reference](#). For more information, see the [AWS SDK for Ruby Getting Started Guide](#).

Programming with DynamoDB and the AWS SDKs

This section covers developer-related topics. If you want to run code examples instead, see [Running the Code Examples in This Developer Guide \(p. 324\)](#).

Note

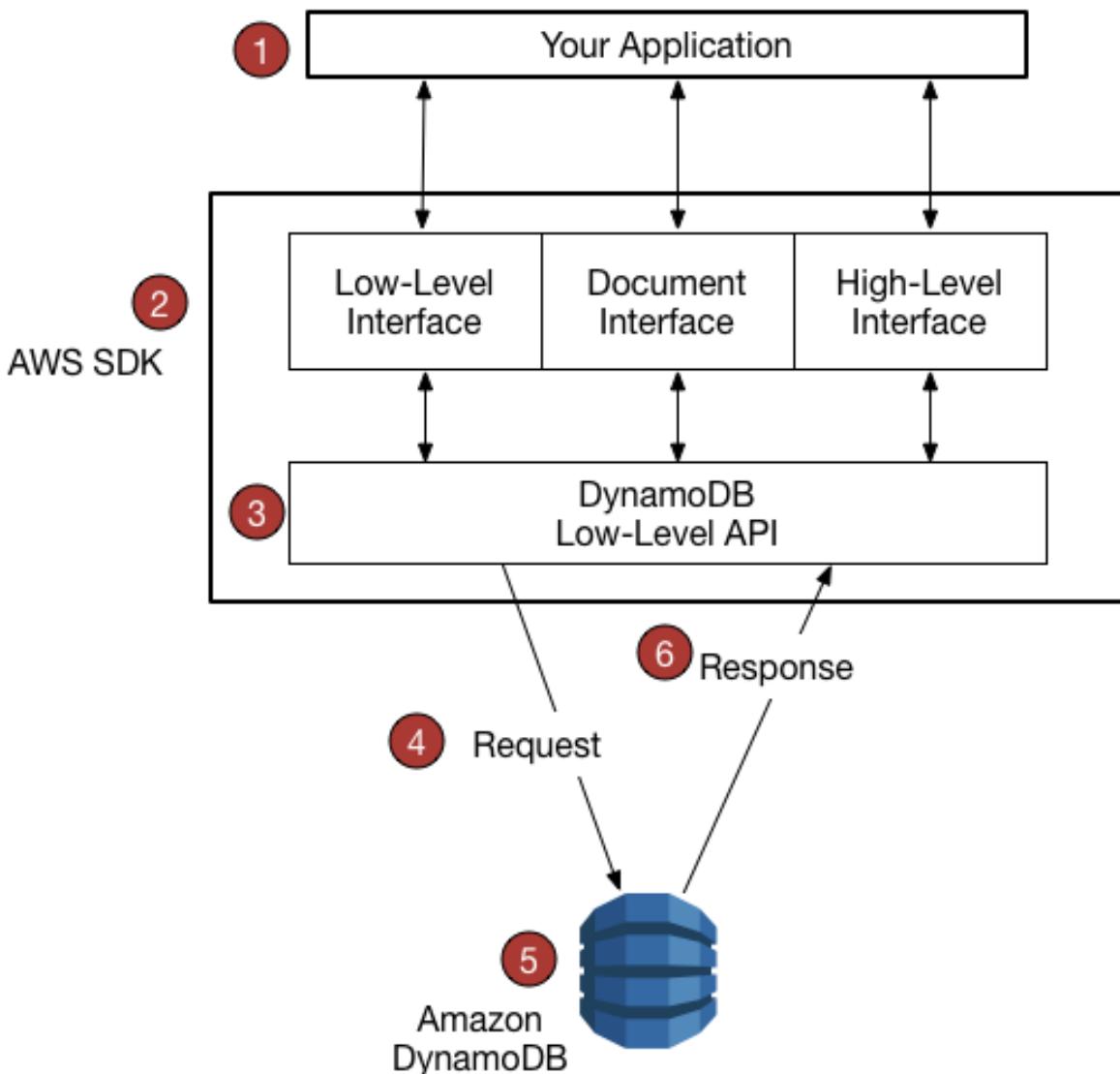
In December 2017, AWS began the process of migrating all Amazon DynamoDB endpoints to use secure certificates issued by Amazon Trust Services (ATS). For more information, see [Troubleshooting SSL/TLS connection establishment issues \(p. 971\)](#).

Topics

- [Overview of AWS SDK Support for DynamoDB \(p. 210\)](#)
- [Programmatic Interfaces \(p. 212\)](#)
- [DynamoDB Low-Level API \(p. 216\)](#)
- [Error Handling with DynamoDB \(p. 219\)](#)
- [Higher-Level Programming Interfaces for DynamoDB \(p. 225\)](#)
- [Running the Code Examples in This Developer Guide \(p. 324\)](#)

Overview of AWS SDK Support for DynamoDB

The following diagram provides a high-level overview of Amazon DynamoDB application programming using the AWS SDKs.



1. You write an application using an AWS SDK for your programming language.
2. Each AWS SDK provides one or more programmatic interfaces for working with DynamoDB. The specific interfaces available depend on which programming language and AWS SDK you use.
3. The AWS SDK constructs HTTP(S) requests for use with the low-level DynamoDB API.
4. The AWS SDK sends the request to the DynamoDB endpoint.
5. DynamoDB executes the request. If the request is successful, DynamoDB returns an HTTP 200 response code (OK). If the request is unsuccessful, DynamoDB returns an HTTP error code and an error message.
6. The AWS SDK processes the response and propagates it back to your application.

Each of the AWS SDKs provides important services to your application, including the following:

- Formatting HTTP(S) requests and serializing request parameters.
- Generating a cryptographic signature for each request.
- Forwarding requests to a DynamoDB endpoint and receiving responses from DynamoDB.

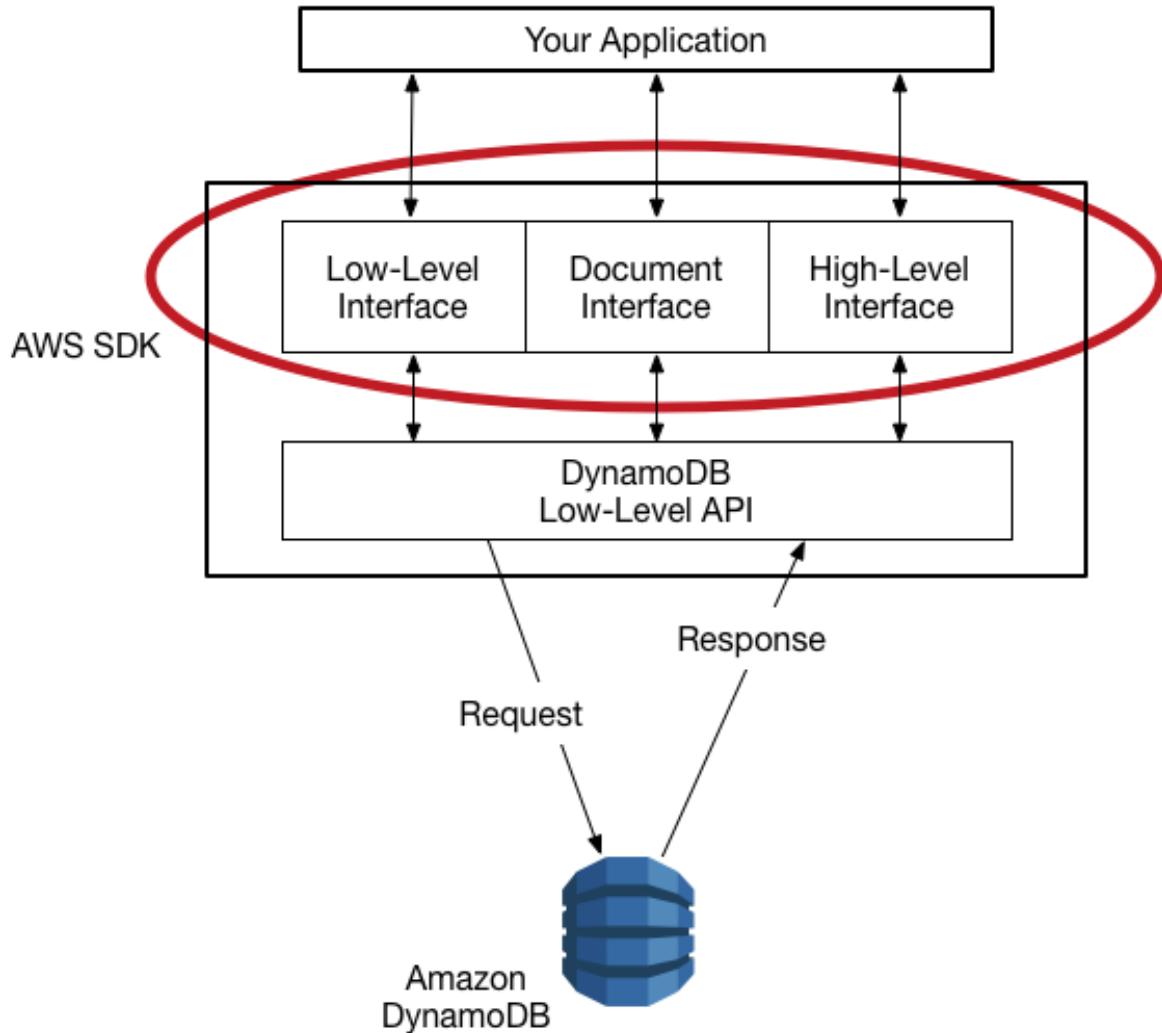
- Extracting the results from those responses.
- Implementing basic retry logic in case of errors.

You do not need to write code for any of these tasks.

Note

For more information about AWS SDKs, including installation instructions and documentation, see [Tools for Amazon Web Services](#).

Programmatic Interfaces



Every [AWS SDK](#) provides one or more programmatic interfaces for working with Amazon DynamoDB. These interfaces range from simple low-level DynamoDB wrappers to object-oriented persistence layers. The available interfaces vary depending on the AWS SDK and programming language that you use.

The following section highlights some of the interfaces available, using the AWS SDK for Java as an example. (Not all interfaces are available in all AWS SDKs.)

Topics

- [Low-Level Interfaces \(p. 213\)](#)
- [Document Interfaces \(p. 214\)](#)
- [Object Persistence Interface \(p. 214\)](#)

Low-Level Interfaces

Every language-specific AWS SDK provides a low-level interface for Amazon DynamoDB, with methods that closely resemble low-level DynamoDB API requests.

In some cases, you will need to identify the data types of the attributes using [Data Type Descriptors \(p. 218\)](#), such as S for string or N for number.

Note

A low-level interface is available in every language-specific AWS SDK.

The following Java program uses the low-level interface of the AWS SDK for Java. The program issues a `GetItem` request for a song in the `Music` table and prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.AmazonDynamoDB` class implements the DynamoDB low-level interface.

```
package com.amazonaws.codesamples;

import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class MusicLowLevelDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Call Me Today"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music")
            .withKey(key);

        try {
            GetItemResult result = client.getItem(request);
            if (result && result.getItem() != null) {
                AttributeValue year = result.getItem().get("Year");
                System.out.println("The song was released in " + year.getN());
            } else {
                System.out.println("No matching song was found");
            }
        } catch (Exception e) {
            System.err.println("Unable to retrieve data: ");
            System.err.println(e.getMessage());
        }
    }
}
```

Document Interfaces

Many AWS SDKs provide a document interface, allowing you to perform data plane operations (create, read, update, delete) on tables and indexes. With a document interface, you do not need to specify [Data Type Descriptors \(p. 218\)](#). The data types are implied by the semantics of the data itself. These AWS SDKs also provide methods to easily convert JSON documents to and from native Amazon DynamoDB data types.

Note

Document interfaces are available in the AWS SDKs for [Java](#), [.NET](#), [Node.js](#), and [JavaScript in the browser](#).

The following Java program uses the document interface of the AWS SDK for Java. The program creates a `Table` object that represents the `Music` table, and then asks that object to use `GetItem` to retrieve a song. The program then prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.document.DynamoDB` class implements the DynamoDB document interface. Note how DynamoDB acts as a wrapper around the low-level client (`AmazonDynamoDB`).

```
package com.amazonaws.codesamples.gsg;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class MusicDocumentDemo {

    public static void main(String[] args) {

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");
        GetItemOutcome outcome = table.getItemOutcome(
            "Artist", "No One You Know",
            "SongTitle", "Call Me Today");

        int year = outcome.getItem().getInt("Year");
        System.out.println("The song was released in " + year);
    }
}
```

Object Persistence Interface

Some AWS SDKs provide an object persistence interface where you do not directly perform data plane operations. Instead, you create objects that represent items in Amazon DynamoDB tables and indexes, and interact only with those objects. This allows you to write object-centric code, rather than database-centric code.

Note

Object persistence interfaces are available in the AWS SDKs for Java and .NET. For more information, see [Higher-Level Programming Interfaces for DynamoDB \(p. 225\)](#).

The following Java program uses `DynamoDBMapper`, the object persistence interface of the AWS SDK for Java. The `MusicItem` class represents an item in the `Music` table.

```
package com.amazonaws.codesamples;
```

```

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="Music")
public class MusicItem {
    private String artist;
    private String songTitle;
    private String albumTitle;
    private int year;

    @DynamoDBHashKey(attributeName="Artist")
    public String getArtist() { return artist; }
    public void setArtist(String artist) {this.artist = artist; }

    @DynamoDBRangeKey(attributeName="SongTitle")
    public String getSongTitle() { return songTitle; }
    public void setSongTitle(String songTitle) {this.songTitle = songTitle; }

    @DynamoDBAttribute(attributeName = "AlbumTitle")
    public String getAlbumTitle() { return albumTitle; }
    public void setAlbumTitle(String albumTitle) {this.albumTitle = albumTitle; }

    @DynamoDBAttribute(attributeName = "Year")
    public int getYear() { return year; }
    public void setYear(int year) { this.year = year; }
}

```

You can then instantiate a `MusicItem` object, and retrieve a song using the `load()` method of `DynamoDBMapper`. The program then prints the year that the song was released.

The `com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper` class implements the DynamoDB object persistence interface. Note how `DynamoDBMapper` acts as a wrapper around the low-level client (`AmazonDynamoDB`).

```

package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;

public class MusicMapperDemo {

    public static void main(String[] args) {
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
        DynamoDBMapper mapper = new DynamoDBMapper(client);

        MusicItem keySchema = new MusicItem();
        keySchema.setArtist("No One You Know");
        keySchema.setSongTitle("Call Me Today");

        try {
            MusicItem result = mapper.load(keySchema);
            if (result != null) {
                System.out.println(
                    "The song was released in "+ result.getYear());
            } else {
                System.out.println("No matching song was found");
            }
        }
    }
}

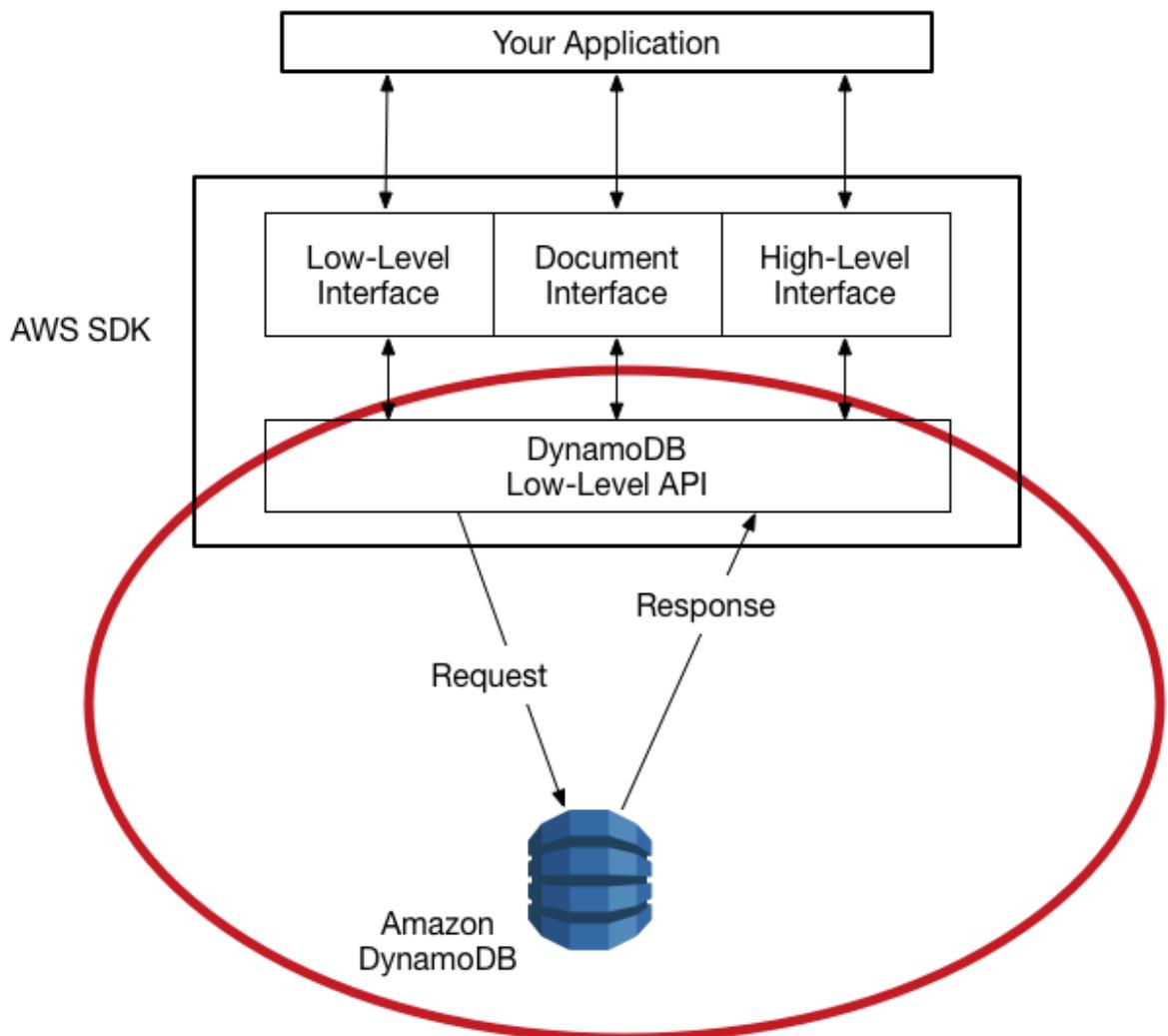
```

```
        } catch (Exception e) {
            System.err.println("Unable to retrieve data: ");
            System.err.println(e.getMessage());
        }
    }
}
```

DynamoDB Low-Level API

Topics

- [Request Format \(p. 217\)](#)
- [Response Format \(p. 218\)](#)
- [Data Type Descriptors \(p. 218\)](#)
- [Numeric Data \(p. 219\)](#)
- [Binary Data \(p. 219\)](#)



The Amazon DynamoDB *low-level API* is the protocol-level interface for DynamoDB. At this level, every HTTP(S) request must be correctly formatted and carry a valid digital signature.

The AWS SDKs construct low-level DynamoDB API requests on your behalf and process the responses from DynamoDB. This lets you focus on your application logic, instead of low-level details. However, you can still benefit from a basic knowledge of how the low-level DynamoDB API works.

For more information about the low-level DynamoDB API, see [Amazon DynamoDB API Reference](#).

Note

DynamoDB Streams has its own low-level API, which is separate from that of DynamoDB and is fully supported by the AWS SDKs.

For more information, see [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#). For the low-level DynamoDB Streams API, see the [Amazon DynamoDB Streams API Reference](#).

The low-level DynamoDB API uses JavaScript Object Notation (JSON) as a wire protocol format. JSON presents data in a hierarchy so that both data values and data structure are conveyed simultaneously. Name-value pairs are defined in the format `name : value`. The data hierarchy is defined by nested brackets of name-value pairs.

DynamoDB uses JSON only as a transport protocol, not as a storage format. The AWS SDKs use JSON to send data to DynamoDB, and DynamoDB responds with JSON. DynamoDB does not store data persistently in JSON format.

Note

For more information about JSON, see [Introducing JSON](#) on the [JSON.org](#) website.

Request Format

The DynamoDB low-level API accepts HTTP(S) POST requests as input. The AWS SDKs construct these requests for you.

Suppose that you have a table named `Pets`, with a key schema consisting of `AnimalType` (partition key) and `Name` (sort key). Both of these attributes are of type `string`. To retrieve an item from `Pets`, the AWS SDK constructs the following request.

```
POST / HTTP/1.1
Host: dynamodb.<region>.<domain>;
Accept-Encoding: identity
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.0
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
Signature=<Signature>
X-Amz-Date: <Date>
X-Amz-Target: DynamoDB_20120810.GetItem

{
    "TableName": "Pets",
    "Key": {
        "AnimalType": {"S": "Dog"},
        "Name": {"S": "Fido"}
    }
}
```

Note the following about this request:

- The `Authorization` header contains information required for DynamoDB to authenticate the request. For more information, see [Signing AWS API Requests](#) and [Signature Version 4 Signing Process](#) in the [Amazon Web Services General Reference](#).

- The `X-Amz-Target` header contains the name of a DynamoDB operation: `GetItem`. (This is also accompanied by the low-level API version, in this case `20120810`.)
- The payload (body) of the request contains the parameters for the operation, in JSON format. For the `GetItem` operation, the parameters are `TableName` and `Key`.

Response Format

Upon receipt of the request, DynamoDB processes it and returns a response. For the request shown previously, the HTTP(S) response payload contains the results from the operation, as shown in the following example.

```
HTTP/1.1 200 OK
x-amzn-RequestId: <RequestId>
x-amz-crc32: <Checksum>
Content-Type: application/x-amz-json-1.0
Content-Length: <PayloadSizeBytes>
Date: <Date>
{
    "Item": {
        "Age": {"N": "8"},
        "Colors": {
            "L": [
                {"S": "White"},
                {"S": "Brown"},
                {"S": "Black"}
            ]
        },
        "Name": {"S": "Fido"},
        "Vaccinations": {
            "M": {
                "Rabies": {
                    "L": [
                        {"S": "2009-03-17"},
                        {"S": "2011-09-21"},
                        {"S": "2014-07-08"}
                    ]
                },
                "Distemper": {"S": "2015-10-13"}
            }
        },
        "Breed": {"S": "Beagle"},
        "AnimalType": {"S": "Dog"}
    }
}
```

At this point, the AWS SDK returns the response data to your application for further processing.

Note

If DynamoDB can't process a request, it returns an HTTP error code and message. The AWS SDK propagates these to your application in the form of exceptions. For more information, see [Error Handling with DynamoDB \(p. 219\)](#).

Data Type Descriptors

The low-level DynamoDB API protocol requires each attribute to be accompanied by a data type descriptor. *Data type descriptors* are tokens that tell DynamoDB how to interpret each attribute.

The examples in [Request Format \(p. 217\)](#) and [Response Format \(p. 218\)](#) show examples of how data type descriptors are used. The `GetItem` request specifies `S` for the `Pets` key schema attributes

(`AnimalType` and `Name`), which are of type `string`. The `GetItem` response contains a `Pets` item with attributes of type `string` (`S`), `number` (`N`), `map` (`M`), and `list` (`L`).

The following is a complete list of DynamoDB data type descriptors:

- **S** – String
- **N** – Number
- **B** – Binary
- **BOOL** – Boolean
- **NULL** – Null
- **M** – Map
- **L** – List
- **SS** – String Set
- **NS** – Number Set
- **BS** – Binary Set

Note

For detailed descriptions of DynamoDB data types, see [Data Types \(p. 13\)](#).

Numeric Data

Different programming languages offer different levels of support for JSON. In some cases, you might decide to use a third-party library for validating and parsing JSON documents.

Some third-party libraries build upon the JSON number type, providing their own types such as `int`, `long`, or `double`. However, the native number data type in DynamoDB does not map exactly to these other data types, so these type distinctions can cause conflicts. In addition, many JSON libraries do not handle fixed-precision numeric values, and they automatically infer a double data type for digit sequences that contain a decimal point.

To solve these problems, DynamoDB provides a single numeric type with no data loss. To avoid unwanted implicit conversions to a double value, DynamoDB uses strings for the data transfer of numeric values. This approach provides flexibility for updating attribute values while maintaining proper sorting semantics, such as putting the values "01", "2", and "03" in the proper sequence.

If number precision is important to your application, you should convert numeric values to strings before you pass them to DynamoDB.

Binary Data

DynamoDB supports binary attributes. However, JSON does not natively support encoding binary data. To send binary data in a request, you will need to encode it in base64 format. Upon receiving the request, DynamoDB decodes the base64 data back to binary.

The base64 encoding scheme used by DynamoDB is described at [RFC 4648](#) on the Internet Engineering Task Force (IETF) website.

Error Handling with DynamoDB

This section describes runtime errors and how to handle them. It also describes error messages and codes that are specific to Amazon DynamoDB.

Topics

- [Error Components \(p. 220\)](#)
- [Error Messages and Codes \(p. 220\)](#)
- [Error Handling in Your Application \(p. 223\)](#)
- [Error Retries and Exponential Backoff \(p. 224\)](#)
- [Batch Operations and Error Handling \(p. 224\)](#)

Error Components

When your program sends a request, DynamoDB attempts to process it. If the request is successful, DynamoDB returns an HTTP success status code (200 OK), along with the results from the requested operation.

If the request is unsuccessful, DynamoDB returns an error. Each error has three components:

- An HTTP status code (such as 400).
- An exception name (such as `ResourceNotFoundException`).
- An error message (such as `Requested resource not found: Table: tablename not found`).

The AWS SDKs take care of propagating errors to your application so that you can take appropriate action. For example, in a Java program, you can write try-catch logic to handle a `ResourceNotFoundException`.

If you are not using an AWS SDK, you need to parse the content of the low-level response from DynamoDB. The following is an example of such a response.

```
HTTP/1.1 400 Bad Request
x-amzn-RequestId: LDM6CJP8RMQ1FHKSC1RBVJFPNVV4KQNSO5AEMF66Q9ASUAAJG
Content-Type: application/x-amz-json-1.0
Content-Length: 240
Date: Thu, 15 Mar 2012 23:56:23 GMT

{"__type":"com.amazonaws.dynamodb.v20120810#ResourceNotFoundException",
"message":"Requested resource not found: Table: tablename not found"}
```

Error Messages and Codes

The following is a list of exceptions returned by DynamoDB, grouped by HTTP status code. If *OK to retry?* is Yes, you can submit the same request again. If *OK to retry?* is No, you need to fix the problem on the client side before you submit a new request.

HTTP Status Code 400

An HTTP 400 status code indicates a problem with your request, such as authentication failure, missing required parameters, or exceeding a table's provisioned throughput. You have to fix the issue in your application before submitting the request again.

AccessDeniedException

Message: *Access denied.*

The client did not correctly sign the request. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the [Signature Version 4 Signing Process](#) in the [AWS General Reference](#).

OK to retry? No

ConditionalCheckFailedException

Message: *The conditional request failed.*

You specified a condition that evaluated to false. For example, you might have tried to perform a conditional update on an item, but the actual value of the attribute did not match the expected value in the condition.

OK to retry? No

IncompleteSignatureException

Message: *The request signature does not conform to AWS standards.*

The request signature did not include all of the required components. If you are using an AWS SDK, requests are signed for you automatically; otherwise, go to the [Signature Version 4 Signing Process](#) in the [AWS General Reference](#).

OK to retry? No

ItemCollectionSizeLimitExceededException

Message: *Collection size exceeded.*

For a table with a local secondary index, a group of items with the same partition key value has exceeded the maximum size limit of 10 GB. For more information on item collections, see [Item Collections \(p. 544\)](#).

OK to retry? Yes

LimitExceededException

Message: *Too many operations for a given subscriber.*

There are too many concurrent control plane operations. The cumulative number of tables and indexes in the CREATING, DELETING, or UPDATING state cannot exceed 50.

OK to retry? Yes

MissingAuthenticationTokenException

Message: *Request must contain a valid (registered) AWS Access Key ID.*

The request did not include the required authorization header, or it was malformed. See [DynamoDB Low-Level API \(p. 216\)](#).

OK to retry? No

ProvisionedThroughputExceededException

Message: *You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. To view performance metrics for provisioned throughput vs. consumed throughput, open the Amazon CloudWatch console.*

Example: Your request rate is too high. The AWS SDKs for DynamoDB automatically retry requests that receive this exception. Your request is eventually successful, unless your retry queue is too large to finish. Reduce the frequency of requests using [Error Retries and Exponential Backoff \(p. 224\)](#).

OK to retry? Yes

RequestLimitExceeded

Message: Throughput exceeds the current throughput limit for your account. To request a limit increase, contact AWS Support at <https://aws.amazon.com/support>.

Example: Rate of on-demand requests exceeds the allowed account throughput.

OK to retry? Yes

ResourceInUseException

Message: *The resource which you are attempting to change is in use.*

Example: You tried to re-create an existing table, or delete a table currently in the CREATING state.

OK to retry? No

ResourceNotFoundException

Message: *Requested resource not found.*

Example: The table that is being requested does not exist, or is too early in the CREATING state.

OK to retry? No

ThrottlingException

Message: *Rate of requests exceeds the allowed throughput.*

This exception is returned as an AmazonServiceException response with a THROTTLING_EXCEPTION status code. This exception might be returned if you perform [control plane](#) API operations too rapidly.

For tables using on-demand mode, this exception might be returned for any [data plane](#) API operation if your request rate is too high. To learn more about on-demand scaling, see [Peak Traffic and Scaling Properties](#)

OK to retry? Yes

UnrecognizedClientException

Message: *The Access Key ID or security token is invalid.*

The request signature is incorrect. The most likely cause is an invalid AWS access key ID or secret key.

OK to retry? Yes

ValidationException

Message: Varies, depending upon the specific error(s) encountered

This error can occur for several reasons, such as a required parameter that is missing, a value that is out of range, or mismatched data types. The error message contains details about the specific part of the request that caused the error.

OK to retry? No

HTTP Status Code 5xx

An HTTP 5xx status code indicates a problem that must be resolved by AWS. This might be a transient error, in which case you can retry your request until it succeeds. Otherwise, go to the [AWS Service Health Dashboard](#) to see if there are any operational issues with the service.

Internal Server Error (HTTP 500)

DynamoDB could not process your request.

OK to retry? Yes

Note

You might encounter internal server errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately.

Service Unavailable (HTTP 503)

DynamoDB is currently unavailable. (This should be a temporary state.)

OK to retry? Yes

Error Handling in Your Application

For your application to run smoothly, you need to add logic to catch and respond to errors. Typical approaches include using `try-catch` blocks or `if-then` statements.

The AWS SDKs perform their own retries and error checking. If you encounter an error while using one of the AWS SDKs, the error code and description can help you troubleshoot it.

You should also see a `Request ID` in the response. The `Request ID` can be helpful if you need to work with AWS Support to diagnose an issue.

The following Java code example tries to get an item from a DynamoDB table and performs rudimentary error handling. (In this case, it simply informs the user that the request failed.)

```
Table table = dynamoDB.getTable("Movies");

try {
    Item item = table.getItem("year", 1978, "title", "Superman");
    if (item != null) {
        System.out.println("Result: " + item);
    } else {
        //No such item exists in the table
        System.out.println("Item not found");
    }
} catch (AmazonServiceException ase) {
    System.err.println("Could not complete operation");
    System.err.println("Error Message: " + ase.getMessage());
    System.err.println("HTTP Status: " + ase.getStatusCode());
```

```
System.err.println("AWS Error Code: " + ase.getErrorCode());
System.err.println("Error Type:      " + ase.getErrorType());
System.err.println("Request ID:     " + ase.getRequestId());

} catch (AmazonClientException ace) {
    System.err.println("Internal error occurred communicating with DynamoDB");
    System.out.println("Error Message:  " + ace.getMessage());
}
```

In this code example, the try-catch construct handles two different kinds of exceptions:

- `AmazonServiceException`—Thrown if the client request was correctly transmitted to DynamoDB, but DynamoDB could not process the request and returned an error response instead.
- `AmazonClientException`—Thrown if the client could not get a response from a service, or if the client could not parse the response from a service.

Error Retries and Exponential Backoff

Numerous components on a network, such as DNS servers, switches, load balancers, and others, can generate errors anywhere in the life of a given request. The usual technique for dealing with these error responses in a networked environment is to implement retries in the client application. This technique increases the reliability of the application.

Each AWS SDK implements retry logic automatically. You can modify the retry parameters to your needs. For example, consider a Java application that requires a fail-fast strategy, with no retries allowed in case of an error. With the AWS SDK for Java, you could use the `ClientConfiguration` class and provide a `maxErrorRetry` value of 0 to turn off the retries. For more information, see the AWS SDK documentation for your programming language.

If you're not using an AWS SDK, you should retry original requests that receive server errors (5xx). However, client errors (4xx, other than a `ThrottlingException` or a `ProvisionedThroughputExceededException`) indicate that you need to revise the request itself to correct the problem before trying again.

In addition to simple retries, each AWS SDK implements an exponential backoff algorithm for better flow control. The concept behind exponential backoff is to use progressively longer waits between retries for consecutive error responses. For example, up to 50 milliseconds before the first retry, up to 100 milliseconds before the second, up to 200 milliseconds before third, and so on. However, after a minute, if the request has not succeeded, the problem might be the request size exceeding your provisioned throughput, and not the request rate. Set the maximum number of retries to stop around one minute. If the request is not successful, investigate your provisioned throughput options.

Note

The AWS SDKs implement automatic retry logic and exponential backoff.

Most exponential backoff algorithms use jitter (randomized delay) to prevent successive collisions. Because you aren't trying to avoid such collisions in these cases, you do not need to use this random number. However, if you use concurrent clients, jitter can help your requests succeed faster. For more information, see the blog post about [Exponential Backoff and Jitter](#).

Batch Operations and Error Handling

The DynamoDB low-level API supports batch operations for reads and writes. `BatchGetItem` reads items from one or more tables, and `BatchWriteItem` puts or deletes items in one or more tables. These batch operations are implemented as wrappers around other non-batch DynamoDB operations. In other words, `BatchGetItem` invokes `GetItem` once for each item in the batch. Similarly, `BatchWriteItem` invokes `DeleteItem` or `PutItem`, as appropriate, for each item in the batch.

A batch operation can tolerate the failure of individual requests in the batch. For example, consider a `BatchGetItem` request to read five items. Even if some of the underlying `GetItem` requests fail, this does not cause the entire `BatchGetItem` operation to fail. However, if all five read operations fail, then the entire `BatchGetItem` fails.

The batch operations return information about individual requests that fail so that you can diagnose the problem and retry the operation. For `BatchGetItem`, the tables and primary keys in question are returned in the `UnprocessedKeys` value of the response. For `BatchWriteItem`, similar information is returned in `UnprocessedItems`.

The most likely cause of a failed read or a failed write is *throttling*. For `BatchGetItem`, one or more of the tables in the batch request does not have enough provisioned read capacity to support the operation. For `BatchWriteItem`, one or more of the tables does not have enough provisioned write capacity.

If DynamoDB returns any unprocessed items, you should retry the batch operation on those items. However, *we strongly recommend that you use an exponential backoff algorithm*. If you retry the batch operation immediately, the underlying read or write requests can still fail due to throttling on the individual tables. If you delay the batch operation using exponential backoff, the individual requests in the batch are much more likely to succeed.

Higher-Level Programming Interfaces for DynamoDB

The AWS SDKs provide applications with low-level interfaces for working with Amazon DynamoDB. These client-side classes and methods correspond directly to the low-level DynamoDB API. However, many developers experience a sense of disconnect, or *impedance mismatch*, when they need to map complex data types to items in a database table. With a low-level database interface, developers must write methods for reading or writing object data to database tables, and vice versa. The amount of extra code required for each combination of object type and database table can seem overwhelming.

To simplify development, the AWS SDKs for Java and .NET provide additional interfaces with higher levels of abstraction. The higher-level interfaces for DynamoDB let you define the relationships between objects in your program and the database tables that store those objects' data. After you define this mapping, you call simple object methods such as `save`, `load`, or `delete`, and the underlying low-level DynamoDB operations are automatically invoked on your behalf. This allows you to write object-centric code, rather than database-centric code.

The higher-level programming interfaces for DynamoDB are available in the AWS SDKs for Java and .NET.

Java

- [Java: DynamoDBMapper \(p. 225\)](#)

.NET

- [.NET: Document Model \(p. 273\)](#)
- [.NET: Object Persistence Model \(p. 295\)](#)

Java: DynamoDBMapper

Topics

- [Supported Data Types \(p. 228\)](#)
- [Java Annotations for DynamoDB \(p. 229\)](#)
- [DynamoDBMapper Class \(p. 233\)](#)
- [Optional Configuration Settings for DynamoDBMapper \(p. 241\)](#)
- [Example: CRUD Operations \(p. 242\)](#)
- [Example: Batch Write Operations \(p. 244\)](#)
- [Example: Query and Scan \(p. 251\)](#)
- [Example: Transaction Operations \(p. 260\)](#)
- [Optimistic Locking with Version Number \(p. 267\)](#)
- [Mapping Arbitrary Data \(p. 269\)](#)

The AWS SDK for Java provides a `DynamoDBMapper` class, allowing you to map your client-side classes to Amazon DynamoDB tables. To use `DynamoDBMapper`, you define the relationship between items in a DynamoDB table and their corresponding object instances in your code. The `DynamoDBMapper` class enables you to access your tables; perform various create, read, update, and delete (CRUD) operations; and execute queries.

Note

The `DynamoDBMapper` class does not allow you to create, update, or delete tables. To perform those tasks, use the low-level SDK for Java interface instead. For more information, see [Working with DynamoDB Tables in Java \(p. 362\)](#).

The SDK for Java provides a set of annotation types so that you can map your classes to tables. For example, consider a `ProductCatalog` table that has `Id` as the partition key.

```
ProductCatalog(Id, ...)
```

You can map a class in your client application to the `ProductCatalog` table as shown in the following Java code. This code defines a plain old Java object (POJO) named `CatalogItem`, which uses annotations to map object fields to DynamoDB attribute names.

Example

```
package com.amazonaws.codesamples;

import java.util.Set;

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIgnore;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) {this.id = id; }

    @DynamoDBAttribute(attributeName="Title")
```

```

public String getTitle() {return title; }
public void setTitle(String title) { this.title = title; }

@DynamoDBAttribute(attributeName="ISBN")
public String getISBN() { return ISBN; }
public void setISBN(String ISBN) { this.ISBN = ISBN; }

@DynamoDBAttribute(attributeName="Authors")
public Set<String> getBookAuthors() { return bookAuthors; }
public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors = bookAuthors; }

@DynamoDBIgnore
public String getSomeProp() { return someProp; }
public void setSomeProp(String someProp) { this.someProp = someProp; }
}

```

In the preceding code, the `@DynamoDBTable` annotation maps the `CatalogItem` class to the `ProductCatalog` table. You can store individual class instances as items in the table. In the class definition, the `@DynamoDBHashKey` annotation maps the `Id` property to the primary key.

By default, the class properties map to the same name attributes in the table. The properties `Title` and `ISBN` map to the same name attributes in the table.

The `@DynamoDBAttribute` annotation is optional when the name of the DynamoDB attribute matches the name of the property declared in the class. When they differ, use this annotation with the `attributeName()` parameter to specify which DynamoDB attribute this property corresponds to.

In the preceding example, the `@DynamoDBAttribute` annotation is added to each property to ensure that the property names match exactly with the tables created in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), and to be consistent with the attribute names used in other code examples in this guide.

Your class definition can have properties that don't map to any attributes in the table. You identify these properties by adding the `@DynamoDBIgnore` annotation. In the preceding example, the `SomeProp` property is marked with the `@DynamoDBIgnore` annotation. When you upload a `CatalogItem` instance to the table, your `DynamoDBMapper` instance does not include the `SomeProp` property. In addition, the mapper does not return this attribute when you retrieve an item from the table.

After you define your mapping class, you can use `DynamoDBMapper` methods to write an instance of that class to a corresponding item in the Catalog table. The following code example demonstrates this technique.

```

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDBMapper mapper = new DynamoDBMapper(client);

CatalogItem item = new CatalogItem();
item.setId(102);
item.setTitle("Book 102 Title");
item.setISBN("222-222222222");
item.setBookAuthors(new HashSet<String>(Arrays.asList("Author 1", "Author 2")));
item.setSomeProp("Test");

mapper.save(item);

```

The following code example shows how to retrieve the item and access some of its attributes.

```

CatalogItem partitionKey = new CatalogItem();
partitionKey.setId(102);

```

```

DynamoDBQueryExpression<CatalogItem> queryExpression = new
    DynamoDBQueryExpression<CatalogItem>()
        .withHashKeyValues(partitionKey);

List<CatalogItem> itemList = mapper.query(CatalogItem.class, queryExpression);

for (int i = 0; i < itemList.size(); i++) {
    System.out.println(itemList.get(i).getTitle());
    System.out.println(itemList.get(i).getBookAuthors());
}

```

DynamoDBMapper offers an intuitive, natural way of working with DynamoDB data within Java. It also provides several built-in features, such as optimistic locking, ACID transactions, autogenerated partition key and sort key values, and object versioning.

Supported Data Types

This section describes the supported primitive Java data types, collections, and arbitrary data types in Amazon DynamoDB.

Amazon DynamoDB supports the following primitive Java data types and primitive wrapper classes.

- `String`
- `Boolean, boolean`
- `Byte, byte`
- Date (as ISO_8601 millisecond-precision string, shifted to UTC)
- Calendar (as ISO_8601 millisecond-precision string, shifted to UTC)
- `Long, long`
- `Integer, int`
- `Double, double`
- `Float, float`
- `BigDecimal`
- `BigInteger`

Note

- For more information about DynamoDB naming rules and the various supported data types, see [Naming Rules and Data Types \(p. 12\)](#).
- Empty Binary values are supported by the DynamoDBMapper.
- Empty String values are supported by AWS SDK for Java 2.0.

In AWS SDK for Java 1.0, DynamoDBMapper supports reading of empty String attribute values, however, it will not write empty String attribute values because these attributes are dropped from the request.

DynamoDB supports the Java [Set](#), [List](#), and [Map](#) collection types. The following table summarizes how these Java types map to the DynamoDB types.

| Java type | DynamoDB type |
|------------------|-----------------|
| All number types | N (number type) |
| Strings | S (string type) |

| Java type | DynamoDB type |
|--------------------------------------|--|
| Boolean | BOOL (Boolean type), 0 or 1. |
| ByteBuffer | B (binary type) |
| Date | S (string type). The Date values are stored as ISO-8601 formatted strings. |
| Set collection types | SS (string set) type, NS (number set) type, or BS (binary set) type. |

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see [Mapping Arbitrary Data \(p. 269\)](#).

Java Annotations for DynamoDB

This section describes the annotations that are available for mapping your classes and properties to tables and attributes in Amazon DynamoDB.

For the corresponding Javadoc documentation, see [Annotation Types Summary](#) in the [AWS SDK for Java API Reference](#).

Note

In the following annotations, only `DynamoDBTable` and the `DynamoDBHashKey` are required.

Topics

- [DynamoDBAttribute \(p. 229\)](#)
- [DynamoDBAutoGeneratedKey \(p. 230\)](#)
- [DynamoDBDocument \(p. 230\)](#)
- [DynamoDBHashKey \(p. 231\)](#)
- [DynamoDBIgnore \(p. 232\)](#)
- [DynamoDBIndexHashKey \(p. 232\)](#)
- [DynamoDBIndexRangeKey \(p. 232\)](#)
- [DynamoDBRangeKey \(p. 232\)](#)
- [DynamoDBTable \(p. 233\)](#)
- [DynamoDBTypeConverted \(p. 233\)](#)
- [DynamoDBTyped \(p. 233\)](#)
- [DynamoDBVersionAttribute \(p. 233\)](#)

DynamoDBAttribute

Maps a property to a table attribute. By default, each class property maps to an item attribute with the same name. However, if the names are not the same, you can use this annotation to map a property to the attribute. In the following Java snippet, the `DynamoDBAttribute` maps the `BookAuthors` property to the `Authors` attribute name in the table.

```
@DynamoDBAttribute(attributeName = "Authors")
public List<String> getBookAuthors() { return BookAuthors; }
public void setBookAuthors(List<String> BookAuthors) { this.BookAuthors = BookAuthors; }
```

The `DynamoDBMapper` uses `Authors` as the attribute name when saving the object to the table.

DynamoDBAutoGeneratedKey

Marks a partition key or sort key property as being autogenerated. `DynamoDBMapper` generates a random [UUID](#) when saving these attributes. Only String properties can be marked as autogenerated keys.

The following example demonstrates using autogenerated keys.

```
@DynamoDBTable(tableName="AutoGeneratedKeysExample")
public class AutoGeneratedKeys {
    private String id;
    private String payload;

    @DynamoDBHashKey(attributeName = "Id")
    @DynamoDBAutoGeneratedKey
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    @DynamoDBAttribute(attributeName="payload")
    public String getPayload() { return this.payload; }
    public void setPayload(String payload) { this.payload = payload; }

    public static void saveItem() {
        AutoGeneratedKeys obj = new AutoGeneratedKeys();
        obj.setPayload("abc123");

        // id field is null at this point
        DynamoDBMapper mapper = new DynamoDBMapper(dynamoDBClient);
        mapper.save(obj);

        System.out.println("Object was saved with id " + obj.getId());
    }
}
```

DynamoDBDocument

Indicates that a class can be serialized as an Amazon DynamoDB document.

For example, suppose that you wanted to map a JSON document to a DynamoDB attribute of type Map (`M`). The following code example defines an item containing a nested attribute (`Pictures`) of type Map.

```
public class ProductCatalogItem {

    private Integer id; //partition key
    private Pictures pictures;
    /* ...other attributes omitted... */

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) {this.id = id; }

    @DynamoDBAttribute(attributeName="Pictures")
    public Pictures getPictures() { return pictures; }
    public void setPictures(Pictures pictures) {this.pictures = pictures; }

    // Additional properties go here.

    @DynamoDBDocument
    public static class Pictures {
        private String frontView;
        private String rearView;
        private String sideView;
```

```

    @DynamoDBAttribute(attributeName = "FrontView")
    public String getFrontView() { return frontView; }
    public void setFrontView(String frontView) { this.frontView = frontView; }

    @DynamoDBAttribute(attributeName = "RearView")
    public String getRearView() { return rearView; }
    public void setRearView(String rearView) { this.rearView = rearView; }

    @DynamoDBAttribute(attributeName = "SideView")
    public String getSideView() { return sideView; }
    public void setSideView(String sideView) { this.sideView = sideView; }

}
}

```

You could then save a new `ProductCatalog` item, with `Pictures`, as shown in the following example.

```

ProductCatalogItem item = new ProductCatalogItem();

Pictures pix = new Pictures();
pix.setFrontView("http://example.com/products/123_front.jpg");
pix.setRearView("http://example.com/products/123_rear.jpg");
pix.setSideView("http://example.com/products/123_left_side.jpg");
item.setPictures(pix);

item.setId(123);

mapper.save(item);

```

The resulting `ProductCatalog` item would look like the following (in JSON format).

```
{
  "Id" : 123
  "Pictures" : {
    "SideView" : "http://example.com/products/123_left_side.jpg",
    "RearView" : "http://example.com/products/123_rear.jpg",
    "FrontView" : "http://example.com/products/123_front.jpg"
  }
}
```

DynamoDBHashKey

Maps a class property to the partition key of the table. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Assume that you have a table, `ProductCatalog`, that has `Id` as the primary key. The following Java code defines a `CatalogItem` class and maps its `Id` property to the primary key of the `ProductCatalog` table using the `@DynamoDBHashKey` tag.

```

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {
    private Integer Id;
    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() {
        return Id;
    }
    public void setId(Integer Id) {
        this.Id = Id;
    }
    // Additional properties go here.
}

```

```
}
```

DynamoDBIgnore

Indicates to the `DynamoDBMapper` instance that the associated property should be ignored. When saving data to the table, the `DynamoDBMapper` does not save this property to the table.

Applied to the getter method or the class field for a non-modeled property. If the annotation is applied directly to the class field, the corresponding getter and setter must be declared in the same class.

DynamoDBIndexHashKey

Maps a class property to the partition key of a global secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to query a global secondary index. You must specify the index name (`globalSecondaryIndexName`). If the name of the class property is different from the index partition key, you also must specify the name of that index attribute (`attributeName`).

DynamoDBIndexRangeKey

Maps a class property to the sort key of a global secondary index or a local secondary index. The property must be one of the scalar string, number, or binary types. The property can't be a collection type.

Use this annotation if you need to query a local secondary index or a global secondary index and want to refine your results using the index sort key. You must specify the index name (either `globalSecondaryIndexName` or `localSecondaryIndexName`). If the name of the class property is different from the index sort key, you must also specify the name of that index attribute (`attributeName`).

DynamoDBRangeKey

Maps a class property to the sort key of the table. The property must be one of the scalar string, number, or binary types. It cannot be a collection type.

If the primary key is composite (partition key and sort key), you can use this tag to map your class field to the sort key. For example, assume that you have a `Reply` table that stores replies for forum threads. Each thread can have many replies. So the primary key of this table is both the `ThreadId` and `ReplyDateTime`. The `ThreadId` is the partition key, and `ReplyDateTime` is the sort key.

The following Java code defines a `Reply` class and maps it to the `Reply` table. It uses both the `@DynamoDBHashKey` and `@DynamoDBRangeKey` tags to identify class properties that map to the primary key.

```
@DynamoDBTable(tableName="Reply")
public class Reply {
    private Integer id;
    private String replyDateTime;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    @DynamoDBRangeKey(attributeName="ReplyDateTime")
    public String getReplyDateTime() { return replyDateTime; }
    public void setReplyDateTime(String replyDateTime) { this.replyDateTime =
        replyDateTime; }
```

```
// Additional properties go here.  
}
```

DynamoDBTable

Identifies the target table in DynamoDB. For example, the following Java code defines a class `Developer` and maps it to the `People` table in DynamoDB.

```
@DynamoDBTable(tableName="People")  
public class Developer { ... }
```

The `@DynamoDBTable` annotation can be inherited. Any new class that inherits from the `Developer` class also maps to the `People` table. For example, assume that you create a `Lead` class that inherits from the `Developer` class. Because you mapped the `Developer` class to the `People` table, the `Lead` class objects are also stored in the same table.

The `@DynamoDBTable` can also be overridden. Any new class that inherits from the `Developer` class by default maps to the same `People` table. However, you can override this default mapping. For example, if you create a class that inherits from the `Developer` class, you can explicitly map it to another table by adding the `@DynamoDBTable` annotation as shown in the following Java code example.

```
@DynamoDBTable(tableName="Managers")  
public class Manager extends Developer { ... }
```

DynamoDBTypeConverted

An annotation to mark a property as using a custom type converter. Can be annotated on a user-defined annotation to pass additional properties to the `DynamoDBTypeConverter`.

The `DynamoDBTypeConverter` interface lets you map your own arbitrary data types to a data type that is natively supported by DynamoDB. For more information, see [Mapping Arbitrary Data \(p. 269\)](#).

DynamoDBTyped

An annotation to override the standard attribute type binding. Standard types do not require the annotation if applying the default attribute binding for that type.

DynamoDBVersionAttribute

Identifies a class property for storing an optimistic locking version number. `DynamoDBMapper` assigns a version number to this property when it saves a new item, and increments it each time you update the item. Only number scalar types are supported. For more information about data types, see [Data Types \(p. 13\)](#). For more information about versioning, see [Optimistic Locking with Version Number \(p. 267\)](#).

DynamoDBMapper Class

The `DynamoDBMapper` class is the entry point to Amazon DynamoDB. It provides access to a DynamoDB endpoint and enables you to access your data in various tables. It also enables you to perform various create, read, update, and delete (CRUD) operations on items, and execute queries and scans against tables. This class provides the following methods for working with DynamoDB.

For the corresponding Javadoc documentation, see [DynamoDBMapper](#) in the [AWS SDK for Java API Reference](#).

Topics

- [save \(p. 234\)](#)
- [load \(p. 234\)](#)
- [delete \(p. 235\)](#)
- [query \(p. 235\)](#)
- [queryPage \(p. 236\)](#)
- [scan \(p. 236\)](#)
- [scanPage \(p. 237\)](#)
- [parallelScan \(p. 237\)](#)
- [batchSave \(p. 238\)](#)
- [batchLoad \(p. 238\)](#)
- [batchDelete \(p. 238\)](#)
- [batchWrite \(p. 239\)](#)
- [transactionWrite \(p. 239\)](#)
- [transactionLoad \(p. 240\)](#)
- [count \(p. 240\)](#)
- [generateCreateTableRequest \(p. 240\)](#)
- [createS3Link \(p. 240\)](#)
- [getS3ClientCache \(p. 241\)](#)

Save

Saves the specified object to the table. The object that you want to save is the only required parameter for this method. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object.

If an item that has the same primary key does not exist, this method creates a new item in the table. If an item that has the same primary key exists, it updates the existing item. If the partition key and sort key are of type String and are annotated with `@DynamoDBAutoGeneratedKey`, they are given a random universally unique identifier (UUID) if left uninitialized. Version fields that are annotated with `@DynamoDBVersionAttribute` are incremented by one. Additionally, if a version field is updated or a key generated, the object passed in is updated as a result of the operation.

By default, only attributes corresponding to mapped class properties are updated. Any additional existing attributes on an item are unaffected. However, if you specify `SaveBehavior.CLOBBER`, you can force the item to be completely overwritten.

```
mapper.save(obj, new DynamoDBMapperConfig(DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see [Optimistic Locking with Version Number \(p. 267\)](#).

load

Retrieves an item from a table. You must provide the primary key of the item that you want to retrieve. You can provide optional configuration parameters using the `DynamoDBMapperConfig` object. For example, you can optionally request strongly consistent reads to ensure that this method retrieves only the latest item values as shown in the following Java statement.

```
CatalogItem item = mapper.load(CatalogItem.class, item.getId(),
    new DynamoDBMapperConfig(DynamoDBMapperConfig.ConsistentReads.CONSISTENT));
```

By default, DynamoDB returns the item that has values that are eventually consistent. For information about the eventual consistency model of DynamoDB, see [Read Consistency \(p. 16\)](#).

delete

Deletes an item from the table. You must pass in an object instance of the mapped class.

If you have versioning enabled, the client-side and server-side item versions must match. However, the version does not need to match if the `SaveBehavior.CLOBBER` option is used. For more information about versioning, see [Optimistic Locking with Version Number \(p. 267\)](#).

query

Queries a table or a secondary index. You can query a table or an index only if it has a composite primary key (partition key and sort key). This method requires you to provide a partition key value and a query filter that is applied on the sort key. A filter expression includes a condition and a value.

Assume that you have a table, `Reply`, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the `Reply` table consists of the `Id` and `ReplyDateTime` fields, where `Id` is the partition key and `ReplyDateTime` is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

Assume that you created a mapping between a `Reply` class and the corresponding `Reply` table in DynamoDB. The following Java code uses `DynamoDBMapper` to find all replies in the past two weeks for a specific thread subject.

Example

```
String forumName = "&DDB;";
String forumSubject = "&DDB; Thread 1";
String partitionKey = forumName + "#" + forumSubject;

long twoWeeksAgoMilli = (new Date()).getTime() - (14L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

Map<String,AttributeValue> eav = new HashMap<String,AttributeValue>();
eav.put(":v1", new AttributeValue().withS(partitionKey));
eav.put(":v2",new AttributeValue().withS(twoWeeksAgoStr.toString()));

DynamoDBQueryExpression<Reply> queryExpression = new DynamoDBQueryExpression<Reply>()
    .withKeyConditionExpression("Id = :v1 and ReplyDateTime > :v2")
    .withExpressionAttributeValues(eav);

List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);
```

The query returns a collection of `Reply` objects.

By default, the `query` method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the `latestReplies` collection.

To query an index, you must first model the index as a mapper class. Suppose that the `Reply` table has a global secondary index named `PostedBy-Message-Index`. The partition key for this index is `PostedBy`, and the sort key is `Message`. The class definition for an item in the index would look like the following.

```
@DynamoDBTable(tableName="Reply")
public class PostedByMessage {
    private String postedBy;
    private String message;

    @DynamoDBIndexHashKey(globalSecondaryIndexName = "PostedBy-Message-Index",
    attributeName = "PostedBy")
    public String getPostedBy() { return postedBy; }
    public void setPostedBy(String postedBy) { this.postedBy = postedBy; }

    @DynamoDBIndexRangeKey(globalSecondaryIndexName = "PostedBy-Message-Index",
    attributeName = "Message")
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }

    // Additional properties go here.
}
```

The `@DynamoDBTable` annotation indicates that this index is associated with the `Reply` table. The `@DynamoDBIndexHashKey` annotation denotes the partition key (`PostedBy`) of the index, and `@DynamoDBIndexRangeKey` denotes the sort key (`Message`) of the index.

Now you can use `DynamoDBMapper` to query the index, retrieving a subset of messages that were posted by a particular user. You must specify `withIndexName` so that DynamoDB knows which index to query. The following code queries a global secondary index. Because global secondary indexes support eventually consistent reads but not strongly consistent reads, you must specify `withConsistentRead(false)`.

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS("User A"));
eav.put(":v2", new AttributeValue().withS("DynamoDB"));

DynamoDBQueryExpression<PostedByMessage> queryExpression = new
DynamoDBQueryExpression<PostedByMessage>()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false)
    .withKeyConditionExpression("PostedBy = :v1 and begins_with(Message, :v2)")
    .withExpressionAttributeValues(eav);

List<PostedByMessage> iList = mapper.query(PostedByMessage.class, queryExpression);
```

The query returns a collection of `PostedByMessage` objects.

queryPage

Queries a table or secondary index and returns a single page of matching results. As with the `query` method, you must specify a partition key value and a query filter that is applied on the sort key attribute. However, `queryPage` returns only the first "page" of data, that is, the amount of data that fits in 1 MB.

scan

Scans an entire table or a secondary index. You can optionally specify a `FilterExpression` to filter the result set.

Assume that you have a table, `Reply`, that stores forum thread replies. Each thread subject can have zero or more replies. The primary key of the `Reply` table consists of the `Id` and `ReplyDateTime` fields, where `Id` is the partition key and `ReplyDateTime` is the sort key of the primary key.

```
Reply ( Id, ReplyDateTime, ... )
```

If you mapped a Java class to the `Reply` table, you can use the `DynamoDBMapper` to scan the table. For example, the following Java code scans the entire `Reply` table, returning only the replies for a particular year.

Example

```
HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":v1", new AttributeValue().withS("2015"));

DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withFilterExpression("begins_with(ReplyDateTime,:v1)")
    .withExpressionAttributeValues(eav);

List<Reply> replies = mapper.scan(Reply.class, scanExpression);
```

By default, the `scan` method returns a "lazy-loaded" collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, iterate over the `replies` collection.

To scan an index, you must first model the index as a mapper class. Suppose that the `Reply` table has a global secondary index named `PostedBy-Message-Index`. The partition key for this index is `PostedBy`, and the sort key is `Message`. A mapper class for this index is shown in the [query \(p. 235\)](#) section. It uses the `@DynamoDBIndexHashKey` and `@DynamoDBIndexRangeKey` annotations to specify the index partition key and sort key.

The following code example scans `PostedBy-Message-Index`. It does not use a scan filter, so all of the items in the index are returned to you.

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
    .withIndexName("PostedBy-Message-Index")
    .withConsistentRead(false);

List<PostedByMessage> iList = mapper.scan(PostedByMessage.class, scanExpression);
Iterator<PostedByMessage> indexItems = iList.iterator();
```

scanPage

Scans a table or secondary index and returns a single page of matching results. As with the `scan` method, you can optionally specify a `FilterExpression` to filter the result set. However, `scanPage` only returns the first "page" of data, that is, the amount of data that fits within 1 MB.

parallelScan

Performs a parallel scan of an entire table or secondary index. You specify a number of logical segments for the table, along with a scan expression to filter the results. The `parallelScan` divides the scan task among multiple workers, one for each logical segment; the workers process the data in parallel and return the results.

The following Java code example performs a parallel scan on the `Product` table.

```
int numberOfThreads = 4;

Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
eav.put(":n", new AttributeValue().withN("100"));
```

```
DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()  
    .withFilterExpression("Price <= :n")  
    .withExpressionAttributeValues(eav);  
  
List<Product> scanResult = mapper.parallelScan(Product.class, scanExpression,  
    numberOfThreads);
```

For a Java code example illustrating the usage of `parallelScan`, see [Example: Query and Scan \(p. 251\)](#).

batchSave

Saves objects to one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code saves two items (books) to the `ProductCatalog` table.

```
Book book1 = new Book();  
book1.id = 901;  
book1.productCategory = "Book";  
book1.title = "Book 901 Title";  
  
Book book2 = new Book();  
book2.id = 902;  
book2.productCategory = "Book";  
book2.title = "Book 902 Title";  
  
mapper.batchSave(Arrays.asList(book1, book2));
```

batchLoad

Retrieves multiple items from one or more tables using their primary keys.

The following Java code retrieves two items from two different tables.

```
ArrayList<Object> itemsToGet = new ArrayList<Object>();  
  
ForumItem forumItem = new ForumItem();  
forumItem.setForumName("Amazon DynamoDB");  
itemsToGet.add(forumItem);  
  
ThreadItem threadItem = new ThreadItem();  
threadItem.setForumName("Amazon DynamoDB");  
threadItem.setSubject("Amazon DynamoDB thread 1 message text");  
itemsToGet.add(threadItem);  
  
Map<String, List<Object>> items = mapper.batchLoad(itemsToGet);
```

batchDelete

Deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees.

The following Java code deletes two items (books) from the `ProductCatalog` table.

```
Book book1 = mapper.load(Book.class, 901);  
Book book2 = mapper.load(Book.class, 902);
```

```
mapper.batchDelete(Arrays.asList(book1, book2));
```

batchWrite

Saves objects to and deletes objects from one or more tables using one or more calls to the `AmazonDynamoDB.batchWriteItem` method. This method does not provide transaction guarantees or support versioning (conditional puts or deletes).

The following Java code writes a new item to the `Forum` table, writes a new item to the `Thread` table, and deletes an item from the `ProductCatalog` table.

```
// Create a Forum item to save
Forum forumItem = new Forum();
forumItem.name = "Test BatchWrite Forum";

// Create a Thread item to save
Thread threadItem = new Thread();
threadItem.forumName = "AmazonDynamoDB";
threadItem.subject = "My sample question";

// Load a ProductCatalog item to delete
Book book3 = mapper.load(Book.class, 903);

List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
List<Book> objectsToDelete = Arrays.asList(book3);

mapper.batchWrite(objectsToWrite, objectsToDelete);
```

transactionWrite

Saves objects to and deletes objects from one or more tables using one call to the `AmazonDynamoDB.transactWriteItems` method.

For a list of transaction-specific exceptions, see [TransactWriteItems errors](#).

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see [Amazon DynamoDB Transactions](#).

Note

This method does not support the following:

- [DynamoDBMapperConfig.SaveBehavior](#).

The following Java code writes a new item to each of the `Forum` and `Thread` tables, transactionally.

```
Thread s3ForumThread = new Thread();
s3ForumThread.forumName = "S3 Forum";
s3ForumThread.subject = "Sample Subject 1";
s3ForumThread.message = "Sample Question 1";

Forum s3Forum = new Forum();
s3Forum.name = "S3 Forum";
s3Forum.category = "Amazon Web Services";
s3Forum.threads = 1;

TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
transactionWriteRequest.addPut(s3Forum);
transactionWriteRequest.addPut(s3ForumThread);
mapper.transactionWrite(transactionWriteRequest);
```

transactionLoad

Loads objects from one or more tables using one call to the `AmazonDynamoDB.transactGetItems` method.

For a list of transaction-specific exceptions, see [TransactGetItems errors](#).

For more information about DynamoDB transactions and the provided atomicity, consistency, isolation, and durability (ACID) guarantees see [Amazon DynamoDB Transactions](#).

The following Java code loads one item from each of the `Forum` and `Thread` tables, transactionally.

```
Forum dynamodbForum = new Forum();
dynamodbForum.name = "DynamoDB Forum";
Thread dynamodbForumThread = new Thread();
dynamodbForumThread.forumName = "DynamoDB Forum";

TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();
transactionLoadRequest.addLoad(dynamodbForum);
transactionLoadRequest.addLoad(dynamodbForumThread);
mapper.transactionLoad(transactionLoadRequest);
```

count

Evaluates the specified scan expression and returns the count of matching items. No item data is returned.

generateCreateTableRequest

Parses a POJO class that represents a DynamoDB table, and returns a `CreateTableRequest` for that table.

createS3Link

Creates a link to an object in Amazon S3. You must specify a bucket name and a key name, which uniquely identifies the object in the bucket.

To use `createS3Link`, your mapper class must define getter and setter methods. The following code example illustrates this by adding a new attribute and getter/setter methods to the `CatalogItem` class.

```
@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    ...

    public S3Link productImage;

    ...

    @DynamoDBAttribute(attributeName = "ProductImage")
    public S3Link getProductImage() {
        return productImage;
    }

    public void setProductImage(S3Link productImage) {
        this.productImage = productImage;
    }

    ...
}
```

```
}
```

The following Java code defines a new item to be written to the `Product` table. The item includes a link to a product image; the image data is uploaded to Amazon S3.

```
CatalogItem item = new CatalogItem();

item.id = 150;
item.title = "Book 150 Title";

String myS3Bucket = "myS3bucket";
String myS3Key = "productImages/book_150_cover.jpg";
item.setProductImage(mapper.createS3Link(myS3Bucket, myS3Key));

item.getProductImage().uploadFrom(new File("/file/path/book_150_cover.jpg"));

mapper.save(item);
```

The `S3Link` class provides many other methods for manipulating objects in Amazon S3. For more information, see the [Javadocs for `S3Link`](#).

getS3ClientCache

Returns the underlying `S3ClientCache` for accessing Amazon S3. An `S3ClientCache` is a smart Map for `AmazonS3Client` objects. If you have multiple clients, an `S3ClientCache` can help you keep the clients organized by AWS Region, and can create new Amazon S3 clients on demand.

Optional Configuration Settings for DynamoDBMapper

When you create an instance of `DynamoDBMapper`, it has certain default behaviors; you can override these defaults by using the `DynamoDBMapperConfig` class.

The following code snippet creates a `DynamoDBMapper` with custom settings:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

DynamoDBMapperConfig mapperConfig = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
    .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT)
    .withTableNameOverride(null)

    .withPaginationLoadingStrategy(DynamoDBMapperConfig.PaginationLoadingStrategy.EAGER_LOADING)
    .build();

DynamoDBMapper mapper = new DynamoDBMapper(client, mapperConfig);
```

For more information, see [DynamoDBMapperConfig](#) in the [AWS SDK for Java API Reference](#).

You can use the following arguments for an instance of `DynamoDBMapperConfig`:

- A `DynamoDBMapperConfig.ConsistentReads` enumeration value:
 - `EVENTUAL`—the mapper instance uses an eventually consistent read request.
 - `CONSISTENT`—the mapper instance uses a strongly consistent read request. You can use this optional setting with `load`, `query`, or `scan` operations. Strongly consistent reads have implications for performance and billing; see the [DynamoDB product detail page](#) for more information.

If you do not specify a read consistency setting for your mapper instance, the default is `EVENTUAL`.

- A `DynamoDBMapperConfig.PaginationLoadingStrategy` enumeration value—Controls how the mapper instance processes a paginated list of data, such as the results from a `query` or `scan`:

- **LAZY_LOADING**—the mapper instance loads data when possible, and keeps all loaded results in memory.
- **EAGER_LOADING**—the mapper instance loads the data as soon as the list is initialized.
- **ITERATION_ONLY**—you can only use an Iterator to read from the list. During the iteration, the list will clear all the previous results before loading the next page, so that the list will keep at most one page of the loaded results in memory. This also means the list can only be iterated once. This strategy is recommended when handling large items, in order to reduce memory overhead.

If you do not specify a pagination loading strategy for your mapper instance, the default is **LAZY_LOADING**.

- A `DynamoDBMapperConfig.SaveBehavior` enumeration value - Specifies how the mapper instance should deal with attributes during save operations:
 - **UPDATE**—during a save operation, all modeled attributes are updated, and unmodeled attributes are unaffected. Primitive number types (byte, int, long) are set to 0. Object types are set to null.
 - **Clobber**—clears and replaces all attributes, included unmodeled ones, during a save operation. This is done by deleting the item and re-creating it. Versioned field constraints are also disregarded.

If you do not specify the save behavior for your mapper instance, the default is **UPDATE**.

Note

DynamoDBMapper transactional operations do not support `DynamoDBMapperConfig.SaveBehavior` enumeration.

- A `DynamoDBMapperConfig.TableNameOverride` object—Instructs the mapper instance to ignore the table name specified by a class's `DynamoDBTable` annotation, and instead use a different table name that you supply. This is useful when partitioning your data into multiple tables at runtime.

You can override the default configuration object for `DynamoDBMapper` per operation, as needed.

Example: CRUD Operations

The following Java code example declares a `CatalogItem` class that has `Id`, `Title`, `ISBN`, and `Authors` properties. It uses the annotations to map these properties to the `ProductCatalog` table in DynamoDB. The example then uses the `DynamoDBMapper` to save a book object, retrieve it, update it, and then delete the book item.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.datamodeling;
```

```
import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapperConfig;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

public class DynamoDBMapperCRUDEExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

    public static void main(String[] args) throws IOException {
        testCRUDOperations();
        System.out.println("Example complete!");
    }

    @DynamoDBTable(tableName = "ProductCatalog")
    public static class CatalogItem {
        private Integer id;
        private String title;
        private String ISBN;
        private Set<String> bookAuthors;

        // Partition key
        @DynamoDBHashKey(attributeName = "Id")
        public Integer getId() {
            return id;
        }

        public void setId(Integer id) {
            this.id = id;
        }

        @DynamoDBAttribute(attributeName = "Title")
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        @DynamoDBAttribute(attributeName = "ISBN")
        public String getISBN() {
            return ISBN;
        }

        public void setISBN(String ISBN) {
            this.ISBN = ISBN;
        }

        @DynamoDBAttribute(attributeName = "Authors")
        public Set<String> getBookAuthors() {
            return bookAuthors;
        }

        public void setBookAuthors(Set<String> bookAuthors) {
            this.bookAuthors = bookAuthors;
        }
    }
}
```

```

        @Override
        public String toString() {
            return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ", id=" + id +
", title=" + title + "]";
        }
    }

    private static void testCRUDOperations() {

        CatalogItem item = new CatalogItem();
        item.setId(601);
        item.setTitle("Book 601");
        item.setISBN("611-1111111111");
        item.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));

        // Save the item (book).
        DynamoDBMapper mapper = new DynamoDBMapper(client);
        mapper.save(item);

        // Retrieve the item.
        CatalogItem itemRetrieved = mapper.load(CatalogItem.class, 601);
        System.out.println("Item retrieved:");
        System.out.println(itemRetrieved);

        // Update the item.
        itemRetrieved.setISBN("622-2222222222");
        itemRetrieved.setBookAuthors(new HashSet<String>(Arrays.asList("Author1",
"Author3")));
        mapper.save(itemRetrieved);
        System.out.println("Item updated:");
        System.out.println(itemRetrieved);

        // Retrieve the updated item.
        DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
            .withConsistentReads(DynamoDBMapperConfig.ConsistentReads.CONSISTENT)
            .build();
        CatalogItem updatedItem = mapper.load(CatalogItem.class, 601, config);
        System.out.println("Retrieved the previously updated item:");
        System.out.println(updatedItem);

        // Delete the item.
        mapper.delete(updatedItem);

        // Try to retrieve deleted item.
        CatalogItem deletedItem = mapper.load(CatalogItem.class, updatedItem.getId(),
config);
        if (deletedItem == null) {
            System.out.println("Done - Sample item is deleted.");
        }
    }
}

```

Example: Batch Write Operations

The following Java code example declares Book, Forum, Thread, and Reply classes and maps them to the Amazon DynamoDB tables using the DynamoDBMapper class.

The code illustrates the following batch write operations:

- `batchSave` to put book items in the `ProductCatalog` table.
- `batchDelete` to delete items from the `ProductCatalog` table.

- `batchWrite` to put and delete items from the `Forum` and the `Thread` tables.

For more information about the tables used in this example, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). For step-by-step instructions for testing the following example, see [Java Code Examples \(p. 330\)](#).

Example

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.datamodeling;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapperConfig;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

public class DynamoDBMapperBatchWriteExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    public static void main(String[] args) throws Exception {
        try {

            DynamoDBMapper mapper = new DynamoDBMapper(client);

            testBatchSave(mapper);
            testBatchDelete(mapper);
            testBatchWrite(mapper);

            System.out.println("Example complete!");

        }
        catch (Throwable t) {
            System.err.println("Error running the DynamoDBMapperBatchWriteExample: " + t);
            t.printStackTrace();
        }
    }
}
```

```

}

private static void testBatchSave(DynamoDBMapper mapper) {

    Book book1 = new Book();
    book1.id = 901;
    book1.inPublication = true;
    book1.ISBN = "902-11-11-1111";
    book1.pageCount = 100;
    book1.price = 10;
    book1.productCategory = "Book";
    book1.title = "My book created in batch write";

    Book book2 = new Book();
    book2.id = 902;
    book2.inPublication = true;
    book2.ISBN = "902-11-12-1111";
    book2.pageCount = 200;
    book2.price = 20;
    book2.productCategory = "Book";
    book2.title = "My second book created in batch write";

    Book book3 = new Book();
    book3.id = 903;
    book3.inPublication = false;
    book3.ISBN = "902-11-13-1111";
    book3.pageCount = 300;
    book3.price = 25;
    book3.productCategory = "Book";
    book3.title = "My third book created in batch write";

    System.out.println("Adding three books to ProductCatalog table.");
    mapper.batchSave(Arrays.asList(book1, book2, book3));
}

private static void testBatchDelete(DynamoDBMapper mapper) {

    Book book1 = mapper.load(Book.class, 901);
    Book book2 = mapper.load(Book.class, 902);
    System.out.println("Deleting two books from the ProductCatalog table.");
    mapper.batchDelete(Arrays.asList(book1, book2));
}

private static void testBatchWrite(DynamoDBMapper mapper) {

    // Create Forum item to save
    Forum forumItem = new Forum();
    forumItem.name = "Test BatchWrite Forum";
    forumItem.threads = 0;
    forumItem.category = "Amazon Web Services";

    // Create Thread item to save
    Thread threadItem = new Thread();
    threadItem.forumName = "AmazonDynamoDB";
    threadItem.subject = "My sample question";
    threadItem.message = "BatchWrite message";
    List<String> tags = new ArrayList<String>();
    tags.add("batch operations");
    tags.add("write");
    threadItem.tags = new HashSet<String>(tags);

    // Load ProductCatalog item to delete
    Book book3 = mapper.load(Book.class, 903);

    List<Object> objectsToWrite = Arrays.asList(forumItem, threadItem);
    List<Book> objectsToDelete = Arrays.asList(book3);
}

```

```
DynamoDBMapperConfig config = DynamoDBMapperConfig.builder()
    .withSaveBehavior(DynamoDBMapperConfig.SaveBehavior.CLOBBER)
    .build();

    mapper.batchWrite(objectsToWrite, objectsToDelete, config);
}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private int price;
    private int pageCount;
    private String productCategory;
    private boolean inPublication;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "ISBN")
    public String getISBN() {
        return ISBN;
    }

    public void setISBN(String ISBN) {
        this.ISBN = ISBN;
    }

    @DynamoDBAttribute(attributeName = "Price")
    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @DynamoDBAttribute(attributeName = "PageCount")
    public int getPageCount() {
        return pageCount;
    }

    public void setPageCount(int pageCount) {
        this.pageCount = pageCount;
    }

    @DynamoDBAttribute(attributeName = "ProductCategory")
    public String getProductCategory() {
```

```

        return productCategory;
    }

    public void setProductCategory(String productCategory) {
        this.productCategory = productCategory;
    }

    @DynamoDBAttribute(attributeName = "InPublication")
    public boolean getInPublication() {
        return inPublication;
    }

    public void setInPublication(boolean inPublication) {
        this.inPublication = inPublication;
    }

    @Override
    public String toString() {
        return "Book [ISBN=" + ISBN + ", price=" + price + ", product category=" +
productCategory + ", id=" + id
        + ", title=" + title + "]";
    }

}

@dynamoDBTable(tableName = "Reply")
public static class Reply {
    private String id;
    private String replyDateTime;
    private String message;
    private String postedBy;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    // Sort key
    @DynamoDBRangeKey(attributeName = "ReplyDateTime")
    public String getReplyDateTime() {
        return replyDateTime;
    }

    public void setReplyDateTime(String replyDateTime) {
        this.replyDateTime = replyDateTime;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @DynamoDBAttribute(attributeName = "PostedBy")
    public String getPostedBy() {
        return postedBy;
    }
}

```

```
public void setPostedBy(String postedBy) {
    this.postedBy = postedBy;
}

}

@DynamoDBTable(tableName = "Thread")
public static class Thread {
    private String forumName;
    private String subject;
    private String message;
    private String lastPostedDateTime;
    private String lastPostedBy;
    private Set<String> tags;
    private int answered;
    private int views;
    private int replies;

    // Partition key
    @DynamoDBHashKey(attributeName = "ForumName")
    public String getForumName() {
        return forumName;
    }

    public void setForumName(String forumName) {
        this.forumName = forumName;
    }

    // Sort key
    @DynamoDBRangeKey(attributeName = "Subject")
    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @DynamoDBAttribute(attributeName = "LastPostedDateTime")
    public String getLastPostedDateTime() {
        return lastPostedDateTime;
    }

    public void setLastPostedDateTime(String lastPostedDateTime) {
        this.lastPostedDateTime = lastPostedDateTime;
    }

    @DynamoDBAttribute(attributeName = "LastPostedBy")
    public String getLastPostedBy() {
        return lastPostedBy;
    }

    public void setLastPostedBy(String lastPostedBy) {
        this.lastPostedBy = lastPostedBy;
    }

    @DynamoDBAttribute(attributeName = "Tags")
    public Set<String> getTags() {
```

```
        return tags;
    }

    public void setTags(Set<String> tags) {
        this.tags = tags;
    }

    @DynamoDBAttribute(attributeName = "Answered")
    public int getAnswered() {
        return answered;
    }

    public void setAnswered(int answered) {
        this.answered = answered;
    }

    @DynamoDBAttribute(attributeName = "Views")
    public int getViews() {
        return views;
    }

    public void setViews(int views) {
        this.views = views;
    }

    @DynamoDBAttribute(attributeName = "Replies")
    public int getReplies() {
        return replies;
    }

    public void setReplies(int replies) {
        this.replies = replies;
    }

}

{@DynamoDBTable(tableName = "Forum")}
public static class Forum {
    private String name;
    private String category;
    private int threads;

    // Partition key
    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    @DynamoDBAttribute(attributeName = "Threads")
    public int getThreads() {
        return threads;
    }
}
```

```
        public void setThreads(int threads) {
            this.threads = threads;
        }
    }
```

Example: Query and Scan

The Java example in this section defines the following classes and maps them to the tables in Amazon DynamoDB. For more information about creating sample tables, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

- The `Book` class maps to `ProductCatalog` table
- The `Forum`, `Thread`, and `Reply` classes map to tables of the same name.

The example then executes the follow query and scan operations using a `DynamoDBMapper` instance.

- Get a book by `Id`.

The `ProductCatalog` table has `Id` as its primary key. It does not have a sort key as part of its primary key. Therefore, you cannot query the table. You can get an item using its `Id` value.

- Execute the following queries against the `Reply` table.

The `Reply` table's primary key is composed of `Id` and `ReplyDateTime` attributes. `ReplyDateTime` is a sort key. Therefore, you can query this table.

- Find replies to a forum thread posted in the last 15 days.
- Find replies to a forum thread posted in a specific date range.
- Scan the `ProductCatalog` table to find books whose price is less than a specified value.

For performance reasons, you should use the query operation instead of the scan operation. However, there are times you might need to scan a table. Suppose that there was a data entry error and one of the book prices was set to less than 0. This example scans the `ProductCategory` table to find book items (`ProductCategory` is `book`) whose price is less than 0.

- Perform a parallel scan of the `ProductCatalog` table to find bicycles of a specific type.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

Example

```
/** 
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
```

```

/*
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.datamodeling;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBQueryExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBScanExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;

public class DynamoDBMapperQueryScanExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

    public static void main(String[] args) throws Exception {
        try {

            DynamoDBMapper mapper = new DynamoDBMapper(client);

            // Get a book - Id=101
            GetBook(mapper, 101);
            // Sample forum and thread to test queries.
            String forumName = "Amazon DynamoDB";
            String threadSubject = "DynamoDB Thread 1";
            // Sample queries.
            FindRepliesInLast15Days(mapper, forumName, threadSubject);
            FindRepliesPostedWithinTimePeriod(mapper, forumName, threadSubject);

            // Scan a table and find book items priced less than specified
            // value.
            FindBooksPricedLessThanSpecifiedValue(mapper, "20");

            // Scan a table with multiple threads and find bicycle items with a
            // specified bicycle type
            int numberOfThreads = 16;
            FindBicyclesOfSpecificTypeWithMultipleThreads(mapper, numberOfThreads, "Road");

            System.out.println("Example complete!");

        }
        catch (Throwable t) {
            System.err.println("Error running the DynamoDBMapperQueryScanExample: " + t);
            t.printStackTrace();
        }
    }

    private static void GetBook(DynamoDBMapper mapper, int id) throws Exception {
        System.out.println("GetBook: Get book Id='101' ");
        System.out.println("Book table has no sort key. You can do GetItem, but not
Query.");
    }
}

```

```

        Book book = mapper.load(Book.class, id);
        System.out.format("Id = %s Title = %s, ISBN = %s %n", book.getId(),
book.getTitle(), book.getISBN());
    }

    private static void FindRepliesInLast15Days(DynamoDBMapper mapper, String forumName,
String threadSubject)
        throws Exception {
        System.out.println("FindRepliesInLast15Days: Replies within last 15 days.");

        String partitionKey = forumName + "#" + threadSubject;

        long twoWeeksAgoMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L * 1000L);
        Date twoWeeksAgo = new Date();
        twoWeeksAgo.setTime(twoWeeksAgoMilli);
        SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
        String twoWeeksAgoStr = dateFormatter.format(twoWeeksAgo);

        Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
        eav.put(":val1", new AttributeValue().withS(partitionKey));
        eav.put(":val2", new AttributeValue().withS(twoWeeksAgoStr.toString()));

        DynamoDBQueryExpression<Reply> queryExpression = new
DynamoDBQueryExpression<Reply>()
            .withKeyConditionExpression("Id = :val1 and ReplyDateTime
> :val2").withExpressionAttributeValues(eav);

        List<Reply> latestReplies = mapper.query(Reply.class, queryExpression);

        for (Reply reply : latestReplies) {
            System.out.format("Id=%s, Message=%s, PostedBy=%s %n, ReplyDateTime=%s %n",
reply.getId(),
                reply.getMessage(), reply.getPostedBy(), reply.getReplyDateTime());
        }
    }

    private static void FindRepliesPostedWithinTimePeriod(DynamoDBMapper mapper, String
forumName, String threadSubject)
        throws Exception {
        String partitionKey = forumName + "#" + threadSubject;

        System.out.println(
            "FindRepliesPostedWithinTimePeriod: Find replies for thread Message = 'DynamoDB
Thread 2' posted within a period.");
        long startDateMilli = (new Date()).getTime() - (14L * 24L * 60L * 60L * 1000L); // //
Two
                                                //
weeks
                                                //
ago.
        long endDateMilli = (new Date()).getTime() - (7L * 24L * 60L * 60L * 1000L); // One
                                                //
week
                                                //
ago.
        SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
        String startDate = dateFormatter.format(startDateMilli);
        String endDate = dateFormatter.format(endDateMilli);

        Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
        eav.put(":val1", new AttributeValue().withS(partitionKey));
        eav.put(":val2", new AttributeValue().withS(startDate));
    }
}

```

```

        eav.put(":val3", new AttributeValue().withS(endDate));

        DynamoDBQueryExpression<Reply> queryExpression = new
DynamoDBQueryExpression<Reply>()
            .withKeyConditionExpression("Id = :val1 and ReplyDateTime between :val2
and :val3")
            .withExpressionAttributeValues(eav);

        List<Reply> betweenReplies = mapper.query(Reply.class, queryExpression);

        for (Reply reply : betweenReplies) {
            System.out.format("Id=%s, Message=%s, PostedBy=%s %n, PostedDateTime=%s %n",
reply.getId(),
                reply.getMessage(), reply.getPostedBy(), reply.getReplyDateTime());
        }
    }

    private static void FindBooksPricedLessThanSpecifiedValue(DynamoDBMapper mapper, String
value) throws Exception {

        System.out.println("FindBooksPricedLessThanSpecifiedValue: Scan ProductCatalog.");

        Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
        eav.put(":val1", new AttributeValue().withN(value));
        eav.put(":val2", new AttributeValue().withS("Book"));

        DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
            .withFilterExpression("Price < :val1 and ProductCategory
= :val2").withExpressionAttributeValues(eav);

        List<Book> scanResult = mapper.scan(Book.class, scanExpression);

        for (Book book : scanResult) {
            System.out.println(book);
        }
    }

    private static void FindBicyclesOfSpecificTypeWithMultipleThreads(DynamoDBMapper
mapper, int numberOfThreads,
        String bicycleType) throws Exception {

        System.out.println("FindBicyclesOfSpecificTypeWithMultipleThreads: Scan
ProductCatalog With Multiple Threads.");
        Map<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
        eav.put(":val1", new AttributeValue().withS("Bicycle"));
        eav.put(":val2", new AttributeValue().withS(bicycleType));

        DynamoDBScanExpression scanExpression = new DynamoDBScanExpression()
            .withFilterExpression("ProductCategory = :val1 and BicycleType
= :val2").withExpressionAttributeValues(eav);

        List<Bicycle> scanResult = mapper.parallelScan(Bicycle.class, scanExpression,
numberOfThreads);
        for (Bicycle bicycle : scanResult) {
            System.out.println(bicycle);
        }
    }

    @DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private int price;
    private int pageCount;
}

```

```
private String productCategory;
private boolean inPublication;

@DynamoDBHashKey(attributeName = "Id")
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

@DynamoDBAttribute(attributeName = "Title")
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

@DynamoDBAttribute(attributeName = "ISBN")
public String getISBN() {
    return ISBN;
}

public void setISBN(String ISBN) {
    this.ISBN = ISBN;
}

@DynamoDBAttribute(attributeName = "Price")
public int getPrice() {
    return price;
}

public void setPrice(int price) {
    this.price = price;
}

@DynamoDBAttribute(attributeName = "PageCount")
public int getPageCount() {
    return pageCount;
}

public void setPageCount(int pageCount) {
    this.pageCount = pageCount;
}

@DynamoDBAttribute(attributeName = "ProductCategory")
public String getProductCategory() {
    return productCategory;
}

public void setProductCategory(String productCategory) {
    this.productCategory = productCategory;
}

@DynamoDBAttribute(attributeName = "InPublication")
public boolean getInPublication() {
    return inPublication;
}

public void setInPublication(boolean inPublication) {
    this.inPublication = inPublication;
}
```

```
    @Override
    public String toString() {
        return "Book [ISBN=" + ISBN + ", price=" + price + ", product category=" +
productCategory + ", id=" + id
        + ", title=" + title + "]";
    }
}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Bicycle {
    private int id;
    private String title;
    private String description;
    private String bicycleType;
    private String brand;
    private int price;
    private List<String> color;
    private String productCategory;

    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "Description")
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @DynamoDBAttribute(attributeName = "BicycleType")
    public String getBicycleType() {
        return bicycleType;
    }

    public void setBicycleType(String bicycleType) {
        this.bicycleType = bicycleType;
    }

    @DynamoDBAttribute(attributeName = "Brand")
    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    @DynamoDBAttribute(attributeName = "Price")
```

```

public int getPrice() {
    return price;
}

public void setPrice(int price) {
    this.price = price;
}

@DynamoDBAttribute(attributeName = "Color")
public List<String> getColor() {
    return color;
}

public void setColor(List<String> color) {
    this.color = color;
}

@DynamoDBAttribute(attributeName = "ProductCategory")
public String getProductCategory() {
    return productCategory;
}

public void setProductCategory(String productCategory) {
    this.productCategory = productCategory;
}

@Override
public String toString() {
    return "Bicycle [Type=" + bicycleType + ", color=" + color + ", price=" + price
+ ", product category="
        + productCategory + ", id=" + id + ", title=" + title + "]";
}
}

@DynamoDBTable(tableName = "Reply")
public static class Reply {
    private String id;
    private String replyDateTime;
    private String message;
    private String postedBy;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    // Range key
    @DynamoDBRangeKey(attributeName = "ReplyDateTime")
    public String getReplyDateTime() {
        return replyDateTime;
    }

    public void setReplyDateTime(String replyDateTime) {
        this.replyDateTime = replyDateTime;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }
}

```

```
public void setMessage(String message) {
    this.message = message;
}

@DynamoDBAttribute(attributeName = "PostedBy")
public String getPostedBy() {
    return postedBy;
}

public void setPostedBy(String postedBy) {
    this.postedBy = postedBy;
}
}

@DynamoDBTable(tableName = "Thread")
public static class Thread {
    private String forumName;
    private String subject;
    private String message;
    private String lastPostedDateTime;
    private String lastPostedBy;
    private Set<String> tags;
    private int answered;
    private int views;
    private int replies;

    // Partition key
    @DynamoDBHashKey(attributeName = "ForumName")
    public String getForumName() {
        return forumName;
    }

    public void setForumName(String forumName) {
        this.forumName = forumName;
    }

    // Range key
    @DynamoDBRangeKey(attributeName = "Subject")
    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    @DynamoDBAttribute(attributeName = "Message")
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @DynamoDBAttribute(attributeName = "LastPostedDateTime")
    public String getLastPostedDateTime() {
        return lastPostedDateTime;
    }

    public void setLastPostedDateTime(String lastPostedDateTime) {
        this.lastPostedDateTime = lastPostedDateTime;
    }

    @DynamoDBAttribute(attributeName = "LastPostedBy")
```

```
public String getLastPostedBy() {
    return lastPostedBy;
}

public void setLastPostedBy(String lastPostedBy) {
    this.lastPostedBy = lastPostedBy;
}

@DynamoDBAttribute(attributeName = "Tags")
public Set<String> getTags() {
    return tags;
}

public void setTags(Set<String> tags) {
    this.tags = tags;
}

@DynamoDBAttribute(attributeName = "Answered")
public int getAnswered() {
    return answered;
}

public void setAnswered(int answered) {
    this.answered = answered;
}

@DynamoDBAttribute(attributeName = "Views")
public int getViews() {
    return views;
}

public void setViews(int views) {
    this.views = views;
}

@DynamoDBAttribute(attributeName = "Replies")
public int getReplies() {
    return replies;
}

public void setReplies(int replies) {
    this.replies = replies;
}

}

@dynamoDBTable(tableName = "Forum")
public static class Forum {
    private String name;
    private String category;
    private int threads;

    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }
}
```

```

        public void setCategory(String category) {
            this.category = category;
        }

        @DynamoDBAttribute(attributeName = "Threads")
        public int getThreads() {
            return threads;
        }

        public void setThreads(int threads) {
            this.threads = threads;
        }
    }
}

```

Example: Transaction Operations

The following Java code example declares a `Forum` and a `Thread` class and maps them to the DynamoDB tables using the `DynamoDBMapper` class.

The code illustrates the following transactional operations:

- `transactionWrite` to add, update, and delete multiple items from one or more tables in one transaction.
- `transactionLoad` to retrieve multiple items from one or more tables in one transaction.

Example

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapperConfig;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.TransactionWriteRequest;
import com.amazonaws.services.dynamodbv2.datamodeling.TransactionLoadRequest;
import com.amazonaws.services.dynamodbv2.model.TransactionCanceledException;

```

```

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTransactionWriteExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTransactionLoadExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMappingException;
import com.amazonaws.services.dynamodbv2.model.ResourceNotFoundException;
import com.amazonaws.services.dynamodbv2.model.InternalServerErrorException;

public class DynamoDBMapperTransactionExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDBMapper mapper;

    public static void main(String[] args) throws Exception {
        try {

            mapper = new DynamoDBMapper(client);

            testPutAndUpdateInTransactionWrite();
            testPutWithConditionalUpdateInTransactionWrite();
            testPutWithConditionCheckInTransactionWrite();
            testMixedOperationsInTransactionWrite();
            testTransactionLoadWithSave();
            testTransactionLoadWithTransactionWrite();
            System.out.println("Example complete");

        }
        catch (Throwable t) {
            System.err.println("Error running the DynamoDBMapperTransactionWriteExample: "
+ t);
            t.printStackTrace();
        }
    }

    private static void testTransactionLoadWithSave() {
        // Create new Forum item for DynamoDB using save
        Forum dynamodbForum = new Forum();
        dynamodbForum.name = "DynamoDB Forum";
        dynamodbForum.category = "Amazon Web Services";
        dynamodbForum.threads = 0;
        mapper.save(dynamodbForum);

        // Add a thread to DynamoDB Forum
        Thread dynamodbForumThread = new Thread();
        dynamodbForumThread.forumName = "DynamoDB Forum";
        dynamodbForumThread.subject = "Sample Subject 1";
        dynamodbForumThread.message = "Sample Question 1";
        mapper.save(dynamodbForumThread);

        // Update DynamoDB Forum to reflect updated thread count
        dynamodbForum.threads = 1;
        mapper.save(dynamodbForum);

        // Read DynamoDB Forum item and Thread item at the same time in a serializable
manner
        TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();

        // Read entire item for DynamoDB Forum
        transactionLoadRequest.addLoad(dynamodbForum);

        // Only read subject and message attributes from Thread item
        DynamoDBTransactionLoadExpression loadExpressionForThread = new
DynamoDBTransactionLoadExpression()

        .withProjectionExpression("Subject, Message");
        transactionLoadRequest.addLoad(dynamodbForumThread, loadExpressionForThread);
    }
}

```

```

// Loaded objects are guaranteed to be in same order as the order in which they are
// added to TransactionLoadRequest
List<Object> loadedObjects = executeTransactionLoad(transactionLoadRequest);
Forum loadedDynamoDBForum = (Forum) loadedObjects.get(0);
System.out.println("Forum: " + loadedDynamoDBForum.name);
System.out.println("Threads: " + loadedDynamoDBForum.threads);
Thread loadedDynamodbForumThread = (Thread) loadedObjects.get(1);
System.out.println("Subject: " + loadedDynamodbForumThread.subject);
System.out.println("Message: " + loadedDynamodbForumThread.message);
}

private static void testTransactionLoadWithTransactionWrite() {
    // Create new Forum item for DynamoDB using save
    Forum dynamodbForum = new Forum();
    dynamodbForum.name = "DynamoDB New Forum";
    dynamodbForum.category = "Amazon Web Services";
    dynamodbForum.threads = 0;
    mapper.save(dynamodbForum);

    // Update Forum item for DynamoDB and add a thread to DynamoDB Forum, in
    // an ACID manner using transactionWrite

    dynamodbForum.threads = 1;
    Thread dynamodbForumThread = new Thread();
    dynamodbForumThread.forumName = "DynamoDB New Forum";
    dynamodbForumThread.subject = "Sample Subject 2";
    dynamodbForumThread.message = "Sample Question 2";
    TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
    transactionWriteRequest.addPut(dynamodbForumThread);
    transactionWriteRequest.addUpdate(dynamodbForum);
    executeTransactionWrite(transactionWriteRequest);

    // Read DynamoDB Forum item and Thread item at the same time in a serializable
manner
    TransactionLoadRequest transactionLoadRequest = new TransactionLoadRequest();

    // Read entire item for DynamoDB Forum
    transactionLoadRequest.addLoad(dynamodbForum);

    // Only read subject and message attributes from Thread item
    DynamoDBTransactionLoadExpression loadExpressionForThread = new
DynamoDBTransactionLoadExpression()

    .withProjectionExpression("Subject, Message");
    transactionLoadRequest.addLoad(dynamodbForumThread, loadExpressionForThread);

    // Loaded objects are guaranteed to be in same order as the order in which they are
    // added to TransactionLoadRequest
    List<Object> loadedObjects = executeTransactionLoad(transactionLoadRequest);
    Forum loadedDynamoDBForum = (Forum) loadedObjects.get(0);
    System.out.println("Forum: " + loadedDynamoDBForum.name);
    System.out.println("Threads: " + loadedDynamoDBForum.threads);
    Thread loadedDynamodbForumThread = (Thread) loadedObjects.get(1);
    System.out.println("Subject: " + loadedDynamodbForumThread.subject);
    System.out.println("Message: " + loadedDynamodbForumThread.message);
}

private static void testPutAndUpdateInTransactionWrite() {
    // Create new Forum item for S3 using save
    Forum s3Forum = new Forum();
    s3Forum.name = "S3 Forum";
    s3Forum.category = "Core Amazon Web Services";
    s3Forum.threads = 0;
    mapper.save(s3Forum);
}

```

```

    // Update Forum item for S3 and Create new Forum item for DynamoDB using
transactionWrite
    s3Forum.category = "Amazon Web Services";
    Forum dynamodbForum = new Forum();
    dynamodbForum.name = "DynamoDB Forum";
    dynamodbForum.category = "Amazon Web Services";
    dynamodbForum.threads = 0;
    TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
    transactionWriteRequest.addUpdate(s3Forum);
    transactionWriteRequest.addPut(dynamodbForum);
    executeTransactionWrite(transactionWriteRequest);
}

private static void testPutWithConditionalUpdateInTransactionWrite() {
    // Create new Thread item for DynamoDB forum and update thread count in DynamoDB
forum
    // if the DynamoDB Forum exists
    Thread dynamodbForumThread = new Thread();
    dynamodbForumThread.forumName = "DynamoDB Forum";
    dynamodbForumThread.subject = "Sample Subject 1";
    dynamodbForumThread.message = "Sample Question 1";

    Forum dynamodbForum = new Forum();
    dynamodbForum.name = "DynamoDB Forum";
    dynamodbForum.category = "Amazon Web Services";
    dynamodbForum.threads = 1;

    DynamoDBTransactionWriteExpression transactionWriteExpression = new
DynamoDBTransactionWriteExpression()

    .withConditionExpression("attribute_exists(Category)");

    TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
    transactionWriteRequest.addPut(dynamodbForumThread);
    transactionWriteRequest.addUpdate(dynamodbForum, transactionWriteExpression);
    executeTransactionWrite(transactionWriteRequest);
}

private static void testPutWithConditionCheckInTransactionWrite() {
    // Create new Thread item for DynamoDB forum and update thread count in DynamoDB
forum if a thread already exists
    Thread dynamodbForumThread2 = new Thread();
    dynamodbForumThread2.forumName = "DynamoDB Forum";
    dynamodbForumThread2.subject = "Sample Subject 2";
    dynamodbForumThread2.message = "Sample Question 2";

    Thread dynamodbForumThread1 = new Thread();
    dynamodbForumThread1.forumName = "DynamoDB Forum";
    dynamodbForumThread1.subject = "Sample Subject 1";
    DynamoDBTransactionWriteExpression conditionExpressionForConditionCheck = new
DynamoDBTransactionWriteExpression()

    .withConditionExpression("attribute_exists(Subject)");

    Forum dynamodbForum = new Forum();
    dynamodbForum.name = "DynamoDB Forum";
    dynamodbForum.category = "Amazon Web Services";
    dynamodbForum.threads = 2;

    TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
    transactionWriteRequest.addPut(dynamodbForumThread2);
    transactionWriteRequest.addConditionCheck(dynamodbForumThread1,
conditionExpressionForConditionCheck);
    transactionWriteRequest.addUpdate(dynamodbForum);
    executeTransactionWrite(transactionWriteRequest);
}

```

```

private static void testMixedOperationsInTransactionWrite() {
    // Create new Thread item for S3 forum and delete "Sample Subject 1" Thread from
DynamoDB forum if
    // "Sample Subject 2" Thread exists in DynamoDB forum
    Thread s3ForumThread = new Thread();
    s3ForumThread.forumName = "S3 Forum";
    s3ForumThread.subject = "Sample Subject 1";
    s3ForumThread.message = "Sample Question 1";

    Forum s3Forum = new Forum();
    s3Forum.name = "S3 Forum";
    s3Forum.category = "Amazon Web Services";
    s3Forum.threads = 1;

    Thread dynamodbForumThread1 = new Thread();
    dynamodbForumThread1.forumName = "DynamoDB Forum";
    dynamodbForumThread1.subject = "Sample Subject 1";

    Thread dynamodbForumThread2 = new Thread();
    dynamodbForumThread2.forumName = "DynamoDB Forum";
    dynamodbForumThread2.subject = "Sample Subject 2";
    DynamoDBTransactionWriteExpression conditionExpressionForConditionCheck = new
DynamoDBTransactionWriteExpression()

    .withConditionExpression("attribute_exists(Subject)");

    Forum dynamodbForum = new Forum();
    dynamodbForum.name = "DynamoDB Forum";
    dynamodbForum.category = "Amazon Web Services";
    dynamodbForum.threads = 1;

    TransactionWriteRequest transactionWriteRequest = new TransactionWriteRequest();
    transactionWriteRequest.addPut(s3ForumThread);
    transactionWriteRequest.addUpdate(s3Forum);
    transactionWriteRequest.addDelete(dynamodbForumThread1);
    transactionWriteRequest.addConditionCheck(dynamodbForumThread2,
conditionExpressionForConditionCheck);
    transactionWriteRequest.addUpdate(dynamodbForum);
    executeTransactionWrite(transactionWriteRequest);
}

private static List<Object> executeTransactionLoad(TransactionLoadRequest
transactionLoadRequest) {
    List<Object> loadedObjects = new ArrayList<Object>();
    try {
        loadedObjects = mapper.transactionLoad(transactionLoadRequest);
    } catch (DynamoDBMappingException ddbme) {
        System.err.println("Client side error in Mapper, fix before retrying. Error: "
+ ddbme.getMessage());
    } catch (ResourceNotFoundException rnfe) {
        System.err.println("One of the tables was not found, verify table exists before
retrying. Error: " + rnfe.getMessage());
    } catch (InternalServerException ise) {
        System.err.println("Internal Server Error, generally safe to retry with back-
off. Error: " + ise.getMessage());
    } catch (TransactionCanceledException tce) {
        System.err.println("Transaction Canceled, implies a client issue, fix before
retrying. Error: " + tce.getMessage());
    } catch (Exception ex) {
        System.err.println("An exception occurred, investigate and configure retry
strategy. Error: " + ex.getMessage());
    }
    return loadedObjects;
}

```

```

    private static void executeTransactionWrite(TransactionWriteRequest transactionWriteRequest) {
        try {
            mapper.transactionWrite(transactionWriteRequest);
        } catch (DynamoDBMappingException ddbme) {
            System.err.println("Client side error in Mapper, fix before retrying. Error: " +
+ ddbme.getMessage());
        } catch (ResourceNotFoundException rnfe) {
            System.err.println("One of the tables was not found, verify table exists before
retrying. Error: " + rnfe.getMessage());
        } catch (InternalServerException ise) {
            System.err.println("Internal Server Error, generally safe to retry with back-
off. Error: " + ise.getMessage());
        } catch (TransactionCanceledException tce) {
            System.err.println("Transaction Canceled, implies a client issue, fix before
retrying. Error: " + tce.getMessage());
        } catch (Exception ex) {
            System.err.println("An exception occurred, investigate and configure retry
strategy. Error: " + ex.getMessage());
        }
    }

    @DynamoDBTable(tableName = "Thread")
    public static class Thread {
        private String forumName;
        private String subject;
        private String message;
        private String lastPostedDateTime;
        private String lastPostedBy;
        private Set<String> tags;
        private int answered;
        private int views;
        private int replies;

        // Partition key
        @DynamoDBHashKey(attributeName = "ForumName")
        public String getForumName() {
            return forumName;
        }

        public void setForumName(String forumName) {
            this.forumName = forumName;
        }

        // Sort key
        @DynamoDBRangeKey(attributeName = "Subject")
        public String getSubject() {
            return subject;
        }

        public void setSubject(String subject) {
            this.subject = subject;
        }

        @DynamoDBAttribute(attributeName = "Message")
        public String getMessage() {
            return message;
        }

        public void setMessage(String message) {
            this.message = message;
        }

        @DynamoDBAttribute(attributeName = "LastPostedDateTime")
        public String getLastPostedDateTime() {
            return lastPostedDateTime;
        }
    }
}

```

```

        }

    public void setLastPostedDateTime(String lastPostedDateTime) {
        this.lastPostedDateTime = lastPostedDateTime;
    }

    @DynamoDBAttribute(attributeName = "LastPostedBy")
    public String getLastPostedBy() {
        return lastPostedBy;
    }

    public void setLastPostedBy(String lastPostedBy) {
        this.lastPostedBy = lastPostedBy;
    }

    @DynamoDBAttribute(attributeName = "Tags")
    public Set<String> getTags() {
        return tags;
    }

    public void setTags(Set<String> tags) {
        this.tags = tags;
    }

    @DynamoDBAttribute(attributeName = "Answered")
    public int getAnswered() {
        return answered;
    }

    public void setAnswered(int answered) {
        this.answered = answered;
    }

    @DynamoDBAttribute(attributeName = "Views")
    public int getViews() {
        return views;
    }

    public void setViews(int views) {
        this.views = views;
    }

    @DynamoDBAttribute(attributeName = "Replies")
    public int getReplies() {
        return replies;
    }

    public void setReplies(int replies) {
        this.replies = replies;
    }

}

{@DynamoDBTable(tableName = "Forum")
public static class Forum {
    private String name;
    private String category;
    private int threads;

    // Partition key
    @DynamoDBHashKey(attributeName = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
}
}

```

```

        this.name = name;
    }

    @DynamoDBAttribute(attributeName = "Category")
    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    @DynamoDBAttribute(attributeName = "Threads")
    public int getThreads() {
        return threads;
    }

    public void setThreads(int threads) {
        this.threads = threads;
    }
}
}

```

Optimistic Locking with Version Number

Optimistic locking is a strategy to ensure that the client-side item that you are updating (or deleting) is the same as the item in Amazon DynamoDB. If you use this strategy, your database writes are protected from being overwritten by the writes of others, and vice versa.

With optimistic locking, each item has an attribute that acts as a version number. If you retrieve an item from a table, the application records the version number of that item. You can update the item, but only if the version number on the server side has not changed. If there is a version mismatch, it means that someone else has modified the item before you did. The update attempt fails, because you have a stale version of the item. If this happens, you simply try again by retrieving the item and then trying to update it. Optimistic locking prevents you from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your changes.

To support optimistic locking, the AWS SDK for Java provides the `@DynamoDBVersionAttribute` annotation. In the mapping class for your table, you designate one property to store the version number, and mark it using this annotation. When you save an object, the corresponding item in the DynamoDB table will have an attribute that stores the version number. The `DynamoDBMapper` assigns a version number when you first save the object, and it automatically increments the version number each time you update the item. Your update or delete requests succeed only if the client-side object version matches the corresponding version number of the item in the DynamoDB table.

`ConditionalCheckFailedException` is thrown if:

- You use optimistic locking with `@DynamoDBVersionAttribute` and the version value on the server is different from the value on the client side.
- You specify your own conditional constraints while saving data by using `DynamoDBMapper` with `DynamoDBSaveExpression` and these constraints failed.

Note

- DynamoDB global tables use a “last writer wins” reconciliation between concurrent updates. If you use global tables, last writer policy wins. So in this case, the locking strategy does not work as expected.
- `DynamoDBMapper` transactional write operations do not support `@DynamoDBVersionAttribute` annotation and condition expressions within

the same API call. If an object within a transactional write is annotated with `@DynamoDBVersionAttribute` and also has a condition expression, then an `SdkClientException` will be thrown.

For example, the following Java code defines a `CatalogItem` class that has several properties. The `Version` property is tagged with the `@DynamoDBVersionAttribute` annotation.

Example

```

@DynamoDBTable(tableName="ProductCatalog")
public class CatalogItem {

    private Integer id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private String someProp;
    private Long version;

    @DynamoDBHashKey(attributeName="Id")
    public Integer getId() { return id; }
    public void setId(Integer Id) { this.id = Id; }

    @DynamoDBAttribute(attributeName="Title")
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    @DynamoDBAttribute(attributeName="ISBN")
    public String getISBN() { return ISBN; }
    public void setISBN(String ISBN) { this.ISBN = ISBN; }

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() { return bookAuthors; }
    public void setBookAuthors(Set<String> bookAuthors) { this.bookAuthors = bookAuthors; }

    @DynamoDBIgnore
    public String getSomeProp() { return someProp; }
    public void setSomeProp(String someProp) { this.someProp = someProp; }

    @DynamoDBVersionAttribute
    public Long getVersion() { return version; }
    public void setVersion(Long version) { this.version = version; }
}

```

You can apply the `@DynamoDBVersionAttribute` annotation to nullable types provided by the primitive wrappers classes that provide a nullable type, such as `Long` and `Integer`.

Optimistic locking has the following impact on these `DynamoDBMapper` methods:

- **save** — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the `save` operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- **delete** — The `delete` method takes an object as a parameter, and the `DynamoDBMapper` performs a version check before deleting the item. The version check can be disabled if `DynamoDBMapperConfig.SaveBehavior.CLOBBER` is specified in the request.

The internal implementation of optimistic locking within `DynamoDBMapper` uses conditional update and conditional delete support provided by DynamoDB.

- **transactionWrite** —

- Put — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the put operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- Update — For a new item, the `DynamoDBMapper` assigns an initial version number of 1. If you retrieve an item, update one or more of its properties, and attempt to save the changes, the update operation succeeds only if the version number on the client side and the server side match. The `DynamoDBMapper` increments the version number automatically.
- Delete — The `DynamoDBMapper` performs a version check before deleting the item. The delete operation succeeds only if the version number on the client side and the server side match.
- ConditionCheck — The `@DynamoDBVersionAttribute` annotation is not supported for ConditionCheck operations. An `SdkClientException` will be thrown when a ConditionCheck item is annotated with `@DynamoDBVersionAttribute`.

Disabling Optimistic Locking

To disable optimistic locking, you can change the `DynamoDBMapperConfig.SaveBehavior` enumeration value from `UPDATE` to `CLOBBER`. You can do this by creating a `DynamoDBMapperConfig` instance that skips version checking and use this instance for all your requests. For information about `DynamoDBMapperConfig.SaveBehavior` and other optional `DynamoDBMapper` parameters, see [Optional Configuration Settings for DynamoDBMapper \(p. 241\)](#).

You can also set locking behavior for a specific operation only. For example, the following Java snippet uses the `DynamoDBMapper` to save a catalog item. It specifies `DynamoDBMapperConfig.SaveBehavior` by adding the optional `DynamoDBMapperConfig` parameter to the `save` method.

Note

The `transactionWrite` method does not support `DynamoDBMapperConfig.SaveBehavior` configuration. Disabling optimistic locking for `transactionWrite` is not supported.

Example

```
DynamoDBMapper mapper = new DynamoDBMapper(client);

// Load a catalog item.
CatalogItem item = mapper.load(CatalogItem.class, 101);
item.setTitle("This is a new title for the item");
...
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

Mapping Arbitrary Data

In addition to the supported Java types (see [Supported Data Types \(p. 228\)](#)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. To map these types, you must provide an implementation that converts your complex type to a DynamoDB supported type and vice versa, and annotate the complex type accessor method using the `@DynamoDBTypeConverted` annotation. The converter code transforms data when objects are saved or loaded. It is also used for all operations that consume complex types. Note that when comparing data during query and scan operations, the comparisons are made against the data stored in DynamoDB.

For example, consider the following `CatalogItem` class that defines a property, `Dimension`, that is of `DimensionType`. This property stores the item dimensions as height, width, and thickness. Assume that you decide to store these item dimensions as a string (such as `8.5x11x.05`) in DynamoDB. The following

example provides converter code that converts the `DimensionType` object to a string and a string to the `DimensionType`.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

Example

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.codesamples.datamodeling;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTypeConverted;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTypeConverter;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

public class DynamoDBMapperExample {

    static AmazonDynamoDB client;

    public static void main(String[] args) throws IOException {

        // Set the AWS region you want to access.
        Regions usWest2 = Regions.US_WEST_2;
        client = AmazonDynamoDBClientBuilder.standard().withRegion(usWest2).build();

        DimensionType dimType = new DimensionType();
        dimType.setHeight("8.00");
        dimType.setLength("11.0");
        dimType.setThickness("1.0");

        Book book = new Book();
        book.setId(502);
        book.setTitle("Book 502");
        book.setISBN("555-5555555555");
        book.setBookAuthors(new HashSet<String>(Arrays.asList("Author1", "Author2")));
        book.setDimensions(dimType);

        DynamoDBMapper mapper = new DynamoDBMapper(client);
        mapper.save(book);
    }
}
```

```
Book bookRetrieved = mapper.load(Book.class, 502);
System.out.println("Book info: " + "\n" + bookRetrieved);

bookRetrieved.getDimensions().setHeight("9.0");
bookRetrieved.getDimensions().setLength("12.0");
bookRetrieved.getDimensions().setThickness("2.0");

mapper.save(bookRetrieved);

bookRetrieved = mapper.load(Book.class, 502);
System.out.println("Updated book info: " + "\n" + bookRetrieved);
}

@DynamoDBTable(tableName = "ProductCatalog")
public static class Book {
    private int id;
    private String title;
    private String ISBN;
    private Set<String> bookAuthors;
    private DimensionType dimensionType;

    // Partition key
    @DynamoDBHashKey(attributeName = "Id")
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @DynamoDBAttribute(attributeName = "Title")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @DynamoDBAttribute(attributeName = "ISBN")
    public String getISBN() {
        return ISBN;
    }

    public void setISBN(String ISBN) {
        this.ISBN = ISBN;
    }

    @DynamoDBAttribute(attributeName = "Authors")
    public Set<String> getBookAuthors() {
        return bookAuthors;
    }

    public void setBookAuthors(Set<String> bookAuthors) {
        this.bookAuthors = bookAuthors;
    }

    @DynamoDBTypeConverted(converter = DimensionTypeConverter.class)
    @DynamoDBAttribute(attributeName = "Dimensions")
    public DimensionType getDimensions() {
        return dimensionType;
    }

    @DynamoDBAttribute(attributeName = "Dimensions")
```

```

public void setDimensions(DimensionType dimensionType) {
    this.dimensionType = dimensionType;
}

@Override
public String toString() {
    return "Book [ISBN=" + ISBN + ", bookAuthors=" + bookAuthors + ",
dimensionType= "
        + dimensionType.getHeight() + " x " + dimensionType.getLength() + " x " +
dimensionType.getThickness()
        + ", Id=" + id + ", Title=" + title + "]";
}
}

static public class DimensionType {

    private String length;
    private String height;
    private String thickness;

    public String getLength() {
        return length;
    }

    public void setLength(String length) {
        this.length = length;
    }

    public String getHeight() {
        return height;
    }

    public void setHeight(String height) {
        this.height = height;
    }

    public String getThickness() {
        return thickness;
    }

    public void setThickness(String thickness) {
        this.thickness = thickness;
    }
}

// Converts the complex type DimensionType to a string and vice-versa.
static public class DimensionTypeConverter implements DynamoDBTypeConverter<String,
DimensionType> {

    @Override
    public String convert(DimensionType object) {
        DimensionType itemDimensions = (DimensionType) object;
        String dimension = null;
        try {
            if (itemDimensions != null) {
                dimension = String.format("%s x %s x %s", itemDimensions.getLength(),
itemDimensions.getHeight(),
                itemDimensions.getThickness());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return dimension;
    }
}

```

```
    @Override
    public DimensionType unconvert(String s) {

        DimensionType itemDimension = new DimensionType();
        try {
            if (s != null && s.length() != 0) {
                String[] data = s.split("x");
                itemDimension.setLength(data[0].trim());
                itemDimension.setHeight(data[1].trim());
                itemDimension.setThickness(data[2].trim());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }

        return itemDimension;
    }
}
```

.NET: Document Model

Topics

- [Operations Not Supported by the Document Model \(p. 273\)](#)
- [Supported Data Types \(p. 273\)](#)
- [Working with Items in DynamoDB Using the AWS SDK for .NET Document Model \(p. 275\)](#)
- [Getting an Item - Table.GetItem \(p. 278\)](#)
- [Deleting an Item - Table.DeleteItem \(p. 279\)](#)
- [Updating an Item - Table.UpdateItem \(p. 280\)](#)
- [Batch Write - Putting and Deleting Multiple Items \(p. 281\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Document Model \(p. 283\)](#)
- [Example: Batch Operations Using the AWS SDK for .NET Document Model API \(p. 286\)](#)
- [Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model \(p. 288\)](#)

The AWS SDK for .NET provides document model classes that wrap some of the low-level Amazon DynamoDB operations, further simplifying your coding. In the document model, the primary classes are `Table` and `Document`. The `Table` class provides data operation methods such as `PutItem`, `GetItem`, and `DeleteItem`. It also provides the `Query` and the `Scan` methods. The `Document` class represents a single item in a table.

The preceding document model classes are available in the `Amazon.DynamoDBv2.DocumentModel` namespace.

Operations Not Supported by the Document Model

You can't use the document model classes to create, update, and delete tables. However, the document model does support most common data operations.

Supported Data Types

The document model supports a set of primitive .NET data types and collections data types. The model supports the following primitive data types.

- `bool`
- `byte`

- `char`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Guid`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `string`
- `UInt16`
- `UInt32`
- `UInt64`

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

| .NET primitive type | DynamoDB type |
|--|---|
| All number types | <code>N</code> (number type) |
| All string types | <code>S</code> (string type) |
| <code>MemoryStream</code> , <code>byte[]</code> | <code>B</code> (binary type) |
| <code>bool</code> | <code>N</code> (number type). 0 represents false and 1 represents true. |
| <code>DateTime</code> | <code>S</code> (string type). The <code>DateTime</code> values are stored as ISO-8601 formatted strings. |
| <code>Guid</code> | <code>S</code> (string type). |
| Collection types (<code>List</code> , <code>HashSet</code> , and <code>array</code>) | <code>BS</code> (binary set) type, <code>SS</code> (string set) type, and <code>NS</code> (number set) type |

AWS .NET SDK defines types for mapping DynamoDB's Boolean, null, list and map types to .NET document model API:

- Use `DynamoDBBool` for Boolean type.
- Use `DynamoDBNull` for null type.
- Use `DynamoDBList` for list type.
- Use `Document` for map type.

Note

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

Working with Items in DynamoDB Using the AWS SDK for .NET Document Model

Topics

- [Putting an Item - Table.PutItem Method \(p. 276\)](#)
- [Specifying Optional Parameters \(p. 277\)](#)

To perform data operations using the document model, you must first call the `Table.LoadTable` method, which creates an instance of the `Table` class that represents a specific table. The following C# example creates a `Table` object that represents the `ProductCatalog` table in Amazon DynamoDB.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
```

Note

In general, you use the `LoadTable` method once at the beginning of your application because it makes a `DescribeTable` call that adds to the round trip to DynamoDB.

You can then use the `Table` object to perform various data operations. Each data operation has two types of overloads: One takes the minimum required parameters and the other takes optional, operation-specific configuration information. For example, to retrieve an item, you must provide the table's primary key value, in which case you can use the following `GetItem` overload.

Example

```
// Get the item from a table that has a primary key that is composed of only a partition key.  
Table.GetItem(Primitive partitionKey);  
// Get the item from a table whose primary key is composed of both a partition key and sort key.  
Table.GetItem(Primitive partitionKey, Primitive sortKey);
```

You also can pass optional parameters to these methods. For example, the preceding `GetItem` returns the entire item including all its attributes. You can optionally specify a list of attributes to retrieve. In this case, you use the following `GetItem` overload that takes in the operation-specific configuration object parameter.

Example

```
// Configuration object that specifies optional parameters.  
GetItemOperationConfig config = new GetItemOperationConfig()  
{  
    AttributesToGet = new List<string>() { "Id", "Title" },  
};  
// Pass in the configuration to the GetItem method.  
// 1. Table that has only a partition key as primary key.  
Table.GetItem(Primitive partitionKey, GetItemOperationConfig config);  
// 2. Table that has both a partition key and a sort key.  
Table.GetItem(Primitive partitionKey, Primitive sortKey, GetItemOperationConfig config);
```

You can use the configuration object to specify several optional parameters such as request a specific list of attributes or specify the page size (number of items per page). Each data operation method has its own configuration class. For example, you can use the `GetItemOperationConfig` class to provide options for the `GetItem` operation. You can use the `PutItemOperationConfig` class to provide optional parameters for the `PutItem` operation.

The following sections discuss each of the data operations that are supported by the `Table` class.

Putting an Item - `Table.PutItem` Method

The `PutItem` method uploads the input `Document` instance to the table. If an item that has a primary key that is specified in the input `Document` exists in the table, the `PutItem` operation replaces the entire existing item. The new item is identical to the `Document` object that you provided to the `PutItem` method. If your original item had any extra attributes, they are no longer present in the new item.

The following are the steps to put a new item into a table using the AWS SDK for .NET document model.

1. Execute the `Table.LoadTable` method that provides the table name in which you want to put an item.
2. Create a `Document` object that has a list of attribute names and their values.
3. Execute `Table.PutItem` by providing the `Document` instance as a parameter.

The following C# code example demonstrates the preceding tasks. The example uploads an item to the `ProductCatalog` table.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;
book["Title"] = "Book 101 Title";
book["ISBN"] = "11-11-11-11";
book["Authors"] = new List<string> { "Author 1", "Author 2" };
book["InStock"] = new DynamoDBBool(true);
book["QuantityOnHand"] = new DynamoDBNull();

table.PutItem(book);
```

In the preceding example, the `Document` instance creates an item that has `Number`, `String`, `String Set`, `Boolean`, and `Null` attributes. (`Null` is used to indicate that the `QuantityOnHand` for this product is unknown.) For `Boolean` and `Null`, use the constructor methods `DynamoDBBool` and `DynamoDBNull`.

In DynamoDB, the `List` and `Map` data types can contain elements composed of other data types. Here is how to map these data types to the document model API:

- `List` — use the `DynamoDBList` constructor.
- `Map` — use the `Document` constructor.

You can modify the preceding example to add a `List` attribute to the item. To do this, use a `DynamoDBList` constructor, as shown in the following code example.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;

/*other attributes omitted for brevity...*/

var relatedItems = new DynamoDBList();
relatedItems.Add(341);
relatedItems.Add(472);
relatedItems.Add(649);
```

```
item.Add("RelatedItems", relatedItems);

table.PutItem(book);
```

To add a `Map` attribute to the book, you define another `Document`. The following code example illustrates how to do this.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 101;

/*other attributes omitted for brevity...*/

var pictures = new Document();
pictures.Add("FrontView", "http://example.com/products/101_front.jpg" );
pictures.Add("RearView", "http://example.com/products/101_rear.jpg" );

book.Add("Pictures", pictures);

table.PutItem(book);
```

These examples are based on the item shown in [Specifying Item Attributes When Using Expressions \(p. 385\)](#). The document model lets you create complex nested attributes, such as the `ProductReviews` attribute shown in the case study.

Specifying Optional Parameters

You can configure optional parameters for the `PutItem` operation by adding the `PutItemOperationConfig` parameter. For a complete list of optional parameters, see [PutItem](#). The following C# code example puts an item in the `ProductCatalog` table. It specifies the following optional parameter:

- The `ConditionalExpression` parameter to make this a conditional put request. The example creates an expression that specifies the `ISBN` attribute must have a specific value that has to be present in the item that you are replacing.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();
book["Id"] = 555;
book["Title"] = "Book 555 Title";
book["Price"] = "25.00";
book["ISBN"] = "55-55-55-55";
book["Name"] = "Item 1 updated";
book["Authors"] = new List<string> { "Author x", "Author y" };
book["InStock"] = new DynamodDBBool(true);
book["QuantityOnHand"] = new DynamoDBNull();

// Create a condition expression for the optional conditional put operation.
Expression expr = new Expression();
expr.ExpressionStatement = "ISBN = :val";
expr.ExpressionAttributeValues[":val"] = "55-55-55-55";

PutItemOperationConfig config = new PutItemOperationConfig()
{
    // Optional parameter.
```

```

        ConditionalExpression = expr
};

table.PutItem(book, config);

```

Getting an Item - Table.GetItem

The `GetItem` operation retrieves an item as a `Document` instance. You must provide the primary key of the item that you want to retrieve as shown in the following C# code example.

Example

```

Table table = Table.LoadTable(client, "ProductCatalog");
Document document = table.GetItem(101); // Primary key 101.

```

The `GetItem` operation returns all the attributes of the item and performs an eventually consistent read (see [Read Consistency \(p. 16\)](#)) by default.

Specifying Optional Parameters

You can configure additional options for the `GetItem` operation by adding the `GetItemOperationConfig` parameter. For a complete list of optional parameters, see [GetItem](#). The following C# code example retrieves an item from the `ProductCatalog` table. It specifies the `GetItemOperationConfig` to provide the following optional parameters:

- The `AttributesToGet` parameter to retrieve only the specified attributes.
- The `ConsistentRead` parameter to request the latest values for all the specified attributes. To learn more about data consistency, see [Read Consistency \(p. 16\)](#).

Example

```

Table table = Table.LoadTable(client, "ProductCatalog");

GetItemOperationConfig config = new GetItemOperationConfig()
{
    AttributesToGet = new List<string>() { "Id", "Title", "Authors", "InStock",
    "QuantityOnHand" },
    ConsistentRead = true
};
Document doc = table.GetItem(101, config);

```

When you retrieve an item using the document model API, you can access individual elements within the `Document` object is returned, as shown in the following example.

Example

```

int id = doc["Id"].AsInt();
string title = doc["Title"].AsString();
List<string> authors = doc["Authors"].AsListOfString();
bool inStock = doc["InStock"].AsBoolean();
DynamoDBNull quantityOnHand = doc["QuantityOnHand"].AsDynamoDBNull();

```

For attributes that are of type `List` or `Map`, here is how to map these attributes to the document model API:

- `List` — Use the `AsDynamoDBList` method.
- `Map` — Use the `AsDocument` method.

The following code example shows how to retrieve a `List (RelatedItems)` and a `Map (Pictures)` from the `Document` object:

Example

```
DynamoDBList relatedItems = doc["RelatedItems"].AsDynamoDBList();
Document pictures = doc["Pictures"].AsDocument();
```

Deleting an Item - Table.DeleteItem

The `DeleteItem` operation deletes an item from a table. You can pass the item's primary key as a parameter. Or, if you've already read an item and have the corresponding `Document` object, you can pass it as a parameter to the `DeleteItem` method, as shown in the following C# code example.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

// Retrieve a book (a Document instance)
Document document = table.GetItem(111);

// 1) Delete using the Document instance.
table.DeleteItem(document);

// 2) Delete using the primary key.
int partitionKey = 222;
table.DeleteItem(partitionKey)
```

Specifying Optional Parameters

You can configure additional options for the `Delete` operation by adding the `DeleteItemOperationConfig` parameter. For a complete list of optional parameters, see [DeleteTable](#). The following C# code example specifies the two following optional parameters:

- The `ConditionalExpression` parameter to ensure that the book item being deleted has a specific value for the `ISBN` attribute.
- The `ReturnValues` parameter to request that the `Delete` method return the item that it deleted.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
int partitionKey = 111;

Expression expr = new Expression();
expr.ExpressionStatement = "ISBN = :val";
expr.ExpressionAttributeValues[":val"] = "11-11-11-11";

// Specify optional parameters for Delete operation.
DeleteItemOperationConfig config = new DeleteItemOperationConfig
{
    ConditionalExpression = expr,
    ReturnValues = ReturnValues.AllOldAttributes // This is the only supported value when
        using the document model.
};

// Delete the book.
Document d = table.DeleteItem(partitionKey, config);
```

Updating an Item - Table.UpdateItem

The `UpdateItem` operation updates an existing item if it is present. If the item that has the specified primary key is not found, the `UpdateItem` operation adds a new item.

You can use the `UpdateItem` operation to update existing attribute values, add new attributes to the existing collection, or delete attributes from the existing collection. You provide these updates by creating a `Document` instance that describes the updates that you want to perform.

The `UpdateItem` action uses the following guidelines:

- If the item does not exist, `UpdateItem` adds a new item using the primary key that is specified in the input.
- If the item exists, `UpdateItem` applies the updates as follows:
 - Replaces the existing attribute values with the values in the update.
 - If an attribute that you provide in the input does not exist, it adds a new attribute to the item.
 - If the input attribute value is null, it deletes the attributes, if it is present.

Note

This midlevel `UpdateItem` operation does not support the `Add` action (see [UpdateItem](#)) that is supported by the underlying DynamoDB operation.

Note

The `PutItem` operation ([Putting an Item - Table.PutItem Method \(p. 276\)](#)) can also perform an update. If you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified on the `Document` that is being put, the `PutItem` operation deletes those attributes. However, `UpdateItem` only updates the specified input attributes. Any other existing attributes of that item remain unchanged.

The following are the steps to update an item using the AWS SDK for .NET document model:

1. Execute the `Table.LoadTable` method by providing the name of the table in which you want to perform the update operation.
2. Create a `Document` instance by providing all the updates that you want to perform.

To delete an existing attribute, specify the attribute value as null.

3. Call the `Table.UpdateItem` method and provide the `Document` instance as an input parameter.

You must provide the primary key either in the `Document` instance or explicitly as a parameter.

The following C# code example demonstrates the preceding tasks. The code example updates an item in the `Book` table. The `UpdateItem` operation updates the existing `Authors` attribute, deletes the `PageCount` attribute, and adds a new `XYZ` attribute. The `Document` instance includes the primary key of the book to update.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");

var book = new Document();

// Set the attributes that you wish to update.
book["Id"] = 111; // Primary key.
// Replace the authors attribute.
book["Authors"] = new List<string> { "Author x", "Author y" };
// Add a new attribute.
```

```
book["XYZ"] = 12345;
// Delete the existing PageCount attribute.
book["PageCount"] = null;

table.Update(book);
```

Specifying Optional Parameters

You can configure additional options for the `UpdateItem` operation by adding the `UpdateItemOperationConfig` parameter. For a complete list of optional parameters, see [UpdateItem](#).

The following C# code example updates a book item price to 25. It specifies the two following optional parameters:

- The `ConditionalExpression` parameter that identifies the `Price` attribute with value 20 that you expect to be present.
- The `ReturnValues` parameter to request the `UpdateItem` operation to return the item that is updated.

Example

```
Table table = Table.LoadTable(client, "ProductCatalog");
string partitionKey = "111";

var book = new Document();
book["Id"] = partitionKey;
book["Price"] = 25;

Expression expr = new Expression();
expr.ExpressionStatement = "Price = :val";
expr.ExpressionAttributeValues[":val"] = "20";

UpdateOperationConfig config = new UpdateOperationConfig()
{
    ConditionalExpression = expr,
    ReturnValues = ReturnValues.AllOldAttributes
};

Document d1 = table.Update(book, config);
```

Batch Write - Putting and Deleting Multiple Items

Batch write refers to putting and deleting multiple items in a batch. The operation enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to put or delete multiple items from a table using the AWS SDK for .NET document model API.

1. Create a `Table` object by executing the `Table.LoadTable` method by providing the name of the table in which you want to perform the batch operation.
2. Execute the `CreateBatchWrite` method on the table instance you created in the preceding step and create a `DocumentBatchWrite` object.
3. Use the `DocumentBatchWrite` object methods to specify the documents that you want to upload or delete.
4. Call the `DocumentBatchWrite.Execute` method to execute the batch operation.

When using the document model API, you can specify any number of operations in a batch. However, DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information about the specific limits, see [BatchWriteItem](#). If the document model API detects that your batch write request exceeded the number of allowed write requests, or the HTTP

payload size of a batch exceeded the limit allowed by `BatchWriteItem`, it breaks the batch into several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the document model API automatically sends another batch request with those unprocessed items.

The following C# code example demonstrates the preceding steps. The example uses batch write operation to perform two writes; upload a book item and delete another book item.

```
Table productCatalog = Table.LoadTable(client, "ProductCatalog");
var batchWrite = productCatalog.CreateBatchWrite();

var book1 = new Document();
book1["Id"] = 902;
book1["Title"] = "My book1 in batch write using .NET document model";
book1["Price"] = 10;
book1["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
book1["InStock"] = new DynamoDBBool(true);
book1["QuantityOnHand"] = 5;

batchWrite.AddDocumentToPut(book1);
// specify delete item using overload that takes PK.
batchWrite.AddKeyToDelete(12345);

batchWrite.Execute();
```

For a working example, see [Example: Batch Operations Using the AWS SDK for .NET Document Model API \(p. 286\)](#).

You can use the `batchWrite` operation to perform put and delete operations on multiple tables. The following are the steps to put or delete multiple items from multiple tables using the AWS SDK for .NET document model.

1. You create a `DocumentBatchWrite` instance for each table in which you want to put or delete multiple items, as described in the preceding procedure.
2. Create an instance of the `MultiTableDocumentBatchWrite` and add the individual `DocumentBatchWrite` objects to it.
3. Execute the `MultiTableDocumentBatchWrite.Execute` method.

The following C# code example demonstrates the preceding steps. The example uses the batch write operation to perform the following write operations:

- Put a new item in the `Forum` table item.
- Put an item in the `Thread` table and delete an item from the same table.

```
// 1. Specify item to add in the Forum table.
Table forum = Table.LoadTable(client, "Forum");
var forumBatchWrite = forum.CreateBatchWrite();

var forum1 = new Document();
forum1["Name"] = "Test BatchWrite Forum";
forum1["Threads"] = 0;
forumBatchWrite.AddDocumentToPut(forum1);

// 2a. Specify item to add in the Thread table.
Table thread = Table.LoadTable(client, "Thread");
var threadBatchWrite = thread.CreateBatchWrite();
```

```

var thread1 = new Document();
thread1["ForumName"] = "Amazon S3 forum";
thread1["Subject"] = "My sample question";
thread1["Message"] = "Message text";
thread1["KeywordTags"] = new List<string>{ "Amazon S3", "Bucket" };
threadBatchWrite.AddDocumentToPut(thread1);

// 2b. Specify item to delete from the Thread table.
threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");

// 3. Create multi-table batch.
var superBatch = new MultiTableDocumentBatchWrite();
superBatch.AddBatch(forumBatchWrite);
superBatch.AddBatch(threadBatchWrite);

superBatch.Execute();

```

Example: CRUD Operations Using the AWS SDK for .NET Document Model

The following C# code example performs the following actions:

- Creates a book item in the `ProductCatalog` table.
- Retrieves the book item.
- Updates the book item. The code example shows a normal update that adds new attributes and updates existing attributes. It also shows a conditional update that updates the book price only if the existing price value is as specified in the code.
- Deletes the book item.

For step-by-step instructions to test the following example, see [.NET Code Examples \(p. 332\)](#).

Example

```


/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class MidlevelItemCRUD
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();


```

```

private static string tableName = "ProductCatalog";
// The sample uses the following id PK value to add book item.
private static int sampleBookId = 555;

static void Main(string[] args)
{
    try
    {
        Table productCatalog = Table.LoadTable(client, tableName);
        CreateBookItem(productCatalog);
        RetrieveBook(productCatalog);
        // Couple of sample updates.
        UpdateMultipleAttributes(productCatalog);
        UpdateBookPriceConditionally(productCatalog);

        // Delete.
        DeleteBook(productCatalog);
        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }
    catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
    catch (Exception e) { Console.WriteLine(e.Message); }
}

// Creates a sample book item.
private static void CreateBookItem(Table productCatalog)
{
    Console.WriteLine("\n*** Executing CreateBookItem() ***");
    var book = new Document();
    book["Id"] = sampleBookId;
    book["Title"] = "Book " + sampleBookId;
    book["Price"] = 19.99;
    book["ISBN"] = "111-111111111";
    book["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
    book["PageCount"] = 500;
    book["Dimensions"] = "8.5x11x.5";
    book["InPublication"] = new DynamoDBBool(true);
    book["InStock"] = new DynamoDBBool(false);
    book["QuantityOnHand"] = 0;

    productCatalog.PutItem(book);
}

private static void RetrieveBook(Table productCatalog)
{
    Console.WriteLine("\n*** Executing RetrieveBook() ***");
    // Optional configuration.
    GetItemOperationConfig config = new GetItemOperationConfig
    {
        AttributesToGet = new List<string> { "Id", "ISBN", "Title", "Authors",
        "Price" },
        ConsistentRead = true
    };
    Document document = productCatalog.GetItem(sampleBookId, config);
    Console.WriteLine("RetrieveBook: Printing book retrieved...");
    PrintDocument(document);
}

private static void UpdateMultipleAttributes(Table productCatalog)
{
    Console.WriteLine("\n*** Executing UpdateMultipleAttributes() ***");
    Console.WriteLine("\nUpdating multiple attributes....");
    int partitionKey = sampleBookId;

    var book = new Document();
}

```

```

book["Id"] = partitionKey;
// List of attribute updates.
// The following replaces the existing authors list.
book["Authors"] = new List<string> { "Author x", "Author y" };
book["newAttribute"] = "New Value";
book["ISBN"] = null; // Remove it.

// Optional parameters.
UpdateItemOperationConfig config = new UpdateItemOperationConfig
{
    // Get updated item in response.
    ReturnValues = ReturnValues.AllNewAttributes
};
Document updatedBook = productCatalog.UpdateItem(book, config);
Console.WriteLine("UpdateMultipleAttributes: Printing item after updates ...");
PrintDocument(updatedBook);
}

private static void UpdateBookPriceConditionally(Table productCatalog)
{
    Console.WriteLine("\n*** Executing UpdateBookPriceConditionally() ***");

    int partitionKey = sampleBookId;

    var book = new Document();
    book["Id"] = partitionKey;
    book["Price"] = 29.99;

    // For conditional price update, creating a condition expression.
    Expression expr = new Expression();
    expr.ExpressionStatement = "Price = :val";
    expr.ExpressionAttributeValues[":val"] = 19.00;

    // Optional parameters.
    UpdateItemOperationConfig config = new UpdateItemOperationConfig
    {
        ConditionalExpression = expr,
        ReturnValues = ReturnValues.AllNewAttributes
    };
    Document updatedBook = productCatalog.UpdateItem(book, config);
    Console.WriteLine("UpdateBookPriceConditionally: Printing item whose price was
conditionally updated");
    PrintDocument(updatedBook);
}

private static void DeleteBook(Table productCatalog)
{
    Console.WriteLine("\n*** Executing DeleteBook() ***");
    // Optional configuration.
    DeleteItemOperationConfig config = new DeleteItemOperationConfig
    {
        // Return the deleted item.
        ReturnValues = ReturnValues.AllOldAttributes
    };
    Document document = productCatalog.DeleteItem(sampleBookId, config);
    Console.WriteLine("DeleteBook: Printing deleted just deleted...");
    PrintDocument(document);
}

private static void PrintDocument(Document updatedDocument)
{
    foreach (var attribute in updatedDocument.GetAttributeNames())
    {
        string stringValue = null;
        var value = updatedDocument[attribute];
        if (value is Primitive)
}

```

```
        stringValue = value.AsPrimitive().Value.ToString();
    else if (value is PrimitiveList)
        stringValue = string.Join(", ", (from primitive
            in value.AsPrimitiveList().Entries
            select primitive.Value).ToArray());
    Console.WriteLine("{0} - {1}", attribute, stringValue);
}
}
}
```

Example: Batch Operations Using the AWS SDK for .NET Document Model API

Topics

- [Example: Batch Write Using the AWS SDK for .NET Document Model \(p. 286\)](#)

Example: Batch Write Using the AWS SDK for .NET Document Model

The following C# code example illustrates single table and multi-table batch write operations. The example performs the following tasks:

- Illustrates a single table batch write. It adds two items to the `ProductCatalog` table.
- Illustrates a multi-table batch write. It adds an item to both the `Forum` and `Thread` tables and deletes an item from the `Thread` table.

If you followed the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you already have the `ProductCatalog`, `Forum`, and `Thread` tables created. You can also create these sample tables programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#). For step-by-step instructions for testing the following example, see [.NET Code Examples \(p. 332\)](#).

Example

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class MidLevelBatchWriteItem
    {
```

```

private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
static void Main(string[] args)
{
    try
    {
        SingleTableBatchWrite();
        MultiTableBatchWrite();
    }
    catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
    catch (Exception e) { Console.WriteLine(e.Message); }

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}

private static void SingleTableBatchWrite()
{
    Table productCatalog = Table.LoadTable(client, "ProductCatalog");
    var batchWrite = productCatalog.CreateBatchWrite();

    var book1 = new Document();
    book1["Id"] = 902;
    book1["Title"] = "My book1 in batch write using .NET helper classes";
    book1["ISBN"] = "902-11-111111";
    book1["Price"] = 10;
    book1["ProductCategory"] = "Book";
    book1["Authors"] = new List<string> { "Author 1", "Author 2", "Author 3" };
    book1["Dimensions"] = "8.5x11x.5";
    book1["InStock"] = new DynamoDBBool(true);
    book1["QuantityOnHand"] = new DynamoDBNull(); //Quantity is unknown at this
time

    batchWrite.AddDocumentToPut(book1);
    // Specify delete item using overload that takes PK.
    batchWrite.AddKeyToDelete(12345);
    Console.WriteLine("Performing batch write in SingleTableBatchWrite()");
    batchWrite.Execute();
}

private static void MultiTableBatchWrite()
{
    // 1. Specify item to add in the Forum table.
    Table forum = Table.LoadTable(client, "Forum");
    var forumBatchWrite = forum.CreateBatchWrite();

    var forum1 = new Document();
    forum1["Name"] = "Test BatchWrite Forum";
    forum1["Threads"] = 0;
    forumBatchWrite.AddDocumentToPut(forum1);

    // 2a. Specify item to add in the Thread table.
    Table thread = Table.LoadTable(client, "Thread");
    var threadBatchWrite = thread.CreateBatchWrite();

    var thread1 = new Document();
    thread1["ForumName"] = "S3 forum";
    thread1["Subject"] = "My sample question";
    thread1["Message"] = "Message text";
    thread1["KeywordTags"] = new List<string> { "S3", "Bucket" };
    threadBatchWrite.AddDocumentToPut(thread1);

    // 2b. Specify item to delete from the Thread table.
    threadBatchWrite.AddKeyToDelete("someForumName", "someSubject");
}

```

```
// 3. Create multi-table batch.  
var superBatch = new MultiTableDocumentBatchWrite();  
superBatch.AddBatch(forumBatchWrite);  
superBatch.AddBatch(threadBatchWrite);  
Console.WriteLine("Performing batch write in MultiTableBatchWrite()");  
superBatch.Execute();  
}  
}
```

Querying Tables in DynamoDB Using the AWS SDK for .NET Document Model

Topics

- [Table.Query Method in the AWS SDK for .NET \(p. 288\)](#)
- [Table.Scan Method in the AWS SDK for .NET \(p. 292\)](#)

Table.Query Method in the AWS SDK for .NET

The `Query` method enables you to query your tables. You can only query the tables that have a composite primary key (partition key and sort key). If your table's primary key is made of only a partition key, then the `Query` operation is not supported. By default, `Query` internally performs queries that are eventually consistent. To learn about the consistency model, see [Read Consistency \(p. 16\)](#).

The `Query` method provides two overloads. The minimum required parameters to the `Query` method are a partition key value and a sort key filter. You can use the following overload to provide these minimum required parameters.

Example

```
Query(Primitive partitionKey, RangeFilter Filter);
```

For example, the following C# code queries for all forum replies that were posted in the last 15 days.

Example

```
string tableName = "Reply";  
Table table = Table.LoadTable(client, tableName);  
  
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);  
RangeFilter filter = new RangeFilter(QueryOperator.GreaterThan, twoWeeksAgoDate);  
Search search = table.Query("DynamoDB Thread 2", filter);
```

This creates a `Search` object. You can now call the `Search.GetNextSet` method iteratively to retrieve one page of results at a time, as shown in the following C# code example. The code prints the attribute values for each item that the query returns.

Example

```
List<Document> documentSet = new List<Document>();  
do  
{  
    documentSet = search.GetNextSet();  
    foreach (var document in documentSet)  
        PrintDocument(document);
```

```

} while (!search.IsDone);

    private static void PrintDocument(Document document)
{
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
            stringValue = value.AsPrimitive().Value;
        else if (value is PrimitiveList)
            stringValue = string.Join(", ", (from primitive
                                              in value.AsPrimitiveList().Entries
                                              select primitive.Value).ToArray());
        Console.WriteLine("{0} - {1}", attribute, stringValue);
    }
}

```

Specifying Optional Parameters

You can also specify optional parameters for `Query`, such as specifying a list of attributes to retrieve, strongly consistent reads, page size, and the number of items returned per page. For a complete list of parameters, see [Query](#). To specify optional parameters, you must use the following overload in which you provide the `QueryOperationConfig` object.

Example

```
Query(QueryOperationConfig config);
```

Assume that you want to execute the query in the preceding example (retrieve forum replies posted in the last 15 days). However, assume that you want to provide optional query parameters to retrieve only specific attributes and also request a strongly consistent read. The following C# code example constructs the request using the `QueryOperationConfig` object.

Example

```

Table table = Table.LoadTable(client, "Reply");
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
QueryOperationConfig config = new QueryOperationConfig()
{
    HashKey = "DynamoDB Thread 2", //Partition key
    AttributesToGet = new List<string>
    {
        "Subject", "ReplyDateTime", "PostedBy"
    },
    ConsistentRead = true,
    Filter = new RangeFilter(QueryOperator.GreaterThan, twoWeeksAgoDate)
};

Search search = table.Query(config);

```

Example: Query Using the Table.Query Method

The following C# code example uses the `Table.Query` method to execute the following sample queries.

- The following queries are executed against the `Reply` table.
 - Find forum thread replies that were posted in the last 15 days.

This query is executed twice. In the first `Table.Query` call, the example provides only the required query parameters. In the second `Table.Query` call, you provide optional query parameters to request a strongly consistent read and a list of attributes to retrieve.

- Find forum thread replies posted during a period of time.

This query uses the `Between` query operator to find replies posted in between two dates.

- Get a product from the `ProductCatalog` table.

Because the `ProductCatalog` table has a primary key that is only a partition key, you can only get items; you cannot query the table. The example retrieves a specific product item using the item ID.

Example

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
using System;
using System.Collections.Generic;
using System.Linq;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class MidLevelQueryAndScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                // Query examples.
                Table replyTable = Table.LoadTable(client, "Reply");
                string forumName = "Amazon DynamoDB";
                string threadSubject = "DynamoDB Thread 2";
                FindRepliesInLast15Days(replyTable, forumName, threadSubject);
                FindRepliesInLast15DaysWithConfig(replyTable, forumName, threadSubject);
                FindRepliesPostedWithinTimePeriod(replyTable, forumName, threadSubject);

                // Get Example.
                Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
                int productId = 101;
                GetProduct(productCatalogTable, productId);

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }
    }
}
```

```

private static void GetProduct(Table tableName, int productId)
{
    Console.WriteLine("**** Executing GetProduct() ****");
    Document productDocument = tableName.GetItem(productId);
    if (productDocument != null)
    {
        PrintDocument(productDocument);
    }
    else
    {
        Console.WriteLine("Error: product " + productId + " does not exist");
    }
}

private static void FindRepliesInLast15Days(Table table, string forumName, string
threadSubject)
{
    string Attribute = forumName + "#" + threadSubject;

    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
    QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal, partitionKey);
    filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);

    // Use Query overloads that takes the minimum required query parameters.
    Search search = table.Query(filter);

    List<Document> documentSet = new List<Document>();
    do
    {
        documentSet = search.GetNextSet();
        Console.WriteLine("\nFindRepliesInLast15Days: printing .....");
        foreach (var document in documentSet)
            PrintDocument(document);
    } while (!search.IsDone);
}

private static void FindRepliesPostedWithinTimePeriod(Table table, string
forumName, string threadSubject)
{
    DateTime startDate = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0, 0));
    DateTime endDate = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0, 0));

    QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal, forumName + "#"
+ threadSubject);
    filter.AddCondition("ReplyDateTime", QueryOperator.Between, startDate,
endDate);

    QueryOperationConfig config = new QueryOperationConfig()
    {
        Limit = 2, // 2 items/page.
        Select = SelectValues.SpecificAttributes,
        AttributesToGet = new List<string> { "Message",
                                              "ReplyDateTime",
                                              "PostedBy" },
        ConsistentRead = true,
        Filter = filter
    };

    Search search = table.Query(config);

    List<Document> documentList = new List<Document>();

    do
    {
        documentList = search.GetNextSet();

```

```

        Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: printing replies
posted within dates: {0} and {1} ....",
                           startDate, endDate);
        foreach (var document in documentList)
        {
            PrintDocument(document);
        }
    } while (!search.IsDone);
}

private static void FindRepliesInLast15DaysWithConfig(Table table, string
forumName, string threadName)
{
    DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
    QueryFilter filter = new QueryFilter("Id", QueryOperator.Equal, forumName + "#"
+ threadName);
    filter.AddCondition("ReplyDateTime", QueryOperator.GreaterThan,
twoWeeksAgoDate);
    // You are specifying optional parameters so use QueryOperationConfig.
    QueryOperationConfig config = new QueryOperationConfig()
    {
        Filter = filter,
        // Optional parameters.
        Select = SelectValues.SpecifyAttributes,
        AttributesToGet = new List<string> { "Message", "ReplyDateTime",
                                             "PostedBy" },
        ConsistentRead = true
    };

    Search search = table.Query(config);

    List<Document> documentSet = new List<Document>();
    do
    {
        documentSet = search.GetNextSet();
        Console.WriteLine("\nFindRepliesInLast15DaysWithConfig:
printing ....");
        foreach (var document in documentSet)
            PrintDocument(document);
    } while (!search.IsDone);
}

private static void PrintDocument(Document document)
{
    // count++;
    Console.WriteLine();
    foreach (var attribute in document.GetAttributeNames())
    {
        string stringValue = null;
        var value = document[attribute];
        if (value is Primitive)
            stringValue = value.AsPrimitive().Value.ToString();
        else if (value is PrimitiveList)
            stringValue = string.Join(",",
                from primitive
                in value.AsPrimitiveList().Entries
                select primitive.Value).ToArray();
        Console.WriteLine("{0} - {1}", attribute, stringValue);
    }
}
}

```

Table.Scan Method in the AWS SDK for .NET

The Scan method performs a full table scan. It provides two overloads. The only parameter required by the Scan method is the scan filter, which you can provide using the following overload.

Example

```
Scan(ScanFilter filter);
```

For example, assume that you maintain a table of forum threads tracking information such as thread subject (primary), the related message, forum Id to which the thread belongs, Tags, and other information. Assume that the subject is the primary key.

Example

```
Thread(Subject, Message, ForumId, Tags, LastPostedDateTime, .... )
```

This is a simplified version of forums and threads that you see on AWS forums (see [Discussion Forums](#)). The following C# code example queries all threads in a specific forum (ForumId = 101) that are tagged "sortkey". Because the ForumId is not a primary key, the example scans the table. The ScanFilter includes two conditions. The query returns all the threads that satisfy both of the conditions.

Example

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, 101);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "sortkey");

Search search = ThreadTable.Scan(scanFilter);
```

Specifying Optional Parameters

You also can specify optional parameters to Scan, such as a specific list of attributes to retrieve or whether to perform a strongly consistent read. To specify optional parameters, you must create a ScanOperationConfig object that includes both the required and optional parameters and use the following overload.

Example

```
Scan(ScanOperationConfig config);
```

The following C# code example executes the same preceding query (find forum threads in which the ForumId is 101 and the Tag attribute contains the "sortkey" keyword). Assume that you want to add an optional parameter to retrieve only a specific attribute list. In this case, you must create a ScanOperationConfig object by providing all the parameters, required and optional parameters, as shown in the following code example.

Example

```
string tableName = "Thread";
Table ThreadTable = Table.LoadTable(client, tableName);

ScanFilter scanFilter = new ScanFilter();
scanFilter.AddCondition("ForumId", ScanOperator.Equal, forumId);
scanFilter.AddCondition("Tags", ScanOperator.Contains, "sortkey");

ScanOperationConfig config = new ScanOperationConfig()
{
    AttributesToGet = new List<string> { "Subject", "Message" } ,
    Filter = scanFilter
};
```

```
Search search = ThreadTable.Scan(config);
```

Example: Scan Using the Table.Scan Method

The `Scan` operation performs a full table scan making it a potentially expensive operation. You should use queries instead. However, there are times when you might need to execute a scan against a table. For example, you might have a data entry error in the product pricing, and you must scan the table as shown in the following C# code example. The example scans the `ProductCatalog` table to find products for which the price value is less than 0. The example illustrates the use of the two `Table.Scan` overloads.

- `Table.Scan` that takes the `ScanFilter` object as a parameter.

You can pass the `ScanFilter` parameter when passing in only the required parameters.

- `Table.Scan` that takes the `ScanOperationConfig` object as a parameter.

You must use the `ScanOperationConfig` parameter if you want to pass any optional parameters to the `Scan` method.

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.DocumentModel;  
  
namespace com.amazonaws.codesamples  
{  
    class MidLevelScanOnly  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
        static void Main(string[] args)  
        {  
            Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");  
            // Scan example.  
            FindProductsWithNegativePrice(productCatalogTable);  
            FindProductsWithNegativePriceWithConfig(productCatalogTable);  
  
            Console.WriteLine("To continue, press Enter");  
            Console.ReadLine();  
        }  
  
        private static void FindProductsWithNegativePrice(Table productCatalogTable)  
        {  
            // Assume there is a price error. So we scan to find items priced < 0.  
            ScanFilter scanFilter = new ScanFilter();
```

```

        scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

        Search search = productCatalogTable.Scan(scanFilter);

        List<Document> documentList = new List<Document>();
        do
        {
            documentList = search.GetNextSet();
            Console.WriteLine("\nFindProductsWithNegativePrice:
printing .....");
            foreach (var document in documentList)
                PrintDocument(document);
        } while (!search.IsDone);
    }

    private static void FindProductsWithNegativePriceWithConfig(Table
productCatalogTable)
    {
        // Assume there is a price error. So we scan to find items priced < 0.
        ScanFilter scanFilter = new ScanFilter();
        scanFilter.AddCondition("Price", ScanOperator.LessThan, 0);

        ScanOperationConfig config = new ScanOperationConfig()
        {
            Filter = scanFilter,
            Select = SelectValues.SpecificAttributes,
            AttributesToGet = new List<string> { "Title", "Id" }
        };

        Search search = productCatalogTable.Scan(config);

        List<Document> documentList = new List<Document>();
        do
        {
            documentList = search.GetNextSet();
            Console.WriteLine("\nFindProductsWithNegativePriceWithConfig:
printing .....");
            foreach (var document in documentList)
                PrintDocument(document);
        } while (!search.IsDone);
    }

    private static void PrintDocument(Document document)
    {
        // count++;
        Console.WriteLine();
        foreach (var attribute in document.GetAttributeNames())
        {
            string stringValue = null;
            var value = document[attribute];
            if (value is Primitive)
                stringValue = value.AsPrimitive().Value.ToString();
            else if (value is PrimitiveList)
                stringValue = string.Join(", ", (from primitive
                                                in value.AsPrimitiveList().Entries
                                                select primitive.Value).ToArray());
            Console.WriteLine("{0} - {1}", attribute, stringValue);
        }
    }
}

```

.NET: Object Persistence Model

Topics

- [DynamoDB Attributes \(p. 297\)](#)
- [DynamoDBContext Class \(p. 299\)](#)
- [Supported Data Types \(p. 304\)](#)
- [Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 305\)](#)
- [Mapping Arbitrary Data with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 307\)](#)
- [Batch Operations Using the AWS SDK for .NET Object Persistence Model \(p. 310\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Object Persistence Model \(p. 313\)](#)
- [Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model \(p. 315\)](#)
- [Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 319\)](#)

The AWS SDK for .NET provides an object persistence model that enables you to map your client-side classes to Amazon DynamoDB tables. Each object instance then maps to an item in the corresponding tables. To save your client-side objects to the tables, the object persistence model provides the `DynamoDBContext` class, an entry point to DynamoDB. This class provides you a connection to DynamoDB and enables you to access tables, perform various CRUD operations, and execute queries.

The object persistence model provides a set of attributes to map client-side classes to tables, and properties/fields to table attributes.

Note

The object persistence model does not provide an API to create, update, or delete tables. It provides only data operations. You can use only the AWS SDK for .NET low-level API to create, update, and delete tables. For more information, see [Working with DynamoDB Tables in .NET \(p. 367\)](#).

The following example shows how the object persistence model works. It starts with the `ProductCatalog` table. It has `Id` as the primary key.

```
ProductCatalog(Id, ...)
```

Suppose that you have a `Book` class with `Title`, `ISBN`, and `Authors` properties. You can map the `Book` class to the `ProductCatalog` table by adding the attributes defined by the object persistence model, as shown in the following C# code example.

Example

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey]
    public int Id { get; set; }

    public string Title { get; set; }
    public int ISBN { get; set; }

    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors { get; set; }

    [DynamoDBIgnore]
    public string CoverPage { get; set; }
}
```

In the preceding example, the `DynamoDBTable` attribute maps the `Book` class to the `ProductCatalog` table.

The object persistence model supports both the explicit and default mapping between class properties and table attributes.

- **Explicit mapping**—To map a property to a primary key, you must use the `DynamoDBHashKey` and `DynamoDBRangeKey` object persistence model attributes. Additionally, for the nonprimary key attributes, if a property name in your class and the corresponding table attribute to which you want to map it are not the same, you must define the mapping by explicitly adding the `DynamoDBProperty` attribute.

In the preceding example, the `Id` property maps to the primary key with the same name, and the `BookAuthors` property maps to the `Authors` attribute in the `ProductCatalog` table.

- **Default mapping**—By default, the object persistence model maps the class properties to the attributes with the same name in the table.

In the preceding example, the properties `Title` and `ISBN` map to the attributes with the same name in the `ProductCatalog` table.

You don't have to map every single class property. You identify these properties by adding the `DynamoDBIgnore` attribute. When you save a `Book` instance to the table, the `DynamoDBContext` does not include the `CoverPage` property. It also does not return this property when you retrieve the book instance.

You can map properties of .NET primitive types such as `int` and `string`. You also can map any arbitrary data types as long as you provide an appropriate converter to map the arbitrary data to one of the DynamoDB types. To learn about mapping arbitrary types, see [Mapping Arbitrary Data with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 307\)](#).

The object persistence model supports optimistic locking. During an update operation, this ensures that you have the latest copy of the item you are about to update. For more information, see [Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 305\)](#).

DynamoDB Attributes

This section describes the attributes that the object persistence model offers so that you can map your classes and properties to DynamoDB tables and attributes.

Note

In the following attributes, only `DynamoDBTable` and `DynamoDBHashKey` are required.

DynamoDBGlobalSecondaryIndexHashKey

Maps a class property to the partition key of a global secondary index. Use this attribute if you need to `Query` a global secondary index.

DynamoDBGlobalSecondaryIndexRangeKey

Maps a class property to the sort key of a global secondary index. Use this attribute if you need to `Query` a global secondary index and want to refine your results using the index sort key.

DynamoDBHashKey

Maps a class property to the partition key of the table's primary key. The primary key attributes cannot be a collection type.

The following C# code example maps the `Book` class to the `ProductCatalog` table, and the `Id` property to the table's primary key partition key.

```
[DynamoDBTable("ProductCatalog")]
public class Book {
    [DynamoDBHashKey]
    public int Id { get; set; }

    // Additional properties go here.
}
```

DynamoDBIgnore

Indicates that the associated property should be ignored. If you don't want to save any of your class properties, you can add this attribute to instruct `DynamoDBContext` not to include this property when saving objects to the table.

DynamoDBLocalSecondaryIndexRangeKey

Maps a class property to the sort key of a local secondary index. Use this attribute if you need to `Query` a local secondary index and want to refine your results using the index sort key.

DynamoDBProperty

Maps a class property to a table attribute. If the class property maps to a table attribute of the same name, you don't need to specify this attribute. However, if the names are not the same, you can use this tag to provide the mapping. In the following C# statement, the `DynamoDBProperty` maps the `BookAuthors` property to the `Authors` attribute in the table.

```
[DynamoDBProperty("Authors")]
public List<string> BookAuthors { get; set; }
```

`DynamoDBContext` uses this mapping information to create the `Authors` attribute when saving object data to the corresponding table.

DynamoDBRenamable

Specifies an alternative name for a class property. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table where the name of a class property is different from a table attribute.

DynamoDBRangeKey

Maps a class property to the sort key of the table's primary key. If the table has a composite primary key (partition key and sort key), you must specify both the `DynamoDBHashKey` and `DynamoDBRangeKey` attributes in your class mapping.

For example, the sample table `Reply` has a primary key made of the `Id` partition key and `Replenishment` sort key. The following C# code example maps the `Reply` class to the `Reply` table. The class definition also indicates that two of its properties map to the primary key.

For more information about sample tables, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

```
[DynamoDBTable("Reply")]
public class Reply {
```

```
[DynamoDBHashKey]
public int ThreadId { get; set; }
[DynamoDBRangeKey]
public string Replenishment { get; set; }
// Additional properties go here.
}
```

DynamoDBTable

Identifies the target table in DynamoDB to which the class maps. For example, the following C# code example maps the `Developer` class to the `People` table in DynamoDB.

```
[DynamoDBTable("People")]
public class Developer { ... }
```

This attribute can be inherited or overridden.

- The `DynamoDBTable` attribute can be inherited. In the preceding example, if you add a new class, `Lead`, that inherits from the `Developer` class, it also maps to the `People` table. Both the `Developer` and `Lead` objects are stored in the `People` table.
- The `DynamoDBTable` attribute can also be overridden. In the following C# code example, the `Manager` class inherits from the `Developer` class. However, the explicit addition of the `DynamoDBTable` attribute maps the class to another table (`Managers`).

```
[DynamoDBTable("Managers")]
public class Manager : Developer { ... }
```

You can add the optional parameter, `LowerCamelCaseProperties`, to request DynamoDB to make the first letter of the property name lowercase when storing the objects to a table, as shown in the following C# example.

```
[DynamoDBTable("People", LowerCamelCaseProperties=true)]
public class Developer {
    string developerName;
    ...
}
```

When saving instances of the `Developer` class, `DynamoDBContext` saves the `DeveloperName` property as the `developerName`.

DynamoDBVersion

Identifies a class property for storing the item version number. For more information about versioning, see [Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 305\)](#).

DynamoDBContext Class

The `DynamoDBContext` class is the entry point to the Amazon DynamoDB database. It provides a connection to DynamoDB and enables you to access your data in various tables, perform various CRUD operations, and execute queries. The `DynamoDBContext` class provides the following methods.

CreateMultiTableBatchGet

Creates a `MultiTableBatchGet` object, composed of multiple individual `BatchGet` objects. Each of these `BatchGet` objects can be used for retrieving items from a single DynamoDB table.

To retrieve the items from tables, use the `ExecuteBatchGet` method, passing the `MultiTableBatchGet` object as a parameter.

CreateMultiTableBatchWrite

Creates a `MultiTableBatchWrite` object, composed of multiple individual `BatchWrite` objects. Each of these `BatchWrite` objects can be used for writing or deleting items in a single DynamoDB table.

To write to tables, use the `ExecuteBatchWrite` method, passing the `MultiTableBatchWrite` object as a parameter.

CreateBatchGet

Creates a `BatchGet` object that you can use to retrieve multiple items from a table. For more information, see [Batch Get: Getting Multiple Items \(p. 312\)](#).

CreateBatchWrite

Creates a `BatchWrite` object that you can use to put multiple items into a table, or to delete multiple items from a table. For more information, see [Batch Write: Putting and Deleting Multiple Items \(p. 310\)](#).

Delete

Deletes an item from the table. The method requires the primary key of the item you want to delete. You can provide either the primary key value or a client-side object containing a primary key value as a parameter to this method.

- If you specify a client-side object as a parameter and you have enabled optimistic locking, the delete succeeds only if the client-side and the server-side versions of the object match.
- If you specify only the primary key value as a parameter, the delete succeeds regardless of whether you have enabled optimistic locking or not.

Note

To perform this operation in the background, use the `DeleteAsync` method instead.

Dispose

Disposes of all managed and unmanaged resources.

ExecuteBatchGet

Reads data from one or more tables, processing all of the `BatchGet` objects in a `MultiTableBatchGet`.

Note

To perform this operation in the background, use the `ExecuteBatchGetAsync` method instead.

ExecuteBatchWrite

Writes or deletes data in one or more tables, processing all of the `BatchWrite` objects in a `MultiTableBatchWrite`.

Note

To perform this operation in the background, use the `ExecuteBatchWriteAsync` method instead.

FromDocument

Given an instance of a `Document`, the `FromDocument` method returns an instance of a client-side class.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see [.NET: Document Model \(p. 273\)](#).

Suppose that you have a `Document` object named `doc`, that contains a representation of a `Forum` item. (To see how to construct this object, see the description for the `ToDocument` method later in this topic.) You can use `FromDocument` to retrieve the `Forum` item from the `Document`, as shown in the following C# code example.

Example

```
forum101 = context.FromDocument<Forum>(101);
```

Note

If your `Document` object implements the `IEnumerable` interface, you can use the `FromDocuments` method instead. This allows you to iterate over all of the class instances in the `Document`.

FromQuery

Executes a `Query` operation, with the query parameters defined in a `QueryOperationConfig` object.

Note

To perform this operation in the background, use the `FromQueryAsync` method instead.

FromScan

Executes a `Scan` operation, with the scan parameters defined in a `ScanOperationConfig` object.

Note

To perform this operation in the background, use the `FromScanAsync` method instead.

GetTargetTable

Retrieves the target table for the specified type. This is useful if you are writing a custom converter for mapping arbitrary data to a DynamoDB table, and you need to determine which table is associated with a custom data type.

Load

Retrieves an item from a table. The method requires only the primary key of the item you want to retrieve.

By default, DynamoDB returns the item with values that are eventually consistent. For information about the eventual consistency model, see [Read Consistency \(p. 16\)](#).

Note

To perform this operation in the background, use the `LoadAsync` method instead.

Query

Queries a table based on query parameters you provide.

You can query a table only if it has a composite primary key (partition key and sort key). When querying, you must specify a partition key and a condition that applies to the sort key.

Suppose that you have a client-side `Reply` class mapped to the `Reply` table in DynamoDB. The following C# code example queries the `Reply` table to find forum thread replies posted in the past 15 days. The `Reply` table has a primary key that has the `Id` partition key and the `ReplyDateTime` sort key. For more information about the `Reply` table, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

Example

```
DynamoDBContext context = new DynamoDBContext(client);

string replyId = "DynamoDB#DynamoDB Thread 1"; //Partition key
DateTime twoWeeksAgoDate = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0)); // Date to
// compare.
IEnumerable<Reply> latestReplies = context.Query<Reply>(replyId, QueryOperator.GreaterThan,
twoWeeksAgoDate);
```

This returns a collection of `Reply` objects.

The `Query` method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you need to iterate only over the `IEnumerable`.

If your table has a simple primary key (partition key), you can't use the `Query` method. Instead, you can use the `Load` method and provide the partition key to retrieve the item.

Note

To perform this operation in the background, use the `QueryAsync` method instead.

Save

Saves the specified object to the table. If the primary key specified in the input object doesn't exist in the table, the method adds a new item to the table. If the primary key exists, the method updates the existing item.

If you have optimistic locking configured, the update succeeds only if the client and the server-side versions of the item match. For more information, see [Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 305\)](#).

Note

To perform this operation in the background, use the `SaveAsync` method instead.

Scan

Performs an entire table scan.

You can filter scan results by specifying a scan condition. The condition can be evaluated on any attributes in the table. Suppose that you have a client-side class `Book` mapped to the `ProductCatalog` table in DynamoDB. The following C# example scans the table and returns only the book items priced less than 0.

Example

```
IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(
    new ScanCondition("Price", ScanOperator.LessThan, price),
```

```
        new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
    );
```

The `Scan` method returns a "lazy-loaded" `IEnumerable` collection. It initially returns only one page of results, and then makes a service call for the next page if needed. To obtain all the matching items, you only need to iterate over the `IEnumerable`.

For performance reasons, you should query your tables and avoid a table scan.

Note

To perform this operation in the background, use the `ScanAsync` method instead.

ToDocument

Returns an instance of the `Document` document model class from your class instance.

This is helpful if you want to use the document model classes along with the object persistence model to perform any data operations. For more information about the document model classes provided by the AWS SDK for .NET, see [.NET: Document Model \(p. 273\)](#).

Suppose that you have a client-side class mapped to the sample `Forum` table. You can then use a `DynamoDBContext` to get an item as a `Document` object from the `Forum` table, as shown in the following C# code example.

Example

```
DynamoDBContext context = new DynamoDBContext(client);

Forum forum101 = context.Load<Forum>(101); // Retrieve a forum by primary key.
Document doc = context.ToDocument<Forum>(forum101);
```

Specifying Optional Parameters for DynamoDBContext

When using the object persistence model, you can specify the following optional parameters for the `DynamoDBContext`.

- **ConsistentRead**—When retrieving data using the `Load`, `Query`, or `Scan` operations, you can add this optional parameter to request the latest values for the data.
- **IgnoreNullValues**—This parameter informs `DynamoDBContext` to ignore null values on attributes during a `Save` operation. If this parameter is false (or if it is not set), then a null value is interpreted as a directive to delete the specific attribute.
- **SkipVersionCheck**— This parameter informs `DynamoDBContext` not to compare versions when saving or deleting an item. For more information about versioning, see [Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model \(p. 305\)](#).
- **TableNamePrefix**— Prefixes all table names with a specific string. If this parameter is null (or if it is not set), then no prefix is used.

The following C# example creates a new `DynamoDBContext` by specifying two of the preceding optional parameters.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context =
```

```
new DynamoDBContext(client, new DynamoDBContextConfig { ConsistentRead = true,
SkipVersionCheck = true});
```

DynamoDBContext includes these optional parameters with each request that you send using this context.

Instead of setting these parameters at the DynamoDBContext level, you can specify them for individual operations you execute using DynamoDBContext, as shown in the following C# code example. The example loads a specific book item. The Load method of DynamoDBContext specifies the preceding optional parameters.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
...
DynamoDBContext context = new DynamoDBContext(client);
Book bookItem = context.Load<Book>(productId, new DynamoDBContextConfig{ ConsistentRead =
true, SkipVersionCheck = true });
```

In this case, DynamoDBContext includes these parameters only when sending the Get request.

Supported Data Types

The object persistence model supports a set of primitive .NET data types, collections, and arbitrary data types. The model supports the following primitive data types.

- bool
- byte
- char
- DateTime
- decimal
- double
- float
- Int16
- Int32
- Int64
- SByte
- string
- UInt16
- UInt32
- UInt64

The object persistence model also supports the .NET collection types. DynamoDBContext is able to convert concrete collection types and simple Plain Old CLR Objects (POCOs).

The following table summarizes the mapping of the preceding .NET types to the DynamoDB types.

| .NET primitive type | DynamoDB type |
|---------------------|-----------------|
| All number types | N (number type) |
| All string types | S (string type) |

| .NET primitive type | DynamoDB type |
|----------------------|--|
| MemoryStream, byte[] | B (binary type) |
| bool | N (number type). 0 represents false and 1 represents true. |
| Collection types | BS (binary set) type, SS (string set) type, and NS (number set) type |
| DateTime | S (string type). The DateTime values are stored as ISO-8601 formatted strings. |

The object persistence model also supports arbitrary data types. However, you must provide converter code to map the complex types to the DynamoDB types.

Note

- Empty binary values are supported.
- Reading of empty string values is supported. Empty string attribute values are supported within attribute values of string Set type while writing to DynamoDB. Empty string attribute values of string type and empty string values contained within List or Map type are dropped from write requests

Optimistic Locking Using a Version Number with DynamoDB Using the AWS SDK for .NET Object Persistence Model

Optimistic locking support in the object persistence model ensures that the item version for your application is the same as the item version on the server side before updating or deleting the item. Suppose that you retrieve an item for update. However, before you send your updates back, some other application updates the same item. Now your application has a stale copy of the item. Without optimistic locking, any update you perform will overwrite the update made by the other application.

The optimistic locking feature of the object persistence model provides the `DynamoDBVersion` tag that you can use to enable optimistic locking. To use this feature, you add a property to your class for storing the version number. You add the `DynamoDBVersion` attribute to the property. When you first save the object, the `DynamoDBContext` assigns a version number and increments this value each time you update the item.

Your update or delete request succeeds only if the client-side object version matches the corresponding version number of the item on the server side. If your application has a stale copy, it must get the latest version from the server before it can update or delete that item.

The following C# code example defines a `Book` class with object persistence attributes mapping it to the `ProductCatalog` table. The `VersionNumber` property in the class decorated with the `DynamoDBVersion` attribute stores the version number value.

Example

```
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id { get; set; }
    [DynamoDBProperty]
    public string Title { get; set; }
```

```
[DynamoDBProperty]
public string ISBN { get; set; }
[DynamoDBProperty("Authors")]
public List<string> BookAuthors { get; set; }
[DynamoDBVersion]
public int? VersionNumber { get; set; }
}
```

Note

You can apply the `DynamoDBVersion` attribute only to a nullable numeric primitive type (such as `int?`).

Optimistic locking has the following impact on `DynamoDBContext` operations:

- For a new item, `DynamoDBContext` assigns initial version number 0. If you retrieve an existing item, update one or more of its properties, and try to save the changes, the save operation succeeds only if the version number on the client side and the server side match. `DynamoDBContext` increments the version number. You don't need to set the version number.
- The `Delete` method provides overloads that can take either a primary key value or an object as parameter, as shown in the following C# code example.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
...
// Load a book.
Book book = context.Load<ProductCatalog>(111);
// Do other operations.
// Delete 1 - Pass in the book object.
context.Delete<ProductCatalog>(book);

// Delete 2 - Pass in the Id (primary key)
context.Delete<ProductCatalog>(222);
```

If you provide an object as the parameter, the delete succeeds only if the object version matches the corresponding server-side item version. However, if you provide a primary key value as the parameter, `DynamoDBContext` is unaware of any version numbers, and it deletes the item without making the version check.

Note that the internal implementation of optimistic locking in the object persistence model code uses the conditional update and the conditional delete API actions in DynamoDB.

Disabling Optimistic Locking

To disable optimistic locking, you use the `SkipVersionCheck` configuration property. You can set this property when creating `DynamoDBContext`. In this case, optimistic locking is disabled for any requests that you make using the context. For more information, see [Specifying Optional Parameters for DynamoDBContext \(p. 303\)](#).

Instead of setting the property at the context level, you can disable optimistic locking for a specific operation, as shown in the following C# code example. The example uses the context to delete a book item. The `Delete` method sets the optional `SkipVersionCheck` property to true, disabling version checking.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
```

```
// Load a book.  
Book book = context.Load<ProductCatalog>(111);  
...  
// Delete the book.  
context.Delete<Book>(book, new DynamoDBContextConfig { SkipVersionCheck = true });
```

Mapping Arbitrary Data with DynamoDB Using the AWS SDK for .NET Object Persistence Model

In addition to the supported .NET types (see [Supported Data Types \(p. 304\)](#)), you can use types in your application for which there is no direct mapping to the Amazon DynamoDB types. The object persistence model supports storing data of arbitrary types as long as you provide the converter to convert data from the arbitrary type to the DynamoDB type and vice versa. The converter code transforms data during both the saving and loading of the objects.

You can create any types on the client-side. However the data stored in the tables is one of the DynamoDB types, and during query and scan, any data comparisons made are against the data stored in DynamoDB.

The following C# code example defines a Book class with `Id`, `Title`, `ISBN`, and `Dimension` properties. The `Dimension` property is of the `DimensionType` that describes `Height`, `Width`, and `Thickness` properties. The example code provides the converter methods `ToEntry` and `FromEntry` to convert data between the `DimensionType` and the DynamoDB string types. For example, when saving a `Book` instance, the converter creates a book `Dimension` string such as "8.5x11x.05". When you retrieve a book, it converts the string to a `DimensionType` instance.

The example maps the `Book` type to the `ProductCatalog` table. It saves a sample `Book` instance, retrieves it, updates its dimensions, and saves the updated `Book` again.

For step-by-step instructions for testing the following example, see [.NET Code Examples \(p. 332\)](#).

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.DataModel;  
using Amazon.DynamoDBv2.DocumentModel;  
using Amazon.Runtime;  
using Amazon.SecurityToken;  
  
namespace com.amazonaws.codesamples  
{  
    class HighLevelMappingArbitraryData  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
```

```

        static void Main(string[] args)
    {
        try
        {
            DynamoDBContext context = new DynamoDBContext(client);

            // 1. Create a book.
            DimensionType myBookDimensions = new DimensionType()
            {
                Length = 8M,
                Height = 11M,
                Thickness = 0.5M
            };

            Book myBook = new Book
            {
                Id = 501,
                Title = "AWS SDK for .NET Object Persistence Model Handling Arbitrary
Data",
                ISBN = "999-9999999999",
                BookAuthors = new List<string> { "Author 1", "Author 2" },
                Dimensions = myBookDimensions
            };

            context.Save(myBook);

            // 2. Retrieve the book.
            Book bookRetrieved = context.Load<Book>(501);

            // 3. Update property (book dimensions).
            bookRetrieved.Dimensions.Height += 1;
            bookRetrieved.Dimensions.Length += 1;
            bookRetrieved.Dimensions.Thickness += 0.2M;
            // Update the book.
            context.Save(bookRetrieved);

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }
        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }
}

[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    [DynamoDBProperty]
    public string Title
    {
        get; set;
    }
    [DynamoDBProperty]
    public string ISBN
    {
        get; set;
    }
    // Multi-valued (set type) attribute.
    [DynamoDBProperty("Authors")]
    public List<string> BookAuthors
}

```

```

    {
        get; set;
    }
    // Arbitrary type, with a converter to map it to DynamoDB type.
    [DynamoDBProperty(typeof(DimensionTypeConverter))]
    public DimensionType Dimensions
    {
        get; set;
    }

    public class DimensionType
    {
        public decimal Length
        {
            get; set;
        }
        public decimal Height
        {
            get; set;
        }
        public decimal Thickness
        {
            get; set;
        }
    }

    // Converts the complex type DimensionType to string and vice-versa.
    public class DimensionTypeConverter : IPropertyConverter
    {
        public DynamoDBEntry ToEntry(object value)
        {
            DimensionType bookDimensions = value as DimensionType;
            if (bookDimensions == null) throw new ArgumentOutOfRangeException();

            string data = string.Format("{1}{0}{2}{0}{3}", " x ",
                bookDimensions.Length, bookDimensions.Height,
                bookDimensions.Thickness);

            DynamoDBEntry entry = new Primitive
            {
                Value = data
            };
            return entry;
        }

        public object FromEntry(DynamoDBEntry entry)
        {
            Primitive primitive = entry as Primitive;
            if (primitive == null || !(primitive.Value is String) ||
string.IsNullOrEmpty((string)primitive.Value))
                throw new ArgumentOutOfRangeException();

            string[] data = ((string)(primitive.Value)).Split(new string[] { " x " },
StringSplitOptions.None);
            if (data.Length != 3) throw new ArgumentOutOfRangeException();

            DimensionType complexData = new DimensionType
            {
                Length = Convert.ToDecimal(data[0]),
                Height = Convert.ToDecimal(data[1]),
                Thickness = Convert.ToDecimal(data[2])
            };
            return complexData;
        }
    }
}

```

}

Batch Operations Using the AWS SDK for .NET Object Persistence Model

Batch Write: Putting and Deleting Multiple Items

To put or delete multiple objects from a table in a single request, do the following:

- Execute the `CreateBatchWrite` method of the `DynamoDBContext`, and create an instance of the `BatchWrite` class.
- Specify the items that you want to put or delete.
 - To put one or more items, use either the `AddPutItem` or the `AddPutItems` method.
 - To delete one or more items, you can specify either the primary key of the item or a client-side object that maps to the item that you want to delete. Use the `AddDeleteItem`, `AddDeleteItems`, and the `AddDeleteKey` methods to specify the list of items to delete.
- Call the `BatchWrite.Execute` method to put and delete all the specified items from the table.

Note

When using the object persistence model, you can specify any number of operations in a batch. However, note that Amazon DynamoDB limits the number of operations in a batch and the total size of the batch in a batch operation. For more information about the specific limits, see [BatchWriteItem](#). If the API detects that your batch write request exceeded the allowed number of write requests or exceeded the maximum allowed HTTP payload size, it breaks the batch into several smaller batches. Additionally, if a response to a batch write returns unprocessed items, the API automatically sends another batch request with those unprocessed items.

Suppose that you defined a C# class `Book` class that maps to the `ProductCatalog` table in DynamoDB. The following C# code example uses the `BatchWrite` object to upload two items and delete one item from the `ProductCatalog` table.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
var bookBatch = context.CreateBatchWrite<Book>();

// 1. Specify two books to add.
Book book1 = new Book
{
    Id = 902,
    ISBN = "902-11-11-1111",
    ProductCategory = "Book",
    Title = "My book3 in batch write"
};
Book book2 = new Book
{
    Id = 903,
    ISBN = "903-11-11-1111",
    ProductCategory = "Book",
    Title = "My book4 in batch write"
};

bookBatch.AddPutItems(new List<Book> { book1, book2 });

// 2. Specify one book to delete.
bookBatch.AddDeleteKey(111);
```

```
bookBatch.Execute();
```

To put or delete objects from multiple tables, do the following:

- Create one instance of the `BatchWrite` class for each type and specify the items you want to put or delete as described in the preceding section.
- Create an instance of `MultiTableBatchWrite` using one of the following methods:
 - Execute the `Combine` method on one of the `BatchWrite` objects that you created in the preceding step.
 - Create an instance of the `MultiTableBatchWrite` type by providing a list of `BatchWrite` objects.
 - Execute the `CreateMultiTableBatchWrite` method of `DynamoDBContext` and pass in your list of `BatchWrite` objects.
- Call the `Execute` method of `MultiTableBatchWrite`, which performs the specified put and delete operations on various tables.

Suppose that you defined `Forum` and `Thread` C# classes that map to the `Forum` and `Thread` tables in DynamoDB. Also, suppose that the `Thread` class has versioning enabled. Because versioning is not supported when using batch operations, you must explicitly disable versioning as shown in the following C# code example. The example uses the `MultiTableBatchWrite` object to perform a multi-table update.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
// Create BatchWrite objects for each of the Forum and Thread classes.
var forumBatch = context.CreateBatchWrite<Forum>();

DynamoDBOperationConfig config = new DynamoDBOperationConfig();
config.SkipVersionCheck = true;
var threadBatch = context.CreateBatchWrite<Thread>(config);

// 1. New Forum item.
Forum newForum = new Forum
{
    Name = "Test BatchWrite Forum",
    Threads = 0
};
forumBatch.AddPutItem(newForum);

// 2. Specify a forum to delete by specifying its primary key.
forumBatch.AddDeleteKey("Some forum");

// 3. New Thread item.
Thread newThread = new Thread
{
    ForumName = "Amazon S3 forum",
    Subject = "My sample question",
    KeywordTags = new List<string> { "Amazon S3", "Bucket" },
    Message = "Message text"
};

threadBatch.AddPutItem(newThread);

// Now execute multi-table batch write.
var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);
superBatch.Execute();
```

For a working example, see [Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model \(p. 315\)](#).

Note

The DynamoDB batch API limits the number of writes in a batch and also limits the size of the batch. For more information, see [BatchWriteItem](#). When using the .NET object persistence model API, you can specify any number of operations. However, if either the number of operations in a batch or the size exceeds the limit, the .NET API breaks the batch write request into smaller batches and sends multiple batch write requests to DynamoDB.

Batch Get: Getting Multiple Items

To retrieve multiple items from a table in a single request, do the following:

- Create an instance of the `CreateBatchGet` class.
- Specify a list of primary keys to retrieve.
- Call the `Execute` method. The response returns the items in the `Results` property.

The following C# code example retrieves three items from the `ProductCatalog` table. The items in the result are not necessarily in the same order in which you specified the primary keys.

Example

```
DynamoDBContext context = new DynamoDBContext(client);
var bookBatch = context.CreateBatchGet<ProductCatalog>();
bookBatch.AddKey(101);
bookBatch.AddKey(102);
bookBatch.AddKey(103);
bookBatch.Execute();
// Process result.
Console.WriteLine(bookBatch.Results.Count);
Book book1 = bookBatch.Results[0];
Book book2 = bookBatch.Results[1];
Book book3 = bookBatch.Results[2];
```

To retrieve objects from multiple tables, do the following:

- For each type, create an instance of the `CreateBatchGet` type and provide the primary key values you want to retrieve from each table.
- Create an instance of the `MultiTableBatchGet` class using one of the following methods:
 - Execute the `Combine` method on one of the `BatchGet` objects you created in the preceding step.
 - Create an instance of the `MultiBatchGet` type by providing a list of `BatchGet` objects.
 - Execute the `CreateMultiTableBatchGet` method of `DynamoDBContext` and pass in your list of `BatchGet` objects.
- Call the `Execute` method of `MultiTableBatchGet`, which returns the typed results in the individual `BatchGet` objects.

The following C# code example retrieves multiple items from the `Order` and `OrderDetail` tables using the `CreateBatchGet` method.

Example

```
var orderBatch = context.CreateBatchGet<Order>();
orderBatch.AddKey(101);
orderBatch.AddKey(102);

var orderDetailBatch = context.CreateBatchGet<OrderDetail>();
orderDetailBatch.AddKey(101, "P1");
```

```

orderDetailBatch.AddKey(101, "P2");
orderDetailBatch.AddKey(102, "P3");
orderDetailBatch.AddKey(102, "P1");

var orderAndDetailSuperBatch = orderBatch.Combine(orderDetailBatch);
orderAndDetailSuperBatch.Execute();

Console.WriteLine(orderBatch.Results.Count);
Console.WriteLine(orderDetailBatch.Results.Count);

Order order1 = orderBatch.Results[0];
Order order2 = orderDetailBatch.Results[1];
OrderDetail orderDetail1 = orderDetailBatch.Results[0];

```

Example: CRUD Operations Using the AWS SDK for .NET Object Persistence Model

The following C# code example declares a `Book` class with `Id`, `Title`, `ISBN`, and `Authors` properties. The example uses object persistence attributes to map these properties to the `ProductCatalog` table in Amazon DynamoDB. The example then uses the `DynamoDBContext` to illustrate typical create, read, update, and delete (CRUD) operations. The example creates a sample `Book` instance and saves it to the `ProductCatalog` table. It then retrieves the book item and updates its `ISBN` and `Authors` properties. Note that the update replaces the existing authors list. Finally, the example deletes the book item.

For more information about the `ProductCatalog` table used in this example, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). For step-by-step instructions to test the following example, see [.NET Code Examples \(p. 332\)](#).

Note

The following example doesn't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

Example

```

/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class HighLevelItemCRUD
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)

```

```

    {
        try
        {
            DynamoDBContext context = new DynamoDBContext(client);
            TestCRUDOperations(context);
            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }
        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }

    private static void TestCRUDOperations(DynamoDBContext context)
    {
        int bookID = 1001; // Some unique value.
        Book myBook = new Book
        {
            Id = bookID,
            Title = "object persistence-AWS SDK for.NET SDK-Book 1001",
            ISBN = "111-111111001",
            BookAuthors = new List<string> { "Author 1", "Author 2" },
        };

        // Save the book.
        context.Save(myBook);
        // Retrieve the book.
        Book bookRetrieved = context.Load<Book>(bookID);

        // Update few properties.
        bookRetrieved.ISBN = "222-2222221001";
        bookRetrieved.BookAuthors = new List<string> { " Author 1", "Author x" }; // Replace existing authors list with this.
        context.Save(bookRetrieved);

        // Retrieve the updated book. This time add the optional ConsistentRead parameter using DynamoDBContextConfig object.
        Book updatedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
        {
            ConsistentRead = true
        });

        // Delete the book.
        context.Delete<Book>(bookID);
        // Try to retrieve deleted book. It should return null.
        Book deletedBook = context.Load<Book>(bookID, new DynamoDBContextConfig
        {
            ConsistentRead = true
        });
        if (deletedBook == null)
            Console.WriteLine("Book is deleted");
    }
}

[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    [DynamoDBProperty]
    public string Title
    {
        get; set;
    }
}

```

```

        }
        [DynamoDBProperty]
        public string ISBN
        {
            get; set;
        }
        [DynamoDBProperty("Authors")] //String Set datatype
        public List<string> BookAuthors
        {
            get; set;
        }
    }
}

```

Example: Batch Write Operation Using the AWS SDK for .NET Object Persistence Model

The following C# code example declares `Book`, `Forum`, `Thread`, and `Reply` classes and maps them to Amazon DynamoDB tables using the object persistence model attributes.

The example then uses the `DynamoDBContext` to illustrate the following batch write operations:

- `BatchWrite` object to put and delete book items from the `ProductCatalog` table.
- `MultiTableBatchWrite` object to put and delete items from the `Forum` and the `Thread` tables.

For more information about the tables used in this example, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). For step-by-step instructions to test the following example, see [.NET Code Examples \(p. 332\)](#).

Note

The following example doesn't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

Example

```


/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DataModel;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class HighLevelBatchWriteItem
    {


```

```

private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

static void Main(string[] args)
{
    try
    {
        DynamoDBContext context = new DynamoDBContext(client);
        SingleTableBatchWrite(context);
        MultiTableBatchWrite(context);
    }
    catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
    catch (Exception e) { Console.WriteLine(e.Message); }

    Console.WriteLine("To continue, press Enter");
    Console.ReadLine();
}

private static void SingleTableBatchWrite(DynamoDBContext context)
{
    Book book1 = new Book
    {
        Id = 902,
        InPublication = true,
        ISBN = "902-11-11-1111",
        PageCount = "100",
        Price = 10,
        ProductCategory = "Book",
        Title = "My book3 in batch write"
    };
    Book book2 = new Book
    {
        Id = 903,
        InPublication = true,
        ISBN = "903-11-11-1111",
        PageCount = "200",
        Price = 10,
        ProductCategory = "Book",
        Title = "My book4 in batch write"
    };

    var bookBatch = context.CreateBatchWrite<Book>();
    bookBatch.AddPutItems(new List<Book> { book1, book2 });

    Console.WriteLine("Performing batch write in SingleTableBatchWrite()..");
    bookBatch.Execute();
}

private static void MultiTableBatchWrite(DynamoDBContext context)
{
    // 1. New Forum item.
    Forum newForum = new Forum
    {
        Name = "Test BatchWrite Forum",
        Threads = 0
    };
    var forumBatch = context.CreateBatchWrite<Forum>();
    forumBatch.AddPutItem(newForum);

    // 2. New Thread item.
    Thread newThread = new Thread
    {
        ForumName = "S3 forum",
        Subject = "My sample question",
        KeywordTags = new List<string> { "S3", "Bucket" },
        Message = "Message text"
    };
}

```

```

DynamoDBOperationConfig config = new DynamoDBOperationConfig();
config.SkipVersionCheck = true;
var threadBatch = context.CreateBatchWrite<Thread>(config);
threadBatch.AddPutItem(newThread);
threadBatch.AddDeleteKey("some partition key value", "some sort key value");

var superBatch = new MultiTableBatchWrite(forumBatch, threadBatch);
Console.WriteLine("Performing batch write in MultiTableBatchWrite().");
superBatch.Execute();
}

}

[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey] //Partition key
    public string Id
    {
        get; set;
    }

    [DynamoDBRangeKey] //Sort key
    public DateTime ReplyDateTime
    {
        get; set;
    }

    // Properties included implicitly.
    public string Message
    {
        get; set;
    }
    // Explicit property mapping with object persistence model attributes.
    [DynamoDBProperty("LastPostedBy")]
    public string PostedBy
    {
        get; set;
    }
    // Property to store version number for optimistic locking.
    [DynamoDBVersion]
    public int? Version
    {
        get; set;
    }
}

[DynamoDBTable("Thread")]
public class Thread
{
    // PK mapping.
    [DynamoDBHashKey]      //Partition key
    public string ForumName
    {
        get; set;
    }
    [DynamoDBRangeKey]      //Sort key
    public String Subject
    {
        get; set;
    }
    // Implicit mapping.
    public string Message
    {
        get; set;
    }
}

```

```

public string LastPostedBy
{
    get; set;
}
public int Views
{
    get; set;
}
public int Replies
{
    get; set;
}
public bool Answered
{
    get; set;
}
public DateTime LastPostedDateTime
{
    get; set;
}
// Explicit mapping (property and table attribute names are different.
[DynamoDBProperty("Tags")]
public List<string> KeywordTags
{
    get; set;
}
// Property to store version number for optimistic locking.
[DynamoDBVersion]
public int? Version
{
    get; set;
}

[DynamoDBTable("Forum")]
public class Forum
{
    [DynamoDBHashKey]          //Partition key
    public string Name
    {
        get; set;
    }
    // All the following properties are explicitly mapped,
    // only to show how to provide mapping.
    [DynamoDBProperty]
    public int Threads
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Views
    {
        get; set;
    }
    [DynamoDBProperty]
    public string LastPostBy
    {
        get; set;
    }
    [DynamoDBProperty]
    public DateTime LastPostDateTime
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Messages
}

```

```

        {
            get; set;
        }

    }

    [DynamoDBTable("ProductCatalog")]
    public class Book
    {
        [DynamoDBHashKey] //Partition key
        public int Id
        {
            get; set;
        }
        public string Title
        {
            get; set;
        }
        public string ISBN
        {
            get; set;
        }
        public int Price
        {
            get; set;
        }
        public string PageCount
        {
            get; set;
        }
        public string ProductCategory
        {
            get; set;
        }
        public bool InPublication
        {
            get; set;
        }
    }
}

```

Example: Query and Scan in DynamoDB Using the AWS SDK for .NET Object Persistence Model

The C# example in this section defines the following classes and maps them to the tables in DynamoDB. For more information about creating the tables used in this example, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

- The `Book` class maps to the `ProductCatalog` table.
- The `Forum`, `Thread`, and `Reply` classes map to tables of the same name.

The example then executes the following query and scan operations using `DynamoDBContext`.

- Get a book by `Id`.

The `ProductCatalog` table has `Id` as its primary key. It does not have a sort key as part of its primary key. Therefore, you cannot query the table. You can get an item using its `Id` value.

- Execute the following queries against the `Reply` table. (The `Reply` table's primary key is composed of `Id` and `ReplyDateTime` attributes. The `ReplyDateTime` is a sort key. Therefore, you can query this table.)
 - Find replies to a forum thread posted in the last 15 days.

- Find replies to a forum thread posted in a specific date range.
- Scan the `ProductCatalog` table to find books whose price is less than zero.

For performance reasons, you should use a query operation instead of a scan operation. However, there are times you might need to scan a table. Suppose that there was a data entry error and one of the book prices is set to less than 0. This example scans the `ProductCategory` table to find book items (the `ProductCategory` is book) at price of less than 0.

For instructions about creating a working sample, see [.NET Code Examples \(p. 332\)](#).

Note

The following example does not work with .NET core because it does not support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using System.Configuration;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.DataModel;  
using Amazon.DynamoDBv2.DocumentModel;  
using Amazon.Runtime;  
  
namespace com.amazonaws.codesamples  
{  
    class HighLevelQueryAndScan  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
        static void Main(string[] args)  
        {  
            try  
            {  
                DynamoDBContext context = new DynamoDBContext(client);  
                // Get an item.  
                GetBook(context, 101);  
  
                // Sample forum and thread to test queries.  
                string forumName = "Amazon DynamoDB";  
                string threadSubject = "DynamoDB Thread 1";  
                // Sample queries.  
                FindRepliesInLast15Days(context, forumName, threadSubject);  
                FindRepliesPostedWithinTimePeriod(context, forumName, threadSubject);  
  
                // Scan table.  
                FindProductsPricedLessThanZero(context);  
                Console.WriteLine("To continue, press Enter");  
                Console.ReadLine();  
            }  
        }  
    }  
}
```

```

        }
        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }

    private static void GetBook(DynamoDBContext context, int productId)
    {
        Book bookItem = context.Load<Book>(productId);

        Console.WriteLine("\nGetBook: Printing result.....");
        Console.WriteLine("Title: {0} \n No.Of threads:{1} \n No. of messages: {2}",
            bookItem.Title, bookItem.ISBN, bookItem.PageCount);
    }

    private static void FindRepliesInLast15Days(DynamoDBContext context,
                                                string forumName,
                                                string threadSubject)
    {
        string replyId = forumName + "#" + threadSubject;
        DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
        IEnumerable<Reply> latestReplies =
            context.Query<Reply>(replyId, QueryOperator.GreaterThan, twoWeeksAgoDate);
        Console.WriteLine("\nFindRepliesInLast15Days: Printing result.....");
        foreach (Reply r in latestReplies)
            Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.PostedBy, r.Message,
r.ReplyDateTime);
    }

    private static void FindRepliesPostedWithinTimePeriod(DynamoDBContext context,
                                                       string forumName,
                                                       string threadSubject)
    {
        string forumId = forumName + "#" + threadSubject;
        Console.WriteLine("\nFindRepliesPostedWithinTimePeriod: Printing result.....");

        DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(30);
        DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(1);

        IEnumerable<Reply> repliesInAPeriod = context.Query<Reply>(forumId,
                                                               QueryOperator.Between, startDate, endDate);
        foreach (Reply r in repliesInAPeriod)
            Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.PostedBy, r.Message,
r.ReplyDateTime);
    }

    private static void FindProductsPricedLessThanZero(DynamoDBContext context)
    {
        int price = 0;
        IEnumerable<Book> itemsWithWrongPrice = context.Scan<Book>(
            new ScanCondition("Price", ScanOperator.LessThan, price),
            new ScanCondition("ProductCategory", ScanOperator.Equal, "Book")
        );
        Console.WriteLine("\nFindProductsPricedLessThanZero: Printing result.....");
        foreach (Book r in itemsWithWrongPrice)
            Console.WriteLine("{0}\t{1}\t{2}\t{3}", r.Id, r.Title, r.Price, r.ISBN);
    }
}

[DynamoDBTable("Reply")]
public class Reply
{
    [DynamoDBHashKey] //Partition key
    public string Id
    {
        get; set;
}

```

```

        }

        [DynamoDBRangeKey] //Sort key
        public DateTime ReplyDateTime
        {
            get; set;
        }

        // Properties included implicitly.
        public string Message
        {
            get; set;
        }
        // Explicit property mapping with object persistence model attributes.
        [DynamoDBProperty("LastPostedBy")]
        public string PostedBy
        {
            get; set;
        }
        // Property to store version number for optimistic locking.
        [DynamoDBVersion]
        public int? Version
        {
            get; set;
        }
    }

    [DynamoDBTable("Thread")]
    public class Thread
    {
        // Partition key mapping.
        [DynamoDBHashKey] //Partition key
        public string ForumName
        {
            get; set;
        }
        [DynamoDBRangeKey] //Sort key
        public DateTime Subject
        {
            get; set;
        }
        // Implicit mapping.
        public string Message
        {
            get; set;
        }
        public string LastPostedBy
        {
            get; set;
        }
        public int Views
        {
            get; set;
        }
        public int Replies
        {
            get; set;
        }
        public bool Answered
        {
            get; set;
        }
        public DateTime LastPostedDateTime
        {
            get; set;
        }
    }
}

```

```

// Explicit mapping (property and table attribute names are different).
[DynamoDBProperty("Tags")]
public List<string> KeywordTags
{
    get; set;
}
// Property to store version number for optimistic locking.
[DynamoDBVersion]
public int? Version
{
    get; set;
}

[DynamoDBTable("Forum")]
public class Forum
{
    [DynamoDBHashKey]
    public string Name
    {
        get; set;
    }
    // All the following properties are explicitly mapped
    // to show how to provide mapping.
    [DynamoDBProperty]
    public int Threads
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Views
    {
        get; set;
    }
    [DynamoDBProperty]
    public string LastPostBy
    {
        get; set;
    }
    [DynamoDBProperty]
    public DateTime LastPostDateTime
    {
        get; set;
    }
    [DynamoDBProperty]
    public int Messages
    {
        get; set;
    }
}
[DynamoDBTable("ProductCatalog")]
public class Book
{
    [DynamoDBHashKey] //Partition key
    public int Id
    {
        get; set;
    }
    public string Title
    {
        get; set;
    }
    public string ISBN
    {
        get; set;
    }
}

```

```
        }
    public int Price
    {
        get; set;
    }
    public string PageCount
    {
        get; set;
    }
    public string ProductCategory
    {
        get; set;
    }
    public bool InPublication
    {
        get; set;
    }
}
```

Running the Code Examples in This Developer Guide

The AWS SDKs provide broad support for Amazon DynamoDB in the following languages:

- [Java](#)
- [JavaScript in the browser](#)
- [.NET](#)
- [Node.js](#)
- [PHP](#)
- [Python](#)
- [Ruby](#)
- [C++](#)
- [Go](#)
- [Android](#)
- [iOS](#)

To get started quickly with these languages, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

The code examples in this developer guide provide more in-depth coverage of DynamoDB operations, using the following programming languages:

- [Java Code Examples \(p. 330\)](#)
- [.NET Code Examples \(p. 332\)](#)

Before you can begin with this exercise, you need to create an AWS account, get your access key and secret key, and set up the AWS Command Line Interface (AWS CLI) on your computer. For more information, see [Setting Up DynamoDB \(Web Service\) \(p. 51\)](#).

Note

If you are using the downloadable version of DynamoDB, you need to use the AWS CLI to create the tables and sample data. You also need to specify the `--endpoint-url` parameter with each AWS CLI command. For more information, see [Setting the Local Endpoint \(p. 50\)](#).

Creating Tables and Loading Data for Code Examples in DynamoDB

In this tutorial, you use the AWS Management Console to create tables in Amazon DynamoDB. You then load data into these tables using the AWS Command Line Interface (AWS CLI).

These tables and their data are used as examples throughout this developer guide.

Note

If you are an application developer, we recommend that you also read [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#), which uses the downloadable version of DynamoDB. This lets you learn about the DynamoDB low-level API without having to pay any fees for throughput, storage, or data transfer.

Topics

- [Step 1: Create Example Tables \(p. 325\)](#)
- [Step 2: Load Data into Tables \(p. 327\)](#)
- [Step 3: Query the Data \(p. 328\)](#)
- [Step 4: \(Optional\) Clean Up \(p. 329\)](#)
- [Summary \(p. 330\)](#)

Step 1: Create Example Tables

In this section, you use the AWS Management Console to create tables in Amazon DynamoDB for two simple use cases.

Use Case 1: Product Catalog

Suppose that you want to store product information in DynamoDB. Each product has its own distinct attributes, so you need to store different information about each of these products.

You can create a `ProductCatalog` table, where each item is uniquely identified by a single, numeric attribute: `Id`.

| Table Name | Primary Key |
|----------------|---|
| ProductCatalog | Partition key: <code>Id</code> (Number) |

Use Case 2: Forum Application

Suppose that you want to build an application for message boards or discussion forums. AWS [Discussion Forums](#) represent one example of such an application. Customers can engage with the developer community, ask questions, or reply to other customers' posts. Each AWS service has a dedicated forum. Anyone can start a new discussion thread by posting a message in a forum. Each thread might receive any number of replies.

You can model this application by creating three tables: `Forum`, `Thread`, and `Reply`.

| Table Name | Primary Key |
|------------|---|
| Forum | Partition key: <code>Name</code> (String) |

| Table Name | Primary Key |
|------------|---|
| Thread | Partition key: <code>ForumName</code> (String) Sort key: <code>Subject</code> (String) |
| Reply | Partition key: <code>Id</code> (String) Sort key: <code>ReplyDateTime</code> (String) |

The Reply table has a global secondary index named `PostedBy-Message-Index`. This index facilitates queries on two non-key attributes of the Reply table.

| Index Name | Primary Key |
|-------------------------------------|--|
| <code>PostedBy-Message-Index</code> | Partition key: <code>PostedBy</code> (String) Sort key: <code>Message</code> (String) |

Create the ProductCatalog Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. On the **Table** name box, enter `ProductCatalog`.
 - b. For the **Primary key**, in the **Partition key** box, enter `Id`. Set the data type to **Number**.
4. When the settings are as you want them, choose **Create**.

Create the Forum Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table** name box, enter `Forum`.
 - b. For the **Primary key**, in the **Partition key** box, enter `Name`. Set the data type to **String**.
4. When the settings are as you want them, choose **Create**.

Create the Thread Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** box, enter `Thread`.
 - b. For the **Primary key**, do the following:
 - In the **Partition key** box, enter `ForumName`. Set the data type to **String**.
 - Choose **Add sort key**.
 - In the **Sort key** box, enter `Subject`. Set the data type to **String**.

4. When the settings are as you want them, choose **Create**.

Create the Reply Table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. In the **Create DynamoDB table** screen, do the following:
 - a. In the **Table name** box, enter **Reply**.
 - b. For the **Primary key**, do the following:
 - In the **Partition key** box, enter **Id**. Set the data type to **String**.
 - Choose **Add sort key**.
 - In the **Sort key** box, enter **ReplyDateTime**. Set the data type to **String**.
 - c. In the **Table settings** section, clear **Use default settings**.
 - d. In the **Secondary indexes** section, choose **Add index**.
 - e. In the **Add index** window, do the following:
 - For the **Primary key**, do the following:
 - In the **Partition key** box, enter **PostedBy**. Set the data type to **String**.
 - Choose **Add sort key**.
 - In the **Sort key** box, enter **Message**. Set the data type to **String**.
 - In the **Index name** box, enter **PostedBy-Message-Index**.
 - Set the **Projected attributes** to **All**.
 - Choose **Add index**.
4. When the settings are as you want them, choose **Create**.

Step 2: Load Data into Tables

In this step, you load sample data into the tables that you created. You could enter the data manually into the Amazon DynamoDB console. However, to save time, you use the AWS Command Line Interface (AWS CLI) instead.

Note

If you have not yet set up the AWS CLI, see [Using the AWS CLI \(p. 56\)](#) for instructions.

You will download a .zip archive that contains JSON files with sample data for each table. For each file, you use the AWS CLI to load the data into DynamoDB. Each successful data load produces the following output.

```
{  
    "UnprocessedItems": {}  
}
```

Download the Sample Data File Archive

1. Download the sample data archive (`sampledata.zip`) using this link:
 - [sampledata.zip](#)
2. Extract the .json data files from the archive.
3. Copy the .json data files to your current directory.

Load the Sample Data into DynamoDB Tables

1. To load the `ProductCatalog` table with data, enter the following command.

```
aws dynamodb batch-write-item --request-items file://ProductCatalog.json
```

2. To load the `Forum` table with data, enter the following command.

```
aws dynamodb batch-write-item --request-items file://Forum.json
```

3. To load the `Thread` table with data, enter the following command.

```
aws dynamodb batch-write-item --request-items file://Thread.json
```

4. To load the `Reply` table with data, enter the following command.

```
aws dynamodb batch-write-item --request-items file://Reply.json
```

Verify Data Load

You can use the AWS Management Console to verify the data that you loaded into the tables.

To verify the data using the AWS Management Console

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose **ProductCatalog**.
4. Choose the **Items** tab to view the data that you loaded into the table.
5. To view an item in the table, choose its ID. (If you want, you can also edit the item.)
6. To return to the list of tables, choose **Cancel**.

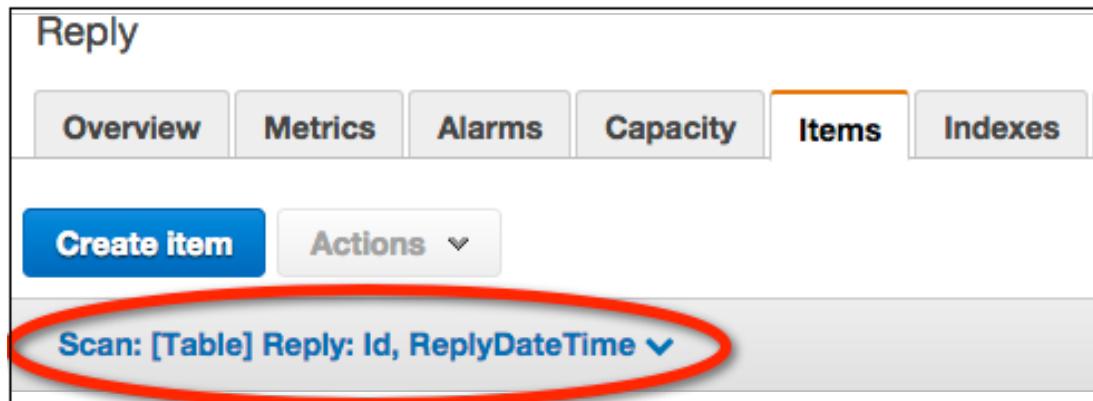
Repeat this procedure for each of the other tables you created:

- `Forum`
- `Thread`
- `Reply`

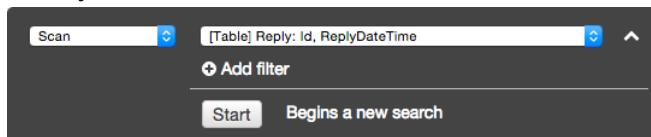
Step 3: Query the Data

In this step, you try some simple queries against the tables that you created, using the Amazon DynamoDB console.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose **Reply**.
4. Choose the **Items** tab to view the data that you loaded into the table.
5. Choose the data filtering link, located just below the **Create item** button.



When you choose the link, the console reveals a data filtering pane.



6. In the data filtering pane, do the following:
 - a. Change the operation from **Scan** to **Query**.
 - b. For the **Partition key**, enter the value **Amazon DynamoDB#DynamoDB Thread 1**.
 - c. Choose **Start**. Only the items that match your query criteria are returned from the **Reply** table.
7. The **Reply** table has a global secondary index on the **PostedBy** and **Message** attributes. You can use the data filtering pane to query the index. Do the following:
 - a. Change the query source from the following:

[Table] Reply: Id, ReplyDateTime

To the following:

[Index] PostedBy-Message-Index: PostedBy, Message

- b. For the **Partition key**, enter the value **User A**.
- c. Choose **Start**. Only the items that match your query criteria are returned from **PostedBy-Message-Index**.

Take some time to explore your other tables using the DynamoDB console:

- ProductCatalog
- Forum
- Thread

Step 4: (Optional) Clean Up

These sample tables are used throughout the *Amazon DynamoDB Developer Guide* to help illustrate table and item operations using the low-level API and various AWS SDKs. If you plan to read the rest of this guide, you might find the tables useful as a reference. However, if you don't want to keep these tables, you should delete them to avoid being charged for resources that you don't need.

To delete the sample tables

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose **ProductCatalog**.
4. Choose **Delete Table**. You are asked to confirm your selection.

Repeat this procedure for each of the other tables you created:

- **Forum**
- **Thread**
- **Reply**

Summary

In this exercise, you used the DynamoDB console to create several tables in DynamoDB. You then used the AWS CLI to load data into the tables, and performed some basic operations on the data using the console.

The console and the AWS CLI are helpful for getting started quickly. However, you probably want to learn more about how DynamoDB works, and how to write application programs with DynamoDB. The rest of this developer guide addresses those topics.

Java Code Examples

Topics

- [Java: Setting Your AWS Credentials \(p. 331\)](#)
- [Java: Setting the AWS Region and Endpoint \(p. 331\)](#)

This Developer Guide contains Java code snippets and ready-to-run programs. You can find these code examples in the following sections:

- [Working with Items and Attributes \(p. 374\)](#)
- [Working with Tables and Data in DynamoDB \(p. 335\)](#)
- [Working with Queries in DynamoDB \(p. 458\)](#)
- [Working with Scans in DynamoDB \(p. 475\)](#)
- [Improving Data Access with Secondary Indexes \(p. 496\)](#)
- [Java: DynamoDBMapper \(p. 225\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#)

You can get started quickly by using Eclipse with the [AWS Toolkit for Eclipse](#). In addition to a full-featured IDE, you also get the AWS SDK for Java with automatic updates, and preconfigured templates for building AWS applications.

To run the Java code examples (using Eclipse)

1. Download and install the [Eclipse](#) IDE.
2. Download and install the [AWS Toolkit for Eclipse](#).
3. Start Eclipse, and on the **Eclipse** menu, choose **File, New, and then Other**.

4. In **Select a wizard**, choose **AWS**, choose **AWS Java Project**, and then choose **Next**.
5. In **Create an AWS Java**, do the following:
 - a. In **Project name**, enter a name for your project.
 - b. In **Select Account**, choose your credentials profile from the list.If this is your first time using the [AWS Toolkit for Eclipse](#), choose **Configure AWS Accounts** to set up your AWS credentials.
6. Choose **Finish** to create the project.
7. From the **Eclipse** menu, choose **File**, **New**, and then **Class**.
8. In **Java Class**, enter a name for your class in **Name** (use the same name as the code example that you want to run), and then choose **Finish** to create the class.
9. Copy the code example from the documentation page into the Eclipse editor.
10. To run the code, choose **Run** on the Eclipse menu.

The SDK for Java provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see the [AWS SDK for Java](#).

Note

The code examples in this guide are intended for use with the latest version of the AWS SDK for Java.

If you are using the AWS Toolkit for Eclipse, you can configure automatic updates for the SDK for Java. To do this in Eclipse, go to **Preferences** and choose **AWS Toolkit**, **AWS SDK for Java**, **Download new SDKs automatically**.

Java: Setting Your AWS Credentials

The SDK for Java requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using an AWS credentials file, as described in [Set Up Your AWS Credentials](#) in the *AWS SDK for Java Developer Guide*.

The following is an example of an AWS credentials file named `~/.aws/credentials`, where the tilde character (~) represents your home directory.

```
[default]
aws_access_key_id = AWS access key ID goes here
aws_secret_access_key = Secret key goes here
```

Java: Setting the AWS Region and Endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AmazonDynamoDB` properties.

The following code example instantiates a new `AmazonDynamoDB`.

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.regions.Regions;
...
// This client will default to US West (Oregon)
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
.withRegion(Regions.US_WEST_2)
.build();
```

You can use the `withRegion` method to run your code against DynamoDB in any Region where it is available. For a complete list, see [AWS Regions and Endpoints](#) in the *Amazon Web Services General Reference*.

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().withEndpointConfiguration(new AwsClientBuilder.EndpointConfiguration("http://localhost:8000", "us-west-2")).build();
```

.NET Code Examples

Topics

- [.NET: Setting Your AWS Credentials \(p. 333\)](#)
- [.NET: Setting the AWS Region and Endpoint \(p. 333\)](#)

This guide contains .NET code snippets and ready-to-run programs. You can find these code examples in the following sections:

- [Working with Items and Attributes \(p. 374\)](#)
- [Working with Tables and Data in DynamoDB \(p. 335\)](#)
- [Working with Queries in DynamoDB \(p. 458\)](#)
- [Working with Scans in DynamoDB \(p. 475\)](#)
- [Improving Data Access with Secondary Indexes \(p. 496\)](#)
- [.NET: Document Model \(p. 273\)](#)
- [.NET: Object Persistence Model \(p. 295\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#)

You can get started quickly by using the AWS SDK for .NET with the Toolkit for Visual Studio.

To run the .NET code examples (using Visual Studio)

1. Download and install [Microsoft Visual Studio](#).
2. Download and install the [Toolkit for Visual Studio](#).
3. Start Visual Studio. Choose **File**, **New**, **Project**.
4. In **New Project**, choose **AWS Empty Project**, and then choose **OK**.
5. In **AWS Access Credentials**, choose **Use existing profile**, choose your credentials profile from the list, and then choose **OK**.

If this is your first time using Toolkit for Visual Studio, choose **Use a new profile** to set up your AWS credentials.

6. In your Visual Studio project, choose the tab for your program's source code (`Program.cs`). Copy the code example from the documentation page into the Visual Studio editor, replacing any other code that you see in the editor.
7. If you see error messages of the form The type or namespace name...could not be found, you need to install the AWS SDK assembly for DynamoDB as follows:
 - a. In Solution Explorer, open the context (right-click) menu for your project, and then choose **Manage NuGet Packages**.
 - b. In NuGet Package Manager, choose **Browse**.

- c. In the search box, enter **AWSSDK.DynamoDBv2**, and wait for the search to complete.
 - d. Choose **AWSSDK.DynamoDBv2**, and then choose **Install**.
 - e. When the installation is complete, choose the **Program.cs** tab to return to your program.
8. To run the code, choose **Start** in the Visual Studio toolbar.

The AWS SDK for .NET provides thread-safe clients for working with DynamoDB. As a best practice, your applications should create one client and reuse the client between threads.

For more information, see [AWS SDK for .NET](#).

Note

The code examples in this guide are intended for use with the latest version of the AWS SDK for .NET.

.NET: Setting Your AWS Credentials

The AWS SDK for .NET requires that you provide AWS credentials to your application at runtime. The code examples in this guide assume that you are using the SDK Store to manage your AWS credentials file, as described in [Using the SDK Store](#) in the *AWS SDK for .NET Developer Guide*.

The Toolkit for Visual Studio supports multiple sets of credentials from any number of accounts. Each set is referred to as a *profile*. Visual Studio adds entries to the project's `App.config` file so that your application can find the AWS credentials at runtime.

The following example shows the default `App.config` file that is generated when you create a new project using Toolkit for Visual Studio.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="AWSProfileName" value="default"/>
    <add key="AWSRegion" value="us-west-2" />
  </appSettings>
</configuration>
```

At runtime, the program uses the `default` set of AWS credentials, as specified by the `AWSProfileName` entry. The AWS credentials themselves are kept in the SDK Store in encrypted form. The Toolkit for Visual Studio provides a graphical user interface for managing your credentials, all from within Visual Studio. For more information, see [Specifying Credentials](#) in the *AWS Toolkit for Visual Studio User Guide*.

Note

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AWSRegion` entry in the `App.config` file. You can set `AWSRegion` to any Region where DynamoDB is available. For a complete list, see [AWS Regions and Endpoints](#) in the *Amazon Web Services General Reference*.

.NET: Setting the AWS Region and Endpoint

By default, the code examples access DynamoDB in the US West (Oregon) Region. You can change the Region by modifying the `AWSRegion` entry in the `App.config` file. Or, you can change the Region by modifying the `AmazonDynamoDBClient` properties.

The following code example instantiates a new `AmazonDynamoDBClient`. The client is modified so that the code runs against DynamoDB in a different Region.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// This client will access the US East 1 region.
```

```
clientConfig.RegionEndpoint = RegionEndpoint.USEast1;
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

For a complete list of Regions, see [AWS Regions and Endpoints in the Amazon Web Services General Reference](#).

If you want to run the code examples using DynamoDB locally on your computer, set the endpoint as follows.

```
AmazonDynamoDBConfig clientConfig = new AmazonDynamoDBConfig();
// Set the endpoint URL
clientConfig.ServiceURL = "http://localhost:8000";
AmazonDynamoDBClient client = new AmazonDynamoDBClient(clientConfig);
```

Working with Tables, Items, Queries, Scans, and Indexes

This section provides details about working with tables, items, queries, and more in Amazon DynamoDB.

Topics

- [Working with Tables and Data in DynamoDB \(p. 335\)](#)
- [Working with Items and Attributes \(p. 374\)](#)
- [Working with Queries in DynamoDB \(p. 458\)](#)
- [Working with Scans in DynamoDB \(p. 475\)](#)
- [Improving Data Access with Secondary Indexes \(p. 496\)](#)
- [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#)

Working with Tables and Data in DynamoDB

This section describes how to use the AWS Command Line Interface (AWS CLI) and the AWS SDKs to create, update, and delete tables in Amazon DynamoDB.

Note

You can also perform these same tasks using the AWS Management Console. For more information, see [Using the Console \(p. 54\)](#).

This section also provides more information about throughput capacity using DynamoDB auto scaling or manually setting provisioned throughput.

Topics

- [Basic Operations on DynamoDB Tables \(p. 335\)](#)
- [Considerations When Changing Read/Write Capacity Mode \(p. 340\)](#)
- [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#)
- [DynamoDB Item Sizes and Formats \(p. 345\)](#)
- [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#)
- [Adding Tags and Labels to Resources \(p. 359\)](#)
- [Working with DynamoDB Tables in Java \(p. 362\)](#)
- [Working with DynamoDB Tables in .NET \(p. 367\)](#)

Basic Operations on DynamoDB Tables

Similar to other database systems, Amazon DynamoDB stores data in tables. You can manage your tables using a few basic operations.

Topics

- [Creating a Table \(p. 336\)](#)
- [Describing a Table \(p. 338\)](#)
- [Updating a Table \(p. 338\)](#)
- [Deleting a Table \(p. 339\)](#)
- [Listing Table Names \(p. 339\)](#)

- [Describing Provisioned Throughput Limits \(p. 339\)](#)

Creating a Table

Use the `CreateTable` operation to create a table in Amazon DynamoDB. To create the table, you must provide the following information:

- **Table name.** The name must conform to the DynamoDB naming rules, and must be unique for the current AWS account and Region. For example, you could create a `People` table in US East (N. Virginia) and another `People` table in Europe (Ireland). However, these two tables would be entirely different from each other. For more information, see [Naming Rules and Data Types \(p. 12\)](#).
- **Primary key.** The primary key can consist of one attribute (partition key) or two attributes (partition key and sort key). You need to provide the attribute names, data types, and the role of each attribute: `HASH` (for a partition key) and `RANGE` (for a sort key). For more information, see [Primary Key \(p. 6\)](#).
- **Throughput settings (for provisioned tables).** If using provisioned mode, you must specify the initial read and write throughput settings for the table. You can modify these settings later, or enable DynamoDB auto scaling to manage the settings for you. For more information, see [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#) and [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#).

Example 1: Create a Provisioned Table

The following AWS CLI example shows how to create a table (`Music`). The primary key consists of `Artist` (partition key) and `SongTitle` (sort key), each of which has a data type of `String`. The maximum throughput for this table is 10 read capacity units and 5 write capacity units.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5
```

The `CreateTable` operation returns metadata for the table, as shown following.

```
{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 5,
            "ReadCapacityUnits": 10
        },
    }
},
```

```

    "TableSizeBytes": 0,
    "TableName": "Music",
    "TableStatus": "CREATING",
    "TableId": "12345678-0123-4567-a123-abcdefghijkl",
    "KeySchema": [
        {
            "KeyType": "HASH",
            "AttributeName": "Artist"
        },
        {
            "KeyType": "RANGE",
            "AttributeName": "SongTitle"
        }
    ],
    "ItemCount": 0,
    "CreationDateTime": 1542397215.37
}
}

```

The `TableStatus` element indicates the current state of the table (`CREATING`). It might take a while to create the table, depending on the values you specify for `ReadCapacityUnits` and `WriteCapacityUnits`. Larger values for these require DynamoDB to allocate more resources for the table.

Example 2: Create an On-Demand Table

To create the same table `Music` using on-demand mode.

```

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

```

The `CreateTable` operation returns metadata for the table, as shown following.

```

{
    "TableDescription": {
        "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music",
        "AttributeDefinitions": [
            {
                "AttributeName": "Artist",
                "AttributeType": "S"
            },
            {
                "AttributeName": "SongTitle",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "WriteCapacityUnits": 0,
            "ReadCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "TableName": "Music",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST"
        }
    }
}

```

```
        },
        "TableStatus": "CREATING",
        "TableId": "12345678-0123-4567-a123-abcdefghijkl",
        "KeySchema": [
            {
                "KeyType": "HASH",
                "AttributeName": "Artist"
            },
            {
                "KeyType": "RANGE",
                "AttributeName": "SongTitle"
            }
        ],
        "ItemCount": 0,
        "CreationDateTime": 1542397468.348
    }
}
```

Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Describing a Table

To view details about a table, use the `DescribeTable` operation. You must provide the table name. The output from `DescribeTable` is in the same format as that from `CreateTable`. It includes the timestamp when the table was created, its key schema, its provisioned throughput settings, its estimated size, and any secondary indexes that are present.

Important

When calling `DescribeTable` on an on-demand table, read capacity units and write capacity units are set to 0.

Example

```
aws dynamodb describe-table --table-name Music
```

The table is ready for use when the `TableStatus` has changed from `CREATING` to `ACTIVE`.

Note

If you issue a `DescribeTable` request immediately after a `CreateTable` request, DynamoDB might return an error (`ResourceNotFoundException`). This is because `DescribeTable` uses an eventually consistent query, and the metadata for your table might not be available at that moment. Wait for a few seconds, and then try the `DescribeTable` request again.

For billing purposes, your DynamoDB storage costs include a per-item overhead of 100 bytes. (For more information, go to [DynamoDB Pricing](#).) This extra 100 bytes per item is not used in capacity unit calculations or by the `DescribeTable` operation.

Updating a Table

The `UpdateTable` operation allows you to do one of the following:

- Modify a table's provisioned throughput settings (for provisioned mode tables).
- Change the table's read/write capacity mode.
- Manipulate global secondary indexes on the table (see [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#)).
- Enable or disable DynamoDB Streams on the table (see [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#)).

Example

The following AWS CLI example shows how to modify a table's provisioned throughput settings.

```
aws dynamodb update-table --table-name Music \
    --provisioned-throughput ReadCapacityUnits=20,WriteCapacityUnits=10
```

Note

When you issue an `UpdateTable` request, the status of the table changes from `AVAILABLE` to `UPDATING`. The table remains fully available for use while it is `UPDATING`. When this process is completed, the table status changes from `UPDATING` to `AVAILABLE`.

Example

The following AWS CLI example shows how to modify a table's read/write capacity mode to on-demand mode.

```
aws dynamodb update-table --table-name Music \
    --billing-mode PAY_PER_REQUEST
```

Deleting a Table

You can remove an unused table with the `DeleteTable` operation. Deleting a table is an unrecoverable operation.

Example

The following AWS CLI example shows how to delete a table.

```
aws dynamodb delete-table --table-name Music
```

When you issue a `DeleteTable` request, the table's status changes from `ACTIVE` to `DELETING`. It might take a while to delete the table, depending on the resources it uses (such as the data stored in the table, and any streams or indexes on the table).

When the `DeleteTable` operation concludes, the table no longer exists in DynamoDB.

Listing Table Names

The `ListTables` operation returns the names of the DynamoDB tables for the current AWS account and Region.

Example

The following AWS CLI example shows how to list the DynamoDB table names.

```
aws dynamodb list-tables
```

Describing Provisioned Throughput Limits

The `DescribeLimits` operation returns the current read and write capacity limits for the current AWS account and Region.

Example

The following AWS CLI example shows how to describe the current provisioned throughput limits.

```
aws dynamodb describe-limits
```

The output shows the upper limits of read and write capacity units for the current AWS account and Region.

For more information about these limits, and how to request limit increases, see [Throughput Default Limits \(p. 961\)](#).

Considerations When Changing Read/Write Capacity Mode

You can switch between read/write capacity modes once every 24 hours. Consider the following when updating your read/write capacity mode in Amazon DynamoDB.

Managing Capacity

When you update a table from provisioned to on-demand mode, you don't need to specify how much read and write throughput you expect your application to perform.

Consider the following when you update a table from on-demand to provisioned mode:

- If you're using the AWS Management Console, the console estimates initial provisioned capacity values based on the consumed read and write capacity of your table and global secondary indexes over the past 30 minutes. To override these recommended values, choose `Override recommended values`.
- If you're using the AWS CLI or AWS SDK, choose the right provisioned capacity settings of your table and global secondary indexes by using Amazon CloudWatch to look at your historical consumption (`ConsumedWriteCapacityUnits` and `ConsumedReadCapacityUnits` metrics) to determine the new throughput settings.

Note

If you're switching a global table to provisioned mode, look at the maximum consumption across all your regional replicas for base tables and global secondary indexes when determining the new throughput settings.

Managing Auto Scaling

When you update a table from provisioned to on-demand mode:

- If you're using the console, all of your auto scaling settings (if any) will be deleted.
- If you're using the AWS CLI or AWS SDK, all of your auto scaling settings will be preserved. These settings can be applied when you update your table to provisioned billing mode again.

When you update a table from on-demand to provisioned mode:

- If you're using the console, DynamoDB recommends enabling auto scaling with the following defaults:
 - Target utilization: 70%
 - Minimum provisioned capacity: 5 units
 - Maximum provisioned capacity: The Region maximum
- If you're using the AWS CLI or SDK, your previous auto scaling settings (if any) are preserved.

Managing Settings on DynamoDB Provisioned Capacity Tables

When you create a new provisioned table in Amazon DynamoDB, you must specify its *provisioned throughput capacity*. This is the amount of read and write activity that the table can support. DynamoDB uses this information to reserve sufficient system resources to meet your throughput requirements.

Note

You can create an on-demand mode table instead so that you don't have to manage any capacity settings for servers, storage, or throughput. DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload. For more information, see [On-Demand Mode \(p. 17\)](#).

You can optionally allow DynamoDB auto scaling to manage your table's throughput capacity. However, you still must provide initial settings for read and write capacity when you create the table. DynamoDB auto scaling uses these initial settings as a starting point, and then adjusts them dynamically in response to your application's requirements. For more information, see [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#).

As your application data and access requirements change, you might need to adjust your table's throughput settings. If you're using DynamoDB auto scaling, the throughput settings are automatically adjusted in response to actual workloads. You can also use the `UpdateTable` operation to manually adjust your table's throughput capacity. You might decide to do this if you need to bulk-load data from an existing data store into your new DynamoDB table. You could create the table with a large write throughput setting and then reduce this setting after the bulk data load is complete.

You specify throughput requirements in terms of *capacity units*—the amount of data your application needs to read or write per second. You can modify these settings later, if needed, or enable DynamoDB auto scaling to modify them automatically.

Topics

- [Read Capacity Units \(p. 341\)](#)
- [Write Capacity Units \(p. 342\)](#)
- [Request Throttling and Burst Capacity \(p. 343\)](#)
- [Request Throttling and Adaptive Capacity \(p. 343\)](#)
- [Choosing Initial Throughput Settings \(p. 344\)](#)
- [Modifying Throughput Settings \(p. 344\)](#)

Read Capacity Units

A *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size.

Note

To learn more about DynamoDB read consistency models, see [Read Consistency \(p. 16\)](#).

For example, suppose that you create a table with 10 provisioned read capacity units. This allows you to perform 10 strongly consistent reads per second, or 20 eventually consistent reads per second, for items up to 4 KB.

Reading an item larger than 4 KB consumes more read capacity units. For example, a strongly consistent read of an item that is 8 KB (4 KB × 2) consumes 2 read capacity units. An eventually consistent read on that same item consumes only 1 read capacity unit.

Item sizes for reads are rounded up to the next 4 KB multiple. For example, reading a 3,500-byte item consumes the same throughput as reading a 4 KB item.

Capacity Unit Consumption for Reads

The following describes how DynamoDB read operations consume read capacity units:

- `GetItem`—Reads a single item from a table. To determine the number of capacity units that `GetItem` will consume, take the item size and round it up to the next 4 KB boundary. If you specified a strongly consistent read, this is the number of capacity units required. For an eventually consistent read (the default), divide this number by two.

For example, if you read an item that is 3.5 KB, DynamoDB rounds the item size to 4 KB. If you read an item of 10 KB, DynamoDB rounds the item size to 12 KB.

- `BatchGetItem`—Reads up to 100 items, from one or more tables. DynamoDB processes each item in the batch as an individual `GetItem` request, so DynamoDB first rounds up the size of each item to the next 4 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if `BatchGetItem` reads a 1.5 KB item and a 6.5 KB item, DynamoDB calculates the size as 12 KB (4 KB + 8 KB), not 8 KB (1.5 KB + 6.5 KB).
- `Query`—Reads multiple items that have the same partition key value. All items returned are treated as a single read operation, where DynamoDB computes the total size of all items and then rounds up to the next 4 KB boundary. For example, suppose your query returns 10 items whose combined size is 40.8 KB. DynamoDB rounds the item size for the operation to 44 KB. If a query returns 1500 items of 64 bytes each, the cumulative size is 96 KB.
- `Scan`—Reads all items in a table. DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan.

If you perform a read operation on an item that does not exist, DynamoDB still consumes provisioned read throughput: A strongly consistent read request consumes one read capacity unit, while an eventually consistent read request consumes 0.5 of a read capacity unit.

For any operation that returns items, you can request a subset of attributes to retrieve. However, doing so has no impact on the item size calculations. In addition, `Query` and `Scan` can return item counts instead of attribute values. Getting the count of items uses the same quantity of read capacity units and is subject to the same item size calculations. This is because DynamoDB has to read each item in order to increment the count.

Read Operations and Read Consistency

The preceding calculations assume strongly consistent read requests. For an eventually consistent read request, the operation consumes only half the capacity units. For an eventually consistent read, if the total item size is 80 KB, the operation consumes only 10 capacity units.

Write Capacity Units

A *write capacity unit* represents one write per second, for an item up to 1 KB in size.

For example, suppose that you create a table with 10 write capacity units. This allows you to perform 10 writes per second, for items up to 1 KB in size per second.

Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item consumes the same throughput as writing a 1 KB item.

Capacity Unit Consumption for Writes

The following describes how DynamoDB write operations consume write capacity units:

- `PutItem`—Writes a single item to a table. If an item with the same primary key exists in the table, the operation replaces the item. For calculating provisioned throughput consumption, the item size that matters is the larger of the two.
- `UpdateItem`—Modifies a single item in the table. DynamoDB considers the size of the item as it appears before and after the update. The provisioned throughput consumed reflects the larger of these item sizes. Even if you update just a subset of the item's attributes, `UpdateItem` will still consume the full amount of provisioned throughput (the larger of the "before" and "after" item sizes).
- `DeleteItem`—Removes a single item from a table. The provisioned throughput consumption is based on the size of the deleted item.
- `BatchWriteItem`—Writes up to 25 items to one or more tables. DynamoDB processes each item in the batch as an individual `PutItem` or `DeleteItem` request (updates are not supported). So DynamoDB first rounds up the size of each item to the next 1 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if `BatchWriteItem` writes a 500-byte item and a 3.5 KB item, DynamoDB calculates the size as 5 KB (1 KB + 4 KB), not 4 KB (500 bytes + 3.5 KB).

For `PutItem`, `UpdateItem`, and `DeleteItem` operations, DynamoDB rounds the item size up to the next 1 KB. For example, if you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.

`PutItem`, `UpdateItem`, and `DeleteItem` allow *conditional writes*, where you specify an expression that must evaluate to true in order for the operation to succeed. If the expression evaluates to false, DynamoDB still consumes write capacity units from the table:

- For an existing item, the number of write capacity units consumed depends on the size of the new item. (For example, a failed conditional write of a 1 KB item would consume one write capacity unit. If the new item were twice that size, the failed conditional write would consume two write capacity units.)
- For a new item, DynamoDB consumes one write capacity unit.

Request Throttling and Burst Capacity

If your application performs reads or writes at a higher rate than your table can support, DynamoDB begins to *throttle* those requests. When DynamoDB throttles a read or write, it returns a `ProvisionedThroughputExceededException` to the caller. The application can then take appropriate action, such as waiting for a short interval before retrying the request.

Note

We recommend that you use the AWS SDKs for software development. The AWS SDKs provide built-in support for retrying throttled requests; you do not need to write this logic yourself. For more information, see [Error Retries and Exponential Backoff \(p. 224\)](#).

The DynamoDB console displays Amazon CloudWatch metrics for your tables, so you can monitor throttled read requests and write requests. If you encounter excessive throttling, you should consider increasing your table's provisioned throughput settings.

In some cases, DynamoDB uses *burst capacity* to accommodate reads or writes in excess of your table's throughput settings. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled. For more information, see [Using Burst Capacity Effectively \(p. 900\)](#).

Request Throttling and Adaptive Capacity

DynamoDB automatically distributes your data across partitions, which are stored on multiple servers in the AWS Cloud (For more information, see [Partitions and Data Distribution \(p. 21\)](#)). It's not always possible to distribute read and write activity evenly all the time. When data access is imbalanced, a

"hot" partition can receive such a higher volume of read and write traffic compared to other partitions. Adaptive capacity works by automatically increasing throughput capacity for partitions that receive more traffic. For more information, see [Understanding DynamoDB Adaptive Capacity \(p. 900\)](#).

Choosing Initial Throughput Settings

Every application has different requirements for reading and writing from a database. When you are determining the initial throughput settings for a DynamoDB table, take the following inputs into consideration:

- **Item sizes.** Some items are small enough that they can be read or written using a single capacity unit. Larger items require multiple capacity units. By estimating the sizes of the items that will be in your table, you can specify accurate settings for your table's provisioned throughput.
- **Expected read and write request rates.** In addition to item size, you should estimate the number of reads and writes you need to perform per second.
- **Read consistency requirements.** Read capacity units are based on strongly consistent read operations, which consume twice as many database resources as eventually consistent reads. You should determine whether your application requires strongly consistent reads, or whether it can relax this requirement and perform eventually consistent reads instead. (Read operations in DynamoDB are eventually consistent, by default. You can request strongly consistent reads for these operations if necessary.)

For example, suppose that you want to read 80 items per second from a table. The items are 3 KB in size, and you want strongly consistent reads. For this scenario, each read requires one provisioned read capacity unit. To determine this number, you divide the item size of the operation by 4 KB, and then round up to the nearest whole number, as in this example:

- $3 \text{ KB} / 4 \text{ KB} = 0.75$, or **1** read capacity unit

For this scenario, you have to set the table's provisioned read throughput to 80 read capacity units:

- $1 \text{ read capacity unit per item} \times 80 \text{ reads per second} = \mathbf{80}$ read capacity units

Now suppose that you want to write 100 items per second to your table, and that the items are 512 bytes in size. For this scenario, each write requires one provisioned write capacity unit. To determine this number, you divide the item size of the operation by 1 KB, and then round up to the nearest whole number:

- $512 \text{ bytes} / 1 \text{ KB} = 0.5$, or **1**

For this scenario, you would want to set the table's provisioned write throughput to 100 write capacity units:

- $1 \text{ write capacity unit per item} \times 100 \text{ writes per second} = \mathbf{100}$ write capacity units

Note

For recommendations on provisioned throughput and related topics, see [Best Practices for Designing and Using Partition Keys Effectively \(p. 900\)](#).

Modifying Throughput Settings

If you have enabled DynamoDB auto scaling for a table, then its throughput capacity is dynamically adjusted in response to actual usage. No manual intervention is required.

You can modify your table's provisioned throughput settings using the AWS Management Console or the [UpdateTable](#) operation. For more information about throughput increases and decreases per day, see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

DynamoDB Item Sizes and Formats

DynamoDB tables are schemaless, except for the primary key, so the items in a table can all have different attributes, sizes, and data types.

The total size of an item is the sum of the lengths of its attribute names and values. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is $(\text{length of attribute name}) + (\text{number of UTF-8-encoded bytes})$.
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed. The size of a number is approximately $(\text{length of attribute name}) + (\text{1 byte per two significant digits}) + (\text{1 byte})$.
- A binary value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size. The size of a binary attribute is $(\text{length of attribute name}) + (\text{number of raw bytes})$.
- The size of a null attribute or a Boolean attribute is $(\text{length of attribute name}) + (\text{1 byte})$.
- An attribute of type `List` or `Map` requires 3 bytes of overhead, regardless of its contents. The size of a `List` or `Map` is $(\text{length of attribute name}) + \text{sum}(\text{size of nested elements}) + (\text{3 bytes})$. The size of an empty `List` or `Map` is $(\text{length of attribute name}) + (\text{3 bytes})$.

Note

We recommend that you choose shorter attribute names rather than long ones. This helps you reduce the amount of storage required, but also can lower the amount of RCU/WCUs you use.

Managing Throughput Capacity Automatically with DynamoDB Auto Scaling

Many database workloads are cyclical in nature or are difficult to predict in advance. For example, consider a social networking app where most of the users are active during daytime hours. The database must be able to handle the daytime activity, but there's no need for the same levels of throughput at night. Another example might be a new mobile gaming app that is experiencing rapid adoption. If the game becomes too popular, it could exceed the available database resources, resulting in slow performance and unhappy customers. These kinds of workloads often require manual intervention to scale database resources up or down in response to varying usage levels.

Amazon DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, Application Auto Scaling decreases the throughput so that you don't pay for unused provisioned capacity.

Note

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default. You can modify your auto scaling settings at any time. For more information, see [Using the AWS Management Console with DynamoDB Auto Scaling \(p. 348\)](#).

With Application Auto Scaling, you create a *scaling policy* for a table or a global secondary index. The scaling policy specifies whether you want to scale read capacity or write capacity (or both), and the minimum and maximum provisioned capacity unit settings for the table or index.

The scaling policy also contains a *target utilization*—the percentage of consumed provisioned throughput at a point in time. Application Auto Scaling uses a *target tracking* algorithm to adjust the provisioned throughput of the table (or index) upward or downward in response to actual workloads, so that the actual capacity utilization remains at or near your target utilization.

You can set the auto scaling target utilization values between 20 and 90 percent for your read and write capacity.

Note

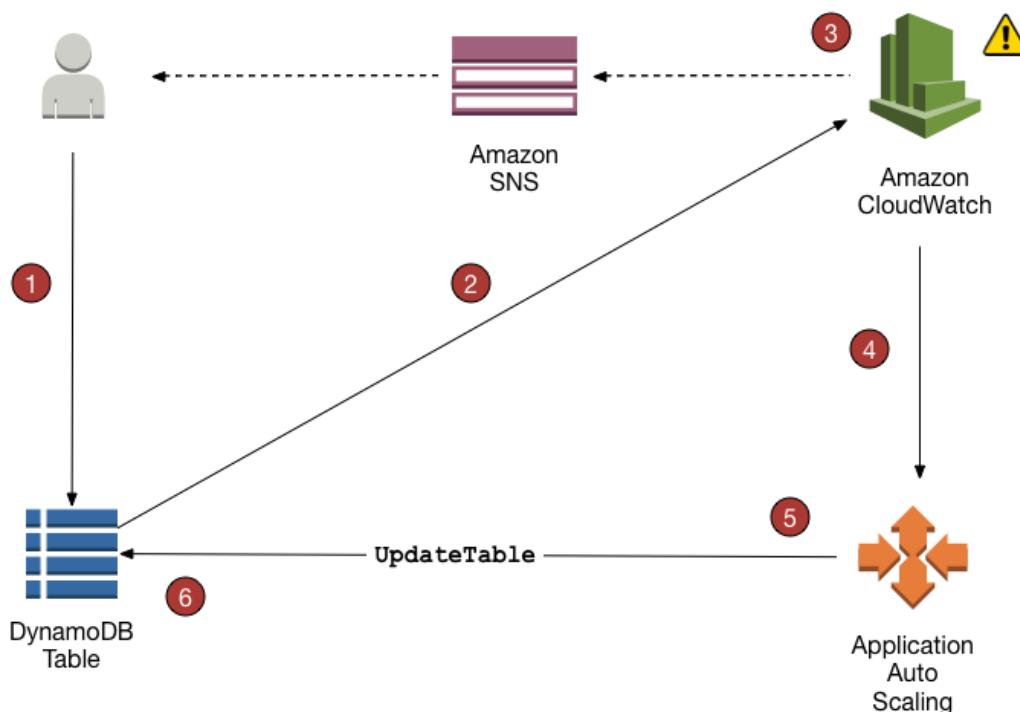
In addition to tables, DynamoDB auto scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table. When you create a scaling policy for a global secondary index, Application Auto Scaling adjusts the provisioned throughput settings for the index to ensure that its actual utilization stays at or near your desired utilization ratio.

How DynamoDB Auto Scaling Works

Note

To get started quickly with DynamoDB auto scaling, see [Using the AWS Management Console with DynamoDB Auto Scaling \(p. 348\)](#).

The following diagram provides a high-level overview of how DynamoDB auto scaling manages throughput capacity for a table.



The following steps summarize the auto scaling process as shown in the previous diagram:

1. You create an Application Auto Scaling policy for your DynamoDB table.
2. DynamoDB publishes consumed capacity metrics to Amazon CloudWatch.
3. If the table's consumed capacity exceeds your target utilization (or falls below the target) for a specific length of time, Amazon CloudWatch triggers an alarm. You can view the alarm on the console and receive notifications using Amazon Simple Notification Service (Amazon SNS).

4. The CloudWatch alarm invokes Application Auto Scaling to evaluate your scaling policy.
5. Application Auto Scaling issues an `UpdateTable` request to adjust your table's provisioned throughput.
6. DynamoDB processes the `UpdateTable` request, dynamically increasing (or decreasing) the table's provisioned throughput capacity so that it approaches your target utilization.

To understand how DynamoDB auto scaling works, suppose that you have a table named `ProductCatalog`. The table is bulk-loaded with data infrequently, so it doesn't incur very much write activity. However, it does experience a high degree of read activity, which varies over time. By monitoring the Amazon CloudWatch metrics for `ProductCatalog`, you determine that the table requires 1,200 read capacity units (to avoid DynamoDB throttling read requests when activity is at its peak). You also determine that `ProductCatalog` requires 150 read capacity units at a minimum, when read traffic is at its lowest point.

Within the range of 150 to 1,200 read capacity units, you decide that a target utilization of 70 percent would be appropriate for the `ProductCatalog` table. *Target utilization* is the ratio of consumed capacity units to provisioned capacity units, expressed as a percentage. Application Auto Scaling uses its target tracking algorithm to ensure that the provisioned read capacity of `ProductCatalog` is adjusted as required so that utilization remains at or near 70 percent.

Note

DynamoDB auto scaling modifies provisioned throughput settings only when the actual workload stays elevated (or depressed) for a sustained period of several minutes. The Application Auto Scaling target tracking algorithm seeks to keep the target utilization at or near your chosen value over the long term. Sudden, short-duration spikes of activity are accommodated by the table's built-in burst capacity. For more information, see [Using Burst Capacity Effectively \(p. 900\)](#).

To enable DynamoDB auto scaling for the `ProductCatalog` table, you create a scaling policy. This policy specifies the following:

- The table or global secondary index that you want to manage
- Which capacity type to manage (read capacity or write capacity)
- The upper and lower boundaries for the provisioned throughput settings
- Your target utilization

When you create a scaling policy, Application Auto Scaling creates a pair of Amazon CloudWatch alarms on your behalf. Each pair represents your upper and lower boundaries for provisioned throughput settings. These CloudWatch alarms are triggered when the table's actual utilization deviates from your target utilization for a sustained period of time.

When one of the CloudWatch alarms is triggered, Amazon SNS sends you a notification (if you have enabled it). The CloudWatch alarm then invokes Application Auto Scaling, which in turn notifies DynamoDB to adjust the `ProductCatalog` table's provisioned capacity upward or downward as appropriate.

Usage Notes

Before you begin using DynamoDB auto scaling, you should be aware of the following:

- DynamoDB auto scaling can increase read capacity or write capacity as often as necessary, in accordance with your auto scaling policy. All DynamoDB limits remain in effect, as described in [Limits in DynamoDB](#).
- DynamoDB auto scaling doesn't prevent you from manually modifying provisioned throughput settings. These manual adjustments don't affect any existing CloudWatch alarms that are related to DynamoDB auto scaling.

- If you enable DynamoDB auto scaling for a table that has one or more global secondary indexes, we highly recommend that you also apply auto scaling uniformly to those indexes. You can do this by choosing **Apply same settings to global secondary indexes** in the AWS Management Console. For more information, see [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 349\)](#).

Using the AWS Management Console with DynamoDB Auto Scaling

When you use the AWS Management Console to create a new table, Amazon DynamoDB auto scaling is enabled for that table by default. You can also use the console to enable auto scaling for existing tables, modify auto scaling settings, or disable auto scaling.

Note

For more advanced features like setting scale-in and scale-out cooldown times, use the AWS Command Line Interface (AWS CLI) to manage DynamoDB auto scaling. For more information, see [Using the AWS CLI to Manage DynamoDB Auto Scaling \(p. 350\)](#).

Topics

- [Before You Begin: Granting User Permissions for DynamoDB Auto Scaling \(p. 348\)](#)
- [Creating a New Table with Auto Scaling Enabled \(p. 348\)](#)
- [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 349\)](#)
- [Viewing Auto Scaling Activities on the Console \(p. 350\)](#)
- [Modifying or Disabling DynamoDB Auto Scaling Settings \(p. 350\)](#)

Before You Begin: Granting User Permissions for DynamoDB Auto Scaling

In AWS Identity and Access Management (IAM), the AWS managed policy `DynamoDBFullAccess` provides the required permissions for using the DynamoDB console. However, for DynamoDB auto scaling, IAM users require additional privileges.

Important

`application-autoscaling:*` permissions are required to delete an auto scaling-enabled table. The AWS managed policy `DynamoDBFullAccess` includes such permissions.

To set up an IAM user for DynamoDB console access and DynamoDB auto scaling, add the following policy.

To attach the `AmazonDynamoDBFullAccess` policy

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. On the IAM console dashboard, choose **Users**, and then choose your IAM user from the list.
3. On the **Summary** page, choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. From the list of policies, choose `AmazonDynamoDBFullAccess`, and then choose **Next: Review**.
6. Choose **Add permissions**.

Creating a New Table with Auto Scaling Enabled

Note

DynamoDB auto scaling requires the presence of a service linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-Linked Roles for Application Auto Scaling](#) in the *Application Auto Scaling User Guide*.

To create a new table with auto scaling enabled

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. On the **Create DynamoDB table** page, enter a **Table name** and **Primary key** details.
4. Ensure that **Use default settings** is selected. (Your AWS account already has `AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`.)

Otherwise, for custom settings:

1. Clear **Use default settings**.
2. In the **Auto scaling** section, set the parameter settings and ensure that `AWSServiceRoleForApplicationAutoScaling_DynamoDBTable` is selected.
5. When the settings are as you want them, choose **Create**. Your table is created with the auto scaling parameters.

Enabling DynamoDB Auto Scaling on Existing Tables

Note

DynamoDB auto scaling requires the presence of a service linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-Linked Roles for Application Auto Scaling](#).

If you have never used DynamoDB auto scaling before, see [Creating a New Table with Auto Scaling Enabled \(p. 348\)](#).

To enable DynamoDB auto scaling for an existing table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose the table that you want to work with, and then choose **Capacity**.
3. In the **Auto Scaling** section, do the following:
 - Select **Read capacity**, **Write capacity**, or both. (For write capacity, you can choose **Same settings as read**.) For each of these, do the following:
 - **Target utilization**—Enter your target utilization percentage for the table.
 - **Minimum provisioned capacity**—Enter your lower boundary for the auto scaling range.
 - **Maximum provisioned capacity**—Enter your upper boundary for the auto scaling range.
 - **Apply same settings to global secondary indexes**—Keep this option at its default setting (enabled).

Note

For best performance, we recommend that you enable **Apply same settings to global secondary indexes**. This option allows DynamoDB auto scaling to uniformly scale all the global secondary indexes on the base table. This includes existing global secondary indexes, and any others that you create for this table in the future. With this option enabled, you can't set a scaling policy on an individual global secondary index.

(For **Write capacity**, you can choose **Same settings as read**.)

In the **IAM Role** section, ensure that `AWSServiceRoleForApplicationAutoScaling_DynamoDBTable` is selected.

4. When the settings are as you want them, choose **Save**.

Viewing Auto Scaling Activities on the Console

As your application drives read and write traffic to your table, DynamoDB auto scaling dynamically modifies the table's throughput settings.

To view these auto scaling activities on the DynamoDB console, choose the table that you want to work with. Choose **Capacity**, and then expand the **Scaling activities** section. When your table's throughput settings are modified, you see informational messages here.

Modifying or Disabling DynamoDB Auto Scaling Settings

You can use the AWS Management Console to modify your DynamoDB auto scaling settings. To do this, go to the **Capacity** tab for your table, and modify the settings in the **Auto Scaling** section. For more information about these settings, see [Enabling DynamoDB Auto Scaling on Existing Tables \(p. 349\)](#).

To disable DynamoDB auto scaling, go to the **Capacity** tab for your table and clear **Read capacity**, **Write capacity**, or both.

Using the AWS CLI to Manage DynamoDB Auto Scaling

Instead of using the AWS Management Console, you can use the AWS Command Line Interface (AWS CLI) to manage Amazon DynamoDB auto scaling. The tutorial in this section demonstrates how to install and configure the AWS CLI for managing DynamoDB auto scaling. In this tutorial, you do the following:

- Create a DynamoDB table named `TestTable`. The initial throughput settings are 5 read capacity units and 5 write capacity units.
- Create an Application Auto Scaling policy for `TestTable`. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity. The range for this metric is between 5 and 10 write capacity units. (Application Auto Scaling is not allowed to adjust the throughput beyond this range.)
- Run a Python program to drive write traffic to `TestTable`. When the target ratio exceeds 50 percent for a sustained period of time, Application Auto Scaling notifies DynamoDB to adjust the throughput of `TestTable` upward to maintain the 50 percent target utilization.
- Verify that DynamoDB has successfully adjusted the provisioned write capacity for `TestTable`.

Topics

- [Before You Begin \(p. 350\)](#)
- [Step 1: Create a DynamoDB Table \(p. 351\)](#)
- [Step 2: Register a Scalable Target \(p. 351\)](#)
- [Step 3: Create a Scaling Policy \(p. 352\)](#)
- [Step 4: Drive Write Traffic to TestTable \(p. 353\)](#)
- [Step 5: View Application Auto Scaling Actions \(p. 354\)](#)
- [\(Optional\) Step 6: Clean Up \(p. 355\)](#)

Before You Begin

Complete the following tasks before starting the tutorial.

Install the AWS CLI

If you haven't already done so, you must install and configure the AWS CLI. To do this, follow these instructions in the *AWS Command Line Interface User Guide*:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

Install Python

Part of this tutorial requires you to run a Python program (see [Step 4: Drive Write Traffic to TestTable \(p. 353\)](#)). If you don't already have it installed, you can [download Python](#).

Step 1: Create a DynamoDB Table

In this step, you use the AWS CLI to create `TestTable`. The primary key consists of `pk` (partition key) and `sk` (sort key). Both of these attributes are of type `Number`. The initial throughput settings are 5 read capacity units and 5 write capacity units.

1. Use the following AWS CLI command to create the table.

```
aws dynamodb create-table \  
  --table-name TestTable \  
  --attribute-definitions \  
    AttributeName=pk,AttributeType=N \  
    AttributeName=sk,AttributeType=N \  
  --key-schema \  
    AttributeName=pk,KeyType=HASH \  
    AttributeName=sk,KeyType=RANGE \  
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

2. To check the status of the table, use the following command.

```
aws dynamodb describe-table \  
  --table-name TestTable \  
  --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

The table is ready for use when its status is `ACTIVE`.

Step 2: Register a Scalable Target

Next you register the table's write capacity as a scalable target with Application Auto Scaling. This allows Application Auto Scaling to adjust the provisioned write capacity for `TestTable`, but only within the range of 5–10 capacity units.

Note

DynamoDB auto scaling requires the presence of a service linked role (`AWSServiceRoleForApplicationAutoScaling_DynamoDBTable`) that performs auto scaling actions on your behalf. This role is created automatically for you. For more information, see [Service-Linked Roles for Application Auto Scaling](#) in the *Application Auto Scaling User Guide*.

1. Enter the following command to register the scalable target.

```
aws application-autoscaling register-scalable-target \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable" \  
  --scalable-dimension "dynamodb:table:WriteCapacityUnits" \  
  --min-capacity 5 \  
  --max-capacity 10
```

```
--max-capacity 10
```

2. To verify the registration, use the following command.

```
aws application-autoscaling describe-scalable-targets \
--service-namespace dynamodb \
--resource-id "table/TestTable"
```

Note

You can also register a scalable target against a global secondary index. For example, for a global secondary index ("test-index"), the resource ID and scalable dimension arguments are updated appropriately.

```
aws application-autoscaling register-scalable-target \
--service-namespace dynamodb \
--resource-id "table/TestTable/index/test-index" \
--scalable-dimension "dynamodb:index:WriteCapacityUnits" \
--min-capacity 5 \
--max-capacity 10
```

Step 3: Create a Scaling Policy

In this step, you create a scaling policy for `TestTable`. The policy defines the details under which Application Auto Scaling can adjust your table's provisioned throughput, and what actions to take when it does so. You associate this policy with the scalable target that you defined in the previous step (write capacity units for the `TestTable` table).

The policy contains the following elements:

- `PredefinedMetricSpecification`—The metric that Application Auto Scaling is allowed to adjust. For DynamoDB, the following values are valid values for `PredefinedMetricType`:
 - `DynamoDBReadCapacityUtilization`
 - `DynamoDBWriteCapacityUtilization`
- `ScaleOutCooldown`—The minimum amount of time (in seconds) between each Application Auto Scaling event that increases provisioned throughput. This parameter allows Application Auto Scaling to continuously, but not aggressively, increase the throughput in response to real-world workloads. The default setting for `ScaleOutCooldown` is 0.
- `ScaleInCooldown`—The minimum amount of time (in seconds) between each Application Auto Scaling event that decreases provisioned throughput. This parameter allows Application Auto Scaling to decrease the throughput gradually and predictably. The default setting for `ScaleInCooldown` is 0.
- `TargetValue`—Application Auto Scaling ensures that the ratio of consumed capacity to provisioned capacity stays at or near this value. You define `TargetValue` as a percentage.

Note

To further understand how `TargetValue` works, suppose that you have a table with a provisioned throughput setting of 200 write capacity units. You decide to create a scaling policy for this table, with a `TargetValue` of 70 percent.

Now suppose that you begin driving write traffic to the table so that the actual write throughput is 150 capacity units. The consumed-to-provisioned ratio is now (150 / 200), or 75 percent. This ratio exceeds your target, so Application Auto Scaling increases the provisioned write capacity

to 215 so that the ratio is (150 / 215), or 69.77 percent—as close to your `TargetValue` as possible, but not exceeding it.

For `TestTable`, you set `TargetValue` to 50 percent. Application Auto Scaling adjusts the table's provisioned throughput within the range of 5–10 capacity units (see [Step 2: Register a Scalable Target \(p. 351\)](#)) so that the consumed-to-provisioned ratio remains at or near 50 percent. You set the values for `ScaleOutCooldown` and `ScaleInCooldown` to 60 seconds.

1. Create a file named `scaling-policy.json` with the following contents.

```
{
    "PredefinedMetricSpecification": {
        "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
    },
    "ScaleOutCooldown": 60,
    "ScaleInCooldown": 60,
    "TargetValue": 50.0
}
```

2. Use the following AWS CLI command to create the policy.

```
aws application-autoscaling put-scaling-policy \
--service-namespace dynamodb \
--resource-id "table/TestTable" \
--scalable-dimension "dynamodb:table:WriteCapacityUnits" \
--policy-name "MyScalingPolicy" \
--policy-type "TargetTrackingScaling" \
--target-tracking-scaling-policy-configuration file://scaling-policy.json
```

3. In the output, note that Application Auto Scaling has created two Amazon CloudWatch alarms—one each for the upper and lower boundary of the scaling target range.
4. Use the following AWS CLI command to view more details about the scaling policy.

```
aws application-autoscaling describe-scaling-policies \
--service-namespace dynamodb \
--resource-id "table/TestTable" \
--policy-name "MyScalingPolicy"
```

5. In the output, verify that the policy settings match your specifications from [Step 2: Register a Scalable Target \(p. 351\)](#) and [Step 3: Create a Scaling Policy \(p. 352\)](#).

Step 4: Drive Write Traffic to TestTable

Now you can test your scaling policy by writing data to `TestTable`. To do this, you run a Python program.

1. Create a file named `bulk-load-test-table.py` with the following contents.

```
import boto3
dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table("TestTable")

filler = "x" * 100000
```

```
i = 0
while (i < 10):
    j = 0
    while (j < 10):
        print (i, j)

        table.put_item(
            Item={
                'pk':i,
                'sk':j,
                'filler':{"S":filler}
            }
        )
        j += 1
    i += 1
```

2. Enter the following command to run the program.

```
python bulk-load-test-table.py
```

The provisioned write capacity for `TestTable` is very low (5 write capacity units), so the program stalls occasionally due to write throttling. This is expected behavior.

Let the program continue running while you move on to the next step.

Step 5: View Application Auto Scaling Actions

In this step, you view the Application Auto Scaling actions that are initiated on your behalf. You also verify that Application Auto Scaling has updated the provisioned write capacity for `TestTable`.

1. Enter the following command to view the Application Auto Scaling actions.

```
aws application-autoscaling describe-scaling-activities \
--service-namespace dynamodb
```

Rerun this command occasionally, while the Python program is running. (It takes several minutes before your scaling policy is invoked.) You should eventually see the following output.

```
...
{
    "ScalableDimension": "dynamodb:table:WriteCapacityUnits",
    "Description": "Setting write capacity units to 10.",
    "ResourceId": "table/TestTable",
    "ActivityId": "0cc6fb03-2a7c-4b51-b67f-217224c6b656",
    "StartTime": 1489088210.175,
    "ServiceNamespace": "dynamodb",
    "EndTime": 1489088246.85,
    "Cause": "monitor alarm AutoScaling-table/TestTable-AlarmHigh-1bb3c8db-1b97-4353-
baf1-4def76f4e1b9 in state ALARM triggered policy MyScalingPolicy",
    "StatusMessage": "Successfully set write capacity units to 10. Change successfully
fulfilled by dynamodb.",
    "StatusCode": "Successful"
},
```

This indicates that Application Auto Scaling has issued an `UpdateTable` request to DynamoDB.

2. Enter the following command to verify that DynamoDB increased the table's write capacity.

```
aws dynamodb describe-table \  
  --table-name TestTable \  
  --query "Table.[TableName,TableStatus,ProvisionedThroughput]"
```

The WriteCapacityUnits should have been scaled from 5 to 10.

(Optional) Step 6: Clean Up

In this tutorial, you created several resources. You can delete these resources if you no longer need them.

1. Delete the scaling policy for `TestTable`.

```
aws application-autoscaling delete-scaling-policy \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable" \  
  --scalable-dimension "dynamodb:table:WriteCapacityUnits" \  
  --policy-name "MyScalingPolicy"
```

2. Deregister the scalable target.

```
aws application-autoscaling deregister-scaling-target \  
  --service-namespace dynamodb \  
  --resource-id "table/TestTable" \  
  --scalable-dimension "dynamodb:table:WriteCapacityUnits"
```

3. Delete the `TestTable` table.

```
aws dynamodb delete-table --table-name TestTable
```

Using the AWS SDK to Configure Auto Scaling on Amazon DynamoDB Tables

In addition to using the AWS Management Console and the AWS Command Line Interface (AWS CLI), you can write applications that interact with Amazon DynamoDB auto scaling. This section contains two Java programs that you can use to test this functionality:

- `EnableDynamoDBAutoscaling.java`
- `DisableDynamoDBAutoscaling.java`

Enabling Application Auto Scaling for a Table

The following program shows an example of setting up an auto scaling policy for a DynamoDB table (`TestTable`). It proceeds as follows:

- The program registers write capacity units as a scalable target for `TestTable`. The range for this metric is between 5 and 10 write capacity units.

- After the scalable target is created, the program builds a target tracking configuration. The policy seeks to maintain a 50 percent target ratio between consumed write capacity and provisioned write capacity.
- The program then creates the scaling policy, based on the target tracking configuration.

The program requires that you provide an Amazon Resource Name (ARN) for a valid Application Auto Scaling service linked role. (For example: arn:aws:iam::122517410325:role/AWSServiceRoleForApplicationAutoScaling_DynamoDBTable.) In the following program, replace SERVICE_ROLE_ARN_Goes_Here with the actual ARN.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.codesamples.autoscaling;

import com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import
    com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClientBuilder;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.MetricType;
import com.amazonaws.services.applicationautoscaling.model.PolicyType;
import com.amazonaws.services.applicationautoscaling.model.PredefinedMetricSpecification;
import com.amazonaws.services.applicationautoscaling.model.PutScalingPolicyRequest;
import com.amazonaws.services.applicationautoscaling.model.RegisterScalableTargetRequest;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;
import
    com.amazonaws.services.applicationautoscaling.model.TargetTrackingScalingPolicyConfiguration;

public class EnableDynamoDBAutoscaling {

    static AWSApplicationAutoScalingClient aaClient = (AWSApplicationAutoScalingClient)
AWSApplicationAutoScalingClientBuilder.standard().build();

    public static void main(String args[]) {

        ServiceNamespace ns = ServiceNamespace.Dynamodb;
        ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
        String resourceId = "table/TestTable";

        // Define the scalable target
        RegisterScalableTargetRequest rstRequest = new RegisterScalableTargetRequest()
            .withServiceNamespace(ns)
            .withResourceId(resourceId)
            .withScalableDimension(tableWCUs)
            .withMinCapacity(5)
            .withMaxCapacity(10)
            .withRoleARN("SERVICE_ROLE_ARN_Goes_Here");

        try {

```

```

        aaClient.registerScalableTarget(rstRequest);
    } catch (Exception e) {
        System.err.println("Unable to register scalable target: ");
        System.err.println(e.getMessage());
    }

    // Verify that the target was created
    DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
        .withServiceNamespace(ns)
        .withScalableDimension(tableWCUs)
        .withResourceIds(resourceID);
    try {
        DescribeScalableTargetsResult dsaResult =
            aaClient.describeScalableTargets(dscRequest);
        System.out.println("DescribeScalableTargets result: ");
        System.out.println(dsaResult);
        System.out.println();
    } catch (Exception e) {
        System.err.println("Unable to describe scalable target: ");
        System.err.println(e.getMessage());
    }

    System.out.println();

    // Configure a scaling policy
    TargetTrackingScalingPolicyConfiguration targetTrackingScalingPolicyConfiguration =
        new TargetTrackingScalingPolicyConfiguration()
        .withPredefinedMetricSpecification(
            new PredefinedMetricSpecification()
                .withPredefinedMetricType(MetricType.DynamoDBWriteCapacityUtilization))
        .withTargetValue(50.0)
        .withScaleInCooldown(60)
        .withScaleOutCooldown(60);

    // Create the scaling policy, based on your configuration
    PutScalingPolicyRequest pspRequest = new PutScalingPolicyRequest()
        .withServiceNamespace(ns)
        .withScalableDimension(tableWCUs)
        .withResourceId(resourceID)
        .withPolicyName("MyScalingPolicy")
        .withPolicyType(PolicyType.TargetTrackingScaling)
        .withTargetTrackingScalingPolicyConfiguration(targetTrackingScalingPolicyConfiguration);

    try {
        aaClient.putScalingPolicy(pspRequest);
    } catch (Exception e) {
        System.err.println("Unable to put scaling policy: ");
        System.err.println(e.getMessage());
    }

    // Verify that the scaling policy was created
    DescribeScalingPoliciesRequest dspRequest = new DescribeScalingPoliciesRequest()
        .withServiceNamespace(ns)
        .withScalableDimension(tableWCUs)
        .withResourceId(resourceID);

    try {
        DescribeScalingPoliciesResult dspResult =
            aaClient.describeScalingPolicies(dspRequest);
        System.out.println("DescribeScalingPolicies result: ");
        System.out.println(dspResult);
    } catch (Exception e) {
        e.printStackTrace();
        System.err.println("Unable to describe scaling policy: ");
        System.err.println(e.getMessage());
    }
}

```

```
    }
}
```

Disabling Application Auto Scaling for a Table

The following program reverses the previous process. It removes the auto scaling policy and then deregisters the scalable target.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.codesamples.autoscaling;

import com.amazonaws.services.applicationautoscaling.AWSApplicationAutoScalingClient;
import com.amazonaws.services.applicationautoscaling.model.DeleteScalingPolicyRequest;
import com.amazonaws.services.applicationautoscaling.model.DeregisterScalableTargetRequest;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsRequest;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalableTargetsResult;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesRequest;
import com.amazonaws.services.applicationautoscaling.model.DescribeScalingPoliciesResult;
import com.amazonaws.services.applicationautoscaling.model.ScalableDimension;
import com.amazonaws.services.applicationautoscaling.model.ServiceNamespace;

public class DisableDynamoDBAutoscaling {

    static AWSApplicationAutoScalingClient aaClient = new
AWSApplicationAutoScalingClient();

    public static void main(String args[]) {

        ServiceNamespace ns = ServiceNamespace.Dynamodb;
        ScalableDimension tableWCUs = ScalableDimension.DynamodbTableWriteCapacityUnits;
        String resourceId = "table/TestTable";

        // Delete the scaling policy
        DeleteScalingPolicyRequest delSPRequest = new DeleteScalingPolicyRequest()
            .withServiceNamespace(ns)
            .withScalableDimension(tableWCUs)
            .withResourceId(resourceId)
            .withPolicyName("MyScalingPolicy");

        try {
            aaClient.deleteScalingPolicy(delSPRequest);
        } catch (Exception e) {
            System.err.println("Unable to delete scaling policy: ");
            System.err.println(e.getMessage());
        }

        // Verify that the scaling policy was deleted
        DescribeScalingPoliciesRequest descSPRequest = new DescribeScalingPoliciesRequest()
            .withServiceNamespace(ns)
```

```

.withScalableDimension(tableWCUs)
.withResourceId(resourceID);

try {
    DescribeScalingPoliciesResult dspResult =
aaClient.describeScalingPolicies(descSPRequest);
    System.out.println("DescribeScalingPolicies result: ");
    System.out.println(dspResult);
} catch (Exception e) {
    e.printStackTrace();
    System.err.println("Unable to describe scaling policy: ");
    System.err.println(e.getMessage());
}

System.out.println();

// Remove the scalable target
DeregisterScalableTargetRequest delSTRequest = new DeregisterScalableTargetRequest()
.withServiceNamespace(ns)
.withScalableDimension(tableWCUs)
.withResourceId(resourceID);

try {
    aaClient.deregisterScalableTarget(delSTRequest);
} catch (Exception e) {
    System.err.println("Unable to deregister scalable target: ");
    System.err.println(e.getMessage());
}

// Verify that the scalable target was removed
DescribeScalableTargetsRequest dscRequest = new DescribeScalableTargetsRequest()
.withServiceNamespace(ns)
.withScalableDimension(tableWCUs)
.withResourceIds(resourceID);

try {
    DescribeScalableTargetsResult dsaResult =
aaClient.describeScalableTargets(dscRequest);
    System.out.println("DescribeScalableTargets result: ");
    System.out.println(dsaResult);
    System.out.println();
} catch (Exception e) {
    System.err.println("Unable to describe scalable target: ");
    System.err.println(e.getMessage());
}
}
}

```

Adding Tags and Labels to Resources

You can label Amazon DynamoDB resources using *tags*. Tags let you categorize your resources in different ways, for example, by purpose, owner, environment, or other criteria. Tags can help you do the following:

- Quickly identify a resource based on the tags that you assigned to it.
 - See AWS bills broken down by tags.

Note

Any local secondary indexes (LSI) and global secondary indexes (GSI) related to tagged tables are labeled with the same tags automatically. Currently, DynamoDB Streams usage cannot be tagged.

Tagging is supported by AWS services like Amazon EC2, Amazon S3, DynamoDB, and more. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

To get started with tagging, do the following:

1. Understand [Tagging Restrictions in DynamoDB \(p. 360\)](#).
2. Create tags by using [Tagging Resources in DynamoDB \(p. 360\)](#).
3. Use [Cost Allocation Reports \(p. 362\)](#) to track your AWS costs per active tag.

Finally, it is good practice to follow optimal tagging strategies. For information, see [AWS Tagging Strategies](#).

Tagging Restrictions in DynamoDB

Each tag consists of a key and a value, both of which you define. The following restrictions apply:

- Each DynamoDB table can have only one tag with the same key. If you try to add an existing tag (same key), the existing tag value is updated to the new value.
- Tag keys and values are case sensitive.
- The maximum key length is 128 Unicode characters.
- The maximum value length is 256 Unicode characters.
- The allowed characters are letters, white space, and numbers, plus the following special characters: + - = . _ : /
- The maximum number of tags per resource is 50.
- AWS-assigned tag names and values are automatically assigned the `aws:` prefix, which you can't assign. AWS-assigned tag names don't count toward the tag limit of 50. User-assigned tag names have the prefix `user:` in the cost allocation report.
- You can't backdate the application of a tag.

Tagging Resources in DynamoDB

You can use the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI) to add, list, edit, or delete tags. You can then activate these user-defined tags so that they appear on the AWS Billing and Cost Management console for cost allocation tracking. For more information, see [Cost Allocation Reports \(p. 362\)](#).

For bulk editing, you can also use Tag Editor on the AWS Management Console. For more information, see [Working with Tag Editor](#).

To use the DynamoDB API instead, see the following operations in the [Amazon DynamoDB API Reference](#):

- [TagResource](#)
- [UntagResource](#)
- [ListTagsOfResource](#)

Topics

- [Adding Tags to New or Existing Tables \(Console\) \(p. 361\)](#)
- [Adding Tags to New or Existing Tables \(AWS CLI\) \(p. 361\)](#)

Adding Tags to New or Existing Tables (Console)

You can use the DynamoDB console to add tags to new tables when you create them, or to add, edit, or delete tags for existing tables.

To tag resources on creation (console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**, and then choose **Create table**.
3. On the **Create DynamoDB table** page, provide a name and primary key. Choose **Add tags** and enter the tags that you want to use.

For information about tag structure, see [Tagging Restrictions in DynamoDB \(p. 360\)](#).

For more information about creating tables, see [Basic Operations on DynamoDB Tables \(p. 335\)](#).

To tag existing resources (console)

Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.

1. In the navigation pane, choose **Tables**.
2. Choose a table in the list, and then choose the **Tags** tab to add, edit, or delete your tags.

Adding Tags to New or Existing Tables (AWS CLI)

The following examples show how to use the AWS CLI to specify tags when you create tables and indexes, and to tag existing resources.

To tag resources on creation (AWS CLI)

- The following example creates a new `Movies` table and adds the `Owner` tag with a value of `blueTeam`:

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

To tag existing resources (AWS CLI)

- The following example adds the `Owner` tag with a value of `blueTeam` for the `Movies` table:

```
aws dynamodb tag-resource \
--resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies \
--tags Key=Owner,Value=blueTeam
```

To list all tags for a table (AWS CLI)

- The following example lists all the tags that are associated with the `Movies` table:

```
aws dynamodb list-tags-of-resource \
--resource-arn arn:aws:dynamodb:us-east-1:123456789012:table/Movies
```

Cost Allocation Reports

AWS uses tags to organize resource costs on your cost allocation report. AWS provides two types of cost allocation tags:

- An AWS-generated tag. AWS defines, creates, and applies this tag for you.
- User-defined tags. You define, create, and apply these tags.

You must activate both types of tags separately before they can appear in Cost Explorer or on a cost allocation report.

To activate AWS-generated tags:

- Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
- In the navigation pane, choose **Cost Allocation Tags**.
- Under **AWS-Generated Cost Allocation Tags**, choose **Activate**.

To activate user-defined tags:

- Sign in to the AWS Management Console and open the Billing and Cost Management console at <https://console.aws.amazon.com/billing/home#/>.
- In the navigation pane, choose **Cost Allocation Tags**.
- Under **User-Defined Cost Allocation Tags**, choose **Activate**.

After you create and activate tags, AWS generates a cost allocation report with your usage and costs grouped by your active tags. The cost allocation report includes all of your AWS costs for each billing period. The report includes both tagged and untagged resources, so that you can clearly organize the charges for resources.

Note

Currently, any data transferred out from DynamoDB won't be broken down by tags on cost allocation reports.

For more information, see [Using Cost Allocation Tags](#).

Working with DynamoDB Tables in Java

You can use the AWS SDK for Java to create, update, and delete Amazon DynamoDB tables, list all the tables in your account, or get information about a specific table.

Topics

- [Creating a Table \(p. 363\)](#)
- [Updating a Table \(p. 364\)](#)
- [Deleting a Table \(p. 364\)](#)

- [Listing Tables \(p. 365\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for Java Document API \(p. 365\)](#)

Creating a Table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values. The following code snippet creates an example table that uses a numeric type attribute ID as its primary key.

To create a table using the AWS SDK for Java API

1. Create an instance of the `DynamoDB` class.
2. Instantiate a `CreateTableRequest` to provide the request information.

You must provide the table name, attribute definitions, key schema, and provisioned throughput values.

3. Execute the `createTable` method by providing the request object as a parameter.

The following code example demonstrates the preceding steps.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

List<AttributeDefinition> attributeDefinitions= new ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH));

CreateTableRequest request = new CreateTableRequest()
    .withTableName(tableName)
    .withKeySchema(keySchema)
    .withAttributeDefinitions(attributeDefinitions)
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits(5L)
        .withWriteCapacityUnits(6L));

Table table = dynamoDB.createTable(request);

table.waitForActive();
```

The table is not ready for use until DynamoDB creates it and sets its status to *ACTIVE*. The `createTable` request returns a `Table` object that you can use to obtain more information about the table.

Example

```
TableDescription tableDescription =
    dynamoDB.getTable(tableName).describe();

System.out.printf("%s: %s \t ReadCapacityUnits: %d \t WriteCapacityUnits: %d",
    tableDescription.getTableStatus(),
    tableDescription.getTableName(),
    tableDescription.getProvisionedThroughput().getReadCapacityUnits(),
    tableDescription.getProvisionedThroughput().getWriteCapacityUnits());
```

You can call the `describe` method of the client to get table information at any time.

Example

```
TableDescription tableDescription = dynamoDB.getTable(tableName).describe();
```

Updating a Table

You can update only the provisioned throughput values of an existing table. Depending on your application requirements, you might need to update these values.

Note

For more information about throughput increases and decreases per day, see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

To update a table using the AWS SDK for Java API

1. Create an instance of the `Table` class.
2. Create an instance of the `ProvisionedThroughput` class to provide the new throughput values.
3. Execute the `updateTable` method by providing the `ProvisionedThroughput` instance as a parameter.

The following code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

ProvisionedThroughput provisionedThroughput = new ProvisionedThroughput()
    .withReadCapacityUnits(15L)
    .withWriteCapacityUnits(12L);

table.updateTable(provisionedThroughput);

table.waitForActive();
```

Deleting a Table

To delete a table using the AWS SDK for Java API

1. Create an instance of the `Table` class.
2. Create an instance of the `DeleteTableRequest` class and provide the table name that you want to delete.
3. Execute the `deleteTable` method by providing the `Table` instance as a parameter.

The following code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

table.delete();
```

```
table.waitForDelete();
```

Listing Tables

To list tables in your account, create an instance of DynamoDB and execute the `listTables` method. The [ListTables](#) operation requires no parameters.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableCollection<ListTablesResult> tables = dynamoDB.listTables();
Iterator<Table> iterator = tables.iterator();

while (iterator.hasNext()) {
    Table table = iterator.next();
    System.out.println(table.getTableName());
}
```

Example: Create, Update, Delete, and List Tables Using the AWS SDK for Java Document API

The following code example uses the AWS SDK for Java Document API to create, update, and delete an Amazon DynamoDB table (`ExampleTable`). As part of the table update, it increases the provisioned throughput values. The example also lists all the tables in your account and gets the description of a specific table. For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.TableCollection;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ListTablesResult;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.TableDescription;
```

```

public class DocumentAPITableExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "ExampleTable";

    public static void main(String[] args) throws Exception {

        createExampleTable();
        listMyTables();
        getTableInformation();
        updateExampleTable();

        deleteExampleTable();
    }

    static void createExampleTable() {

        try {

            List<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
            attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

            List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
            keySchema.add(new
KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition
// key

            CreateTableRequest request = new
CreateTableRequest().withTableName(tableName).withKeySchema(keySchema)
                .withAttributeDefinitions(attributeDefinitions).withProvisionedThroughput(
                    new
ProvisionedThroughput().withReadCapacityUnits(5L).withWriteCapacityUnits(6L));

            System.out.println("Issuing CreateTable request for " + tableName);
            Table table = dynamoDB.createTable(request);

            System.out.println("Waiting for " + tableName + " to be created...this may take
a while...");
            table.waitForActive();

            getTableInformation();

        }
        catch (Exception e) {
            System.err.println("CreateTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    static void listMyTables() {

        TableCollection<ListTablesResult> tables = dynamoDB.listTables();
        Iterator<Table> iterator = tables.iterator();

        System.out.println("Listing table names");

        while (iterator.hasNext()) {
            Table table = iterator.next();
            System.out.println(table.getTableName());
        }
    }
}

```

```

        }

    static void getTableInformation() {
        System.out.println("Describing " + tableName);

        TableDescription tableDescription = dynamoDB.getTable(tableName).describe();
        System.out.format(
            "Name: %s:\n" + "Status: %s \n" + "Provisioned Throughput (read capacity units/
sec): %d \n"
            + "Provisioned Throughput (write capacity units/sec): %d \n",
            tableDescription.getTableName(), tableDescription.getTableStatus(),
            tableDescription.getProvisionedThroughput().getReadCapacityUnits(),
            tableDescription.getProvisionedThroughput().getWriteCapacityUnits());
    }

    static void updateExampleTable() {
        Table table = dynamoDB.getTable(tableName);
        System.out.println("Modifying provisioned throughput for " + tableName);

        try {
            table.updateTable(new
                ProvisionedThroughput().withReadCapacityUnits(6L).withWriteCapacityUnits(7L));

            table.waitForActive();
        }
        catch (Exception e) {
            System.err.println("UpdateTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }

    static void deleteExampleTable() {
        Table table = dynamoDB.getTable(tableName);
        try {
            System.out.println("Issuing DeleteTable request for " + tableName);
            table.delete();

            System.out.println("Waiting for " + tableName + " to be deleted...this may take
a while...");

            table.waitForDelete();
        }
        catch (Exception e) {
            System.err.println("DeleteTable request failed for " + tableName);
            System.err.println(e.getMessage());
        }
    }
}

```

Working with DynamoDB Tables in .NET

You can use the AWS SDK for .NET to create, update, and delete tables, list all the tables in your account, or get information about a specific table.

The following are the common steps for Amazon DynamoDB table operations using the AWS SDK for .NET.

1. Create an instance of the `AmazonDynamoDBClient` class (the client).

2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and `UpdateTableRequest` object to update an existing table.

3. Execute the appropriate method provided by the client that you created in the preceding step.

Note

The examples in this section don't work with .NET core because it doesn't support synchronous methods. For more information, see [AWS Asynchronous APIs for .NET](#).

Topics

- [Creating a Table \(p. 368\)](#)
- [Updating a Table \(p. 369\)](#)
- [Deleting a Table \(p. 370\)](#)
- [Listing Tables \(p. 370\)](#)
- [Example: Create, Update, Delete, and List Tables Using the AWS SDK for .NET Low-Level API \(p. 371\)](#)

Creating a Table

To create a table, you must provide the table name, its primary key, and the provisioned throughput values.

To create a table using the AWS SDK for .NET low-level API

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.
You must provide the table name, primary key, and the provisioned throughput values.
3. Execute the `AmazonDynamoDBClient.CreateTable` method by providing the request object as a parameter.

The following C# example demonstrates the preceding steps. The sample creates a table (`ProductCatalog`) that uses `Id` as the primary key and set of provisioned throughput values. Depending on your application requirements, you can update the provisioned throughput values by using the `UpdateTable` API.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new CreateTableRequest
{
    TableName = tableName,
    AttributeDefinitions = new List<AttributeDefinition>()
    {
        new AttributeDefinition
        {
            AttributeName = "Id",
            AttributeType = "N"
        }
    },
    KeySchema = new List<KeySchemaElement>()
    {
        new KeySchemaElement
        {
```

```
        AttributeName = "Id",
        KeyType = "HASH" //Partition key
    },
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = 10,
        WriteCapacityUnits = 5
    }
};

var response = client.CreateTable(request);
```

You must wait until DynamoDB creates the table and sets its status to ACTIVE. The `CreateTable` response includes the `TableDescription` property that provides the necessary table information.

Example

```
var result = response.CreateTableResult;
var tableDescription = result.TableDescription;
Console.WriteLine("{1}: {0} \t ReadCapacityUnits: {2} \t WriteCapacityUnits: {3}",
    tableDescription.TableStatus,
    tableDescription.TableName,
    tableDescription.ProvisionedThroughput.ReadCapacityUnits,
    tableDescription.ProvisionedThroughput.WriteCapacityUnits);

string status = tableDescription.TableStatus;
Console.WriteLine(tableName + " - " + status);
```

You can also call the `DescribeTable` method of the client to get table information at any time.

Example

```
var res = client.DescribeTable(new DescribeTableRequest{TableName = "ProductCatalog"});
```

Updating a Table

You can update only the provisioned throughput values of an existing table. Depending on your application requirements, you might need to update these values.

Note

You can increase throughput capacity as often as needed, and decrease it within certain constraints. For more information about throughput increases and decreases per day, see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

To update a table using the AWS SDK for .NET low-level API

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `UpdateTableRequest` class to provide the request information.

You must provide the table name and the new provisioned throughput values.
3. Execute the `AmazonDynamoDBClient.UpdateTable` method by providing the request object as a parameter.

The following C# example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
```

```
string tableName = "ExampleTable";

var request = new UpdateTableRequest()
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput()
    {
        // Provide new values.
        ReadCapacityUnits = 20,
        WriteCapacityUnits = 10
    }
};
var response = client.UpdateTable(request);
```

Deleting a Table

Follow these steps to delete a table using the .NET low-level API.

To delete a table using the AWS SDK for .NET low-level API

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `DeleteTableRequest` class, and provide the table name that you want to delete.
3. Execute the `AmazonDynamoDBClient.DeleteTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ExampleTable";

var request = new DeleteTableRequest{ TableName = tableName };
var response = client.DeleteTable(request);
```

Listing Tables

To list tables in your account using the AWS SDK for .NET low-level API, create an instance of the `AmazonDynamoDBClient` and execute the `ListTables` method.

The `ListTables` operation requires no parameters. However, you can specify optional parameters. For example, you can set the `Limit` parameter if you want to use paging to limit the number of table names per page. This requires you to create a `ListTablesRequest` object and provide optional parameters as shown in the following C# example. Along with the page size, the request sets the `ExclusiveStartTableName` parameter. Initially, `ExclusiveStartTableName` is null. However, after fetching the first page of results, to retrieve the next page of results, you must set this parameter value to the `LastEvaluatedTableName` property of the current result.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

// Initial value for the first page of table names.
string lastEvaluatedTableName = null;
do
{
    // Create a request object to specify optional parameters.
    var request = new ListTablesRequest
```

```

{
    Limit = 10, // Page size.
    ExclusiveStartTableName = lastEvaluatedTableName
};

var response = client.ListTables(request);
ListTablesResult result = response.ListTablesResult;
foreach (string name in result.TableNames)
    Console.WriteLine(name);

lastEvaluatedTableName = result.LastEvaluatedTableName;

} while (lastEvaluatedTableName != null);

```

Example: Create, Update, Delete, and List Tables Using the AWS SDK for .NET Low-Level API

The following C# example creates, updates, and deletes a table (`ExampleTable`). It also lists all the tables in your account and gets the description of a specific table. The table update increases the provisioned throughput values. For step-by-step instructions to test the following example, see [.NET Code Examples \(p. 332\)](#).

```


/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelTableExample
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "ExampleTable";

        static void Main(string[] args)
        {
            try
            {
                CreateExampleTable();
                ListMyTables();
                GetTableInformation();
                UpdateExampleTable();

                DeleteExampleTable();

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }
    }
}


```

```

        catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
        catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
        catch (Exception e) { Console.WriteLine(e.Message); }
    }

    private static void CreateExampleTable()
    {
        Console.WriteLine("\n*** Creating table ***");
        var request = new CreateTableRequest
        {
            AttributeDefinitions = new List<AttributeDefinition>()
            {
                new AttributeDefinition
                {
                    AttributeName = "Id",
                    AttributeType = "N"
                },
                new AttributeDefinition
                {
                    AttributeName = "ReplyDateTime",
                    AttributeType = "N"
                }
            },
            KeySchema = new List<KeySchemaElement>
            {
                new KeySchemaElement
                {
                    AttributeName = "Id",
                    KeyType = "HASH" //Partition key
                },
                new KeySchemaElement
                {
                    AttributeName = "ReplyDateTime",
                    KeyType = "RANGE" //Sort key
                }
            },
            ProvisionedThroughput = new ProvisionedThroughput
            {
                ReadCapacityUnits = 5,
                WriteCapacityUnits = 6
            },
            TableName = tableName
        };

        var response = client.CreateTable(request);

        var tableDescription = response.TableDescription;
        Console.WriteLine("{1}: {0} \t ReadsPerSec: {2} \t WritesPerSec: {3}",
            tableDescription.TableStatus,
            tableDescription.TableName,
            tableDescription.ProvisionedThroughput.ReadCapacityUnits,
            tableDescription.ProvisionedThroughput.WriteCapacityUnits);

        string status = tableDescription.TableStatus;
        Console.WriteLine(tableName + " - " + status);

        WaitUntilTableReady(tableName);
    }

    private static void ListMyTables()
    {
        Console.WriteLine("\n*** listing tables ***");
        string lastTableNameEvaluated = null;
        do
        {
            var request = new ListTablesRequest

```

```

        {
            Limit = 2,
            ExclusiveStartTableName = lastTableNameEvaluated
        };

        var response = client.ListTables(request);
        foreach (string name in response.TableNames)
            Console.WriteLine(name);

        lastTableNameEvaluated = response.LastEvaluatedTableName;
    } while (lastTableNameEvaluated != null);
}

private static void GetTableInformation()
{
    Console.WriteLine("\n*** Retrieving table information ***");
    var request = new DescribeTableRequest
    {
        TableName = tableName
    };

    var response = client.DescribeTable(request);

    TableDescription description = response.Table;
    Console.WriteLine("Name: {0}", description.TableName);
    Console.WriteLine("# of items: {0}", description.ItemCount);
    Console.WriteLine("Provision Throughput (reads/sec): {0}",
                      description.ProvisionedThroughput.ReadCapacityUnits);
    Console.WriteLine("Provision Throughput (writes/sec): {0}",
                      description.ProvisionedThroughput.WriteCapacityUnits);
}

private static void UpdateExampleTable()
{
    Console.WriteLine("\n*** Updating table ***");
    var request = new UpdateTableRequest()
    {
        TableName = tableName,
        ProvisionedThroughput = new ProvisionedThroughput()
        {
            ReadCapacityUnits = 6,
            WriteCapacityUnits = 7
        }
    };

    var response = client.UpdateTable(request);

    WaitUntilTableReady(tableName);
}

private static void DeleteExampleTable()
{
    Console.WriteLine("\n*** Deleting table ***");
    var request = new DeleteTableRequest
    {
        TableName = tableName
    };

    var response = client.DeleteTable(request);

    Console.WriteLine("Table is being deleted...");
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
}

```

```
// Let us wait until table is created. Call DescribeTable.
do
{
    System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
    try
    {
        var res = client.DescribeTable(new DescribeTableRequest
        {
            TableName = tableName
        });

        Console.WriteLine("Table name: {0}, status: {1}",
            res.Table.TableName,
            res.Table.TableStatus);
        status = res.Table.TableStatus;
    }
    catch (ResourceNotFoundException)
    {
        // DescribeTable is eventually consistent. So you might
        // get resource not found. So we handle the potential exception.
    }
} while (status != "ACTIVE");
}
```

Working with Items and Attributes

In Amazon DynamoDB, an *item* is a collection of attributes. Each attribute has a name and a value. An attribute value can be a scalar, a set, or a document type. For more information, see [Amazon DynamoDB: How It Works \(p. 2\)](#).

DynamoDB provides four operations for basic create, read, update, and delete (CRUD) functionality:

- `PutItem` — Create an item.
- `GetItem` — Read an item.
- `UpdateItem` — Update an item.
- `DeleteItem` — Delete an item.

Each of these operations requires that you specify the primary key of the item that you want to work with. For example, to read an item using `GetItem`, you must specify the partition key and sort key (if applicable) for that item.

In addition to the four basic CRUD operations, DynamoDB also provides the following:

- `BatchGetItem` — Read up to 100 items from one or more tables.
- `BatchWriteItem` — Create or delete up to 25 items in one or more tables.

These batch operations combine multiple CRUD operations into a single request. In addition, the batch operations read and write items in parallel to minimize response latencies.

This section describes how to use these operations and includes related topics, such as conditional updates and atomic counters. This section also includes example code that uses the AWS SDKs.

Topics

- [Reading an Item \(p. 375\)](#)
- [Writing an Item \(p. 375\)](#)

- [Return Values \(p. 377\)](#)
- [Batch Operations \(p. 378\)](#)
- [Atomic Counters \(p. 380\)](#)
- [Conditional Writes \(p. 380\)](#)
- [Using Expressions in DynamoDB \(p. 385\)](#)
- [Expiring Items By Using DynamoDB Time to Live \(TTL\) \(p. 408\)](#)
- [Working with Items: Java \(p. 415\)](#)
- [Working with Items: .NET \(p. 435\)](#)

Reading an Item

To read an item from a DynamoDB table, use the `GetItem` operation. You must provide the name of the table, along with the primary key of the item you want.

Example

The following AWS CLI example shows how to read an item from the `ProductCatalog` table.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}'
```

Note

With `GetItem`, you must specify the *entire* primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must supply a value for the partition key and a value for the sort key.

A `GetItem` request performs an eventually consistent read by default. You can use the `ConsistentRead` parameter to request a strongly consistent read instead. (This consumes additional read capacity units, but it returns the most up-to-date version of the item.)

`GetItem` returns all of the item's attributes. You can use a *projection expression* to return only some of the attributes. For more information, see [Projection Expressions \(p. 388\)](#).

To return the number of read capacity units consumed by `GetItem`, set the `ReturnConsumedCapacity` parameter to `TOTAL`.

Example

The following AWS Command Line Interface (AWS CLI) example shows some of the optional `GetItem` parameters.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--consistent-read \
--projection-expression "Description, Price, RelatedItems" \
--return-consumed-capacity TOTAL
```

Writing an Item

To create, update, or delete an item in a DynamoDB table, use one of the following operations:

- [PutItem](#)
- [UpdateItem](#)
- [DeleteItem](#)

For each of these operations, you must specify the entire primary key, not just part of it. For example, if a table has a composite primary key (partition key and sort key), you must provide a value for the partition key and a value for the sort key.

To return the number of write capacity units consumed by any of these operations, set the `ReturnConsumedCapacity` parameter to one of the following:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

PutItem

`PutItem` creates a new item. If an item with the same key already exists in the table, it is replaced with the new item.

Example

Write a new item to the `Thread` table. The primary key for `Thread` consists of `ForumName` (partition key) and `Subject` (sort key).

```
aws dynamodb put-item \
--table-name Thread \
--item file://item.json
```

The arguments for `--item` are stored in the `item.json` file.

```
{
    "ForumName": {"S": "Amazon DynamoDB"},
    "Subject": {"S": "New discussion thread"},
    "Message": {"S": "First post in this thread"},
    "LastPostedBy": {"S": "fred@example.com"},
    "LastPostDateTime": {"S": "201603190422"}
}
```

UpdateItem

If an item with the specified key does not exist, `UpdateItem` creates a new item. Otherwise, it modifies an existing item's attributes.

You use an *update expression* to specify the attributes that you want to modify and their new values. For more information, see [Update Expressions \(p. 400\)](#).

Within the update expression, you use expression attribute values as placeholders for the actual values. For more information, see [Expression Attribute Values \(p. 391\)](#).

Example

Modify various attributes in the `Thread` item. The optional `ReturnValues` parameter shows the item as it appears after the update. For more information, see [Return Values \(p. 377\)](#).

```
aws dynamodb update-item \  
    --table-name Thread \  
    --key file://key.json \  
    --update-expression "SET Answered = :zero, Replies = :zero, LastPostedBy = :lastpostedby" \  
    --expression-attribute-values file://expression-attribute-values.json \  
    --return-values ALL_NEW
```

The arguments for `--key` are stored in the `key.json` file.

```
{  
    "ForumName": {"S": "Amazon DynamoDB"},  
    "Subject": {"S": "New discussion thread"}  
}
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{  
    ":zero": {"N": "0"},  
    ":lastpostedby": {"S": "barney@example.com"}  
}
```

DeleteItem

`DeleteItem` deletes the item with the specified key.

Example

The following AWS CLI example shows how to delete the `Thread` item.

```
aws dynamodb delete-item \  
    --table-name Thread \  
    --key file://key.json
```

Return Values

In some cases, you might want DynamoDB to return certain attribute values as they appeared before or after you modified them. The `PutItem`, `UpdateItem`, and `DeleteItem` operations have a `ReturnValues` parameter that you can use to return the attribute values before or after they are modified.

The default value for `ReturnValues` is `NONE`, meaning that DynamoDB does not return any information about attributes that were modified.

The following are the other valid settings for `ReturnValues`, organized by DynamoDB API operation.

PutItem

- `ReturnValues: ALL_OLD`
 - If you overwrite an existing item, `ALL_OLD` returns the entire item as it appeared before the overwrite.
 - If you write a nonexistent item, `ALL_OLD` has no effect.

UpdateItem

The most common usage for `UpdateItem` is to update an existing item. However, `UpdateItem` actually performs an *upsert*, meaning that it automatically creates the item if it doesn't already exist.

- `ReturnValues: ALL_OLD`
 - If you update an existing item, `ALL_OLD` returns the entire item as it appeared before the update.
 - If you update a nonexistent item (upsert), `ALL_OLD` has no effect.
- `ReturnValues: ALL_NEW`
 - If you update an existing item, `ALL_NEW` returns the entire item as it appeared after the update.
 - If you update a nonexistent item (upsert), `ALL_NEW` returns the entire item.
- `ReturnValues: UPDATED_OLD`
 - If you update an existing item, `UPDATED_OLD` returns only the updated attributes, as they appeared before the update.
 - If you update a nonexistent item (upsert), `UPDATED_OLD` has no effect.
- `ReturnValues: UPDATED_NEW`
 - If you update an existing item, `UPDATED_NEW` returns only the affected attributes, as they appeared after the update.
 - If you update a nonexistent item (upsert), `UPDATED_NEW` returns only the updated attributes, as they appear after the update.

DeleteItem

- `ReturnValues: ALL_OLD`
 - If you delete an existing item, `ALL_OLD` returns the entire item as it appeared before you deleted it.
 - If you delete a nonexistent item, `ALL_OLD` doesn't return any data.

Batch Operations

For applications that need to read or write multiple items, DynamoDB provides the `BatchGetItem` and `BatchWriteItem` operations. Using these operations can reduce the number of network round trips from your application to DynamoDB. In addition, DynamoDB performs the individual read or write operations in parallel. Your applications benefit from this parallelism without having to manage concurrency or threading.

The batch operations are essentially wrappers around multiple read or write requests. For example, if a `BatchGetItem` request contains five items, DynamoDB performs five `GetItem` operations on your behalf. Similarly, if a `BatchWriteItem` request contains two put requests and four delete requests, DynamoDB performs two `PutItem` and four `DeleteItem` requests.

In general, a batch operation does not fail unless *all* the requests in the batch fail. For example, suppose that you perform a `BatchGetItem` operation, but one of the individual `GetItem` requests in the batch fails.

API Version 2012-08-10

fails. In this case, `BatchGetItem` returns the keys and data from the `GetItem` request that failed. The other `GetItem` requests in the batch are not affected.

BatchGetItem

A single `BatchGetItem` operation can contain up to 100 individual `GetItem` requests and can retrieve up to 16 MB of data. In addition, a `BatchGetItem` operation can retrieve items from multiple tables.

Example

Retrieve two items from the `Thread` table, using a projection expression to return only some of the attributes.

```
aws dynamodb batch-get-item \
--request-items file://request-items.json
```

The arguments for `--request-items` are stored in the `request-items.json` file.

```
{
    "Thread": {
        "Keys": [
            {
                "ForumName": {"S": "Amazon DynamoDB"},
                "Subject": {"S": "DynamoDB Thread 1"}
            },
            {
                "ForumName": {"S": "Amazon S3"},
                "Subject": {"S": "S3 Thread 1"}
            }
        ],
        "ProjectionExpression": "ForumName, Subject, LastPostedDateTime, Replies"
    }
}
```

BatchWriteItem

The `BatchWriteItem` operation can contain up to 25 individual `PutItem` and `DeleteItem` requests and can write up to 16 MB of data. (The maximum size of an individual item is 400 KB.) In addition, a `BatchWriteItem` operation can put or delete items in multiple tables.

Note

`BatchWriteItem` does not support `UpdateItem` requests.

Example

Write two items to the `ProductCatalog` table.

```
aws dynamodb batch-write-item \
--request-items file://request-items.json
```

The arguments for `--request-items` are stored in the `request-items.json` file.

```
{
    "ProductCatalog": [
```

```
{
    "PutRequest": {
        "Item": {
            "Id": { "N": "601" },
            "Description": { "S": "Snowboard" },
            "QuantityOnHand": { "N": "5" },
            "Price": { "N": "100" }
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": { "N": "602" },
            "Description": { "S": "Snow shovel" }
        }
    }
}
]
```

Atomic Counters

You can use the `UpdateItem` operation to implement an *atomic counter*—a numeric attribute that is incremented, unconditionally, without interfering with other write requests. (All write requests are applied in the order in which they were received.) With an atomic counter, the updates are not idempotent. In other words, the numeric value increments each time you call `UpdateItem`.

You might use an atomic counter to track the number of visitors to a website. In this case, your application would increment a numeric value, regardless of its current value. If an `UpdateItem` operation fails, the application could simply retry the operation. This would risk updating the counter twice, but you could probably tolerate a slight overcounting or undercounting of website visitors.

An atomic counter would not be appropriate where overcounting or undercounting can't be tolerated (for example, in a banking application). In this case, it is safer to use a conditional update instead of an atomic counter.

For more information, see [Incrementing and Decrementing Numeric Attributes \(p. 404\)](#).

Example

The following AWS CLI example increments the `Price` of a product by 5. (Because `UpdateItem` is not idempotent, the `Price` increases every time you run this example.)

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id": { "N": "601" }}' \
--update-expression "SET Price = Price + :incr" \
--expression-attribute-values '{":incr":{"N":"5"}}' \
--return-values UPDATED_NEW
```

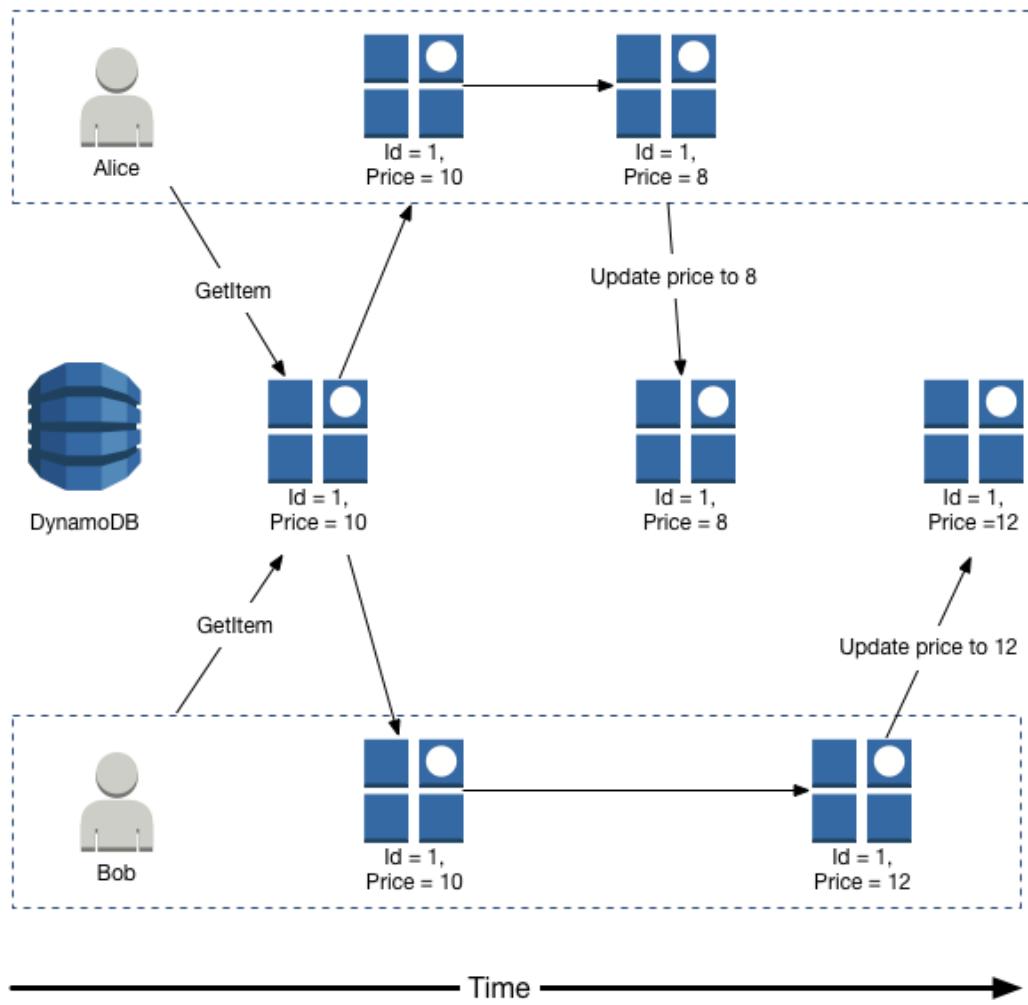
Conditional Writes

By default, the DynamoDB write operations (`PutItem`, `UpdateItem`, `DeleteItem`) are *unconditional*: Each operation overwrites an existing item that has the specified primary key.

DynamoDB optionally supports conditional writes for these operations. A conditional write succeeds only if the item attributes meet one or more expected conditions. Otherwise, it returns an error. Conditional

writes are helpful in many situations. For example, you might want a `PutItem` operation to succeed only if there is not already an item with the same primary key. Or you could prevent an `UpdateItem` operation from modifying an item if one of its attributes has a certain value.

Conditional writes are helpful in cases where multiple users attempt to modify the same item. Consider the following diagram, in which two users (Alice and Bob) are working with the same item from a DynamoDB table.



Suppose that Alice uses the AWS CLI to update the `Price` attribute to 8.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--update-expression "SET Price = :newval" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the file `expression-attribute-values.json`:

```
{  
    ":newval": {"N": "8"}  
}
```

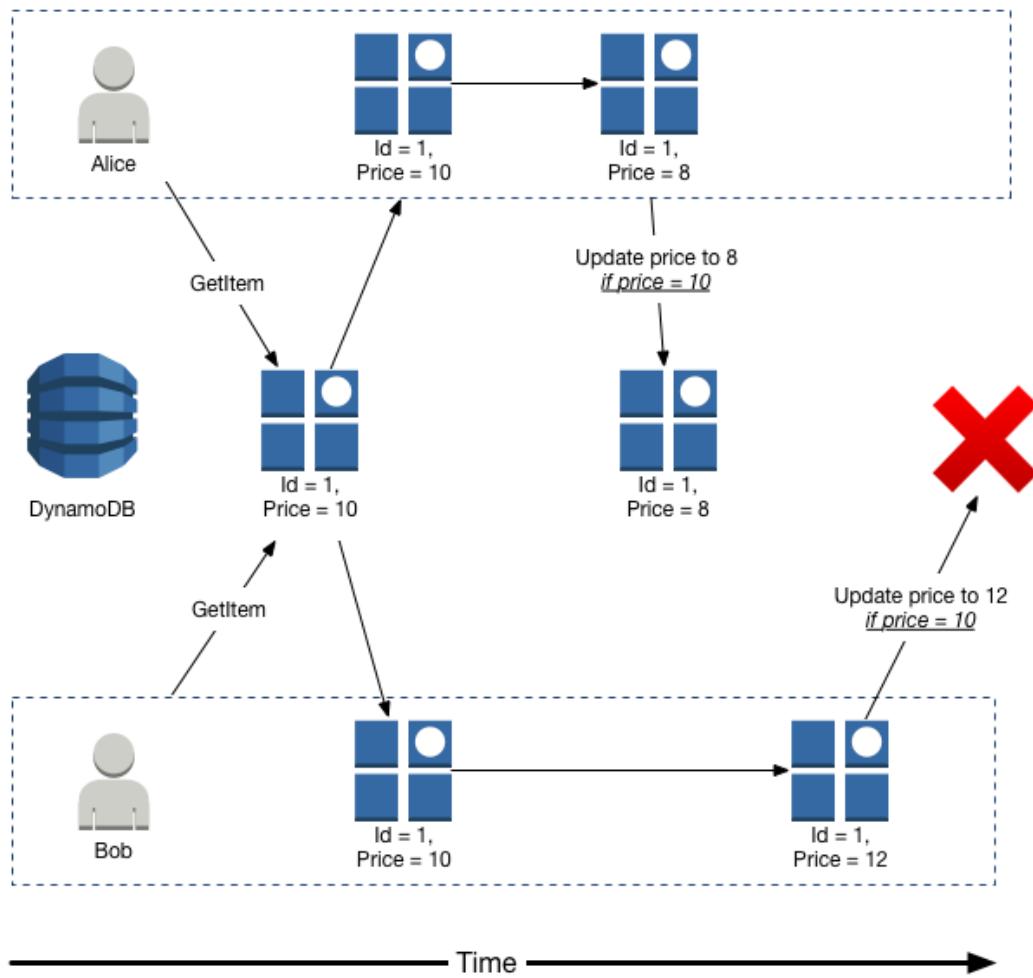
Now suppose that Bob issues a similar `UpdateItem` request later, but changes the `Price` to 12. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{  
    ":newval": {"N": "12"}  
}
```

Bob's request succeeds, but Alice's earlier update is lost.

To request a conditional `PutItem`, `DeleteItem`, or `UpdateItem`, you specify a condition expression. A *condition expression* is a string containing attribute names, conditional operators, and built-in functions. The entire expression must evaluate to true. Otherwise, the operation fails.

Now consider the following diagram, showing how conditional writes would prevent Alice's update from being overwritten.



Alice first tries to update `Price` to 8, but only if the current `Price` is 10.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--update-expression "SET Price = :newval" \
--condition-expression "Price = :currval" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for `--expression-attribute-values` are stored in the `expression-attribute-values.json` file.

```
{
  ":newval":{"N":"8"}, 
  ":currval":{"N":"10"}}
```

```
}
```

Alice's update succeeds because the condition evaluates to true.

Next, Bob attempts to update the `Price` to 12, but only if the current `Price` is 10. For Bob, the `--expression-attribute-values` parameter looks like the following.

```
{
  ":newval": {"N": "12"},
  ":currval": {"N": "10"}
}
```

Because Alice has previously changed the `Price` to 8, the condition expression evaluates to false, and Bob's update fails.

For more information, see [Condition Expressions \(p. 392\)](#).

Conditional Write Idempotence

Conditional writes can be *idempotent* if the conditional check is on the same attribute that is being updated. This means that DynamoDB performs a given write request only if certain attribute values in the item match what you expect them to be at the time of the request.

For example, suppose that you issue an `UpdateItem` request to increase the `Price` of an item by 3, but only if the `Price` is currently 20. After you send the request, but before you get the results back, a network error occurs, and you don't know whether the request was successful. Because this conditional write is idempotent, you can retry the same `UpdateItem` request, and DynamoDB updates the item only if the `Price` is currently 20.

Capacity Units Consumed by Conditional Writes

If a `ConditionExpression` evaluates to false during a conditional write, DynamoDB still consumes write capacity from the table:

- If the item does not currently exist in the table, DynamoDB consumes one write capacity unit.
- If the item does exist, the number of write capacity units consumed depends on the size of the item. For example, a failed conditional write of a 1 KB item would consume one write capacity unit. If the item were twice that size, the failed conditional write would consume two write capacity units.

Note

Write operations consume *write* capacity units only. They never consume *read* capacity units.

A failed conditional write returns a `ConditionalCheckFailedException`. When this occurs, you don't receive any information in the response about the write capacity that was consumed. However, you can view the `ConsumedWriteCapacityUnits` metric for the table in Amazon CloudWatch. For more information, see [DynamoDB Metrics \(p. 858\)](#) in [Logging and Monitoring in DynamoDB \(p. 855\)](#).

To return the number of write capacity units consumed during a conditional write, you use the `ReturnConsumedCapacity` parameter:

- `TOTAL` — Returns the total number of write capacity units consumed.
- `INDEXES` — Returns the total number of write capacity units consumed, with subtotals for the table and any secondary indexes that were affected by the operation.
- `NONE` — No write capacity details are returned. (This is the default.)

Note

Unlike a global secondary index, a local secondary index shares its provisioned throughput capacity with its table. Read and write activity on a local secondary index consumes provisioned throughput capacity from the table.

Using Expressions in DynamoDB

In Amazon DynamoDB, you use *expressions* to denote the attributes that you want to read from an item. You also use expressions when writing an item to indicate any conditions that must be met (also known as a conditional update), and to indicate how the attributes are to be updated. This section describes the basic expression grammar and the available kinds of expressions.

Note

For backward compatibility, DynamoDB also supports conditional parameters that do not use expressions. For more information, see [Legacy Conditional Parameters \(p. 1035\)](#).

New applications should use expressions rather than the legacy parameters.

Topics

- [Specifying Item Attributes When Using Expressions \(p. 385\)](#)
- [Projection Expressions \(p. 388\)](#)
- [Expression Attribute Names in DynamoDB \(p. 389\)](#)
- [Expression Attribute Values \(p. 391\)](#)
- [Condition Expressions \(p. 392\)](#)
- [Update Expressions \(p. 400\)](#)

Specifying Item Attributes When Using Expressions

This section describes how to refer to item attributes in an expression in Amazon DynamoDB. You can work with any attribute, even if it is deeply nested within multiple lists and maps.

Topics

- [Top-Level Attributes \(p. 386\)](#)
- [Nested Attributes \(p. 387\)](#)
- [Document Paths \(p. 387\)](#)

A Sample Item: ProductCatalog

The following is a representation of an item in the `ProductCatalog` table. (This table is described in [Example Tables and Data \(p. 973\)](#).)

```
{  
    "Id": 123,  
    "Title": "Bicycle 123",  
    "Description": "123 description",  
    "BicycleType": "Hybrid",  
    "Brand": "Brand-Company C",  
    "Price": 500,  
    "Color": ["Red", "Black"],  
    "ProductCategory": "Bicycle",  
    "InStock": true,  
    "QuantityOnHand": null,  
    "RelatedItems": [  
        341,  
        472,  
        649  
    ],  
}
```

```
"Pictures": {  
    "FrontView": "http://example.com/products/123_front.jpg",  
    "RearView": "http://example.com/products/123_rear.jpg",  
    "SideView": "http://example.com/products/123_left_side.jpg"  
},  
"ProductReviews": {  
    "FiveStar": [  
        "Excellent! Can't recommend it highly enough! Buy it!",  
        "Do yourself a favor and buy this."  
    ],  
    "OneStar": [  
        "Terrible product! Do not buy this."  
    ]  
},  
"Comment": "This product sells out quickly during the summer",  
"Safety.Warning": "Always wear a helmet"  
}
```

Note the following:

- The partition key value (`Id`) is 123. There is no sort key.
- Most of the attributes have scalar data types, such as `String`, `Number`, `Boolean`, and `Null`.
- One attribute (`Color`) is a `String Set`.
- The following attributes are document data types:
 - A list of `RelatedItems`. Each element is an `Id` for a related product.
 - A map of `Pictures`. Each element is a short description of a picture, along with a URL for the corresponding image file.
 - A map of `ProductReviews`. Each element represents a rating and a list of reviews corresponding to that rating. Initially, this map is populated with five-star and one-star reviews.

Top-Level Attributes

An attribute is said to be *top level* if it is not embedded within another attribute. For the `ProductCatalog` item, the top-level attributes are as follows:

- `Id`
- `Title`
- `Description`
- `BicycleType`
- `Brand`
- `Price`
- `Color`
- `ProductCategory`
- `InStock`
- `QuantityOnHand`
- `RelatedItems`
- `Pictures`
- `ProductReviews`
- `Comment`
- `Safety.Warning`

All of these top-level attributes are scalars, except for `Color` (list), `RelatedItems` (list), `Pictures` (map), and `ProductReviews` (map).

Nested Attributes

An attribute is said to be *nested* if it is embedded within another attribute. To access a nested attribute, you use *dereference operators*:

- `[n]` — for list elements
- `.` (dot) — for map elements

Accessing List Elements

The dereference operator for a list element is `[n]`, where *n* is the element number. List elements are zero-based, so `[0]` represents the first element in the list, `[1]` represents the second, and so on. Here are some examples:

- `MyList[0]`
- `AnotherList[12]`
- `ThisList[5][11]`

The element `ThisList[5]` is itself a nested list. Therefore, `ThisList[5][11]` refers to the 12th element in that list.

The number within the square brackets must be a non-negative integer. Therefore, the following expressions are not valid:

- `MyList[-1]`
- `MyList[0.4]`

Accessing Map Elements

The dereference operator for a map element is `.` (a dot). Use a dot as a separator between elements in a map:

- `MyMap.nestedField`
- `MyMap.nestedField.deeplyNestedField`

Document Paths

In an expression, you use a *document path* to tell DynamoDB where to find an attribute. For a top-level attribute, the document path is simply the attribute name. For a nested attribute, you construct the document path using dereference operators.

The following are some examples of document paths. (Refer to the item shown in [Specifying Item Attributes When Using Expressions \(p. 385\)](#).)

- A top-level scalar attribute.
`ProductDescription`
- A top-level list attribute. (This returns the entire list, not just some of the elements.)
`RelatedItems`
- The third element from the `RelatedItems` list. (Remember that list elements are zero-based.)
`RelatedItems[2]`
- The front-view picture of the product.

```
Pictures.FrontView
• All of the five-star reviews.

ProductReviews.FiveStar
• The first of the five-star reviews.

ProductReviews.FiveStar[0]
```

Note

The maximum depth for a document path is 32. Therefore, the number of dereferences operators in a path cannot exceed this limit.

You can use any attribute name in a document path, provided that the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. If an attribute name does not meet this requirement, you must define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

Projection Expressions

To read data from a table, you use operations such as `GetItem`, `Query`, or `Scan`. Amazon DynamoDB returns all the item attributes by default. To get only some, rather than all of the attributes, use a projection expression.

A *projection expression* is a string that identifies the attributes that you want. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated.

The following are some examples of projection expressions, based on the `ProductCatalog` item from [Specifying Item Attributes When Using Expressions \(p. 385\)](#):

- A single top-level attribute.

```
Title
```

- Three top-level attributes. DynamoDB retrieves the entire `Color` set.

```
Title, Price, Color
```

- Four top-level attributes. DynamoDB returns the entire contents of `RelatedItems` and `ProductReviews`.

```
Title, Description, RelatedItems, ProductReviews
```

You can use any attribute name in a projection expression, provided that the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. If an attribute name does not meet this requirement, you must define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

The following AWS CLI example shows how to use a projection expression with a `GetItem` operation. This projection expression retrieves a top-level scalar attribute (`Description`), the first element in a list (`RelatedItems[0]`), and a list nested within a map (`ProductReviews.FiveStar`).

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key file://key.json \
--projection-expression "Description, RelatedItems[0], ProductReviews.FiveStar"
```

The arguments for `--key` are stored in the `key.json` file.

```
{  
    "Id": { "N": "123" }  
}
```

For programming language-specific code examples, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Expression Attribute Names in DynamoDB

An *expression attribute name* is a placeholder that you use in an Amazon DynamoDB expression as an alternative to an actual attribute name. An expression attribute name must begin with a pound sign (#), and be followed by one or more alphanumeric characters.

This section describes several situations in which you must use expression attribute names.

Note

The examples in this section use the AWS Command Line Interface (AWS CLI). For programming language-specific code examples, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Topics

- [Reserved Words \(p. 389\)](#)
- [Attribute Names Containing Dots \(p. 390\)](#)
- [Nested Attributes \(p. 390\)](#)
- [Repeating Attribute Names \(p. 391\)](#)

Reserved Words

Sometimes you might need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word. (For a complete list of reserved words, see [Reserved Words in DynamoDB \(p. 1026\)](#).)

For example, the following AWS CLI example would fail because `COMMENT` is a reserved word.

```
aws dynamodb get-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"123"}}' \  
    --projection-expression "Comment"
```

To work around this, you can replace `Comment` with an expression attribute name such as `#c`. The # (pound sign) is required and indicates that this is a placeholder for an attribute name. The AWS CLI example would now look like the following.

```
aws dynamodb get-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"123"}}' \  
    --projection-expression "#c" \  
    --expression-attribute-names '{"#c":"Comment"}'
```

Note

If an attribute name begins with a number or contains a space, a special character, or a reserved word, you *must* use an expression attribute name to replace that attribute's name in the expression.

Attribute Names Containing Dots

In an expression, a dot (".") is interpreted as a separator character in a document path. However, DynamoDB also allows you to use a dot character as part of an attribute name. This can be ambiguous in some cases. To illustrate, suppose that you wanted to retrieve the `Safety.Warning` attribute from a `ProductCatalog` item (see [Specifying Item Attributes When Using Expressions \(p. 385\)](#)).

Suppose that you wanted to access `Safety.Warning` using a projection expression.

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"123"}}' \  
  --projection-expression "Safety.Warning"
```

DynamoDB would return an empty result, rather than the expected string ("Always wear a helmet"). This is because DynamoDB interprets a dot in an expression as a document path separator. In this case, you must define an expression attribute name (such as `#sw`) as a substitute for `Safety.Warning`. You could then use the following projection expression.

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"123"}}' \  
  --projection-expression "#sw" \  
  --expression-attribute-names '{"#sw":"Safety.Warning"}'
```

DynamoDB would then return the correct result.

Nested Attributes

Suppose that you wanted to access the nested attribute `ProductReviews.OneStar`, using the following projection expression.

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"123"}}' \  
  --projection-expression "ProductReviews.OneStar"
```

The result would contain all of the one-star product reviews, which is expected.

But what if you decided to use an expression attribute name instead? For example, what would happen if you were to define `#pr1star` as a substitute for `ProductReviews.OneStar`?

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"123"}}' \  
  --projection-expression "#pr1star" \  
  --expression-attribute-names '{"#pr1star":"ProductReviews.OneStar"}'
```

DynamoDB would return an empty result instead of the expected map of one-star reviews. This is because DynamoDB interprets a dot in an expression attribute value as a character within an attribute's

name. When DynamoDB evaluates the expression attribute name `#pr1star`, it determines that `ProductReviews.OneStar` refers to a scalar attribute—which is not what was intended.

The correct approach would be to define an expression attribute name for each element in the document path:

- `#pr - ProductReviews`
- `#1star - OneStar`

You could then use `#pr.#1star` for the projection expression.

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"123"}}' \
  --projection-expression "#pr.#1star" \
  --expression-attribute-names '{"#pr":"ProductReviews", "#1star":"OneStar"}'
```

DynamoDB would then return the correct result.

Repeating Attribute Names

Expression attribute names are helpful when you need to refer to the same attribute name repeatedly. For example, consider the following expression for retrieving some of the reviews from a `ProductCatalog` item.

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"123"}}' \
  --projection-expression "ProductReviews.FiveStar, ProductReviews.ThreeStar,
  ProductReviews.OneStar"
```

To make this more concise, you can replace `ProductReviews` with an expression attribute name such as `#pr`. The revised expression would now look like the following.

- `#pr.FiveStar, #pr.ThreeStar, #pr.OneStar`

```
aws dynamodb get-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"123"}}' \
  --projection-expression "#pr.FiveStar, #pr.ThreeStar, #pr.OneStar" \
  --expression-attribute-names '{"#pr":"ProductReviews"}'
```

If you define an expression attribute name, you must use it consistently throughout the entire expression. Also, you cannot omit the `#` symbol.

Expression Attribute Values

If you need to compare an attribute with a value, define an expression attribute value as a placeholder. *Expression attribute values* in Amazon DynamoDB are substitutes for the actual values that you want to

compare—values that you might not know until runtime. An expression attribute value must begin with a colon (:) and be followed by one or more alphanumeric characters.

For example, suppose that you wanted to return all of the `ProductCatalog` items that are available in `Black` and cost 500 or less. You could use a `Scan` operation with a filter expression, as in this AWS Command Line Interface (AWS CLI) example.

```
aws dynamodb scan \
    --table-name ProductCatalog \
    --filter-expression "contains(Color, :c) and Price <= :p" \
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
  ":c": { "S": "Black" },
  ":p": { "N": "500" }
}
```

Note

A `Scan` operation reads every item in a table. So you should avoid using `Scan` with large tables. The filter expression is applied to the `Scan` results, and items that don't match the filter expression are discarded.

If you define an expression attribute value, you must use it consistently throughout the entire expression. Also, you can't omit the `:` symbol.

Expression attribute values are used with condition expressions, update expressions, and filter expressions.

Note

For programming language-specific code examples, see [Getting Started with DynamoDB and AWS SDKs \(p. 76\)](#).

Condition Expressions

To manipulate data in an Amazon DynamoDB table, you use the `PutItem`, `UpdateItem`, and `DeleteItem` operations. (You can also use `BatchWriteItem` to perform multiple `PutItem` or `DeleteItem` operations in a single call.)

For these data manipulation operations, you can specify a *condition expression* to determine which items should be modified. If the condition expression evaluates to true, the operation succeeds; otherwise, the operation fails.

The following are some AWS Command Line Interface (AWS CLI) examples of using condition expressions. These examples are based on the `ProductCatalog` table, which was introduced in [Specifying Item Attributes When Using Expressions \(p. 385\)](#). The partition key for this table is `Id`; there is no sort key. The following `PutItem` operation creates a sample `ProductCatalog` item that the examples refer to.

```
aws dynamodb put-item \
    --table-name ProductCatalog \
    --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file. (For simplicity, only a few item attributes are used.)

```
{  
    "Id": {"N": "456"},  
    "ProductCategory": {"S": "Sporting Goods"},  
    "Price": {"N": "650"}  
}
```

Topics

- [Preventing Overwrites of an Existing Item \(p. 393\)](#)
- [Checking for Attributes in an Item \(p. 393\)](#)
- [Conditional Deletes \(p. 394\)](#)
- [Conditional Updates \(p. 395\)](#)
- [Comparison Operator and Function Reference \(p. 395\)](#)

Preventing Overwrites of an Existing Item

The `PutItem` operation overwrites an item with the same key (if it exists). If you want to avoid this, use a condition expression. This allows the write to proceed only if the item in question does not already have the same key.

```
aws dynamodb put-item \  
    --table-name ProductCatalog \  
    --item file://item.json \  
    --condition-expression "attribute_not_exists(Id)"
```

If the condition expression evaluates to false, DynamoDB returns the following error message: The conditional request failed.

Note

For more information about `attribute_not_exists` and other functions, see [Comparison Operator and Function Reference \(p. 395\)](#).

Checking for Attributes in an Item

You can check for the existence (or nonexistence) of any attribute. If the condition expression evaluates to true, the operation succeeds; otherwise, it fails.

The following example uses `attribute_not_exists` to delete a product only if it does not have a `Price` attribute.

```
aws dynamodb delete-item \  
    --table-name ProductCatalog \  
    --key '{"Id": {"N": "456"}}' \  
    --condition-expression "attribute_not_exists(Price)"
```

DynamoDB also provides an `attribute_exists` function. The following example deletes a product only if it has received poor reviews.

```
aws dynamodb delete-item \  
    --table-name ProductCatalog \  
    --key '{"Id": {"N": "456"}}'
```

```
--condition-expression "attribute_exists(ProductReviews.OneStar)"
```

Note

For more information about `attribute_not_exists`, `attribute_exists`, and other functions, see [Comparison Operator and Function Reference \(p. 395\)](#).

Conditional Deletes

To perform a conditional delete, you use a `DeleteItem` operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Consider the item from [Condition Expressions \(p. 392\)](#).

```
{  
    "Id": {  
        "N": "456"  
    },  
    "Price": {  
        "N": "650"  
    },  
    "ProductCategory": {  
        "S": "Sporting Goods"  
    }  
}
```

Suppose that you wanted to delete the item, but only under the following conditions:

- The `ProductCategory` is either "Sporting Goods" or "Gardening Supplies."
- The `Price` is between 500 and 600.

The following example tries to delete the item.

```
aws dynamodb delete-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"456"}}' \  
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (Price between :lo  
    and :hi)" \  
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
    ":cat1": {"S": "Sporting Goods"},  
    ":cat2": {"S": "Gardening Supplies"},  
    ":lo": {"N": "500"},  
    ":hi": {"N": "600"}  
}
```

Note

In the condition expression, the `:` (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see [Expression Attribute Values \(p. 391\)](#). For more information about `IN`, `AND`, and other keywords, see [Comparison Operator and Function Reference \(p. 395\)](#).

In this example, the `ProductCategory` comparison evaluates to true, but the `Price` comparison evaluates to false. This causes the condition expression to evaluate to false and the `DeleteItem` operation to fail.

Conditional Updates

To perform a conditional update, you use an `UpdateItem` operation with a condition expression. The condition expression must evaluate to true in order for the operation to succeed; otherwise, the operation fails.

Note

`UpdateItem` also supports *update expressions*, where you specify the modifications you want to make to an item. For more information, see [Update Expressions \(p. 400\)](#).

Suppose that you started with the item shown in [Condition Expressions \(p. 392\)](#).

```
{  
    "Id": { "N": "456"},  
    "Price": {"N": "650"},  
    "ProductCategory": {"S": "Sporting Goods"}  
}
```

The following example performs an `UpdateItem` operation. It tries to reduce the `Price` of a product by 75—but the condition expression prevents the update if the current `Price` is below 500.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id": {"N": "456"}}' \  
    --update-expression "SET Price = Price - :discount" \  
    --condition-expression "Price > :limit" \  
    --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
    ":discount": { "N": "75"},  
    ":limit": {"N": "500"}  
}
```

If the starting `Price` is 650, the `UpdateItem` operation reduces the `Price` to 575. If you run the `UpdateItem` operation again, the `Price` is reduced to 500. If you run it a third time, the condition expression evaluates to false, and the update fails.

Note

In the condition expression, the `:` (colon character) indicates an *expression attribute value*—a placeholder for an actual value. For more information, see [Expression Attribute Values \(p. 391\)](#). For more information about `>` and other operators, see [Comparison Operator and Function Reference \(p. 395\)](#).

Comparison Operator and Function Reference

This section covers the built-in functions and keywords for writing condition expressions in Amazon DynamoDB.

Topics

- [Syntax for Condition Expressions \(p. 396\)](#)
- [Making Comparisons \(p. 396\)](#)

- [Functions \(p. 397\)](#)
- [Logical Evaluations \(p. 399\)](#)
- [Parentheses \(p. 400\)](#)
- [Precedence in Conditions \(p. 400\)](#)

Syntax for Condition Expressions

In the following syntax summary, an *operand* can be the following:

- A top-level attribute name, such as `Id`, `Title`, `Description`, or `ProductCategory`
- A document path that references a nested attribute

```

condition-expression ::= 
    operand comparator operand
    | operand BETWEEN operand AND operand
    | operand IN ( operand (',', operand (, , ...)) )
    | function
    | condition AND condition
    | condition OR condition
    | NOT condition
    | ( condition )

comparator ::= 
    =
    | <>
    | <
    | <=
    | >
    | >=

function ::= 
    attribute_exists (path)
    | attribute_not_exists (path)
    | attribute_type (path, type)
    | begins_with (path, substr)
    | contains (path, operand)
    | size (path)

```

Making Comparisons

Use these comparators to compare an operand against a range of values or an enumerated list of values:

- *a* = *b* — true if *a* is equal to *b*
- *a* <*>* *b* — true if *a* is not equal to *b*
- *a* < *b* — true if *a* is less than *b*
- *a* <= *b* — true if *a* is less than or equal to *b*
- *a* > *b* — true if *a* is greater than *b*
- *a* >= *b* — true if *a* is greater than or equal to *b*

Use the `BETWEEN` and `IN` keywords to compare an operand against a range of values or an enumerated list of values:

- *a* `BETWEEN` *b* `AND` *c* — true if *a* is greater than or equal to *b*, and less than or equal to *c*.
- *a* `IN` (*b*, *c*, *d*) — true if *a* is equal to any value in the list — for example, any of *b*, *c* or *d*. The list can contain up to 100 values, separated by commas.

Functions

Use the following functions to determine whether an attribute exists in an item, or to evaluate the value of an attribute. These function names are case sensitive. For a nested attribute, you must provide its full document path.

| Function | Description |
|--|---|
| <code>attribute_exists (path)</code> | <p>True if the item contains the attribute specified by path.</p> <p>Example: Check whether an item in the <code>Product</code> table has a side view picture.</p> <ul style="list-style-type: none"> • <code>attribute_exists (Pictures.SideView)</code> |
| <code>attribute_not_exists (path)</code> | <p>True if the attribute specified by path does not exist in the item.</p> <p>Example: Check whether an item has a <code>Manufacturer</code> attribute.</p> <ul style="list-style-type: none"> • <code>attribute_not_exists (Manufacturer)</code> |
| <code>attribute_type (path, type)</code> | <p>True if the attribute at the specified path is of a particular data type. The type parameter must be one of the following:</p> <ul style="list-style-type: none"> • <code>S</code> — String • <code>SS</code> — String Set • <code>N</code> — Number • <code>NS</code> — Number Set • <code>B</code> — Binary • <code>BS</code> — Binary Set • <code>BOOL</code> — Boolean • <code>NULL</code> — Null • <code>L</code> — List • <code>M</code> — Map <p>You must use an expression attribute value for the type parameter.</p> <p>Example: Check whether the <code>QuantityOnHand</code> attribute is of type List. In this example, <code>:v_sub</code> is a placeholder for the string <code>L</code>.</p> <ul style="list-style-type: none"> • <code>attribute_type (ProductReviews.FiveStar, :v_sub)</code> <p>You must use an expression attribute value for the type parameter.</p> |
| <code>begins_with (path, substr)</code> | True if the attribute specified by path begins with a particular substring. |

| Function | Description |
|---------------------------------------|--|
| | <p>Example: Check whether the first few characters of the front view picture URL are <code>http://</code>.</p> <ul style="list-style-type: none"> • <code>begins_with (Pictures.FrontView, :v_sub)</code> <p>The expression attribute value <code>:v_sub</code> is a placeholder for <code>http://</code>.</p> |
| <code>contains (path, operand)</code> | <p>True if the attribute specified by path is one of the following:</p> <ul style="list-style-type: none"> • A String that contains a particular substring. • A Set that contains a particular element within the set. <p>In either case, operand must be a String.</p> <p>The path and the operand must be distinct; that is, <code>contains (a, a)</code> returns an error.</p> <p>Example: Check whether the Brand attribute contains the substring Company.</p> <ul style="list-style-type: none"> • <code>contains (Brand, :v_sub)</code> <p>The expression attribute value <code>:v_sub</code> is a placeholder for Company.</p> <p>Example: Check whether the product is available in red.</p> <ul style="list-style-type: none"> • <code>contains (Color, :v_sub)</code> <p>The expression attribute value <code>:v_sub</code> is a placeholder for Red.</p> |

| Function | Description |
|--------------------------|---|
| <code>size (path)</code> | <p>Returns a number representing an attribute's size. The following are valid data types for use with <code>size</code>.</p> <p>If the attribute is of type <code>String</code>, <code>size</code> returns the length of the string.</p> <p>Example: Check whether the string <code>Brand</code> is less than or equal to 20 characters. The expression attribute value <code>:v_sub</code> is a placeholder for 20.</p> <ul style="list-style-type: none"> • <code>size (Brand) <= :v_sub</code> <p>If the attribute is of type <code>Binary</code>, <code>size</code> returns the number of bytes in the attribute value.</p> <p>Example: Suppose that the <code>ProductCatalog</code> item has a binary attribute named <code>VideoClip</code>, which contains a short video of the product in use. The following expression checks whether <code>VideoClip</code> exceeds 64,000 bytes. The expression attribute value <code>:v_sub</code> is a placeholder for 64000.</p> <ul style="list-style-type: none"> • <code>size(VideoClip) > :v_sub</code> <p>If the attribute is a <code>Set</code> data type, <code>size</code> returns the number of elements in the set.</p> <p>Example: Check whether the product is available in more than one color. The expression attribute value <code>:v_sub</code> is a placeholder for 1.</p> <ul style="list-style-type: none"> • <code>size (Color) < :v_sub</code> <p>If the attribute is of type <code>List</code> or <code>Map</code>, <code>size</code> returns the number of child elements.</p> <p>Example: Check whether the number of <code>OneStar</code> reviews has exceeded a certain threshold. The expression attribute value <code>:v_sub</code> is a placeholder for 3.</p> <ul style="list-style-type: none"> • <code>size(ProductReviews.OneStar) > :v_sub</code> |

Logical Evaluations

Use the `AND`, `OR`, and `NOT` keywords to perform logical evaluations. In the list following, `a` and `b` represent conditions to be evaluated.

- `a AND b` — true if `a` and `b` are both true.
- `a OR b` — true if either `a` or `b` (or both) are true.

- NOT *a* — true if *a* is false; false if *a* is true.

Parentheses

Use parentheses to change the precedence of a logical evaluation. For example, suppose that conditions *a* and *b* are true, and that condition *c* is false. The following expression evaluates to true:

- *a* OR *b* AND *c*

However, if you enclose a condition in parentheses, it is evaluated first. For example, the following evaluates to false:

- (*a* OR *b*) AND *c*

Note

You can nest parentheses in an expression. The innermost ones are evaluated first.

Precedence in Conditions

DynamoDB evaluates conditions from left to right using the following precedence rules:

- = <> < <= > >=
- IN
- BETWEEN
- attribute_exists attribute_not_exists begins_with contains
- Parentheses
- NOT
- AND
- OR

Update Expressions

To update an existing item in an Amazon DynamoDB table, you use the `UpdateItem` operation. You must provide the key of the item that you want to update. You must also provide an update expression, indicating the attributes that you want to modify and the values that you want to assign to them.

An *update expression* specifies how `UpdateItem` will modify the attributes of an item—for example, setting a scalar value or removing elements from a list or a map.

The following is a syntax summary for update expressions.

```
update-expression ::=  
[ SET action [, action] ... ]  
[ REMOVE action [, action] ... ]  
[ ADD action [, action] ... ]  
[ DELETE action [, action] ... ]
```

An update expression consists of one or more clauses. Each clause begins with a `SET`, `REMOVE`, `ADD`, or `DELETE` keyword. You can include any of these clauses in an update expression, in any order. However, each action keyword can appear only once.

Within each clause, there are one or more actions separated by commas. Each action represents a data modification.

The examples in this section are based on the `ProductCatalog` item shown in [Projection Expressions \(p. 388\)](#).

Topics

- [SET—Modifying or Adding Item Attributes \(p. 401\)](#)
- [REMOVE—Deleting Attributes from an Item \(p. 406\)](#)
- [ADD—Updating Numbers and Sets \(p. 407\)](#)
- [DELETE—Removing Elements from a Set \(p. 408\)](#)

SET—Modifying or Adding Item Attributes

Use the `SET` action in an update expression to add one or more attributes to an item. If any of these attributes already exists, they are overwritten by the new values.

You can also use `SET` to add or subtract from an attribute that is of type `Number`. To perform multiple `SET` actions, separate them with commas.

In the following syntax summary:

- The `path` element is the document path to the item.
- An `operand` element can be either a document path to an item or a function.

```
set-action ::=  
    path = value  
  
value ::=  
    operand  
    | operand '+' operand  
    | operand '-' operand  
  
operand ::=  
    path | function
```

The following `PutItem` operation creates a sample item that the examples refer to.

```
aws dynamodb put-item \  
  --table-name ProductCatalog \  
  --item file://item.json
```

The arguments for `--item` are stored in the `item.json` file. (For simplicity, only a few item attributes are used.)

```
{  
    "Id": {"N": "789"},  
    "ProductCategory": {"S": "Home Improvement"},  
    "Price": {"N": "52"},  
    "InStock": {"BOOL": true},  
    "Brand": {"S": "Acme"}  
}
```

Topics

- [Modifying Attributes \(p. 402\)](#)

- [Adding Lists and Maps \(p. 402\)](#)
- [Adding Elements to a List \(p. 403\)](#)
- [Adding Nested Map Attributes \(p. 403\)](#)
- [Incrementing and Decrementing Numeric Attributes \(p. 404\)](#)
- [Appending Elements to a List \(p. 404\)](#)
- [Preventing Overwrites of an Existing Attribute \(p. 405\)](#)

Modifying Attributes

Example

Update the `ProductCategory` and `Price` attributes.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "SET ProductCategory = :c, Price = :p" \  
    --expression-attribute-values file://values.json \  
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
    ":c": { "S": "Hardware" },  
    ":p": { "N": "60" }  
}
```

Note

In the `UpdateItem` operation, `--return-values ALL_NEW` causes DynamoDB to return the item as it appears after the update.

Adding Lists and Maps

Example

Add a new list and a new map.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "SET RelatedItems = :ri, ProductReviews = :pr" \  
    --expression-attribute-values file://values.json \  
    --return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{  
    ":ri": {  
        "L": [  
            { "S": "Hammer" }  
        ]  
    }  
}
```

```
},
":pr": {
    "M": {
        "FiveStar": {
            "L": [
                { "S": "Best product ever!" }
            ]
        }
    }
}
```

Adding Elements to a List

Example

Add a new attribute to the `RelatedItems` list. (Remember that list elements are zero-based, so [0] represents the first element in the list, [1] represents the second, and so on.)

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "SET RelatedItems[1] = :ri" \
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
    ":ri": { "S": "Nails" }
}
```

Note

When you use `SET` to update a list element, the contents of that element are replaced with the new data that you specify. If the element doesn't already exist, `SET` appends the new element at the end of the list.

If you add multiple elements in a single `SET` operation, the elements are sorted in order by element number.

Adding Nested Map Attributes

Example

Add some nested map attributes.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "SET #pr.#5star[1] = :r5, #pr.#3star = :r3" \
--expression-attribute-names file://names.json \
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for `--expression-attribute-names` are stored in the `names.json` file.

```
{  
    "#pr": "ProductReviews",  
    "#5star": "FiveStar",  
    "#3star": "ThreeStar"  
}
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{  
    ":r5": { "S": "Very happy with my purchase" },  
    ":r3": {  
        "L": [  
            { "S": "Just OK - not that great" }  
        ]  
    }  
}
```

Incrementing and Decrementing Numeric Attributes

You can add to or subtract from an existing numeric attribute. To do this, use the + (plus) and - (minus) operators.

Example

Decrease the Price of an item.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "SET Price = Price - :p" \  
    --expression-attribute-values '{":p": {"N":"15"}}' \  
    --return-values ALL_NEW
```

To increase the Price, you would use the + operator in the update expression.

Appending Elements to a List

You can add elements to the end of a list. To do this, use SET with the list_append function. (The function name is case sensitive.) The list_append function is specific to the SET action and can only be used in an update expression. The syntax is as follows.

- `list_append (list1, list2)`

The function takes two lists as input and appends all elements from *list2* to *list1*.

Example

In [Adding Elements to a List \(p. 403\)](#), you create the RelatedItems list and populate it with two elements: Hammer and Nails. Now you append two more elements to the end of RelatedItems.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "SET #ri = list_append(#ri, :vals)" \  
    --expression-attribute-names {"#ri": "RelatedItems"} \  
    --expression-attribute-values '{":vals": ["Screws", "Bolts"]}'
```

```
--expression-attribute-names '{"#ri": "RelatedItems"}' \
--expression-attribute-values file://values.json \
--return-values ALL_NEW
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{
  ":vals": {
    "L": [
      { "S": "Screwdriver" },
      { "S": "Hacksaw" }
    ]
  }
}
```

Finally, you append one more element to the *beginning* of RelatedItems. To do this, swap the order of the list_append elements. (Remember that list_append takes two lists as input and appends the second list to the first.)

```
aws dynamodb update-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"789"}}' \
  --update-expression "SET #ri = list_append(:vals, #ri)" \
  --expression-attribute-names '{"#ri": "RelatedItems"}' \
  --expression-attribute-values '{":vals": {"L": [ { "S": "Chisel" } ]}}' \
  --return-values ALL_NEW
```

The resulting RelatedItems attribute now contains five elements, in the following order: Chisel, Hammer, Nails, Screwdriver, Hacksaw.

Preventing Overwrites of an Existing Attribute

If you want to avoid overwriting an existing attribute, you can use SET with the if_not_exists function. (The function name is case sensitive.) The if_not_exists function is specific to the SET action and can only be used in an update expression. The syntax is as follows.

- `if_not_exists (path, value)`

If the item does not contain an attribute at the specified `path`, `if_not_exists` evaluates to `value`; otherwise, it evaluates to `path`.

Example

Set the Price of an item, but only if the item does not already have a Price attribute. (If Price already exists, nothing happens.)

```
aws dynamodb update-item \
  --table-name ProductCatalog \
  --key '{"Id":{"N":"789"}}' \
  --update-expression "SET Price = if_not_exists(Price, :p)" \
  --expression-attribute-values '{":p": {"N": "100"}}' \
  --return-values ALL_NEW
```

REMOVE—Deleting Attributes from an Item

Use the `REMOVE` action in an update expression to remove one or more attributes from an item in Amazon DynamoDB. To perform multiple `REMOVE` actions, separate them with commas.

The following is a syntax summary for `REMOVE` in an update expression. The only operand is the document path for the attribute that you want to remove.

```
remove-action ::=  
    path
```

Example

Remove some attributes from an item. (If the attributes don't exist, nothing happens.)

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "REMOVE Brand, InStock, QuantityOnHand" \  
    --return-values ALL_NEW
```

Removing Elements from a List

You can use `REMOVE` to delete individual elements from a list.

Example

In [Appending Elements to a List \(p. 404\)](#), you modify a list attribute (`RelatedItems`) so that it contained five elements:

- [0]—Chisel
- [1]—Hammer
- [2]—Nails
- [3]—Screwdriver
- [4]—Hacksaw

The following AWS Command Line Interface (AWS CLI) example deletes Hammer and Nails from the list.

```
aws dynamodb update-item \  
    --table-name ProductCatalog \  
    --key '{"Id":{"N":"789"}}' \  
    --update-expression "REMOVE RelatedItems[1], RelatedItems[2]" \  
    --return-values ALL_NEW
```

After Hammer and Nails are removed, the remaining elements are shifted. The list now contains the following:

- [0]—Chisel
- [1]—Screwdriver
- [2]—Hacksaw

ADD—Updating Numbers and Sets

Note

In general, we recommend using `SET` rather than `ADD`.

Use the `ADD` action in an update expression to add a new attribute and its values to an item.

If the attribute already exists, the behavior of `ADD` depends on the attribute's data type:

- If the attribute is a number, and the value you are adding is also a number, the value is mathematically added to the existing attribute. (If the value is a negative number, it is subtracted from the existing attribute.)
- If the attribute is a set, and the value you are adding is also a set, the value is appended to the existing set.

Note

The `ADD` action supports only number and set data types.

To perform multiple `ADD` actions, separate them with commas.

In the following syntax summary:

- The `path` element is the document path to an attribute. The attribute must be either a `Number` or a set data type.
- The `value` element is a number that you want to add to the attribute (for `Number` data types), or a set to append to the attribute (for set types).

```
add-action ::=  
    path value
```

Adding a Number

Assume that the `QuantityOnHand` attribute does not exist. The following AWS CLI example sets `QuantityOnHand` to 5.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "ADD QuantityOnHand :q" \  
  --expression-attribute-values '{":q": {"N": "5"}}' \  
  --return-values ALL_NEW
```

Now that `QuantityOnHand` exists, you can rerun the example to increment `QuantityOnHand` by 5 each time.

Adding Elements to a Set

Assume that the `Color` attribute does not exist. The following AWS CLI example sets `Color` to a string set with two elements.

```
aws dynamodb update-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"789"}}' \  
  --update-expression "ADD Color :v" \  
  --expression-attribute-values '{":v": {"S": "Red", "S": "Blue"}}' \  
  --return-values ALL_NEW
```

```
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "ADD Color :c" \
--expression-attribute-values '{":c": {"SS":["Orange", "Purple"]}}' \
--return-values ALL_NEW
```

Now that `Color` exists, you can add more elements to it.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "ADD Color :c" \
--expression-attribute-values '{":c": {"SS":["Yellow", "Green", "Blue"]}}' \
--return-values ALL_NEW
```

DELETE—Removing Elements from a Set

Important

The `DELETE` action supports only Set data types.

Use the `DELETE` action in an update expression to remove one or more elements from a set. To perform multiple `DELETE` actions, separate them with commas.

In the following syntax summary:

- The `path` element is the document path to an attribute. The attribute must be a set data type.
- The `subset` is one or more elements that you want to delete from `path`. You must specify `subset` as a set type.

```
delete-action ::=  
    path value
```

Example

In [Adding Elements to a Set \(p. 407\)](#), you create the `Colors` string set. This example removes some of the elements from that set.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"789"}}' \
--update-expression "DELETE Color :p" \
--expression-attribute-values '{":p": {"SS": ["Yellow", "Purple"]}}' \
--return-values ALL_NEW
```

Expiring Items By Using DynamoDB Time to Live (TTL)

Amazon DynamoDB Time to Live (TTL) allows you to define a per-item timestamp to determine when an item is no longer needed. Shortly after the date and time of the specified timestamp, DynamoDB deletes

the item from your table without consuming any write throughput. TTL is provided at no extra cost as a means to reduce stored data volumes by retaining only the items that remain current for your workload's needs.

TTL is useful if you store items that lose relevance after a specific time. The following are example TTL use cases:

- Remove user or sensor data after one year of inactivity in an application.
- Archive expired items to an Amazon S3 data lake via DynamoDB Streams and AWS Lambda.
- Retain sensitive data for a certain amount of time according to contractual or regulatory obligations.

For more information about TTL, see these topics:

- [Using Time to Live](#).
- [Time to Live: How It Works](#).
- [Enabling Time to Live](#).

How It Works: DynamoDB Time to Live (TTL)

When enabling TTL on a DynamoDB table, you must identify a specific attribute name that the service will look for when determining if an item is eligible for expiration. After you enable TTL on a table, a per-partition scanner background process automatically and continuously evaluates the expiry status of items in the table.

The scanner background process compares the current time, in [Unix epoch time format in seconds](#), to the value stored in the user-defined attribute of an item. If the attribute is a `Number` data type, the attribute's value is a timestamp in [Unix epoch time format in seconds](#), is older than the current time, but not five years or older, then the item is set to expired. For specifics about formatting TTL attributes, see [Formatting the item's TTL attribute](#). A second background process scans for expired items and deletes them. Both processes take place automatically in the background, do not affect read or write traffic to the table, and do not have a monetary cost.

As items are deleted from the table, two background operations happen simultaneously:

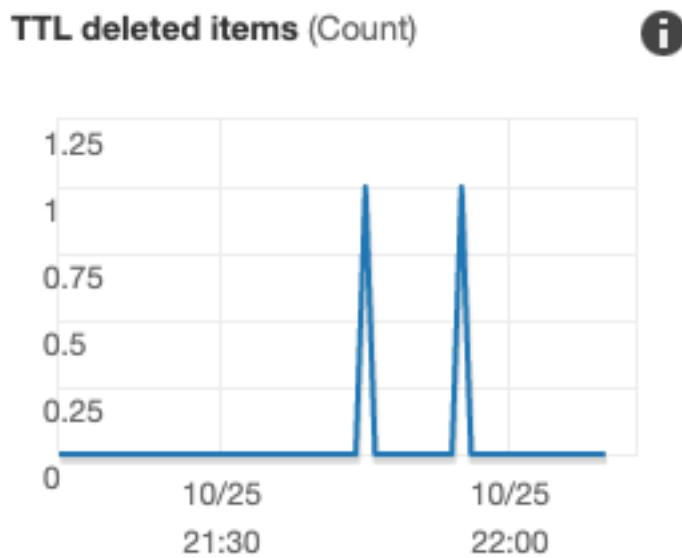
- Items are removed from any local secondary index and global secondary index in the same way as a `DeleteItem` operation.
- A delete operation for each item enters the DynamoDB Stream, but is tagged as a system delete and not a regular delete. For more information about how to use this system delete, see [DynamoDB Streams and Time to Live](#).

Important

- Depending on the size and activity level of a table, the actual delete operation of an expired item can vary. Because TTL is meant to be a background process, the nature of the capacity used to expire and delete items via TTL is variable (but free of charge). Deletion of an item can take up to 48 hours.
- Items that have expired, but haven't yet been deleted by TTL, still appear in reads, queries, and scans. If you do not want expired items in the result set, you must filter them out. To do this, use a filter expression that returns only items where the Time to Live expiration value is greater than the current time in epoch format. For more information, see [Filter Expressions for Scan](#).
- Items that are past their expiration, but have not yet been deleted can still be updated, and successful updates to change or remove the expiration attribute will be honored.

You can monitor TTL rates on the CloudWatch metrics tab for a table, and see when and the rate at which items are deleted.

Time to Live (TTL)



Time to Live Example

For example, consider a table named `SessionData` that tracks the session history of users. Each item in `SessionData` is identified by the partition key (`UserName`) and the sort key (`SessionId`). Additional attributes like `UserName`, `SessionId`, `CreationTime`, and `ExpirationTime` track the session information. The `ExpirationTime` attribute is set as the `TTL` attribute on the table (not all of the attributes on each item are shown).

SessionData

| UserName | SessionId | CreationTime | ExpirationTime (TTL) | SessionInfo | ... |
|----------|--------------------------|--------------|----------------------|-------------|-----|
| user1 | 746865726527731571820360 | 1571827560 | {JSON Document} | ... | ... |
| user2 | 6e6f7468696e671571820180 | 1571827380 | {JSON Document} | ... | ... |
| user3 | 746f20736565201571820923 | 1571828123 | {JSON Document} | ... | ... |
| user4 | 686572652121211571820683 | 1571827883 | {JSON Document} | ... | ... |
| user5 | 6e6572642e2e2e1571820743 | 1571831543 | {JSON Document} | ... | ... |
| ... | ... | ... | ... | ... | ... |

In this example, each item has an `ExpirationTime` attribute value set when it is created. Consider the following table item.

SessionData

| UserName | SessionId | CreationTime | ExpirationTime (TTL) | SessionInfo | ... |
|----------|----------------|--------------|----------------------|-----------------|-----|
| user1 | 74686572652773 | 1571820360 | 1571827560 | {JSON Document} | ... |

In this example, the item `CreationTime` is set to Wednesday, October 23 08:46 AM UTC 2019, and the `ExpirationTime` is set 2 hours later at Wednesday, October 23 10:46 AM UTC 2019. The item expires when the current time, in epoch format, is greater than the time in the `ExpirationTime` attribute. In this case, the item with the key `{ Username: user1, SessionId: 74686572652773 }` expires 10:00 AM (1571827560).

Using DynamoDB Time to Live (TTL)

When using TTL, most of the hard work is done behind the scenes by DynamoDB on your behalf. You should, though, be aware of a few considerations to help your implementation proceed smoothly.

Topics

- [Formatting an item's TTL attribute \(p. 411\)](#)
- [Usage Notes \(p. 411\)](#)
- [Troubleshooting TTL \(p. 412\)](#)

Formatting an item's TTL attribute

When enabling TTL on a table, DynamoDB requires you to identify a specific attribute name that the service will look for when determining if an item is eligible for expiration. In addition, further requirements ensure that the background TTL processes uses the value of the TTL attribute. If an item is to be eligible for expiration via TTL:

- The item must contain the attribute specified when TTL was enabled on the table. For example, if you specify for a table to use the attribute name `expdate` as the TTL attribute, but an item does not have an attribute with that name, the TTL process ignores the item.
- The TTL attribute's value must be a `Number` data type. For example, if you specify for a table to use the attribute name `expdate` as the TTL attribute, but the attribute on an item is a `String` data type, the TTL processes ignore the item.
- The TTL attribute's value must be a timestamp in [Unix epoch time format in seconds](#). If you use any other format, the TTL processes ignore the item. For example, if you set the value of the attribute to 1645119622, that is Thursday, February 17, 2022 17:40:22 (GMT), the item will be expired after that time. For a visual web form to test values and see code examples for different languages for epoch time format, see [Epoch Converter](#).
- The TTL attribute value must be a datetimestamp with an expiration of more than five years in the past. For example, if you set the value of the attribute to 1171734022, that is February 17, 2007 17:40:22 (GMT) and older than five years. As a result the TTL processes do not expire the item.

Usage Notes

When using TTL, consider the following:

- Enabling, disabling, or changing TTL settings on a table can take approximately one hour for the settings to propagate and to allow the execution of any further TTL related actions.
- You cannot reconfigure TTL to look for a different attribute. You must disable TTL, and then reenable TTL with the new attribute going forward.
- When you use AWS CloudFormation, you can enable TTL when creating a DynamoDB table. For more information, see the [AWS CloudFormation User Guide](#).
- You can use AWS Identity and Access Management (IAM) policies to prevent unauthorized updates to the TTL attribute on an item or the configuration of TTL. If you allow access to only specified actions in your existing IAM policies, ensure that your policies are updated to allow dynamodb:UpdateTimeToLive for roles that need to enable or disable TTL on tables. For more information, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB](#).
- Consider whether you need to do any post processing of deleted items via DynamoDB Streams, such as archiving items to an Amazon S3 data lake. The streams records of TTL deletes are marked as system deletes and as normal deletes, and you can filter for system deletes by using an AWS Lambda function. For more information about the additions to the streams records, see [DynamoDB Streams and Time to Live](#).
- If data recovery is a concern, we recommend that you back up your tables.
 - For fully managed table backups, use DynamoDB [on-demand backups](#) or [continuous backups with point-in-time recovery](#).
 - For a 24-hour recovery window, you can use DynamoDB Streams. For more information, see [DynamoDB Streams and Time to Live](#).

Troubleshooting TTL

If DynamoDB TTL is not working, check the following:

- Confirm that you have enabled TTL on the table, and the name of the attribute selected for TTL is set to what your code is writing into items. You can confirm this information on a table's *Overview* tab on the DynamoDB console.
- Look at Amazon CloudWatch metrics on the *Metrics* tab of the DynamoDB console to confirm that TTL is deleting items as you expect them to be deleted..
- Confirm that the TTL attribute value is properly formatted. For more information see, [Formatting an item's TTL attribute \(p. 411\)](#).

Enabling Time to Live (TTL)

You can use the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI) to enable Time to Live. To use the API instead, see [Amazon DynamoDB API Reference](#).

Topics

- [Enable Time to Live \(Console\) \(p. 412\)](#)
- [Enable Time to Live \(AWS CLI\) \(p. 414\)](#)

Enable Time to Live (Console)

Follow these steps to enable Time to Live using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Tables**, and then choose the table that you want to modify.
3. In **Table details**, next to **TTL attribute**, choose **Manage TTL**.

Table details

| | |
|------------------------|-------------------------------------|
| Table name | ttl-test |
| Primary partition key | name (String) |
| Primary sort key | - |
| Time to live attribute | DISABLED Manage TTL |
| Table status | Active |
| Creation date | April 20, 2016 at 2:23:18 PM UTC-7 |

- In the **Manage TTL** dialog box, choose **Enable TTL**, and then enter the **TTL attribute** name.

Enable TTL

TTL is a mechanism to set a specific timestamp for expiring items from your table. The timestamp should be expressed as an attribute on the items in the table. The attribute should be a Number data type containing time in epoch format. Once the timestamp expires, the corresponding item is deleted from the table in the background.

TTL attribute

DynamoDB Streams Enable with view type **New and old images**
 Streams are currently not enabled

Preview TTL

Before enabling TTL, it is recommended you run a preview to see samples of what items will be deleted once TTL is enabled on this table.

Run preview preview items expiring by : UTC-8

Cancel **Continue**

There are three settings in **Manage TTL**:

- Enable TTL** – Choose this to either enable or disable TTL on the table. It can take up to one hour for the change to fully process.
 - TTL Attribute** – The name of the DynamoDB attribute to store the TTL timestamp for items.
 - 24-hour backup streams** – Choose this setting to enable Amazon DynamoDB Streams on the table. For more information about how you can use DynamoDB Streams for backup, see [DynamoDB Streams and Time to Live \(p. 578\)](#).
- (Optional) To preview some of the items that will be deleted when TTL is enabled, choose **Run preview**.
 - Warning**
 This provides you with a sample list of items. It does not provide you with a complete list of items that will be deleted by TTL.
 - Choose **Continue** to save the settings and enable TTL.

Now that TTL is enabled, the TTL attribute is marked **TTL** when you view items on the DynamoDB console.

You can view the date and time that an item expires by hovering your pointer over the attribute.

| | id | ttl (TTL) | |
|--|-----------|------------------|---|
| | 7 | 1460232057 | |
| | 10 | 1459700753 | |
| | 3 | 1459221815 | UTC: March 29, 2016 at 3:23:35 AM UTC Local: March 28, 2016 at 8:23:35 PM UTC-7 Region (N. Virginia): March 28, 2016 at 11:23:35 PM UTC-4 |
| | 2 | 1459221806 | |
| | 21 | 1459703052 | |

Enable Time to Live (AWS CLI)

1. Enable TTL on the TTLExample table.

```
aws dynamodb update-time-to-live --table-name TTLExample --time-to-live-specification
  "Enabled=true, AttributeName=ttl"
```

2. Describe TTL on the TTLExample table.

```
aws dynamodb describe-time-to-live --table-name TTLExample
{
    "TimeToLiveDescription": {
        "AttributeName": "ttl",
        "TimeToLiveStatus": "ENABLED"
    }
}
```

3. Add an item to the TTLExample table with the Time to Live attribute set using the BASH shell and the AWS CLI.

```
EXP=`date -d '+5 days' +%%s`
aws dynamodb put-item --table-name "TTLExample" --item '{"id": {"N": "1"}, "ttl": {"N": "'$EXP'"}}'
```

This example starts with the current date and adds 5 days to it to create an expiration time. Then, it converts the expiration time to epoch time format to finally add an item to the "TTLExample" table.

Note

One way to set expiration values for Time to Live is to calculate the number of seconds to add to the expiration time. For example, 5 days is 432,000 seconds. However, it is often preferable to start with a date and work from there.

It is fairly simple to get the current time in epoch time format, as in the following examples.

- Linux Terminal: `date +%%s`
- Python: `import time; int(time.time())`
- Java: `System.currentTimeMillis() / 1000L`
- JavaScript: `Math.floor(Date.now() / 1000)`

Working with Items: Java

You can use the AWS SDK for Java Document API to perform typical create, read, update, and delete (CRUD) operations on Amazon DynamoDB items in a table.

Note

The SDK for Java also provides an object persistence model, allowing you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

This section contains Java examples to perform several Java Document API item actions and several complete working examples.

Topics

- [Putting an Item \(p. 415\)](#)
- [Getting an Item \(p. 418\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 420\)](#)
- [Batch Get: Getting Multiple Items \(p. 421\)](#)
- [Updating an Item \(p. 422\)](#)
- [Deleting an Item \(p. 423\)](#)
- [Example: CRUD Operations Using the AWS SDK for Java Document API \(p. 424\)](#)
- [Example: Batch Operations Using AWS SDK for Java Document API \(p. 428\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 432\)](#)

Putting an Item

The `putItem` method stores an item in a table. If the item exists, it replaces the entire item. Instead of replacing the entire item, if you want to update only specific attributes, you can use the `updateItem` method. For more information, see [Updating an Item \(p. 422\)](#).

Follow these steps:

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the table you want to work with.
3. Create an instance of the `Item` class to represent the new item. You must specify the new item's primary key and its attributes.
4. Call the `putItem` method of the `Table` object, using the `Item` that you created in the preceding step.

The following Java code example demonstrates the preceding tasks. The code writes a new item to the `ProductCatalog` table.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

// Build a list of related items
List<Number> relatedItems = new ArrayList<Number>();
relatedItems.add(341);
relatedItems.add(472);
relatedItems.add(649);
```

```

//Build a map of product pictures
Map<String, String> pictures = new HashMap<String, String>();
pictures.put("FrontView", "http://example.com/products/123_front.jpg");
pictures.put("RearView", "http://example.com/products/123_rear.jpg");
pictures.put("SideView", "http://example.com/products/123_left_side.jpg");

//Build a map of product reviews
Map<String, List<String>> reviews = new HashMap<String, List<String>>();

List<String> fiveStarReviews = new ArrayList<String>();
fiveStarReviews.add("Excellent! Can't recommend it highly enough! Buy it!");
fiveStarReviews.add("Do yourself a favor and buy this");
reviews.put("FiveStar", fiveStarReviews);

List<String> oneStarReviews = new ArrayList<String>();
oneStarReviews.add("Terrible product! Do not buy this.");
reviews.put("OneStar", oneStarReviews);

// Build the item
Item item = new Item()
    .withPrimaryKey("Id", 123)
    .withString("Title", "Bicycle 123")
    .withString("Description", "123 description")
    .withString("BicycleType", "Hybrid")
    .withString("Brand", "Brand-Company C")
    .withNumber("Price", 500)
    .withStringSet("Color", new HashSet<String>(Arrays.asList("Red", "Black")))
    .withString("ProductCategory", "Bicycle")
    .withBoolean("InStock", true)
    .withNull("QuantityOnHand")
    .withList("RelatedItems", relatedItems)
    .withMap("Pictures", pictures)
    .withMap("Reviews", reviews);

// Write the item to the table
PutItemOutcome outcome = table.putItem(item);

```

In the preceding example, the item has attributes that are scalars (`String`, `Number`, `Boolean`, `Null`), sets (`String Set`), and document types (`List`, `Map`).

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters to the `putItem` method. For example, the following Java code example uses an optional parameter to specify a condition for uploading the item. If the condition you specify is not met, the AWS SDK for Java throws a `ConditionalCheckFailedException`. The code example specifies the following optional parameters in the `putItem` method:

- A `ConditionExpression` that defines the conditions for the request. The code defines the condition that the existing item with the same primary key is replaced only if it has an `ISBN` attribute that equals a specific value.
- A map for `ExpressionAttributeValues` that is used in the condition. In this case, there is only one substitution required: The placeholder `:val` in the condition expression is replaced at runtime with the actual `ISBN` value to be checked.

The following example adds a new book item using these optional parameters.

Example

```
Item item = new Item()
```

```

.withPrimaryKey("Id", 104)
.withString("Title", "Book 104 Title")
.withString("ISBN", "444-444444444")
.withNumber("Price", 20)
.withStringSet("Authors",
    new HashSet<String>(Arrays.asList("Author1", "Author2")));

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", "444-444444444");

PutItemOutcome outcome = table.putItem(
    item,
    "ISBN = :val", // ConditionExpression parameter
    null,           // ExpressionAttributeNames parameter - we're not using it for this
    example
    expressionAttributeValues);

```

PutItem and JSON Documents

You can store a JSON document as an attribute in a DynamoDB table. To do this, use the `withJSON` method of `Item`. This method parses the JSON document and maps each element to a native DynamoDB data type.

Suppose that you wanted to store the following JSON document, containing vendors that can fulfill orders for a particular product.

Example

```
{
    "V01": {
        "Name": "Acme Books",
        "Offices": [ "Seattle" ]
    },
    "V02": {
        "Name": "New Publishers, Inc.",
        "Offices": [ "London", "New York" ]
    },
    "V03": {
        "Name": "Better Buy Books",
        "Offices": [ "Tokyo", "Los Angeles", "Sydney" ]
    }
}
```

You can use the `withJSON` method to store this in the `ProductCatalog` table, in a `Map` attribute named `VendorInfo`. The following Java code example demonstrates how to do this.

```

// Convert the document into a String. Must escape all double-quotes.
String vendorDocument = "{"
    + "    \"V01\": {"
    + "        \"Name\": \"Acme Books\","
    + "        \"Offices\": [ \"Seattle\" ]"
    + "    },"
    + "    \"V02\": {"
    + "        \"Name\": \"New Publishers, Inc.\","
    + "        \"Offices\": [ \"London\", \"New York\" + \"\"]" + "},"
    + "    \"V03\": {"
    + "        \"Name\": \"Better Buy Books\","
    + "        \"Offices\": [ \"Tokyo\", \"Los Angeles\", \"Sydney\" ]"
    + "    }"
}
```

```

+ "      }"
+ "    }";

Item item = new Item()
    .withPrimaryKey("Id", 210)
    .withString("Title", "Book 210 Title")
    .withString("ISBN", "210-2102102102")
    .withNumber("Price", 30)
    .withJSON("VendorInfo", vendorDocument);

PutItemOutcome outcome = table.putItem(item);

```

Getting an Item

To retrieve a single item, use the `getItem` method of a `Table` object. Follow these steps:

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the table you want to work with.
3. Call the `getItem` method of the `Table` instance. You must specify the primary key of the item that you want to retrieve.

The following Java code example demonstrates the preceding steps. The code gets the item that has the specified partition key.

```

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Item item = table.getItem("Id", 101);

```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `getItem` method. For example, the following Java code example uses an optional method to retrieve only a specific list of attributes and to specify strongly consistent reads. (To learn more about read consistency, see [Read Consistency \(p. 16\)](#).)

You can use a `ProjectionExpression` to retrieve only specific attributes or elements, rather than an entire item. A `ProjectionExpression` can specify top-level or nested attributes using document paths. For more information, see [Projection Expressions \(p. 388\)](#).

The parameters of the `getItem` method don't let you specify read consistency. However, you can create a `GetItemSpec`, which provides full access to all of the inputs to the low-level `GetItem` operation. The following code example creates a `GetItemSpec` and uses that spec as input to the `getItem` method.

Example

```

GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 206)
    .withProjectionExpression("Id, Title, RelatedItems[0], Reviews.FiveStar")
    .withConsistentRead(true);

Item item = table.getItem(spec);

System.out.println(item.toJSONString());

```

To print an `Item` in a human-readable format, use the `toJSONPretty` method. The output from the previous example looks like the following.

```
{
    "RelatedItems" : [ 341 ],
    "Reviews" : {
        "FiveStar" : [ "Excellent! Can't recommend it highly enough! Buy it!", "Do yourself a
favor and buy this" ]
    },
    "Id" : 123,
    "Title" : "20-Bicycle 123"
}
```

GetItem and JSON Documents

In the [PutItem and JSON Documents \(p. 417\)](#) section, you store a JSON document in a `Map` attribute named `VendorInfo`. You can use the `getItem` method to retrieve the entire document in JSON format. Or you can use document path notation to retrieve only some of the elements in the document. The following Java code example demonstrates these techniques.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210);

System.out.println("All vendor info:");
spec.withProjectionExpression("VendorInfo");
System.out.println(table.getItem(spec).toJSON());

System.out.println("A single vendor:");
spec.withProjectionExpression("VendorInfo.V03");
System.out.println(table.getItem(spec).toJSON());

System.out.println("First office location for this vendor:");
spec.withProjectionExpression("VendorInfo.V03.Offices[0]");
System.out.println(table.getItem(spec).toJSON());
```

The output from the previous example looks like the following.

```
All vendor info:
{"VendorInfo": {"V03": {"Name": "Better Buy Books", "Offices": ["Tokyo", "Los
Angeles", "Sydney"]}, "V02": {"Name": "New Publishers, Inc.", "Offices": ["London", "New
York"]}, "V01": {"Name": "Acme Books", "Offices": ["Seattle"]}}}
A single vendor:
{"VendorInfo": {"V03": {"Name": "Better Buy Books", "Offices": ["Tokyo", "Los
Angeles", "Sydney"]}}}
First office location for a single vendor:
{"VendorInfo": {"V03": {"Offices": ["Tokyo"]}}}
```

Note

You can use the `toJSON` method to convert any item (or its attributes) to a JSON-formatted string. The following code retrieves several top-level and nested attributes and prints the results as JSON.

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("Id", 210)
    .withProjectionExpression("VendorInfo.V01, Title, Price");

Item item = table.getItem(spec);
System.out.println(item.toJSON());
```

The output looks like the following.

```
{"VendorInfo":{"V01":{"Name":"Acme Books","Offices":["Seattle"]}}, "Price":30, "Title":"Book 210 Title"}
```

Batch Write: Putting and Deleting Multiple Items

Batch write refers to putting and deleting multiple items in a batch. The `batchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to put or delete multiple items using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `TableWriteItems` class that describes all the put and delete operations for a table. If you want to write to multiple tables in a single batch write operation, you must create one `TableWriteItems` instance per table.
3. Call the `batchWriteItem` method by providing the `TableWriteItems` objects that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput limit or some other transient error. Also, DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

The following Java code example demonstrates the preceding steps. The example performs a `batchWriteItem` operation on two tables: `Forum` and `Thread`. The corresponding `TableWriteItems` objects define the following actions:

- Put an item in the `Forum` table.
- Put and delete an item in the `Thread` table.

The code then calls `batchWriteItem` to perform the operation.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableWriteItems forumTableWriteItems = new TableWriteItems("Forum")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("Name", "Amazon RDS")
            .withNumber("Threads", 0));

TableWriteItems threadTableWriteItems = new TableWriteItems("Thread")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("ForumName", "Amazon RDS", "Subject", "Amazon RDS Thread 1")
            .withHashAndRangeKeysToDelete("ForumName", "Some partition key value", "Amazon S3",
                "Some sort key value"));

BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
    threadTableWriteItems);

// Code for checking unprocessed items is omitted in this example
```

For a working example, see [Example: Batch Write Operation Using the AWS SDK for Java Document API \(p. 428\)](#).

Batch Get: Getting Multiple Items

The `batchGetItem` method enables you to retrieve multiple items from one or more tables. To retrieve a single item, you can use the `getItem` method.

Follow these steps:

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `TableKeysAndAttributes` class that describes a list of primary key values to retrieve from a table. If you want to read from multiple tables in a single batch get operation, you must create one `TableKeysAndAttributes` instance per table.
3. Call the `batchGetItem` method by providing the `TableKeysAndAttributes` objects that you created in the preceding step.

The following Java code example demonstrates the preceding steps. The example retrieves two items from the `Forum` table and three items from the `Thread` table.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
"Amazon S3",
"Amazon DynamoDB");

TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
"Amazon DynamoDB", "DynamoDB Thread 1",
"Amazon DynamoDB", "DynamoDB Thread 2",
"Amazon S3", "S3 Thread 1");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(
    forumTableKeysAndAttributes, threadTableKeysAndAttributes);

for (String tableName : outcome.getTableItems().keySet()) {
    System.out.println("Items in table " + tableName);
    List<Item> items = outcome.getTableItems().get(tableName);
    for (Item item : items) {
        System.out.println(item);
    }
}
```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters when using `batchGetItem`. For example, you can provide a `ProjectionExpression` with each `TableKeysAndAttributes` you define. This allows you to specify the attributes that you want to retrieve from the table.

The following code example retrieves two items from the `Forum` table. The `withProjectionExpression` parameter specifies that only the `Threads` attribute is to be retrieved.

Example

```
TableKeysAndAttributes forumTableKeysAndAttributes = new TableKeysAndAttributes("Forum")
    .withProjectionExpression("Threads");
```

```

forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name",
    "Amazon S3",
    "Amazon DynamoDB");

BatchGetItemOutcome outcome = dynamoDB.batchGetItem(forumTableKeysAndAttributes);

```

Updating an Item

The `updateItem` method of a `Table` object can update existing attribute values, add new attributes, or delete attributes from an existing item.

The `updateItem` method behaves as follows:

- If an item does not exist (no item in the table with the specified primary key), `updateItem` adds a new item to the table.
- If an item exists, `updateItem` performs the update as specified by the `UpdateExpression` parameter.

Note

It is also possible to "update" an item using `putItem`. For example, if you call `putItem` to add an item to the table, but there is already an item with the specified primary key, `putItem` replaces the entire item. If there are attributes in the existing item that are not specified in the input, `putItem` removes those attributes from the item.

In general, we recommend that you use `updateItem` whenever you want to modify any item attributes. The `updateItem` method only modifies the item attributes that you specify in the input, and the other attributes in the item remain unchanged.

Follow these steps:

1. Create an instance of the `Table` class to represent the table that you want to work with.
2. Call the `updateTable` method of the `Table` instance. You must specify the primary key of the item that you want to retrieve, along with an `UpdateExpression` that describes the attributes to modify and how to modify them.

The following Java code example demonstrates the preceding tasks. The code updates a book item in the `ProductCatalog` table. It adds a new author to the set of `Authors` and deletes the existing `ISBN` attribute. It also reduces the price by one.

An `ExpressionAttributeValues` map is used in the `UpdateExpression`. The placeholders `:val1` and `:val2` are replaced at runtime with the actual values for `Authors` and `Price`.

```

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#A", "Authors");
expressionAttributeNames.put("#P", "Price");
expressionAttributeNames.put("#I", "ISBN");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1",
    new HashSet<String>(Arrays.asList("Author YY","Author ZZ")));
expressionAttributeValues.put(":val2", 1);    //Price

UpdateItemOutcome outcome = table.updateItem(
    "Id",           // key attribute name
    ...
    "Set"
);

```

```

101,           // key attribute value
"add #A :val1 set #P = #P - :val2 remove #I", // UpdateExpression
expressionAttributeNames,
expressionAttributeValues);

```

Specifying Optional Parameters

Along with the required parameters, you can also specify optional parameters for the `updateItem` method, including a condition that must be met in order for the update is to occur. If the condition you specify is not met, the AWS SDK for Java throws a `ConditionalCheckFailedException`. For example, the following Java code example conditionally updates a book item price to 25. It specifies a `ConditionExpression` stating that the price should be updated only if the existing price is 20.

Example

```

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#P", "Price");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1", 25); // update Price to 25...
expressionAttributeValues.put(":val2", 20); //...but only if existing Price is 20

UpdateItemOutcome outcome = table.updateItem(
    new PrimaryKey("Id",101),
    "set #P = :val1", // UpdateExpression
    "#P = :val2", // ConditionExpression
    expressionAttributeNames,
    expressionAttributeValues);

```

Atomic Counter

You can use `updateItem` to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To increment an atomic counter, use an `UpdateExpression` with a `set` action to add a numeric value to an existing attribute of type `Number`.

The following example demonstrates this, incrementing the `Quantity` attribute by one. It also demonstrates the use of the `ExpressionAttributeNames` parameter in an `UpdateExpression`.

```

Table table = dynamoDB.getTable("ProductCatalog");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#p", "PageCount");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", 1);

UpdateItemOutcome outcome = table.updateItem(
    "Id", 121,
    "set #p = #p + :val",
    expressionAttributeNames,
    expressionAttributeValues);

```

Deleting an Item

The `deleteItem` method deletes an item from a table. You must provide the primary key of the item that you want to delete.

Follow these steps:

1. Create an instance of the DynamoDB client.
2. Call the `deleteItem` method by providing the key of the item you want to delete.

The following Java example demonstrates these tasks.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("ProductCatalog");

DeleteItemOutcome outcome = table.deleteItem("Id", 101);
```

Specifying Optional Parameters

You can specify optional parameters for `deleteItem`. For example, the following Java code example specifies a `ConditionExpression`, stating that a book item in `ProductCatalog` can only be deleted if the book is no longer in publication (the `InPublication` attribute is false).

Example

```
Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("Id", 103,
    "InPublication = :val",
    null, // ExpressionAttributeNames - not used in this example
    expressionAttributeValues);
```

Example: CRUD Operations Using the AWS SDK for Java Document API

The following code example illustrates CRUD operations on an Amazon DynamoDB item. The example creates an item, retrieves it, performs various updates, and finally deletes the item.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 */
```

```

/*
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DeleteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.UpdateItemOutcome;
import com.amazonaws.services.dynamodbv2.document.spec.DeleteItemSpec;
import com.amazonaws.services.dynamodbv2.document.spec.UpdateItemSpec;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.ReturnValue;

public class DocumentAPIItemCRUDExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws IOException {
        createItems();

        retrieveItem();

        // Perform various updates.
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Delete the item.
        deleteItem();
    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tableName);
        try {

            Item item = new Item().withPrimaryKey("Id", 120).withString("Title", "Book 120
Title")
                .withString("ISBN", "120-1111111111")
                .withStringSet("Authors", new HashSet<String>(Arrays.asList("Author12",
"Author22")))
                .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
                .withBoolean("InPublication", false).withString("ProductCategory", "Book");
            table.putItem(item);
        }
    }
}

```

```

        item = new Item().withPrimaryKey("Id", 121).withString("Title", "Book 121
Title")
            .withString("ISBN", "121-1111111111")
            .withStringSet("Authors", new HashSet<String>(Arrays.asList("Author21",
"Author 22")))
            .withNumber("Price", 20).withString("Dimensions",
"8.5x11.0x.75").withNumber("PageCount", 500)
            .withBoolean("InPublication", true).withString("ProductCategory", "Book");
    table.putItem(item);

    }
    catch (Exception e) {
        System.err.println("Create items failed.");
        System.err.println(e.getMessage());
    }
}

private static void retrieveItem() {
    Table table = dynamoDB.getTable(tableName);

    try {

        Item item = table.getItem("Id", 120, "Id, ISBN, Title, Authors", null);

        System.out.println("Printing item after retrieving it....");
        System.out.println(item.toJSONString());

    }
    catch (Exception e) {
        System.err.println("GetItem failed.");
        System.err.println(e.getMessage());
    }
}

private static void updateAddNewAttribute() {
    Table table = dynamoDB.getTable(tableName);

    try {

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id", 121)
            .withUpdateExpression("set #na = :val1").withNameMap(new
NameMap().with("#na", "NewAttribute"))
            .WithValueMap(new ValueMap().withString(":val1", "Some
value")).withReturnValues(ReturnValue.ALL_NEW);

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after adding new attribute...");
        System.out.println(outcome.getItem().toJSONPretty());

    }
    catch (Exception e) {
        System.err.println("Failed to add new attribute in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void updateMultipleAttributes() {
    Table table = dynamoDB.getTable(tableName);

    try {

```

```

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id", 120)
            .withUpdateExpression("add #a :val1 set #na=:val2")
            .withNameMap(new NameMap().with("#a", "Authors").with("#na",
        "NewAttribute"))
            .withValueMap(
                new ValueMap().withStringSet(":val1", "Author YY", "Author
ZZ").withString(":val2", "someValue"))
            .withReturnValues(ReturnValue.ALL_NEW);

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after multiple attribute update...");
        System.out.println(outcome.getItem().toJSONPretty());

    }
    catch (Exception e) {
        System.err.println("Failed to update multiple attributes in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void updateExistingAttributeConditionally() {

    Table table = dynamoDB.getTable(tableName);

    try {

        // Specify the desired price (25.00) and also the condition (price =
        // 20.00)

        UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("Id", 120)
            .withReturnValues(ReturnValue.ALL_NEW).withUpdateExpression("set #p
= :val1")
            .withConditionExpression("#p = :val2").withNameMap(new NameMap().with("#p",
        "Price"))
            .withValueMap(new ValueMap().withNumber(":val1", 25).withNumber(":val2",
        20));

        UpdateItemOutcome outcome = table.updateItem(updateItemSpec);

        // Check the response.
        System.out.println("Printing item after conditional update to new
attribute...");
        System.out.println(outcome.getItem().toJSONPretty());

    }
    catch (Exception e) {
        System.err.println("Error updating item in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void deleteItem() {

    Table table = dynamoDB.getTable(tableName);

    try {

        DeleteItemSpec deleteItemSpec = new DeleteItemSpec().withPrimaryKey("Id", 120)
            .withConditionExpression("#ip = :val").withNameMap(new
NameMap().with("#ip", "InPublication"))
            .withValueMap(new ValueMap().withBoolean(":val",
        false)).withReturnValues(ReturnValue.ALL_OLD);
    }
}

```

```
        DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);

        // Check the response.
        System.out.println("Printing item that was deleted...");
        System.out.println(outcome.getItem().toJSONPretty());

    }
    catch (Exception e) {
        System.err.println("Error deleting item in " + tableName);
        System.err.println(e.getMessage());
    }
}
```

Example: Batch Operations Using AWS SDK for Java Document API

This section provides examples of batch write and batch get operations in Amazon DynamoDB using the AWS SDK for Java Document API.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

Topics

- [Example: Batch Write Operation Using the AWS SDK for Java Document API \(p. 428\)](#)
- [Example: Batch Get Operation Using the AWS SDK for Java Document API \(p. 430\)](#)

Example: Batch Write Operation Using the AWS SDK for Java Document API

The following Java code example uses the `batchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
- Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, `batchWriteItem` limits the size of a batch write request and the number of put and delete operations in a single batch write operation. If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `batchWriteItem` request with unprocessed items in the request. If you followed the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section, you should already have created the `Forum` and `Thread` tables. You can also create these tables and upload sample data programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 982\)](#).

For step-by-step instructions for testing the following sample, see [Java Code Examples \(p. 330\)](#).

Example

```
/**
```

```
* Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
*
* This file is licensed under the Apache License, Version 2.0 (the "License").
* You may not use this file except in compliance with the License. A copy of
* the License is located at
*
* http://aws.amazon.com/apache2.0/
*
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
* CONDITIONS OF ANY KIND, either express or implied. See the License for the
* specific language governing permissions and limitations under the License.
*/



package com.amazonaws.codesamples.document;

import java.io.IOException;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchWriteItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableWriteItems;
import com.amazonaws.services.dynamodbv2.model.WriteRequest;

public class DocumentAPIBatchWrite {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {
        writeMultipleItemsBatchWrite();
    }

    private static void writeMultipleItemsBatchWrite() {
        try {

            // Add a new item to Forum
            TableWriteItems forumTableWriteItems = new TableWriteItems(forumTableName) // Forum
                .withItemsToPut(new Item().withPrimaryKey("Name", "Amazon RDS").withNumber("Threads", 0));

            // Add a new item, and delete an existing item, from Thread
            // This table has a partition key and range key, so need to specify both of them
            TableWriteItems threadTableWriteItems = new TableWriteItems(threadTableName)
                .withItemsToPut(new Item().withPrimaryKey("ForumName", "Amazon RDS", "Subject", "Amazon RDS Thread 1")
                    .withString("Message", "ElastiCache Thread 1 message")
                    .withStringSet("Tags", new HashSet<String>(Arrays.asList("cache", "in-memory"))))
                .withHashAndRangeKeysToDelete("ForumName", "Subject", "Amazon S3", "S3 Thread 100");

            System.out.println("Making the request.");
        }
    }
}
```

```

        BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
threadTableWriteItems);

        do {

            // Check for unprocessed keys which could happen if you exceed
            // provisioned throughput

            Map<String, List<WriteRequest>> unprocessedItems =
outcome.getUnprocessedItems();

            if (outcome.getUnprocessedItems().size() == 0) {
                System.out.println("No unprocessed items found");
            }
            else {
                System.out.println("Retrieving the unprocessed items");
                outcome = dynamoDB.batchWriteItemUnprocessed(unprocessedItems);
            }

            } while (outcome.getUnprocessedItems().size() > 0);

        }
        catch (Exception e) {
            System.err.println("Failed to retrieve items: ");
            e.printStackTrace(System.err);
        }
    }
}

```

Example: Batch Get Operation Using the AWS SDK for Java Document API

The following Java code example uses the `batchGetItem` method to retrieve multiple items from the `Forum` and the `Thread` tables. The `BatchGetItemRequest` specifies the table names and a list of keys for each item to get. The example processes the response by printing the items retrieved.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

Example

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

```

```

import java.io.IOException;
import java.util.List;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.BatchGetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.TableKeysAndAttributes;
import com.amazonaws.services.dynamodbv2.model.KeysAndAttributes;

public class DocumentAPIBatchGet {
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {
        retrieveMultipleItemsBatchGet();
    }

    private static void retrieveMultipleItemsBatchGet() {

        try {

            TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
                // Add a partition key
                forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name", "Amazon S3", "Amazon
DynamoDB");

            TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
                // Add a partition key and a sort key
                threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
"Amazon DynamoDB",
                    "DynamoDB Thread 1", "Amazon DynamoDB", "DynamoDB Thread 2", "Amazon S3",
"S3 Thread 1");

            System.out.println("Making the request.");

            BatchGetItemOutcome outcome =
dynamoDB.batchGetItem(forumTableKeysAndAttributes,
                    threadTableKeysAndAttributes);

            Map<String, KeysAndAttributes> unprocessed = null;

            do {
                for (String tableName : outcome.getTableItems().keySet()) {
                    System.out.println("Items in table " + tableName);
                    List<Item> items = outcome.getTableItems().get(tableName);
                    for (Item item : items) {
                        System.out.println(item.toJSONPretty());
                    }
                }
            }

            // Check for unprocessed keys which could happen if you exceed
            // provisioned
            // throughput or reach the limit on response size.
            unprocessed = outcome.getUnprocessedKeys();

            if (unprocessed.isEmpty()) {
                System.out.println("No unprocessed keys found");
            }
        }
    }
}

```

```

        else {
            System.out.println("Retrieving the unprocessed keys");
            outcome = dynamoDB.batchGetItemUnprocessed(unprocessed);
        }

    } while (!unprocessed.isEmpty());

}

catch (Exception e) {
    System.err.println("Failed to retrieve items.");
    System.err.println(e.getMessage());
}

}
}

```

Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API

The following Java code example illustrates handling binary type attributes. The example adds an item to the `Reply` table. The item includes a binary type attribute (`ExtendedMessage`) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the `GZIPOutputStream` class to compress a sample stream and assign it to the `ExtendedMessage` attribute. When the binary attribute is retrieved, it is decompressed using the `GZIPInputStream` class.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

If you followed the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section, you should already have created the `Reply` table. You can also create this table programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 982\)](#).

For step-by-step instructions for testing the following sample, see [Java Code Examples \(p. 330\)](#).

Example

```


/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/


package com.amazonaws.codesamples.document;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

```

```

import java.io.IOException;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimeZone;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.GetItemSpec;

public class DocumentAPIItemBinaryExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "Reply";
    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    public static void main(String[] args) throws IOException {
        try {

            // Format the primary key values
            String threadId = "Amazon DynamoDB#DynamoDB Thread 2";

            dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));
            String replyDateTime = dateFormatter.format(new Date());

            // Add a new reply with a binary attribute type
            createItem(threadId, replyDateTime);

            // Retrieve the reply with a binary attribute type
            retrieveItem(threadId, replyDateTime);

            // clean up by deleting the item
            deleteItem(threadId, replyDateTime);
        }
        catch (Exception e) {
            System.err.println("Error running the binary attribute type example: " + e);
            e.printStackTrace(System.err);
        }
    }

    public static void createItem(String threadId, String replyDateTime) throws IOException
    {

        Table table = dynamoDB.getTable(tableName);

        // Craft a long message
        String messageInput = "Long message to be compressed in a lengthy forum reply";

        // Compress the long message
        ByteBuffer compressedMessage = compressString(messageInput.toString());

        table.putItem(new Item().withPrimaryKey("Id", threadId).withString("ReplyDateTime",
        replyDateTime)
            .withString("Message", "Long message follows").withBinary("ExtendedMessage",
        compressedMessage)
            .withString("PostedBy", "User A"));
    }
}

```

```

public static void retrieveItem(String threadId, String replyDateTime) throws
IOException {

    Table table = dynamoDB.getTable(tableName);

    GetItemSpec spec = new GetItemSpec().withPrimaryKey("Id", threadId,
"ReplyDateTime", replyDateTime)
        .withConsistentRead(true);

    Item item = table.getItem(spec);

    // Uncompress the reply message and print
    String uncompressed =
uncompressString(ByteBuffer.wrap(item.getBinary("ExtendedMessage")));

    System.out.println("Reply message:\n" + " Id: " + item.getString("Id") + "\n" +
"ReplyDateTime: "
        + item.getString("ReplyDateTime") + "\n" + " PostedBy: " +
item.getString("PostedBy") + "\n" + " Message: "
        + item.getString("Message") + "\n" + " ExtendedMessage (uncompressed): " +
uncompressed + "\n");
}

public static void deleteItem(String threadId, String replyDateTime) {

    Table table = dynamoDB.getTable(tableName);
    table.deleteItem("Id", threadId, "ReplyDateTime", replyDateTime);
}

private static ByteBuffer compressString(String input) throws IOException {
    // Compress the UTF-8 encoded String into a byte[]
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPOutputStream os = new GZIPOutputStream(baos);
    os.write(input.getBytes("UTF-8"));
    os.close();
    baos.close();
    byte[] compressedBytes = baos.toByteArray();

    // The following code writes the compressed bytes to a ByteBuffer.
    // A simpler way to do this is by simply calling
    // ByteBuffer.wrap(compressedBytes);
    // However, the longer form below shows the importance of resetting the
    // position of the buffer
    // back to the beginning of the buffer if you are writing bytes directly
    // to it, since the SDK
    // will consider only the bytes after the current position when sending
    // data to DynamoDB.
    // Using the "wrap" method automatically resets the position to zero.
    ByteBuffer buffer = ByteBuffer.allocate(compressedBytes.length);
    buffer.put(compressedBytes, 0, compressedBytes.length);
    buffer.position(0); // Important: reset the position of the ByteBuffer
                      // to the beginning
    return buffer;
}

private static String uncompressString(ByteBuffer input) throws IOException {
    byte[] bytes = input.array();
    ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    GZIPInputStream is = new GZIPInputStream(bais);

    int chunkSize = 1024;
    byte[] buffer = new byte[chunkSize];
    int length = 0;
    while ((length = is.read(buffer, 0, chunkSize)) != -1) {
        baos.write(buffer, 0, length);
}

```

```
        }

        String result = new String(baos.toByteArray(), "UTF-8");

        is.close();
        baos.close();
        bais.close();

        return result;
    }
}
```

Working with Items: .NET

You can use the AWS SDK for .NET low-level API to perform typical create, read, update, and delete (CRUD) operations on an item in a table. The following are the common steps that you follow to perform data CRUD operations using the .NET low-level API:

1. Create an instance of the `AmazonDynamoDBClient` class (the client).
2. Provide the operation-specific required parameters in a corresponding request object.
For example, use the `PutItemRequest` request object when uploading an item and use the `GetItemRequest` request object when retrieving an existing item.
You can use the request object to provide both the required and optional parameters.
3. Execute the appropriate method provided by the client by passing in the request object that you created in the preceding step.

The `AmazonDynamoDBClient` client provides `PutItem`, `GetItem`, `UpdateItem`, and `DeleteItem` methods for the CRUD operations.

Topics

- [Putting an Item \(p. 435\)](#)
- [Getting an Item \(p. 437\)](#)
- [Updating an Item \(p. 438\)](#)
- [Atomic Counter \(p. 440\)](#)
- [Deleting an Item \(p. 440\)](#)
- [Batch Write: Putting and Deleting Multiple Items \(p. 441\)](#)
- [Batch Get: Getting Multiple Items \(p. 443\)](#)
- [Example: CRUD Operations Using the AWS SDK for .NET Low-Level API \(p. 445\)](#)
- [Example: Batch Operations Using the AWS SDK for .NET Low-Level API \(p. 449\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 455\)](#)

Putting an Item

The `PutItem` method uploads an item to a table. If the item exists, it replaces the entire item.

Note

Instead of replacing the entire item, if you want to update only specific attributes, you can use the `UpdateItem` method. For more information, see [Updating an Item \(p. 438\)](#).

The following are the steps to upload an item using the low-level .NET SDK API:

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Provide the required parameters by creating an instance of the `PutItemRequest` class.

To put an item, you must provide the table name and the item.

3. Execute the `PutItem` method by providing the `PutItemRequest` object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example uploads an item to the `ProductCatalog` table.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new PutItemRequest
{
    TableName = tableName,
    Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue { N = "201" } },
        { "Title", new AttributeValue { S = "Book 201 Title" } },
        { "ISBN", new AttributeValue { S = "11-11-11-11" } },
        { "Price", new AttributeValue { S = "20.00" } },
        {
            "Authors",
            new AttributeValue
            { SS = new List<string>{"Author1", "Author2"} }
        }
    }
};
client.PutItem(request);
```

In the preceding example, you upload a book item that has the `Id`, `Title`, `ISBN`, and `Authors` attributes. Note that `Id` is a numeric type attribute, and all other attributes are of the string type. `Authors` is a `String` set.

Specifying Optional Parameters

You can also provide optional parameters using the `PutItemRequest` object as shown in the following C# example. The example specifies the following optional parameters:

- `ExpressionAttributeNames`, `ExpressionAttributeValues`, and `ConditionExpression` specify that the item can be replaced only if the existing item has the `ISBN` attribute with a specific value.
- `ReturnValues` parameter to request the old item in the response.

Example

```
var request = new PutItemRequest
{
    TableName = tableName,
    Item = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue { N = "104" } },
        { "Title", new AttributeValue { S = "Book 104 Title" } },
        { "ISBN", new AttributeValue { S = "444-4444444444" } },
        { "Authors",
            new AttributeValue { SS = new List<string>{"Author3"} }
        }
    }
};
```

```
// Optional parameters.
ExpressionAttributeNames = new Dictionary<string, string>()
{
    {"#I", "ISBN"}
},
ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
{
    {":isbn", new AttributeValue {S = "444-4444444444"}}
},
ConditionExpression = "#I = :isbn"

};

var response = client.PutItem(request);
```

For more information, see [PutItem](#).

Getting an Item

The `GetItem` method retrieves an item.

Note

To retrieve multiple items, you can use the `BatchGetItem` method. For more information, see [Batch Get: Getting Multiple Items \(p. 443\)](#).

The following are the steps to retrieve an existing item using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `GetItemRequest` class.
To get an item, you must provide the table name and primary key of the item.
3. Execute the `GetItem` method by providing the `GetItemRequest` object that you created in the preceding step.

The following C# example demonstrates the preceding steps. The example retrieves an item from the `ProductCatalog` table.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N = "202" } } },
};
var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item; // Attribute list in the response.
```

Specifying Optional Parameters

You can also provide optional parameters using the `GetItemRequest` object, as shown in the following C# example. The sample specifies the following optional parameters:

- `ProjectionExpression` parameter to specify the attributes to retrieve.
- `ConsistentRead` parameter to perform a strongly consistent read. To learn more read consistency, see [Read Consistency \(p. 16\)](#).

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new GetItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
"202" } } },
    // Optional parameters.
    ProjectionExpression = "Id, ISBN, Title, Authors",
    ConsistentRead = true
};

var response = client.GetItem(request);

// Check the response.
var result = response.GetItemResult;
var attributeMap = result.Item;
```

For more information, see [GetItem](#).

Updating an Item

The `UpdateItem` method updates an existing item if it is present. You can use the `UpdateItem` operation to update existing attribute values, add new attributes, or delete attributes from the existing collection. If the item that has the specified primary key is not found, it adds a new item.

The `UpdateItem` operation uses the following guidelines:

- If the item does not exist, `UpdateItem` adds a new item using the primary key that is specified in the input.
- If the item exists, `UpdateItem` applies the updates as follows:
 - Replaces the existing attribute values by the values in the update.
 - If the attribute that you provide in the input does not exist, it adds a new attribute to the item.
 - If the input attribute is null, it deletes the attribute, if it is present.
 - If you use ADD for the `Action`, you can add values to an existing set (string or number set), or mathematically add (use a positive number) or subtract (use a negative number) from the existing numeric attribute value.

Note

The `PutItem` operation also can perform an update. For more information, see [Putting an Item \(p. 435\)](#). For example, if you call `PutItem` to upload an item and the primary key exists, the `PutItem` operation replaces the entire item. If there are attributes in the existing item and those attributes are not specified in the input, the `PutItem` operation deletes those attributes. However, `UpdateItem` updates only the specified input attributes. Any other existing attributes of that item remain unchanged.

The following are the steps to update an existing item using the low-level .NET SDK API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `UpdateItemRequest` class.

This is the request object in which you describe all the updates, such as add attributes, update existing attributes, or delete attributes. To delete an existing attribute, specify the attribute name with null value.

3. Execute the `UpdateItem` method by providing the `UpdateItemRequest` object that you created in the preceding step.

The following C# code example demonstrates the preceding steps. The example updates a book item in the `ProductCatalog` table. It adds a new author to the `Authors` collection, and deletes the existing `ISBN` attribute. It also reduces the price by one.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
    "202" } } },
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
        {"#A", "Authors"},
        {"#P", "Price"},
        {"#NA", "NewAttribute"},
        {"#I", "ISBN"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"::auth",new AttributeValue { SS = {"Author YY","Author ZZ"} }},
        {"::p",new AttributeValue { N = "1"}},
        {"::newattr",new AttributeValue { S = "someValue"}}
    },
    // This expression does the following:
    // 1) Adds two new authors to the list
    // 2) Reduces the price
    // 3) Adds a new attribute to the item
    // 4) Removes the ISBN attribute from the item
    UpdateExpression = "ADD #A :auth SET #P = #P - :p, #NA = :newattr REMOVE #I"
};
var response = client.UpdateItem(request);
```

Specifying Optional Parameters

You can also provide optional parameters using the `UpdateItemRequest` object, as shown in the following C# example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the price can be updated only if the existing price is 20.00.
- `ReturnValues` parameter to request the updated item in the response.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
    "202" } } },
    // Update price only if the current price is 20.00.
    ExpressionAttributeNames = new Dictionary<string,string>()
    {
```

```

        {"#P", "Price"}
},
ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
{
    {":newprice",new AttributeValue {N = "22"}},
    {":currprice",new AttributeValue {N = "20"}}
},
UpdateExpression = "SET #P = :newprice",
ConditionExpression = "#P = :currprice",
TableName = tableName,
ReturnValues = "ALL_NEW" // Return all the attributes of the updated item.
};

var response = client.UpdateItem(request);

```

For more information, see [UpdateItem](#).

Atomic Counter

You can use `updateItem` to implement an atomic counter, where you increment or decrement the value of an existing attribute without interfering with other write requests. To update an atomic counter, use `updateItem` with an attribute of type `Number` in the `UpdateExpression` parameter, and `ADD` as the `Action`.

The following example demonstrates this, incrementing the `Quantity` attribute by one.

```

AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new UpdateItemRequest
{
    Key = new Dictionary<string, AttributeValue>() { { "Id", new AttributeValue { N =
"121" } } },
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#Q", "Quantity"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":incr",new AttributeValue {N = "1"}}
    },
    UpdateExpression = "SET #Q = #Q + :incr",
    TableName = tableName
};

var response = client.UpdateItem(request);

```

Deleting an Item

The `DeleteItem` method deletes an item from a table.

The following are the steps to delete an item using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `DeleteItemRequest` class.

To delete an item, the table name and item's primary key are required.

3. Execute the `DeleteItem` method by providing the `DeleteItemRequest` object that you created in the preceding step.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "ProductCatalog";

var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
    "201" } } },
};

var response = client.DeleteItem(request);
```

Specifying Optional Parameters

You can also provide optional parameters using the `DeleteItemRequest` object as shown in the following C# code example. It specifies the following optional parameters:

- `ExpressionAttributeValues` and `ConditionExpression` to specify that the book item can be deleted only if it is no longer in publication (the `InPublication` attribute value is false).
- `ReturnValues` parameter to request the deleted item in the response.

Example

```
var request = new DeleteItemRequest
{
    TableName = tableName,
    Key = new Dictionary<string,AttributeValue>() { { "Id", new AttributeValue { N =
    "201" } } },

    // Optional parameters.
    ReturnValues = "ALL_OLD",
    ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"::inpub", new AttributeValue {BOOL = false}}
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);
```

For more information, see [DeleteItem](#).

Batch Write: Putting and Deleting Multiple Items

Batch write refers to putting and deleting multiple items in a batch. The `BatchWriteItem` method enables you to put and delete multiple items from one or more tables in a single call. The following are the steps to retrieve multiple items using the low-level .NET SDK API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Describe all the put and delete operations by creating an instance of the `BatchWriteItemRequest` class.
3. Execute the `BatchWriteItem` method by providing the `BatchWriteItemRequest` object that you created in the preceding step.

4. Process the response. You should check if there were any unprocessed request items returned in the response. This could happen if you reach the provisioned throughput limit or some other transient error. Also, DynamoDB limits the request size and the number of operations you can specify in a request. If you exceed these limits, DynamoDB rejects the request. For more information, see [BatchWriteItem](#).

The following C# code example demonstrates the preceding steps. The example creates a `BatchWriteItemRequest` to perform the following write operations:

- Put an item in `Forum` table.
- Put and delete an item from `Thread` table.

The code then executes `BatchWriteItem` to perform a batch operation.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchWriteItemRequest
{
    RequestItems = new Dictionary<string, List<WriteRequest>>
    {
        {
            table1Name, new List<WriteRequest>
            {
                new WriteRequest
                {
                    PutRequest = new PutRequest
                    {
                        Item = new Dictionary<string,AttributeValue>
                        {
                            { "Name", new AttributeValue { S = "Amazon S3 forum" } },
                            { "Threads", new AttributeValue { N = "0" } }
                        }
                    }
                }
            }
        },
        {
            table2Name, new List<WriteRequest>
            {
                new WriteRequest
                {
                    PutRequest = new PutRequest
                    {
                        Item = new Dictionary<string,AttributeValue>
                        {
                            { "ForumName", new AttributeValue { S = "Amazon S3 forum" } },
                            { "Subject", new AttributeValue { S = "My sample question" } },
                            { "Message", new AttributeValue { S = "Message Text." } },
                            { "KeywordTags", new AttributeValue { SS = new List<string> { "Amazon S3", "Bucket" } } }
                        }
                    }
                },
                new WriteRequest
                {
                    DeleteRequest = new DeleteRequest
                    {
                        Key = new Dictionary<string,AttributeValue>()
                    }
                }
            }
        }
    }
};
```

```

        { "ForumName", new AttributeValue { S = "Some forum name" } },
        { "Subject", new AttributeValue { S = "Some subject" } }
    }
}
}
}
};

response = client.BatchWriteItem(request);

```

For a working example, see [Example: Batch Operations Using the AWS SDK for .NET Low-Level API \(p. 449\)](#).

Batch Get: Getting Multiple Items

The `BatchGetItem` method enables you to retrieve multiple items from one or more tables.

Note

To retrieve a single item, you can use the `GetItem` method.

The following are the steps to retrieve multiple items using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required parameters by creating an instance of the `BatchGetItemRequest` class.
To retrieve multiple items, the table name and a list of primary key values are required.
3. Execute the `BatchGetItem` method by providing the `BatchGetItemRequest` object that you created in the preceding step.
4. Process the response. You should check if there were any unprocessed keys, which could happen if you reach the provisioned throughput limit or some other transient error.

The following C# code example demonstrates the preceding steps. The example retrieves items from two tables, `Forum` and `Thread`. The request specifies two items in the `Forum` and three items in the `Thread` table. The response includes items from both of the tables. The code shows how you can process the response.

```

AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string table1Name = "Forum";
string table2Name = "Thread";

var request = new BatchGetItemRequest
{
    RequestItems = new Dictionary<string, KeysAndAttributes>()
    {
        { table1Name,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "DynamoDB" } }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "Amazon S3" } }
                    }
                }
            }
        }
    }
};

```

```

        }
    },
    {
        table2Name,
        new KeysAndAttributes
        {
            Keys = new List<Dictionary<string, AttributeValue>>()
            {
                new Dictionary<string, AttributeValue>()
                {
                    { "ForumName", new AttributeValue { S = "DynamoDB" } },
                    { "Subject", new AttributeValue { S = "DynamoDB Thread 1" } }
                },
                new Dictionary<string, AttributeValue>()
                {
                    { "ForumName", new AttributeValue { S = "DynamoDB" } },
                    { "Subject", new AttributeValue { S = "DynamoDB Thread 2" } }
                },
                new Dictionary<string, AttributeValue>()
                {
                    { "ForumName", new AttributeValue { S = "Amazon S3" } },
                    { "Subject", new AttributeValue { S = "Amazon S3 Thread 1" } }
                }
            }
        }
    };
}

var response = client.BatchGetItem(request);

// Check the response.
var result = response.BatchGetItemResult;
var responses = result.Responses; // The attribute list in the response.

var table1Results = responses[table1Name];
Console.WriteLine("Items in table {0}" + table1Name);
foreach (var item1 in table1Results.Items)
{
    PrintItem(item1);
}

var table2Results = responses[table2Name];
Console.WriteLine("Items in table {1}" + table2Name);
foreach (var item2 in table2Results.Items)
{
    PrintItem(item2);
}
// Any unprocessed keys? could happen if you exceed ProvisionedThroughput or some other
// error.
Dictionary<string, KeysAndAttributes> unprocessedKeys = result.UnprocessedKeys;
foreach (KeyValuePair<string, KeysAndAttributes> pair in unprocessedKeys)
{
    Console.WriteLine(pair.Key, pair.Value);
}

```

Specifying Optional Parameters

You can also provide optional parameters using the `BatchGetItemRequest` object as shown in the following C# code example. The example retrieves two items from the `Forum` table. It specifies the following optional parameter:

- `ProjectionExpression` parameter to specify the attributes to retrieve.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

string tableName = "Forum";

var request = new BatchGetItemRequest
{
    RequestItems = new Dictionary<string, KeysAndAttributes>()
    {
        { tableName,
            new KeysAndAttributes
            {
                Keys = new List<Dictionary<string, AttributeValue>>()
                {
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "DynamoDB" } }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue { S = "Amazon S3" } }
                    }
                },
                // Optional - name of an attribute to retrieve.
                ProjectionExpression = "Title"
            }
        };
    };
};

var response = client.BatchGetItem(request);
```

For more information, see [BatchGetItem](#).

Example: CRUD Operations Using the AWS SDK for .NET Low-Level API

The following C# code example illustrates CRUD operations on an Amazon DynamoDB item. The example adds an item to the `ProductCatalog` table, retrieves it, performs various updates, and finally deletes the item. If you followed the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you already have the `ProductCatalog` table created. You can also create these sample tables programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#).

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

Example

```
/** 
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
```

```

    * specific language governing permissions and limitations under the License.
*/
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class LowLevelItemCRUDExample
    {
        private static string tableName = "ProductCatalog";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                CreateItem();
                RetrieveItem();

                // Perform various updates.
                UpdateMultipleAttributes();
                UpdateExistingAttributeConditionally();

                // Delete item.
                DeleteItem();
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }

        private static void CreateItem()
        {
            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = new Dictionary<string, AttributeValue>()
            };
            {
                { "Id", new AttributeValue {
                    N = "1000"
                }},
                { "Title", new AttributeValue {
                    S = "Book 201 Title"
                }},
                { "ISBN", new AttributeValue {
                    S = "11-11-11-11"
                }},
                { "Authors", new AttributeValue {
                    SS = new List<string>{"Author1", "Author2" }
                }},
                { "Price", new AttributeValue {
                    N = "20.00"
                }},
                { "Dimensions", new AttributeValue {
                    S = "8.5x11.0x.75"
                }},
                { "InPublication", new AttributeValue {

```

```

        BOOL = false
    } }
}
};

client.PutItem(request);
}

private static void RetrieveItem()
{
    var request = new GetItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },
        ProjectionExpression = "Id, ISBN, Title, Authors",
        ConsistentRead = true
    };
    var response = client.GetItem(request);

    // Check the response.
    var attributeList = response.Item; // attribute list in the response.
    Console.WriteLine("\nPrinting item after retrieving it .....");
    PrintItem(attributeList);
}

private static void UpdateMultipleAttributes()
{
    var request = new UpdateItemRequest
    {
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },
        // Perform the following updates:
        // 1) Add two new authors to the list
        // 1) Set a new attribute
        // 2) Remove the ISBN attribute
        ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#A","Authors"}, 
        {"#NA","NewAttribute"}, 
        {"#I","ISBN"}
    },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {"":auth",new AttributeValue {
            SS = {"Author YY", "Author ZZ"}
        }},
        {"":new",new AttributeValue {
            S = "New Value"
        }}
    },
        UpdateExpression = "ADD #A :auth SET #NA = :new REMOVE #I",
        TableName = tableName,
        ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
    };
    var response = client.UpdateItem(request);
}

```

```

        // Check the response.
        var attributeList = response.Attributes; // attribute list in the response.
                                                // print attributeList.
        Console.WriteLine("\nPrinting item after multiple attribute
update .....");
        PrintItem(attributeList);
    }

    private static void UpdateExistingAttributeConditionally()
    {
        var request = new UpdateItemRequest
        {
            Key = new Dictionary<string, AttributeValue>()
        {
            { "Id", new AttributeValue {
                N = "1000"
            } }
        },
        ExpressionAttributeNames = new Dictionary<string, string>()
        {
            {"#P", "Price"}
        },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
        {
            {":newprice",new AttributeValue {
                N = "22.00"
            }},
            {":currprice",new AttributeValue {
                N = "20.00"
            }}
        },
        // This updates price only if current price is 20.00.
        UpdateExpression = "SET #P = :newprice",
        ConditionExpression = "#P = :currprice",

        TableName = tableName,
        ReturnValues = "ALL_NEW" // Give me all attributes of the updated item.
    };
    var response = client.UpdateItem(request);

    // Check the response.
    var attributeList = response.Attributes; // attribute list in the response.
    Console.WriteLine("\nPrinting item after updating price value
conditionally .....");
    PrintItem(attributeList);
}

private static void DeleteItem()
{
    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            N = "1000"
        } }
    },

        // Return the entire item as it appeared before the update.
        ReturnValues = "ALL_OLD",
        ExpressionAttributeNames = new Dictionary<string, string>()
    {
        {"#IP", "InPublication"}
    },
        ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
}

```

```

    {
        ":" :> new AttributeValue {
            BOOL = false
        }
    },
    ConditionExpression = "#IP = :inpub"
};

var response = client.DeleteItem(request);

// Check the response.
var attributeList = response.Attributes; // Attribute list in the response.
                                         // Print item.
Console.WriteLine("\nPrinting item that was just deleted .....");
PrintItem(attributeList);
}

private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
{
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
    {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;

        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[ " + value.S + " ]") +
            (value.N == null ? "" : "N=[ " + value.N + " ]") +
            (value.SS == null ? "" : "SS=[ " + string.Join(", ", value.SS.ToArray()) +
            " ]") +
            (value.NS == null ? "" : "NS=[ " + string.Join(", ", value.NS.ToArray()) +
            " ]")
        );
    }
    Console.WriteLine("*****");
}
}
}

```

Example: Batch Operations Using the AWS SDK for .NET Low-Level API

Topics

- Example: Batch Write Operation Using the AWS SDK for .NET Low-Level API (p. 449)
 - Example: Batch Get Operation Using the AWS SDK for .NET Low-Level API (p. 452)

This section provides examples of batch operations, *batch write* and *batch get*, that Amazon DynamoDB supports.

Example: Batch Write Operation Using the AWS SDK for .NET Low-Level API

The following C# code example uses the `BatchWriteItem` method to perform the following put and delete operations:

- Put one item in the `Forum` table.
 - Put one item and delete one item from the `Thread` table.

You can specify any number of put and delete requests against one or more tables when creating your batch write request. However, `DynamoDBBatchWriteItem` limits the size of a batch write request and

the number of put and delete operations in a single batch write operation. For more information, see [BatchWriteItem](#). If your request exceeds these limits, your request is rejected. If your table does not have sufficient provisioned throughput to serve this request, the unprocessed request items are returned in the response.

The following example checks the response to see if it has any unprocessed request items. If it does, it loops back and resends the `BatchWriteItem` request with unprocessed items in the request. If you followed the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you already have the `Forum` and `Thread` tables created. You can also create these sample tables and upload sample data programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#).

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.Model;  
using Amazon.Runtime;  
  
namespace com.amazonaws.codesamples  
{  
    class LowLevelBatchWrite  
    {  
        private static string table1Name = "Forum";  
        private static string table2Name = "Thread";  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
  
        static void Main(string[] args)  
        {  
            try  
            {  
                TestBatchWrite();  
            }  
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }  
            catch (Exception e) { Console.WriteLine(e.Message); }  
  
            Console.WriteLine("To continue, press Enter");  
            Console.ReadLine();  
        }  
  
        private static void TestBatchWrite()  
        {  
            var request = new BatchWriteItemRequest  
            {  
                ReturnConsumedCapacity = "TOTAL",  
                RequestItems = new Dictionary<string, List<WriteRequest>>  
            }  
        }  
    }  
}
```

```

    {
        table1Name, new List<WriteRequest>
    {
        new WriteRequest
        {
            PutRequest = new PutRequest
            {
                Item = new Dictionary<string, AttributeValue>
                {
                    { "Name", new AttributeValue {
                        S = "S3 forum"
                    } },
                    { "Threads", new AttributeValue {
                        N = "0"
                    } }
                }
            }
        }
    },
    {
        table2Name, new List<WriteRequest>
    {
        new WriteRequest
        {
            PutRequest = new PutRequest
            {
                Item = new Dictionary<string, AttributeValue>
                {
                    { "ForumName", new AttributeValue {
                        S = "S3 forum"
                    } },
                    { "Subject", new AttributeValue {
                        S = "My sample question"
                    } },
                    { "Message", new AttributeValue {
                        S = "Message Text."
                    } },
                    { "KeywordTags", new AttributeValue {
                        SS = new List<string> { "S3", "Bucket" }
                    } }
                }
            }
        },
        new WriteRequest
        {
            // For the operation to delete an item, if you provide a
            // primary key value
            // that does not exist in the table, there is no error, it is
            // just a no-op.
            DeleteRequest = new DeleteRequest
            {
                Key = new Dictionary<string, AttributeValue>()
                {
                    { "ForumName", new AttributeValue {
                        S = "Some partition key value"
                    } },
                    { "Subject", new AttributeValue {
                        S = "Some sort key value"
                    } }
                }
            }
        }
    }
}
}

```

```

    };

    CallBatchWriteTillCompletion(request);
}

private static void CallBatchWriteTillCompletion(BatchWriteItemRequest request)
{
    BatchWriteItemResponse response;

    int callCount = 0;
    do
    {
        Console.WriteLine("Making request");
        response = client.BatchWriteItem(request);
        callCount++;

        // Check the response.

        var tableConsumedCapacities = response.ConsumedCapacity;
        var unprocessed = response.UnprocessedItems;

        Console.WriteLine("Per-table consumed capacity");
        foreach (var tableConsumedCapacity in tableConsumedCapacities)
        {
            Console.WriteLine("{0} - {1}", tableConsumedCapacity.TableName,
tableConsumedCapacity.CapacityUnits);
        }

        Console.WriteLine("Unprocessed");
        foreach (var unp in unprocessed)
        {
            Console.WriteLine("{0} - {1}", unp.Key, unp.Value.Count);
        }
        Console.WriteLine();

        // For the next iteration, the request will have unprocessed items.
        request.RequestItems = unprocessed;
    } while (response.UnprocessedItems.Count > 0);

    Console.WriteLine("Total # of batch write API calls made: {0}", callCount);
}
}

```

Example: Batch Get Operation Using the AWS SDK for .NET Low-Level API

The following C# code example uses the `BatchGetItem` method to retrieve multiple items from the `Forum` and the `Thread` tables in Amazon DynamoDB. The `BatchGetItemRequest` specifies the table names and a list of primary keys for each table. The example processes the response by printing the items retrieved.

If you followed the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you already have these tables created with sample data. You can also create these sample tables and upload sample data programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#).

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

Example

```


/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelBatchGet
    {
        private static string table1Name = "Forum";
        private static string table2Name = "Thread";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                RetrieveMultipleItemsBatchGet();

                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void RetrieveMultipleItemsBatchGet()
        {
            var request = new BatchGetItemRequest
            {
                RequestItems = new Dictionary<string, KeysAndAttributes>()
            };
            {
                table1Name,
                new KeysAndAttributes
                {
                    Keys = new List<Dictionary<string, AttributeValue>>()
                    {
                        new Dictionary<string, AttributeValue>()
                        {
                            { "Name", new AttributeValue {
                                S = "Amazon DynamoDB"
                            } },
                            { "Name", new AttributeValue {
                                S = "Amazon S3"
                            } }
                        }
                    },
                    new Dictionary<string, AttributeValue>()
                    {
                        { "Name", new AttributeValue {
                            S = "Amazon S3"
                        } }
                    }
                }
            }, 
        }
    }
}


```

```

        table2Name,
        new KeysAndAttributes
    {
        Keys = new List<Dictionary<string, AttributeValue>>()
        {
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon DynamoDB"
                } },
                { "Subject", new AttributeValue {
                    S = "DynamoDB Thread 1"
                } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon DynamoDB"
                } },
                { "Subject", new AttributeValue {
                    S = "DynamoDB Thread 2"
                } }
            },
            new Dictionary<string, AttributeValue>()
            {
                { "ForumName", new AttributeValue {
                    S = "Amazon S3"
                } },
                { "Subject", new AttributeValue {
                    S = "S3 Thread 1"
                } }
            }
        }
    };
}

BatchGetItemResponse response;
do
{
    Console.WriteLine("Making request");
    response = client.BatchGetItem(request);

    // Check the response.
    var responses = response.Responses; // Attribute list in the response.

    foreach (var tableResponse in responses)
    {
        var tableResults = tableResponse.Value;
        Console.WriteLine("Items retrieved from table {0}", tableResponse.Key);
        foreach (var item1 in tableResults)
        {
            PrintItem(item1);
        }
    }

    // Any unprocessed keys? could happen if you exceed ProvisionedThroughput
    or some other error.
    Dictionary<string, KeysAndAttributes> unprocessedKeys =
    response.UnprocessedKeys;
    foreach (var unprocessedTableKeys in unprocessedKeys)
    {
        // Print table name.
        Console.WriteLine(unprocessedTableKeys.Key);
        // Print unprocessed primary keys.
    }
}

```

```

        foreach (var key in unprocessedTableKeys.Value.Keys)
        {
            PrintItem(key);
        }
    }

    request.RequestItems = unprocessedKeys;
} while (response.UnprocessedKeys.Count > 0);
}

private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
{
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
    {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;

        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[ " + value.S + " ]") +
            (value.N == null ? "" : "N=[ " + value.N + " ]") +
            (value.SS == null ? "" : "SS=[ " + string.Join(", ", value.SS.ToArray()) +
            " ]" ) +
            (value.NS == null ? "" : "NS=[ " + string.Join(", ", value.NS.ToArray()) +
            " ]"));
    }
    Console.WriteLine("*****");
}
}

```

Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API

The following C# code example illustrates the handling of binary type attributes. The example adds an item to the `Reply` table. The item includes a binary type attribute (`ExtendedMessage`) that stores compressed data. The example then retrieves the item and prints all the attribute values. For illustration, the example uses the `GZipStream` class to compress a sample stream and assigns it to the `ExtendedMessage` attribute, and decompresses it when printing the attribute value.

If you followed the steps in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you already have the `Reply` table created. You can also create these sample tables programmatically. For more information, see [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#).

For step-by-step instructions for testing the following example, see [.NET Code Examples \(p. 332\)](#).

Example

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the

```

```

/*
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelItemBinaryExample
    {
        private static string tableName = "Reply";
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            // Reply table primary key.
            string replyIdPartitionKey = "Amazon DynamoDB#DynamoDB Thread 1";
            string replyDateTimeSortKey = Convert.ToString(DateTime.UtcNow);

            try
            {
                CreateItem(replyIdPartitionKey, replyDateTimeSortKey);
                RetrieveItem(replyIdPartitionKey, replyDateTimeSortKey);
                // Delete item.
                DeleteItem(replyIdPartitionKey, replyDateTimeSortKey);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void CreateItem(string partitionKey, string sortKey)
        {
            MemoryStream compressedMessage = ToGzipMemoryStream("Some long extended message
to compress.");
            var request = new PutItemRequest
            {
                TableName = tableName,
                Item = new Dictionary<string, AttributeValue>()
            };
            {
                { "Id", new AttributeValue {
                    S = partitionKey
                }},
                { "ReplyDateTime", new AttributeValue {
                    S = sortKey
                }},
                { "Subject", new AttributeValue {
                    S = "Binary type "
                }},
                { "Message", new AttributeValue {
                    S = "Some message about the binary type"
                }},
                { "ExtendedMessage", new AttributeValue {
                    B = compressedMessage
                }}
            };
            client.PutItem(request);
        }
    }
}

```

```

private static void RetrieveItem(string partitionKey, string sortKey)
{
    var request = new GetItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            S = partitionKey
        } },
        { "ReplyDateTime", new AttributeValue {
            S = sortKey
        } }
    },
        ConsistentRead = true
    };
    var response = client.GetItem(request);

    // Check the response.
    var attributeList = response.Item; // attribute list in the response.
    Console.WriteLine("\nPrinting item after retrieving it .....");

    PrintItem(attributeList);
}

private static void DeleteItem(string partitionKey, string sortKey)
{
    var request = new DeleteItemRequest
    {
        TableName = tableName,
        Key = new Dictionary<string, AttributeValue>()
    {
        { "Id", new AttributeValue {
            S = partitionKey
        } },
        { "ReplyDateTime", new AttributeValue {
            S = sortKey
        } }
    }
};
    var response = client.DeleteItem(request);
}

private static void PrintItem(Dictionary<string, AttributeValue> attributeList)
{
    foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
    {
        string attributeName = kvp.Key;
        AttributeValue value = kvp.Value;

        Console.WriteLine(
            attributeName + " " +
            (value.S == null ? "" : "S=[[" + value.S + "]]") +
            (value.N == null ? "" : "N=[[" + value.N + "]]") +
            (value.SS == null ? "" : "SS=[[" + string.Join(",", value.SS.ToArray()) +
            "]]") +
            (value.NS == null ? "" : "NS=[[" + string.Join(",", value.NS.ToArray()) +
            "]]") +
            (value.B == null ? "" : "B=[[" + FromGzipMemoryStream(value.B) + "]]");
        }
        Console.WriteLine("*****");
    }
}

private static MemoryStream ToGzipMemoryStream(string value)
{

```

```
        MemoryStream output = new MemoryStream();
        using (GZipStream zipStream = new GZipStream(output, CompressionMode.Compress,
true))
        {
            using (StreamWriter writer = new StreamWriter(zipStream))
            {
                writer.Write(value);
            }
            return output;
        }

        private static string FromGzipMemoryStream(MemoryStream stream)
        {
            using (GZipStream zipStream = new GZipStream(stream,
CompressionMode.Decompress))
            using (StreamReader reader = new StreamReader(zipStream))
            {
                return reader.ReadToEnd();
            }
        }
    }
}
```

Working with Queries in DynamoDB

The `Query` operation in Amazon DynamoDB finds items based on primary key values.

You must provide the name of the partition key attribute and a single value for that attribute. `Query` returns all items with that partition key value. Optionally, you can provide a sort key attribute and use a comparison operator to refine the search results.

Topics

- [Key Condition Expression \(p. 458\)](#)
- [Filter Expressions for Query \(p. 460\)](#)
- [Limiting the Number of Items in the Result Set \(p. 461\)](#)
- [Paginating Table Query Results \(p. 461\)](#)
- [Counting the Items in the Results \(p. 462\)](#)
- [Capacity Units Consumed by Query \(p. 463\)](#)
- [Read Consistency for Query \(p. 463\)](#)
- [Querying Tables and Indexes: Java \(p. 463\)](#)
- [Querying Tables and Indexes: .NET \(p. 469\)](#)

Key Condition Expression

To specify the search criteria, you use a *key condition expression*—a string that determines the items to be read from the table or index.

You must specify the partition key name and value as an equality condition.

You can optionally provide a second condition for the sort key (if present). The sort key condition must use one of the following comparison operators:

- `a = b` — true if the attribute `a` is equal to the value `b`
- `a < b` — true if `a` is less than `b`
- `a <= b` — true if `a` is less than or equal to `b`

- $a > b$ — true if a is greater than b
- $a \geq b$ — true if a is greater than or equal to b
- $a \text{ BETWEEN } b \text{ AND } c$ — true if a is greater than or equal to b , and less than or equal to c .

The following function is also supported:

- `begins_with (a, substr)`— true if the value of attribute a begins with a particular substring.

The following AWS Command Line Interface (AWS CLI) examples demonstrate the use of key condition expressions. These expressions use placeholders (such as `:name` and `:sub`) instead of actual values. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#) and [Expression Attribute Values \(p. 391\)](#).

Example

Query the `Thread` table for a particular `ForumName` (partition key). All of the items with that `ForumName` value are read by the query because the sort key (`Subject`) is not included in `KeyConditionExpression`.

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name" \
--expression-attribute-values '{"":name":{"S":"Amazon DynamoDB"}}'
```

Example

Query the `Thread` table for a particular `ForumName` (partition key), but this time return only the items with a given `Subject` (sort key).

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name and Subject = :sub" \
--expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
  ":name":{"S":"Amazon DynamoDB"}, 
  ":sub":{"S":"DynamoDB Thread 1"}
}
```

Example

Query the `Reply` table for a particular `Id` (partition key), but return only those items whose `ReplyDateTime` (sort key) begins with certain characters.

```
aws dynamodb query \
--table-name Reply \
--key-condition-expression "Id = :id and begins_with(ReplyDateTime, :dt)" \
--expression-attribute-values file://values.json
```

The arguments for --expression-attribute-values are stored in the values.json file.

```
{  
    ":id": {"S": "Amazon DynamoDB#DynamoDB Thread 1"},  
    ":dt": {"S": "2015-09"}  
}
```

You can use any attribute name in a key condition expression, provided that the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. In addition, the attribute name must not be a DynamoDB reserved word. (For a complete list of these, see [Reserved Words in DynamoDB \(p. 1026\)](#).) If an attribute name does not meet these requirements, you must define an expression attribute name as a placeholder. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

For items with a given partition key value, DynamoDB stores these items close together, in sorted order by sort key value. In a `Query` operation, DynamoDB retrieves the items in sorted order, and then processes the items using `KeyConditionExpression` and any `FilterExpression` that might be present. Only then are the `Query` results sent back to the client.

A `Query` operation always returns a result set. If no matching items are found, the result set is empty.

`Query` results are always sorted by the sort key value. If the data type of the sort key is `Number`, the results are returned in numeric order. Otherwise, the results are returned in order of UTF-8 bytes. By default, the sort order is ascending. To reverse the order, set the `ScanIndexForward` parameter to `false`.

A single `Query` operation can retrieve a maximum of 1 MB of data. This limit applies before any `FilterExpression` is applied to the results. If `LastEvaluatedKey` is present in the response and is non-null, you must paginate the result set (see [Paginating Table Query Results \(p. 461\)](#)).

Filter Expressions for Query

If you need to further refine the `Query` results, you can optionally provide a filter expression. A *filter expression* determines which items within the `Query` results should be returned to you. All of the other results are discarded.

A filter expression is applied after a `Query` finishes, but before the results are returned. Therefore, a `Query` consumes the same amount of read capacity, regardless of whether a filter expression is present.

A `Query` operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

A filter expression cannot contain partition key or sort key attributes. You need to specify those attributes in the key condition expression, not the filter expression.

The syntax for a filter expression is identical to that of a condition expression. Filter expressions can use the same comparators, functions, and logical operators as a condition expression. For more information, [Condition Expressions \(p. 392\)](#).

Example

The following AWS CLI example queries the `Thread` table for a particular `ForumName` (partition key) and `Subject` (sort key). Of the items that are found, only the most popular discussion threads are returned—in other words, only those threads with more than a certain number of `Views`.

```
aws dynamodb query \
```

```
--table-name Thread \
--key-condition-expression "ForumName = :fn and Subject = :sub" \
--filter-expression "#v >= :num" \
--expression-attribute-names '{"#v": "Views"}' \
--expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the `values.json` file.

```
{
  ":fn": {"S": "Amazon DynamoDB"},
  ":sub": {"S": "DynamoDB Thread 1"},
  ":num": {"N": "3"}
}
```

Note that `Views` is a reserved word in DynamoDB (see [Reserved Words in DynamoDB \(p. 1026\)](#)), so this example uses `#v` as a placeholder. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

Note

A filter expression removes items from the `Query` result set. If possible, avoid using `Query` where you expect to retrieve a large number of items but also need to discard most of those items.

Limiting the Number of Items in the Result Set

The `Query` operation allows you to limit the number of items that it reads. To do this, set the `Limit` parameter to the maximum number of items that you want.

For example, suppose that you `Query` a table, with a `Limit` value of 6, and without a filter expression. The `Query` result contains the first six items from the table that match the key condition expression from the request.

Now suppose that you add a filter expression to the `Query`. In this case, DynamoDB reads up to six items, and then returns only those that match the filter expression. The final `Query` result contains six items or fewer, even if more items would have matched the filter expression if DynamoDB had kept reading more items.

Paginating Table Query Results

DynamoDB *paginates* the results from `Query` operations. With pagination, the `Query` results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single `Query` only returns a result set that fits within the 1 MB size limit. To determine whether there are more results, and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level `Query` result:
 - If the result contains a `LastEvaluatedKey` element, proceed to step 2.
 - If there is *not* a `LastEvaluatedKey` in the result, there are no more items to be retrieved.
2. Construct a new `Query` request, with the same parameters as the previous one. However, this time, take the `LastEvaluatedKey` value from step 1 and use it as the `ExclusiveStartKey` parameter in the new `Query` request.
3. Run the new `Query` request.
4. Go to step 1.

In other words, the `LastEvaluatedKey` from a `Query` response should be used as the `ExclusiveStartKey` for the next `Query` request. If there is not a `LastEvaluatedKey` element in a `Query` response, then you have retrieved the final page of results. If `LastEvaluatedKey` is not empty, it does not necessarily mean that there is more data in the result set. The only way to know when you have reached the end of the result set is when `LastEvaluatedKey` is empty.

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level `Query` requests to DynamoDB repeatedly, until `LastEvaluatedKey` is no longer present in the results. Consider the following AWS CLI example that retrieves movie titles from a particular year.

```
aws dynamodb query --table-name Movies \
--projection-expression "title" \
--key-condition-expression "#y = :yyyy" \
--expression-attribute-names '{"#y":"year"}' \
--expression-attribute-values '{":yyyy":{"N":"1993"}}' \
--page-size 5 \
--debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 11:13:15,603 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":5,"Items":[{"title":{"S":"A Bronx Tale"}}, {"title":{"S":"A Perfect World"}}, {"title":{"S":"Addams Family Values"}}, {"title":{"S":"Alive"}}, {"title":{"S":"Benny & Joon"}}, "LastEvaluatedKey":{"year":{"N":"1993"}, "title":{"S":"Benny & Joon"}}, "ScannedCount":5}'
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another `Query` request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 11:13:16,291 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":1,"Items":[{"title":{"S":"What\'s Eating Gilbert Grape"}]}, "ScannedCount":1}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

Note

The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating `Query` results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.

For code examples in various programming languages, see the [Amazon DynamoDB Getting Started Guide](#) and the AWS SDK documentation for your language.

For more information about querying with DynamoDB, see [Working with Queries in DynamoDB \(p. 458\)](#).

Counting the Items in the Results

In addition to the items that match your criteria, the `Query` response contains the following elements:

- `ScannedCount` — The number of items that matched the key condition expression *before* a filter expression (if present) was applied.
- `Count` — The number of items that remain *after* a filter expression (if present) was applied.

Note

If you don't use a filter expression, `ScannedCount` and `Count` have the same value.

If the size of the `Query` result set is larger than 1 MB, `ScannedCount` and `Count` represent only a partial count of the total items. You need to perform multiple `Query` operations to retrieve all the results (see [Paginating Table Query Results \(p. 461\)](#)).

Each `Query` response contains the `ScannedCount` and `Count` for the items that were processed by that particular `Query` request. To obtain grand totals for all of the `Query` requests, you could keep a running tally of both `ScannedCount` and `Count`.

Capacity Units Consumed by Query

You can `Query` any table or secondary index, provided that it has a composite primary key (partition key and sort key). `Query` operations consume read capacity units, as follows.

| If you <code>Query</code> a... | DynamoDB consumes read capacity units from... |
|--------------------------------|---|
| Table | The table's provisioned read capacity. |
| Global secondary index | The index's provisioned read capacity. |
| Local secondary index | The base table's provisioned read capacity. |

By default, a `Query` operation does not return any data on how much read capacity it consumes. However, you can specify the `ReturnConsumedCapacity` parameter in a `Query` request to obtain this information. The following are the valid settings for `ReturnConsumedCapacity`:

- `NONE` — No consumed capacity data is returned. (This is the default.)
- `TOTAL` — The response includes the aggregate number of read capacity units consumed.
- `INDEXES` — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on item size, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression.

Read Consistency for Query

A `Query` operation performs eventually consistent reads, by default. This means that the `Query` results might not reflect changes due to recently completed `PutItem` or `UpdateItem` operations. For more information, see [Read Consistency \(p. 16\)](#).

If you require strongly consistent reads, set the `ConsistentRead` parameter to `true` in the `Query` request.

Querying Tables and Indexes: Java

The `Query` operation enables you to query a table or a secondary index in Amazon DynamoDB. You must provide a partition key value and an equality condition. If the table or index has a sort key, you can refine the results by providing a sort key value and a condition.

Note

The AWS SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

The following are the steps to retrieve an item using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the table you want to work with.
3. Call the `query` method of the `Table` instance. You must specify the partition key value of the items that you want to retrieve, along with any optional query parameters.

The response includes an `ItemCollection` object that provides all items returned by the query.

The following Java code example demonstrates the preceding tasks. The example assumes that you have a `Reply` table that stores replies for forum threads. For more information, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

```
Reply ( Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the `Id` attribute of the `Reply` table is composed of both the forum name and forum subject. `Id` (partition key) and `ReplyDateTime` (sort key) make up the composite primary key for the table.

The following query retrieves all replies for a specific thread subject. The query requires both the table name and the `Subject` value.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
    .withRegion(Regions.US_WEST_2).build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("Reply");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Id = :v_id")
    .WithValueMap(new ValueMap()
        .withString(":v_id", "Amazon DynamoDB#DynamoDB Thread 1"));

ItemCollection<QueryOutcome> items = table.query(spec);

Iterator<Item> iterator = items.iterator();
Item item = null;
while (iterator.hasNext()) {
    item = iterator.next();
    System.out.println(item.toJSONPretty());
}
```

Specifying Optional Parameters

The `query` method supports several optional parameters. For example, you can optionally narrow the results from the preceding query to return replies in the past two weeks by specifying a condition. The condition is called a sort key condition, because DynamoDB evaluates the query condition that you specify against the sort key of the primary key. You can specify other optional parameters to retrieve only a specific list of attributes from items in the query result.

The following Java code example retrieves forum thread replies posted in the past 15 days. The example specifies optional parameters using the following:

- A `KeyConditionExpression` to retrieve the replies from a specific discussion forum (partition key) and, within that set of items, replies that were posted within the last 15 days (sort key).
- A `FilterExpression` to return only the replies from a specific user. The filter is applied after the query is processed, but before the results are returned to the user.
- A `ValueMap` to define the actual values for the `KeyConditionExpression` placeholders.
- A `ConsistentRead` setting of true, to request a strongly consistent read.

This example uses a `QuerySpec` object that gives access to all of the low-level `Query` input parameters.

Example

```
Table table = dynamoDB.getTable("Reply");

long twoWeeksAgoMilli = (new Date()).getTime() - (15L*24L*60L*60L*1000L);
Date twoWeeksAgo = new Date();
twoWeeksAgo.setTime(twoWeeksAgoMilli);
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
String twoWeeksAgoStr = df.format(twoWeeksAgo);

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Id = :v_id and ReplyDateTime > :v_reply_dt_tm")
    .withFilterExpression("PostedBy = :v_posted_by")
    .withValueMap(new ValueMap()
        .withString(":v_id", "Amazon DynamoDB#DynamoDB Thread 1")
        .withString(":v_reply_dt_tm", twoWeeksAgoStr)
        .withString(":v_posted_by", "User B"))
    .withConsistentRead(true);

ItemCollection<QueryOutcome> items = table.query(spec);

Iterator<Item> iterator = items.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next().toJSONPretty());
}
```

You can also optionally limit the number of items per page by using the `withMaxPageSize` method. When you call the `query` method, you get an `ItemCollection` that contains the resulting items. You can then step through the results, processing one page at a time, until there are no more pages.

The following Java code example modifies the query specification shown previously. This time, the query spec uses the `withMaxPageSize` method. The `Page` class provides an iterator that allows the code to process the items on each page.

Example

```
spec.withMaxPageSize(10);

ItemCollection<QueryOutcome> items = table.query(spec);

// Process each page of results
int pageNum = 0;
for (Page<Item, QueryOutcome> page : items.pages()) {

    System.out.println("\nPage: " + ++pageNum);

    // Process each item on the current page
    Iterator<Item> item = page.iterator();
    while (item.hasNext()) {
```

```
        System.out.println(item.next().toJSONPretty());
    }
}
```

Example - Query Using Java

The following tables store information about a collection of forums. For more information, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

Example

```
Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( Id, ReplyDateTime, Message, PostedBy, ... )
```

In this Java code example, you execute variations of finding replies for a thread "DynamoDB Thread 1" in forum "DynamoDB".

- Find replies for a thread.
- Find replies for a thread, specifying a limit on the number of items per page of results. If the number of items in the result set exceeds the page size, you get only the first page of results. This coding pattern ensures that your code processes all the pages in the query result.
- Find replies in the last 15 days.
- Find replies in a specific date range.

The preceding two queries show how you can specify sort key conditions to narrow the query results and use other optional query parameters.

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;
```

```

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.Page;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;

public class DocumentAPIQuery {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static String tableName = "Reply";

    public static void main(String[] args) throws Exception {

        String forumName = "Amazon DynamoDB";
        String threadSubject = "DynamoDB Thread 1";

        findRepliesForAThread(forumName, threadSubject);
        findRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);
        findRepliesInLast15DaysWithConfig(forumName, threadSubject);
        findRepliesPostedWithinTimePeriod(forumName, threadSubject);
        findRepliesUsingAFilterExpression(forumName, threadSubject);
    }

    private static void findRepliesForAThread(String forumName, String threadSubject) {

        Table table = dynamoDB.getTable(tableName);

        String replyId = forumName + "#" + threadSubject;

        QuerySpec spec = new QuerySpec().withKeyConditionExpression("Id = :v_id")
            .WithValueMap(new ValueMap().withString(":v_id", replyId));

        ItemCollection<QueryOutcome> items = table.query(spec);

        System.out.println("\nfindRepliesForAThread results:");

        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }

    private static void findRepliesForAThreadSpecifyOptionalLimit(String forumName, String
threadSubject) {

        Table table = dynamoDB.getTable(tableName);

        String replyId = forumName + "#" + threadSubject;

        QuerySpec spec = new QuerySpec().withKeyConditionExpression("Id = :v_id")
            .WithValueMap(new ValueMap().withString(":v_id", replyId)).withMaxPageSize(1);

        ItemCollection<QueryOutcome> items = table.query(spec);

        System.out.println("\nfindRepliesForAThreadSpecifyOptionalLimit results:");

        // Process each page of results
    }
}

```

```

        int pageNum = 0;
        for (Page<Item, QueryOutcome> page : items.pages()) {

            System.out.println("\nPage: " + ++pageNum);

            // Process each item on the current page
            Iterator<Item> item = page.iterator();
            while (item.hasNext()) {
                System.out.println(item.next().toJSONPretty());
            }
        }

        private static void findRepliesInLast15DaysWithConfig(String forumName, String
threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    long twoWeeksAgoMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L * 1000L);
    Date twoWeeksAgo = new Date();
    twoWeeksAgo.setTime(twoWeeksAgoMilli);
    SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");
    String twoWeeksAgoStr = df.format(twoWeeksAgo);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withProjectionExpression("Message, ReplyDateTime,
PostedBy")
        .withKeyConditionExpression("Id = :v_id and ReplyDateTime <= :v_reply_dt_tm")
        .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_reply_dt_tm", twoWeeksAgoStr));

    ItemCollection<QueryOutcome> items = table.query(spec);

    System.out.println("\nfindRepliesInLast15DaysWithConfig results:");
    Iterator<Item> iterator = items.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }
}

private static void findRepliesPostedWithinTimePeriod(String forumName, String
threadSubject) {

    Table table = dynamoDB.getTable(tableName);

    long startDateMilli = (new Date()).getTime() - (15L * 24L * 60L * 60L * 1000L);
    long endDateMilli = (new Date()).getTime() - (5L * 24L * 60L * 60L * 1000L);
    java.text.SimpleDateFormat df = new java.text.SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");
    String startDate = df.format(startDateMilli);
    String endDate = df.format(endDateMilli);

    String replyId = forumName + "#" + threadSubject;

    QuerySpec spec = new QuerySpec().withProjectionExpression("Message, ReplyDateTime,
PostedBy")
        .withKeyConditionExpression("Id = :v_id and ReplyDateTime between :v_start_dt
and :v_end_dt")
        .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_start_dt", startDate)
        .withString(":v_end_dt", endDate));

    ItemCollection<QueryOutcome> items = table.query(spec);
}

```

```

        System.out.println("\nfindRepliesPostedWithinTimePeriod results:");
        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }

    private static void findRepliesUsingAFilterExpression(String forumName, String
threadSubject) {

        Table table = dynamoDB.getTable(tableName);

        String replyId = forumName + "#" + threadSubject;

        QuerySpec spec = new QuerySpec().withProjectionExpression("Message, ReplyDateTime,
PostedBy")
            .withKeyConditionExpression("Id = :v_id").withFilterExpression("PostedBy
= :v_postedby")
            .withValueMap(new ValueMap().withString(":v_id",
replyId).withString(":v_postedby", "User B"));

        ItemCollection<QueryOutcome> items = table.query(spec);

        System.out.println("\nfindRepliesUsingAFilterExpression results:");
        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}

```

Querying Tables and Indexes: .NET

The `Query` operation enables you to query a table or a secondary index in Amazon DynamoDB. You must provide a partition key value and an equality condition. If the table or index has a sort key, you can refine the results by providing a sort key value and a condition.

The following are the steps to query a table using the low-level AWS SDK for .NET API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class and provide query operation parameters.
3. Execute the `Query` method and provide the `QueryRequest` object that you created in the preceding step.

The response includes the `QueryResult` object that provides all items returned by the query.

The following C# code example demonstrates the preceding tasks. The code assumes that you have a `Reply` table that stores replies for forum threads. For more information, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

Example

```
Reply Id, ReplyDateTime, ... )
```

Each forum thread has a unique ID and can have zero or more replies. Therefore, the primary key is composed of both the `Id` (partition key) and `ReplyDateTime` (sort key).

The following query retrieves all replies for a specific thread subject. The query requires both the table name and the Subject value.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

var request = new QueryRequest
{
    TableName = "Reply",
    KeyConditionExpression = "Id = :v_Id",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {"":":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 1" }}}
};

var response = client.Query(request);

foreach (Dictionary<string, AttributeValue> item in response.Items)
{
    // Process the result.
    PrintItem(item);
}
```

Specifying Optional Parameters

The `Query` method supports several optional parameters. For example, you can optionally narrow the query result in the preceding query to return replies in the past two weeks by specifying a condition. The condition is called a *sort key condition*, because DynamoDB evaluates the query condition that you specify against the sort key of the primary key. You can specify other optional parameters to retrieve only a specific list of attributes from items in the query result. For more information, see [Query](#).

The following C# code example retrieves forum thread replies posted in the past 15 days. The example specifies the following optional parameters:

- A `KeyConditionExpression` to retrieve only the replies in the past 15 days.
- A `ProjectionExpression` parameter to specify a list of attributes to retrieve for items in the query result.
- A `ConsistentRead` parameter to perform a strongly consistent read.

Example

```
DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
string twoWeeksAgoString = twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601DateFormat);

var request = new QueryRequest
{
    TableName = "Reply",
    KeyConditionExpression = "Id = :v_Id and ReplyDateTime > :v_twoWeeksAgo",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {"":":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 2" }},
        {"":":v_twoWeeksAgo", new AttributeValue { S = twoWeeksAgoString }}}
},
    ProjectionExpression = "Subject, ReplyDateTime, PostedBy",
    ConsistentRead = true
};

var response = client.Query(request);

foreach (Dictionary<string, AttributeValue> item in response.Items)
{
```

```

    // Process the result.
    PrintItem(item);
}

```

You can also optionally limit the page size, or the number of items per page, by adding the optional `Limit` parameter. Each time you execute the `Query` method, you get one page of results that has the specified number of items. To fetch the next page, you execute the `Query` method again by providing the primary key value of the last item in the previous page so that the method can return the next set of items. You provide this information in the request by setting the `ExclusiveStartKey` property. Initially, this property can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following C# example queries the `Reply` table. In the request, it specifies the `Limit` and `ExclusiveStartKey` optional parameters. The do/while loop continues to scan one page at a time until the `LastEvaluatedKey` returns a null value.

Example

```

Dictionary<string,AttributeValue> lastKeyEvaluated = null;

do
{
    var request = new QueryRequest
    {
        TableName = "Reply",
        KeyConditionExpression = "Id = :v_Id",
        ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
            {":v_Id", new AttributeValue { S = "Amazon DynamoDB#DynamoDB Thread 2" }},
        },

        // Optional parameters.
        Limit = 1,
        ExclusiveStartKey = lastKeyEvaluated
    };

    var response = client.Query(request);

    // Process the query result.
    foreach (Dictionary<string, AttributeValue> item in response.Items)
    {
        PrintItem(item);
    }

    lastKeyEvaluated = response.LastEvaluatedKey;
} while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

```

Example - Querying Using the AWS SDK for .NET

The following tables store information about a collection of forums. For more information, see [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

Example

```

Forum ( Name, ... )
Thread ( ForumName, Subject, Message, LastPostedBy, LastPostDateTime, ... )
Reply ( Id, ReplyDateTime, Message, PostedBy, ... )

```

In this example, you execute variations of "Find replies for a thread "DynamoDB Thread 1" in forum "DynamoDB".

- Find replies for a thread.
- Find replies for a thread. Specify the `Limit` query parameter to set page size.

This function illustrates the use of pagination to process multipage result. DynamoDB has a page size limit and if your result exceeds the page size, you get only the first page of results. This coding pattern ensures your code processes all the pages in the query result.

- Find replies in the last 15 days.
- Find replies in a specific date range.

The preceding two queries show how you can specify sort key conditions to narrow query results and use other optional query parameters.

For step-by-step instructions for testing the following example, see [.NET Code Examples \(p. 332\)](#).

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.Util;

namespace com.amazonaws.codesamples
{
    class LowLevelQuery
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                // Query a specific forum and thread.
                string forumName = "Amazon DynamoDB";
                string threadSubject = "DynamoDB Thread 1";

                FindRepliesForAThread(forumName, threadSubject);
                FindRepliesForAThreadSpecifyOptionalLimit(forumName, threadSubject);
                FindRepliesInLast15DaysWithConfig(forumName, threadSubject);
                FindRepliesPostedWithinTimePeriod(forumName, threadSubject);

                Console.WriteLine("Example complete. To continue, press Enter");
                Console.ReadLine();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message);
                Console.ReadLine(); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message);
                Console.ReadLine(); }
            catch (Exception e) { Console.WriteLine(e.Message); Console.ReadLine(); }
        }
    }
}
```

```

        }

        private static void FindRepliesPostedWithinTimePeriod(string forumName, string
threadSubject)
        {
            Console.WriteLine("**** Executing FindRepliesPostedWithinTimePeriod() ***");
            string replyId = forumName + "#" + threadSubject;
            // You must provide date value based on your test data.
            DateTime startDate = DateTime.UtcNow - TimeSpan.FromDays(21);
            string start = startDate.ToString(AWSSDKUtils.ISO8601DateFormat);

            // You provide date value based on your test data.
            DateTime endDate = DateTime.UtcNow - TimeSpan.FromDays(5);
            string end = endDate.ToString(AWSSDKUtils.ISO8601DateFormat);

            var request = new QueryRequest
            {
                TableName = "Reply",
                ReturnConsumedCapacity = "TOTAL",
                KeyConditionExpression = "Id = :v_replyId and ReplyDateTime
between :v_start and :v_end",
                ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
                    {"":v_replyId", new AttributeValue {
                        S = replyId
                    }},
                    {"":v_start", new AttributeValue {
                        S = start
                    }},
                    {"":v_end", new AttributeValue {
                        S = end
                    }}
                };
            };

            var response = client.Query(request);

            Console.WriteLine("\nNo. of reads used (by query in
FindRepliesPostedWithinTimePeriod) {0}",
                response.ConsumedCapacity.CapacityUnits);
            foreach (Dictionary<string, AttributeValue> item
                in response.Items)
            {
                PrintItem(item);
            }
            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void FindRepliesInLast15DaysWithConfig(string forumName, string
threadSubject)
        {
            Console.WriteLine("**** Executing FindRepliesInLast15DaysWithConfig() ***");
            string replyId = forumName + "#" + threadSubject;

            DateTime twoWeeksAgoDate = DateTime.UtcNow - TimeSpan.FromDays(15);
            string twoWeeksAgoString =
                twoWeeksAgoDate.ToString(AWSSDKUtils.ISO8601DateFormat);

            var request = new QueryRequest
            {
                TableName = "Reply",
                ReturnConsumedCapacity = "TOTAL",
                KeyConditionExpression = "Id = :v_replyId and ReplyDateTime > :v_interval",
                ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
                    {"":v_replyId", new AttributeValue {
                        S = replyId
                    }},
                    {"":v_interval", new AttributeValue {
                        S = twoWeeksAgoString
                    }}
                };
            };
        }
    }
}

```

```

        },
        {"<v_interval", new AttributeValue {
            S = twoWeeksAgoString
        })
    },
    // Optional parameter.
    ProjectionExpression = "Id, ReplyDateTime, PostedBy",
    // Optional parameter.
    ConsistentRead = true
};

var response = client.Query(request);

Console.WriteLine("No. of reads used (by query in
FindRepliesInLast15DaysWithConfig) {0}",
    response.ConsumedCapacity.CapacityUnits);
foreach (Dictionary<string, AttributeValue> item
    in response.Items)
{
    PrintItem(item);
}
Console.WriteLine("To continue, press Enter");
Console.ReadLine();
}

private static void FindRepliesForAThreadSpecifyOptionalLimit(string forumName,
string threadSubject)
{
    Console.WriteLine("**** Executing FindRepliesForAThreadSpecifyOptionalLimit()
****");
    string replyId = forumName + "#" + threadSubject;

    Dictionary<string, AttributeValue> lastKeyEvaluated = null;
    do
    {
        var request = new QueryRequest
        {
            TableName = "Reply",
            ReturnConsumedCapacity = "TOTAL",
            KeyConditionExpression = "Id = :v_replyId",
            ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
                {"<v_replyId", new AttributeValue {
                    S = replyId
                }}
            },
            Limit = 2, // The Reply table has only a few sample items. So the page
size is smaller.
            ExclusiveStartKey = lastKeyEvaluated
        };

        var response = client.Query(request);

        Console.WriteLine("No. of reads used (by query in
FindRepliesForAThreadSpecifyLimit) {0}\n",
            response.ConsumedCapacity.CapacityUnits);
        foreach (Dictionary<string, AttributeValue> item
            in response.Items)
        {
            PrintItem(item);
        }
        lastKeyEvaluated = response.LastEvaluatedKey;
    } while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

    Console.WriteLine("To continue, press Enter");
}

```

```

        Console.ReadLine();
    }

    private static void FindRepliesForAThread(string forumName, string threadSubject)
    {
        Console.WriteLine("**** Executing FindRepliesForAThread() ****");
        string replyId = forumName + "#" + threadSubject;

        var request = new QueryRequest
        {
            TableName = "Reply",
            ReturnConsumedCapacity = "TOTAL",
            KeyConditionExpression = "Id = :v_replyId",
            ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
                {":v_replyId", new AttributeValue {
                    S = replyId
                }}
            }
        };

        var response = client.Query(request);
        Console.WriteLine("No. of reads used (by query in FindRepliesForAThread) {0}\n",
            response.ConsumedCapacity.CapacityUnits);
        foreach (Dictionary<string, AttributeValue> item in response.Items)
        {
            PrintItem(item);
        }
        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }

    private static void PrintItem(
        Dictionary<string, AttributeValue> attributeList)
    {
        foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
        {
            string attributeName = kvp.Key;
            AttributeValue value = kvp.Value;

            Console.WriteLine(
                attributeName + " " +
                (value.S == null ? "" : "S=[ " + value.S + " ]") +
                (value.N == null ? "" : "N=[ " + value.N + " ]") +
                (value.SS == null ? "" : "SS=[ " + string.Join(",", value.SS.ToArray()) +
                " ]" ) +
                (value.NS == null ? "" : "NS=[ " + string.Join(",", value.NS.ToArray()) +
                " ]" ) );
            Console.WriteLine("*****");
        }
    }
}

```

Working with Scans in DynamoDB

A Scan operation in Amazon DynamoDB reads every item in a table or a secondary index. By default, a Scan operation returns all of the data attributes for every item in the table or index. You can use the

`ProjectionExpression` parameter so that `Scan` only returns some of the attributes, rather than all of them.

`Scan` always returns a result set. If no matching items are found, the result set is empty.

A single `Scan` request can retrieve a maximum of 1 MB of data. Optionally, DynamoDB can apply a filter expression to this data, narrowing the results before they are returned to the user.

Topics

- [Filter Expressions for Scan \(p. 476\)](#)
- [Limiting the Number of Items in the Result Set \(p. 476\)](#)
- [Paginating the Results \(p. 477\)](#)
- [Counting the Items in the Results \(p. 478\)](#)
- [Capacity Units Consumed by Scan \(p. 478\)](#)
- [Read Consistency for Scan \(p. 479\)](#)
- [Parallel Scan \(p. 479\)](#)
- [Scanning Tables and Indexes: Java \(p. 481\)](#)
- [Scanning Tables and Indexes: .NET \(p. 488\)](#)

Filter Expressions for Scan

If you need to further refine the `Scan` results, you can optionally provide a filter expression. A *filter expression* determines which items within the `Scan` results should be returned to you. All of the other results are discarded.

A filter expression is applied after a `Scan` finishes but before the results are returned. Therefore, a `Scan` consumes the same amount of read capacity, regardless of whether a filter expression is present.

A `Scan` operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated.

With `Scan`, you can specify any attributes in a filter expression—including partition key and sort key attributes.

The syntax for a filter expression is identical to that of a condition expression. Filter expressions can use the same comparators, functions, and logical operators as a condition expression. For more information, [Condition Expressions \(p. 392\)](#).

Example

The following AWS Command Line Interface (AWS CLI) example scans the `Thread` table and returns only the items that were last posted to by a particular user.

```
aws dynamodb scan \
    --table-name Thread \
    --filter-expression "LastPostedBy = :name" \
    --expression-attribute-values '{":name": {"S": "User A"}}'
```

Limiting the Number of Items in the Result Set

The `Scan` operation enables you to limit the number of items that it returns in the result. To do this, set the `Limit` parameter to the maximum number of items that you want the `Scan` operation to return, prior to filter expression evaluation.

For example, suppose that you `Scan` a table with a `Limit` value of 6 and without a filter expression. The `Scan` result contains the first six items from the table that match the key condition expression from the request.

Now suppose that you add a filter expression to the `Scan`. In this case, DynamoDB applies the filter expression to the six items that were returned, discarding those that do not match. The final `Scan` result contains six items or fewer, depending on the number of items that were filtered.

Paginating the Results

DynamoDB *paginates* the results from `Scan` operations. With pagination, the `Scan` results are divided into "pages" of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on.

A single `Scan` only returns a result set that fits within the 1 MB size limit. To determine whether there are more results and to retrieve them one page at a time, applications should do the following:

1. Examine the low-level `Scan` result:
 - If the result contains a `LastEvaluatedKey` element, proceed to step 2.
 - If there is *not* a `LastEvaluatedKey` in the result, then there are no more items to be retrieved.
2. Construct a new `Scan` request, with the same parameters as the previous one. However, this time, take the `LastEvaluatedKey` value from step 1 and use it as the `ExclusiveStartKey` parameter in the new `Scan` request.
3. Run the new `Scan` request.
4. Go to step 1.

In other words, the `LastEvaluatedKey` from a `Scan` response should be used as the `ExclusiveStartKey` for the next `Scan` request. If there is *not* a `LastEvaluatedKey` element in a `Scan` response, you have retrieved the final page of results. (The absence of `LastEvaluatedKey` is the only way to know that you have reached the end of the result set.)

You can use the AWS CLI to view this behavior. The AWS CLI sends low-level `Scan` requests to DynamoDB, repeatedly, until `LastEvaluatedKey` is no longer present in the results. Consider the following AWS CLI example that scans the entire `Movies` table but returns only the movies from a particular genre.

```
aws dynamodb scan \
--table-name Movies \
--projection-expression "title" \
--filter-expression 'contains(info.genres,:gen)' \
--expression-attribute-values '{":gen":{"S":"Sci-Fi"}}' \
--page-size 100 \
--debug
```

Ordinarily, the AWS CLI handles pagination automatically. However, in this example, the AWS CLI `--page-size` parameter limits the number of items per page. The `--debug` parameter prints low-level information about requests and responses.

If you run the example, the first response from DynamoDB looks similar to the following.

```
2017-07-07 12:19:14,389 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":7,"Items":[{"title":{"S":"Monster on the Campus"}}, {"title":{"S":"+1"}}, {"title":{"S":"100 Degrees Below Zero"}}, {"title":{"S":"About Time"}}, {"title":{"S":"After Earth"}}, {"title":{"S":"Age of Dinosaurs"}}, {"title":{"S":"Cloudy with a Chance of Meatballs 2"}}]'
```

```
"LastEvaluatedKey": {"year": {"N": "2013"}, "title": {"S": "Curse of Chucky"}}, "ScannedCount": 100}'
```

The `LastEvaluatedKey` in the response indicates that not all of the items have been retrieved. The AWS CLI then issues another `Scan` request to DynamoDB. This request and response pattern continues, until the final response.

```
2017-07-07 12:19:17,830 - MainThread - botocore.parsers - DEBUG - Response body: b'{"Count":1,"Items":[{"title":{"S":"WarGames"}}], "ScannedCount":6}'
```

The absence of `LastEvaluatedKey` indicates that there are no more items to retrieve.

Note

The AWS SDKs handle the low-level DynamoDB responses (including the presence or absence of `LastEvaluatedKey`) and provide various abstractions for paginating `Scan` results. For example, the SDK for Java document interface provides `java.util.Iterator` support so that you can walk through the results one at a time.

For code examples in various programming languages, see the [Amazon DynamoDB Getting Started Guide](#) and the AWS SDK documentation for your language.

Counting the Items in the Results

In addition to the items that match your criteria, the `Scan` response contains the following elements:

- `ScannedCount` — The number of items evaluated, before any `ScanFilter` is applied. A high `ScannedCount` value with few, or no, `Count` results indicates an inefficient `Scan` operation. If you did not use a filter in the request, `ScannedCount` is the same as `Count`.
- `Count` — The number of items that remain, *after* a filter expression (if present) was applied.

Note

If you do not use a filter expression, `ScannedCount` and `Count` have the same value.

If the size of the `Scan` result set is larger than 1 MB, `ScannedCount` and `Count` represent only a partial count of the total items. You need to perform multiple `Scan` operations to retrieve all the results (see [Paginating the Results \(p. 477\)](#)).

Each `Scan` response contains the `ScannedCount` and `Count` for the items that were processed by that particular `Scan` request. To get grand totals for all of the `Scan` requests, you could keep a running tally of both `ScannedCount` and `Count`.

Capacity Units Consumed by Scan

You can `Scan` any table or secondary index. `Scan` operations consume read capacity units, as follows.

| If you Scan a... | DynamoDB consumes read capacity units from... |
|------------------------|---|
| Table | The table's provisioned read capacity. |
| Global secondary index | The index's provisioned read capacity. |
| Local secondary index | The base table's provisioned read capacity. |

By default, a `Scan` operation does not return any data on how much read capacity it consumes. However, you can specify the `ReturnConsumedCapacity` parameter in a `Scan` request to obtain this information. The following are the valid settings for `ReturnConsumedCapacity`:

- **NONE** — No consumed capacity data is returned. (This is the default.)
- **TOTAL** — The response includes the aggregate number of read capacity units consumed.
- **INDEXES** — The response shows the aggregate number of read capacity units consumed, together with the consumed capacity for each table and index that was accessed.

DynamoDB calculates the number of read capacity units consumed based on item size, not on the amount of data that is returned to an application. For this reason, the number of capacity units consumed is the same whether you request all of the attributes (the default behavior) or just some of them (using a projection expression). The number is also the same whether or not you use a filter expression.

Read Consistency for Scan

A `Scan` operation performs eventually consistent reads, by default. This means that the `Scan` results might not reflect changes due to recently completed `PutItem` or `UpdateItem` operations. For more information, see [Read Consistency \(p. 16\)](#).

If you require strongly consistent reads, as of the time that the `Scan` begins, set the `ConsistentRead` parameter to `true` in the `Scan` request. This ensures that all of the write operations that completed before the `Scan` began are included in the `Scan` response.

Setting `ConsistentRead` to `true` can be useful in table backup or replication scenarios, in conjunction with [DynamoDB Streams](#). You first use `Scan` with `ConsistentRead` set to `true` to obtain a consistent copy of the data in the table. During the `Scan`, DynamoDB Streams records any additional write activity that occurs on the table. After the `Scan` is complete, you can apply the write activity from the stream to the table.

Note

A `Scan` operation with `ConsistentRead` set to `true` consumes twice as many read capacity units as compared to leaving `ConsistentRead` at its default value (`false`).

Parallel Scan

By default, the `Scan` operation processes data sequentially. Amazon DynamoDB returns data to the application in 1 MB increments, and an application performs additional `Scan` operations to retrieve the next 1 MB of data.

The larger the table or index being scanned, the more time the `Scan` takes to complete. In addition, a sequential `Scan` might not always be able to fully use the provisioned read throughput capacity: Even though DynamoDB distributes a large table's data across multiple physical partitions, a `Scan` operation can only read one partition at a time. For this reason, the throughput of a `Scan` is constrained by the maximum throughput of a single partition.

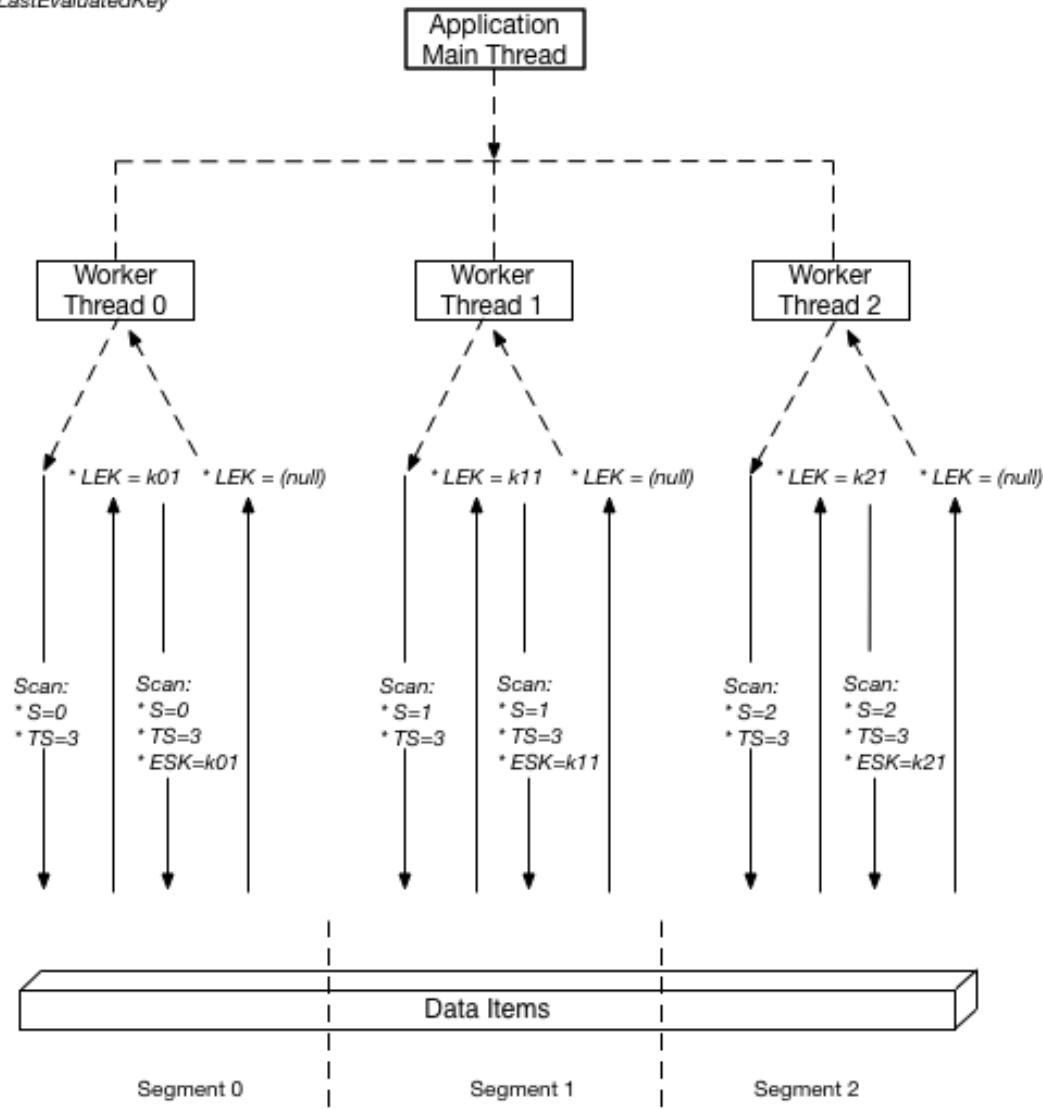
To address these issues, the `Scan` operation can logically divide a table or secondary index into multiple *segments*, with multiple application workers scanning the segments in parallel. Each worker can be a thread (in programming languages that support multithreading) or an operating system process. To perform a parallel `Scan`, each worker issues its own `Scan` request with the following parameters:

- `Segment` — A segment to be scanned by a particular worker. Each worker should use a different value for `Segment`.
- `TotalSegments` — The total number of segments for the parallel `Scan`. This value must be the same as the number of workers that your application will use.

The following diagram shows how a multithreaded application performs a parallel `Scan` with three degrees of parallelism.

S: Segment
TS: TotalSegments

ESK: ExclusiveStartKey
LEK: LastEvaluatedKey



In this diagram, the application spawns three threads and assigns each thread a number. (Segments are zero-based, so the first number is always 0.) Each thread issues a Scan request, setting Segment to its designated number and setting TotalSegments to 3. Each thread scans its designated segment, retrieving data 1 MB at a time, and returns the data to the application's main thread.

The values for Segment and TotalSegments apply to individual Scan requests, and you can use different values at any time. You might need to experiment with these values, and the number of workers you use, until your application achieves its best performance.

Note

A parallel scan with a large number of workers can easily consume all of the provisioned throughput for the table or index being scanned. It is best to avoid such scans if the table or index is also incurring heavy read or write activity from other applications.

To control the amount of data returned per request, use the `Limit` parameter. This can help prevent situations where one worker consumes all of the provisioned throughput, at the expense of all other workers.

Scanning Tables and Indexes: Java

The `Scan` operation reads all of the items in a table or index in Amazon DynamoDB.

The following are the steps to scan a table using the AWS SDK for Java Document API.

1. Create an instance of the `AmazonDynamoDB` class.
2. Create an instance of the `ScanRequest` class and provide scan parameter.
The only required parameter is the table name.
3. Execute the `scan` method and provide the `ScanRequest` object that you created in the preceding step.

The following `Reply` table stores replies for forum threads.

Example

```
Reply ( Id, ReplyDateTime, Message, PostedBy )
```

The table maintains all the replies for various forum threads. Therefore, the primary key is composed of both the `Id` (partition key) and `ReplyDateTime` (sort key). The following Java code example scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

ScanRequest scanRequest = new ScanRequest()
    .withTableName("Reply");

ScanResult result = client.scan(scanRequest);
for (Map<String, AttributeValue> item : result.getItems()){
    printItem(item);
}
```

Specifying Optional Parameters

The `scan` method supports several optional parameters. For example, you can optionally use a filter expression to filter the scan result. In a filter expression, you can specify a condition and attribute names and values on which you want the condition evaluated. For more information, see [Scan](#).

The following Java example scans the `ProductCatalog` table to find items that are priced less than 0. The example specifies the following optional parameters:

- A filter expression to retrieve only the items priced less than 0 (error condition).
- A list of attributes to retrieve for items in the query results.

Example

```
Map<String, AttributeValue> expressionAttributeValues =
    new HashMap<String, AttributeValue>();
expressionAttributeValues.put(":val", new AttributeValue().withN("0"));
```

```

ScanRequest scanRequest = new ScanRequest()
    .withTableName("ProductCatalog")
    .withFilterExpression("Price < :val")
    .withProjectionExpression("Id")
    .withExpressionAttributeValues(expressionAttributeValues);

ScanResult result = client.scan(scanRequest);
for (Map<String, AttributeValue> item : result.getItems()) {
    printItem(item);
}

```

You can also optionally limit the page size, or the number of items per page, by using the `withLimit` method of the scan request. Each time you execute the `scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute the `scan` method again by providing the primary key value of the last item in the previous page so that the `scan` method can return the next set of items. You provide this information in the request by using the `withExclusiveStartKey` method. Initially, the parameter of this method can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following Java code example scans the `ProductCatalog` table. In the request, the `withLimit` and `withExclusiveStartKey` methods are used. The do/while loop continues to scan one page at a time until the `getLastEvaluatedKey` method of the result returns a value of null.

Example

```

Map<String, AttributeValue> lastKeyEvaluated = null;
do {
    ScanRequest scanRequest = new ScanRequest()
        .withTableName("ProductCatalog")
        .withLimit(10)
        .withExclusiveStartKey(lastKeyEvaluated);

    ScanResult result = client.scan(scanRequest);
    for (Map<String, AttributeValue> item : result.getItems()){
        printItem(item);
    }
    lastKeyEvaluated = result.getLastEvaluatedKey();
} while (lastKeyEvaluated != null);

```

Example - Scan Using Java

The following Java code example provides a working sample that scans the `ProductCatalog` table to find items that are priced less than 100.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 */

```

```

/*
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class DocumentAPIScan {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);
    static String tableName = "ProductCatalog";

    public static void main(String[] args) throws Exception {
        findProductsForPriceLessThanOneHundred();
    }

    private static void findProductsForPriceLessThanOneHundred() {
        Table table = dynamoDB.getTable(tableName);

        Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
        expressionAttributeValues.put(":pr", 100);

        ItemCollection<ScanOutcome> items = table.scan("Price < :pr", // FilterExpression
            "Id, Title, ProductCategory, Price", // ProjectionExpression
            null, // ExpressionAttributeNames - not used in this example
            expressionAttributeValues);

        System.out.println("Scan of " + tableName + " for items with a price less than
100.");
        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}

```

Example - Parallel Scan Using Java

The following Java code example demonstrates a parallel scan. The program deletes and re-creates a table named `ParallelScanTest`, and then loads the table with data. When the data load is finished,

the program spawns multiple threads and issues parallel Scan requests. The program prints runtime statistics for each parallel request.

Note

The SDK for Java also provides an object persistence model, enabling you to map your client-side classes to DynamoDB tables. This approach can reduce the amount of code that you have to write. For more information, see [Java: DynamoDBMapper \(p. 225\)](#).

Note

This code example assumes that you have already loaded data into DynamoDB for your account by following the instructions in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section.

For step-by-step instructions to run the following example, see [Java Code Examples \(p. 330\)](#).

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples.document;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.List;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.TimeUnit;  
  
import com.amazonaws.AmazonServiceException;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Item;  
import com.amazonaws.services.dynamodbv2.document.ItemCollection;  
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.ScanSpec;  
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;  
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;  
import com.amazonaws.services.dynamodbv2.model.KeyType;  
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;  
  
public class DocumentAPIParallelScan {  
  
    // total number of sample items  
    static int scanItemCount = 300;  
  
    // number of items each scan request should return  
    static int scanItemLimit = 10;  
  
    // number of logical segments for parallel scan  
    static int parallelScanThreads = 16;
```

```

// table that will be used for scanning
static String parallelScanTestTableName = "ParallelScanTest";

static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
static DynamoDB dynamoDB = new DynamoDB(client);

public static void main(String[] args) throws Exception {
    try {

        // Clean up the table
        deleteTable(parallelScanTestTableName);
        createTable(parallelScanTestTableName, 10L, 5L, "Id", "N");

        // Upload sample data for scan
        uploadSampleProducts(parallelScanTestTableName, scanItemCount);

        // Scan the table using multiple threads
        parallelScan(parallelScanTestTableName, scanItemLimit, parallelScanThreads);
    }
    catch (AmazonServiceException ase) {
        System.err.println(ase.getMessage());
    }
}

private static void parallelScan(String tableName, int itemLimit, int numberOfThreads)
{
    System.out.println(
        "Scanning " + tableName + " using " + numberOfThreads + " threads " + itemLimit
+ " items at a time");
    ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);

    // Divide DynamoDB table into logical segments
    // Create one task for scanning each segment
    // Each thread will be scanning one segment
    int totalSegments = numberOfThreads;
    for (int segment = 0; segment < totalSegments; segment++) {
        // Runnable task that will only scan one segment
        ScanSegmentTask task = new ScanSegmentTask(tableName, itemLimit, totalSegments,
segment);

        // Execute the task
        executor.execute(task);
    }

    shutDownExecutorService(executor);
}

// Runnable task for scanning a single segment of a DynamoDB table
private static class ScanSegmentTask implements Runnable {

    // DynamoDB table to scan
    private String tableName;

    // number of items each scan request should return
    private int itemLimit;

    // Total number of segments
    // Equals to total number of threads scanning the table in parallel
    private int totalSegments;

    // Segment that will be scanned with by this task
    private int segment;

    public ScanSegmentTask(String tableName, int itemLimit, int totalSegments, int
segment) {
        this.tableName = tableName;
}

```

```

        this.itemLimit = itemLimit;
        this.totalSegments = totalSegments;
        this.segment = segment;
    }

    @Override
    public void run() {
        System.out.println("Scanning " + tableName + " segment " + segment + " out of "
+ totalSegments
                + " segments " + itemLimit + " items at a time...");
        int totalScannedItemCount = 0;

        Table table = dynamoDB.getTable(tableName);

        try {
            ScanSpec spec = new
ScanSpec().withMaxResultSize(itemLimit).withTotalSegments(totalSegments)
                    .withSegment(segment);

            ItemCollection<ScanOutcome> items = table.scan(spec);
            Iterator<Item> iterator = items.iterator();

            Item currentItem = null;
            while (iterator.hasNext()) {
                totalScannedItemCount++;
                currentItem = iterator.next();
                System.out.println(currentItem.toString());
            }
        }
        catch (Exception e) {
            System.err.println(e.getMessage());
        }
        finally {
            System.out.println("Scanned " + totalScannedItemCount + " items from
segment " + segment + " out of "
                    + totalSegments + " of " + tableName);
        }
    }
}

private static void uploadSampleProducts(String tableName, int itemCount) {
    System.out.println("Adding " + itemCount + " sample items to " + tableName);
    for (int productIndex = 0; productIndex < itemCount; productIndex++) {
        uploadProduct(tableName, productIndex);
    }
}

private static void uploadProduct(String tableName, int productIndex) {

    Table table = dynamoDB.getTable(tableName);

    try {
        System.out.println("Processing record #" + productIndex);

        Item item = new Item().withPrimaryKey("Id", productIndex)
                .withString("Title", "Book " + productIndex + " Title").withString("ISBN",
"111-1111111111")
                .withStringSet("Authors", new
HashSet<String>(Arrays.asList("Author1"))).withNumber("Price", 2)
                .withString("Dimensions", "8.5 x 11.0 x 0.5").withNumber("PageCount", 500)
                .withBoolean("InPublication", true).withString("ProductCategory", "Book");
        table.putItem(item);

    }
    catch (Exception e) {

```

```

        System.err.println("Failed to create item " + productIndex + " in " +
tableName);
        System.err.println(e.getMessage());
    }

}

private static void deleteTable(String tableName) {
try {

    Table table = dynamoDB.getTable(tableName);
    table.delete();
    System.out.println("Waiting for " + tableName + " to be deleted...this may take
a while...");
    table.waitForDelete();

}
catch (Exception e) {
    System.err.println("Failed to delete table " + tableName);
    e.printStackTrace(System.err);
}
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
String partitionKeyName, String partitionKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
String partitionKeyName, String partitionKeyType, String sortKeyName, String
sortKeyType) {

try {
    System.out.println("Creating table " + tableName);

    List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
    keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); // 
Partition

    // key

    List<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
    attributeDefinitions
        .add(new
AttributeDefinition().withAttributeName(partitionKeyName).withAttributeType(partitionKeyType));

    if (sortKeyName != null) {
        keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

        // key
        attributeDefinitions
            .add(new
AttributeDefinition().withAttributeName(sortKeyName).withAttributeType(sortKeyType));
    }

    Table table = dynamoDB.createTable(tableName, keySchema, attributeDefinitions,
new ProvisionedThroughput()

    .withReadCapacityUnits(readCapacityUnits).withWriteCapacityUnits(writeCapacityUnits));
}
}

```

```

        System.out.println("Waiting for " + tableName + " to be created...this may take
a while...");
        table.waitForActive();

    }
    catch (Exception e) {
        System.err.println("Failed to create table " + tableName);
        e.printStackTrace(System.err);
    }
}

private static void shutDownExecutorService(ExecutorService executor) {
    executor.shutdown();
    try {
        if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }
    }
    catch (InterruptedException e) {
        executor.shutdownNow();

        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}
}

```

Scanning Tables and Indexes: .NET

The Scan operation reads all of the items in a table or index in Amazon DynamoDB.

The following are the steps to scan a table using the AWS SDK for .NET low-level API:

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `ScanRequest` class and provide scan operation parameters.
The only required parameter is the table name.
3. Execute the `Scan` method and provide the `QueryRequest` object that you created in the preceding step.

The following Reply table stores replies for forum threads.

Example

```
>Reply ( <emphasis role="underline">Id</emphasis>, <emphasis
role="underline">ReplyDateTime</emphasis>, Message, PostedBy )
```

The table maintains all the replies for various forum threads. Therefore, the primary key is composed of both the `Id` (partition key) and `ReplyDateTime` (sort key). The following C# code example scans the entire table. The `ScanRequest` instance specifies the name of the table to scan.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();

var request = new ScanRequest
{
```

```

        TableName = "Reply",
};

var response = client.Scan(request);
var result = response.ScanResult;

foreach (Dictionary<string, AttributeValue> item in response.ScanResult.Items)
{
    // Process the result.
    PrintItem(item);
}

```

Specifying Optional Parameters

The `Scan` method supports several optional parameters. For example, you can optionally use a scan filter to filter the scan result. In a scan filter, you can specify a condition and an attribute name on which you want the condition evaluated. For more information, see [Scan](#).

The following C# code scans the `ProductCatalog` table to find items that are priced less than 0. The sample specifies the following optional parameters:

- A `FilterExpression` parameter to retrieve only the items priced less than 0 (error condition).
- A `ProjectionExpression` parameter to specify the attributes to retrieve for items in the query results.

The following C# example scans the `ProductCatalog` table to find all items priced less than 0.

Example

```

var forumScanRequest = new ScanRequest
{
    TableName = "ProductCatalog",
    // Optional parameters.
    ExpressionAttributeValues = new Dictionary<string,AttributeValue> {
        {":val", new AttributeValue { N = "0" } }
    },
    FilterExpression = "Price < :val",
    ProjectionExpression = "Id"
};

```

You can also optionally limit the page size or the number of items per page, by adding the optional `Limit` parameter. Each time you execute the `Scan` method, you get one page of results that has the specified number of items. To fetch the next page, you execute the `Scan` method again by providing the primary key value of the last item in the previous page so that the `Scan` method can return the next set of items. You provide this information in the request by setting the `ExclusiveStartKey` property. Initially, this property can be null. To retrieve subsequent pages, you must update this property value to the primary key of the last item in the preceding page.

The following C# code example scans the `ProductCatalog` table. In the request, it specifies the `Limit` and `ExclusiveStartKey` optional parameters. The do/while loop continues to scan one page at time until the `LastEvaluatedKey` returns a null value.

Example

```

Dictionary<string, AttributeValue> lastKeyEvaluated = null;
do
{
    var request = new ScanRequest

```

```

    {
        TableName = "ProductCatalog",
        Limit = 10,
        ExclusiveStartKey = lastKeyEvaluated
    };

    var response = client.Scan(request);

    foreach (Dictionary<string, AttributeValue> item
        in response.Items)
    {
        PrintItem(item);
    }
    lastKeyEvaluated = response.LastEvaluatedKey;

} while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

```

Example - Scan Using .NET

The following C# code provides a working example that scans the `ProductCatalog` table to find items priced less than 0.

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

```

/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                FindProductsForPriceLessThanZero();

                Console.WriteLine("Example complete. To continue, press Enter");
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine("To continue, press Enter");
                Console.ReadLine();
            }
        }
    }
}

```

```

        }

    }

    private static void FindProductsForPriceLessThanZero()
    {
        Dictionary<string, AttributeValue> lastKeyEvaluated = null;
        do
        {
            var request = new ScanRequest
            {
                TableName = "ProductCatalog",
                Limit = 2,
                ExclusiveStartKey = lastKeyEvaluated,
                ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
                    {"< :val", new AttributeValue {
                        N = "0"
                    }}
                },
                FilterExpression = "Price < :val",
                ProjectionExpression = "Id, Title, Price"
            };

            var response = client.Scan(request);

            foreach (Dictionary<string, AttributeValue> item
                     in response.Items)
            {
                Console.WriteLine("\nScanThreadTableUsePaging - printing.....");
                PrintItem(item);
            }
            lastKeyEvaluated = response.LastEvaluatedKey;
        } while (lastKeyEvaluated != null && lastKeyEvaluated.Count != 0);

        Console.WriteLine("To continue, press Enter");
        Console.ReadLine();
    }

    private static void PrintItem(
        Dictionary<string, AttributeValue> attributeList)
    {
        foreach (KeyValuePair<string, AttributeValue> kvp in attributeList)
        {
            string attributeName = kvp.Key;
            AttributeValue value = kvp.Value;

            Console.WriteLine(
                attributeName + " " +
                (value.S == null ? "" : "S=[[" + value.S + "]]") +
                (value.N == null ? "" : "N=[[" + value.N + "]]") +
                (value.SS == null ? "" : "SS=[[" + string.Join(",", value.SS.ToArray()) +
                "]]") +
                (value.NS == null ? "" : "NS=[[" + string.Join(",", value.NS.ToArray()) +
                "]]")
            );
        }
        Console.WriteLine("*****");
    }
}

```

Example - Parallel Scan Using .NET

The following C# code example demonstrates a parallel scan. The program deletes and then re-creates the `ProductCatalog` table and then loads the table with data. When the data load is finished, the

program spawns multiple threads and issues parallel Scan requests. Finally, the program prints a summary of runtime statistics.

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

```


/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;

namespace com.amazonaws.codesamples
{
    class LowLevelParallelScan
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();
        private static string tableName = "ProductCatalog";
        private static int exampleItemCount = 100;
        private static int scanItemLimit = 10;
        private static int totalSegments = 5;

        static void Main(string[] args)
        {
            try
            {
                DeleteExampleTable();
                CreateExampleTable();
                UploadExampleData();
                ParallelScanExampleTable();
            }
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }

            Console.WriteLine("To continue, press Enter");
            Console.ReadLine();
        }

        private static void ParallelScanExampleTable()
        {
            Console.WriteLine("\n*** Creating {0} Parallel Scan Tasks to scan {1}",
totalSegments, tableName);
            Task[] tasks = new Task[totalSegments];
            for (int segment = 0; segment < totalSegments; segment++)
            {
                int tmpSegment = segment;
                Task task = Task.Factory.StartNew(() =>
                {


```

```

        ScanSegment(totalSegments, tmpSegment);
    });

    tasks[segment] = task;
}

Console.WriteLine("All scan tasks are created, waiting for them to complete.");
Task.WaitAll(tasks);

Console.WriteLine("All scan tasks are completed.");
}

private static void ScanSegment(int totalSegments, int segment)
{
    Console.WriteLine("**** Starting to Scan Segment {0} of {1} out of {2} total
segments ***", segment, tableName, totalSegments);
    Dictionary<string, AttributeValue> lastEvaluatedKey = null;
    int totalScannedItemCount = 0;
    int totalScanRequestCount = 0;
    do
    {
        var request = new ScanRequest
        {
            TableName = tableName,
            Limit = scanItemLimit,
            ExclusiveStartKey = lastEvaluatedKey,
            Segment = segment,
            TotalSegments = totalSegments
        };

        var response = client.Scan(request);
        lastEvaluatedKey = response.LastEvaluatedKey;
        totalScanRequestCount++;
        totalScannedItemCount += response.ScannedCount;
        foreach (var item in response.Items)
        {
            Console.WriteLine("Segment: {0}, Scanned Item with Title: {1}",
segment, item["Title"].S);
        }
    } while (lastEvaluatedKey.Count != 0);

    Console.WriteLine("**** Completed Scan Segment {0} of {1}.
TotalScanRequestCount: {2}, TotalScannedItemCount: {3} ***", segment, tableName,
totalScanRequestCount, totalScannedItemCount);
}

private static void UploadExampleData()
{
    Console.WriteLine("\n*** Uploading {0} Example Items to {1} Table***",
exampleItemCount, tableName);
    Console.Write("Uploading Items: ");
    for (int itemIndex = 0; itemIndex < exampleItemCount; itemIndex++)
    {
        Console.Write("{0}, ", itemIndex);
        CreateItem(itemIndex.ToString());
    }
    Console.WriteLine();
}

private static void CreateItem(string itemIndex)
{
    var request = new PutItemRequest
    {
        TableName = tableName,
        Item = new Dictionary<string, AttributeValue>()
    }
}

```

```

        {
            "Id", new AttributeValue {
                N = itemIndex
            },
            {
                "Title", new AttributeValue {
                    S = "Book " + itemIndex + " Title"
                },
                {
                    "ISBN", new AttributeValue {
                        S = "11-11-11-11"
                    },
                    {
                        "Authors", new AttributeValue {
                            SS = new List<string>{"Author1", "Author2" }
                        },
                        {
                            "Price", new AttributeValue {
                                N = "20.00"
                            },
                            {
                                "Dimensions", new AttributeValue {
                                    S = "8.5x11.0x.75"
                                },
                                {
                                    "InPublication", new AttributeValue {
                                        BOOL = false
                                    }
                                }
                            }
                        };
                    client.PutItem(request);
                }
            }

private static void CreateExampleTable()
{
    Console.WriteLine("\n*** Creating {0} Table ***", tableName);
    var request = new CreateTableRequest
    {
        AttributeDefinitions = new List<AttributeDefinition>()
    {
        new AttributeDefinition
        {
            AttributeName = "Id",
            AttributeType = "N"
        },
        KeySchema = new List<KeySchemaElement>
    {
        new KeySchemaElement
        {
            AttributeName = "Id",
            KeyType = "HASH" //Partition key
        }
    },
        ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = 5,
        WriteCapacityUnits = 6
    },
        TableName = tableName
    };

    var response = client.CreateTable(request);

    var result = response;
    var tableDescription = result.TableDescription;
    Console.WriteLine("{1}: {0} \t ReadsPerSec: {2} \t WritesPerSec: {3}",
        tableDescription.TableStatus,
        tableDescription.TableName,
        tableDescription.ProvisionedThroughput.ReadCapacityUnits,
        tableDescription.ProvisionedThroughput.WriteCapacityUnits);

    string status = tableDescription.TableStatus;
}

```

```

        Console.WriteLine(tableName + " - " + status);

        WaitUntilTableReady(tableName);
    }

    private static void DeleteExampleTable()
    {
        try
        {
            Console.WriteLine("\n*** Deleting {0} Table ***", tableName);
            var request = new DeleteTableRequest
            {
                TableName = tableName
            };

            var response = client.DeleteTable(request);
            var result = response;
            Console.WriteLine("{0} is being deleted...", tableName);
            WaitUntilTableDeleted(tableName);
        }
        catch (ResourceNotFoundException)
        {
            Console.WriteLine("{0} Table delete failed: Table does not exist",
tableName);
        }
    }

    private static void WaitUntilTableReady(string tableName)
    {
        string status = null;
        // Let us wait until table is created. Call DescribeTable.
        do
        {
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

                Console.WriteLine("Table name: {0}, status: {1}",
                    res.Table.TableName,
                    res.Table.TableStatus);
                status = res.Table.TableStatus;
            }
            catch (ResourceNotFoundException)
            {
                // DescribeTable is eventually consistent. So you might
                // get resource not found. So we handle the potential exception.
            }
        } while (status != "ACTIVE");
    }

    private static void WaitUntilTableDeleted(string tableName)
    {
        string status = null;
        // Let us wait until table is deleted. Call DescribeTable.
        do
        {
            System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
            try
            {
                var res = client.DescribeTable(new DescribeTableRequest
                {
                    TableName = tableName
                });

```

```
        });

        Console.WriteLine("Table name: {0}, status: {1}",
            res.Table.TableName,
            res.Table.TableStatus);
        status = res.Table.TableStatus;
    }
    catch (ResourceNotFoundException)
    {
        Console.WriteLine("Table name: {0} is not found. It is deleted",
    tableName);
        return;
    }
} while (status == "DELETING");
}
}
```

Improving Data Access with Secondary Indexes

Topics

- [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#)
- [Local Secondary Indexes \(p. 536\)](#)

Amazon DynamoDB provides fast access to items in a table by specifying primary key values. However, many applications might benefit from having one or more secondary (or alternate) keys available, to allow efficient access to data with attributes other than the primary key. To address this, you can create one or more secondary indexes on a table and issue `Query` or `Scan` requests against these indexes.

A *secondary index* is a data structure that contains a subset of attributes from a table, along with an alternate key to support `Query` operations. You can retrieve data from the index using a `Query`, in much the same way as you use `Query` with a table. A table can have multiple secondary indexes, which give your applications access to many different query patterns.

Note

You can also `Scan` an index, in much the same way as you would `Scan` a table.

Every secondary index is associated with exactly one table, from which it obtains its data. This is called the *base table* for the index. When you create an index, you define an alternate key for the index (partition key and sort key). You also define the attributes that you want to be *projected*, or copied, from the base table into the index. DynamoDB copies these attributes into the index, along with the primary key attributes from the base table. You can then query or scan the index just as you would query or scan a table.

Every secondary index is automatically maintained by DynamoDB. When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes.

DynamoDB supports two types of secondary indexes:

- **Global secondary index** — An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index is stored in its own partition space away from the base table and scales separately from the base table.
- **Local secondary index** — An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value.

You should consider your application's requirements when you determine which type of index to use. The following table shows the main differences between a global secondary index and a local secondary index.

| Characteristic | Global Secondary Index | Local Secondary Index |
|--|--|---|
| Key Schema | The primary key of a global secondary index can be either simple (partition key) or composite (partition key and sort key). | The primary key of a local secondary index must be composite (partition key and sort key). |
| Key Attributes | The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary. | The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary. |
| Size Restrictions Per Partition Key Value | There are no size restrictions for global secondary indexes. | For each partition key value, the total size of all indexed items must be 10 GB or less. |
| Online Index Operations | Global secondary indexes can be created at the same time that you create a table. You can also add a new global secondary index to an existing table, or delete an existing global secondary index. For more information, see Managing Global Secondary Indexes (p. 507) . | Local secondary indexes are created at the same time that you create a table. You cannot add a local secondary index to an existing table, nor can you delete any local secondary indexes that currently exist. |
| Queries and Partitions | A global secondary index lets you query over the entire table, across all partitions. | A local secondary index lets you query over a single partition, as specified by the partition key value in the query. |
| Read Consistency | Queries on global secondary indexes support eventual consistency only. | When you query a local secondary index, you can choose either eventual consistency or strong consistency. |
| Provisioned Throughput Consumption | Every global secondary index has its own provisioned throughput settings for read and write activity. Queries or scans on a global secondary index consume capacity units from the index, not from the base table. The same holds true for global secondary index updates due to table writes. | Queries or scans on a local secondary index consume read capacity units from the base table. When you write to a table, its local secondary indexes are also updated; these updates consume write capacity units from the base table. |
| Projected Attributes | With global secondary index queries or scans, you can only request the attributes that | If you query or scan a local secondary index, you can request attributes that are |

| Characteristic | Global Secondary Index | Local Secondary Index |
|----------------|--|--|
| | are projected into the index. DynamoDB does not fetch any attributes from the table. | not projected in to the index. DynamoDB automatically fetches those attributes from the table. |

If you want to create more than one table with secondary indexes, you must do so sequentially. For example, you would create the first table and wait for it to become **ACTIVE**, create the next table and wait for it to become **ACTIVE**, and so on. If you try to concurrently create more than one table with a secondary index, DynamoDB returns a `LimitExceededException`.

For each secondary index, you must specify the following:

- The type of index to be created – either a global secondary index or a local secondary index.
- A name for the index. The naming rules for indexes are the same as those for tables, as listed in [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#). The name must be unique for the base table it is associated with, but you can use the same name for indexes that are associated with different base tables.
- The key schema for the index. Every attribute in the index key schema must be a top-level attribute of type `String`, `Number`, or `Binary`. Other data types, including documents and sets, are not allowed. Other requirements for the key schema depend on the type of index:
 - For a global secondary index, the partition key can be any scalar attribute of the base table. A sort key is optional, and it too can be any scalar attribute of the base table.
 - For a local secondary index, the partition key must be the same as the base table's partition key, and the sort key must be a non-key base table attribute.
- Additional attributes, if any, to project from the base table into the index. These attributes are in addition to the table's key attributes, which are automatically projected into every index. You can project attributes of any data type, including scalars, documents, and sets.
- The provisioned throughput settings for the index, if necessary:
 - For a global secondary index, you must specify read and write capacity unit settings. These provisioned throughput settings are independent of the base table's settings.
 - For a local secondary index, you do not need to specify read and write capacity unit settings. Any read and write operations on a local secondary index draw from the provisioned throughput settings of its base table.

For maximum query flexibility, you can create up to 20 global secondary indexes (default limit) and up to 5 local secondary indexes per table.

The limit of global secondary indexes per table is five for the following AWS Regions:

- AWS GovCloud (US-East)
- AWS GovCloud (US-West)
- Europe (Stockholm)

To get a detailed listing of secondary indexes on a table, use the `DescribeTable` operation. `DescribeTable` returns the name, storage size, and item counts for every secondary index on the table. These values are not updated in real time, but they are refreshed approximately every six hours.

You can access the data in a secondary index using either the `Query` or `Scan` operation. You must specify the name of the base table and the name of the index that you want to use, the attributes to be returned in the results, and any condition expressions or filters that you want to apply. DynamoDB can return the results in ascending or descending order.

When you delete a table, all of the indexes associated with that table are also deleted.

For best practices, see [Best Practices for Using Secondary Indexes in DynamoDB \(p. 906\)](#).

Using Global Secondary Indexes in DynamoDB

Some applications might need to perform many kinds of queries, using a variety of different attributes as query criteria. To support these requirements, you can create one or more *global secondary indexes* and issue `Query` requests against these indexes in Amazon DynamoDB.

Topics

- [Scenario: Using a Global Secondary Index \(p. 499\)](#)
- [Attribute Projections \(p. 502\)](#)
- [Querying a Global Secondary Index \(p. 503\)](#)
- [Scanning a Global Secondary Index \(p. 504\)](#)
- [Data Synchronization Between Tables and Global Secondary Indexes \(p. 504\)](#)
- [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 505\)](#)
- [Storage Considerations for Global Secondary Indexes \(p. 506\)](#)
- [Managing Global Secondary Indexes \(p. 507\)](#)
- [Working with Global Secondary Indexes: Java \(p. 518\)](#)
- [Working with Global Secondary Indexes: .NET \(p. 526\)](#)

Scenario: Using a Global Secondary Index

To illustrate, consider a table named `GameScores` that tracks users and scores for a mobile gaming application. Each item in `GameScores` is identified by a partition key (`UserId`) and a sort key (`GameTitle`). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)

GameScores

| UserId | GameTitle | TopScore | TopScoreDateTime | Wins | Losses | |
|--------|-------------------|----------|-----------------------|------|--------|-----|
| "101" | "Galaxy Invaders" | 5842 | "2015-09-15:17:24:31" | 21 | 72 | ... |
| "101" | "Meteor Blasters" | 1000 | "2015-10-22:23:18:01" | 12 | 3 | ... |
| "101" | "Starship X" | 24 | "2015-08-31:13:14:21" | 4 | 9 | ... |
| "102" | "Alien Adventure" | 192 | "2015-07-12:11:07:56" | 32 | 192 | ... |
| "102" | "Galaxy Invaders" | 0 | "2015-09-18:07:33:42" | 0 | 5 | ... |
| "103" | "Attack Ships" | 3 | "2015-10-19:01:13:24" | 1 | 8 | ... |
| "103" | "Galaxy Invaders" | 2317 | "2015-09-11:06:53:00" | 40 | 3 | ... |
| "103" | "Meteor Blasters" | 723 | "2015-10-19:01:13:24" | 22 | 12 | ... |
| "103" | "Starship X" | 42 | "2015-07-11:06:53:00" | 4 | 19 | ... |
| ... | ... | ... | ... | ... | ... | ... |

Now suppose that you wanted to write a leaderboard application to display top scores for each game. A query that specified the key attributes (`UserId` and `GameTitle`) would be very efficient. However, if the application needed to retrieve data from `GameScores` based on `GameTitle` only, it would need to use a `Scan` operation. As more items are added to the table, scans of all the data would become slow and inefficient. This makes it difficult to answer questions such as the following:

- What is the top score ever recorded for the game Meteor Blasters?
- Which user had the highest score for Galaxy Invaders?
- What was the highest ratio of wins vs. losses?

To speed up queries on non-key attributes, you can create a global secondary index. A global secondary index contains a selection of attributes from the base table, but they are organized by a primary key that is different from that of the table. The index key does not need to have any of the key attributes from the table. It doesn't even need to have the same key schema as a table.

For example, you could create a global secondary index named `GameTitleIndex`, with a partition key of `GameTitle` and a sort key of `TopScore`. The base table's primary key attributes are always projected into an index, so the `UserId` attribute is also present. The following diagram shows what `GameTitleIndex` index would look like.

GameTitleIndex

| <i>GameTitle</i> | <i>TopScore</i> | <i>UserId</i> |
|-------------------|-----------------|---------------|
| “Alien Adventure” | 192 | “102” |
| “Attack Ships” | 3 | “103” |
| “Galaxy Invaders” | 0 | “102” |
| “Galaxy Invaders” | 2317 | “103” |
| “Galaxy Invaders” | 5842 | “101” |
| “Meteor Blasters” | 723 | “103” |
| “Meteor Blasters” | 1000 | “101” |
| “Starship X” | 24 | “101” |
| “Starship X” | 42 | “103” |

*** *** ***

Now you can query `GameTitleIndex` and easily obtain the scores for Meteor Blasters. The results are ordered by the sort key values, `TopScore`. If you set the `ScanIndexForward` parameter to `false`, the results are returned in descending order, so the highest score is returned first.

Every global secondary index must have a partition key, and can have an optional sort key. The index key schema can be different from the base table schema. You could have a table with a simple primary key (partition key), and create a global secondary index with a composite primary key (partition key and sort key)—or vice versa. The index key attributes can consist of any top-level `String`, `Number`, or `Binary` attributes from the base table. Other scalar types, document types, and set types are not allowed.

You can project other base table attributes into the index if you want. When you query the index, DynamoDB can retrieve these projected attributes efficiently. However, global secondary index queries cannot fetch attributes from the base table. For example, if you query `GameTitleIndex` as shown in the previous diagram, the query could not access any non-key attributes other than `TopScore` (although the key attributes `GameTitle` and `UserId` would automatically be projected).

In a DynamoDB table, each key value must be unique. However, the key values in a global secondary index do not need to be unique. To illustrate, suppose that a game named Comet Quest is especially difficult, with many new users trying but failing to get a score above zero. The following is some data that could represent this.

| UserId | GameTitle | TopScore |
|--------|-------------|----------|
| 123 | Comet Quest | 0 |
| 201 | Comet Quest | 0 |
| 301 | Comet Quest | 0 |

When this data is added to the `GameScores` table, DynamoDB propagates it to `GameTitleIndex`. If we then query the index using `Comet Quest` for `GameTitle` and `0` for `TopScore`, the following data is returned.

| GameTitle | TopScore | UserId |
|---------------|----------|--------|
| “Comet Quest” | 0 | “123” |
| “Comet Quest” | 0 | “201” |
| “Comet Quest” | 0 | “301” |

Only the items with the specified key values appear in the response. Within that set of data, the items are in no particular order.

A global secondary index only tracks data items where its key attributes actually exist. For example, suppose that you added another new item to the `GameScores` table, but only provided the required primary key attributes.

| UserId | GameTitle |
|--------|-------------|
| 400 | Comet Quest |

Because you didn't specify the `TopScore` attribute, DynamoDB would not propagate this item to `GameTitleIndex`. Thus, if you queried `GameScores` for all the Comet Quest items, you would get the following four items.

| User Id | Game Title | Top Score |
|---------|---------------|-----------|
| “123” | “Comet Quest” | 0 |
| “201” | “Comet Quest” | 0 |
| “301” | “Comet Quest” | 0 |
| “400” | “Comet Quest” | |

A similar query on GameTitleIndex would still return three items, rather than four. This is because the item with the nonexistent TopScore is not propagated to the index.

| Game Title | Top Score | User Id |
|---------------|-----------|---------|
| “Comet Quest” | 0 | “123” |
| “Comet Quest” | 0 | “201” |
| “Comet Quest” | 0 | “301” |

Attribute Projections

A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- *KEYS_ONLY* – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The *KEYS_ONLY* option results in the smallest possible secondary index.
- *INCLUDE* – In addition to the attributes described in *KEYS_ONLY*, the secondary index will include other non-key attributes that you specify.
- *ALL* – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an *ALL* projection results in the largest possible secondary index.

In the previous diagram, GameTitleIndex has only one projected attribute: *User Id*. So while an application can efficiently determine the *User Id* of the top scorers for each game using *Game Title* and *Top Score* in queries, it can't efficiently determine the highest ratio of wins vs. losses for the top scorers. To do so, it would have to perform an additional query on the base table to fetch the wins and losses for each of the top scorers. A more efficient way to support queries on this data would be to project these attributes from the base table into the global secondary index, as shown in this diagram.

GameTitleIndex

| <i>GameTitle</i> | <i>TopScore</i> | <i>UserId</i> | <i>Wins</i> | <i>Losses</i> |
|-------------------|-----------------|---------------|-------------|---------------|
| “Alien Adventure” | 192 | “102” | 32 | 192 |
| “Attack Ships” | 3 | “103” | 1 | 8 |
| “Galaxy Invaders” | 0 | “102” | 0 | 5 |
| “Galaxy Invaders” | 2317 | “103” | 40 | 3 |
| “Galaxy Invaders” | 5842 | “101” | 21 | 72 |
| “Meteor Blasters” | 723 | “103” | 22 | 12 |
| “Meteor Blasters” | 1000 | “101” | 12 | 3 |
| “Starship X” | 24 | “101” | 4 | 9 |
| “Starship X” | 42 | “103” | 4 | 19 |
| ... | ... | ... | ... | ... |

Because the non-key attributes *Wins* and *Losses* are projected into the index, an application can determine the wins vs. losses ratio for any game, or for any combination of game and user ID.

When you choose the attributes to project into a global secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a global secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a global secondary index. The additional storage costs for the global secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table—into a global secondary index. This gives you maximum flexibility. However, your storage cost would increase, or even double.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting **KEYS_ONLY**. The global secondary index would be of minimal size, but would still be available when needed for query activity.

Querying a Global Secondary Index

You can use the `Query` operation to access one or more items in a global secondary index. The query must specify the name of the base table and the name of the index that you want to use, the attributes

to be returned in the query results, and any query conditions that you want to apply. DynamoDB can return the results in ascending or descending order.

Consider the following data returned from a `Query` that requests gaming data for a leaderboard application.

```
{  
    "TableName": "GameScores",  
    "IndexName": "GameTitleIndex",  
    "KeyConditionExpression": "GameTitle = :v_title",  
    "ExpressionAttributeValues": {  
        ":v_title": {"S": "Meteor Blasters"}  
    },  
    "ProjectionExpression": "UserId, TopScore",  
    "ScanIndexForward": false  
}
```

In this query:

- DynamoDB accesses `GameTitleIndex`, using the `GameTitle` partition key to locate the index items for Meteor Blasters. All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this game, DynamoDB uses the index to access all of the user IDs and top scores for this game.
- The results are returned, sorted in descending order because the `ScanIndexForward` parameter is set to `false`.

Scanning a Global Secondary Index

You can use the `Scan` operation to retrieve all of the data from a global secondary index. You must provide the base table name and the index name in the request. With a `Scan`, DynamoDB reads all of the data in the index and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the `FilterExpression` parameter of the `Scan` operation. For more information, see [Filter Expressions for Scan \(p. 476\)](#).

Data Synchronization Between Tables and Global Secondary Indexes

DynamoDB automatically synchronizes each global secondary index with its base table. When an application writes or deletes items in a table, any global secondary indexes on that table are updated asynchronously, using an eventually consistent model. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations When Changing Read/Write Capacity Mode \(p. 340\)](#).

When you create a global secondary index, you specify one or more index key attributes and their data types. This means that whenever you write an item to the base table, the data types for those attributes must match the index key schema's data types. In the case of `GameTitleIndex`, the `GameTitle` partition key in the index is defined as a `String` data type. The `TopScore` sort key in the index is of type `Number`. If you try to add an item to the `GameScores` table and specify a different data type for either `GameTitle` or `TopScore`, DynamoDB returns a `ValidationException` because of the data type mismatch.

When you put or delete items in a table, the global secondary indexes on that table are updated in an eventually consistent fashion. Changes to the table data are propagated to the global secondary indexes

within a fraction of a second, under normal conditions. However, in some unlikely failure scenarios, longer propagation delays might occur. Because of this, your applications need to anticipate and handle situations where a query on a global secondary index returns results that are not up to date.

If you write an item to a table, you don't have to specify the attributes for any global secondary index sort key. Using `GameTitleIndex` as an example, you would not need to specify a value for the `TopScore` attribute to write a new item to the `GameScores` table. In this case, DynamoDB does not write any data to the index for this particular item.

A table with many global secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 505\)](#).

Provisioned Throughput Considerations for Global Secondary Indexes

When you create a global secondary index on a provisioned mode table, you must specify read and write capacity units for the expected workload on that index. The provisioned throughput settings of a global secondary index are separate from those of its base table. A `Query` operation on a global secondary index consumes read capacity units from the index, not the base table. When you put, update or delete items in a table, the global secondary indexes on that table are also updated. These index updates consume write capacity units from the index, not from the base table.

For example, if you `Query` a global secondary index and exceed its provisioned read capacity, your request will be throttled. If you perform heavy write activity on the table, but a global secondary index on that table has insufficient write capacity, the write activity on the table will be throttled.

Note

To avoid potential throttling, the provisioned write capacity for a global secondary index should be equal or greater than the write capacity of the base table because new updates write to both the base table and global secondary index.

To view the provisioned throughput settings for a global secondary index, use the `DescribeTable` operation. Detailed information about all of the table's global secondary indexes is returned.

Read Capacity Units

Global secondary indexes support eventually consistent reads, each of which consume one half of a read capacity unit. This means that a single global secondary index query can retrieve up to $2 \times 4 \text{ KB} = 8 \text{ KB}$ per read capacity unit.

For global secondary index queries, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned. The result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#).

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned.

For example, consider a global secondary index where each item contains 2,000 bytes of data. Now suppose that you `Query` this index and that the query returns eight items. The total size of the matching items is $2,000 \text{ bytes} \times 8 \text{ items} = 16,000 \text{ bytes}$. This result is then rounded up to the nearest 4 KB boundary. Because global secondary index queries are eventually consistent, the total cost is $0.5 \times (16 \text{ KB} / 4 \text{ KB})$, or 2 read capacity units.

Write Capacity Units

When an item in a table is added, updated, or deleted, and a global secondary index is affected by this, the global secondary index consumes provisioned write capacity units for the operation. The total provisioned throughput cost for a write consists of the sum of write capacity units consumed by writing to the base table and those consumed by updating the global secondary indexes. If a write to a table does not require a global secondary index update, no write capacity is consumed from the index.

For a table write to succeed, the provisioned throughput settings for the table and all of its global secondary indexes must have enough write capacity to accommodate the write. Otherwise, the write to the table is throttled.

The cost of writing an item to a global secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required, one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.
- If an update to the table only changes the value of projected attributes in the index key schema, but does not change the value of any indexed key attribute, one write is required to update the values of the projected attributes into the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries will need to return and projecting only those attributes into the index.

Storage Considerations for Global Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any global secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any global secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)
- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a global secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the base table that have the global secondary index key attributes.

If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index partition key or sort key, DynamoDB doesn't write any data for that item to the index.

Managing Global Secondary Indexes

This section describes how to create, modify, and delete global secondary indexes in Amazon DynamoDB.

Topics

- [Creating a Table with Global Secondary Indexes \(p. 507\)](#)
- [Describing the Global Secondary Indexes on a Table \(p. 507\)](#)
- [Adding a Global Secondary Index to an Existing Table \(p. 508\)](#)
- [Deleting a Global Secondary Index \(p. 510\)](#)
- [Modifying a Global Secondary Index during Creation \(p. 510\)](#)
- [Detecting and Correcting Index Key Violations \(p. 510\)](#)

Creating a Table with Global Secondary Indexes

To create a table with one or more global secondary indexes, use the `CreateTable` operation with the `GlobalSecondaryIndexes` parameter. For maximum query flexibility, you can create up to 20 global secondary indexes (default limit) per table.

You must specify one attribute to act as the index partition key. You can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table. For example, in the *GameScores* table (see [Using Global Secondary Indexes in DynamoDB \(p. 499\)](#)), neither `TopScore` nor `TopScoreDateTime` are key attributes. You could create a global secondary index with a partition key of `TopScore` and a sort key of `TopScoreDateTime`. You might use such an index to determine whether there is a correlation between high scores and the time of day a game is played.

Each index key attribute must be a scalar of type `String`, `Number`, or `Binary`. (It cannot be a document or a set.) You can project attributes of any data type into a global secondary index. This includes scalars, documents, and sets. For a complete list of data types, see [Data Types \(p. 13\)](#).

If using provisioned mode, you must provide `ProvisionedThroughput` settings for the index, consisting of `ReadCapacityUnits` and `WriteCapacityUnits`. These provisioned throughput settings are separate from those of the table, but behave in similar ways. For more information, see [Provisioned Throughput Considerations for Global Secondary Indexes \(p. 505\)](#).

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations When Changing Read/Write Capacity Mode \(p. 340\)](#).

Describing the Global Secondary Indexes on a Table

To view the status of all the global secondary indexes on a table, use the `DescribeTable` operation. The `GlobalSecondaryIndexes` portion of the response shows all of the indexes on the table, along with the current status of each (`IndexStatus`).

The `IndexStatus` for a global secondary index will be one of the following:

- **CREATING** — The index is currently being created, and is not yet available for use.
- **ACTIVE** — The index is ready for use, and applications can perform `Query` operations on the index.
- **UPDATING** — The provisioned throughput settings of the index are being changed.
- **DELETING** — The index is currently being deleted, and can no longer be used.

When DynamoDB has finished building a global secondary index, the index status changes from **CREATING** to **ACTIVE**.

Adding a Global Secondary Index to an Existing Table

To add a global secondary index to an existing table, use the `UpdateTable` operation with the `GlobalSecondaryIndexUpdates` parameter. You must provide the following:

- An index name. The name must be unique among all the indexes on the table.
- The key schema of the index. You must specify one attribute for the index partition key; you can optionally specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table. The data types for each schema attribute must be scalar: `String`, `Number`, or `Binary`.
- The attributes to be projected from the table into the index:
 - `KEYS_ONLY` — Each item in the index consists only of the table partition key and sort key values, plus the index key values.
 - `INCLUDE` — In addition to the attributes described in `KEYS_ONLY`, the secondary index includes other non-key attributes that you specify.
 - `ALL` — The index includes all of the attributes from the source table.
- The provisioned throughput settings for the index, consisting of `ReadCapacityUnits` and `WriteCapacityUnits`. These provisioned throughput settings are separate from those of the table.

You can only create one global secondary index per `UpdateTable` operation.

Phases of Index Creation

When you add a new global secondary index to an existing table, the table continues to be available while the index is being built. However, the new index is not available for Query operations until its status changes from `CREATING` to `ACTIVE`.

Behind the scenes, DynamoDB builds the index in two phases:

Resource Allocation

DynamoDB allocates the compute and storage resources that are needed for building the index.

During the resource allocation phase, the `IndexStatus` attribute is `CREATING` and the `Backfilling` attribute is false. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is in the resource allocation phase, you can't delete the index or delete its parent table. You also can't modify the provisioned throughput of the index or the table. You cannot add or delete other indexes on the table. However, you can modify the provisioned throughput of these other indexes.

Backfilling

For each item in the table, DynamoDB determines which set of attributes to write to the index based on its projection (`KEYS_ONLY`, `INCLUDE`, or `ALL`). It then writes these attributes to the index. During the backfill phase, DynamoDB tracks the items that are being added, deleted, or updated in the table. The attributes from these items are also added, deleted, or updated in the index as appropriate.

During the backfilling phase, the `IndexStatus` attribute is set to `CREATING`, and the `Backfilling` attribute is true. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is backfilling, you cannot delete its parent table. However, you can still delete the index or modify the provisioned throughput of the table and any of its global secondary indexes.

Note

During the backfilling phase, some writes of violating items might succeed while others are rejected. After backfilling, all writes to items that violate the new index's key schema are rejected. We recommend that you run the Violation Detector tool after the backfill phase finishes to detect and resolve any key violations that might have occurred. For more information, see [Detecting and Correcting Index Key Violations \(p. 510\)](#).

While the resource allocation and backfilling phases are in progress, the index is in the `CREATING` state. During this time, DynamoDB performs read operations on the table. You are not charged for this read activity.

When the index build is complete, its status changes to `ACTIVE`. You can't `Query` or `Scan` the index until it is `ACTIVE`.

Note

In some cases, DynamoDB can't write data from the table to the index because of index key violations. This can occur if:

- The data type of an attribute value does not match the data type of an index key schema data type.
- The size of an attribute exceeds the maximum length for an index key attribute.
- An index key attribute has an empty String or empty Binary attribute value.

Index key violations do not interfere with global secondary index creation. However, when the index becomes `ACTIVE`, the violating keys are not present in the index.

DynamoDB provides a standalone tool for finding and resolving these issues. For more information, see [Detecting and Correcting Index Key Violations \(p. 510\)](#).

Adding a Global Secondary Index to a Large Table

The time required for building a global secondary index depends on several factors, such as the following:

- The size of the table
- The number of items in the table that qualify for inclusion in the index
- The number of attributes projected into the index
- The provisioned write capacity of the index
- Write activity on the main table during index builds

If you are adding a global secondary index to a very large table, it might take a long time for the creation process to complete. To monitor progress and determine whether the index has sufficient write capacity, consult the following Amazon CloudWatch metrics:

- `OnlineIndexPercentageProgress`
- `OnlineIndexConsumedWriteCapacity`
- `OnlineIndexThrottleEvents`

Note

For more information about CloudWatch metrics related to DynamoDB, see [DynamoDB Metrics \(p. 858\)](#).

If the provisioned write throughput setting on the index is too low, the index build will take longer to complete. To shorten the time it takes to build a new global secondary index, you can increase its provisioned write capacity temporarily.

Note

As a general rule, we recommend setting the provisioned write capacity of the index to 1.5 times the write capacity of the table. This is a good setting for many use cases. However, your actual requirements might be higher or lower.

While an index is being backfilled, DynamoDB uses internal system capacity to read from the table. This is to minimize the impact of the index creation and to assure that your table does not run out of read capacity.

However, it is possible that the volume of incoming write activity might exceed the provisioned write capacity of the index. This is a bottleneck scenario, in which the index creation takes more time because the write activity to the index is throttled. During the index build, we recommend that you monitor the Amazon CloudWatch metrics for the index to determine whether its consumed write capacity is exceeding its provisioned capacity. In a bottleneck scenario, you should increase the provisioned write capacity on the index to avoid write throttling during the backfill phase.

After the index has been created, you should set its provisioned write capacity to reflect the normal usage of your application.

Deleting a Global Secondary Index

If you no longer need a global secondary index, you can delete it using the `UpdateTable` operation.

You can delete only one global secondary index per `UpdateTable` operation.

While the global secondary index is being deleted, there is no effect on any read or write activity in the parent table. While the deletion is in progress, you can still modify the provisioned throughput on other indexes.

Note

When you delete a table using the `DeleteTable` action, all of the global secondary indexes on that table are also deleted.

Modifying a Global Secondary Index during Creation

While an index is being built, you can use the `DescribeTable` operation to determine what phase it is in. The description for the index includes a Boolean attribute, `Backfilling`, to indicate whether DynamoDB is currently loading the index with items from the table. If `Backfilling` is true, the resource allocation phase is complete and the index is now backfilling.

While the backfill is proceeding, you can update the provisioned throughput parameters for the index. You might decide to do this in order to speed up the index build: You can increase the write capacity of the index while it is being built, and then decrease it afterward. To modify the provisioned throughput settings of the index, use the `UpdateTable` operation. The index status changes to `UPDATING`, and `Backfilling` is true until the index is ready for use.

During the backfilling phase, you can delete the index that is being created. During this phase, you can't add or delete other indexes on the table.

Note

For indexes that were created as part of a `CreateTable` operation, the `Backfilling` attribute does not appear in the `DescribeTable` output. For more information, see [Phases of Index Creation \(p. 508\)](#).

Detecting and Correcting Index Key Violations

During the backfill phase of global secondary index creation, Amazon DynamoDB examines each item in the table to determine whether it is eligible for inclusion in the index. Some items might not be eligible because they would cause index key violations. In these cases, the items remain in the table, but the index doesn't have a corresponding entry for that item.

An *index key violation* occurs in the following situations:

- There is a data type mismatch between an attribute value and the index key schema data type. For example, suppose that one of the items in the GameScores table had a TopScore value of type String. If you added a global secondary index with a partition key of TopScore, of type Number, the item from the table would violate the index key.
- An attribute value from the table exceeds the maximum length for an index key attribute. The maximum length of a partition key is 2048 bytes, and the maximum length of a sort key is 1024 bytes. If any of the corresponding attribute values in the table exceed these limits, the item from the table would violate the index key.

Note

If a String or Binary attribute value is set for an attribute that is used as an index key, then the attribute value must have a length greater than zero; otherwise, the item from the table would violate the index key.

This tool does not flag this index key violation, at this time.

If an index key violation occurs, the backfill phase continues without interruption. However, any violating items are not included in the index. After the backfill phase completes, all writes to items that violate the new index's key schema will be rejected.

To identify and fix attribute values in a table that violate an index key, use the Violation Detector tool. To run Violation Detector, you create a configuration file that specifies the name of a table to be scanned, the names and data types of the global secondary index partition key and sort key, and what actions to take if any index key violations are found. Violation Detector can run in one of two different modes:

- **Detection mode** — Detect index key violations. Use detection mode to report the items in the table that would cause key violations in a global secondary index. (You can optionally request that these violating table items be deleted immediately when they are found.) The output from detection mode is written to a file, which you can use for further analysis.
- **Correction mode** — Correct index key violations. In correction mode, Violation Detector reads an input file with the same format as the output file from detection mode. Correction mode reads the records from the input file and, for each record, it either deletes or updates the corresponding items in the table. (Note that if you choose to update the items, you must edit the input file and set appropriate values for these updates.)

Downloading and Running Violation Detector

Violation Detector is available as an executable Java Archive (.jar file), and runs on Windows, macOS, or Linux computers. Violation Detector requires Java 1.7 (or later) and Apache Maven.

- [Download Violation Detector from GitHub](#)

Follow the instructions in the README.md file to download and install Violation Detector using Maven.

To start Violation Detector, go to the directory where you have built ViolationDetector.java and enter the following command.

```
java -jar ViolationDetector.jar [options]
```

The Violation Detector command line accepts the following options:

- `-h | --help` — Prints a usage summary and options for Violation Detector.
- `-p | --configFilePath value` — The fully qualified name of a Violation Detector configuration file. For more information, see [The Violation Detector Configuration File \(p. 512\)](#).

- **-t | --detect value** — Detect index key violations in the table, and write them to the Violation Detector output file. If the value of this parameter is set to `keep`, items with key violations are not modified. If the value is set to `delete`, items with key violations are deleted from the table.
- **-c | --correct value** — Read index key violations from an input file, and take corrective actions on the items in the table. If the value of this parameter is set to `update`, items with key violations are updated with new, non-violating values. If the value is set to `delete`, items with key violations are deleted from the table.

The Violation Detector Configuration File

At runtime, the Violation Detector tool requires a configuration file. The parameters in this file determine which DynamoDB resources that Violation Detector can access, and how much provisioned throughput it can consume. The following table describes these parameters.

| Parameter Name | Description | Required? |
|---------------------------------|--|-----------|
| <code>awsCredentialsFile</code> | <p>The fully qualified name of a file containing your AWS credentials. The credentials file must be in the following format:</p> <pre>accessKey = <i>access_key_id_goes_here</i> secretKey = <i>secret_key_goes_here</i></pre> | Yes |
| <code>dynamoDBRegion</code> | The AWS Region in which the table resides. For example: <code>us-west-2</code> . | Yes |
| <code>tableName</code> | The name of the DynamoDB table to be scanned. | Yes |
| <code>gsiHashKeyName</code> | The name of the index partition key. | Yes |
| <code>gsiHashKeyType</code> | <p>The data type of the index partition key—<code>String</code>, <code>Number</code>, or <code>Binary</code>:</p> <p>S N B</p> | Yes |
| <code>gsiRangeKeyName</code> | The name of the index sort key. Do not specify this parameter if the index only has a simple primary key (partition key). | No |
| <code>gsiRangeKeyType</code> | <p>The data type of the index sort key—<code>String</code>, <code>Number</code>, or <code>Binary</code>:</p> <p>S N B</p> <p>Do not specify this parameter if the index only has a simple primary key (partition key).</p> | No |

| Parameter Name | Description | Required? |
|--|--|-----------|
| <code>recordDetails</code> | Whether to write the full details of index key violations to the output file. If set to <code>true</code> (the default), full information about the violating items is reported. If set to <code>false</code> , only the number of violations is reported. | No |
| <code>recordGsiValueInViolationReport</code> | Whether to write the values of the violating index keys to the output file. If set to <code>true</code> (default), the key values are reported. If set to <code>false</code> , the key values are not reported. | No |
| <code>detectionOutputPath</code> | <p>The full path of the Violation Detector output file. This parameter supports writing to a local directory or to Amazon Simple Storage Service (Amazon S3). The following are examples:</p> <pre> detectionOutputPath = //local/path/ filename.csv detectionOutputPath = s3://bucket/filename.csv </pre> <p>Information in the output file appears in comma-separated values (CSV) format. If you don't set <code>detectionOutputPath</code>, the output file is named <code>violation_detection.csv</code> and is written to your current working directory.</p> | No |

| Parameter Name | Description | Required? |
|----------------------|--|-----------|
| numOfSegments | <p>The number of parallel scan segments to be used when Violation Detector scans the table. The default value is 1, meaning that the table is scanned in a sequential manner. If the value is 2 or higher, then Violation Detector divides the table into that many logical segments and an equal number of scan threads.</p> <p>The maximum setting for <code>numOfSegments</code> is 4096.</p> <p>For larger tables, a parallel scan is generally faster than a sequential scan. In addition, if the table is large enough to span multiple partitions, a parallel scan distributes its read activity evenly across multiple partitions.</p> <p>For more information about parallel scans in DynamoDB, see Parallel Scan (p. 479).</p> | No |
| numOfViolations | The upper limit of index key violations to write to the output file. If set to -1 (the default), the entire table is scanned. If set to a positive integer, then Violation Detector stops after it encounters that number of violations. | No |
| numOfRecords | The number of items in the table to be scanned. If set to -1 (the default), the entire table is scanned. If set to a positive integer, Violation Detector stops after it scans that many items in the table. | No |
| readWriteIOPSPercent | Regulates the percentage of provisioned read capacity units that are consumed during the table scan. Valid values range from 1 to 100. The default value (25) means that Violation Detector will consume no more than 25% of the table's provisioned read throughput. | No |

| Parameter Name | Description | Required? |
|----------------------|--|-----------|
| correctionInputPath | <p>The full path of the Violation Detector correction input file. If you run Violation Detector in correction mode, the contents of this file are used to modify or delete data items in the table that violate the global secondary index.</p> <p>The format of the <code>correctionInputPath</code> file is the same as that of the <code>detectionOutputPath</code> file. This lets you process the output from detection mode as input in correction mode.</p> | No |
| correctionOutputPath | <p>The full path of the Violation Detector correction output file. This file is created only if there are update errors.</p> <p>This parameter supports writing to a local directory or to Amazon S3. The following are examples:</p> <pre>correctionOutputPath = //local/path/ filename.csv</pre> <pre>correctionOutputPath = s3://bucket/filename.csv</pre> <p>Information in the output file appears in CSV format. If you don't set <code>correctionOutputPath</code>, the output file is named <code>violation_update_errors.csv</code> and is written to your current working directory.</p> | No |

Detection

To detect index key violations, use Violation Detector with the `--detect` command line option. To show how this option works, consider the `ProductCatalog` table shown in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). The following is a list of items in the table. Only the primary key (`Id`) and the `Price` attribute are shown.

| Id (Primary Key) | Price |
|-------------------------|--------------|
| 101 | 5 |
| 102 | 20 |

| Id (Primary Key) | Price |
|-------------------------|--------------|
| 103 | 200 |
| 201 | 100 |
| 202 | 200 |
| 203 | 300 |
| 204 | 400 |
| 205 | 500 |

All of the values for `Price` are of type `Number`. However, because DynamoDB is schemaless, it is possible to add an item with a non-numeric `Price`. For example, suppose that you add another item to the `ProductCatalog` table.

| Id (Primary Key) | Price |
|-------------------------|--------------|
| 999 | "Hello" |

The table now has a total of nine items.

Now you add a new global secondary index to the table: `PriceIndex`. The primary key for this index is a partition key, `Price`, which is of type `Number`. After the index has been built, it will contain eight items—but the `ProductCatalog` table has nine items. The reason for this discrepancy is that the value "Hello" is of type `String`, but `PriceIndex` has a primary key of type `Number`. The `String` value violates the global secondary index key, so it is not present in the index.

To use Violation Detector in this scenario, you first create a configuration file such as the following.

```
# Properties file for violation detection tool configuration.
# Parameters that are not specified will use default values.

awsCredentialsFile = /home/alice/credentials.txt
dynamoDBRegion = us-west-2
tableName = ProductCatalog
gsiHashKeyName = Price
gsiHashKeyType = N
recordDetails = true
recordGsiValueInViolationRecord = true
detectionOutputPath = ./gsi_violation_check.csv
correctionInputPath = ./gsi_violation_check.csv
numOfSegments = 1
readWriteIOPSPercent = 40
```

Next, you run Violation Detector as in the following example.

```
$ java -jar ViolationDetector.jar --configFilePath config.txt --detect keep

Violation detection started: sequential scan, Table name: ProductCatalog, GSI name:
PriceIndex
Progress: Items scanned in total: 9,    Items scanned by this thread: 9,    Violations
found by this thread: 1, Violations deleted by this thread: 0
Violation detection finished: Records scanned: 9, Violations found: 1, Violations deleted:
0, see results at: ./gsi_violation_check.csv
```

If the `recordDetails` config parameter is set to `true`, Violation Detector writes details of each violation to the output file, as in the following example.

```
Table Hash Key,GSI Hash Key Value,GSI Hash Key Violation Type,GSI Hash Key Violation Description,GSI Hash Key Update Value(FOR USER),Delete Blank Attributes When Updating?(Y/N)  
999,"{""S"":""Hello""}",Type Violation,Expected: N Found: S,,
```

The output file is in CSV format. The first line in the file is a header, followed by one record per item that violates the index key. The fields of these violation records are as follows:

- **Table Hash Key** — The partition key value of the item in the table.
- **Table Range Key** — The sort key value of the item in the table.
- **GSI Hash Key Value** — The partition key value of the global secondary index.
- **GSI Hash Key Violation Type** — Either `Type Violation` or `Size Violation`.
- **GSI Hash Key Violation Description** — The cause of the violation.
- **GSI Hash Key Update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **GSI Range Key Value** — The sort key value of the global secondary index.
- **GSI Range Key Violation Type** — Either `Type Violation` or `Size Violation`.
- **GSI Range Key Violation Description** — The cause of the violation.
- **GSI Range Key Update Value(FOR USER)** — In correction mode, a new user-supplied value for the attribute.
- **Delete Blank Attribute When Updating(Y/N)** — In correction mode, determines whether to delete (Y) or keep (N) the violating item in the table—but only if either of the following fields are blank:
 - `GSI Hash Key Update Value(FOR USER)`
 - `GSI Range Key Update Value(FOR USER)`

If either of these fields are non-blank, then `Delete Blank Attribute When Updating(Y/N)` has no effect.

Note

The output format might vary, depending on the configuration file and command line options. For example, if the table has a simple primary key (without a sort key), no sort key fields will be present in the output.

The violation records in the file might not be in sorted order.

Correction

To correct index key violations, use Violation Detector with the `--correct` command line option. In correction mode, Violation Detector reads the input file specified by the `correctionInputPath` parameter. This file has the same format as the `detectionOutputPath` file, so that you can use the output from detection as input for correction.

Violation Detector provides two different ways to correct index key violations:

- **Delete violations** — Delete the table items that have violating attribute values.
- **Update violations** — Update the table items, replacing the violating attributes with non-violating values.

In either case, you can use the output file from detection mode as input for correction mode.

Continuing with the `ProductCatalog` example, suppose that you want to delete the violating item from the table. To do this, you use the following command line.

```
$ java -jar ViolationDetector.jar --configFilePath config.txt --correct delete
```

At this point, you are asked to confirm whether you want to delete the violating items.

```
Are you sure to delete all violations on the table?y/n
y
Confirmed, will delete violations on the table...
Violation correction from file started: Reading records from file: ./gsi_violation_check.csv, will delete these records from table.
Violation correction from file finished: Violations delete: 1, Violations Update: 0
```

Now both `ProductCatalog` and `PriceIndex` have the same number of items.

Working with Global Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations.

1. Create an instance of the `DynamoDB` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a Table with a Global Secondary Index \(p. 518\)](#)
- [Describe a Table with a Global Secondary Index \(p. 520\)](#)
- [Query a Global Secondary Index \(p. 520\)](#)
- [Example: Global Secondary Indexes Using the AWS SDK for Java Document API \(p. 521\)](#)

Create a Table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following Java code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the DynamoDB document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.

3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because `Precipitation` is not a key attribute for the table, it is not required. However, `WeatherData` items without `Precipitation` do not appear in `PrecipIndex`.

```

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

// Attribute definitions
ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<AttributeDefinition>();

attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Location")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Date")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Precipitation")
    .withAttributeType("N"));

// Table key schema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Location")
    .withKeyType(KeyType.HASH)); //Partition key
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.RANGE)); //Sort key

// PrecipIndex
GlobalSecondaryIndex precipIndex = new GlobalSecondaryIndex()
    .withIndexName("PrecipIndex")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 10)
        .withWriteCapacityUnits((long) 1))
    .withProjection(new Projection().withProjectionType(ProjectionType.ALL));

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();

indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.HASH)); //Partition key
indexKeySchema.add(new KeySchemaElement()
    .withAttributeName("Precipitation")
    .withKeyType(KeyType.RANGE)); //Sort key

precipIndex.setKeySchema(indexKeySchema);

CreateTableRequest createTableRequest = new CreateTableRequest()
    .withTableName("WeatherData")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 5)
        .withWriteCapacityUnits((long) 1))
    .withAttributeDefinitions(attributeDefinitions)
    .withKeySchema(tableKeySchema)
    .withGlobalSecondaryIndexes(precipIndex);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());

```

You must wait until DynamoDB creates the table and sets the table status to **ACTIVE**. After that, you can begin putting data items into the table.

Describe a Table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information a table.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the index you want to work with.
3. Call the `describe` method on the `Table` object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
TableDescription tableDesc = table.describe();

Iterator<GlobalSecondaryIndexDescription> gsiIter =
    tableDesc.getGlobalSecondaryIndexes().iterator();
while (gsiIter.hasNext()) {
    GlobalSecondaryIndexDescription gsiDesc = gsiIter.next();
    System.out.println("Info for index "
        + gsiDesc.getIndexName() + ":");

    Iterator<KeySchemaElement> kseIter = gsiDesc.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
    Projection projection = gsiDesc.getProjection();
    System.out.println("\tThe projection type is: "
        + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: "
            + projection.getNonKeyAttributes());
    }
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class to represent the index you want to work with.

3. Create an instance of the `Index` class for the index you want to query.
4. Call the `query` method on the `Index` object.

The attribute name `Date` is a DynamoDB reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

Table table = dynamoDB.getTable("WeatherData");
Index index = table.getIndex("PrecipIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("#d = :v_date and Precipitation = :v_precip")
    .withNameMap(new NameMap()
        .with("#d", "Date"))
    .withValueMap(new ValueMap()
        .withString(":v_date", "2013-08-10")
        .withNumber(":v_precip", 0));

ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next().toJSONPretty());
}
```

Example: Global Secondary Indexes Using the AWS SDK for Java Document API

The following Java code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following example, see [Java Code Examples \(p. 330\)](#).

Example

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
```

```

/*
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples.document;

import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class DocumentAPIGlobalSecondaryIndexExample {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    public static String tableName = "Issues";

    public static void main(String[] args) throws Exception {
        createTable();
        loadData();

        queryIndex("CreateDateIndex");
        queryIndex("TitleIndex");
        queryIndex("DueDateIndex");

        deleteTable(tableName);
    }

    public static void createTable() {
        // Attribute definitions
        ArrayList<AttributeDefinition> attributeDefinitions = new
        ArrayList<AttributeDefinition>();

        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("IssueId").withAttributeType("S"));
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("Title").withAttributeType("S"));
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("CreateDate").withAttributeType("S"));
    }
}

```

```

        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("DueDate").withAttributeType("S"));

        // Key schema for table
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.HASH)); // Partition

        // key
tableKeySchema.add(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.RANGE)); // Sort

        // key

        // Initial provisioned throughput settings for the indexes
ProvisionedThroughput ptIndex = new
ProvisionedThroughput().withReadCapacityUnits(1L)
.withWriteCapacityUnits(1L);

        // CreateDateIndex
GlobalSecondaryIndex createDateIndex = new
GlobalSecondaryIndex().withIndexName("CreateDateIndex")
.withProvisionedThroughput(ptIndex)
.withKeySchema(new
KeySchemaElement().withAttributeName("CreateDate").withKeyType(KeyType.HASH), // Partition

        // key
new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

        // key
.withProjection(
new
Projection().withProjectionType("INCLUDE").withNonKeyAttributes("Description", "Status"));

        // TitleIndex
GlobalSecondaryIndex titleIndex = new
GlobalSecondaryIndex().withIndexName("TitleIndex")
.withProvisionedThroughput(ptIndex)
.withKeySchema(new
KeySchemaElement().withAttributeName("Title").withKeyType(KeyType.HASH), // Partition

        // key
new
KeySchemaElement().withAttributeName("IssueId").withKeyType(KeyType.RANGE)) // Sort

        // key
.withProjection(new Projection().withProjectionType("KEYS_ONLY"));

        // DueDateIndex
GlobalSecondaryIndex dueDateIndex = new
GlobalSecondaryIndex().withIndexName("DueDateIndex")
.withProvisionedThroughput(ptIndex)
.withKeySchema(new
KeySchemaElement().withAttributeName("DueDate").withKeyType(KeyType.HASH)) // Partition

        // key
.withProjection(new Projection().withProjectionType("ALL"));

CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
.withProvisionedThroughput(
new ProvisionedThroughput().withReadCapacityUnits((long)
1).withWriteCapacityUnits((long) 1))
.withAttributeDefinitions(attributeDefinitions).withKeySchema(tableKeySchema)
.withGlobalSecondaryIndexes(createDateIndex, titleIndex, dueDateIndex);

```

```

        System.out.println("Creating table " + tableName + "...");
        dynamoDB.createTable(createTableRequest);

        // Wait for table to become active
        System.out.println("Waiting for " + tableName + " to become ACTIVE...");
        try {
            Table table = dynamoDB.getTable(tableName);
            table.waitForActive();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void queryIndex(String indexName) {

        Table table = dynamoDB.getTable(tableName);

        System.out.println("\n*****");
\n");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        QuerySpec querySpec = new QuerySpec();

        if (indexName == "CreateDateIndex") {
            System.out.println("Issues filed on 2013-11-01");
            querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
                .withValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        }
        else if (indexName == "TitleIndex") {
            System.out.println("Compilation errors");
            querySpec.withKeyConditionExpression("Title = :v_title and
begins_with(IssueId, :v_issue)")
                .withValueMap(new ValueMap().withString(":v_title", "Compilation
error").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        }
        else if (indexName == "DueDateIndex") {
            System.out.println("Items that are due on 2013-11-30");
            querySpec.withKeyConditionExpression("DueDate = :v_date")
                .withValueMap(new ValueMap().withString(":v_date", "2013-11-30"));
            items = index.query(querySpec);
        }
        else {
            System.out.println("\nNo valid index name provided");
            return;
        }

        Iterator<Item> iterator = items.iterator();

        System.out.println("Query: printing results...");

        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}

```

```

public static void deleteTable(String tableName) {
    System.out.println("Deleting table " + tableName + "...");

    Table table = dynamoDB.getTable(tableName);
    table.delete();

    // Wait for table to be deleted
    System.out.println("Waiting for " + tableName + " to be deleted...");
    try {
        table.waitForDelete();
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void loadData() {
    System.out.println("Loading data into table " + tableName + "...");

    // IssueId, Title,
    // Description,
    // CreateDate, LastUpdateDate, DueDate,
    // Priority, Status

    putItem("A-101", "Compilation error", "Can't compile Project X - bad version
number. What does this mean?",
           "2013-11-01", "2013-11-02", "2013-11-10", 1, "Assigned");

    putItem("A-102", "Can't read data file", "The main data file is missing, or the
permissions are incorrect",
           "2013-11-01", "2013-11-04", "2013-11-30", 2, "In progress");

    putItem("A-103", "Test failure", "Functional test of Project X produces errors",
"2013-11-01", "2013-11-02",
           "2013-11-10", 1, "In progress");

    putItem("A-104", "Compilation error", "Variable 'messageCount' was not
initialized.", "2013-11-15",
           "2013-11-16", "2013-11-30", 3, "Assigned");

    putItem("A-105", "Network issue", "Can't ping IP address 127.0.0.1. Please fix
this.", "2013-11-15",
           "2013-11-16", "2013-11-19", 5, "Assigned");
}

public static void putItem(
    String issueId, String title, String description, String createDate, String
lastUpdateDate, String dueDate,
    Integer priority, String status) {

    Table table = dynamoDB.getTable(tableName);

    Item item = new Item().withPrimaryKey("IssueId", issueId).withString("Title",
title)
        .withString("Description", description).withString("CreateDate", createDate)
        .withString("LastUpdateDate", lastUpdateDate).withString("DueDate", dueDate)
        .withNumber("Priority", priority).withString("Status", status);

    table.putItem(item);
}
}

```

Working with Global Secondary Indexes: .NET

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more global secondary indexes, describe the indexes on the table, and perform queries using the indexes. These operations map to the corresponding DynamoDB operations. For more information, see the [Amazon DynamoDB API Reference](#).

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and `QueryRequest` object to query a table or an index.
3. Execute the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a Table with a Global Secondary Index \(p. 526\)](#)
- [Describe a Table with a Global Secondary Index \(p. 528\)](#)
- [Query a Global Secondary Index \(p. 528\)](#)
- [Example: Global Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 529\)](#)

Create a Table with a Global Secondary Index

You can create global secondary indexes at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more global secondary indexes. The following C# code example creates a table to hold information about weather data. The partition key is `Location` and the sort key is `Date`. A global secondary index named `PrecipIndex` allows fast access to precipitation data for various locations.

The following are the steps to create a table with a global secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the global secondary index, you must provide the index name, its provisioned throughput settings, the attribute definitions for the index sort key, the key schema for the index, and the attribute projection.

3. Execute the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (`WeatherData`) with a global secondary index (`PrecipIndex`). The index partition key is `Date` and its sort key is `Precipitation`. All of the table attributes are projected into the index. Users can query this index to obtain weather data for a particular date, optionally sorting the data by precipitation amount.

Because `Precipitation` is not a key attribute for the table, it is not required. However, `WeatherData` items without `Precipitation` do not appear in `PrecipIndex`.

```
client = new AmazonDynamoDBClient();  
string tableName = "WeatherData";
```

```

// Attribute definitions
var attributeDefinitions = new List<AttributeDefinition>()
{
    {new AttributeDefinition{
        AttributeName = "Location",
        AttributeType = "S"}},
    {new AttributeDefinition{
        AttributeName = "Date",
        AttributeType = "S"}},
    {new AttributeDefinition(){
        AttributeName = "Precipitation",
        AttributeType = "N"}}
};

// Table key schema
var tableKeySchema = new List<KeySchemaElement>()
{
    {new KeySchemaElement {
        AttributeName = "Location",
        KeyType = "HASH"}}, //Partition key
    {new KeySchemaElement {
        AttributeName = "Date",
        KeyType = "RANGE"} //Sort key
},
};

// PrecipIndex
var precipIndex = new GlobalSecondaryIndex
{
    IndexName = "PrecipIndex",
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)10,
        WriteCapacityUnits = (long)1
    },
    Projection = new Projection { ProjectionType = "ALL" }
};

var indexKeySchema = new List<KeySchemaElement> {
    {new KeySchemaElement { AttributeName = "Date", KeyType = "HASH"}}, //Partition key
    {new KeySchemaElement{AttributeName = "Precipitation",KeyType = "RANGE"}} //Sort key
};

precipIndex.KeySchema = indexKeySchema;

CreateTableRequest createTableRequest = new CreateTableRequest
{
    TableName = tableName,
    ProvisionedThroughput = new ProvisionedThroughput
    {
        ReadCapacityUnits = (long)5,
        WriteCapacityUnits = (long)1
    },
    AttributeDefinitions = attributeDefinitions,
    KeySchema = tableKeySchema,
    GlobalSecondaryIndexes = { precipIndex }
};

CreateTableResponse response = client.CreateTable(createTableRequest);
Console.WriteLine(response.CreateTableResult.TableDescription.TableName);
Console.WriteLine(response.CreateTableResult.TableDescription.TableStatus);

```

You must wait until DynamoDB creates the table and sets the table status to **ACTIVE**. After that, you can begin putting data items into the table.

Describe a Table with a Global Secondary Index

To get information about global secondary indexes on a table, use `DescribeTable`. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access global secondary index information for a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Execute the `describeTable` method by providing the request object as a parameter.

Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.

- 3.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();
string tableName = "WeatherData";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest { TableName
= tableName});

List<GlobalSecondaryIndexDescription> globalSecondaryIndexes =
response.DescribeTableResult.Table.GlobalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

foreach (GlobalSecondaryIndexDescription gsiDescription in globalSecondaryIndexes) {
    Console.WriteLine("Info for index " + gsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in gsiDescription.KeySchema) {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = gsiDescription.Projection;
    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE")) {
        Console.WriteLine("\t\tThe non-key projected attributes are: "
+ projection.NonKeyAttributes);
    }
}
```

Query a Global Secondary Index

You can use `Query` on a global secondary index, in much the same way you `Query` a table. You need to specify the index name, the query criteria for the index partition key and sort key (if present), and the attributes that you want to return. In this example, the index is `PrecipIndex`, which has a partition key of `Date` and a sort key of `Precipitation`. The index query returns all of the weather data for a particular date, where the precipitation is greater than zero.

The following are the steps to query a global secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.

2. Create an instance of the `QueryRequest` class to provide the request information.
3. Execute the `query` method by providing the request object as a parameter.

The attribute name `Date` is a DynamoDB reserved word. Therefore, you must use an expression attribute name as a placeholder in the `KeyConditionExpression`.

The following C# code example demonstrates the preceding steps.

Example

```
client = new AmazonDynamoDBClient();

QueryRequest queryRequest = new QueryRequest
{
    TableName = "WeatherData",
    IndexName = "PrecipIndex",
    KeyConditionExpression = "#dt = :v_date and Precipitation > :v_precip",
    ExpressionAttributeNames = new Dictionary<String, String> {
        {"#dt", "Date"}
    },
    ExpressionAttributeValues = new Dictionary<string, AttributeValue> {
        {"":v_date", new AttributeValue { S = "2013-08-01" }},
        {"":v_precip", new AttributeValue { N = "0" }}
    },
    ScanIndexForward = true
};

var result = client.Query(queryRequest);

var items = result.Items;
foreach (var currentItem in items)
{
    foreach (string attr in currentItem.Keys)
    {
        Console.Write(attr + "---> ");
        if (attr == "Precipitation")
        {
            Console.WriteLine(currentItem[attr].N);
        }
        else
        {
            Console.WriteLine(currentItem[attr].S);
        }
    }
    Console.WriteLine();
}
```

Example: Global Secondary Indexes Using the AWS SDK for .NET Low-Level API

The following C# code example shows how to work with global secondary indexes. The example creates a table named `Issues`, which might be used in a simple bug tracking system for software development. The partition key is `IssueId` and the sort key is `Title`. There are three global secondary indexes on this table:

- `CreateDateIndex` — The partition key is `CreateDate` and the sort key is `IssueId`. In addition to the table keys, the attributes `Description` and `Status` are projected into the index.
- `TitleIndex` — The partition key is `Title` and the sort key is `IssueId`. No attributes other than the table keys are projected into the index.
- `DueDateIndex` — The partition key is `DueDate`, and there is no sort key. All of the table attributes are projected into the index.

After the `Issues` table is created, the program loads the table with data representing software bug reports. It then queries the data using the global secondary indexes. Finally, the program deletes the `Issues` table.

For step-by-step instructions for testing the following sample, see [.NET Code Examples \(p. 332\)](#).

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.DataModel;  
using Amazon.DynamoDBv2.DocumentModel;  
using Amazon.DynamoDBv2.Model;  
using Amazon.Runtime;  
using Amazon.SecurityToken;  
  
namespace com.amazonaws.codesamples  
{  
    class LowLevelGlobalSecondaryIndexExample  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
        public static String tableName = "Issues";  
  
        public static void Main(string[] args)  
        {  
            CreateTable();  
            LoadData();  
  
            QueryIndex("CreateDateIndex");  
            QueryIndex("TitleIndex");  
            QueryIndex("DueDateIndex");  
  
            DeleteTable(tableName);  
  
            Console.WriteLine("To continue, press enter");  
            Console.Read();  
        }  
  
        private static void CreateTable()  
        {  
            // Attribute definitions  
            var attributeDefinitions = new List<AttributeDefinition>()  
            {  
                {new AttributeDefinition {  
                    AttributeName = "IssueId", AttributeType = "S"  
                }},  
                {new AttributeDefinition {  
                    AttributeName = "Title", AttributeType = "S"  
                }},  
            };  
        }  
    }  
}
```

```

        {new AttributeDefinition {
            AttributeName = "CreateDate", AttributeType = "S"
        }},
        {new AttributeDefinition {
            AttributeName = "DueDate", AttributeType = "S"
        }}
    };

    // Key schema for table
    var tableKeySchema = new List<KeySchemaElement>() {
    {
        new KeySchemaElement {
            AttributeName= "IssueId",
            KeyType = "HASH" //Partition key
        }
    },
    {
        new KeySchemaElement {
            AttributeName = "Title",
            KeyType = "RANGE" //Sort key
        }
    }
};

// Initial provisioned throughput settings for the indexes
var ptIndex = new ProvisionedThroughput
{
    ReadCapacityUnits = 1L,
    WriteCapacityUnits = 1L
};

// CreateDateIndex
var createDateIndex = new GlobalSecondaryIndex()
{
    IndexName = "CreateDateIndex",
    ProvisionedThroughput = ptIndex,
    KeySchema = {
        new KeySchemaElement {
            AttributeName = "CreateDate", KeyType = "HASH" //Partition key
        },
        new KeySchemaElement {
            AttributeName = "IssueId", KeyType = "RANGE" //Sort key
        }
    },
    Projection = new Projection
    {
        ProjectionType = "INCLUDE",
        NonKeyAttributes = {
            "Description", "Status"
        }
    }
};

// TitleIndex
var titleIndex = new GlobalSecondaryIndex()
{
    IndexName = "TitleIndex",
    ProvisionedThroughput = ptIndex,
    KeySchema = {
        new KeySchemaElement {
            AttributeName = "Title", KeyType = "HASH" //Partition key
        },
        new KeySchemaElement {
            AttributeName = "IssueId", KeyType = "RANGE" //Sort key
        }
    },
}

```

```

        Projection = new Projection
        {
            ProjectionType = "KEYS_ONLY"
        };
    }

    // DueDateIndex
    var dueDateIndex = new GlobalSecondaryIndex()
    {
        IndexName = "DueDateIndex",
        ProvisionedThroughput = ptIndex,
        KeySchema = [
            new KeySchemaElement {
                AttributeName = "DueDate",
                KeyType = "HASH" //Partition key
            }
        ],
        Projection = new Projection
        {
            ProjectionType = "ALL"
        };
    };

    var createTableRequest = new CreateTableRequest
    {
        TableName = tableName,
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = (long)1,
            WriteCapacityUnits = (long)1
        },
        AttributeDefinitions = attributeDefinitions,
        KeySchema = tableKeySchema,
        GlobalSecondaryIndexes = [
            createDateIndex, titleIndex, dueDateIndex
        ]
    };

    Console.WriteLine("Creating table " + tableName + "...");
    client.CreateTable(createTableRequest);

    WaitUntilTableReady(tableName);
}

private static void LoadData()
{
    Console.WriteLine("Loading data into table " + tableName + "...");

    // IssueId, Title,
    // Description,
    // CreateDate, LastUpdateDate, DueDate,
    // Priority, Status

    putItem("A-101", "Compilation error",
        "Can't compile Project X - bad version number. What does this mean?",
        "2013-11-01", "2013-11-02", "2013-11-10",
        1, "Assigned");

    putItem("A-102", "Can't read data file",
        "The main data file is missing, or the permissions are incorrect",
        "2013-11-01", "2013-11-04", "2013-11-30",
        2, "In progress");

    putItem("A-103", "Test failure",

```

```

    "Functional test of Project X produces errors",
    "2013-11-01", "2013-11-02", "2013-11-10",
    1, "In progress");

    putItem("A-104", "Compilation error",
        "Variable 'messageCount' was not initialized.",
        "2013-11-15", "2013-11-16", "2013-11-30",
        3, "Assigned");

    putItem("A-105", "Network issue",
        "Can't ping IP address 127.0.0.1. Please fix this.",
        "2013-11-15", "2013-11-16", "2013-11-19",
        5, "Assigned");
}

private static void putItem(
    String issueId, String title,
    String description,
    String createDate, String lastUpdateDate, String dueDate,
    Int32 priority, String status)
{
    Dictionary<String, AttributeValue> item = new Dictionary<string,
    AttributeValue>();

    item.Add("IssueId", new AttributeValue
    {
        S = issueId
    });
    item.Add("Title", new AttributeValue
    {
        S = title
    });
    item.Add("Description", new AttributeValue
    {
        S = description
    });
    item.Add("CreateDate", new AttributeValue
    {
        S = createDate
    });
    item.Add("LastUpdateDate", new AttributeValue
    {
        S = lastUpdateDate
    });
    item.Add("DueDate", new AttributeValue
    {
        S = dueDate
    });
    item.Add("Priority", new AttributeValue
    {
        N = priority.ToString()
    });
    item.Add("Status", new AttributeValue
    {
        S = status
    });

    try
    {
        client.PutItem(new PutItemRequest
        {
            TableName = tableName,
            Item = item
        });
    }
    catch (Exception e)

```



```

        // Select
        queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
    }
    else
    {
        Console.WriteLine("\nNo valid index name provided");
        return;
    }

    queryRequest.KeyConditionExpression = keyConditionExpression;
    queryRequest.ExpressionAttributeValues = expressionAttributeValues;

    var result = client.Query(queryRequest);
    var items = result.Items;
    foreach (var currentItem in items)
    {
        foreach (string attr in currentItem.Keys)
        {
            if (attr == "Priority")
            {
                Console.WriteLine(attr + "----> " + currentItem[attr].N);
            }
            else
            {
                Console.WriteLine(attr + "----> " + currentItem[attr].S);
            }
        }
        Console.WriteLine();
    }
}

private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest
    {
        TableName = tableName
    });
    WaitForTableToDelete(tableName);
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
        catch (ResourceNotFoundException)
        {
            // DescribeTable is eventually consistent. So you might
            // get resource not found. So we handle the potential exception.
        }
    } while (status != "ACTIVE");
}

```

```

        }

        private static void WaitForTableToDelete(string tableName)
        {
            bool tablePresent = true;

            while (tablePresent)
            {
                System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
                try
                {
                    var res = client.DescribeTable(new DescribeTableRequest
                    {
                        TableName = tableName
                    });

                    Console.WriteLine("Table name: {0}, status: {1}",
                        res.Table.TableName,
                        res.Table.TableStatus);
                }
                catch (ResourceNotFoundException)
                {
                    tablePresent = false;
                }
            }
        }
    }
}

```

Local Secondary Indexes

Some applications only need to query data using the base table's primary key. However, there might be situations where an alternative sort key would be helpful. To give your application a choice of sort keys, you can create one or more local secondary indexes on an Amazon DynamoDB table and issue `Query` or `Scan` requests against these indexes.

Topics

- [Scenario: Using a Local Secondary Index \(p. 536\)](#)
- [Attribute Projections \(p. 539\)](#)
- [Creating a Local Secondary Index \(p. 540\)](#)
- [Querying a Local Secondary Index \(p. 541\)](#)
- [Scanning a Local Secondary Index \(p. 541\)](#)
- [Item Writes and Local Secondary Indexes \(p. 542\)](#)
- [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 542\)](#)
- [Storage Considerations for Local Secondary Indexes \(p. 544\)](#)
- [Item Collections \(p. 544\)](#)
- [Working with Local Secondary Indexes: Java \(p. 547\)](#)
- [Working with Local Secondary Indexes: .NET \(p. 556\)](#)
- [Working with Local Secondary Indexes: AWS CLI \(p. 571\)](#)

Scenario: Using a Local Secondary Index

As an example, consider the `Thread` table that is defined in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). This table is useful for an application such as the [AWS Discussion](#)

[Forums](#). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)

| Thread | | | | |
|-----------|---------|-----------------------|---------|-----|
| ForumName | Subject | LastPostDateTime | Replies | |
| "S3" | "aaa" | "2015-03-15:17:24:31" | 12 | ... |
| "S3" | "bbb" | "2015-01-22:23:18:01" | 3 | ... |
| "S3" | "ccc" | "2015-02-31:13:14:21" | 4 | ... |
| "S3" | "ddd" | "2015-01-03:09:21:11" | 9 | ... |
| "EC2" | "yyy" | "2015-02-12:11:07:56" | 18 | ... |
| "EC2" | "zzz" | "2015-01-18:07:33:42" | 0 | ... |
| "RDS" | "rrr" | "2015-01-19:01:13:24" | 3 | ... |
| "RDS" | "sss" | "2015-03-11:06:53:00" | 11 | ... |
| "RDS" | "ttt" | "2015-10-22:12:19:44" | 5 | ... |
| ... | ... | ... | ... | ... |

DynamoDB stores all of the items with the same partition key value contiguously. In this example, given a particular `ForumName`, a `Query` operation could immediately locate all of the threads for that forum. Within a group of items with the same partition key value, the items are sorted by sort key value. If the sort key (`Subject`) is also provided in the query, DynamoDB can narrow down the results that are returned—for example, returning all of the threads in the "S3" forum that have a `Subject` beginning with the letter "a".

Some requests might require more complex data access patterns. For example:

- Which forum threads get the most views and replies?
- Which thread in a particular forum has the largest number of messages?
- How many threads were posted in a particular forum within a particular time period?

To answer these questions, the `Query` action would not be sufficient. Instead, you would have to `Scan` the entire table. For a table with millions of items, this would consume a large amount of provisioned read throughput and take a long time to complete.

However, you can specify one or more local secondary indexes on non-key attributes, such as `Replies` or `LastPostDateTime`.

A *local secondary index* maintains an alternate sort key for a given partition key value. A local secondary index also contains a copy of some or all of the attributes from its base table. You specify which attributes are projected into the local secondary index when you create the table. The data in a local secondary index is organized by the same partition key as the base table, but with a different sort key. This lets you access data items efficiently across this different dimension. For greater query or scan flexibility, you can create up to five local secondary indexes per table.

Suppose that an application needs to find all of the threads that have been posted within the last three months in a particular forum. Without a local secondary index, the application would have to `Scan` the entire `Thread` table and discard any posts that were not within the specified time frame. With a local secondary index, a `Query` operation could use `LastPostDateTime` as a sort key and find the data quickly.

The following diagram shows a local secondary index named `LastPostIndex`. Note that the partition key is the same as that of the `Thread` table, but the sort key is `LastPostDateTime`.

`LastPostIndex`

| <i>ForumName</i> | <i>LastPostDateTime</i> | <i>Subject</i> |
|------------------|-------------------------|----------------|
| “S3” | “2015-01-03:09:21:11” | “ddd” |
| | “2015-01-22:23:18:01” | “bbb” |
| | “2015-02-31:13:14:21” | “ccc” |
| | “2015-03-15:17:24:31” | “aaa” |
| “EC2” | “2015-01-18:07:33:42” | “zzz” |
| | “2015-02-12:11:07:56” | “yyy” |
| “RDS” | “2015-01-19:01:13:24” | “rr” |
| | “2015-02-22:12:19:44” | “ttt” |
| | “2015-03-11:06:53:00” | “sss” |
| *** | *** | *** |

Every local secondary index must meet the following conditions:

- The partition key is the same as that of its base table.
- The sort key consists of exactly one scalar attribute.
- The sort key of the base table is projected into the index, where it acts as a non-key attribute.

In this example, the partition key is `ForumName` and the sort key of the local secondary index is `LastPostDateTime`. In addition, the sort key value from the base table (in this example, `Subject`) is projected into the index, but it is not a part of the index key. If an application needs a list that is based on `ForumName` and `LastPostDateTime`, it can issue a `Query` request against `LastPostIndex`. The query results are sorted by `LastPostDateTime`, and can be returned in ascending or descending order. The query can also apply key conditions, such as returning only items that have a `LastPostDateTime` within a particular time span.

Every local secondary index automatically contains the partition and sort keys from its base table; you can optionally project non-key attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently. When you query a local secondary index, the query can also retrieve attributes that are *not* projected into the index. DynamoDB automatically fetches these attributes from the base table, but at a greater latency and with higher provisioned throughput costs.

For any local secondary index, you can store up to 10 GB of data per distinct partition key value. This figure includes all of the items in the base table, plus all of the items in the indexes, that have the same partition key value. For more information, see [Item Collections \(p. 544\)](#).

Attribute Projections

With `LastPostIndex`, an application could use `ForumName` and `LastPostDateTime` as query criteria. However, to retrieve any additional attributes, DynamoDB must perform additional read operations against the `Threads` table. These extra reads are known as *fetches*, and they can increase the total amount of provisioned throughput required for a query.

Suppose that you wanted to populate a webpage with a list of all the threads in "S3" and the number of replies for each thread, sorted by the last reply date/time beginning with the most recent reply. To populate this list, you would need the following attributes:

- `Subject`
- `Replies`
- `LastPostDateTime`

The most efficient way to query this data and to avoid fetch operations would be to project the `Replies` attribute from the table into the local secondary index, as shown in this diagram.

LastPostIndex

| <i>ForumName</i> | <i>LastPostDateTime</i> | <i>Subject</i> | <i>Replies</i> |
|------------------|-------------------------|----------------|----------------|
| "S3" | "2015-01-03:09:21:11" | "ddd" | 9 |
| "S3" | "2015-01-22:23:18:01" | "bbb" | 3 |
| "S3" | "2015-02-31:13:14:21" | "ccc" | 4 |
| "S3" | "2015-03-15:17:24:31" | "aaa" | 12 |
| "EC2" | "2015-01-18:07:33:42" | "zzz" | 0 |
| "EC2" | "2015-02-12:11:07:56" | "yyy" | 18 |
| "RDS" | "2015-01-19:01:13:24" | "rrr" | 3 |
| "RDS" | "2015-02-22:12:19:44" | "ttt" | 5 |
| "RDS" | "2015-03-11:06:53:00" | "sss" | 11 |
| ... | ... | ... | ... |

A *projection* is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, you need to specify the attributes that will be projected into the index. DynamoDB provides three different options for this:

- **KEYS_ONLY** – Each item in the index consists only of the table partition key and sort key values, plus the index key values. The **KEYS_ONLY** option results in the smallest possible secondary index.
- **INCLUDE** – In addition to the attributes described in **KEYS_ONLY**, the secondary index will include other non-key attributes that you specify.
- **ALL** – The secondary index includes all of the attributes from the source table. Because all of the table data is duplicated in the index, an **ALL** projection results in the largest possible secondary index.

In the previous diagram, the non-key attribute `Replies` is projected into `LastPostIndex`. An application can query `LastPostIndex` instead of the full `Thread` table to populate a webpage with `Subject`, `Replies`, and `LastPostDateTime`. If any other non-key attributes are requested, DynamoDB would need to fetch those attributes from the `Thread` table.

From an application's point of view, fetching additional attributes from the base table is automatic and transparent, so there is no need to rewrite any application logic. However, such fetching can greatly reduce the performance advantage of using a local secondary index.

When you choose the attributes to project into a local secondary index, you must consider the tradeoff between provisioned throughput costs and storage costs:

- If you need to access just a few attributes with the lowest possible latency, consider projecting only those attributes into a local secondary index. The smaller the index, the less that it costs to store it, and the less your write costs are. If there are attributes that you occasionally need to fetch, the cost for provisioned throughput may well outweigh the longer-term cost of storing those attributes.
- If your application frequently accesses some non-key attributes, you should consider projecting those attributes into a local secondary index. The additional storage costs for the local secondary index offset the cost of performing frequent table scans.
- If you need to access most of the non-key attributes on a frequent basis, you can project these attributes—or even the entire base table—into a local secondary index. This gives you maximum flexibility and lowest provisioned throughput consumption, because no fetching would be required. However, your storage cost would increase, or even double if you are projecting all attributes.
- If your application needs to query a table infrequently, but must perform many writes or updates against the data in the table, consider projecting **KEYS_ONLY**. The local secondary index would be of minimal size, but would still be available when needed for query activity.

Creating a Local Secondary Index

To create one or more local secondary indexes on a table, use the `LocalSecondaryIndexes` parameter of the `CreateTable` operation. Local secondary indexes on a table are created when the table is created. When you delete a table, any local secondary indexes on that table are also deleted.

You must specify one non-key attribute to act as the sort key of the local secondary index. The attribute that you choose must be a scalar `String`, `Number`, or `Binary`. Other scalar types, document types, and set types are not allowed. For a complete list of data types, see [Data Types \(p. 13\)](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see [Item Collection Size Limit \(p. 546\)](#).

You can project attributes of any data type into a local secondary index. This includes scalars, documents, and sets. For a complete list of data types, see [Data Types \(p. 13\)](#).

Querying a Local Secondary Index

In a DynamoDB table, the combined partition key value and sort key value for each item must be unique. However, in a local secondary index, the sort key value does not need to be unique for a given partition key value. If there are multiple items in the local secondary index that have the same sort key value, a `Query` operation returns all of the items that have the same partition key value. In the response, the matching items are not returned in any particular order.

You can query a local secondary index using either eventually consistent or strongly consistent reads. To specify which type of consistency you want, use the `ConsistentRead` parameter of the `Query` operation. A strongly consistent read from a local secondary index always returns the latest updated values. If the query needs to fetch additional attributes from the base table, those attributes will be consistent with respect to the index.

Example

Consider the following data returned from a `Query` that requests data from the discussion threads in a particular forum.

```
{  
    "TableName": "Thread",  
    "IndexName": "LastPostIndex",  
    "ConsistentRead": false,  
    "ProjectionExpression": "Subject, LastPostDateTime, Replies, Tags",  
    "KeyConditionExpression":  
        "ForumName = :v_forum and LastPostDateTime between :v_start and :v_end",  
    "ExpressionAttributeValues": {  
        ":v_start": {"S": "2015-08-31T00:00:00.000Z"},  
        ":v_end": {"S": "2015-11-31T00:00:00.000Z"},  
        ":v_forum": {"S": "EC2"}  
    }  
}
```

In this query:

- DynamoDB accesses `LastPostIndex`, using the `ForumName` partition key to locate the index items for "EC2". All of the index items with this key are stored adjacent to each other for rapid retrieval.
- Within this forum, DynamoDB uses the index to look up the keys that match the specified `LastPostDateTime` condition.
- Because the `Replies` attribute is projected into the index, DynamoDB can retrieve this attribute without consuming any additional provisioned throughput.
- The `Tags` attribute is not projected into the index, so DynamoDB must access the `Thread` table and fetch this attribute.
- The results are returned, sorted by `LastPostDateTime`. The index entries are sorted by partition key value and then by sort key value, and `Query` returns them in the order they are stored. (You can use the `ScanIndexForward` parameter to return the results in descending order.)

Because the `Tags` attribute is not projected into the local secondary index, DynamoDB must consume additional read capacity units to fetch this attribute from the base table. If you need to run this query often, you should project `Tags` into `LastPostIndex` to avoid fetching from the base table. However, if you needed to access `Tags` only occasionally, the additional storage cost for projecting `Tags` into the index might not be worthwhile.

Scanning a Local Secondary Index

You can use `Scan` to retrieve all of the data from a local secondary index. You must provide the base table name and the index name in the request. With a `Scan`, DynamoDB reads all of the data in the index

and returns it to the application. You can also request that only some of the data be returned, and that the remaining data should be discarded. To do this, use the `FilterExpression` parameter of the `Scan` API. For more information, see [Filter Expressions for Scan \(p. 476\)](#).

Item Writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective base tables. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

When you create a local secondary index, you specify an attribute to serve as the sort key for the index. You also specify a data type for that attribute. This means that whenever you write an item to the base table, if the item defines an index key attribute, its type must match the index key schema's data type. In the case of `LastPostIndex`, the `LastPostDateTime` sort key in the index is defined as a `String` data type. If you try to add an item to the `Thread` table and specify a different data type for `LastPostDateTime` (such as `Number`), DynamoDB returns a `ValidationException` because of the data type mismatch.

There is no requirement for a one-to-one relationship between the items in a base table and the items in a local secondary index. In fact, this behavior can be advantageous for many applications.

A table with many local secondary indexes incurs higher costs for write activity than tables with fewer indexes. For more information, see [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 542\)](#).

Important

For tables with local secondary indexes, there is a 10 GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB. For more information, see [Item Collection Size Limit \(p. 546\)](#).

Provisioned Throughput Considerations for Local Secondary Indexes

When you create a table in DynamoDB, you provision read and write capacity units for the table's expected workload. That workload includes read and write activity on the table's local secondary indexes.

To view the current rates for provisioned throughput capacity, see [Amazon DynamoDB pricing](#).

Read Capacity Units

When you query a local secondary index, the number of read capacity units consumed depends on how the data is accessed.

As with table queries, an index query can use either eventually consistent or strongly consistent reads depending on the value of `ConsistentRead`. One strongly consistent read consumes one read capacity unit; an eventually consistent read consumes only half of that. Thus, by choosing eventually consistent reads, you can reduce your read capacity unit charges.

For index queries that request only index keys and projected attributes, DynamoDB calculates the provisioned read activity in the same way as it does for queries against tables. The only difference is that the calculation is based on the sizes of the index entries, rather than the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned; the result is then rounded up to the next 4 KB boundary. For more information about how DynamoDB calculates provisioned throughput usage, see [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#).

For index queries that read attributes that are not projected into the local secondary index, DynamoDB needs to fetch those attributes from the base table, in addition to reading the projected attributes from the index. These fetches occur when you include any non-projected attributes in the `Select` or `ProjectionExpression` parameters of the `Query` operation. Fetching causes additional latency in query responses, and it also incurs a higher provisioned throughput cost: In addition to the reads from the local secondary index described previously, you are charged for read capacity units for every base table item fetched. This charge is for reading each entire item from the table, not just the requested attributes.

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all the attribute names and values across all of the items returned. However, if a `Query` against a local secondary index causes DynamoDB to fetch item attributes from the base table, the maximum size of the data in the results might be lower. In this case, the result size is the sum of:

- The size of the matching items in the index, rounded up to the next 4 KB.
- The size of each matching item in the base table, with each item individually rounded up to the next 4 KB.

Using this formula, the maximum size of the results returned by a `Query` operation is still 1 MB.

For example, consider a table where the size of each item is 300 bytes. There is a local secondary index on that table, but only 200 bytes of each item is projected into the index. Now suppose that you `Query` this index, that the query requires table fetches for each item, and that the query returns 4 items. DynamoDB sums up the following:

- The size of the matching items in the index: $200 \text{ bytes} \times 4 \text{ items} = 800 \text{ bytes}$; this is then rounded up to 4 KB.
- The size of each matching item in the base table: $(300 \text{ bytes, rounded up to 4 KB}) \times 4 \text{ items} = 16 \text{ KB}$.

The total size of the data in the result is therefore 20 KB.

Write Capacity Units

When an item in a table is added, updated, or deleted, updating the local secondary indexes consumes provisioned write capacity units for the table. The total provisioned throughput cost for a write is the sum of write capacity units consumed by writing to the table and those consumed by updating the local secondary indexes.

The cost of writing an item to a local secondary index depends on several factors:

- If you write a new item to the table that defines an indexed attribute, or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required: one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1 KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries need to return and projecting only those attributes into the index.

Storage Considerations for Local Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any local secondary indexes in which those attributes should appear. Your AWS account is charged for storage of the item in the base table and also for storage of attributes in any local secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- The size in bytes of the base table primary key (partition key and sort key)
- The size in bytes of the index key attribute
- The size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a local secondary index, you can estimate the average size of an item in the index and then multiply by the number of items in the base table.

If a table contains an item where a particular attribute is not defined, but that attribute is defined as an index sort key, then DynamoDB does not write any data for that item to the index.

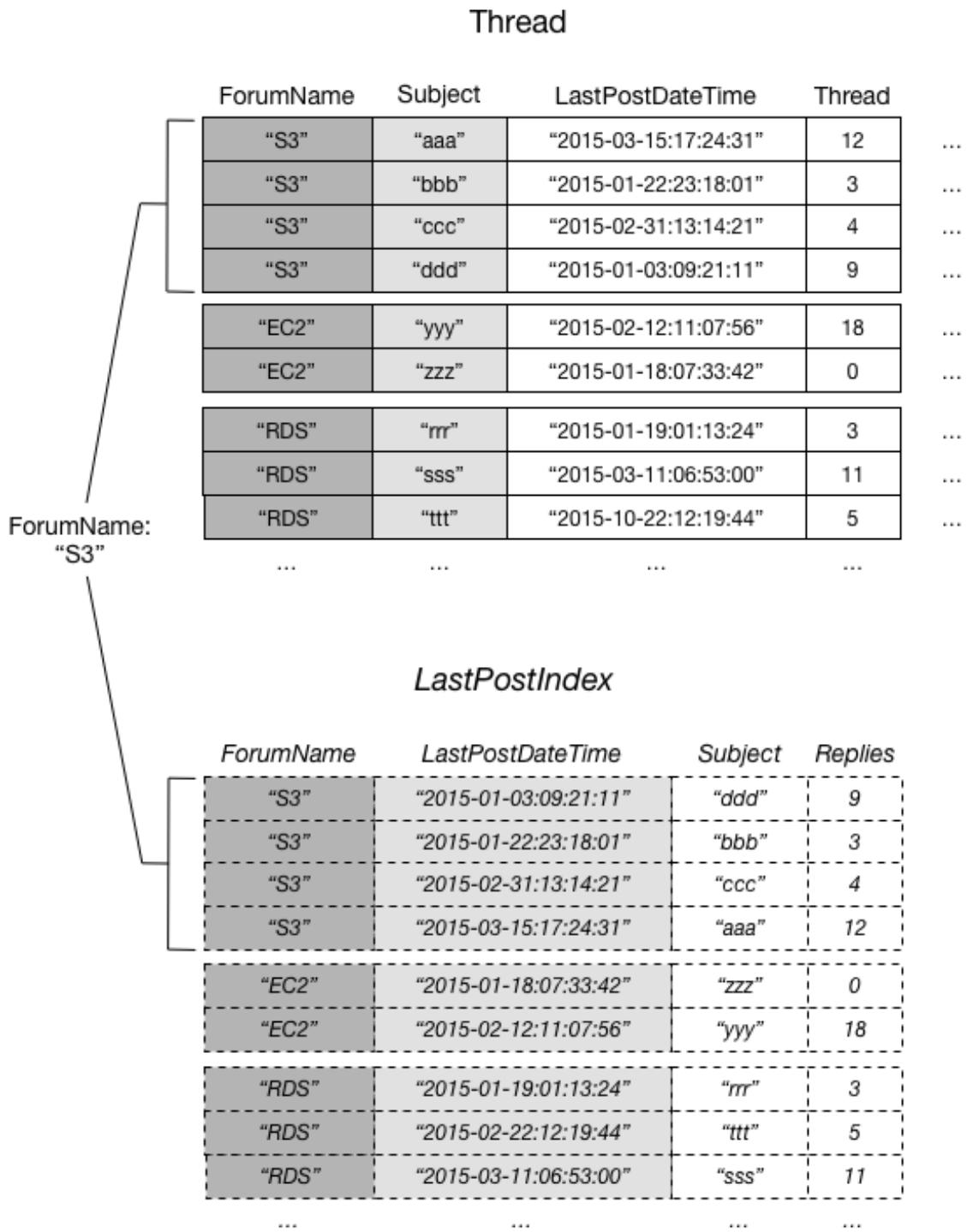
Item Collections

Note

This section pertains only to tables that have local secondary indexes.

In DynamoDB, an *item collection* is any group of items that have the same partition key value in a table and all of its local secondary indexes. In the examples used throughout this section, the partition key for the `Thread` table is `ForumName`, and the partition key for `LastPostIndex` is also `ForumName`. All the table and index items with the same `ForumName` are part of the same item collection. For example, in the `Thread` table and the `LastPostIndex` local secondary index, there is an item collection for forum `EC2` and a different item collection for forum `RDS`.

The following diagram shows the item collection for forum `S3`.



In this diagram, the item collection consists of all the items in `Thread` and `LastPostIndex` where the `ForumName` partition key value is "S3". If there were other local secondary indexes on the table, any items in those indexes with `ForumName` equal to "S3" would also be part of the item collection.

You can use any of the following operations in DynamoDB to return information about item collections:

- `BatchWriteItem`

- `DeleteItem`
- `PutItem`
- `UpdateItem`

Each of these operations supports the `ReturnItemCollectionMetrics` parameter. When you set this parameter to `SIZE`, you can view information about the size of each item collection in the index.

Example

The following is an example from the output of an `UpdateItem` operation on the `Thread` table, with `ReturnItemCollectionMetrics` set to `SIZE`. The item that was updated had a `ForumName` value of "EC2", so the output includes information about that item collection.

```
{  
    ItemCollectionMetrics: {  
        ItemCollectionKey: {  
            ForumName: "EC2"  
        },  
        SizeEstimateRangeGB: [0.0, 1.0]  
    }  
}
```

The `SizeEstimateRangeGB` object shows that the size of this item collection is between 0 and 1 GB. DynamoDB periodically updates this size estimate, so the numbers might be different next time the item is modified.

Item Collection Size Limit

The maximum size of any item collection is 10 GB. This limit does not apply to tables without local secondary indexes. Only tables that have one or more local secondary indexes are affected.

If an item collection exceeds the 10 GB limit, DynamoDB returns an `ItemCollectionSizeLimitExceeded` exception, and you won't be able to add more items to the item collection or increase the sizes of items that are in the item collection. (Read and write operations that shrink the size of the item collection are still allowed.) You can still add items to other item collections.

To reduce the size of an item collection, you can do one of the following:

- Delete any unnecessary items with the partition key value in question. When you delete these items from the base table, DynamoDB also removes any index entries that have the same partition key value.
- Update the items by removing attributes or by reducing the size of the attributes. If these attributes are projected into any local secondary indexes, DynamoDB also reduces the size of the corresponding index entries.
- Create a new table with the same partition key and sort key, and then move items from the old table to the new table. This might be a good approach if a table has historical data that is infrequently accessed. You might also consider archiving this historical data to Amazon Simple Storage Service (Amazon S3).

When the total size of the item collection drops below 10 GB, you can once again add items with the same partition key value.

We recommend as a best practice that you instrument your application to monitor the sizes of your item collections. One way to do so is to set the `ReturnItemCollectionMetrics` parameter to `SIZE` whenever you use `BatchWriteItem`, `DeleteItem`, `PutItem`, or `UpdateItem`. Your application

should examine the `ReturnItemCollectionMetrics` object in the output and log an error message whenever an item collection exceeds a user-defined limit (8 GB, for example). Setting a limit that is less than 10 GB would provide an early warning system so you know that an item collection is approaching the limit in time to do something about it.

Item Collections and Partitions

The table and index data for each item collection is stored in a single partition. Referring to the `Thread` table example, all of the base table and index items with the same `ForumName` attribute would be stored in the same partition. The "S3" item collection would be stored on one partition, "EC2" in another partition, and "RDS" in a third partition.

You should design your applications so that table data is evenly distributed across distinct partition key values. For tables with local secondary indexes, your applications should not create "hot spots" of read and write activity within a single item collection on a single partition.

Working with Local Secondary Indexes: Java

You can use the AWS SDK for Java Document API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes.

The following are the common steps for table operations using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.
3. Call the appropriate method provided by the client that you created in the preceding step.

Topics

- [Create a Table with a Local Secondary Index \(p. 547\)](#)
- [Describe a Table with a Local Secondary Index \(p. 549\)](#)
- [Query a Local Secondary Index \(p. 549\)](#)
- [Example: Local Secondary Indexes Using the Java Document API \(p. 550\)](#)

Create a Table with a Local Secondary Index

Local secondary indexes must be created at the same time you create a table. To do this, use the `createTable` method and provide your specifications for one or more local secondary indexes. The following Java code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the DynamoDB document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type for the index sort key, the key schema for the index, and the attribute projection.

3. Call the `createTable` method by providing the request object as a parameter.

The following Java code example demonstrates the preceding steps. The code creates a table (`Music`) with a secondary index on the `AlbumTitle` attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```

AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

CreateTableRequest createTableRequest = new CreateTableRequest().withTableName(tableName);

//ProvisionedThroughput
createTableRequest.setProvisionedThroughput(new
    ProvisionedThroughput().withReadCapacityUnits((long)5).withWriteCapacityUnits((long)5));

//AttributeDefinitions
ArrayList<AttributeDefinition> attributeDefinitions= new ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("Artist").withAttributeType("S"));
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("SongTitle").withAttributeType("S"));
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("AlbumTitle").withAttributeType("S"));

createTableRequest.setAttributeDefinitions(attributeDefinitions);

//KeySchema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
    KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH)); //Partition
key
tableKeySchema.add(new
    KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE)); //Sort key

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
    KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH)); //Partition
key
indexKeySchema.add(new
    KeySchemaElement().withAttributeName("AlbumTitle").withKeyType(KeyType.RANGE)); //Sort
key

Projection projection = new Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("Genre");
nonKeyAttributes.add("Year");
projection.setNonKeyAttributes(nonKeyAttributes);

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
    .withIndexName("AlbumTitleIndex").withKeySchema(indexKeySchema).withProjection(projection);

ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
    ArrayList<LocalSecondaryIndex>();
localSecondaryIndexes.add(localSecondaryIndex);
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());

```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a Table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describeTable` method. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information a table using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class. You must provide the table name.
3. Call the `describeTable` method on the `Table` object.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);

TableDescription tableDescription = table.describe();

List<LocalSecondaryIndexDescription> localSecondaryIndexes
    = tableDescription.getLocalSecondaryIndexes();

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.

Iterator<LocalSecondaryIndexDescription> lsiIter = localSecondaryIndexes.iterator();
while (lsiIter.hasNext()) {

    LocalSecondaryIndexDescription lsiDescription = lsiIter.next();
    System.out.println("Info for index " + lsiDescription.getIndexName() + ":");
    Iterator<KeySchemaElement> kseIter = lsiDescription.getKeySchema().iterator();
    while (kseIter.hasNext()) {
        KeySchemaElement kse = kseIter.next();
        System.out.printf("\t%s: %s\n", kse.getAttributeName(), kse.getKeyType());
    }
    Projection projection = lsiDescription.getProjection();
    System.out.println("\tThe projection type is: " + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tThe non-key projected attributes are: " +
            projection.getNonKeyAttributes());
    }
}
```

Query a Local Secondary Index

You can use the `Query` operation on a local secondary index in much the same way that you `Query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute Projections \(p. 539\)](#).

The following are the steps to query a local secondary index using the AWS SDK for Java Document API.

1. Create an instance of the `DynamoDB` class.
2. Create an instance of the `Table` class. You must provide the table name.
3. Create an instance of the `Index` class. You must provide the index name.
4. Call the `query` method of the `Index` class.

The following Java code example demonstrates the preceding steps.

Example

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
DynamoDB dynamoDB = new DynamoDB(client);

String tableName = "Music";

Table table = dynamoDB.getTable(tableName);
Index index = table.getIndex("AlbumTitleIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Artist = :v_artist and AlbumTitle = :v_title")
    .WithValueMap(new ValueMap()
        .withString(":v_artist", "Acme Band")
        .withString(":v_title", "Songs About Life"));

ItemCollection<QueryOutcome> items = index.query(spec);

Iterator<Item> itemsIter = items.iterator();

while (itemsIter.hasNext()) {
    Item item = itemsIter.next();
    System.out.println(item.toJSONPretty());
}
```

Example: Local Secondary Indexes Using the Java Document API

The following Java code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
 - `ProductCategory`
 - `ProductName`
 - `OrderStatus`
 - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following sample, see [Java Code Examples \(p. 330\)](#).

Example

```
/***
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
```

```
* This file is licensed under the Apache License, Version 2.0 (the "License").  
* You may not use this file except in compliance with the License. A copy of  
* the License is located at  
*  
* http://aws.amazon.com/apache2.0/  
*  
* This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
* CONDITIONS OF ANY KIND, either express or implied. See the License for the  
* specific language governing permissions and limitations under the License.  
*/  
  
package com.amazonaws.codesamples.document;  
  
import java.util.ArrayList;  
import java.util.Iterator;  
  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Index;  
import com.amazonaws.services.dynamodbv2.document.Item;  
import com.amazonaws.services.dynamodbv2.document.ItemCollection;  
import com.amazonaws.services.dynamodbv2.document.PutItemOutcome;  
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;  
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;  
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;  
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;  
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;  
import com.amazonaws.services.dynamodbv2.model.KeyType;  
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;  
import com.amazonaws.services.dynamodbv2.model.Projection;  
import com.amazonaws.services.dynamodbv2.model.ProjectionType;  
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;  
import com.amazonaws.services.dynamodbv2.model.ReturnConsumedCapacity;  
import com.amazonaws.services.dynamodbv2.model.Select;  
  
public class DocumentAPILocalSecondaryIndexExample {  
  
    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();  
    static DynamoDB dynamoDB = new DynamoDB(client);  
  
    public static String tableName = "CustomerOrders";  
  
    public static void main(String[] args) throws Exception {  
  
        createTable();  
        loadData();  
  
        query(null);  
        query("IsOpenIndex");  
        query("OrderCreationDateIndex");  
  
        deleteTable(tableName);  
    }  
  
    public static void createTable() {  
  
        CreateTableRequest createTableRequest = new  
CreateTableRequest().withTableName(tableName)  
            .withProvisionedThroughput(  
                new ProvisionedThroughput().withReadCapacityUnits((long)  
1).withWriteCapacityUnits((long) 1));  
    }  
}
```

```

        // Attribute definitions for table partition and sort keys
        ArrayList<AttributeDefinition> attributeDefinitions = new
        ArrayList<AttributeDefinition>();
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("CustomerId").withAttributeType("S"));
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("OrderId").withAttributeType("N"));

        // Attribute definition for index primary key attributes
        attributeDefinitions
            .add(new
        AttributeDefinition().withAttributeName("OrderCreationDate").withAttributeType("N"));
        attributeDefinitions.add(new
        AttributeDefinition().withAttributeName("IsOpen").withAttributeType("N"));

        createTableRequest.setAttributeDefinitions(attributeDefinitions);

        // Key schema for table
        ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
        tableKeySchema.add(new
        KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition

        // key
        tableKeySchema.add(new
        KeySchemaElement().withAttributeName("OrderId").withKeyType(KeyType.RANGE)); // Sort

        // key

        createTableRequest.setKeySchema(tableKeySchema);

        ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
        ArrayList<LocalSecondaryIndex>();

        // OrderCreationDateIndex
        LocalSecondaryIndex orderCreationDateIndex = new
        LocalSecondaryIndex().withIndexName("OrderCreationDateIndex");

        // Key schema for OrderCreationDateIndex
        ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
        indexKeySchema.add(new
        KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition

        // key
        indexKeySchema.add(new
        KeySchemaElement().withAttributeName("OrderCreationDate").withKeyType(KeyType.RANGE)); // Sort

        // key

        orderCreationDateIndex.setKeySchema(indexKeySchema);

        // Projection (with list of projected attributes) for
        // OrderCreationDateIndex
        Projection projection = new
        Projection().withProjectionType(ProjectionType.INCLUDE);
        ArrayList<String> nonKeyAttributes = new ArrayList<String>();
        nonKeyAttributes.add("ProductCategory");
        nonKeyAttributes.add("ProductName");
        projection.setNonKeyAttributes(nonKeyAttributes);

        orderCreationDateIndex.setProjection(projection);

        localSecondaryIndexes.add(orderCreationDateIndex);
    
```

```

    // IsOpenIndex
    LocalSecondaryIndex isOpenIndex = new
    LocalSecondaryIndex().withIndexName("IsOpenIndex");

    // Key schema for IsOpenIndex
    indexKeySchema = new ArrayList<KeySchemaElement>();
    indexKeySchema.add(new
    KeySchemaElement().withAttributeName("CustomerId").withKeyType(KeyType.HASH)); // Partition

    // key
    indexKeySchema.add(new
    KeySchemaElement().withAttributeName("IsOpen").withKeyType(KeyType.RANGE)); // Sort

    // key

    // Projection (all attributes) for IsOpenIndex
    projection = new Projection().withProjectionType(ProjectionType.ALL);

    isOpenIndex.setKeySchema(indexKeySchema);
    isOpenIndex.setProjection(projection);

    localSecondaryIndexes.add(isOpenIndex);

    // Add index definitions to CreateTable request
    createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

    System.out.println("Creating table " + tableName + "...");
    System.out.println(dynamoDB.createTable(createTableRequest));

    // Wait for table to become active
    System.out.println("Waiting for " + tableName + " to become ACTIVE...");
    try {
        Table table = dynamoDB.getTable(tableName);
        table.waitForActive();
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void query(String indexName) {

    Table table = dynamoDB.getTable(tableName);

    System.out.println("\n*****\n");
    System.out.println("Querying table " + tableName + "...");

    QuerySpec querySpec = new
    QuerySpec().withConsistentRead(true).withScanIndexForward(true)
        .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

    if (indexName == "IsOpenIndex") {

        System.out.println("\nUsing index: '" + indexName + "' : Bob's orders that are
open.");
        System.out.println("Only a user-specified list of attributes are returned\n");
        Index index = table.getIndex(indexName);

        querySpec.withKeyConditionExpression("CustomerId = :v_custid and IsOpen
= :v_isopen")
            .withValueMap(new ValueMap().withString(":v_custid",
"bob@example.com").withNumber(":v_isopen", 1));
    }
}

```

```

querySpec.withProjectionExpression("OrderCreationDate, ProductCategory,
ProductName, OrderStatus");

    ItemCollection<QueryOutcome> items = index.query(querySpec);
    Iterator<Item> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

}

else if (indexName == "OrderCreationDateIndex") {
    System.out.println("\nUsing index: '" + indexName + "' : Bob's orders that were
placed after 01/31/2015.");
    System.out.println("Only the projected attributes are returned\n");
    Index index = table.getIndex(indexName);

    querySpec.withKeyConditionExpression("CustomerId = :v_custid and
OrderCreationDate >= :v_orddate")
        .withValueMap(
            new ValueMap().withString(":v_custid",
"bob@example.com").withNumber(":v_orddate", 20150131));

    querySpec.withSelect(Select.ALL_PROJECTED_ATTRIBUTES);

    ItemCollection<QueryOutcome> items = index.query(querySpec);
    Iterator<Item> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

}

else {
    System.out.println("\nNo index: All of Bob's orders, by OrderId:\n");

    querySpec.withKeyConditionExpression("CustomerId = :v_custid")
        .withValueMap(new ValueMap().withString(":v_custid", "bob@example.com"));

    ItemCollection<QueryOutcome> items = table.query(querySpec);
    Iterator<Item> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

}

public static void deleteTable(String tableName) {

    Table table = dynamoDB.getTable(tableName);
    System.out.println("Deleting table " + tableName + "...");
    table.delete();

    // Wait for table to be deleted
    System.out.println("Waiting for " + tableName + " to be deleted...");
    try {
        table.waitForDelete();
    }
}

```

```

        }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void loadData() {

    Table table = dynamoDB.getTable(tableName);

    System.out.println("Loading data into table " + tableName + "...");

    Item item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 1)
    .withNumber("IsOpen", 1).withNumber("OrderCreationDate",
20150101).withString("ProductCategory", "Book")
    .withString("ProductName", "The Great Outdoors").withString("OrderStatus",
"PACKING ITEMS");
    // no ShipmentTrackingId attribute

    PutItemOutcome putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 2)
    .withNumber("IsOpen", 1).withNumber("OrderCreationDate",
20150221).withString("ProductCategory", "Bike")
    .withString("ProductName", "Super Mountain").withString("OrderStatus", "ORDER RECEIVED");
    // no ShipmentTrackingId attribute

    putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"alice@example.com").withNumber("OrderId", 3)
    // no IsOpen attribute
    .withNumber("OrderCreationDate", 20150304).withString("ProductCategory",
"Music")
    .withString("ProductName", "A Quiet Interlude").withString("OrderStatus", "IN TRANSIT")
    .withString("ShipmentTrackingId", "176493");

    putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 1)
    // no IsOpen attribute
    .withNumber("OrderCreationDate", 20150111).withString("ProductCategory",
"Movie")
    .withString("ProductName", "Calm Before The Storm").withString("OrderStatus",
"SHIPPING DELAY")
    .withString("ShipmentTrackingId", "859323");

    putItemOutcome = table.putItem(item);

    item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 2)
    // no IsOpen attribute
    .withNumber("OrderCreationDate", 20150124).withString("ProductCategory",
"Music")
    .withString("ProductName", "E-Z Listening").withString("OrderStatus",
"DELIVERED")
    .withString("ShipmentTrackingId", "756943");

    putItemOutcome = table.putItem(item);
}

```

```

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 3)
            // no IsOpen attribute
            .withNumber("OrderCreationDate", 20150221).withString("ProductCategory",
"Music")
            .withString("ProductName", "Symphony 9").withString("OrderStatus", "DELIVERED")
            .withString("ShipmentTrackingId", "645193");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 4)
            .withNumber("IsOpen", 1).withNumber("OrderCreationDate",
20150222).withString("ProductCategory", "Hardware")
            .withString("ProductName", "Extra Heavy Hammer").withString("OrderStatus",
"PACKING ITEMS");
            // no ShipmentTrackingId attribute

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 5)
            /* no IsOpen attribute */
            .withNumber("OrderCreationDate", 20150309).withString("ProductCategory",
"Book")
            .withString("ProductName", "How To Cook").withString("OrderStatus", "IN
TRANSIT")
            .withString("ShipmentTrackingId", "440185");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 6)
            // no IsOpen attribute
            .withNumber("OrderCreationDate", 20150318).withString("ProductCategory",
"Luggage")
            .withString("ProductName", "Really Big Suitcase").withString("OrderStatus",
"DELIVERED")
            .withString("ShipmentTrackingId", "893927");

        putItemOutcome = table.putItem(item);

        item = new Item().withPrimaryKey("CustomerId",
"bob@example.com").withNumber("OrderId", 7)
            /* no IsOpen attribute */
            .withNumber("OrderCreationDate", 20150324).withString("ProductCategory",
"Golf")
            .withString("ProductName", "PGA Pro II").withString("OrderStatus", "OUT FOR
DELIVERY")
            .withString("ShipmentTrackingId", "383283");

        putItemOutcome = table.putItem(item);
        assert putItemOutcome != null;
    }
}

```

Working with Local Secondary Indexes: .NET

Topics

- [Create a Table with a Local Secondary Index \(p. 557\)](#)
- [Describe a Table with a Local Secondary Index \(p. 559\)](#)

- [Query a Local Secondary Index \(p. 559\)](#)
- [Example: Local Secondary Indexes Using the AWS SDK for .NET Low-Level API \(p. 560\)](#)

You can use the AWS SDK for .NET low-level API to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes. These operations map to the corresponding low-level DynamoDB API actions. For more information, see [.NET Code Examples \(p. 332\)](#).

The following are the common steps for table operations using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Provide the required and optional parameters for the operation by creating the corresponding request objects.

For example, create a `CreateTableRequest` object to create a table and an `QueryRequest` object to query a table or an index.

3. Execute the appropriate method provided by the client that you created in the preceding step.

Create a Table with a Local Secondary Index

Local secondary indexes must be created at the same time that you create a table. To do this, use `CreateTable` and provide your specifications for one or more local secondary indexes. The following C# code example creates a table to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex`, facilitates queries by album title.

The following are the steps to create a table with a local secondary index, using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `CreateTableRequest` class to provide the request information.

You must provide the table name, its primary key, and the provisioned throughput values. For the local secondary index, you must provide the index name, the name and data type of the index sort key, the key schema for the index, and the attribute projection.

3. Execute the `CreateTable` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps. The code creates a table (`Music`) with a secondary index on the `AlbumTitle` attribute. The table partition key and sort key, plus the index sort key, are the only attributes projected into the index.

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

CreateTableRequest createTableRequest = new CreateTableRequest()
{
    TableName = tableName
};

//ProvisionedThroughput
createTableRequest.ProvisionedThroughput = new ProvisionedThroughput()
{
    ReadCapacityUnits = (long)5,
    WriteCapacityUnits = (long)5
};

//AttributeDefinitions
```

```

List<AttributeDefinition> attributeDefinitions = new List<AttributeDefinition>();

attributeDefinitions.Add(new AttributeDefinition()
{
    AttributeName = "Artist",
    AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
{
    AttributeName = "SongTitle",
    AttributeType = "S"
});

attributeDefinitions.Add(new AttributeDefinition()
{
    AttributeName = "AlbumTitle",
    AttributeType = "S"
});

createTableRequest.AttributeDefinitions = attributeDefinitions;

//KeySchema
List<KeySchemaElement> tableKeySchema = new List<KeySchemaElement>();

tableKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType = "HASH" });
    //Partition key
tableKeySchema.Add(new KeySchemaElement() { AttributeName = "SongTitle", KeyType =
    "RANGE" });    //Sort key

createTableRequest.KeySchema = tableKeySchema;

List<KeySchemaElement> indexKeySchema = new List<KeySchemaElement>();
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "Artist", KeyType = "HASH" });
    //Partition key
indexKeySchema.Add(new KeySchemaElement() { AttributeName = "AlbumTitle", KeyType =
    "RANGE" });    //Sort key

Projection projection = new Projection() { ProjectionType = "INCLUDE" };

List<string> nonKeyAttributes = new List<string>();
nonKeyAttributes.Add("Genre");
nonKeyAttributes.Add("Year");
projection.NonKeyAttributes = nonKeyAttributes;

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
{
    IndexName = "AlbumTitleIndex",
    KeySchema = indexKeySchema,
    Projection = projection
};

List<LocalSecondaryIndex> localSecondaryIndexes = new List<LocalSecondaryIndex>();
localSecondaryIndexes.Add(localSecondaryIndex);
createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

CreateTableResponse result = client.CreateTable(createTableRequest);
Console.WriteLine(result.CreateTableResult.TableDescription.TableName);
Console.WriteLine(result.CreateTableResult.TableDescription.TableStatus);

```

You must wait until DynamoDB creates the table and sets the table status to ACTIVE. After that, you can begin putting data items into the table.

Describe a Table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `DescribeTable` API. For each index, you can access its name, key schema, and projected attributes.

The following are the steps to access local secondary index information a table using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `DescribeTableRequest` class to provide the request information. You must provide the table name.
3. Execute the `describeTable` method by providing the request object as a parameter.
- 4.

The following C# code example demonstrates the preceding steps.

Example

```
AmazonDynamoDBClient client = new AmazonDynamoDBClient();
string tableName = "Music";

DescribeTableResponse response = client.DescribeTable(new DescribeTableRequest()
{ TableName = tableName });
List<LocalSecondaryIndexDescription> localSecondaryIndexes =
    response.DescribeTableResult.Table.LocalSecondaryIndexes;

// This code snippet will work for multiple indexes, even though
// there is only one index in this example.
foreach (LocalSecondaryIndexDescription lsiDescription in localSecondaryIndexes)
{
    Console.WriteLine("Info for index " + lsiDescription.IndexName + ":");

    foreach (KeySchemaElement kse in lsiDescription.KeySchema)
    {
        Console.WriteLine("\t" + kse.AttributeName + ": key type is " + kse.KeyType);
    }

    Projection projection = lsiDescription.Projection;

    Console.WriteLine("\tThe projection type is: " + projection.ProjectionType);

    if (projection.ProjectionType.ToString().Equals("INCLUDE"))
    {
        Console.WriteLine("\t\tThe non-key projected attributes are:");

        foreach (String s in projection.NonKeyAttributes)
        {
            Console.WriteLine("\t\t\t" + s);
        }
    }
}
```

Query a Local Secondary Index

You can use `Query` on a local secondary index in much the same way you `Query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex`, and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute Projections \(p. 539\)](#)

The following are the steps to query a local secondary index using the .NET low-level API.

1. Create an instance of the `AmazonDynamoDBClient` class.
2. Create an instance of the `QueryRequest` class to provide the request information.
3. Execute the `query` method by providing the request object as a parameter.

The following C# code example demonstrates the preceding steps.

Example

```
QueryRequest queryRequest = new QueryRequest
{
    TableName = "Music",
    IndexName = "AlbumTitleIndex",
    Select = "ALL_ATTRIBUTES",
    ScanIndexForward = true,
    KeyConditionExpression = "Artist = :v_artist and AlbumTitle = :v_title",
    ExpressionAttributeValues = new Dictionary<string, AttributeValue>()
    {
        {":v_artist", new AttributeValue {S = "Acme Band"}},
        {":v_title", new AttributeValue {S = "Songs About Life"}}
    },
};

QueryResponse response = client.Query(queryRequest);

foreach (var attrbs in response.Items)
{
    foreach (var attrib in attrbs)
    {
        Console.WriteLine(attrib.Key + " ---> " + attrib.Value.S);
    }
    Console.WriteLine();
}
```

Example: Local Secondary Indexes Using the AWS SDK for .NET Low-Level API

The following C# code example shows how to work with local secondary indexes in Amazon DynamoDB. The example creates a table named `CustomerOrders` with a partition key of `CustomerId` and a sort key of `OrderId`. There are two local secondary indexes on this table:

- `OrderCreationDateIndex` — The sort key is `OrderCreationDate`, and the following attributes are projected into the index:
 - `ProductCategory`
 - `ProductName`
 - `OrderStatus`
 - `ShipmentTrackingId`
- `IsOpenIndex` — The sort key is `IsOpen`, and all of the table attributes are projected into the index.

After the `CustomerOrders` table is created, the program loads the table with data representing customer orders. It then queries the data using the local secondary indexes. Finally, the program deletes the `CustomerOrders` table.

For step-by-step instructions for testing the following example, see [.NET Code Examples \(p. 332\)](#).

Example

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using Amazon.DynamoDBv2;  
using Amazon.DynamoDBv2.DataModel;  
using Amazon.DynamoDBv2.DocumentModel;  
using Amazon.DynamoDBv2.Model;  
using Amazon.Runtime;  
using Amazon.SecurityToken;  
  
namespace com.amazonaws.codesamples  
{  
    class LowLevelLocalSecondaryIndexExample  
    {  
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();  
        private static string tableName = "CustomerOrders";  
  
        static void Main(string[] args)  
        {  
            try  
            {  
                CreateTable();  
                LoadData();  
  
                Query(null);  
                Query("IsOpenIndex");  
                Query("OrderCreationDateIndex");  
  
                DeleteTable(tableName);  
  
                Console.WriteLine("To continue, press Enter");  
                Console.ReadLine();  
            }  
            catch (AmazonDynamoDBException e) { Console.WriteLine(e.Message); }  
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }  
            catch (Exception e) { Console.WriteLine(e.Message); }  
        }  
  
        private static void CreateTable()  
        {  
            var createTableRequest =  
                new CreateTableRequest()  
            {  
                TableName = tableName,  
                ProvisionedThroughput =  
                    new ProvisionedThroughput()  
                {  
                   
```

```

        ReadCapacityUnits = (long)1,
        WriteCapacityUnits = (long)1
    }
};

var attributeDefinitions = new List<AttributeDefinition>()
{
    // Attribute definitions for table primary key
    { new AttributeDefinition() {
        AttributeName = "CustomerId", AttributeType = "S"
    } },
    { new AttributeDefinition() {
        AttributeName = "OrderId", AttributeType = "N"
    } },
    // Attribute definitions for index primary key
    { new AttributeDefinition() {
        AttributeName = "OrderCreationDate", AttributeType = "N"
    } },
    { new AttributeDefinition() {
        AttributeName = "IsOpen", AttributeType = "N"
    } }
};

createTableRequest.AttributeDefinitions = attributeDefinitions;

// Key schema for table
var tableKeySchema = new List<KeySchemaElement>()
{
    { new KeySchemaElement() {
        AttributeName = "CustomerId", KeyType = "HASH"           //Partition key
    } },
    { new KeySchemaElement() {
        AttributeName = "OrderId", KeyType = "RANGE"           //Sort key
    } }
};

createTableRequest.KeySchema = tableKeySchema;

var localSecondaryIndexes = new List<LocalSecondaryIndex>();

// OrderCreationDateIndex
LocalSecondaryIndex orderCreationDateIndex = new LocalSecondaryIndex()
{
    IndexName = "OrderCreationDateIndex"
};

// Key schema for OrderCreationDateIndex
var indexKeySchema = new List<KeySchemaElement>()
{
    { new KeySchemaElement() {
        AttributeName = "CustomerId", KeyType = "HASH"           //Partition key
    } },
    { new KeySchemaElement() {
        AttributeName = "OrderCreationDate", KeyType = "RANGE"      //Sort key
    } }
};

orderCreationDateIndex.KeySchema = indexKeySchema;

// Projection (with list of projected attributes) for
// OrderCreationDateIndex
var projection = new Projection()
{
    ProjectionType = "INCLUDE"
};

```

```

        var nonKeyAttributes = new List<string>()
    {
        "ProductCategory",
        "ProductName"
    };
    projection.NonKeyAttributes = nonKeyAttributes;

    orderCreationDateIndex.Projection = projection;

    localSecondaryIndexes.Add(orderCreationDateIndex);

    // IsOpenIndex
    LocalSecondaryIndex isOpenIndex
        = new LocalSecondaryIndex()
    {
        IndexName = "IsOpenIndex"
    };

    // Key schema for IsOpenIndex
    indexKeySchema = new List<KeySchemaElement>()
    {
        { new KeySchemaElement() {
            AttributeName = "CustomerId", KeyType = "HASH"
        }}, //Partition key
        { new KeySchemaElement() {
            AttributeName = "IsOpen", KeyType = "RANGE"
        }} //Sort key
    };

    // Projection (all attributes) for IsOpenIndex
    projection = new Projection()
    {
        ProjectionType = "ALL"
    };

    isOpenIndex.KeySchema = indexKeySchema;
    isOpenIndex.Projection = projection;

    localSecondaryIndexes.Add(isOpenIndex);

    // Add index definitions to CreateTable request
    createTableRequest.LocalSecondaryIndexes = localSecondaryIndexes;

    Console.WriteLine("Creating table " + tableName + "...");
    Client.CreateTable(createTableRequest);
    WaitUntilTableReady(tableName);
}

public static void Query(string indexName)
{
    Console.WriteLine("\n*****\n");
    Console.WriteLine("Querying table " + tableName + "...");

    QueryRequest queryRequest = new QueryRequest()
    {
        TableName = tableName,
        ConsistentRead = true,
        ScanIndexForward = true,
        ReturnConsumedCapacity = "TOTAL"
    };

    String keyConditionExpression = "CustomerId = :v_customerId";
    Dictionary<string, AttributeValue> expressionAttributeValues = new
    Dictionary<string, AttributeValue> {

```

```

        {":v_customerId", new AttributeValue {
            S = "bob@example.com"
        })
    };

    if (indexName == "IsOpenIndex")
    {
        Console.WriteLine("\nUsing index: '" + indexName
                        + "':: Bob's orders that are open.");
        Console.WriteLine("Only a user-specified list of attributes are returned
\n");
        queryRequest.IndexName = indexName;

        keyConditionExpression += " and IsOpen = :v_isOpen";
        expressionAttributeValues.Add(":v_isOpen", new AttributeValue
        {
            N = "1"
        });

        // ProjectionExpression
        queryRequest.ProjectionExpression = "OrderCreationDate, ProductCategory,
ProductName, OrderStatus";
    }
    else if (indexName == "OrderCreationDateIndex")
    {
        Console.WriteLine("\nUsing index: '" + indexName
                        + "':: Bob's orders that were placed after 01/31/2013.");
        Console.WriteLine("Only the projected attributes are returned\n");
        queryRequest.IndexName = indexName;

        keyConditionExpression += " and OrderCreationDate > :v_Date";
        expressionAttributeValues.Add(":v_Date", new AttributeValue
        {
            N = "20130131"
        });

        // Select
        queryRequest.Select = "ALL_PROJECTED_ATTRIBUTES";
    }
    else
    {
        Console.WriteLine("\nNo index: All of Bob's orders, by OrderId:\n");
    }
    queryRequest.KeyConditionExpression = keyConditionExpression;
    queryRequest.ExpressionAttributeValues = expressionAttributeValues;

    var result = client.Query(queryRequest);
    var items = result.Items;
    foreach (var currentItem in items)
    {
        foreach (string attr in currentItem.Keys)
        {
            if (attr == "OrderId" || attr == "IsOpen"
                || attr == "OrderCreationDate")
            {
                Console.WriteLine(attr + "---> " + currentItem[attr].N);
            }
            else
            {
                Console.WriteLine(attr + "---> " + currentItem[attr].S);
            }
        }
        Console.WriteLine();
    }
}

```

```

        Console.WriteLine("\nConsumed capacity: " +
result.ConsumedCapacity.CapacityUnits + "\n");
    }

    private static void DeleteTable(string tableName)
{
    Console.WriteLine("Deleting table " + tableName + "...");
    client.DeleteTable(new DeleteTableRequest()
    {
        TableName = tableName
    });
    WaitForTableToDelete(tableName);
}

public static void LoadData()
{
    Console.WriteLine("Loading data into table " + tableName + "...");

    Dictionary<string, AttributeValue> item = new Dictionary<string,
AttributeValue>();

    item["CustomerId"] = new AttributeValue
    {
        S = "alice@example.com"
    };
    item["OrderId"] = new AttributeValue
    {
        N = "1"
    };
    item["IsOpen"] = new AttributeValue
    {
        N = "1"
    };
    item["OrderCreationDate"] = new AttributeValue
    {
        N = "20130101"
    };
    item["ProductCategory"] = new AttributeValue
    {
        S = "Book"
    };
    item["ProductName"] = new AttributeValue
    {
        S = "The Great Outdoors"
    };
    item["OrderStatus"] = new AttributeValue
    {
        S = "PACKING ITEMS"
    };
    /* no ShipmentTrackingId attribute */
    PutItemRequest putItemRequest = new PutItemRequest
    {
        TableName = tableName,
        Item = item,
        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);

    item = new Dictionary<string, AttributeValue>();
    item["CustomerId"] = new AttributeValue
    {
        S = "alice@example.com"
    };
    item["OrderId"] = new AttributeValue
    {
        N = "2"
    }
}

```

```

};

item["IsOpen"] = new AttributeValue
{
    N = "1"
};
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130221"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Bike"
};
item["ProductName"] = new AttributeValue
{
    S = "Super Mountain"
};
item["OrderStatus"] = new AttributeValue
{
    S = "ORDER RECEIVED"
};
/* no ShipmentTrackingId attribute */
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "alice@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "3"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130304"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "A Quiet Interlude"
};
item["OrderStatus"] = new AttributeValue
{
    S = "IN TRANSIT"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "176493"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};

```

```
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "1"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130111"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Movie"
};
item["ProductName"] = new AttributeValue
{
    S = "Calm Before The Storm"
};
item["OrderStatus"] = new AttributeValue
{
    S = "SHIPPING DELAY"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "859323"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "2"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130124"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "E-Z Listening"
};
item["OrderStatus"] = new AttributeValue
{
    S = "DELIVERED"
};
```

```
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "756943"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "3"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130221"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Music"
};
item["ProductName"] = new AttributeValue
{
    S = "Symphony 9"
};
item["OrderStatus"] = new AttributeValue
{
    S = "DELIVERED"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "645193"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "4"
};
item["IsOpen"] = new AttributeValue
{
    N = "1"
};
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130222"
};
```

```

};

item["ProductCategory"] = new AttributeValue
{
    S = "Hardware"
};
item["ProductName"] = new AttributeValue
{
    S = "Extra Heavy Hammer"
};
item["OrderStatus"] = new AttributeValue
{
    S = "PACKING ITEMS"
};
/* no ShipmentTrackingId attribute */
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "5"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130309"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Book"
};
item["ProductName"] = new AttributeValue
{
    S = "How To Cook"
};
item["OrderStatus"] = new AttributeValue
{
    S = "IN TRANSIT"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "440185"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "5"
};
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130309"
};
item["OrderStatus"] = new AttributeValue
{
    S = "IN TRANSIT"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "440185"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

```

```

{
    N = "6"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130318"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Luggage"
};
item["ProductName"] = new AttributeValue
{
    S = "Really Big Suitcase"
};
item["OrderStatus"] = new AttributeValue
{
    S = "DELIVERED"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "893927"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
    ReturnItemCollectionMetrics = "SIZE"
};
client.PutItem(putItemRequest);

item = new Dictionary<string, AttributeValue>();
item["CustomerId"] = new AttributeValue
{
    S = "bob@example.com"
};
item["OrderId"] = new AttributeValue
{
    N = "7"
};
/* no IsOpen attribute */
item["OrderCreationDate"] = new AttributeValue
{
    N = "20130324"
};
item["ProductCategory"] = new AttributeValue
{
    S = "Golf"
};
item["ProductName"] = new AttributeValue
{
    S = "PGA Pro II"
};
item["OrderStatus"] = new AttributeValue
{
    S = "OUT FOR DELIVERY"
};
item["ShipmentTrackingId"] = new AttributeValue
{
    S = "383283"
};
putItemRequest = new PutItemRequest
{
    TableName = tableName,
    Item = item,
}

```

```

        ReturnItemCollectionMetrics = "SIZE"
    };
    client.PutItem(putItemRequest);
}

private static void WaitUntilTableReady(string tableName)
{
    string status = null;
    // Let us wait until table is created. Call DescribeTable.
    do
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
        catch (ResourceNotFoundException)
        {
            // DescribeTable is eventually consistent. So you might
            // get resource not found. So we handle the potential exception.
        }
    } while (status != "ACTIVE");
}

private static void WaitForTableToDelete(string tableName)
{
    bool tablePresent = true;

    while (tablePresent)
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });

            Console.WriteLine("Table name: {0}, status: {1}",
                res.Table.TableName,
                res.Table.TableStatus);
        }
        catch (ResourceNotFoundException)
        {
            tablePresent = false;
        }
    }
}
}

```

Working with Local Secondary Indexes: AWS CLI

You can use the AWS CLI to create an Amazon DynamoDB table with one or more local secondary indexes, describe the indexes on the table, and perform queries using the indexes.

Topics

- [Create a Table with a Local Secondary Index \(p. 572\)](#)
- [Describe a Table with a Local Secondary Index \(p. 572\)](#)
- [Query a Local Secondary Index \(p. 572\)](#)

Create a Table with a Local Secondary Index

Local secondary indexes must be created at the same time you create a table. To do this, use the `create-table` parameter and provide your specifications for one or more local secondary indexes. The following example creates a table (`Music`) to hold information about songs in a music collection. The partition key is `Artist` and the sort key is `SongTitle`. A secondary index, `AlbumTitleIndex` on the `AlbumTitle` attribute facilitates queries by album title.

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

You must wait until DynamoDB creates the table and sets the table status to `ACTIVE`. After that, you can begin putting data items into the table. You can use [describe-table](#) to determine the status of the table creation.

Describe a Table with a Local Secondary Index

To get information about local secondary indexes on a table, use the `describe-table` parameter. For each index, you can access its name, key schema, and projected attributes.

```
aws dynamodb describe-table --table-name Music
```

Query a Local Secondary Index

You can use the `query` operation on a local secondary index in much the same way that you `query` a table. You must specify the index name, the query criteria for the index sort key, and the attributes that you want to return. In this example, the index is `AlbumTitleIndex` and the index sort key is `AlbumTitle`.

The only attributes returned are those that have been projected into the index. You could modify this query to select non-key attributes too, but this would require table fetch activity that is relatively expensive. For more information about table fetches, see [Attribute Projections \(p. 539\)](#).

```
aws dynamodb query --table-name Music \
--index-name AlbumTitleIndex \
--key-condition-expression "Artist = :v_artist and AlbumTitle = :v_title" \
--expression-attribute-values '{\":v_artist\":{\"S\":\"Acme Band\"},\":v_title\":{\"S\":\"Songs About Life\"} }'
```

Capturing Table Activity with DynamoDB Streams

Many applications can benefit from the ability to capture changes to items stored in a DynamoDB table, at the point in time when such changes occur. The following are some example use cases:

- An application in one AWS Region modifies the data in a DynamoDB table. A second application in another Region reads these data modifications and writes the data to another table, creating a replica that stays in sync with the original table.
- A popular mobile app modifies data in a DynamoDB table, at the rate of thousands of updates per second. Another application captures and stores data about these updates, providing near-real-time usage metrics for the mobile app.
- A global multi-player game has a multi-master topology, storing data in multiple AWS Regions. Each master stays in sync by consuming and replaying the changes that occur in the remote Regions.
- An application automatically sends notifications to the mobile devices of all friends in a group as soon as one friend uploads a new picture.
- A new customer adds data to a DynamoDB table. This event invokes another application that sends a welcome email to the new customer.

DynamoDB Streams enables solutions such as these, and many others. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near-real time.

Encryption at rest encrypts the data in DynamoDB streams. For more information, see [DynamoDB Encryption at Rest \(p. 815\)](#).

A *DynamoDB stream* is an ordered flow of information about changes to items in a DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A *stream record* contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams helps ensure the following:

- Each stream record appears exactly once in the stream.
- For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

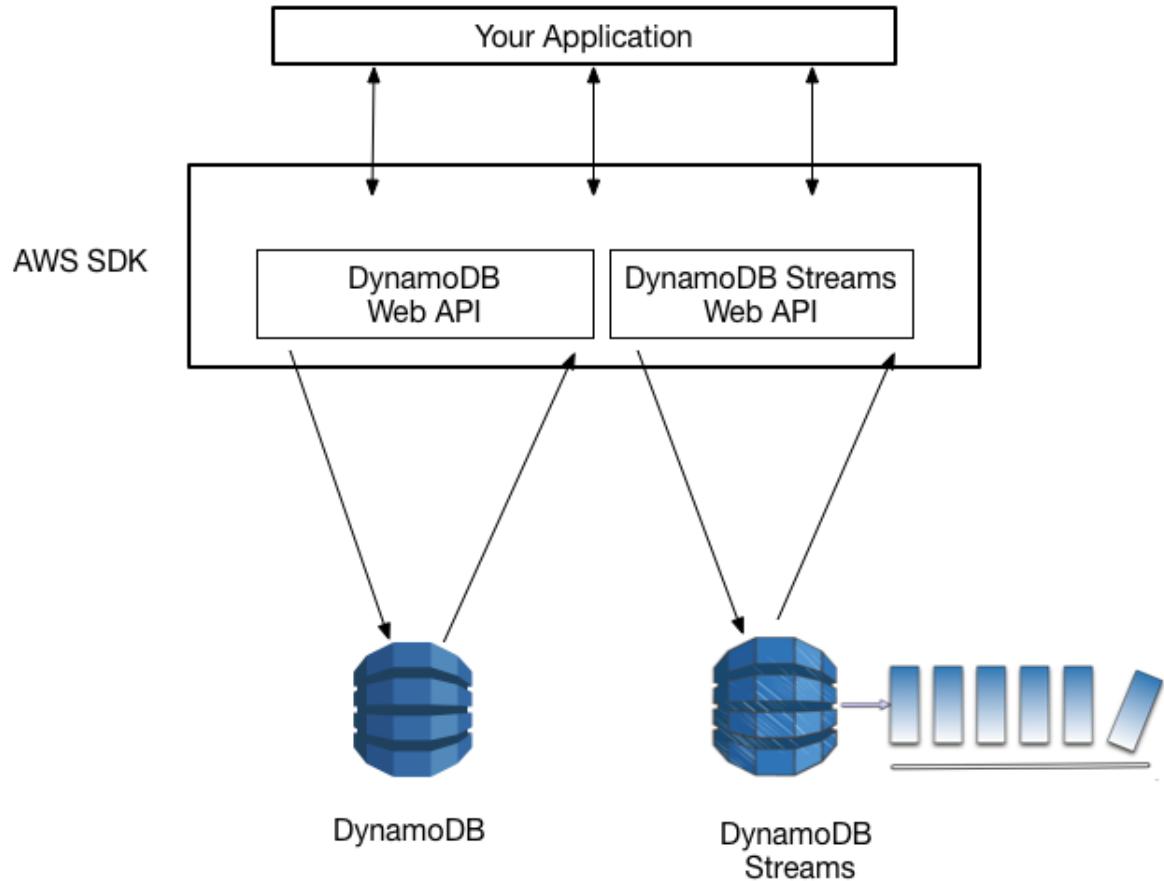
DynamoDB Streams writes stream records in near-real time so that you can build applications that consume these streams and take action based on the contents.

Topics

- [Endpoints for DynamoDB Streams \(p. 574\)](#)
- [Enabling a Stream \(p. 575\)](#)
- [Reading and Processing a Stream \(p. 576\)](#)
- [DynamoDB Streams and Time to Live \(p. 578\)](#)
- [Using the DynamoDB Streams Kinesis Adapter to Process Stream Records \(p. 578\)](#)
- [DynamoDB Streams Low-Level API: Java Example \(p. 590\)](#)
- [Cross-Region Replication \(p. 594\)](#)

- [DynamoDB Streams and AWS Lambda Triggers \(p. 594\)](#)

Endpoints for DynamoDB Streams



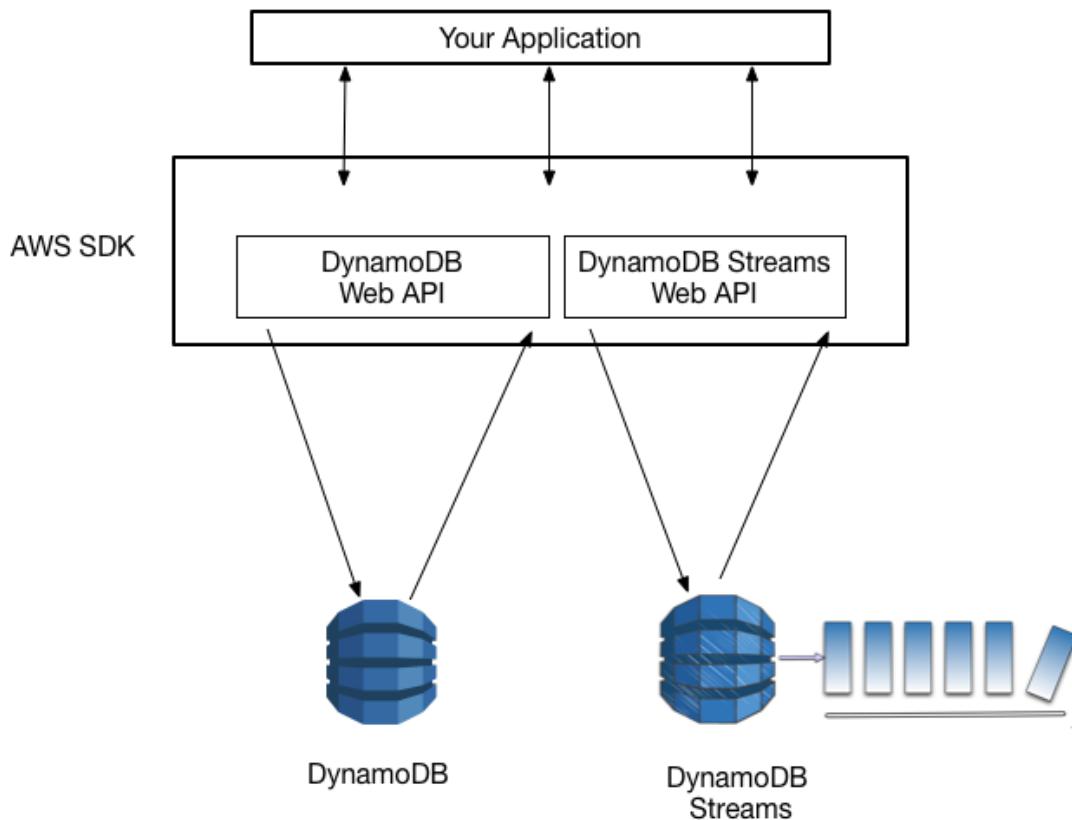
AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application must access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application must access a DynamoDB Streams endpoint in the same Region.

The naming convention for DynamoDB Streams endpoints is `streams.dynamodb.<region>.amazonaws.com`. For example, if you use the endpoint `dynamodb.us-west-2.amazonaws.com` to access DynamoDB, you would use the endpoint `streams.dynamodb.us-west-2.amazonaws.com` to access DynamoDB Streams.

Note

For a complete list of DynamoDB and DynamoDB Streams Regions and endpoints, see [Regions and Endpoints](#) in the *AWS General Reference*.

The AWS SDKs provide separate clients for DynamoDB and DynamoDB Streams. Depending on your requirements, your application can access a DynamoDB endpoint, a DynamoDB Streams endpoint, or both at the same time. To connect to both endpoints, your application must instantiate two clients—one for DynamoDB and one for DynamoDB Streams.



Enabling a Stream

You can enable a stream on a new table when you create it. You can also enable or disable a stream on an existing table, or change the settings of a stream. DynamoDB Streams operates asynchronously, so there is no performance impact on a table if you enable a stream.

The easiest way to manage DynamoDB Streams is by using the AWS Management Console.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. On the DynamoDB console dashboard, choose **Tables**.
3. On the **Overview** tab, choose **Manage Stream**.
4. In the **Manage Stream** window, choose the information that will be written to the stream whenever the data in the table is modified:
 - **Keys only** — Only the key attributes of the modified item.
 - **New image** — The entire item, as it appears after it was modified.
 - **Old image** — The entire item, as it appeared before it was modified.
 - **New and old images** — Both the new and the old images of the item.

When the settings are as you want them, choose **Enable**.

5. (Optional) To disable an existing stream, choose **Manage Stream** and then choose **Disable**.

You can also use the `CreateTable` or `UpdateTable` API operations to enable or modify a stream. The `StreamSpecification` parameter determines how the stream is configured:

- `StreamEnabled` — Specifies whether a stream is enabled (`true`) or disabled (`false`) for the table.
- `StreamViewType` — Specifies the information that will be written to the stream whenever data in the table is modified:
 - `KEYS_ONLY` — Only the key attributes of the modified item.
 - `NEW_IMAGE` — The entire item, as it appears after it was modified.
 - `OLD_IMAGE` — The entire item, as it appeared before it was modified.
 - `NEW_AND_OLD_IMAGES` — Both the new and the old images of the item.

You can enable or disable a stream at any time. However, you receive a `ResourceInUseException` if you try to enable a stream on a table that already has a stream. You receive a `ValidationException` if you try to disable a stream on a table that doesn't have a stream.

When you set `StreamEnabled` to `true`, DynamoDB creates a new stream with a unique stream descriptor assigned to it. If you disable and then re-enable a stream on the table, a new stream is created with a different stream descriptor.

Every stream is uniquely identified by an Amazon Resource Name (ARN). The following is an example ARN for a stream on a DynamoDB table named `TestTable`.

```
arn:aws:dynamodb:us-west-2:111122223333:table/TestTable/stream/2015-05-11T21:21:33.291
```

To determine the latest stream descriptor for a table, issue a DynamoDB `DescribeTable` request and look for the `LatestStreamArn` element in the response.

Reading and Processing a Stream

To read and process a stream, your application must connect to a DynamoDB Streams endpoint and issue API requests.

A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

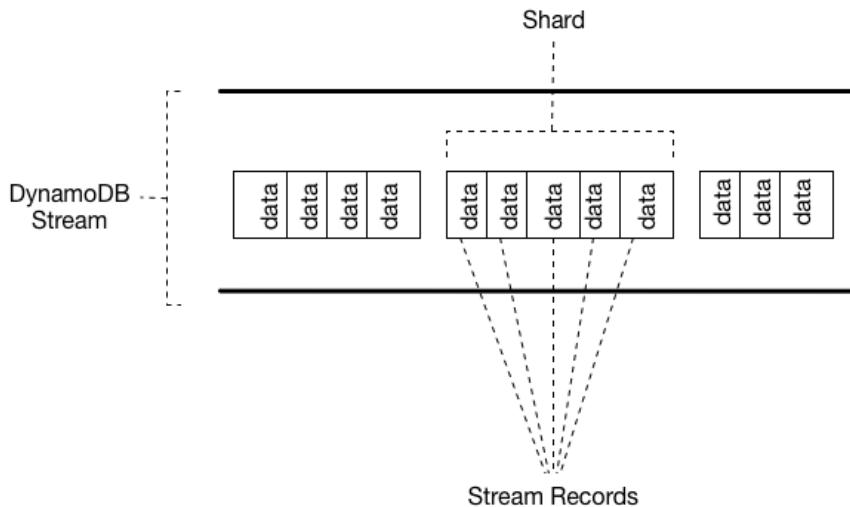
Stream records are organized into groups, or *shards*. Each shard acts as a container for multiple stream records, and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours.

Shards are ephemeral: They are created and deleted automatically, as needed. Any shard can also split into multiple new shards; this also occurs automatically. (It's also possible for a parent shard to have just one child shard.) A shard might split in response to high levels of write activity on its parent table, so that applications can process records from multiple shards in parallel.

If you disable a stream, any shards that are open will be closed.

Because shards have a lineage (parent and children), an application must always process a parent shard before it processes a child shard. This helps ensure that the stream records are also processed in the correct order. (If you use the DynamoDB Streams Kinesis Adapter, this is handled for you. Your application processes the shards and stream records in the correct order. It automatically handles new or expired shards, in addition to shards that split while the application is running. For more information, see [Using the DynamoDB Streams Kinesis Adapter to Process Stream Records \(p. 578\)](#).)

The following diagram shows the relationship between a stream, shards in the stream, and stream records in the shards.



Note

If you perform a `PutItem` or `UpdateItem` operation that does not change any data in an item, DynamoDB Streams does *not* write a stream record for that operation.

To access a stream and process the stream records within, you must do the following:

- Determine the unique ARN of the stream that you want to access.
- Determine which shards in the stream contain the stream records that you are interested in.
- Access the shards and retrieve the stream records that you want.

Note

No more than two processes at most should be reading from the same streams shard at the same time. Having more than two readers per shard can result in throttling.

The DynamoDB Streams API provides the following actions for use by application programs:

- `ListStreams` — Returns a list of stream descriptors for the current account and endpoint. You can optionally request just the stream descriptors for a particular table name.
- `DescribeStream` — Returns detailed information about a given stream. The output includes a list of shards associated with the stream, including the shard IDs.
- `GetShardIterator` — Returns a *shard iterator*, which describes a location within a shard. You can request that the iterator provide access to the oldest point, the newest point, or a particular point in the stream.
- `GetRecords` — Returns the stream records from within a given shard. You must provide the shard iterator returned from a `GetShardIterator` request.

For complete descriptions of these API operations, including example requests and responses, see the [Amazon DynamoDB Streams API Reference](#).

Data Retention Limit for DynamoDB Streams

All data in DynamoDB Streams is subject to a 24-hour lifetime. You can retrieve and analyze the last 24 hours of activity for any given table. However, data that is older than 24 hours is susceptible to trimming (removal) at any moment.

If you disable a stream on a table, the data in the stream continues to be readable for 24 hours. After this time, the data expires and the stream records are automatically deleted. There is no mechanism for

manually deleting an existing stream. You must wait until the retention limit expires (24 hours), and all the stream records will be deleted.

DynamoDB Streams and Time to Live

You can back up, or otherwise process, items that are deleted by Time to Live (TTL) by enabling Amazon DynamoDB Streams on the table and processing the streams records of the expired items.

The streams record contains a user identity field `Records[<index>].userIdentity`.

Items that are deleted by the Time to Live process after expiration have the following fields:

- `Records[<index>].userIdentity.type`
 "Service"
- `Records[<index>].userIdentity.principalId`
 "dynamodb.amazonaws.com"

The following JSON shows the relevant portion of a single streams record.

```
"Records": [
    {
        ...
        "userIdentity":{
            "type":"Service",
            "principalId":"dynamodb.amazonaws.com"
        }
        ...
    }
]
```

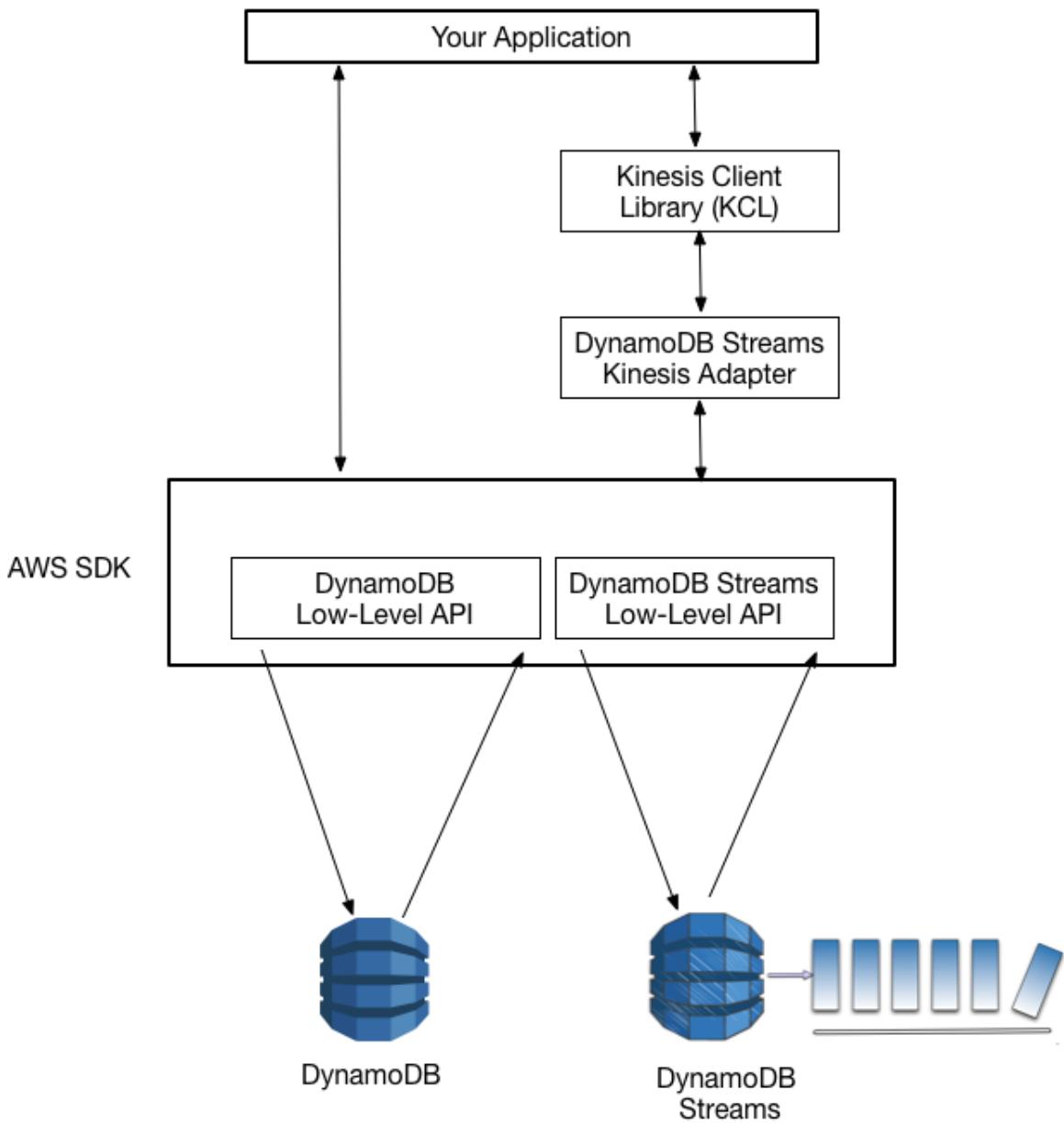
Using the DynamoDB Streams Kinesis Adapter to Process Stream Records

Using the Amazon Kinesis Adapter is the recommended way to consume streams from Amazon DynamoDB. The DynamoDB Streams API is intentionally similar to that of Kinesis Data Streams, a service for real-time processing of streaming data at massive scale. In both services, data streams are composed of shards, which are containers for stream records. Both services' APIs contain `ListStreams`, `DescribeStream`, `GetShards`, and `GetShardIterator` operations. (Although these DynamoDB Streams actions are similar to their counterparts in Kinesis Data Streams, they are not 100 percent identical.)

You can write applications for Kinesis Data Streams using the Kinesis Client Library (KCL). The KCL simplifies coding by providing useful abstractions above the low-level Kinesis Data Streams API. For more information about the KCL, see the [Developing Consumers Using the Kinesis Client Library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

As a DynamoDB Streams user, you can use the design patterns found within the KCL to process DynamoDB Streams shards and stream records. To do this, you use the DynamoDB Streams Kinesis Adapter. The Kinesis Adapter implements the Kinesis Data Streams interface so that the KCL can be used for consuming and processing records from DynamoDB Streams.

The following diagram shows how these libraries interact with one another.



With the DynamoDB Streams Kinesis Adapter in place, you can begin developing against the KCL interface, with the API calls seamlessly directed at the DynamoDB Streams endpoint.

When your application starts, it calls the KCL to instantiate a worker. You must provide the worker with configuration information for the application, such as the stream descriptor and AWS credentials, and the name of a record processor class that you provide. As it runs the code in the record processor, the worker performs the following tasks:

- Connects to the stream.
- Enumerates the shards within the stream.
- Coordinates shard associations with other workers (if any).
- Instantiates a record processor for every shard it manages.

- Pulls records from the stream.
- Pushes the records to the corresponding record processor.
- Checkpoints processed records.
- Balances shard-worker associations when the worker instance count changes.
- Balances shard-worker associations when shards are split.

Note

For a description of the KCL concepts listed here, see [Developing Consumers Using the Kinesis Client Library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

Walkthrough: DynamoDB Streams Kinesis Adapter

This section is a walkthrough of a Java application that uses the Amazon Kinesis Client Library and the Amazon DynamoDB Streams Kinesis Adapter. The application shows an example of data replication, in which write activity from one table is applied to a second table, with both tables' contents staying in sync. For the source code, see [Complete Program: DynamoDB Streams Kinesis Adapter \(p. 583\)](#).

The program does the following:

1. Creates two DynamoDB tables named `KCL-Demo-src` and `KCL-Demo-dst`. Each of these tables has a stream enabled on it.
2. Generates update activity in the source table by adding, updating, and deleting items. This causes data to be written to the table's stream.
3. Reads the records from the stream, reconstructs them as DynamoDB requests, and applies the requests to the destination table.
4. Scans the source and destination tables to ensure that their contents are identical.
5. Cleans up by deleting the tables.

These steps are described in the following sections, and the complete application is shown at the end of the walkthrough.

Topics

- [Step 1: Create DynamoDB Tables \(p. 580\)](#)
- [Step 2: Generate Update Activity in Source Table \(p. 581\)](#)
- [Step 3: Process the Stream \(p. 581\)](#)
- [Step 4: Ensure That Both Tables Have Identical Contents \(p. 582\)](#)
- [Step 5: Clean Up \(p. 582\)](#)
- [Complete Program: DynamoDB Streams Kinesis Adapter \(p. 583\)](#)

Step 1: Create DynamoDB Tables

The first step is to create two DynamoDB tables—a source table and a destination table. The `StreamViewType` on the source table's stream is `NEW_IMAGE`. This means that whenever an item is modified in this table, the item's "after" image is written to the stream. In this way, the stream keeps track of all write activity on the table.

The following example shows the code that is used for creating both tables.

```
java.util.List<AttributeDefinition> attributeDefinitions = new
    ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
    AttributeDefinition().withAttributeName("Id").withAttributeType("N"));

java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition
// key

ProvisionedThroughput provisionedThroughput = new
    ProvisionedThroughput().withReadCapacityUnits(2L)
        .withWriteCapacityUnits(2L);

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
CreateTableRequest createTableRequest = new CreateTableRequest().withTableName(tableName)
    .withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

    .withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification);
```

Step 2: Generate Update Activity in Source Table

The next step is to generate some write activity on the source table. While this activity is taking place, the source table's stream is also updated in near-real time.

The application defines a helper class with methods that call the `PutItem`, `UpdateItem`, and `DeleteItem` API operations for writing the data. The following code example shows how these methods are used.

```
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
```

Step 3: Process the Stream

Now the program begins processing the stream. The DynamoDB Streams Kinesis Adapter acts as a transparent layer between the KCL and the DynamoDB Streams endpoint, so that the code can fully use KCL rather than having to make low-level DynamoDB Streams calls. The program performs the following tasks:

- It defines a record processor class, `StreamsRecordProcessor`, with methods that comply with the KCL interface definition: `initialize`, `processRecords`, and `shutdown`. The `processRecords` method contains the logic required for reading from the source table's stream and writing to the destination table.
- It defines a class factory for the record processor class (`StreamsRecordProcessorFactory`). This is required for Java programs that use the KCL.
- It instantiates a new KCL Worker, which is associated with the class factory.
- It shuts down the worker when record processing is complete.

To learn more about the KCL interface definition, see [Developing Consumers Using the Kinesis Client Library](#) in the *Amazon Kinesis Data Streams Developer Guide*.

The following code example shows the main loop in `StreamsRecordProcessor`. The `case` statement determines what action to perform, based on the `OperationType` that appears in the stream record.

```
for (Record record : records) {
    String data = new String(record.getData().array(), Charset.forName("UTF-8"));
    System.out.println(data);
    if (record instanceof RecordAdapter) {
        com.amazonaws.services.dynamodbv2.model.Record streamRecord =
        ((RecordAdapter) record)
            .getInternalObject();

        switch (streamRecord.getEventName()) {
            case "INSERT":
            case "MODIFY":
                StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,
                    streamRecord.getDynamodb().getNewImage());
                break;
            case "REMOVE":
                StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,
                    streamRecord.getDynamodb().getKeys().get("Id").getN());
        }
    }
    checkpointCounter += 1;
    if (checkpointCounter % 10 == 0) {
        try {
            checkpointer.checkpoint();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Ensure That Both Tables Have Identical Contents

At this point, the source and destination tables' contents are in sync. The application issues `Scan` requests against both tables to verify that their contents are, in fact, identical.

The `DemoHelper` class contains a `ScanTable` method that calls the low-level `Scan` API. The following example shows how this is used.

```
if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
    .equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient, destTable).getItems())) {
    System.out.println("Scan result is equal.");
}
else {
    System.out.println("Tables are different!");
}
```

Step 5: Clean Up

The demo is complete, so the application deletes the source and destination tables. See the following code example. Even after the tables are deleted, their streams remain available for up to 24 hours, after which they are automatically deleted.

```
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
```

Complete Program: DynamoDB Streams Kinesis Adapter

The following is the complete Java program that performs the tasks described in [Walkthrough: DynamoDB Streams Kinesis Adapter \(p. 580\)](#). When you run it, you should see output similar to the following.

```
Creating table KCL-Demo-src
Creating table KCL-Demo-dest
Table is active.
Creating worker for stream: arn:aws:dynamodb:us-west-2:111122223333:table/KCL-Demo-src/
stream/2015-05-19T22:48:56.601
Starting worker...
Scan result is equal.
Done.
```

Important

To run this program, ensure that the client application has access to DynamoDB and Amazon CloudWatch using policies. For more information, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 828\)](#).

The source code consists of four .java files:

- StreamsAdapterDemo.java
- StreamsRecordProcessor.java
- StreamsRecordProcessorFactory.java
- StreamsAdapterDemoHelper.java

StreamsAdapterDemo.java

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples;

import com.amazonaws.auth.AWS CredentialsProvider;
import com.amazonaws.auth.DefaultAWS CredentialsProviderChain;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.cloudwatch.AmazonCloudWatch;
import com.amazonaws.services.cloudwatch.AmazonCloudWatchClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreams;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreamsClientBuilder;
import com.amazonaws.services.dynamodbv2.model.DeleteTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.streamsadapter.AmazonDynamoDBStreamsAdapterClient;
import com.amazonaws.services.dynamodbv2.streamsadapter.StreamsWorkerFactory;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
```

```
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.InitialPositionInStream;
import
com.amazonaws.services.kinesis.clientlibrary.lib.worker.KinesisClientLibConfiguration;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;

public class StreamsAdapterDemo {
    private static Worker worker;
    private static KinesisClientLibConfiguration workerConfig;
    private static IRecordProcessorFactory recordProcessorFactory;

    private static AmazonDynamoDB dynamoDBClient;
    private static AmazonCloudWatch cloudWatchClient;
    private static AmazonDynamoDBStreams dynamoDBStreamsClient;
    private static AmazonDynamoDBStreamsAdapterClient adapterClient;

    private static String tablePrefix = "KCL-Demo";
    private static String streamArn;

    private static Regions awsRegion = Regions.US_EAST_2;

    private static AWSCredentialsProvider awsCredentialsProvider =
DefaultAWSCredentialsProviderChain.getInstance();

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        System.out.println("Starting demo...");

        dynamoDBClient = AmazonDynamoDBClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        cloudWatchClient = AmazonCloudWatchClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        dynamoDBStreamsClient = AmazonDynamoDBStreamsClientBuilder.standard()
            .withRegion(awsRegion)
            .build();
        adapterClient = new AmazonDynamoDBStreamsAdapterClient(dynamoDBStreamsClient);
        String srcTable = tablePrefix + "-src";
        String destTable = tablePrefix + "-dest";
        recordProcessorFactory = new StreamsRecordProcessorFactory(dynamoDBClient,
destTable);

        setUpTables();

        workerConfig = new KinesisClientLibConfiguration("streams-adapter-demo",
streamArn,
awsCredentialsProvider,
"streams-demo-worker")
            .withMaxRecords(1000)
            .withIdleTimeBetweenReadsInMillis(500)
            .withInitialPositionInStream(InitialPositionInStream.TRIM_HORIZON);

        System.out.println("Creating worker for stream: " + streamArn);
        worker = StreamsWorkerFactory.createDynamoDbStreamsWorker(recordProcessorFactory,
workerConfig, adapterClient, dynamoDBClient, cloudWatchClient);
        System.out.println("Starting worker...");
        Thread t = new Thread(worker);
        t.start();

        Thread.sleep(25000);
        worker.shutdown();
        t.join();

        if (StreamsAdapterDemoHelper.scanTable(dynamoDBClient, srcTable).getItems()
```

```
.equals(StreamsAdapterDemoHelper.scanTable(dynamoDBClient, destTable).getItems())) {
    System.out.println("Scan result is equal.");
}
else {
    System.out.println("Tables are different!");
}

System.out.println("Done.");
cleanupAndExit(0);
}

private static void setUpTables() {
    String srcTable = tablePrefix + "-src";
    String destTable = tablePrefix + "-dest";
    streamArn = StreamsAdapterDemoHelper.createTable(dynamoDBClient, srcTable);
    StreamsAdapterDemoHelper.createTable(dynamoDBClient, destTable);

    awaitTableCreation(srcTable);

    performOps(srcTable);
}

private static void awaitTableCreation(String tableName) {
    Integer retries = 0;
    Boolean created = false;
    while (!created && retries < 100) {
        DescribeTableResult result =
StreamsAdapterDemoHelper.describeTable(dynamoDBClient, tableName);
        created = result.getTable().getTableStatus().equals("ACTIVE");
        if (created) {
            System.out.println("Table is active.");
            return;
        }
        else {
            retries++;
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                // do nothing
            }
        }
    }
    System.out.println("Timeout after table creation. Exiting...");
    cleanupAndExit(1);
}

private static void performOps(String tableName) {
    StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "101", "test1");
    StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "101", "test2");
    StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "101");
    StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName, "102", "demo3");
    StreamsAdapterDemoHelper.updateItem(dynamoDBClient, tableName, "102", "demo4");
    StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName, "102");
}

private static void cleanupAndExit(Integer returnValue) {
    String srcTable = tablePrefix + "-src";
    String destTable = tablePrefix + "-dest";
    dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(srcTable));
    dynamoDBClient.deleteTable(new DeleteTableRequest().withTableName(destTable));
    System.exit(returnValue);
}
```

StreamsRecordProcessor.java

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
  
package com.amazonaws.codesamples;  
  
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;  
import com.amazonaws.services.dynamodbv2.streamsadapter.model.RecordAdapter;  
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;  
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;  
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;  
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;  
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;  
import com.amazonaws.services.kinesis.model.Record;  
  
import java.nio.charset.Charset;  
  
public class StreamsRecordProcessor implements IRecordProcessor {  
    private Integer checkpointCounter;  
  
    private final AmazonDynamoDB dynamoDBClient;  
    private final String tableName;  
  
    public StreamsRecordProcessor(AmazonDynamoDB dynamoDBClient2, String tableName) {  
        this.dynamoDBClient = dynamoDBClient2;  
        this.tableName = tableName;  
    }  
  
    @Override  
    public void initialize(InitializationInput initializationInput) {  
        checkpointCounter = 0;  
    }  
  
    @Override  
    public void processRecords(ProcessRecordsInput processRecordsInput) {  
        for (Record record : processRecordsInput.getRecords()) {  
            String data = new String(record.getData().array(), Charset.forName("UTF-8"));  
            System.out.println(data);  
            if (record instanceof RecordAdapter) {  
                com.amazonaws.services.dynamodbv2.model.Record streamRecord =  
                ((RecordAdapter) record)  
                    .getInternalObject();  
  
                switch (streamRecord.getEventName()) {  
                    case "INSERT":  
                    case "MODIFY":  
                        StreamsAdapterDemoHelper.putItem(dynamoDBClient, tableName,  
                            streamRecord.getDynamodb().getNewImage());  
                }  
            }  
        }  
    }  
}
```

```
        break;
    case "REMOVE":
        StreamsAdapterDemoHelper.deleteItem(dynamoDBClient, tableName,
streamRecord.getDynamodb().getKeys().get("Id").getN());
    }
}
checkpointCounter += 1;
if (checkpointCounter % 10 == 0) {
    try {
        processRecordsInput.getCheckpointer().checkpoint();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

@Override
public void shutdown(ShutdownInput shutdownInput) {
    if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
        try {
            shutdownInput.getCheckpointer().checkpoint();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

StreamsRecordProcessorFactory.java

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class StreamsRecordProcessorFactory implements IRecordProcessorFactory {
    private final String tableName;
    private final AmazonDynamoDB dynamoDBClient;

    public StreamsRecordProcessorFactory(AmazonDynamoDB dynamoDBClient, String tableName) {
        this.tableName = tableName;
    }
}
```

```
        this.dynamoDBClient = dynamoDBClient;
    }

    @Override
    public IRecordProcessor createProcessor() {
        return new StreamsRecordProcessor(dynamoDBClient, tableName);
    }
}
```

StreamsAdapterDemoHelper.java

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.CreateTableResult;
import com.amazonaws.services.dynamodbv2.model.DeleteItemRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.PutItemRequest;
import com.amazonaws.services.dynamodbv2.model.ResourceInUseException;
import com.amazonaws.services.dynamodbv2.model.ScanRequest;
import com.amazonaws.services.dynamodbv2.model.ScanResult;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.model.UpdateItemRequest;

public class StreamsAdapterDemoHelper {

    /**
     * @return StreamArn
     */
    public static String createTable(AmazonDynamoDB client, String tableName) {
        java.util.List<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
        attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Id").withAttributeType("N"));
    }
}
```

```
java.util.List<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
keySchema.add(new
KeySchemaElement().withAttributeName("Id").withKeyType(KeyType.HASH)); // Partition

// key

ProvisionedThroughput provisionedThroughput = new
ProvisionedThroughput().withReadCapacityUnits(2L)
    .withWriteCapacityUnits(2L);

StreamSpecification streamSpecification = new StreamSpecification();
streamSpecification.setStreamEnabled(true);
streamSpecification.setStreamViewType(StreamViewType.NEW_IMAGE);
CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName)
    .withAttributeDefinitions(attributeDefinitions).withKeySchema(keySchema)

.withProvisionedThroughput(provisionedThroughput).withStreamSpecification(streamSpecification);

try {
    System.out.println("Creating table " + tableName);
    CreateTableResult result = client.createTable(createTableRequest);
    return result.getTableDescription().getLatestStreamArn();
}
catch (ResourceInUseException e) {
    System.out.println("Table already exists.");
    return describeTable(client, tableName).getTable().getLatestStreamArn();
}
}

public static DescribeTableResult describeTable(AmazonDynamoDB client, String
tableName) {
    return client.describeTable(new DescribeTableRequest().withTableName(tableName));
}

public static ScanResult scanTable(AmazonDynamoDB dynamoDBClient, String tableName) {
    return dynamoDBClient.scan(new ScanRequest().withTableName(tableName));
}

public static void putItem(AmazonDynamoDB dynamoDBClient, String tableName, String id,
String val) {
    java.util.Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
    item.put("Id", new AttributeValue().withN(id));
    item.put("attribute-1", new AttributeValue().withS(val));

    PutItemRequest putItemRequest = new
PutItemRequest().withTableName(tableName).withItem(item);
    dynamoDBClient.putItem(putItemRequest);
}

public static void putItem(AmazonDynamoDB dynamoDBClient, String tableName,
    java.util.Map<String, AttributeValue> items) {
    PutItemRequest putItemRequest = new
PutItemRequest().withTableName(tableName).withItem(items);
    dynamoDBClient.putItem(putItemRequest);
}

public static void updateItem(AmazonDynamoDB dynamoDBClient, String tableName, String
id, String val) {
    java.util.Map<String, AttributeValue> key = new HashMap<String, AttributeValue>();
    key.put("Id", new AttributeValue().withN(id));

    Map<String, AttributeValueUpdate> attributeUpdates = new HashMap<String,
AttributeValueUpdate>();
    AttributeValueUpdate update = new
AttributeValueUpdate().withAction(AttributeAction.PUT)
```

```

        .withValue(new AttributeValue().withS(val));
    attributeUpdates.put("attribute-2", update);

    UpdateItemRequest updateItemRequest = new
UpdateItemRequest().withTableName(tableName).withKey(key)
        .withAttributeUpdates(attributeUpdates);
    dynamoDBClient.updateItem(updateItemRequest);
}

public static void deleteItem(AmazonDynamoDB dynamoDBClient, String tableName, String
id) {
    java.util.Map<String, AttributeValue> key = new HashMap<String, AttributeValue>();
    key.put("Id", new AttributeValue().withN(id));

    DeleteItemRequest deleteItemRequest = new
DeleteItemRequest().withTableName(tableName).withKey(key);
    dynamoDBClient.deleteItem(deleteItemRequest);
}

}

```

DynamoDB Streams Low-Level API: Java Example

Note

The code on this page is not exhaustive and does not handle all scenarios for consuming Amazon DynamoDB Streams. The recommended way to consume stream records from DynamoDB is through the Amazon Kinesis Adapter using the Kinesis Client Library (KCL), as described in [Using the DynamoDB Streams Kinesis Adapter to Process Stream Records \(p. 578\)](#).

This section contains a Java program that shows DynamoDB Streams in action. The program does the following:

1. Creates a DynamoDB table with a stream enabled.
2. Describes the stream settings for this table.
3. Modifies data in the table.
4. Describes the shards in the stream.
5. Reads the stream records from the shards.
6. Cleans up.

When you run the program, you will see output similar to the following.

```

Issuing CreateTable request for TestTableForStreams
Waiting for TestTableForStreams to be created...
Current stream ARN for TestTableForStreams: arn:aws:dynamodb:us-east-2:123456789012:table/
TestTableForStreams/stream/2018-03-20T16:49:55.208
Stream enabled: true
Update view type: NEW_AND_OLD_IMAGES

Performing write activities on TestTableForStreams
Processing item 1 of 100
Processing item 2 of 100
Processing item 3 of 100
...
Processing item 100 of 100

Shard: {ShardId: shardId-1234567890-..., SequenceNumberRange: {StartingSequenceNumber:
01234567890..., }, }

```

```

Shard iterator: EjYFEkX2a26eVTWe...
    ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys:
    {Id={N: 1,}},NewImage: {Message={S: New item!,}, Id={N: 1,}},SequenceNumber:
    100000000003218256368,SizeBytes: 24,StreamViewType: NEW_AND_OLD_IMAGES}
        {ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys: {Id={N:
    1,}},NewImage: {Message={S: This item has changed,}, Id={N: 1,}},OldImage:
    {Message={S: New item!,}, Id={N: 1,}},SequenceNumber: 200000000003218256412,SizeBytes:
    56,StreamViewType: NEW_AND_OLD_IMAGES}
            {ApproximateCreationDateTime: Tue Mar 20 09:50:00 PDT 2018,Keys: {Id={N:
    1,}},OldImage: {Message={S: This item has changed,}, Id={N: 1,}},SequenceNumber:
    300000000003218256413,SizeBytes: 36,StreamViewType: NEW_AND_OLD_IMAGES}
...
Deleting the table...
Demo complete

```

Example

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.amazonaws.AmazonClientException;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreams;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreamsClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeAction;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.AttributeValueUpdate;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeStreamRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeStreamResult;
import com.amazonaws.services.dynamodbv2.model.DescribeTableResult;
import com.amazonaws.services.dynamodbv2.model.GetRecordsRequest;
import com.amazonaws.services.dynamodbv2.model.GetRecordsResult;
import com.amazonaws.services.dynamodbv2.model.GetShardIteratorRequest;
import com.amazonaws.services.dynamodbv2.model.GetShardIteratorResult;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.Record;

```

```

import com.amazonaws.services.dynamodbv2.model.Shard;
import com.amazonaws.services.dynamodbv2.model.ShardIteratorType;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.util.TableUtils;

public class StreamsLowLevelDemo {

    public static void main(String args[]) throws InterruptedException {

        AmazonDynamoDB dynamoDBClient = AmazonDynamoDBClientBuilder
            .standard()
            .withRegion(Regions.US_EAST_2)
            .withCredentials(new DefaultAWSCredentialsProviderChain())
            .build();

        AmazonDynamoDBStreams streamsClient =
            AmazonDynamoDBStreamsClientBuilder
                .standard()
                .withRegion(Regions.US_EAST_2)
                .withCredentials(new DefaultAWSCredentialsProviderChain())
                .build();

        // Create a table, with a stream enabled
        String tableName = "TestTableForStreams";

        ArrayList<AttributeDefinition> attributeDefinitions = new ArrayList<>(
            Arrays.asList(new AttributeDefinition()
                .withAttributeName("Id")
                .withAttributeType("N")));

        ArrayList<KeySchemaElement> keySchema = new ArrayList<>(
            Arrays.asList(new KeySchemaElement()
                .withAttributeName("Id")
                .withKeyType(KeyType.HASH))); // Partition key

        StreamSpecification streamSpecification = new StreamSpecification()
            .withStreamEnabled(true)
            .withStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES);

        CreateTableRequest createTableRequest = new
        CreateTableRequest().withTableName(tableName)
            .withKeySchema(keySchema).withAttributeDefinitions(attributeDefinitions)
            .withProvisionedThroughput(new ProvisionedThroughput()
                .withReadCapacityUnits(10L)
                .withWriteCapacityUnits(10L))
            .withStreamSpecification(streamSpecification);

        System.out.println("Issuing CreateTable request for " + tableName);
        dynamoDBClient.createTable(createTableRequest);
        System.out.println("Waiting for " + tableName + " to be created...");

        try {
            TableUtils.waitUntilActive(dynamoDBClient, tableName);
        } catch (AmazonClientException e) {
            e.printStackTrace();
        }

        // Print the stream settings for the table
        DescribeTableResult describeTableResult = dynamoDBClient.describeTable(tableName);
        String streamArn = describeTableResult.getTable().getLatestStreamArn();
        System.out.println("Current stream ARN for " + tableName + ": " +
            describeTableResult.getTable().getLatestStreamArn());
        StreamSpecification streamSpec =
        describeTableResult.getTable().getStreamSpecification();
        System.out.println("Stream enabled: " + streamSpec.getStreamEnabled());
    }
}

```

```

System.out.println("Update view type: " + streamSpec.getStreamViewType());
System.out.println();

// Generate write activity in the table

System.out.println("Performing write activities on " + tableName);
int maxItemCount = 100;
for (Integer i = 1; i <= maxItemCount; i++) {
    System.out.println("Processing item " + i + " of " + maxItemCount);

    // Write a new item
    Map<String, AttributeValue> item = new HashMap<>();
    item.put("Id", new AttributeValue().withN(i.toString()));
    item.put("Message", new AttributeValue().withS("New item!"));
    dynamoDBClient.putItem(tableName, item);

    // Update the item
    Map<String, AttributeValue> key = new HashMap<>();
    key.put("Id", new AttributeValue().withN(i.toString()));
    Map<String, AttributeValueUpdate> attributeUpdates = new HashMap<>();
    attributeUpdates.put("Message", new AttributeValueUpdate()
        .withAction(AttributeAction.PUT)
        .WithValue(new AttributeValue()
            .withS("This item has changed")));
    dynamoDBClient.updateItem(tableName, key, attributeUpdates);

    // Delete the item
    dynamoDBClient.deleteItem(tableName, key);
}

// Get all the shard IDs from the stream. Note that DescribeStream returns
// the shard IDs one page at a time.
String lastEvaluatedShardId = null;

do {
    DescribeStreamResult describeStreamResult = streamsClient.describeStream(
        new DescribeStreamRequest()
            .withStreamArn(streamArn)
            .withExclusiveStartShardId(lastEvaluatedShardId));
    List<Shard> shards = describeStreamResult.getStreamDescription().getShards();

    // Process each shard on this page

    for (Shard shard : shards) {
        String shardId = shard.getShardId();
        System.out.println("Shard: " + shard);

        // Get an iterator for the current shard

        GetShardIteratorRequest getShardIteratorRequest = new
GetShardIteratorRequest()
            .withStreamArn(streamArn)
            .withShardId(shardId)
            .withShardIteratorType(ShardIteratorType.TRIM_HORIZON);
        GetShardIteratorResult getShardIteratorResult =
            streamsClient.getShardIterator(getShardIteratorRequest);
        String currentShardIter = getShardIteratorResult.getShardIterator();

        // Shard iterator is not null until the Shard is sealed (marked as
READ_ONLY).
        // To prevent running the loop until the Shard is sealed, which will be on
average
        // 4 hours, we process only the items that were written into DynamoDB and
then exit.
        int processedRecordCount = 0;
}
}

```

```
        while (currentShardIter != null && processedRecordCount < maxItemCount) {
            System.out.println("    Shard iterator: " +
currentShardIter.substring(380));

            // Use the shard iterator to read the stream records

            GetRecordsResult getRecordsResult = streamsClient.getRecords(new
GetRecordsRequest()
                .withShardIterator(currentShardIter));
            List<Record> records = getRecordsResult.getRecords();
            for (Record record : records) {
                System.out.println("        " + record.getDynamodb());
            }
            processedRecordCount += records.size();
            currentShardIter = getRecordsResult.getNextShardIterator();
        }

        // If LastEvaluatedShardId is set, then there is
        // at least one more page of shard IDs to retrieve
        lastEvaluatedShardId =
describeStreamResult.getStreamDescription().getLastEvaluatedShardId();

    } while (lastEvaluatedShardId != null);

    // Delete the table
    System.out.println("Deleting the table...");
    dynamoDBClient.deleteTable(tableName);

    System.out.println("Demo complete");
}
```

Cross-Region Replication

You can create tables that are automatically replicated across two or more AWS Regions, with full support for multimaster writes. This gives you the ability to build fast, massively scaled applications for a global user base without having to manage the replication process. For more information, see [Global Tables: Multi-Region Replication with DynamoDB \(p. 624\)](#).

DynamoDB Streams and AWS Lambda Triggers

Topics

- [Tutorial: Process New Items with DynamoDB Streams and Lambda \(p. 595\)](#)
- [Best Practices with Lambda \(p. 602\)](#)

Amazon DynamoDB is integrated with AWS Lambda so that you can create *triggers*—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with an AWS Lambda function that you write. Immediately after an item in the table is modified, a new record appears in the table's stream. AWS Lambda polls the stream and invokes your Lambda function synchronously when it detects new stream records.

The Lambda function can perform any actions you specify, such as sending a notification or initiating a workflow. For example, you can write a Lambda function to simply copy each stream record to persistent storage, such as Amazon Simple Storage Service (Amazon S3), to create a permanent audit trail of write

activity in your table. Or suppose that you have a mobile gaming app that writes to a GameScores table. Whenever the TopScore attribute of the GameScores table is updated, a corresponding stream record is written to the table's stream. This event could then trigger a Lambda function that posts a congratulatory message on a social media network. (The function would simply ignore any stream records that are not updates to GameScores or that do not modify the TopScore attribute.)

For more information about AWS Lambda, see the [AWS Lambda Developer Guide](#).

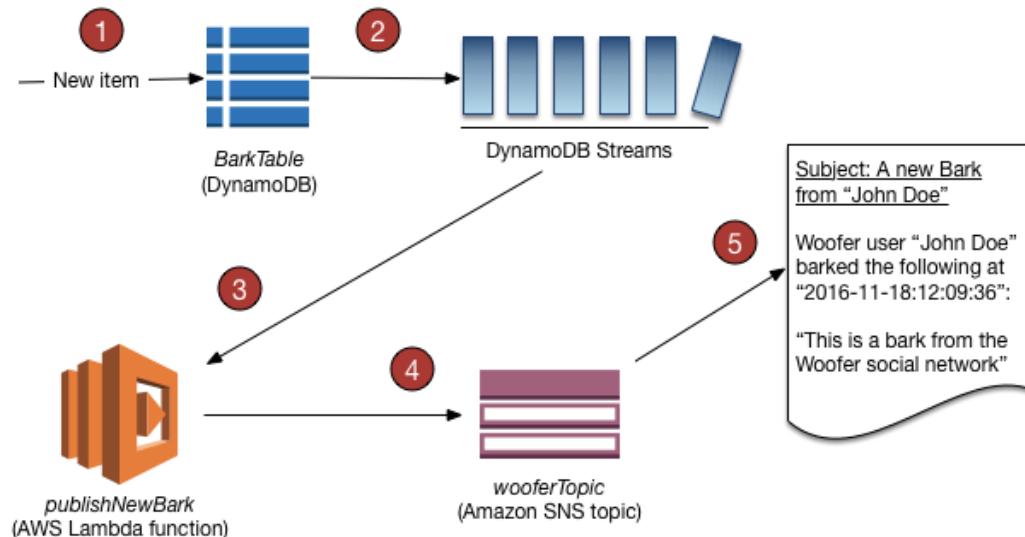
Tutorial: Process New Items with DynamoDB Streams and Lambda

Topics

- [Step 1: Create a DynamoDB Table with a Stream Enabled \(p. 596\)](#)
- [Step 2: Create a Lambda Execution Role \(p. 596\)](#)
- [Step 3: Create an Amazon SNS Topic \(p. 598\)](#)
- [Step 4: Create and Test a Lambda Function \(p. 598\)](#)
- [Step 5: Create and Test a Trigger \(p. 601\)](#)

In this tutorial, you create an AWS Lambda trigger to process a stream from a DynamoDB table.

The scenario for this tutorial is Woofer, a simple social network. Woofer users communicate using *barks* (short text messages) that are sent to other Woofer users. The following diagram shows the components and workflow for this application.



1. A user writes an item to a DynamoDB table (**BarkTable**). Each item in the table represents a bark.
2. A new stream record is written to reflect that a new item has been added to **BarkTable**.
3. The new stream record triggers an AWS Lambda function (**publishNewBark**).
4. If the stream record indicates that a new item was added to **BarkTable**, the Lambda function reads the data from the stream record and publishes a message to a topic in Amazon Simple Notification Service (Amazon SNS).
5. The message is received by subscribers to the Amazon SNS topic. (In this tutorial, the only subscriber is an email address.)

Before You Begin

This tutorial uses the AWS Command Line Interface AWS CLI. If you have not done so already, follow the instructions in the [AWS Command Line Interface User Guide](#) to install and configure the AWS CLI.

Step 1: Create a DynamoDB Table with a Stream Enabled

In this step, you create a DynamoDB table (`BarkTable`) to store all of the barks from Woofer users. The primary key is composed of `Username` (partition key) and `Timestamp` (sort key). Both of these attributes are of type string.

`BarkTable` has a stream enabled. Later in this tutorial, you create a trigger by associating an AWS Lambda function with the stream.

1. Enter the following command to create the table.

```
aws dynamodb create-table \
    --table-name BarkTable \
    --attribute-definitions AttributeName=Username,AttributeType=S
    AttributeName=Timestamp,AttributeType=S \
    --key-schema AttributeName=Username,KeyType=HASH
    AttributeName=Timestamp,KeyType=RANGE \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

2. In the output, look for the `LatestStreamArn`.

```
...
"LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/stream/timestamp
..."
```

Make a note of the `region` and the `accountID`, because you need them for the other steps in this tutorial.

Step 2: Create a Lambda Execution Role

In this step, you create an AWS Identity and Access Management (IAM) role (`WooferLambdaRole`) and assign permissions to it. This role is used by the Lambda function that you create in [Step 4: Create and Test a Lambda Function \(p. 598\)](#).

You also create a policy for the role. The policy contains all of the permissions that the Lambda function needs at runtime.

1. Create a file named `trust-relationship.json` with the following contents.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

2. Enter the following command to create `WooferLambdaRole`.

```
aws iam create-role --role-name WooferLambdaRole \
--path "/service-role/" \
--assume-role-policy-document file://trust-relationship.json
```

3. Create a file named `role-policy.json` with the following contents. (Replace `region` and `accountID` with your AWS Region and account ID.)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:region:accountID:function:publishNewBark*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:region:accountID:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb>ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:region:accountID:table/BarkTable/stream/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

The policy has four statements that allow `WooferLambdaRole` to do the following:

- Execute a Lambda function (`publishNewBark`). You create the function later in this tutorial.
 - Access Amazon CloudWatch Logs. The Lambda function writes diagnostics to CloudWatch Logs at runtime.
 - Read data from the DynamoDB stream for `BarkTable`.
 - Publish messages to Amazon SNS.
4. Enter the following command to attach the policy to `WooferLambdaRole`.

```
aws iam put-role-policy --role-name WooferLambdaRole \
```

```
--policy-name WooferLambdaRolePolicy \
--policy-document file://role-policy.json
```

Step 3: Create an Amazon SNS Topic

In this step, you create an Amazon SNS topic (`wooferTopic`) and subscribe an email address to it. Your Lambda function uses this topic to publish new barks from Woofer users.

1. Enter the following command to create a new Amazon SNS topic.

```
aws sns create-topic --name wooferTopic
```

2. Enter the following command to subscribe an email address to `wooferTopic`. (Replace `region` and `accountID` with your AWS Region and account ID, and replace `example@example.com` with a valid email address.)

```
aws sns subscribe \
  --topic-arn arn:aws:sns:region:accountID:wooferTopic \
  --protocol email \
  --notification-endpoint example@example.com
```

3. Amazon SNS sends a confirmation message to your email address. Choose the **Confirm subscription** link in that message to complete the subscription process.

Step 4: Create and Test a Lambda Function

In this step, you create an AWS Lambda function (`publishNewBark`) to process stream records from `BarkTable`.

The `publishNewBark` function processes only the stream events that correspond to new items in `BarkTable`. The function reads data from such an event, and then invokes Amazon SNS to publish it.

1. Create a file named `publishNewBark.js` with the following contents. Replace `region` and `accountID` with your AWS Region and account ID.

```
'use strict';
var AWS = require("aws-sdk");
var sns = new AWS.SNS();

exports.handler = (event, context, callback) => {

    event.Records.forEach((record) => {
        console.log('Stream record: ', JSON.stringify(record, null, 2));

        if (record.eventName == 'INSERT') {
            var who = JSON.stringify(record.dynamodb.NewImage.Username.S);
            var when = JSON.stringify(record.dynamodb.NewImage.Timestamp.S);
            var what = JSON.stringify(record.dynamodb.NewImage.Message.S);
            var params = {
                Subject: 'A new bark from ' + who,
                Message: 'Woofer user ' + who + ' barked the following at ' + when + ':'
            \n\n' + what,
                TopicArn: 'arn:aws:sns:region:accountID:wooferTopic'
            };
            sns.publish(params, function(err, data) {
                if (err) {
                    console.error("Unable to send message. Error JSON:",
                    JSON.stringify(err, null, 2));
                }
            });
        }
    });
}
```

```

        } else {
            console.log("Results from sending message: ", JSON.stringify(data,
null, 2));
        }
    });
});
callback(null, `Successfully processed ${event.Records.length} records.`);
};

```

2. Create a zip file to contain `publishNewBark.js`. If you have the `zip` command-line utility, you can enter the following command to do this.

```
zip publishNewBark.zip publishNewBark.js
```

3. When you create the Lambda function, you specify the Amazon Resource Name (ARN) for `WooferLambdaRole`, which you created in [Step 2: Create a Lambda Execution Role \(p. 596\)](#). Enter the following command to retrieve this ARN.

```
aws iam get-role --role-name WooferLambdaRole
```

In the output, look for the ARN for `WooferLambdaRole`.

```

...
"Arn": "arn:aws:iam::region:role/service-role/WooferLambdaRole"
...

```

Enter the following command to create the Lambda function. Replace `roleARN` with the ARN for `WooferLambdaRole`.

```

aws lambda create-function \
--region us-east-1 \
--function-name publishNewBark \
--zip-file fileb://publishNewBark.zip \
--role roleARN \
--handler publishNewBark.handler \
--timeout 5 \
--runtime nodejs10.x

```

4. Now test `publishNewBark` to verify that it works. To do this, you provide input that resembles a real record from DynamoDB Streams.

Create a file named `payload.json` with the following contents.

```
{
    "Records": [
        {
            "eventID": "7de3041dd709b024af6f29e4fa13d34c",
            "eventName": "INSERT",
            "eventVersion": "1.1",
            "eventSource": "aws:dynamodb",
            "awsRegion": "us-west-2",
            "dynamodb": {
                "ApproximateCreationDateTime": 1479499740,
                "Keys": {
                    "Timestamp": {
                        "S": "2016-11-18:12:09:36"

```

```
        },
        "Username": {
            "S": "John Doe"
        }
    },
    "NewImage": {
        "Timestamp": {
            "S": "2016-11-18:12:09:36"
        },
        "Message": {
            "S": "This is a bark from the Woofer social network"
        },
        "Username": {
            "S": "John Doe"
        }
    },
    "SequenceNumber": "1302160000000001596893679",
    "SizeBytes": 112,
    "StreamViewType": "NEW_IMAGE"
},
"eventSourceARN": "arn:aws:dynamodb:us-east-1:123456789012:table/BarkTable/
stream/2016-11-16T20:42:48.104"
}
]
```

Enter the following command to test the publishNewBark function.

```
aws lambda invoke --function-name publishNewBark --payload file://payload.json
output.txt
```

If the test was successful, you will see the following output.

```
{
    "StatusCode": 200
}
```

In addition, the `output.txt` file will contain the following text.

```
"Successfully processed 1 records."
```

You will also receive a new email message within a few minutes.

Note

AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes:

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Logs**.
3. Choose the following log group: /aws/lambda/publishNewBark
4. Choose the latest log stream to view the output (and errors) from the function.

Step 5: Create and Test a Trigger

In [Step 4: Create and Test a Lambda Function \(p. 598\)](#), you tested the Lambda function to ensure that it ran correctly. In this step, you create a *trigger* by associating the Lambda function (`publishNewBark`) with an event source (the `BarkTable` stream).

- When you create the trigger, you need to specify the ARN for the `BarkTable` stream. Enter the following command to retrieve this ARN.

```
aws dynamodb describe-table --table-name BarkTable
```

In the output, look for the `LatestStreamArn`.

```
...
"LatestStreamArn": "arn:aws:dynamodb:region:accountID:table/BarkTable/stream/timestamp
...
```

- Enter the following command to create the trigger. Replace `streamARN` with the actual stream ARN.

```
aws lambda create-event-source-mapping \
    --region us-east-1 \
    --function-name publishNewBark \
    --event-source streamARN \
    --batch-size 1 \
    --starting-position TRIM_HORIZON
```

- Test the trigger. Enter the following command to add an item to `BarkTable`.

```
aws dynamodb put-item \
    --table-name BarkTable \
    --item Username={"S":"Jane
Doe"},Timestamp={"S":"2016-11-18:14:32:17"},Message={"S":"Testing...1...2...3"}
```

You should receive a new email message within a few minutes.

- Open the DynamoDB console and add a few more items to `BarkTable`. You must specify values for the `Username` and `Timestamp` attributes. (You should also specify a value for `Message`, even though it is not required.) You should receive a new email message for each item you add to `BarkTable`.

The Lambda function processes only new items that you add to `BarkTable`. If you update or delete an item in the table, the function does nothing.

Note

AWS Lambda writes diagnostic information to Amazon CloudWatch Logs. If you encounter errors with your Lambda function, you can use these diagnostics for troubleshooting purposes.

- Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
- In the navigation pane, choose **Logs**.
- Choose the following log group: `/aws/lambda/publishNewBark`
- Choose the latest log stream to view the output (and errors) from the function.

Best Practices with Lambda

An AWS Lambda function runs within a *container*—an execution environment that is isolated from other functions. When you run a function for the first time, AWS Lambda creates a new container and begins executing the function's code.

A Lambda function has a *handler* that is executed once per invocation. The handler contains the main business logic for the function. For example, the Lambda function shown in [Step 4: Create and Test a Lambda Function \(p. 598\)](#) has a handler that can process records in a DynamoDB stream.

You can also provide initialization code that runs one time only—after the container is created, but before AWS Lambda executes the handler for the first time. The Lambda function shown in [Step 4: Create and Test a Lambda Function \(p. 598\)](#) has initialization code that imports the SDK for JavaScript in Node.js, and creates a client for Amazon SNS. These objects should only be defined once, outside of the handler.

After the function executes, AWS Lambda might opt to reuse the container for subsequent invocations of the function. In this case, your function handler might be able to reuse the resources that you defined in your initialization code. (You cannot control how long AWS Lambda will retain the container, or whether the container will be reused at all.)

For DynamoDB triggers using AWS Lambda, we recommend the following:

- AWS service clients should be instantiated in the initialization code, not in the handler. This allows AWS Lambda to reuse existing connections, for the duration of the container's lifetime.
- In general, you do not need to explicitly manage connections or implement connection pooling because AWS Lambda manages this for you.

For more information, see [Best Practices for Working with AWS Lambda Functions](#) in the *AWS Lambda Developer Guide*.

On-Demand Backup and Restore for DynamoDB

You can create on-demand backups for your Amazon DynamoDB tables or enable continuous backups with point-in-time recovery. For information about continuous backups with point-in-time recovery, see [Point-in-Time Recovery for DynamoDB \(p. 618\)](#).

You can use the DynamoDB on-demand backup capability to create full backups of your tables for long-term retention and archival for regulatory compliance needs. You can back up and restore your table data anytime with a single click on the AWS Management Console or with a single API call. Backup and restore actions execute with zero impact on table performance or availability.

The on-demand backup and restore process scales without degrading the performance or availability of your applications. It uses a new and unique distributed technology that lets you complete backups in seconds regardless of table size. You can create backups that are consistent within seconds across thousands of partitions without worrying about schedules or long-running backup processes. All backups are cataloged, easily discoverable, and retained until explicitly deleted.

In addition, on-demand backup and restore operations don't affect performance or API latencies. Backups are preserved regardless of table deletion. For more information, see [Back Up and Restore DynamoDB Tables: How It Works \(p. 603\)](#).

You can create table backups using the console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. For more information, see [Backing Up a DynamoDB Table \(p. 605\)](#).

For information about restoring a table from a backup, see [Restoring a DynamoDB Table from a Backup \(p. 607\)](#).

DynamoDB on-demand backups are available at no additional cost beyond the normal pricing that's associated with backup storage size. For more information about AWS Region availability and pricing, see [Amazon DynamoDB pricing](#).

Topics

- [Back Up and Restore DynamoDB Tables: How It Works \(p. 603\)](#)
- [Backing Up a DynamoDB Table \(p. 605\)](#)
- [Restoring a DynamoDB Table from a Backup \(p. 607\)](#)
- [Deleting a DynamoDB Table Backup \(p. 612\)](#)
- [Using IAM with DynamoDB Backup and Restore \(p. 614\)](#)

Back Up and Restore DynamoDB Tables: How It Works

You can use the on-demand backup feature to create full backups of your Amazon DynamoDB tables. This section provides an overview of what happens during the backup and restore process.

Backups

When you create an on-demand backup, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot. Backup requests are processed instantaneously and become available for restore within minutes.

Note

Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that can be taken.

All backups in DynamoDB work without consuming any provisioned throughput on the table.

DynamoDB backups do not guarantee causal consistency across items; however, the skew between updates in a backup is usually much less than a second.

While a backup is in progress, you can't do the following:

- Pause or cancel the backup operation.
- Delete the source table of the backup.
- Disable backups on a table if a backup for that table is in progress.

You can schedule periodic or future backups by using AWS Lambda functions. For more information, see the blog post [A serverless solution to schedule your Amazon DynamoDB On-Demand Backup](#).

If you don't want to create scheduling scripts and cleanup jobs, you can use AWS Backup to create backup plans with schedules and retention policies for your DynamoDB tables. AWS Backup executes the backups and deletes them when they expire. For more information, see the [AWS Backup Developer Guide](#).

If you're using the console, any backups created using AWS Backup are listed on the **Backups** tab with the **Backup type** set to **AWS**.

Note

You can't delete backups using the DynamoDB console. To manage these backups, use the AWS Backup console.

To learn how to perform a backup, see [Backing Up a DynamoDB Table \(p. 605\)](#).

Restores

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore from your DynamoDB backup, or you can configure the destination table settings. When you do a restore, you can change the following table settings:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

Important

When you do a full table restore, the destination table is set with the same provisioned read capacity units and write capacity units as the source table, as recorded at the time the backup was requested. The restore process also restores the local secondary indexes and the global secondary indexes.

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the backup resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data that you transfer out of the source Region and for restoring to a new table in the destination Region.

Note

Cross-Region restore is not supported if the source or destination Region is Asia Pacific (Hong Kong) or Middle East (Bahrain).

Restores can be faster and more cost-efficient if you choose to exclude some or all secondary indexes from being created on the new restored table.

You must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings

You can only restore the entire table data to a new table from a backup. You can write to the restored table only after it becomes active.

Note

You can't overwrite an existing table during a restore operation.

The time it takes you to restore a table varies based on multiple factors. The restore times are not always correlated directly to the size of the table. For example, because of parallelization, restoring a 300 GB table could take the same amount of time as restoring a 3 GB table.

The following are some considerations for restore times:

- You restore backups to a new table. It can take up to 20 minutes (even if the table is empty) to perform all the actions to create the new table and initiate the restore process.
- For tables with even data distribution across your primary keys, the restore time is proportional to the largest single partition by item count and not the overall table size. For the largest partitions with billions of items, a restore could take less than 10 hours.
- If your source table contains data with significant skew, the time to restore might increase. For example, if your table's primary key is using the month of the year for partitioning, and all your data is from the month of December, you have skewed data.

To learn how to perform a restore, see [Restoring a DynamoDB Table from a Backup \(p. 607\)](#).

You can use IAM policies for access control. For more information, see [Using IAM with DynamoDB Backup and Restore \(p. 614\)](#).

All backup and restore console and API actions are captured and recorded in AWS CloudTrail for logging, continuous monitoring, and auditing.

Backing Up a DynamoDB Table

This section describes how to use the Amazon DynamoDB console or the AWS Command Line Interface to back up a table.

Topics

- [Creating a Table Backup \(Console\) \(p. 606\)](#)

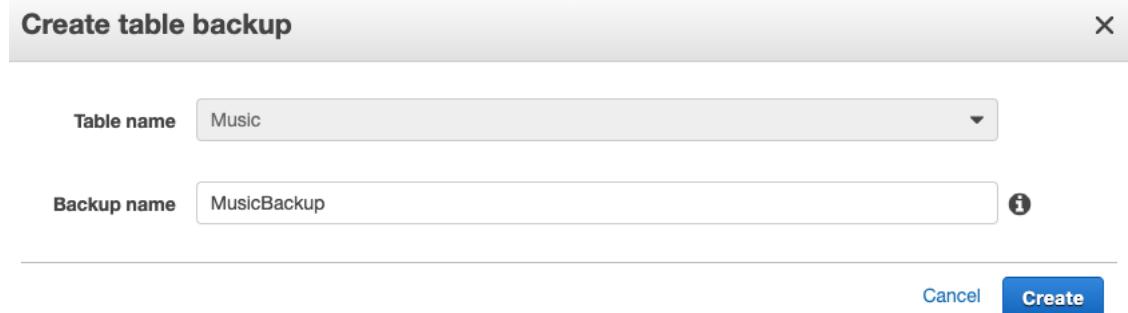
- [Creating a Table Backup \(AWS CLI\) \(p. 606\)](#)

Creating a Table Backup (Console)

Follow these steps to create a backup named `MusicBackup` for an existing `Music` table using the AWS Management Console.

To create a table backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. You can create a backup by doing one of the following:
 - On the **Backups** tab of the `Music` table, choose **Create backup**.
 - In the navigation pane on the left side of the console, choose **Backups**. Then choose **Create backup**.
3. Make sure that `Music` is the table name, and enter `MusicBackup` for the backup name. Then, choose **Create** to create the backup.



Note

If you create backups using the **Backups** section in the navigation pane, the table isn't preselected for you. You have to manually choose the source table name for the backup.

While the backup is being created, the backup status is set to **Creating**. After the backup is complete, the backup status changes to **Available**.

The screenshot shows the "Backups" tab in the DynamoDB console. At the top are buttons for "Create backup", "Restore backup", and "Delete backup", followed by a refresh icon. Below is a search bar "Filter by backup name" and a "Time Range" dropdown set to "Last 30 Days". A "Backup type" dropdown is set to "All backups". The main area displays a table with columns "Backup name", "Status", and "Creation time". One backup is listed: "MusicBackup" (Status: Available, Creation time: February 12, 2020 at 3:04:59 PM UTC-8).

Creating a Table Backup (AWS CLI)

Follow these steps to create a backup for an existing table `Music` using the AWS CLI.

To create a table backup

- Create a backup with the name `MusicBackup` for the `Music` table.

```
aws dynamodb create-backup --table-name Music \
--backup-name MusicBackup
```

While the backup is being created, the backup status is set to `CREATING`.

```
{
    "BackupDetails": {
        "BackupName": "MusicBackup",
        "BackupArn": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc",
        "BackupStatus": "CREATING",
        "BackupCreationDateTime": 1489602797.149
    }
}
```

After the backup is complete, its `BackupStatus` should change to `AVAILABLE`. To confirm this, use the `describe-backup` command. You can get the input value of `backup-arn` from the output of the previous step or by using the `list-backups` command.

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-
b308cd7d
```

To keep track of your backups, you can use the `list-backups` command. It lists all your backups that are in `CREATING` or `AVAILABLE` status.

```
aws dynamodb list-backups
```

The `list-backups` command and the `describe-backup` command are useful to check information about the source table of the backup.

Restoring a DynamoDB Table from a Backup

This section describes how to restore a table from a backup using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Note

If you want to use the AWS CLI, you must configure it first. For more information, see [Accessing DynamoDB \(p. 54\)](#).

Topics

- [Restoring a Table from a Backup \(Console\) \(p. 607\)](#)
- [Restoring a Table from a Backup \(AWS CLI\) \(p. 610\)](#)

Restoring a Table from a Backup (Console)

The following procedure shows how to restore the `Music` table by using the `MusicBackup` file that is created in the [Backing Up a DynamoDB Table \(p. 605\)](#) tutorial.

Note

This procedure assumes that the **Music** table no longer exists before restoring it using the **MusicBackup** file.

To restore a table from a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose **MusicBackup**.

The screenshot shows the AWS Management Console interface for DynamoDB Backups. At the top, there are three buttons: 'Create backup' (blue), 'Restore backup' (grayed out), and 'Delete backup'. Below the buttons is a search bar labeled 'Filter by backup name' with a magnifying glass icon and a clear button 'X'. To the right of the search bar are 'Time Range' and 'Last 30 Days' dropdown menus. Underneath these are 'Backup type' and 'All backups' dropdown menus. On the far right is a refresh icon. The main area displays a table with one backup entry. The table has columns for 'Backup name', 'Status', and 'Creation time'. The entry shows 'MusicBackup' as the backup name, 'Available' as the status, and 'February 12, 2020 at 3:04:59 PM UTC-8' as the creation time. Below the table are navigation arrows: '< < > >'.

4. Choose **Restore backup**.

The screenshot is identical to the previous one, showing the AWS Management Console interface for DynamoDB Backups. The 'Restore backup' button is highlighted with a red circle. The rest of the interface, including the table with the 'MusicBackup' entry, remains the same.

5. Enter **Music** as the new table name. Confirm the backup name and other backup details. Then choose **Restore table** to start the restore process.

Note

You can restore the table to the same AWS Region or to a different Region from where the backup resides. You can also exclude secondary indexes from being created on the new restored table. In addition, you can specify a different encryption mode.

Restore table from backup



New table name*

- Restore entire table data
 - Restored table will include all local secondary indexes and global secondary indexes.
- Restore without secondary indexes
 - Restored table will exclude the local secondary indexes and global secondary indexes. Note: Restores can be faster and more cost efficient if you choose to exclude secondary indexes from being created.

Cross region restore

- Same region
 - Restore the table to the same AWS Region.
- Cross region
 - Restore the table to a different AWS Region.

Select the encryption type

DEFAULT

The key is owned by Amazon DynamoDB. You are not charged any fee for using these CMKs.

KMS - Customer managed CMK

The key is stored in your account that you create, own, and manage. AWS Key Management Service (KMS) charges apply. [Learn more](#)

KMS - AWS managed CMK

The key is stored in your account and is managed by AWS Key Management Service (KMS). AWS KMS charges apply.

Backup table details

| | |
|----------------------------------|---|
| Original table name | Music |
| Backup name | MusicBackup |
| Backup ARN | arn:aws:dynamodb:eu-north-1:063317358631:table/Music/backup/01581548699157-81ef0887 |
| Primary partition key | Artist |
| Sort key | SongTitle |
| Read/write capacity mode | Provisioned |
| Provisioned read capacity units | 5 |
| Provisioned write capacity units | 5 |
| Encryption Type | DEFAULT |
| Encryption Status | - |
| KMS Master Key ARN | Not Applicable |
| Auto Scaling | DISABLED |
| Stream enabled | No |

Indexes

There are no global secondary indexes or local secondary indexes.

Restores can take several hours to complete.

* Required

Cancel

Restore table

The table that is being restored is shown with the status **Creating**. After the restore process is finished, the status of the `Music` table changes to **Active**.

A screenshot of the AWS DynamoDB console. At the top, there are two buttons: "Create table" (blue) and "Delete table" (gray). Below them is a search bar with the placeholder "Filter by table name" and a clear button "X". To the right of the search bar is a dropdown menu with the first item "Ch...". The main area shows a table with two columns: "Name" and "Status". There is one row visible, representing the "Music" table. The "Name" column shows a blue circular icon followed by the text "Music". The "Status" column shows the text "Active" in green. The entire interface has a light gray background.

| Name | Status |
|-------|--------|
| Music | Active |

Restoring a Table from a Backup (AWS CLI)

Follow these steps to use the AWS CLI to restore the `Music` table using the `MusicBackup` that is created in the [Backing Up a DynamoDB Table \(p. 605\)](#) tutorial.

To restore a table from a backup

1. Confirm the backup that you want to restore by using the `list-backups` command. This example uses `MusicBackup`.

```
aws dynamodb list-backups
```

To get additional details for the backup, use the `describe-backup` command. You can get the input `backup-arn` from the previous step.

```
aws dynamodb describe-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-
b308cd7d
```

2. Restore the table from the backup. In this case, the `MusicBackup` restores the `Music` table to the same AWS Region.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-
b308cd7d
```

3. Restore the table from the backup with custom table settings. In this case, the `MusicBackup` restores the `Music` table and specifies an encryption mode for the restored table.

Note

The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see [Managing Encrypted Tables in DynamoDB \(p. 818\)](#).

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
```

```
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01581080476474-e177be2 \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-a123-ab1234a1b234
```

You can restore the table to a different AWS Region from where the backup resides.

Note

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores in the same Region as the source table.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see [Command Line Options](#) in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01581080476474-e177be2 \
--sse-specification-override Enabled=true,SSEType=KMS
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01489173575360-b308cd7d \
--billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

Note

Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the restored table.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/backup/01581081403719-db9c1f91 \
--global-secondary-index-override '[]' \
--sse-specification-override Enabled=true,SSEType=KMS
```

Note

The secondary indexes provided should match existing indexes. You cannot create new indexes at the time of restore.

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-from-backup \
--target-table-name Music \
--backup-arn arn:aws:dynamodb:eu-west-1:123456789012:table/Music/backup/01581082594992-303b6239 \
--billing-mode-override PROVISIONED \
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \
--global-secondary-index-override IndexName=singers-index,KeySchema=[{"AttributeName=SingerName,KeyType=HASH"}],Projection="{ProjectionType=KEYS_ONLY}",ProjectionType=KEYS_ONLY \
--sse-specification-override Enabled=true,SSEType=KMS
```

To verify the restore, use the `describe-table` command to describe the `Music` table.

```
aws dynamodb describe-table --table-name Music
```

The table that is being restored from the backup is shown with the status **Creating**. After the restore process is finished, the status of the `Music` table changes to **Active**.

Important

While a restore is in progress, don't modify or delete your IAM role policy; otherwise, unexpected behavior can result. For example, suppose that you removed write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableFromBackup` operation would not be able to write any of the restored data to the table. Note that IAM policies involving source IP restrictions for accessing the target restore table might similarly cause issues.

After the restore operation is complete, you can modify or delete your IAM role policy. If your backup is encrypted with an AWS managed CMK or a customer managed CMK, don't disable or delete the key while a restore is in progress, or the restore will fail.

After the restore operation is complete, you can change the encryption key for the restored table and disable or delete the old key.

Deleting a DynamoDB Table Backup

This section describes how to use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to delete an Amazon DynamoDB table backup.

Note

If you want to use the AWS CLI, you have to configure it first. For more information, see [Using the AWS CLI \(p. 56\)](#).

Topics

- [Deleting a Table Backup \(Console\) \(p. 612\)](#)
- [Deleting a Table Backup \(AWS CLI\) \(p. 613\)](#)

Deleting a Table Backup (Console)

The following procedure shows how to use the console to delete the `MusicBackup` that is created in the [Backing Up a DynamoDB Table \(p. 605\)](#) tutorial.

To delete a backup

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose `MusicBackup`.

Backup type All backups

Backup name Status Creation time

MusicBackup Available February 12, 2020 at 3:04:59 PM UTC-8

4. Choose **Delete backup**:

Backup type All backups

Backup name Status Creation time

MusicBackup Available February 12, 2020 at 3:04:59 PM UTC-8

Choose **Delete** to delete the backup.

Delete backup

X

Are you sure you want to delete the backup "MusicBackup" of source table "Music"?

Cancel

Delete

Deleting a Table Backup (AWS CLI)

The following example deletes a backup for an existing table `Music` table using the AWS CLI.

```
aws dynamodb delete-backup \
--backup-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music/
backup/01489602797149-73d8d5bc
```

Using IAM with DynamoDB Backup and Restore

You can use AWS Identity and Access Management (IAM) to restrict Amazon DynamoDB backup and restore actions for some resources. The `CreateBackup` and `RestoreTableFromBackup` APIs operate on a per-table basis.

For more information about using IAM policies in DynamoDB, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 828\)](#).

The following are examples of IAM policies that you can use to configure specific backup and restore functionality in DynamoDB.

Example 1: Allow the `CreateBackup` and `RestoreTableFromBackup` Actions

The following IAM policy grants permissions to allow the `CreateBackup` and `RestoreTableFromBackup` DynamoDB actions on all tables:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:CreateBackup",  
                "dynamodb:RestoreTableFromBackup",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Important

DynamoDB write permissions are necessary for restore functionality.

Example 2: Allow `CreateBackup` and Deny `RestoreTableFromBackup`

The following IAM policy grants permissions for the `CreateBackup` action and denies the `RestoreTableFromBackup` action:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb:CreateBackup"],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": ["dynamodb:RestoreTableFromBackup"],  
            "Resource": "*"  
        }  
    ]  
}
```

```
{  
    "Effect": "Deny",  
    "Action": [ "dynamodb:RestoreTableFromBackup" ],  
    "Resource": "*"  
}  
]  
}
```

Example 3: Allow ListBackups and Deny CreateBackup and RestoreTableFromBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `CreateBackup` and `RestoreTableFromBackup` actions:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [ "dynamodb>ListBackups" ],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb>CreateBackup",  
                "dynamodb>RestoreTableFromBackup"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example 4: Allow ListBackups and Deny DeleteBackup

The following IAM policy grants permissions for the `ListBackups` action and denies the `DeleteBackup` action:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [ "dynamodb>ListBackups" ],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": [ "dynamodb>DeleteBackup" ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example 5: Allow RestoreTableFromBackup and DescribeBackup for All Resources and Deny DeleteBackup for a Specific Backup

The following IAM policy grants permissions for the `RestoreTableFromBackup` and `DescribeBackup` actions and denies the `DeleteBackup` action for a specific backup resource:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeBackup",  
                "dynamodb:RestoreTableFromBackup",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": "*"  
        },  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb:DeleteBackup"  
            ],  
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/Music/  
backup/01489173575360-b308cd7d"  
        }  
    ]  
}
```

Important

DynamoDB write permissions are necessary for restore functionality.

Example 6: Allow CreateBackup for a Specific Table

The following IAM policy grants permissions for the `CreateBackup` action on the `Movies` table only:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb>CreateBackup"],  
            "Resource": [  
                "arn:aws:dynamodb:us-east-1:123456789012:table/Movies"  
            ]  
        }  
    ]  
}
```

Example 7: Allow ListBackups

The following IAM policy grants permissions for the `ListBackups` action:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb>ListBackups"],  
            "Resource": "*"  
        }  
    ]  
}
```

Important

You cannot grant permissions for the `ListBackups` action on a specific table.

Point-in-Time Recovery for DynamoDB

You can create on-demand backups of your Amazon DynamoDB tables, or you can enable continuous backups using point-in-time recovery. For more information about on-demand backups, see [On-Demand Backup and Restore for DynamoDB \(p. 603\)](#).

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point-in-time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your table.

In addition, point-in-time operations don't affect performance or API latencies. For more information, see [Point-in-Time Recovery: How It Works \(p. 618\)](#).

You can restore a DynamoDB table to a point in time using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the DynamoDB API. For more information, see [Restoring a DynamoDB Table to a Point in Time \(p. 620\)](#).

For more information about AWS Region availability and pricing, see [Amazon DynamoDB pricing](#).

Topics

- [Point-in-Time Recovery: How It Works \(p. 618\)](#)
- [Before You Begin Using Point-in-Time Recovery \(p. 619\)](#)
- [Restoring a DynamoDB Table to a Point in Time \(p. 620\)](#)

Point-in-Time Recovery: How It Works

Amazon DynamoDB point-in-time recovery (PITR) provides automatic backups of your DynamoDB table data. This section provides an overview of how the process works in DynamoDB.

You can enable point-in-time recovery using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the DynamoDB API. When enabled, point-in-time recovery provides continuous backups until you explicitly turn it off. For more information, see [Restoring a DynamoDB Table to a Point in Time \(p. 620\)](#).

After you enable point-in-time recovery, you can restore to any point in time within `EarliestRestorableDateTime` and `LatestRestorableDateTime`. `LatestRestorableDateTime` is typically 5 minutes before the current time.

Note

The point-in-time recovery process always restores to a new table.

For `EarliestRestorableDateTime`, you can restore your table to any point in time during the last 35 days. The retention period is a fixed 35 days (5 calendar weeks) and can't be modified. Any number of users can execute up to four concurrent restores (any type of restore) in a given account.

Important

If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.

When you restore using point-in-time recovery, DynamoDB restores your table data to the state based on the selected date and time (day:hour:minute:second) to a new table.

You restore a table without consuming any provisioned throughput on the table. You can do a full table restore using point-in-time recovery, or you can configure the destination table settings. You can change the following table settings on the restored table:

- Global secondary indexes (GSIs)
- Local secondary indexes (LSIs)
- Billing mode
- Provisioned read and write capacity
- Encryption settings

Important

When you do a full table restore, all table settings for the restored table come from the current settings of the source table at the time of the restore. For example, suppose that a table's provisioned throughput was recently lowered to 50 read capacity units and 50 write capacity units. You then restore the table's state to three weeks ago, at which time its provisioned throughput was set to 100 read capacity units and 100 write capacity units. In this case, DynamoDB restores your table data to that point in time, but uses the current provisioned throughput (50 read capacity units and 50 write capacity units).

You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.

Note

Cross-Region restore is not supported if the source or destination Region is Asia Pacific (Hong Kong) or Middle East (Bahrain).

Restores can be faster and more cost-efficient if you exclude some or all indexes from being created on the restored table.

You must manually set the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Point-in-time recovery settings

The time it takes you to restore a table varies based on multiple factors. The point-in-time restore times are not always correlated directly to the size of the table. For more information, see [Restores \(p. 604\)](#).

Before You Begin Using Point-in-Time Recovery

Before you enable point-in-time recovery (PITR) on an Amazon DynamoDB table, consider the following:

- If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.
- If you delete a table with point-in-time recovery enabled, a system backup is automatically created and is retained for 35 days (at no additional cost). System backups allow you to restore the deleted table to the state it was in just before the point of deletion. All system backups follow a standard naming convention: `table-name$DeletedTableBackup`.
- You can enable point-in-time recovery on each local replica of a global table. When you restore the table, the backup restores to an independent table that is not part of the global table. If you are using [Version 2019.11.21 \(Current\) \(p. 625\)](#) of global tables, you can create a new global table from the restored table. For more information, see [Global Tables: How It Works \(p. 626\)](#).
- You can also restore your DynamoDB table data across AWS Regions such that the restored table is created in a different Region from where the source table resides. You can do cross-Region restores between AWS commercial Regions, AWS China Regions, and AWS GovCloud (US) Regions. You pay only for the data you transfer out of the source Region and for restoring to a new table in the destination Region.
- AWS CloudTrail logs all console and API actions for point-in-time recovery to enable logging, continuous monitoring, and auditing. For more information, see [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 877\)](#).

Restoring a DynamoDB Table to a Point in Time

Amazon DynamoDB point-in-time recovery (PITR) provides continuous backups of your DynamoDB table data. You can restore a table to a point in time using the DynamoDB console or the AWS Command Line Interface (AWS CLI).

If you want to use the AWS CLI, you must configure it first. For more information, see [Accessing DynamoDB \(p. 54\)](#).

Topics

- [Restoring a Table to a Point in Time \(Console\) \(p. 620\)](#)
- [Restoring a Table to a Point in Time \(AWS CLI\) \(p. 621\)](#)

Restoring a Table to a Point in Time (Console)

The following example demonstrates how to use the DynamoDB console to restore an existing table named `Music` to a point in time.

Note

This procedure assumes that you have enabled point-in-time recovery. To enable it for the `Music` table, on the **Overview** tab, in the **Table details** section, choose **Enable for Point-in-time recovery**.

To restore a table to a point in time

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. In the list of tables, choose the `Music` table.
4. On the **Backups** tab of the `Music` table, in the **Point-in-time recovery** section, choose **Restore to point-in-time**.

5. For the new table name, enter **MusicMinutesAgo**.

Note

You can restore the table to the same AWS Region or to a different Region from where the source table resides. You can also exclude secondary indexes from being created on the restored table. In addition, you can specify a different encryption mode.

6. To confirm the restorable time, set the **Restore date and time** to the **Latest restore date**. Then choose **Restore table** to start the restore process.

Note

You can restore to any point in time within **Earliest restore date** and **Latest restore date**. DynamoDB restores your table data to the state based on the selected date and time (day:hour:minute:second).

The table that is being restored is shown with the status **Restoring**. After the restore process is finished, the status of the `Music` table changes to **Active**.

Restoring a Table to a Point in Time (AWS CLI)

The following procedure shows how to use the AWS CLI to restore an existing table named `Music` to a point in time.

Note

This procedure assumes that you have enabled point-in-time recovery. To enable it for the `Music` table, run the following command.

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

To restore a table to a point in time

1. Confirm that point-in-time recovery is enabled for the `Music` table by using the `describe-continuous-backups` command.

```
aws dynamodb describe-continuous-backups \
--table-name Music
```

Continuous backups (automatically enabled on table creation) and point-in-time recovery are enabled.

```
{
    "ContinuousBackupsDescription": {
        "PointInTimeRecoveryDescription": {
            "PointInTimeRecoveryStatus": "ENABLED",
            "EarliestRestorableDateTime": 1519257118.0,
            "LatestRestorableDateTime": 1520018653.01
        },
        "ContinuousBackupsStatus": "ENABLED"
    }
}
```

2. Restore the table to a point in time. In this case, the `Music` table is restored to the `LatestRestorableDateTime` (~5 minutes ago) to the same AWS Region.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
```

```
--use-latest-restorable-time
```

Note

You can also restore to a specific point in time. To do this, run the command using the `--restore-date-time` argument, and specify a timestamp. You can specify any point in time during the last 35 days. For example, the following command restores the table to the `EarliestRestorableDateTime`.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicEarliestRestorableDateTime \
--no-use-latest-restorable-time \
--restore-date-time 1519257118.0
```

Specifying the `--no-use-latest-restorable-time` argument is optional when restoring to a specific point in time.

3. Restore the table to a point in time with custom table settings. In this case, the `Music` table is restored to the `LatestRestorableDateTime` (~5 minutes ago).

You can specify a different encryption mode for the restored table, as follows.

Note

The `sse-specification-override` parameter takes the same values as the `sse-specification-override` parameter used in the `CreateTable` command. To learn more, see [Managing Encrypted Tables in DynamoDB \(p. 818\)](#).

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can restore the table to a different AWS Region from where the source table resides.

Note

- The `sse-specification-override` parameter is mandatory for cross-Region restores but optional for restores to the same Region as the source table.
- The `source-table-arn` parameter must be provided for cross-Region restores.
- When performing a cross-Region restore from the command line, you must set the default AWS Region to the desired destination Region. To learn more, see [Command Line Options](#) in the *AWS Command Line Interface User Guide*.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-arn arn:aws:dynamodb:us-east-1:123456789012:table/Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
--sse-specification-override Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-
abcd-1234-a123-ab1234a1b234
```

You can override the billing mode and the provisioned throughput for the restored table.

```
aws dynamodb restore-table-to-point-in-time \
--source-table-name Music \
--target-table-name MusicMinutesAgo \
--use-latest-restorable-time \
```

```
--billing-mode-override PAY_PER_REQUEST
```

You can exclude some or all secondary indexes from being created on the restored table.

Note

Restores can be faster and more cost-efficient if you exclude some or all secondary indexes from being created on the new restored table.

```
aws dynamodb restore-table-to-point-in-time \  
--source-table-name Music \  
--target-table-name MusicMinutesAgo \  
--use-latest-restorable-time \  
--global-secondary-index-override '[]'
```

You can use a combination of different overrides. For example, you can use a single global secondary index and change provisioned throughput at the same time, as follows.

```
aws dynamodb restore-table-to-point-in-time \  
--source-table-name Music \  
--target-table-name MusicMinutesAgo \  
--billing-mode-override PROVISIONED \  
--provisioned-throughput-override ReadCapacityUnits=100,WriteCapacityUnits=100 \  
--global-secondary-index-override IndexName=singers-  
index,KeySchema=[ {"AttributeName=SingerName,KeyType=HASH"} ],Projection="{ProjectionType=KEYS_ONLY}" \  
 \  
--sse-specification-override Enabled=true,SSEType=KMS \  
--use-latest-restorable-time
```

To verify the restore, use the `describe-table` command to describe the `MusicEarliestRestorableDateTime` table.

```
aws dynamodb describe-table --table-name MusicEarliestRestorableDateTime
```

The table that is being restored is shown with the status **Creating** and restore in progress as **true**. After the restore process is finished, the status of the `MusicEarliestRestorableDateTime` table changes to **Active**.

Important

While a restore is in progress, don't modify or delete the AWS Identity and Access Management (IAM) policies that grant the IAM entity (for example, user, group, or role) permission to perform the restore. Otherwise, unexpected behavior can result. For example, suppose that you remove write permissions for a table while that table is being restored. In this case, the underlying `RestoreTableToPointInTime` operation can't write any of the restored data to the table. IAM policies involving source IP restrictions for accessing the target restore table can similarly cause issues.

You can modify or delete permissions only after the restore operation is completed.

Global Tables: Multi-Region Replication with DynamoDB

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

Amazon DynamoDB global tables provide a fully managed solution for deploying a multiregion, multi-master database, without having to build and maintain your own replication solution. With global tables you can specify the AWS Regions where you want the table to be available. DynamoDB performs all of the necessary tasks to create identical tables in these Regions and propagate ongoing data changes to all of them.

For example, suppose that you have a large customer base spread across three geographic areas—the US East Coast, the US West Coast, and Western Europe. Customers can update their profile information using your application. To satisfy this use case, you need to create three identical DynamoDB tables named `CustomerProfiles`, in three different AWS Regions where the customers are located. These three tables would be entirely separate from each other. Changes to the data in one table would not be reflected in the other tables. Without a managed replication solution, you could write code to replicate data changes among these tables. However, doing this would be a time-consuming and labor-intensive effort.

Instead of writing your own code, you could create a global table consisting of your three Region-specific `CustomerProfiles` tables. DynamoDB would then automatically replicate data changes among those tables so that changes to `CustomerProfiles` data in one Region would seamlessly propagate to the other Regions. In addition, if one of the AWS Regions were to become temporarily unavailable, your customers could still access the same `CustomerProfiles` data in the other Regions.

DynamoDB global tables are ideal for massively scaled applications with globally dispersed users. In such an environment, users expect very fast application performance. Global tables provide automatic multi-master replication to AWS Regions worldwide. They enable you to deliver low-latency data access to your users no matter where they are located.

Note

- There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). We recommend using [Version 2019.11.21 \(Current\) \(p. 625\)](#) of global tables, which enables you to dynamically add new replica tables from a table populated with data. [Version 2019.11.21 \(Current\) \(p. 625\)](#) is more efficient and consumes less write capacity than [Version 2017.11.29 \(p. 638\)](#).
- Region support for global tables [Version 2017.11.29 \(p. 638\)](#) is limited to US East (N. Virginia), US East (Ohio), US West (N. California), US West (Oregon), Europe (Ireland), Europe (London), Europe (Frankfurt), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Tokyo), and Asia Pacific (Seoul).
- If you are using [Version 2019.11.21 \(Current\) \(p. 625\)](#) of global tables and you also use the [Time to Live](#) feature, DynamoDB replicates TTL deletes to all replica tables. The initial TTL delete does not consume write capacity in the region in which the TTL expiry occurs. However, the replicated TTL delete to the replica table(s) consumes a replicated write capacity unit.

when using provisioned capacity, or replicated write when using on-demand capacity mode, in each of the replica regions and applicable charges will apply.

- Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the region where the write is made originally. Transactions are not supported across regions in global tables. For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) regions and perform a `TransactWriteItems` operation in the US East (N. Virginia) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated. Changes will only be replicated to other regions once they have been committed in the source region.

For information about the AWS Region availability and pricing, see [Amazon DynamoDB Pricing](#).

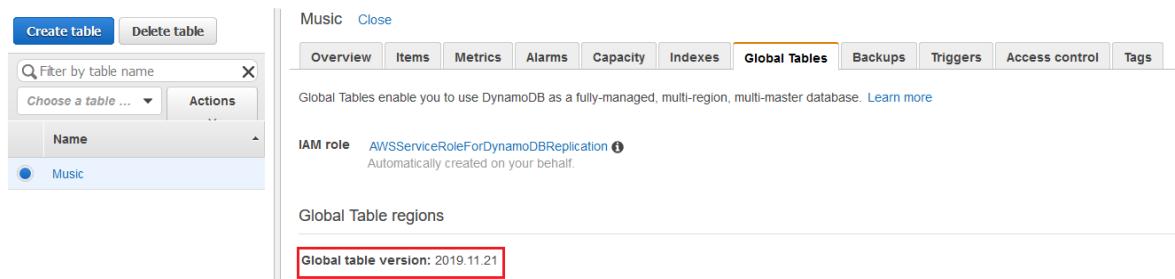
Topics

- [Determining the Version of Global Tables You Are Using \(p. 625\)](#)
- [Version 2019.11.21 \(Current\) \(p. 625\)](#)
- [Version 2017.11.29 \(p. 638\)](#)

Determining the Version of Global Tables You Are Using

To find out which version of global tables you are using, do the following:

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table you want to use.
4. Choose the **Global Tables** tab.
5. The **Global table version** displays the version of global tables in use.



Version 2019.11.21 (Current)

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

Topics

- [Global Tables: How It Works \(p. 626\)](#)
- [Best Practices and Requirements for Managing Global Tables \(p. 627\)](#)
- [Tutorial: Creating a Global Table \(p. 627\)](#)
- [Monitoring Global Tables \(p. 632\)](#)
- [Using IAM with Global Tables \(p. 633\)](#)
- [Updating Global Tables to Version 2019.11.21 \(Current\) \(p. 635\)](#)

Global Tables: How It Works

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

The following sections help you understand the concepts and behavior of global tables in Amazon DynamoDB.

Global Table Concepts

A *global table* is a collection of one or more replica tables, all owned by a single AWS account.

A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per AWS Region. For more information about how to get started with global tables, see [Tutorial: Creating a Global Table \(p. 627\)](#).

When you create a DynamoDB global table, it consists of multiple replica tables (one per Region) that DynamoDB treats as a single unit. Every replica has the same table name and the same primary key schema. When an application writes data to a replica table in one Region, DynamoDB propagates the write to the other replica tables in the other AWS Regions automatically.

You can add replica tables to the global table so that it can be available in additional Regions.

Consistency and Conflict Resolution

Any changes made to any item in any replica table are replicated to all the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within a second.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. If your application only uses eventually consistent reads and only issues reads against one AWS Region, it will work without any modification. However, if your application requires strongly consistent reads, it must perform all of its strongly consistent reads and writes in the same Region. DynamoDB does not support strongly consistent reads across Regions. Therefore, if you write to one Region and read from another Region, the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region.

If applications update the same item in different Regions at about the same time, conflicts can arise. To help ensure eventual consistency, DynamoDB global tables use a *last writer wins* reconciliation between concurrent updates, in which DynamoDB makes a best effort to determine the last writer. With this conflict resolution mechanism, all the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Availability and Durability

If a single AWS Region becomes isolated or degraded, your application can redirect to a different Region and perform reads and writes against a different replica table. You can apply custom business logic to determine when to redirect requests to other Regions.

If a Region becomes isolated or degraded, DynamoDB keeps track of any writes that have been performed but have not yet been propagated to all of the replica tables. When the Region comes back online, DynamoDB resumes propagating any pending writes from that Region to the replica tables in other Regions. It also resumes propagating writes from other replica tables to the Region that is now back online.

Best Practices and Requirements for Managing Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

Using Amazon DynamoDB global tables, you can replicate your table data across AWS Regions. It is important that the replica tables and secondary indexes in your global table have identical write capacity settings to ensure proper replication of data.

Best Practices and Requirements for Managing Capacity

Consider the following when managing capacity settings for replica tables in DynamoDB.

When using the latest version of global tables, you must either use on-demand capacity or enable autoscaling on the table. Using autoscaling or on-demand capacity ensures that you always have sufficient capacity to perform replicated writes to all regions of the global table. The number of replicated write capacity units (rWCUs) or replicated write request units will be equal in all Regions of the global table. For example, suppose that you expect 5 writes per second to your replica table in Ohio and 5 writes per second to your replica table in N. Virginia. In this case, you should expect to consume 10 rWCUs (or 10 replicated write request units, if using on-demand capacity) in both Ohio and N. Virginia. When you use the provisioned capacity mode, you manage your auto scaling policy with `UpdateTableReplicaAutoScaling`. Minimum and maximum throughput and target utilization are established globally for the table and passed to all replicas of the table.

For details about autoscaling and DynamoDB, see [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#).

Tutorial: Creating a Global Table

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

This section describes how to create a global table using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Topics

- [Creating a Global Table \(Console\) \(p. 628\)](#)
- [Creating a Global Table \(AWS CLI\) \(p. 629\)](#)
- [Creating a Global Table \(Java\) \(p. 630\)](#)

Creating a Global Table (Console)

Follow these steps to create a global table using the console. The following example creates a global table with replica tables in United States and Europe.

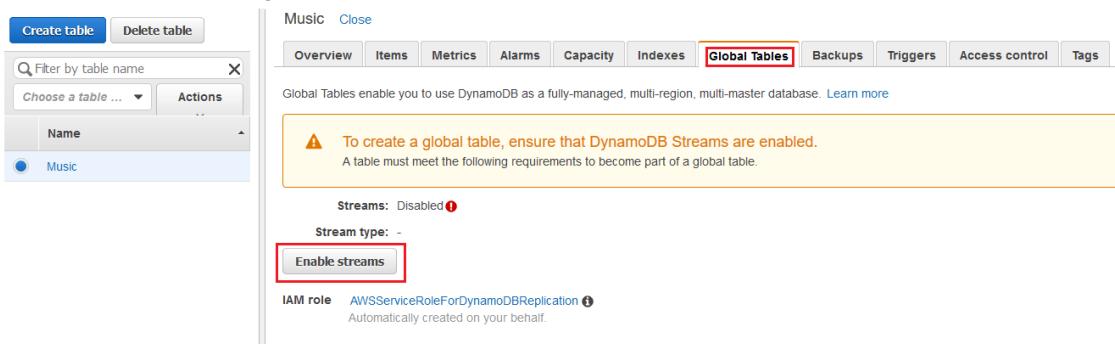
1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>. For this example, choose the us-east-2 (US East Ohio) Region.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**.

For **Table name**, enter **Music**.

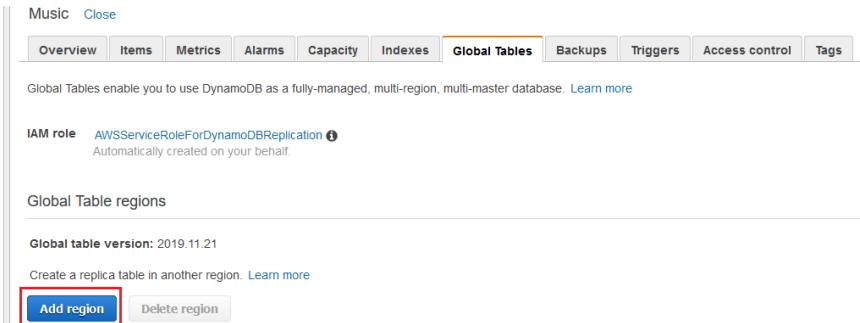
For **Primary key** enter **Artist**. Choose **Add sort key**, and enter **SongTitle**. (**Artist** and **SongTitle** should both be strings.)

To create the table, choose **Create**. This table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.

4. Choose the **Global Tables** tab, and then choose **Enable streams**. Keep the **View type** at its default value (New and old images).



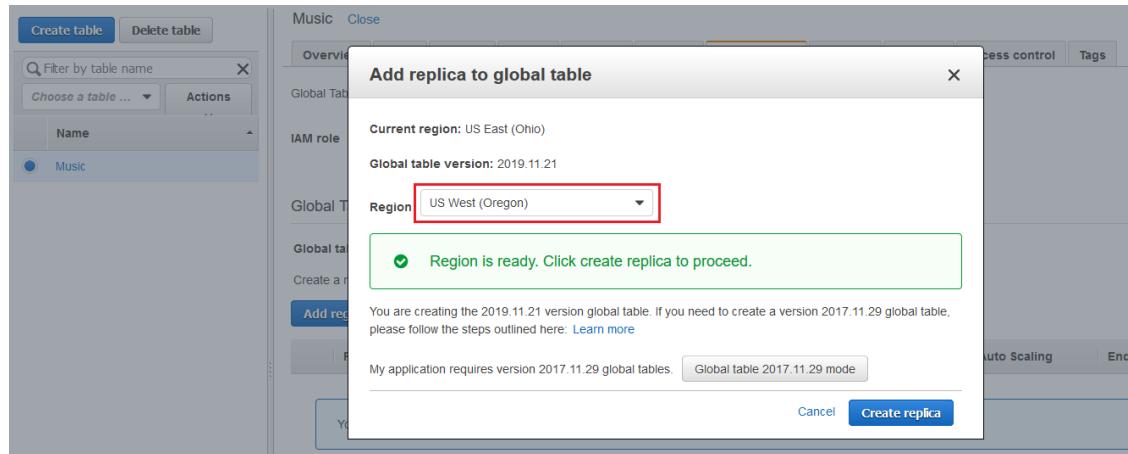
5. Choose **Add region**.



6. In **Region**, choose **US West (Oregon)**.

The console checks to ensure that a table with the same name doesn't exist in the selected Region. If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.

Choose **Create Replica**, this starts the table creation process in US West (Oregon).



The **Global Table** tab for the selected table (and for any other replica tables) shows that the table has been replicated in multiple Regions.

7. Now add another Region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 5, but this time, specify **EU (Frankfurt)** instead of **US West (Oregon)**.
8. You should still be using the AWS Management Console in the us-east-2 (US East Ohio) Region. For the **Music** table, choose the **Items** tab, and then choose **Create Item**. For **Artist**, enter **item_1**. For **SongTitle**, enter **Song Value 1**. To write the item, choose **Save**.

After a short time, the item is replicated across all three Regions of your global table. To verify this, in the console, on the Region selector in the upper-right corner, choose **Europe (Frankfurt)**. The **Music** table in Europe (Frankfurt) should contain the new item.

Repeat this for US West (Oregon).

Creating a Global Table (AWS CLI)

Follow these steps to create a global table **Music** using the AWS CLI. The following example creates a global table, with replica tables in the United States and in Europe.

1. Create a new table (**Music**) in US East (Ohio), with DynamoDB Streams enabled (**NEW_AND_OLD_IMAGES**).

```
aws dynamodb create-table \
--table-name Music \
--attribute-definitions \
  AttributeName=Artist,AttributeType=S \
  AttributeName=SongTitle,AttributeType=S \
--key-schema \
  AttributeName=Artist,KeyType=HASH \
  AttributeName=SongTitle,KeyType=RANGE \
--billing-mode PAY_PER_REQUEST \
--stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
--region us-east-2
```

2. Create an identical **Music** table in US East (N. Virginia).

```
aws dynamodb update-table --table-name Music --cli-input-json \
'{
  "ReplicaUpdates": [
    {
      "Region": "us-west-2",
      "Status": "ENABLED"
    }
  ]
}'
```

```
{
    "Create": [
        {
            "RegionName": "us-east-1"
        }
    ]
}'
```

3. Repeat step 2 to create a table in Europe (Ireland) (eu-west-1).
4. You can view the list of replicas created using `describe-table`.

```
aws dynamodb describe-table --table-name Music --region us-east-2
```

5. To verify that replication is working, add a new item to the `Music` table in US East (Ohio).

```
aws dynamodb put-item \
--table-name Music \
--item '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-2
```

6. Wait for a few seconds, and then check to see whether the item has been successfully replicated to US East (N. Virginia) and Europe (Ireland).

```
aws dynamodb get-item \
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region us-east-1
```

```
aws dynamodb get-item \
--table-name Music \
--key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
--region eu-west-1
```

7. Delete the replica table in Europe (Ireland) Region.

```
aws dynamodb update-table --table-name Music --cli-input-json \
'{
    "ReplicaUpdates": [
        {
            "Delete": {
                "RegionName": "eu-west-1"
            }
        }
    ]
}'
```

Creating a Global Table (Java)

The following java code sample, create a `Music` table in Europe (Ireland) region then creates a replica in Asia Pacific (Seoul) region.

```
package com.amazonaws.codesamples.gtv2
import java.util.logging.Logger;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
```

```

import com.amazonaws.services.dynamodbv2.model.AmazonDynamoDBException;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.BillingMode;
import com.amazonaws.services.dynamodbv2.model.CreateReplicationGroupMemberAction;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.DescribeTableRequest;
import com.amazonaws.services.dynamodbv2.model.GlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughputOverride;
import com.amazonaws.services.dynamodbv2.model.ReplicaGlobalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.ReplicationGroupUpdate;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.services.dynamodbv2.model.StreamSpecification;
import com.amazonaws.services.dynamodbv2.model.StreamViewType;
import com.amazonaws.services.dynamodbv2.model.UpdateTableRequest;
import com.amazonaws.waiters.WaiterParameters;

```

```

public class App
{
    private final static Logger LOGGER = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);

    public static void main( String[] args )
    {

        String tableName = "Music";
        String indexName = "index1";

        Regions calledRegion = Regions.EU_WEST_1;
        Regions destRegion = Regions.AP_NORTHEAST_2;

        AmazonDynamoDB ddbClient = AmazonDynamoDBClientBuilder.standard()
            .withCredentials(new ProfileCredentialsProvider("default"))
            .withRegion(calledRegion)
            .build();

        LOGGER.info("Creating a regional table - TableName: " + tableName + ", IndexName: "
+ indexName + " ....");
        ddbClient.createTable(new CreateTableRequest()
            .withTableName(tableName)
            .withAttributeDefinitions(
                new AttributeDefinition()

                .withAttributeName("Artist").withAttributeType(ScalarAttributeType.S),
                new AttributeDefinition()

                .withAttributeName("SongTitle").withAttributeType(ScalarAttributeType.S))
            .withKeySchema(
                new
            KeySchemaElement().withAttributeName("Artist").withKeyType(KeyType.HASH),
                new
            KeySchemaElement().withAttributeName("SongTitle").withKeyType(KeyType.RANGE))
            .withBillingMode(BillingMode.PAY_PER_REQUEST)
            .withGlobalSecondaryIndexes(new GlobalSecondaryIndex()
                .withIndexName(indexName)
                .withKeySchema(new KeySchemaElement()
                    .withAttributeName("SongTitle")
                    .withKeyType(KeyType.HASH)))
            .withProjection(new
        Projection().withProjectionType(ProjectionType.ALL)))
            .withStreamSpecification(new StreamSpecification())
    }
}

```

```
        .withStreamEnabled(true)
        .withStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES));

    LOGGER.info("Waiting for ACTIVE table status ....");
    ddbClient.waiters().tableExists().run(new WaiterParameters<>(new
DescribeTableRequest(tableName)));

    LOGGER.info("Testing parameters for adding a new Replica in " + destRegion +
" ....");

    CreateReplicationGroupMemberAction createReplicaAction = new
CreateReplicationGroupMemberAction()
    .withRegionName(destRegion.getName())
    .withGlobalSecondaryIndexes(new ReplicaGlobalSecondaryIndex()
        .withIndexName(indexName)
        .withProvisionedThroughputOverride(new
ProvisionedThroughputOverride()
            .withReadCapacityUnits(15L)));
}

ddbClient.updateTable(new UpdateTableRequest()
    .withTableName(tableName)
    .withReplicaUpdates(new ReplicationGroupUpdate()
        .withCreate(createReplicaAction.withKMSMasterKeyId(null))));

}

}
```

Monitoring Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

You can use Amazon CloudWatch to monitor the behavior and performance of a global table. Amazon DynamoDB publishes `ReplicationLatency` metric for each replica in the global table.

- **ReplicationLatency**—The elapsed time between when an item is written to a replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source- and destination-Region pair.

During normal operation, `ReplicationLatency` should be fairly constant. An elevated value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. Over time, this could result in other replica tables *falling behind* because they no longer receive updates consistently. In this case, you should verify that the read capacity units (RCUs) and write capacity units (WCUs) are identical for each of the replica tables. In addition, when choosing WCU settings, follow the recommendations in [Best Practices and Requirements for Managing Capacity \(p. 627\)](#).

`ReplicationLatency` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

For more information, see [DynamoDB Metrics and Dimensions \(p. 857\)](#).

Using IAM with Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

When you create a global table for the first time, Amazon DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role is named `AWSServiceRoleForDynamoDBReplication`, and it allows DynamoDB to manage cross-Region replication for global tables on your behalf. Don't delete this service-linked role. If you do, all of your global tables will no longer function.

For more information about service-linked roles, see [Using Service-Linked Roles](#) in the *IAM User Guide*.

To create replica tables in DynamoDB, you must have the following permissions in the source region.

- `dynamodb:UpdateTable`

To create replica tables in DynamoDB, you must have the following permissions in destination regions.

- `dynamodb CreateTable`
- `dynamodb CreateTableReplica`
- `dynamodb Scan`
- `dynamodb Query`
- `dynamodb UpdateItem`
- `dynamodb PutItem`
- `dynamodb GetItem`
- `dynamodb DeleteItem`
- `dynamodb BatchWriteItem`

To delete replica tables in DynamoDB, you must have the following permissions in the destination regions.

- `dynamodb DeleteTable`
- `dynamodb DeleteTableReplica`

To update replica auto-scaling policy through `UpdateTableReplicaAutoScaling`, you must have the following permissions in all Regions where table replicas exist

- `application-autoscaling>DeleteScalingPolicy`
- `application-autoscaling>DeleteScheduledAction`
- `application-autoscaling>DeregisterScalableTarget`
- `application-autoscaling>DescribeScalableTargets`
- `application-autoscaling>DescribeScalingActivities`
- `application-autoscaling>DescribeScalingPolicies`

- application-autoscaling:DescribeScheduledActions
- application-autoscaling:PutScalingPolicy
- application-autoscaling:PutScheduledAction
- application-autoscaling:RegisterScalableTarget

To use `UpdateTimeToLive` you must have permission for `dynamodb:UpdateTimeToLive` in all Regions where table replicas exist.

Example: Add Replica

The following IAM policy grants permissions to allow you to add replicas to a global table.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:CreateTable",  
                "dynamodb:DescribeTable",  
                "dynamodb:UpdateTable",  
                "dynamodb:CreateTableReplica",  
                "iam:CreateServiceLinkedRole"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Update AutoScaling policy

The following IAM policy grants permissions to allow you to update replica auto-scaling policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "application-autoscaling:RegisterScalableTarget",  
                "application-autoscaling:DeleteScheduledAction",  
                "application-autoscaling:DescribeScalableTargets",  
                "application-autoscaling:DescribeScalingActivities",  
                "application-autoscaling:DescribeScalingPolicies",  
                "application-autoscaling:PutScalingPolicy",  
                "application-autoscaling:DescribeScheduledActions",  
                "application-autoscaling:DeleteScalingPolicy",  
                "application-autoscaling:PutScheduledAction",  
                "application-autoscaling:DeregisterScalableTarget"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Allow Replica Creations for a Specific Table Name and Regions

The following IAM policy grants permissions to allow table and replica creation for *Customers* table with replicas in three Regions.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:CreateTable",  
                "dynamodb:DescribeTable",  
                "dynamodb:UpdateTable"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",  
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers",  
                "arn:aws:dynamodb:eu-east-2:123456789012:table/Customers"  
            ]  
        }  
    ]  
}
```

Updating Global Tables to Version 2019.11.21 (Current)

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

This section describes how to update your global tables to *version 2019.11.21 (current)*.

Topics

- [Before You Begin \(p. 635\)](#)
- [Updating to Version 2019.11.21 \(Current\) \(p. 637\)](#)

Before You Begin

DynamoDB global tables version 2019.11.21 (current) introduces the following requirements:

- The [global secondary indexes](#) on the replica tables must be consistent across Regions.
- The [encryption](#) setting on the replica tables must be consistent across Regions.
- The [Time To Live \(TTL\)](#) settings on the replica tables must be consistent across Regions. If TTL is enabled on a replica table, TTL deletes are replicated.
- DynamoDB auto scaling or [on-demand](#) capacity must be enabled for write capacity units across all replica tables.

- Global tables control plane operations (create, delete, update, and describe) APIs differ. For more information, see [Global Tables: Multi-Region Replication with DynamoDB](#).

DynamoDB global tables version 2019.11.21 (current) introduces the following changes in behavior:

- DynamoDB Streams publishes only one record (instead of two) for each write.
- For new inserts, `aws:rep:*` attributes in the item record are not added.
- For updates to items that contain `aws:rep:*` attributes, those attributes are not updated.
- DynamoDB mapper must not require the `aws:rep:*` attributes for this table.
- When updating from version 2017.11.21 to version 2019.11.29, you might see an increase in the `ReplicationLatency` metric. This is expected because version 2019.11.29 (current) captures the full end-to-end measurement of replication delay between global tables Regions. For more information, see the `ReplicationLatency` documentation for version [Version 2017.11.21](#) and [Version 2019.11.29 \(Current\)](#).

Required Permissions

To update to version 2019.11.21 (current), you must have `dynamodb:UpdateGlobalTableVersion` permissions across replica Regions. These permissions are in addition to the permissions needed for accessing the DynamoDB console and viewing tables.

The following IAM policy grants permissions to update any global table to version 2019.11.21 (current).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:UpdateGlobalTableVersion",  
            "Resource": "*"  
        }  
    ]  
}
```

The following IAM policy grants permissions to update only the `Music` global table, with replicas in two Regions, to version 2019.11.21 (current).

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "dynamodb:UpdateGlobalTableVersion",  
            "Resource": [  
                "arn:aws:dynamodb::123456789012:global-table/Music",  
                "arn:aws:dynamodb:ap-southeast-1:123456789012:table/Music",  
                "arn:aws:dynamodb:us-east-2:123456789012:table/Music"  
            ]  
        }  
    ]  
}
```

Update Process Overview

During the update process, the state of your global table changes from **ACTIVE** to **UPDATING**.

The process takes several minutes, but it should finish in less than an hour. During the update process, your table remains available for read and write traffic. However, auto scaling does not alter the provisioned capacity on your table during the update. As such, before you begin the update, we recommend that you switch your table to [on-demand](#) capacity.

If you choose to use [provisioned](#) capacity with auto scaling during the update, you must increase the minimum read and write throughput on your policies to accommodate expected increases in traffic for the duration of the update.

When the update process is complete, your table status returns to **ACTIVE**. You can check the state of your table by using [DescribeTable](#), or verify it using the **Tables** view on the DynamoDB console.

Important

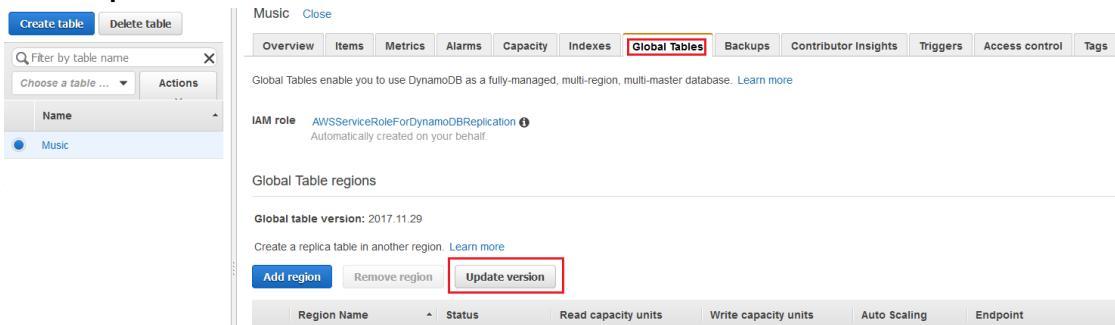
- Updating from version 2017.11.29 to version 2019.11.21 (current) is a one-time action and cannot be reversed. Before proceeding with the update, ensure that you have performed all necessary testing. Allow up to 60 minutes before attempting to update a newly created global table.
- In version 2017.11.29 of global tables, DynamoDB performed a write operation to insert three attributes into the item record. These attributes (`aws:rep:*`) were used to enact replication and manage conflict resolution. In version 2019.11.21 (current) of global tables, replication activity is managed natively and is not exposed to users.
- Updating to version 2019.11.21 (current) is only available through the DynamoDB console.

Updating to Version 2019.11.21 (Current)

Follow these steps to update your version of DynamoDB global tables using the AWS Management Console.

To update global tables to version 2019.11.21

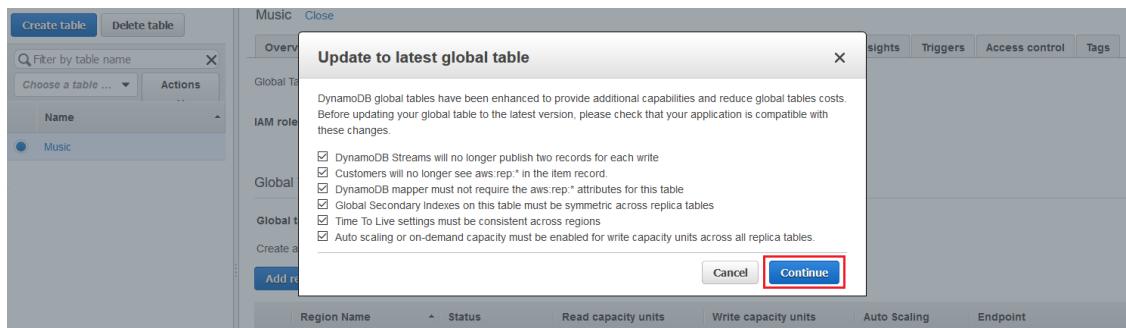
1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>.
2. In the navigation pane on the left side of the console, choose **Tables**, and then select the global table that you want to update to 2019.11.21 (current).
3. Choose the **Global Tables** tab.
4. Choose **Update version**.



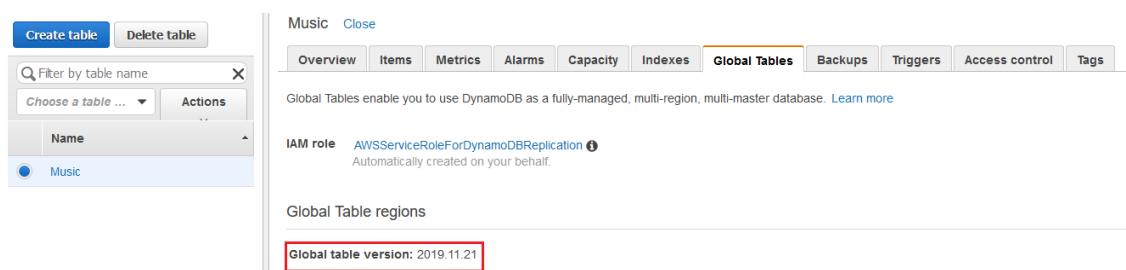
5. Read and agree to the new requirements, and then choose **Continue**.

Important

Updating from version 2017.11.29 to version 2019.11.21 (current) is a one-time action and cannot be reversed. Before starting the update, ensure that you have performed all necessary testing.



6. After the update process is complete, the global tables version that appears on the console changes to **2019.11.21**.



Version 2017.11.29

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\)](#) (p. 625) and [Version 2017.11.29](#) (p. 638). To find out which version you are using, see [Determine Version](#) (p. 625).

Topics

- [Global Tables: How It Works](#) (p. 638)
- [Best Practices and Requirements for Managing Global Tables](#) (p. 640)
- [Creating a Global Table](#) (p. 642)
- [Monitoring Global Tables](#) (p. 644)
- [Using IAM with Global Tables](#) (p. 645)

Global Tables: How It Works

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\)](#) (p. 625) and [Version 2017.11.29](#) (p. 638). To find out which version you are using, see [Determine Version](#) (p. 625).

The following sections help you understand the concepts and behavior of global tables in Amazon DynamoDB.

Global Table Concepts

A *global table* is a collection of one or more replica tables, all owned by a single AWS account.

A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per AWS Region.

The following is a conceptual overview of how a global table is created.

1. Create an ordinary DynamoDB table, with DynamoDB Streams enabled, in an AWS Region.
2. Repeat step 1 for every other Region where you want to replicate your data.
3. Define a DynamoDB global table based on the tables that you have created.

The AWS Management Console automates these tasks, so you can create a global table more quickly and easily. For more information, see [Creating a Global Table \(p. 642\)](#).

The resulting DynamoDB global table consists of multiple replica tables, one per Region, that DynamoDB treats as a single unit. Every replica has the same table name and the same primary key schema. When an application writes data to a replica table in one Region, DynamoDB automatically propagates the write to the other replica tables in the other AWS Regions.

Important

To keep your table data in sync, global tables automatically create the following attributes for every item:

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

Do not modify these attributes or create attributes with the same name.

You can add replica tables to the global table so that it can be available in additional Regions. (To do this, the global table must be empty. In other words, none of the replica tables can have any data in them.)

You can also remove a replica table from a global table. If you do this, the table is completely disassociated from the global table. This newly independent table no longer interacts with the global table, and data is no longer propagated to or from the global table.

Consistency and Conflict Resolution

Any changes made to any item in any replica table are replicated to all the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within seconds.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. If your application only uses eventually consistent reads and only issues reads against one AWS Region, it will work without any modification. However, if your application requires strongly consistent reads, it must perform all of its strongly consistent reads and writes in the same Region. DynamoDB does not support strongly consistent reads across Regions. Therefore, if you write to one Region and read from another Region, the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region.

Conflicts can arise if applications update the same item in different Regions at about the same time. To help ensure eventual consistency, DynamoDB global tables use a *last writer wins* reconciliation between concurrent updates, in which DynamoDB makes a best effort to determine the last writer. With this conflict resolution mechanism, all the replicas will agree on the latest update and converge toward a state in which they all have identical data.

Availability and Durability

If a single AWS Region becomes isolated or degraded, your application can redirect to a different Region and perform reads and writes against a different replica table. You can apply custom business logic to determine when to redirect requests to other Regions.

If a Region becomes isolated or degraded, DynamoDB keeps track of any writes that have been performed but have not yet been propagated to all of the replica tables. When the Region comes back online, DynamoDB resumes propagating any pending writes from that Region to the replica tables in other Regions. It also resumes propagating writes from other replica tables to the Region that is now back online.

Best Practices and Requirements for Managing Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\)](#) (p. 625) and [Version 2017.11.29](#) (p. 638). To find out which version you are using, see [Determine Version](#) (p. 625).

Using Amazon DynamoDB global tables, you can replicate your table data across AWS Regions. It is important that the replica tables and secondary indexes in your global table have identical write capacity settings to ensure proper replication of data.

Topics

- [Requirements for Adding a New Replica Table](#) (p. 640)
- [Best Practices and Requirements for Managing Capacity](#) (p. 641)

Requirements for Adding a New Replica Table

If you want to add a new replica table to a global table, each of the following conditions must be true:

- The table must have the same partition key as all of the other replicas.
- The table must have the same write capacity management settings specified.
- The table must have the same name as all of the other replicas.
- The table must have DynamoDB Streams enabled, with the stream containing both the new and the old images of the item.
- None of the new or existing replica tables in the global table can contain any data.

If global secondary indexes are specified, the following conditions must also be met:

- The global secondary indexes must have the same name.
- The global secondary indexes must have the same partition key and sort key (if present).

Important

Write capacity settings should be set consistently across all of your global tables' replica tables and matching secondary indexes. To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the

`UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see [UpdateGlobalTableSettings](#) in the [Amazon DynamoDB API Reference](#).

We strongly recommend that you enable auto scaling to manage provisioned write capacity settings. If you prefer to manage write capacity settings manually, you should provision equal replicated write capacity units to all of your replica tables. Also provision equal replicated write capacity units to matching secondary indexes across your global table.

You must also have appropriate AWS Identity and Access Management (IAM) permissions. For more information, see [Using IAM with Global Tables \(p. 645\)](#).

Best Practices and Requirements for Managing Capacity

Consider the following when managing capacity settings for replica tables in DynamoDB.

Using DynamoDB Auto Scaling

Using DynamoDB auto scaling is the recommended way to manage throughput capacity settings for replica tables that use the provisioned mode. DynamoDB auto scaling automatically adjusts read capacity units (RCUs) and write capacity units (WCUs) for each replica table based upon your actual application workload. For more information, see [Managing Throughput Capacity Automatically with DynamoDB Auto Scaling \(p. 345\)](#).

If you create your replica tables using the AWS Management Console, auto scaling is enabled by default for each replica table, with default auto scaling settings for managing read capacity units and write capacity units.

Changes to auto scaling settings for a replica table or secondary index made through the DynamoDB console or using the `UpdateGlobalTableSettings` call are applied to all of the replica tables and matching secondary indexes in the global table automatically. These changes overwrite any existing auto scaling settings. This ensures that provisioned write capacity settings are consistent across the replica tables and secondary indexes in your global table. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` calls, you should apply the change to each replica table and matching secondary index individually.

Note

If auto scaling doesn't satisfy your application's capacity changes (unpredictable workload), or if you don't want to configure its settings (target settings for minimum, maximum, or utilization threshold), you can use on-demand mode to manage capacity for your global tables. For more information, see [On-Demand Mode \(p. 17\)](#).

If you enable on-demand mode on a global table, your consumption of replicated write request units (rWCUs) will be consistent with how rWCUs are provisioned. For example, if you perform 10 writes to a local table that is replicated in two additional Regions, you will consume 60 write request units ($10 + 10 + 10 = 30; 30 \times 2 = 60$). The consumed 60 write request units include the extra write consumed by global tables Version 2017.11.29 to update the `aws:rep:deleting`, `aws:rep:updatetime`, and `aws:rep:updateregion` attributes.

Managing Capacity Manually

If you decide not to use DynamoDB auto scaling, you must manually set the read capacity and write capacity settings on each replica table and secondary index.

The provisioned replicated write capacity units (rWCUs) on every replica table should be set to the total number of rWCUs needed for application writes across all Regions multiplied by two. This accommodates application writes that occur in the local Region and replicated application writes coming from other Regions. For example, suppose that you expect 5 writes per second to your replica table in Ohio and 5 writes per second to your replica table in N. Virginia. In this case, you should provision 20 WCUs to each replica table ($5 + 5 = 10; 10 \times 2 = 20$).

To update write capacity settings for your global table, we strongly recommend using the DynamoDB console or the `UpdateGlobalTableSettings` API operation. `UpdateGlobalTableSettings` applies changes to write capacity settings to all replica tables and matching secondary indexes in a global table automatically. If you use the `UpdateTable`, `RegisterScalableTarget`, or `PutScalingPolicy` operations, you should apply the change to each replica table and matching secondary index individually. For more information, see [Amazon DynamoDB API Reference](#).

Note

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions. For more information, see [Using IAM with Global Tables \(p. 645\)](#).

Creating a Global Table

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

This section describes how to create a global table using the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

Topics

- [Creating a Global Table \(Console\) \(p. 642\)](#)
- [Creating a Global Table \(AWS CLI\) \(p. 643\)](#)

Creating a Global Table (Console)

Follow these steps to create a global table using the console. The following example creates a global table with replica tables in United States and Europe.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/home>. For this example, choose the us-east-2 (US East Ohio) Region.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**.

For **Table name**, enter **Music**.

For **Primary key** enter **Artist**. Choose **Add sort key**, and enter **SongTitle**. (**Artist** and **SongTitle** should both be strings.)

To create the table, choose **Create**. This table serves as the first replica table in a new global table. It is the prototype for other replica tables that you add later.

4. Choose the **Global Tables** tab, and then choose **Enable streams**. Keep the **View type** at its default value (New and old images).
5. Choose **Add region**, and choose **Global table 2017.11.29 mode**. You can now choose another Region where you want to deploy another replica table. In this case, choose **US West (Oregon)**, and then choose **Continue**. This starts the table creation process in US West (Oregon).

The console checks to ensure that there is no table with the same name in the selected Region. (If a table with the same name does exist, you must delete the existing table before you can create a new replica table in that Region.)

The **Global Table** tab for the selected table (and for any other replica tables) will show that the table is replicated in multiple Regions.

6. Now add another Region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 5, but this time, specify **EU (Frankfurt)** instead of **US West (Oregon)**.
7. You should still be using the AWS Management Console in the us-east-2 (US East Ohio) Region. For the **Music** table, choose the **Items** tab, and then choose **Create Item**. For **Artist**, enter **item_1**. For **SongTitle**, enter **Song Value 1**. To write the item, choose **Save**.

After a short time, the item is replicated across all three Regions of your global table. To verify this, in the console, on the Region selector in the upper-right corner, choose **Europe (Frankfurt)**. The **Music** table in Europe (Frankfurt) should contain the new item.

Repeat this for US West (Oregon).

Creating a Global Table (AWS CLI)

Follow these steps to create a global table **Music** using the AWS CLI. The following example creates a global table, with replica tables in the United States and in Europe.

1. Create a new table (**Music**) in US East (Ohio), with DynamoDB Streams enabled (**NEW_AND_OLD_IMAGES**).

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-2
```

2. Create an identical **Music** table in US East (N. Virginia).

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region us-east-1
```

3. Create a global table (**Music**) consisting of replica tables in the **us-east-2** and **us-east-1** Regions.

```
aws dynamodb create-global-table \
    --global-table-name Music \
    --replication-group RegionName=us-east-2 RegionName=us-east-1 \
    --region us-east-2
```

Note

The global table name (`Music`) must match the name of each of the replica tables (`Music`). For more information, see [Best Practices and Requirements for Managing Global Tables \(p. 640\)](#).

4. Create another table in Europe (Ireland), with the same settings as those you created in step 1 and step 2.

```
aws dynamodb create-table \
    --table-name Music \
    --attribute-definitions \
        AttributeName=Artist,AttributeType=S \
        AttributeName=SongTitle,AttributeType=S \
    --key-schema \
        AttributeName=Artist,KeyType=HASH \
        AttributeName=SongTitle,KeyType=RANGE \
    --provisioned-throughput \
        ReadCapacityUnits=10,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES \
    --region eu-west-1
```

After doing this step, add the new table to the `Music` global table.

```
aws dynamodb update-global-table \
    --global-table-name Music \
    --replica-updates 'Create={RegionName=eu-west-1}' \
    --region us-east-2
```

5. To verify that replication is working, add a new item to the `Music` table in US East (Ohio).

```
aws dynamodb put-item \
    --table-name Music \
    --item '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-2
```

6. Wait for a few seconds, and then check to see whether the item has been successfully replicated to US East (N. Virginia) and Europe (Ireland).

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
    --region us-east-1
```

```
aws dynamodb get-item \
    --table-name Music \
    --key '{"Artist": {"S":"item_1"}, "SongTitle": {"S":"Song Value 1"}}' \
    --region eu-west-1
```

Monitoring Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

You can use Amazon CloudWatch to monitor the behavior and performance of a global table. Amazon DynamoDB publishes `ReplicationLatency` and `PendingReplicationCount` metrics for each replica in the global table.

- **ReplicationLatency**—The elapsed time between when an updated item appears in the DynamoDB stream for one replica table, and when that item appears in another replica in the global table. `ReplicationLatency` is expressed in milliseconds and is emitted for every source- and destination-Region pair.

During normal operation, `ReplicationLatency` should be fairly constant. An elevated value for `ReplicationLatency` could indicate that updates from one replica are not propagating to other replica tables in a timely manner. Over time, this could result in other replica tables *falling behind* because they no longer receive updates consistently. In this case, you should verify that the read capacity units (RCUs) and write capacity units (WCUs) are identical for each of the replica tables. In addition, when choosing WCU settings, follow the recommendations in [Best Practices and Requirements for Managing Capacity \(p. 641\)](#).

`ReplicationLatency` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

- **PendingReplicationCount**—The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table. `PendingReplicationCount` is expressed in number of items and is emitted for every source- and destination-Region pair.

During normal operation, `PendingReplicationCount` should be very low. If `PendingReplicationCount` increases for extended periods, investigate whether your replica tables' provisioned write capacity settings are sufficient for your current workload.

`PendingReplicationCount` can increase if an AWS Region becomes degraded and you have a replica table in that Region. In this case, you can temporarily redirect your application's read and write activity to a different AWS Region.

For more information, see [DynamoDB Metrics and Dimensions \(p. 857\)](#).

Using IAM with Global Tables

There are two versions of DynamoDB global tables available: [Version 2019.11.21 \(Current\) \(p. 625\)](#) and [Version 2017.11.29 \(p. 638\)](#). To find out which version you are using, see [Determine Version \(p. 625\)](#).

When you create a global table for the first time, Amazon DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role is named `AWSServiceRoleForDynamoDBReplication`, and it allows DynamoDB to manage cross-Region replication for global tables on your behalf. Don't delete this service-linked role. If you do, then all of your global tables will no longer function.

For more information about service-linked roles, see [Using Service-Linked Roles](#) in the *IAM User Guide*.

To create and maintain global tables in DynamoDB, you must have the `dynamodb:CreateGlobalTable` permission to access each of the following:

- The replica table that you want to add.
- Each existing replica that's already part of the global table.
- The global table itself.

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The `application-autoscaling:DeleteScalingPolicy` and `application-autoscaling:DeregisterScalableTarget` permissions are required when updating an existing scaling policy. This is so that the global tables service can remove the old scaling policy before attaching the new policy to the table or secondary index.

If you use an IAM policy to manage access to one replica table, you should apply an identical policy to all other replicas within that global table. This practice helps you maintain a consistent permissions model across all of the replica tables.

By using identical IAM policies on all replicas in a global table, you can also avoid granting unintended read and write access to your global table data. For example, consider a user who has access to only one replica in a global table. If that user can write to this replica, DynamoDB propagates the write to all of the other replica tables. In effect, the user can (indirectly) write to all of the other replicas in the global table. This scenario can be avoided by using consistent IAM policies on all of the replica tables.

Example: Allow the `CreateGlobalTable` Action

Before you can add a replica to a global table, you must have the `dynamodb:CreateGlobalTable` permission for the global table and for each of its replica tables.

The following IAM policy grants permissions to allow the `CreateGlobalTable` action on all tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": ["dynamodb:CreateGlobalTable"],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Allow the `UpdateGlobalTable`, `DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` Actions

To update the settings (`UpdateGlobalTableSettings`) for a global table in DynamoDB, you must have the `dynamodb:UpdateGlobalTable`, `dynamodb:DescribeLimits`, `application-autoscaling:DeleteScalingPolicy`, and `application-autoscaling:DeregisterScalableTarget` permissions.

The following IAM policy grants permissions to allow the `UpdateGlobalTableSettings` action on all tables.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:UpdateGlobalTable",  
                "dynamodb:DescribeLimits",  
                "application-autoscaling:DeleteScalingPolicy",  
                "application-autoscaling:DeregisterScalableTarget"  
            ]  
        }  
    ]  
}
```

```
        "dynamodb:UpdateGlobalTable",
        "dynamodb:DescribeLimits",
        "application-autoscaling:DeleteScalingPolicy",
        "application-autoscaling:DeregisterScalableTarget"
    ],
    "Resource": "*"
}
]
```

Example: Allow the CreateGlobalTable Action for a Specific Global Table Name with Replicas Allowed in Certain Regions Only

The following IAM policy grants permissions to allow the `CreateGlobalTable` action to create a global table named `Customers` with replicas in two Regions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "dynamodb>CreateGlobalTable",
            "Resource": [
                "arn:aws:dynamodb::123456789012:global-table/Customers",
                "arn:aws:dynamodb:us-east-1:123456789012:table/Customers",
                "arn:aws:dynamodb:us-west-1:123456789012:table/Customers"
            ]
        }
    ]
}
```

Managing Complex Workflows with DynamoDB Transactions

Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, helping you to maintain data correctness in your applications.

You can use the DynamoDB transactional read and write APIs to manage complex business workflows that require adding, updating, or deleting multiple items as a single, all-or-nothing operation. For example, a video game developer can ensure that players' profiles are updated correctly when they exchange items in a game or make in-game purchases.

With the transaction write API, you can group multiple `Put`, `Update`, `Delete`, and `ConditionCheck` actions. You can then submit the actions as a single `TransactWriteItems` operation that either succeeds or fails as a unit. The same is true for multiple `Get` actions, which you can group and submit as a single `TransactGetItems` operation.

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. These two underlying read/write operations are visible in your Amazon CloudWatch metrics.

To get started with DynamoDB transactions, download the latest AWS SDK or the AWS Command Line Interface (AWS CLI). Then follow the [DynamoDB Transactions Example \(p. 656\)](#).

The following sections provide a detailed overview of the transaction APIs and how you can use them in DynamoDB.

Topics

- [Amazon DynamoDB Transactions: How It Works \(p. 648\)](#)
- [Using IAM with DynamoDB Transactions \(p. 654\)](#)
- [DynamoDB Transactions Example \(p. 656\)](#)

Amazon DynamoDB Transactions: How It Works

With Amazon DynamoDB transactions, you can group multiple actions together and submit them as a single all-or-nothing `TransactWriteItems` or `TransactGetItems` operation. The following sections describe API operations, capacity management, best practices, and other details about using transactional operations in DynamoDB.

Topics

- [TransactWriteItems API \(p. 649\)](#)
- [TransactGetItems API \(p. 650\)](#)
- [Isolation Levels for DynamoDB Transactions \(p. 650\)](#)
- [Transaction Conflict Handling in DynamoDB \(p. 652\)](#)
- [Using Transactional APIs in DynamoDB Accelerator \(DAX\) \(p. 652\)](#)

- [Capacity Management for Transactions \(p. 652\)](#)
- [Best Practices for Transactions \(p. 653\)](#)
- [Using Transactional APIs with Global Tables \(p. 653\)](#)
- [DynamoDB Transactions vs. the AWS Labs Transactions Client Library \(p. 653\)](#)

TransactWriteItems API

`TransactWriteItems` is a synchronous and idempotent write operation that groups up to 25 write actions in a single all-or-nothing operation. These actions can target up to 25 distinct items in one or more DynamoDB tables within the same AWS account and in the same Region. The aggregate size of the items in the transaction cannot exceed 4 MB. The actions are completed atomically so that either all of them succeed or none of them succeeds.

A `TransactWriteItems` operation differs from a `BatchWriteItem` operation in that all the actions it contains must be completed successfully, or no changes are made at all. With a `BatchWriteItem` operation, it is possible that only some of the actions in the batch succeed while the others do not.

You can't target the same item with multiple operations within the same transaction. For example, you can't perform a `ConditionCheck` and also an `Update` action on the same item in the same transaction.

You can add the following types of actions to a transaction:

- `Put` — Initiates a `PutItem` operation to create a new item or replace an old item with a new item, conditionally or without specifying any condition.
- `Update` — Initiates an `UpdateItem` operation to edit an existing item's attributes or add a new item to the table if it does not already exist. Use this action to add, delete, or update attributes on an existing item conditionally or without a condition.
- `Delete` — Initiates a `DeleteItem` operation to delete a single item in a table identified by its primary key.
- `ConditionCheck` — Checks that an item exists or checks the condition of specific attributes of the item.

Once a transaction completes, the changes made within that transaction are propagated to global secondary indexes (GSIs), streams, and backups. Since propagation is not immediate or instantaneous, if a table is restored from backup to a point mid-propagation, it might contain some but not all of the changes made during a recent transaction.

Idempotency

You can optionally include a client token when you make a `TransactWriteItems` call to ensure that the request is *idempotent*. Making your transactions idempotent helps prevent application errors if the same operation is submitted multiple times due to a connection time-out or other connectivity issue.

If the original `TransactWriteItems` call was successful, the subsequent `TransactWriteItems` calls with the same client token return successfully without making any changes. If the `ReturnConsumedCapacity` parameter is set, the initial `TransactWriteItems` call returns the number of write capacity units consumed in making the changes. Subsequent `TransactWriteItems` calls with the same client token return the number of read capacity units consumed in reading the item.

Important Points about Idempotency

- A client token is valid for 10 minutes after the request that uses it finishes. After 10 minutes, any request that uses the same client token is treated as a new request. You should not reuse the same client token for the same request after 10 minutes.

- If you repeat a request with the same client token within the 10-minute idempotency window but change some other request parameter, DynamoDB returns an `IdempotentParameterMismatch` exception.

Error Handling for Writing

Write transactions don't succeed under the following circumstances:

- When a condition in one of the condition expressions is not met.
- When a transaction validation error occurs because more than one action in the same `TransactWriteItems` operation targets the same item.
- When a `TransactWriteItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactWriteItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When an item size becomes too large (larger than 400 KB), or a local secondary index (LSI) becomes too large, or a similar validation error occurs because of changes made by the transaction.
- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactWriteItems` operations are handled, see [Transaction Conflict Handling in DynamoDB \(p. 652\)](#).

TransactGetItems API

`TransactGetItems` is a synchronous read operation that groups up to 25 Get actions together. These actions can target up to 25 distinct items in one or more DynamoDB tables within the same AWS account and Region. The aggregate size of the items in the transaction can't exceed 4 MB.

The Get actions are performed atomically so that either all of them succeed or all of them fail:

- `Get` — Initiates a `GetItem` operation to retrieve a set of attributes for the item with the given primary key. If no matching item is found, `Get` does not return any data.

Error Handling for Reading

Read transactions don't succeed under the following circumstances:

- When a `TransactGetItems` request conflicts with an ongoing `TransactWriteItems` operation on one or more items in the `TransactGetItems` request. In this case, the request fails with a `TransactionCanceledException`.
- When there is insufficient provisioned capacity for the transaction to be completed.
- When there is a user error, such as an invalid data format.

For more information about how conflicts with `TransactGetItems` operations are handled, see [Transaction Conflict Handling in DynamoDB \(p. 652\)](#).

Isolation Levels for DynamoDB Transactions

The isolation levels of transactional operations (`TransactWriteItems` or `TransactGetItems`) and other operations are as follows.

SERIALIZABLE

Serializable isolation ensures that the results of multiple concurrent operations are the same as if no operation begins until the previous one has finished.

There is serializable isolation between the following types of operation:

- Between any transactional operation and any standard write operation (`PutItem`, `UpdateItem`, or `DeleteItem`).
- Between any transactional operation and any standard read operation (`GetItem`).
- Between a `TransactWriteItems` operation and a `TransactGetItems` operation.

Although there is serializable isolation between transactional operations, and each individual standard writes in a `BatchWriteItem` operation, there is no serializable isolation between the transaction and the `BatchWriteItem` operation as a unit.

Similarly, the isolation level between a transactional operation and individual `GetItems` in a `BatchGetItem` operation is serializable. But the isolation level between the transaction and the `BatchGetItem` operation as a unit is *read-committed*.

READ-COMMITTED

Read-committed isolation ensures that read operations always return committed values for an item. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

The isolation level is read-committed between any transactional operation and any read operation that involves multiple standard reads (`BatchGetItem`, `Query`, or `Scan`). If a transactional write updates an item in the middle of a `BatchGetItem`, `Query`, or `Scan` operation, the read operation returns the new committed value.

Operation Summary

To summarize, the following table shows the isolation levels between a transaction operation (`TransactWriteItems` or `TransactGetItems`) and other operations.

| Operation | Isolation Level |
|-------------------------------|--------------------------|
| <code>DeleteItem</code> | <i>Serializable</i> |
| <code>PutItem</code> | <i>Serializable</i> |
| <code>UpdateItem</code> | <i>Serializable</i> |
| <code>GetItem</code> | <i>Serializable</i> |
| <code>BatchGetItem</code> | <i>Read-committed*</i> |
| <code>BatchWriteItem</code> | <i>NOT Serializable*</i> |
| <code>Query</code> | <i>Read-committed</i> |
| <code>Scan</code> | <i>Read-committed</i> |
| Other transactional operation | <i>Serializable</i> |

Levels marked with an asterisk (*) apply to the operation as a unit. However, individual actions within those operations have a *Serializable* isolation level.

Transaction Conflict Handling in DynamoDB

A transactional conflict can occur during concurrent item-level requests on an item within a transaction. Transaction conflicts can occur in the following scenarios:

- A `PutItem`, `UpdateItem`, or `DeleteItem` request for an item conflicts with an ongoing `TransactWriteItems` request that includes the same item.
- An item within a `TransactWriteItems` request is part of another ongoing `TransactWriteItems` request.
- An item within a `TransactGetItems` request is part of an ongoing `TransactWriteItems`, `BatchWriteItem`, `PutItem`, `UpdateItem`, or `DeleteItem` request.

Note

- When a `PutItem`, `UpdateItem`, or `DeleteItem` request is rejected, the request fails with a `TransactionConflictException`.
- If any item-level request within `TransactWriteItems` or `TransactGetItems` is rejected, the request fails with a `TransactionCanceledException`.

If you are using the AWS SDK for Java, the exception contains the list of `CancellationReasons`, ordered according to the list of items in the `TransactItems` request parameter. For other languages, a string representation of the list is included in the exception's error message.

- If an ongoing `TransactWriteItems` or `TransactGetItems` operation conflicts with a concurrent `GetItem` request, both operations can succeed.

The [TransactionConflict CloudWatch metric](#) is incremented for each failed item-level request.

Using Transactional APIs in DynamoDB Accelerator (DAX)

`TransactWriteItems` and `TransactGetItems` are both supported in DynamoDB Accelerator (DAX) with the same isolation levels as in DynamoDB.

`TransactWriteItems` writes through DAX. DAX passes a `TransactWriteItems` call to DynamoDB and returns the response. To populate the cache after the write, DAX calls `TransactGetItems` in the background for each item in the `TransactWriteItems` operation, which consumes additional read capacity units. (For more information, see [Capacity Management for Transactions \(p. 652\)](#).) This functionality enables you to keep your application logic simple and use DAX for both transactional operations and nontransactional ones.

`TransactGetItems` calls are passed through DAX without the items being cached locally. This is the same behavior as for strongly consistent read APIs in DAX.

Capacity Management for Transactions

There is no additional cost to enable transactions for your DynamoDB tables. You pay only for the reads or writes that are part of your transaction. DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction. The two underlying read/write operations are visible in your Amazon CloudWatch metrics.

Plan for the additional reads and writes that are required by transactional APIs when you are provisioning capacity to your tables. For example, suppose that your application executes one transaction per second, and each transaction writes three 500-byte items in your table. Each item requires two write capacity units (WCUs): one to prepare the transaction and one to commit the transaction. Therefore, you would need to provision six WCUs to the table.

If you were using DynamoDB Accelerator (DAX) in the previous example, you would also use two read capacity units (RCUs) for each item in the `TransactWriteItems` call. So you would need to provision six additional RCUs to the table.

Similarly, if your application executes one read transaction per second, and each transaction reads three 500-byte items in your table, you would need to provision six read capacity units (RCUs) to the table. Reading each item require two RCUs: one to prepare the transaction and one to commit the transaction.

Also, default SDK behavior is to retry transactions in case of a `TransactionInProgressException` exception. Plan for the additional read-capacity units (RCUs) that these retries consume. The same is true if you are retrying transactions in your own code using a `ClientRequestToken`.

Best Practices for Transactions

Consider the following recommended practices when using DynamoDB transactions.

- Enable automatic scaling on your tables, or ensure that you have provisioned enough throughput capacity to perform the two read or write operations for every item in your transaction.
- If you are not using an AWS provided SDK, include a `ClientRequestToken` attribute when you make a `TransactWriteItems` call to ensure that the request is idempotent.
- Don't group operations together in a transaction if it's not necessary. For example, if a single transaction with 10 operations can be broken up into multiple transactions without compromising the application correctness, we recommend splitting up the transaction. Simpler transactions improve throughput and are more likely to succeed.
- Multiple transactions updating the same items simultaneously can cause conflicts that cancel the transactions. We recommend following DynamoDB best practices for data modeling to minimize such conflicts.
- If a set of attributes is often updated across multiple items as part of a single transaction, consider grouping the attributes into a single item to reduce the scope of the transaction.
- Avoid using transactions for ingesting data in bulk. For bulk writes, it is better to use `BatchWriteItem`.

Using Transactional APIs with Global Tables

Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the region where the write is made originally. Transactions are not supported across regions in global tables. For example, if you have a global table with replicas in the US East (Ohio) and US West (Oregon) regions and perform a `TransactWriteItems` operation in the US East (N. Virginia) Region, you may observe partially completed transactions in US West (Oregon) Region as changes are replicated. Changes will only be replicated to other regions once they have been committed in the source region.

DynamoDB Transactions vs. the AWS Labs Transactions Client Library

DynamoDB transactions provide a more cost-effective, robust, and performant replacement for the [AWS Labs](#) transactions client library. We suggest that you update your applications to use the native, server-side transaction APIs.

Using IAM with DynamoDB Transactions

You can use AWS Identity and Access Management (IAM) to restrict the actions that transactional operations can perform in Amazon DynamoDB. For more information about using IAM policies in DynamoDB, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 828\)](#).

Permissions for Put, Update, Delete, and Get actions are governed by the permissions used for the underlying PutItem, UpdateItem, DeleteItem, and GetItem operations. For the ConditionCheck action, you can use the dynamodb:ConditionCheck permission in IAM policies.

The following are examples of IAM policies that you can use to configure the DynamoDB transactions.

Example 1: Allow Transactional Operations

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:ConditionCheckItem",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ]  
        }  
    ]  
}
```

Example 2: Allow Only Transactional Operations

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:ConditionCheckItem",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:GetItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ],  
            "Condition": {  
                "ForAnyValue:StringEquals": {  
                    "dynamodb:EnclosingOperation": [  
                        "TransactWriteItems",  
                        "TransactGetItems"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
    ]  
}
```

Example 3: Allow Nontransactional Reads and Writes, and Block Transactional Reads and Writes

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": [  
                "dynamodb:ConditionCheckItem",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb>DeleteItem",  
                "dynamodb:GetItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ],  
            "Condition": {  
                "ForAnyValue:StringEquals": {  
                    "dynamodb:EnclosingOperation": [  
                        "TransactWriteItems",  
                        "TransactGetItems"  
                    ]  
                }  
            }  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:PutItem",  
                "dynamodb>DeleteItem",  
                "dynamodb:GetItem",  
                "dynamodb:UpdateItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:*:*:table/table04"  
            ]  
        }  
    ]  
}
```

Example 4: Prevent Information from Being Returned on a ConditionCheck Failure

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:ConditionCheckItem",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb>DeleteItem",  
                "dynamodb:GetItem"  
            ]  
        }  
    ]  
}
```

```
        ],
        "Resource": "arn:aws:dynamodb:*:*:table/table01",
        "Condition": {
            "StringEqualsIfExists": {
                "dynamodb:ReturnValues": "NONE"
            }
        }
    ]
}
```

DynamoDB Transactions Example

As an example of a situation in which Amazon DynamoDB transactions can be useful, consider this sample Java application for an online marketplace.

The application has three DynamoDB tables in the backend:

- **Customers** — This table stores details about the marketplace customers. Its primary key is a `CustomerId` unique identifier.
- **ProductCatalog** — This table stores details such as price and availability about the products for sale in the marketplace. Its primary key is a `ProductId` unique identifier.
- **Orders** — This table stores details about orders from the marketplace. Its primary key is an `OrderId` unique identifier.

Making an Order

The following code snippets illustrate how to use DynamoDB transactions to coordinate the multiple steps that are required to create and process an order. Using a single all-or-nothing operation ensures that if any part of the transaction fails, no actions in the transaction are executed and no changes are made.

In this example, you set up an order from a customer whose `customerId` is `09e8e9c8-ec48`. You then execute it as a single transaction using the following simple order-processing workflow:

1. Determine that the customer ID is valid.
2. Make sure that the product is `IN_STOCK`, and update the product status to `SOLD`.
3. Make sure that the order does not already exist, and create the order.

Validate the Customer

First, define an action to verify that a customer with `customerId` equal to `09e8e9c8-ec48` exists in the customer table.

```
final String CUSTOMER_TABLE_NAME = "Customers";
final String CUSTOMER_PARTITION_KEY = "CustomerId";
final String customerId = "09e8e9c8-ec48";
final HashMap<String, AttributeValue> customerItemKey = new HashMap<>();
customerItemKey.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));

ConditionCheck checkCustomerValid = new ConditionCheck()
    .withTableName(CUSTOMER_TABLE_NAME)
    .withKey(customerItemKey)
    .withConditionExpression("attribute_exists(" + CUSTOMER_PARTITION_KEY + ")");
```

Update the Product Status

Next, define an action to update the product status to `SOLD` if the condition that the product status is currently set to `IN_STOCK` is true. Setting the `ReturnValuesOnConditionCheckFailure` parameter returns the item if the item's product status attribute was not equal to `IN_STOCK`.

```
final String PRODUCT_TABLE_NAME = "ProductCatalog";
final String PRODUCT_PARTITION_KEY = "ProductId";
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

Map<String, AttributeValue> expressionAttributeValues = new HashMap<>();
expressionAttributeValues.put(":new_status", new AttributeValue("SOLD"));
expressionAttributeValues.put(":expected_status", new AttributeValue("IN_STOCK"));

Update markItemSold = new Update()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey)
    .withUpdateExpression("SET ProductStatus = :new_status")
    .withExpressionAttributeValues(expressionAttributeValues)
    .withConditionExpression("ProductStatus = :expected_status")

    .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD);
```

Create the Order

Lastly, create the order as long as an order with that `OrderId` does not already exist.

```
final String ORDER_PARTITION_KEY = "OrderId";
final String ORDER_TABLE_NAME = "Orders";

HashMap<String, AttributeValue> orderItem = new HashMap<>();
orderItem.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));
orderItem.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));
orderItem.put(CUSTOMER_PARTITION_KEY, new AttributeValue(customerId));
orderItem.put("OrderStatus", new AttributeValue("CONFIRMED"));
orderItem.put("OrderTotal", new AttributeValue("100"));

Put createOrder = new Put()
    .withTableName(ORDER_TABLE_NAME)
    .withItem(orderItem)
    .withReturnValuesOnConditionCheckFailure(ReturnValuesOnConditionCheckFailure.ALL_OLD)
    .withConditionExpression("attribute_not_exists(" + ORDER_PARTITION_KEY + ")");
```

Execute the Transaction

The following example illustrates how to execute the actions defined previously as a single all-or-nothing operation.

```
Collection<TransactWriteItem> actions = Arrays.asList(
    new TransactWriteItem().withConditionCheck(checkCustomerValid),
    new TransactWriteItem().withUpdate(markItemSold),
    new TransactWriteItem().withPut(createOrder));

TransactWriteItemsRequest placeOrderTransaction = new TransactWriteItemsRequest()
    .withTransactItems(actions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Execute the transaction and process the result.
try {
```

```
client.transactWriteItems(placeOrderTransaction);
System.out.println("Transaction Successful");

} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found" +
rnf.getMessage());
} catch (InternalServerException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.out.println("Transaction Canceled " + tce.getMessage());
}
```

Reading the Order Details

The following example shows how to read the completed order transactionally across the `Orders` and `ProductCatalog` tables.

```
HashMap<String, AttributeValue> productItemKey = new HashMap<>();
productItemKey.put(PRODUCT_PARTITION_KEY, new AttributeValue(productKey));

HashMap<String, AttributeValue> orderKey = new HashMap<>();
orderKey.put(ORDER_PARTITION_KEY, new AttributeValue(orderId));

Get readProductSold = new Get()
    .withTableName(PRODUCT_TABLE_NAME)
    .withKey(productItemKey);
Get readCreatedOrder = new Get()
    .withTableName(ORDER_TABLE_NAME)
    .withKey(orderKey);

Collection<TransactGetItem> getActions = Arrays.asList(
    new TransactGetItem().withGet(readProductSold),
    new TransactGetItem().withGet(readCreatedOrder));

TransactGetItemsRequest readCompletedOrder = new TransactGetItemsRequest()
    .withTransactItems(getActions)
    .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

// Execute the transaction and process the result.
try {
    TransactGetItemsResult result = client.transactGetItems(readCompletedOrder);
    System.out.println(result.getResponses());
} catch (ResourceNotFoundException rnf) {
    System.err.println("One of the table involved in the transaction is not found" +
rnf.getMessage());
} catch (InternalServerException ise) {
    System.err.println("Internal Server Error" + ise.getMessage());
} catch (TransactionCanceledException tce) {
    System.err.println("Transaction Canceled" + tce.getMessage());
}
```

Additional Examples

- [Using transactions from DynamoDBMapper](#)

In-Memory Acceleration with DynamoDB Accelerator (DAX)

Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, DynamoDB Accelerator (DAX) delivers fast response times for accessing eventually consistent data.

DAX is a DynamoDB-compatible caching service that enables you to benefit from fast in-memory performance for demanding applications. DAX addresses three core scenarios:

1. As an in-memory cache, DAX reduces the response times of eventually consistent read workloads by an order of magnitude from single-digit milliseconds to microseconds.
2. DAX reduces operational and application complexity by providing a managed service that is API-compatible with DynamoDB. Therefore, it requires only minimal functional changes to use with an existing application.
3. For read-heavy or bursty workloads, DAX provides increased throughput and potential operational cost savings by reducing the need to overprovision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

DAX supports server-side encryption. With encryption at rest, the data persisted by DAX on disk will be encrypted. DAX writes data to disk as part of propagating changes from the primary node to read replicas. For more information, see [DAX Encryption at Rest \(p. 755\)](#).

Topics

- [Use Cases for DAX \(p. 659\)](#)
- [DAX Usage Notes \(p. 660\)](#)
- [DAX: How It Works \(p. 661\)](#)
- [DAX Cluster Components \(p. 664\)](#)
- [Creating a DAX Cluster \(p. 668\)](#)
- [DAX and DynamoDB Consistency Models \(p. 676\)](#)
- [Developing with the DynamoDB Accelerator \(DAX\) Client \(p. 682\)](#)
- [Managing DAX Clusters \(p. 726\)](#)
- [Monitoring DAX \(p. 732\)](#)
- [DAX Access Control \(p. 745\)](#)
- [DAX Encryption at Rest \(p. 755\)](#)
- [Using Service-Linked IAM Roles for DAX \(p. 756\)](#)
- [Accessing DAX Across AWS Accounts \(p. 759\)](#)
- [DAX Cluster Sizing Guide \(p. 766\)](#)
- [DAX API Reference \(p. 768\)](#)

Use Cases for DAX

DAX provides access to eventually consistent data from DynamoDB tables, with microsecond latency. A Multi-AZ DAX cluster can serve millions of requests per second.

DAX is ideal for the following types of applications:

- Applications that require the fastest possible response time for reads. Some examples include real-time bidding, social gaming, and trading applications. DAX delivers fast, in-memory read performance for these use cases.
- Applications that read a small number of items more frequently than others. For example, consider an ecommerce system that has a one-day sale on a popular product. During the sale, demand for that product (and its data in DynamoDB) would sharply increase, compared to all of the other products. To mitigate the impacts of a "hot" key and a non-uniform traffic distribution, you could offload the read activity to a DAX cache until the one-day sale is over.
- Applications that are read-intensive, but are also cost-sensitive. With DynamoDB, you provision the number of reads per second that your application requires. If read activity increases, you can increase your tables' provisioned read throughput (at an additional cost). Or, you can offload the activity from your application to a DAX cluster, and reduce the number of read capacity units that you need to purchase otherwise.
- Applications that require repeated reads against a large set of data. Such an application could potentially divert database resources from other applications. For example, a long-running analysis of regional weather data could temporarily consume all the read capacity in a DynamoDB table. This situation would negatively impact other applications that need to access the same data. With DAX, the weather analysis could be performed against cached data instead.

DAX is *not* ideal for the following types of applications:

- Applications that require strongly consistent reads (or that cannot tolerate eventually consistent reads).
- Applications that do not require microsecond response times for reads, or that do not need to offload repeated read activity from underlying tables.
- Applications that are write-intensive, or that do not perform much read activity.
- Applications that are already using a different caching solution with DynamoDB, and are using their own client-side logic for working with that caching solution.

DAX Usage Notes

- For a list of AWS Regions where DAX is available, see [Amazon DynamoDB pricing](#).
- DAX supports applications written in Go, Java, Node.js, Python, and .NET, using AWS-provided clients for those programming languages.
- DAX does not support Transport Layer Security (TLS).
- DAX is only available for the EC2-VPC platform. (There is no support for the EC2-Classic platform.)
- The DAX cluster service role policy must allow the dynamodb:DescribeTable action in order to maintain metadata about the DynamoDB table.
- DAX clusters maintain metadata about the attribute names of items they store. That metadata is maintained indefinitely (even after the item has expired or been evicted from the cache). Applications that use an unbounded number of attribute names can, over time, cause memory exhaustion in the DAX cluster. This limitation applies only to top-level attribute names, not nested attribute names. Examples of problematic top-level attribute names include timestamps, UUIDs, and session IDs.

This limitation applies only to attribute names, not their values. Items like the following are not a problem.

```
{  
    "Id": 123,  
    "Title": "Bicycle 123",  
    "CreationDate": "2017-10-24T01:02:03+00:00"  
}
```

But items like the following are a problem if there are enough of them and they each have a different timestamp.

```
{  
    "Id": 123,  
    "Title": "Bicycle 123",  
    "2017-10-24T01:02:03+00:00": "created"  
}
```

DAX: How It Works

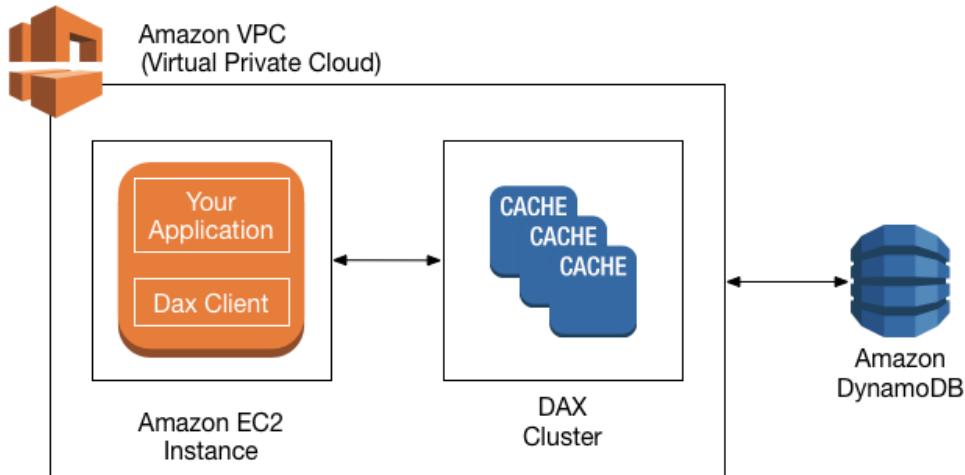
Amazon DynamoDB Accelerator (DAX) is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. The Amazon VPC service defines a virtual network that closely resembles a traditional data center. With a VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings. You can launch a DAX cluster in your virtual network and control access to the cluster by using Amazon VPC security groups.

Note

If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. The VPC is ready for you to use immediately—without having to perform any additional configuration steps.

For more information, see [Default VPC and Default Subnets](#) in the *Amazon VPC User Guide*.

The following diagram shows a high-level overview of DAX.



To create a DAX cluster, you use the AWS Management Console. Unless you specify otherwise, your DAX cluster runs within your default VPC. To run your application, you launch an Amazon EC2 instance into your Amazon VPC. You then deploy your application (with the DAX client) on the EC2 instance.

At runtime, the DAX client directs all of your application's DynamoDB API requests to the DAX cluster. If DAX can process one of these API requests directly, it does so. Otherwise, it passes the request through to DynamoDB.

Finally, the DAX cluster returns the results to your application.

Topics

- [How DAX Processes Requests \(p. 662\)](#)
- [Item Cache \(p. 663\)](#)
- [Query Cache \(p. 663\)](#)

How DAX Processes Requests

A DAX cluster consists of one or more nodes. Each node runs its own instance of the DAX caching software. One of the nodes serves as the primary node for the cluster. Additional nodes (if present) serve as read replicas. For more information, see [Nodes \(p. 664\)](#).

Your application can access DAX by specifying the endpoint for the DAX cluster. The DAX client software works with the cluster endpoint to perform intelligent load balancing and routing. Incoming requests are evenly distributed across all of the nodes in the cluster.

Read Operations

DAX can respond to the following API calls:

- [GetItem](#)
- [BatchGetItem](#)
- [Query](#)
- [Scan](#)

If the request specifies *eventually consistent reads* (the default behavior), it tries to read the item from DAX:

- If DAX has the item available (a *cache hit*), DAX returns the item to the application without accessing DynamoDB.
- If DAX does not have the item available (a *cache miss*), DAX passes the request through to DynamoDB. When it receives the response from DynamoDB, DAX returns the results to the application. But it also writes the results to the cache on the primary node.

Note

If there are any read replicas in the cluster, DAX automatically keeps the replicas in sync with the primary node. For more information, see [Clusters \(p. 664\)](#).

If the request specifies *strongly consistent reads*, DAX passes the request through to DynamoDB. The results from DynamoDB are not cached in DAX. Instead, they are simply returned to the application.

Write Operations

The following DAX API operations are considered "write-through":

- [BatchWriteItem](#)
- [UpdateItem](#)
- [DeleteItem](#)
- [PutItem](#)

With these operations, data is first written to the DynamoDB table, and then to the DAX cluster. The operation is successful only if the data is successfully written to *both* the table and to DAX.

Other Operations

DAX does not recognize any DynamoDB operations for managing tables (such as `CreateTable`, `UpdateTable`, and so on). If your application needs to perform these operations, it must access DynamoDB directly rather than using DAX.

For detailed information about DAX and DynamoDB consistency, see [DAX and DynamoDB Consistency Models \(p. 676\)](#).

For information about how transactions work in DAX, see [Using Transactional APIs in DynamoDB Accelerator \(DAX\) \(p. 652\)](#).

Request Rate Limiting

If the number of requests sent to DAX exceeds the capacity of a node, DAX limits the rate at which it accepts additional requests by returning a `ThrottlingException`. DAX continuously evaluates your CPU utilization to determine the volume of requests it can process while maintaining a healthy cluster state.

You can monitor the [ThrottledRequestCount metric \(p. 734\)](#) that DAX publishes to Amazon CloudWatch. If you see these exceptions regularly, you should consider [scaling up your cluster \(p. 728\)](#).

Item Cache

DAX maintains an *item cache* to store the results from `GetItem` and `BatchGetItem` operations. The items in the cache represent eventually consistent data from DynamoDB, and are stored by their primary key values.

When an application sends a `GetItem` or `BatchGetItem` request, DAX tries to read the items directly from the item cache using the specified key values. If the items are found (cache hit), DAX returns them to the application immediately. If the items are not found (cache miss), DAX sends the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads and returns the items to DAX. DAX stores them in the item cache and then returns them to the application.

The item cache has a Time to Live (TTL) setting, which is 5 minutes by default. DAX assigns a timestamp to every item that it writes to the item cache. An item expires if it has remained in the cache for longer than the TTL setting. If you issue a `GetItem` request on an expired item, this is considered a cache miss, and DAX sends the `GetItem` request to DynamoDB.

Note

You can specify the TTL setting for the item cache when you create a new DAX cluster. For more information, see [Managing DAX Clusters \(p. 726\)](#).

DAX also maintains a least recently used (LRU) list for the item cache. The LRU list tracks when an item was first written to the cache, and when the item was last read from the cache. If the item cache becomes full, DAX evicts older items (even if they haven't expired yet) to make room for new items. The LRU algorithm is always enabled for the item cache and is not user-configurable.

For detailed information about the consistency of the item cache in DAX, see [DAX Item Cache Behavior \(p. 676\)](#).

Query Cache

DAX also maintains a *query cache* to store the results from `Query` and `Scan` operations. The items in this cache represent result sets from queries and scans on DynamoDB tables. These result sets are stored by their parameter values.

When an application sends a `Query` or `Scan` request, DAX tries to read a matching result set from the query cache using the specified parameter values. If the result set is found (cache hit), DAX returns it

to the application immediately. If the result set is not found (cache miss), DAX sends the request to DynamoDB. DynamoDB processes the requests using eventually consistent reads and returns the result set to DAX. DAX stores it in the query cache and then returns it to the application.

Note

You can specify the TTL setting for the query cache when you create a new DAX cluster. For more information, see [Managing DAX Clusters \(p. 726\)](#).

DAX also maintains an LRU list for the query cache. The list tracks when a result set was first written to the cache, and when the result was last read from the cache. If the query cache becomes full, DAX evicts older result sets (even if they have not expired yet) to make room for new result sets. The LRU algorithm is always enabled for the query cache, and is not user-configurable.

For detailed information about the consistency of the query cache in DAX, see [DAX Query Cache Behavior \(p. 678\)](#).

DAX Cluster Components

An Amazon DynamoDB Accelerator (DAX) cluster consists of AWS infrastructure components. This section describes these components and how they work together.

Topics

- [Nodes \(p. 664\)](#)
- [Clusters \(p. 664\)](#)
- [Regions and Availability Zones \(p. 665\)](#)
- [Parameter Groups \(p. 666\)](#)
- [Security Groups \(p. 666\)](#)
- [Cluster ARN \(p. 666\)](#)
- [Cluster Endpoint \(p. 666\)](#)
- [Node Endpoints \(p. 666\)](#)
- [Subnet Groups \(p. 667\)](#)
- [Events \(p. 667\)](#)
- [Maintenance Window \(p. 667\)](#)

Nodes

A *node* is the smallest building block of a DAX cluster. Each node runs an instance of the DAX software, and maintains a single replica of the cached data.

You can scale your DAX cluster in one of two ways:

- By adding more nodes to the cluster. This increases the overall read throughput of the cluster.
- By using a larger node type. Larger node types provide more capacity and can increase throughput. (You must create a new cluster with the new node type.)

Every node within a cluster is of the same node type and runs the same DAX caching software. For a list of available node types, see [Amazon DynamoDB pricing](#).

Clusters

A *cluster* is a logical grouping of one or more nodes that DAX manages as a unit. One of the nodes in the cluster is designated as the *primary* node, and the other nodes (if any) are *read replicas*.

The primary node is responsible for the following:

- Fulfilling application requests for cached data.
- Handling write operations to DynamoDB.
- Evicting data from the cache according to the cluster's eviction policy.

When changes are made to cached data on the primary node, DAX propagates the changes to all of the read replica nodes.

Read replicas are responsible for the following:

- Fulfilling application requests for cached data.
- Evicting data from the cache according to the cluster's eviction policy.

However, unlike the primary node, read replicas don't write to DynamoDB.

Read replicas serve two additional purposes:

- **Scalability.** If you have a large number of application clients that need to access DAX concurrently, you can add more replicas for read-scaling. DAX spreads the load evenly across all the nodes in the cluster. (Another way to increase throughput is to use larger cache node types.)
- **High availability.** In the event of a primary node failure, DAX automatically fails over to a read replica and designates it as the new primary. If a replica node fails, other nodes in the DAX cluster can still serve requests until the failed node can be recovered. For maximum fault tolerance, you should deploy read replicas in separate Availability Zones. This configuration ensures that your DAX cluster can continue to function, even if an entire Availability Zone becomes unavailable.

A DAX cluster can support up to 10 nodes per cluster (the primary node, plus a maximum of nine read replicas).

Important

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

Regions and Availability Zones

A DAX cluster in an AWS Region can only interact with DynamoDB tables that are in the same Region. For this reason, ensure that you launch your DAX cluster in the correct Region. If you have DynamoDB tables in other Regions, you must launch DAX clusters in those Regions too.

Each Region is designed to be completely isolated from the other Regions. Within each Region are multiple Availability Zones. By launching your nodes in different Availability Zones, you can achieve the greatest possible fault tolerance.

Important

Don't place all of your cluster's nodes in a single Availability Zone. In this configuration, your DAX cluster becomes unavailable if there is an Availability Zone failure.

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in different Availability Zones. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

Parameter Groups

Parameter groups are used to manage runtime settings for DAX clusters. DAX has several parameters that you can use to optimize performance (such as defining a TTL policy for cached data). A parameter group is a named set of parameters that you can apply to a cluster. You can thereby ensure that all the nodes in that cluster are configured in exactly the same way.

Security Groups

A DAX cluster runs in an Amazon Virtual Private Cloud (Amazon VPC) environment. This environment is a virtual network that is dedicated to your AWS account and is isolated from other VPCs. A *security group* acts as a virtual firewall for your VPC, allowing you to control inbound and outbound network traffic.

When you launch a cluster in your VPC, you add an *ingress* rule to your security group to allow incoming network traffic. The ingress rule specifies the protocol (TCP) and port number (8111) for your cluster. After you add this rule to your security group, the applications that are running within your VPC can access the DAX cluster.

Cluster ARN

Every DAX cluster is assigned an *Amazon Resource Name* (ARN). The ARN format is as follows.

```
arn:aws:dax:region:accountID:cache/clusterName
```

You use the cluster ARN in an IAM policy to define permissions for DAX API operations. For more information, see [DAX Access Control \(p. 745\)](#).

Cluster Endpoint

Every DAX cluster provides a *cluster endpoint* for use by your application. By accessing the cluster using its endpoint, your application does not need to know the hostnames and port numbers of individual nodes in the cluster. Your application automatically "knows" all the nodes in the cluster, even if you add or remove read replicas.

The following is an example of a cluster endpoint.

```
myDAXcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Node Endpoints

Each of the individual nodes in a DAX cluster has its own hostname and port number. The following is an example of a *node endpoint*.

```
myDAXcluster-a.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111
```

Your application can access a node directly by using its endpoint. However, we recommend that you treat the DAX cluster as a single unit and access it using the cluster endpoint instead. The cluster endpoint insulates your application from having to maintain a list of nodes and keep that list up to date when you add or remove nodes from the cluster.

Subnet Groups

Access to DAX cluster nodes is restricted to applications running on Amazon EC2 instances within an Amazon VPC environment. You can use *subnet groups* to grant cluster access from Amazon EC2 instances running on specific subnets. A subnet group is a collection of subnets (typically private) that you can designate for your clusters running in an Amazon VPC environment.

When you create a DAX cluster, you must specify a subnet group. DAX uses that subnet group to select a subnet and IP addresses within that subnet to associate with your nodes.

Events

DAX records significant events within your clusters, such as a failure to add a node, success in adding a node, or changes to security groups. By monitoring key events, you can know the current state of your clusters and, depending upon the event, be able to take corrective action. You can access these events using the AWS Management Console or the `DescribeEvents` action in the DAX management API.

You can also request that notifications be sent to a specific Amazon Simple Notification Service (Amazon SNS) topic. Then you will know immediately when an event occurs in your DAX cluster.

Maintenance Window

Every cluster has a weekly maintenance window during which any system changes are applied. If you don't specify a preferred maintenance window when you create or modify a cache cluster, DAX assigns a 60-minute maintenance window on a randomly selected day of the week.

The 60-minute maintenance window is selected at random from an 8-hour block of time per AWS Region. The following table lists the time blocks for each Region from which the default maintenance windows are assigned.

| Region Code | Region Name | Maintenance Window |
|----------------|----------------------------------|--------------------|
| ap-northeast-1 | Asia Pacific (Tokyo) Region | 13:00–21:00 UTC |
| ap-southeast-1 | Asia Pacific (Singapore) Region | 14:00–22:00 UTC |
| ap-southeast-2 | Asia Pacific (Sydney) Region | 12:00–20:00 UTC |
| ap-south-1 | Asia Pacific (Mumbai) Region | 17:30–1:30 UTC |
| cn-northwest-1 | China (Ningxia) Region | 23:00–07:00 UTC |
| eu-central-1 | Europe (Frankfurt) Region | 23:00–07:00 UTC |
| eu-west-1 | Europe (Ireland) Region | 22:00–06:00 UTC |
| eu-west-2 | Europe (London) Region | 23:00–07:00 UTC |
| eu-west-3 | Europe (Paris) Region | 23:00–07:00 UTC |
| sa-east-1 | South America (São Paulo) Region | 01:00–09:00 UTC |
| us-east-1 | US East (N. Virginia) Region | 03:00–11:00 UTC |
| us-east-2 | US East (Ohio) Region | 23:00–07:00 UTC |
| us-west-1 | US West (N. California) Region | 06:00–14:00 UTC |

| Region Code | Region Name | Maintenance Window |
|-------------|-------------------------|--------------------|
| us-west-2 | US West (Oregon) Region | 06:00–14:00 UTC |

The maintenance window should fall at the time of lowest usage and thus might need modification from time to time. You can specify a time range of up to 24 hours in duration during which any maintenance activities that you request should occur.

Creating a DAX Cluster

This section walks you through the first-time setup and usage of Amazon DynamoDB Accelerator (DAX) in your default Amazon Virtual Private Cloud (Amazon VPC) environment. You can create your first DAX cluster using either the AWS Command Line Interface (AWS CLI) or the AWS Management Console.

After you create your DAX cluster, you can access it from an Amazon EC2 instance running in the same VPC. You can then use your DAX cluster with an application program. For more information, see [Developing with the DynamoDB Accelerator \(DAX\) Client \(p. 682\)](#).

Topics

- [Creating an IAM Service Role for DAX to Access DynamoDB \(p. 668\)](#)
- [Creating a DAX Cluster Using the AWS CLI \(p. 669\)](#)
- [Creating a DAX Cluster Using the AWS Management Console \(p. 673\)](#)

Creating an IAM Service Role for DAX to Access DynamoDB

For your DAX cluster to access DynamoDB tables on your behalf, you must create a *service role*. A service role is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables, as if you were accessing those tables yourself. You must create the service role before you can create the DAX cluster.

If you are using the console, the workflow for creating a cluster checks for the presence of a pre-existing DAX service role. If none is found, the console creates a new service role for you. For more information, see [the section called “Step 2: Create a DAX Cluster” \(p. 674\)](#).

If you are using the AWS CLI, you must specify a DAX service role that you have created previously. Otherwise, you need to create a new service role beforehand. For more information, see [Step 1: Create an IAM Service Role for DAX to Access DynamoDB Using the AWS CLI \(p. 670\)](#).

Permissions Required to Create a Service Role

The AWS managed `AdministratorAccess` policy provides all the permissions needed for creating a DAX cluster and a service role. If your IAM user has `AdministratorAccess` attached, no further action is needed.

Otherwise, you must add the following permissions to your IAM policy so that your IAM user can create the service role:

- `iam:CreateRole`
- `iam:CreatePolicy`
- `iam:AttachRolePolicy`

- `iam:PassRole`

Attach these permissions to the user who is trying to perform the action.

Note

The `iam:CreateRole`, `iam:CreatePolicy`, `iam:AttachPolicy`, and `iam:PassRole` permissions are not included in the AWS managed policies for DynamoDB. This is by design because these permissions provide the possibility of privilege escalation: That is, a user could use these permissions to create a new administrator policy and then attach that policy to an existing role. For this reason, you (the administrator of your DAX cluster) must explicitly add these permissions to your policy.

Troubleshooting

If your user policy is missing the `iam:CreateRole`, `iam:CreatePolicy`, and `iam:AttachPolicy` permissions, you will get error messages. The following table lists these messages and describes how to correct the problems.

| If you see this error message... | Do the following: |
|---|--|
| User: <code>arn:aws:iam::accountID:user/<i>userName</i></code> is not authorized to perform: <code>iam:CreateRole</code> on resource: <code>arn:aws:iam::accountID:role/service-role/<i>roleName</i></code> | Add <code>iam:CreateRole</code> to your user policy. |
| User: <code>arn:aws:iam::accountID:user/<i>userName</i></code> is not authorized to perform: <code>iam:CreatePolicy</code> on resource: policy <code><i>policyName</i></code> | Add <code>iam:CreatePolicy</code> to your user policy. |
| User: <code>arn:aws:iam::accountID:user/<i>userName</i></code> is not authorized to perform: <code>iam:AttachRolePolicy</code> on resource: role <code><i>daxServiceRole</i></code> | Add <code>iam:AttachRolePolicy</code> to your user policy. |

For more information about the IAM policies required for DAX cluster administration, see [DAX Access Control \(p. 745\)](#).

Creating a DAX Cluster Using the AWS CLI

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI). If you haven't already done so, you must install and configure the AWS CLI. To do this, see the following instructions in the *AWS Command Line Interface User Guide*:

- [Installing the AWS CLI](#)
- [Configuring the AWS CLI](#)

Important

To manage DAX clusters using the AWS CLI, install or upgrade to version 1.11.110 or higher.

All of the AWS CLI examples use the `us-west-2` Region and fictitious account IDs.

Topics

- [Step 1: Create an IAM Service Role for DAX to Access DynamoDB Using the AWS CLI \(p. 670\)](#)
- [Step 2: Create a Subnet Group \(p. 671\)](#)
- [Step 3: Create a DAX Cluster Using the AWS CLI \(p. 672\)](#)
- [Step 4: Configure Security Group Inbound Rules Using the AWS CLI \(p. 673\)](#)

Step 1: Create an IAM Service Role for DAX to Access DynamoDB Using the AWS CLI

Before you can create an Amazon DynamoDB Accelerator (DAX) cluster, you must create a service role for it. A *service role* is an AWS Identity and Access Management (IAM) role that authorizes an AWS service to act on your behalf. The service role allows DAX to access your DynamoDB tables as if you were accessing those tables yourself.

In this step, you create an IAM policy and then attach that policy to an IAM role. This enables you to assign the role to a DAX cluster so that it can perform DynamoDB operations on your behalf.

To create an IAM service role for DAX

1. Create a file named `service-trust-relationship.json` with the following contents.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "dax.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

2. Create the service role.

```
aws iam create-role \  
    --role-name DAXServiceRoleForDynamoDBAccess \  
    --assume-role-policy-document file://service-trust-relationship.json
```

3. Create a file named `service-role-policy.json` with the following contents.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dynamodb:DescribeTable",  
                "dynamodb:PutItem",  
                "dynamodb:GetItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:Query",  
                "dynamodb:Scan",  
                "dynamodb:BatchGetItem",  
                "dynamodb:BatchWriteItem",  
                "dynamodb:ConditionCheckItem"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

```
        "Effect": "Allow",
        "Resource": [
            "arn:aws:dynamodb:us-west-2:accountID:*"
        ]
    ]
}
```

Replace **accountID** with your AWS account ID. To find your AWS account ID, in the upper-right corner of the console, choose your login ID. Your AWS account ID appears in the drop-down menu.

In the Amazon Resource Name (ARN) in the example, **accountID** must be a 12-digit number. Don't use hyphens or any other punctuation.

4. Create an IAM policy for the service role.

```
aws iam create-policy \
--policy-name DAXServicePolicyForDynamoDBAccess \
--policy-document file://service-role-policy.json
```

In the output, note the ARN for the policy that you created, as in the following example.

```
arn:aws:iam::123456789012:policy/DAXServicePolicyForDynamoDBAccess
```

5. Attach the policy to the service role. Replace **arn** in the following code with the actual role ARN from the previous step.

```
aws iam attach-role-policy \
--role-name DAXServiceRoleForDynamoDBAccess \
--policy-arn arn
```

Next, you specify a subnet group for your default VPC. A *subnet group* is a collection of one or more subnets within your VPC. See [Step 2: Create a Subnet Group \(p. 671\)](#).

Step 2: Create a Subnet Group

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Command Line Interface (AWS CLI).

Note

If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud environment (Amazon VPC). If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see [Default VPC and Default Subnets](#) in the *Amazon VPC User Guide*.

To create a subnet group

1. To determine the identifier for your default VPC, enter the following command.

```
aws ec2 describe-vpcs
```

In the output, note the identifier for your default VPC, as in the following example.

```
vpc-12345678
```

2. Determine the subnet IDs associated with your default VPC. Replace **vpcID** with your actual VPC ID—for example, vpc-12345678.

```
aws ec2 describe-subnets \
--filters "Name=vpc-id,Values=vpcID" \
--query "Subnets[*].SubnetId"
```

In the output, note the subnet identifiers—for example, subnet-1111111.

3. Create the subnet group. Ensure that you specify at least one subnet ID in the --subnet-ids parameter.

```
aws dax create-subnet-group \
--subnet-group-name my-subnet-group \
--subnet-ids subnet-11111111 subnet-22222222 subnet-33333333 subnet-44444444
```

To create the cluster, see [Step 3: Create a DAX Cluster Using the AWS CLI \(p. 672\)](#).

Step 3: Create a DAX Cluster Using the AWS CLI

Follow this procedure to use the AWS Command Line Interface (AWS CLI) to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Get the Amazon Resource Name (ARN) for your service role.

```
aws iam get-role \
--role-name DAXServiceRoleForDynamoDBAccess \
--query "Role.Arn" --output text
```

In the output, note the service role ARN, as in the following example.

arn:aws:iam::123456789012:role/DAXServiceRoleForDynamoDBAccess

2. Create the DAX cluster. Replace `roleARN` with the ARN from the previous step.

```
aws dax create-cluster \
--cluster-name mydaxcluster \
--node-type dax.r4.large \
--replication-factor 3 \
--iam-role-arn roleARN \
--subnet-group my-subnet-group \
--sse-specification Enabled=true \
--region us-west-2
```

All of the nodes in the cluster are of type `dax.r4.large` (--node-type). There are three nodes (--replication-factor)—one primary node and two replicas.

To view the cluster status, enter the following command.

```
aws dax describe-clusters
```

The status is shown in the output—for example, "Status": "creating".

Note

Creating the cluster takes several minutes. When the cluster is ready, its status changes to available. In the meantime, proceed to [Step 4: Configure Security Group Inbound Rules Using the AWS CLI \(p. 673\)](#) and follow the instructions there.

Step 4: Configure Security Group Inbound Rules Using the AWS CLI

The nodes in your Amazon DynamoDB Accelerator (DAX) cluster use the default security group for your Amazon VPC. For the default security group, you must authorize inbound traffic on TCP port 8111. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

Note

If you launched your DAX cluster with a different security group (other than `default`), you must perform this procedure for that group instead.

To configure security group inbound rules

1. To determine the default security group identifier, enter the following command. Replace `vpcID` with your actual VPC ID (from [Step 2: Create a Subnet Group \(p. 671\)](#)).

```
aws ec2 describe-security-groups \
--filters Name=vpc-id,Values=vpcID,Name=group-name,Values=default \
--query "SecurityGroups[*].{GroupName:GroupName,GroupId:GroupId}"
```

In the output, note the security group identifier—for example, `sg-01234567`.

2. Then enter the following. Replace `sgID` with your actual security group identifier.

```
aws ec2 authorize-security-group-ingress \
--group-id sgID --protocol tcp --port 8111
```

Creating a DAX Cluster Using the AWS Management Console

This section describes how to create an Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

Topics

- [Step 1: Create a Subnet Group Using the AWS Management Console \(p. 673\)](#)
- [Step 2: Create a DAX Cluster Using the AWS Management Console \(p. 674\)](#)
- [Step 3: Configure Security Group Inbound Rules Using the AWS Management Console \(p. 675\)](#)

Step 1: Create a Subnet Group Using the AWS Management Console

Follow this procedure to create a subnet group for your Amazon DynamoDB Accelerator (DAX) cluster using the AWS Management Console.

Note

If you already created a subnet group for your default VPC, you can skip this step.

DAX is designed to run within an Amazon Virtual Private Cloud (Amazon VPC) environment. If you created your AWS account after December 4, 2013, you already have a default VPC in each AWS Region. For more information, see [Default VPC and Default Subnets](#) in the *Amazon VPC User Guide*.

As part of the creation process for a DAX cluster, you must specify a *subnet group*. A subnet group is a collection of one or more subnets within your VPC. When you create your DAX cluster, the nodes are deployed to the subnets within the subnet group.

To create a subnet group

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **DAX**.
3. Choose **Create subnet group**.
4. In the **Create subnet group** window, do the following:
 - a. **Name**—Enter a short name for the subnet group.
 - b. **Description**—Enter a description for the subnet group.
 - c. **VPC ID**—Choose the identifier for your Amazon VPC environment.
 - d. **Subnets**—Choose one or more subnets from the list.

Note

The subnets are distributed among multiple Availability Zones. If you plan to create a multi-node DAX cluster (a primary node and one or more read replicas), we recommend that you choose multiple subnet IDs. Then DAX can deploy the cluster nodes into multiple Availability Zones. If an Availability Zone becomes unavailable, DAX automatically fails over to a surviving Availability Zone. Your DAX cluster will continue to function without interruption.

When the settings are as you want them, choose **Create subnet group**.

To create the cluster, see [Step 2: Create a DAX Cluster Using the AWS Management Console \(p. 674\)](#).

Step 2: Create a DAX Cluster Using the AWS Management Console

Follow this procedure to create an Amazon DynamoDB Accelerator (DAX) cluster in your default Amazon VPC.

To create a DAX cluster

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. In the **Create cluster** window, do the following:
 - a. **Cluster name**—Enter a short name for your DAX cluster.
 - b. **Cluster description**—Enter a description for the cluster.
 - c. **Node type**—Choose the node type for all of the nodes in the cluster.
 - d. **Cluster size**—Choose the number of nodes in the cluster. A cluster consists of one primary node and up to nine read replicas.

Note

If you want to create a single-node cluster, choose **1**. Your cluster will consist of one primary node.

If you want to create a multi-node cluster, choose a number between **3** (one primary and two read replicas) and **10** (one primary and nine read replicas).

Important

For production usage, we strongly recommend using DAX with at least three nodes, where each node is placed in a different Availability Zone. Three nodes are required for a DAX cluster to be fault-tolerant.

A DAX cluster can be deployed with one or two nodes for development or test workloads. One- and two-node clusters are not fault-tolerant, and we don't recommend using fewer than three nodes for production use. If a one- or two-node cluster encounters software or hardware errors, the cluster can become unavailable or lose cached data.

- e. **Encryption**—Choose **enable encryption** for your DAX cluster to help protect data at rest. For more information, see [DAX Encryption at Rest \(p. 755\)](#).
- f. **IAM service role for DynamoDB access**—Choose **Create new**, and enter the following information:
 - **IAM role name**—Enter a name for an IAM role, for example, `DAXServiceRole`. The console creates a new IAM role, and your DAX cluster assumes this role at runtime.
 - **IAM policy name**—Enter a name for an IAM policy, for example, `DAXServicePolicy`. The console creates a new IAM policy and attaches the policy to the IAM role.
 - **IAM role policy**—Choose **Read/Write**. This allows the DAX cluster to perform read and write operations in DynamoDB.
 - **Target DynamoDB table**—Choose **All tables**.
- g. **Subnet group**—Choose the subnet group that you created in [Step 1: Create a Subnet Group Using the AWS Management Console \(p. 673\)](#).
- h. **Security Groups**—Choose **default**.

A separate service role for DAX to access Amazon EC2 is also required. DAX automatically creates this service role for you. For more information, see [Using Service-Linked Roles for DAX](#).

5. When the settings are as you want them, choose **Launch cluster**.

On the **Clusters** screen, your DAX cluster will be listed with a status of **Creating**.

Note

Creating the cluster takes several minutes. When the cluster is ready, its status changes to **Available**.

In the meantime, proceed to [Step 3: Configure Security Group Inbound Rules Using the AWS Management Console \(p. 675\)](#) and follow the instructions there.

Step 3: Configure Security Group Inbound Rules Using the AWS Management Console

Your Amazon DynamoDB Accelerator (DAX) cluster uses TCP port 8111 for communication, so you must authorize inbound traffic on that port. This allows Amazon EC2 instances in your Amazon VPC to access your DAX cluster.

Note

If you launched your DAX cluster with a different security group (other than `default`), you must perform this procedure for that group instead.

To configure security group inbound rules

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the navigation pane, choose **Security Groups**.
3. Choose the **default** security group. On the **Actions** menu, choose **Edit inbound rules**.
4. Choose **Add Rule**, and enter the following information:
 - **Port Range**—Enter **8111**.

Source—Enter **default**, and then choose the identifier for your default security group.

When the settings are as you want them, choose **Save**.

DAX and DynamoDB Consistency Models

Amazon DynamoDB Accelerator (DAX) is a write-through caching service that is designed to simplify the process of adding a cache to DynamoDB tables. Because DAX operates separately from DynamoDB, it is important that you understand the consistency models of both DAX and DynamoDB to ensure that your applications behave as you expect.

In many use cases, the way that your application uses DAX affects the consistency of data within the DAX cluster, and the consistency of data between DAX and DynamoDB.

Topics

- [Consistency Among DAX Cluster Nodes \(p. 676\)](#)
- [DAX Item Cache Behavior \(p. 676\)](#)
- [DAX Query Cache Behavior \(p. 678\)](#)
- [Strongly Consistent and Transactional Reads \(p. 679\)](#)
- [Negative Caching \(p. 679\)](#)
- [Strategies for Writes \(p. 680\)](#)

Consistency Among DAX Cluster Nodes

To achieve high availability for your application, we recommend that you provision your DAX cluster with at least three nodes. Then place those nodes in multiple Availability Zones within a Region.

When your DAX cluster is running, it replicates the data among all of the nodes in the cluster (assuming that you provisioned more than one node). Consider an application that performs a successful `UpdateItem` using DAX. This action causes the item cache in the primary node to be modified with the new value. That value is then replicated to all the other nodes in the cluster. This replication is eventually consistent and usually takes less than one second to complete.

In this scenario, it's possible for two clients to read the same key from the same DAX cluster but receive different values, depending on the node that each client accessed. The nodes are all consistent when the update has been fully replicated throughout all the nodes in the cluster. (This behavior is similar to the eventually consistent nature of DynamoDB.)

If you are building an application that uses DAX, that application should be designed so that it can tolerate eventually consistent data.

DAX Item Cache Behavior

Every DAX cluster has two distinct caches—an *item* cache and a *query* cache. For more information, see [DAX: How It Works \(p. 661\)](#).

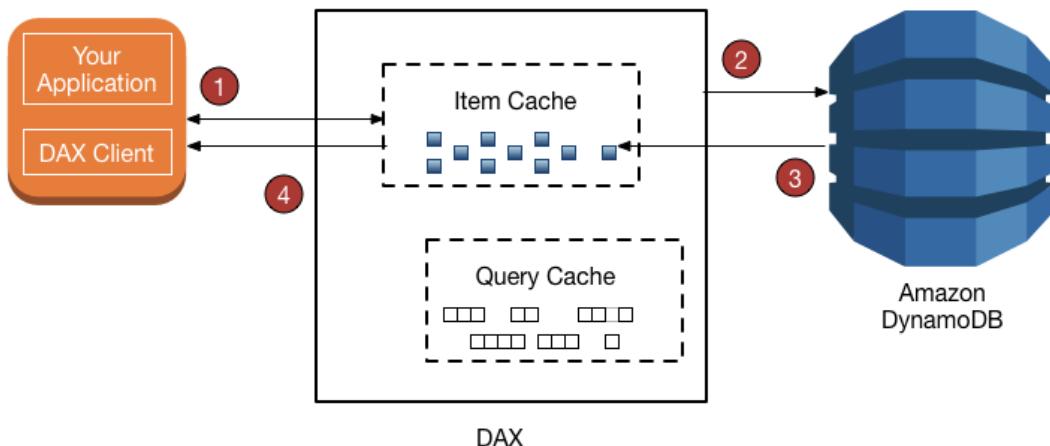
This section addresses the consistency implications of reading from and writing to the DAX item cache.

Consistency of Reads

With DynamoDB, the `GetItem` operation performs an eventually consistent read by default. Suppose that you use `UpdateItem` with the DynamoDB client. If you then try to read the same item immediately afterward, you might see the data as it appeared before the update. This is due to propagation delay

across all the DynamoDB storage locations. Consistency is usually reached within seconds. So if you retry the read, you will likely see the updated item.

When you use `GetItem` with the DAX client, the operation (in this case, an eventually consistent read) proceeds as shown following.



1. The DAX client issues a `GetItem` request. DAX tries to read the requested item from the item cache. If the item is in the cache (*cache hit*), DAX returns it to the application.
2. If the item is not available (*cache miss*), DAX performs an eventually consistent `GetItem` operation against DynamoDB.
3. DynamoDB returns the requested item, and DAX stores it in the item cache.
4. DAX returns the item to the application.
5. (Not shown) If the DAX cluster contains more than one node, the item is replicated to all the other nodes in the cluster.

The item remains in the DAX item cache, subject to the Time to Live (TTL) setting and the least recently used (LRU) algorithm for the cache. For more information, see [DAX: How It Works \(p. 661\)](#).

However, during this period, DAX doesn't re-read the item from DynamoDB. If someone else updates the item using a DynamoDB client, bypassing DAX entirely, a `GetItem` request using the DAX client yields different results from the same `GetItem` request using the DynamoDB client. In this scenario, DAX and DynamoDB hold inconsistent values for the same key until the TTL for the DAX item expires.

If an application modifies data in an underlying DynamoDB table, bypassing DAX, the application needs to anticipate and tolerate data inconsistencies that might arise.

Note

In addition to `GetItem`, the DAX client also supports `BatchGetItem` requests. `BatchGetItem` is essentially a wrapper around one or more `GetItem` requests, so DAX treats each of these as an individual `GetItem` operation.

Consistency of Writes

DAX is a write-through cache, which simplifies the process of keeping the DAX item cache consistent with the underlying DynamoDB tables.

The DAX client supports the same write API operations as DynamoDB (`PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`). When you use these operations with the DAX client, the items are modified in both DAX and DynamoDB. DAX updates the items in its item cache, regardless of the TTL value for these items.

For example, suppose that you issue a `GetItem` request from the DAX client to read an item from the `ProductCatalog` table. (The partition key is `Id`, and there is no sort key.) You retrieve the item whose `Id` is 101. The `QuantityOnHand` value for that item is 42. DAX stores the item in its item cache with a specific TTL. For this example, assume that the TTL is 10 minutes. Then, 3 minutes later, another application uses the DAX client to update the same item so that its `QuantityOnHand` value is now 41. Assuming that the item is not updated again, any subsequent reads of the same item during the next 10 minutes return the cached value for `QuantityOnHand` (41).

How DAX Processes Writes

DAX is intended for applications that require high-performance reads. As a write-through cache, DAX allows you to issue writes directly so that your writes are immediately reflected in the item cache. You don't need to manage cache invalidation logic because DAX handles it for you.

DAX supports the following write operations: `PutItem`, `UpdateItem`, `DeleteItem`, `BatchWriteItem`, and `TransactWriteItems`.

When you send a `PutItem`, `UpdateItem`, `DeleteItem`, or `BatchWriteItem` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the write succeeded.
- DAX writes the item to its item cache.
- DAX returns success to the requester.

When you send a `TransactWriteItems` request to DAX, the following occurs:

- DAX sends the request to DynamoDB.
- DynamoDB replies to DAX, confirming that the transaction completed.
- DAX returns success to the requester.
- In the background, DAX makes a `TransactGetItems` request for each item in the `TransactWriteItems` request to store the item in the item cache. `TransactGetItems` is used to ensure [serializable isolation \(p. 651\)](#).

If a write to DynamoDB fails for any reason, including throttling, the item is not cached in DAX. The exception for the failure is returned to the requester. This ensures that data is not written to the DAX cache unless it is first written successfully to DynamoDB.

Note

Every write to DAX alters the state of the item cache. However, writes to the item cache don't affect the query cache. (The DAX item cache and query cache serve different purposes, and operate independently from one another.)

DAX Query Cache Behavior

DAX caches the results from `Query` and `Scan` requests in its query cache. However, these results don't affect the item cache at all. When your application issues a `Query` or `Scan` request with DAX, the result set is saved in the query cache—not in the item cache. You can't "warm up" the item cache by performing a `Scan` operation because the item cache and query cache are separate entities.

Consistency of Query-Update-Query

Updates to the item cache, or to the underlying DynamoDB table, do not invalidate or modify the results stored in the query cache.

To illustrate, consider the following scenario. An application is working with the `DocumentRevisions` table, which has `DocId` as its partition key and `RevisionNumber` as its sort key.

1. A client issues a `Query` for `DocId` 101, for all items with `RevisionNumber` greater than or equal to 5. DAX stores the result set in the query cache and returns the result set to the user.
2. The client issues a `PutItem` request for `DocId` 101 with a `RevisionNumber` value of 20.
3. The client issues the same `Query` as described in step 1 (`DocId` 101 and `RevisionNumber >= 5`).

In this scenario, the cached result set for the `Query` issued in step 3 is identical to the result set that was cached in step 1. The reason is that DAX does not invalidate `Query` or `Scan` result sets based on updates to individual items. The `PutItem` operation from step 2 is only reflected in the DAX query cache when the TTL for the `Query` expires.

Your application should consider the TTL value for the query cache and how long your application can tolerate inconsistent results between the query cache and the item cache.

Strongly Consistent and Transactional Reads

To perform a strongly consistent `GetItem`, `BatchGetItem`, `Query`, or `Scan` request, you set the `ConsistentRead` parameter to true. DAX passes strongly consistent read requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it does not cache the results. DAX can't serve strongly consistent reads by itself because it's not tightly coupled to DynamoDB. For this reason, any subsequent reads from DAX would have to be eventually consistent reads. And any subsequent strongly consistent reads would have to be passed through to DynamoDB.

DAX handles `TransactGetItems` requests the same way it handles strongly consistent reads. DAX passes all `TransactGetItems` requests to DynamoDB. When it receives a response from DynamoDB, DAX returns the results to the client, but it doesn't cache the results.

Negative Caching

DAX supports negative cache entries in both the item cache and the query cache. A *negative cache entry* occurs when DAX can't find requested items in an underlying DynamoDB table. Instead of generating an error, DAX caches an empty result and returns that result to the user.

For example, suppose that an application sends a `GetItem` request to a DAX cluster, and that there is no matching item in the DAX item cache. This causes DAX to read the corresponding item from the underlying DynamoDB table. If the item doesn't exist in DynamoDB, DAX stores an empty item in its item cache and returns the empty item to the application. Now suppose that the application sends another `GetItem` request for the same item. DAX finds the empty item in the item cache and returns it to the application immediately. It does not consult DynamoDB at all.

A negative cache entry remains in the DAX item cache until its item TTL has expired, its LRU is invoked, or the item is modified using `PutItem`, `UpdateItem`, or `DeleteItem`.

The DAX query cache handles negative cache results in a similar way. If an application performs a `Query` or `Scan`, and the DAX query cache doesn't contain a cached result, DAX sends the request to DynamoDB. If there are no matching items in the result set, DAX stores an empty result set in the query cache and returns the empty result set to the application. Subsequent `Query` or `Scan` requests yield the same (empty) result set, until the TTL for that result set has expired.

Strategies for Writes

The write-through behavior of DAX is appropriate for many application patterns. However, there are some application patterns where a write-through model might not be appropriate.

For applications that are sensitive to latency, writing through DAX incurs an extra network hop. So a write to DAX is a little slower than a write directly to DynamoDB. If your application is sensitive to write latency, you can reduce the latency by writing directly to DynamoDB instead. For more information, see [Write-Around \(p. 681\)](#).

For write-intensive applications (such as those that perform bulk data loading), you might not want to write all of the data through DAX because only a small percentage of that data is ever read by the application. When you write large amounts of data through DAX, it must invoke its LRU algorithm to make room in the cache for the new items to be read. This diminishes the effectiveness of DAX as a read cache.

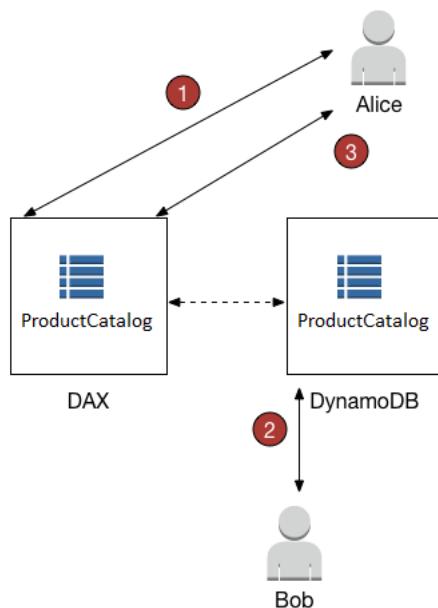
When you write an item to DAX, the item cache state is altered to accommodate the new item. (For example, DAX might need to evict older data from the item cache to make room for the new item.) The new item remains in the item cache, subject to the cache's LRU algorithm and the TTL setting for the cache. As long as the item persists in the item cache, DAX doesn't re-read the item from DynamoDB.

Write-Through

The DAX item cache implements a write-through policy. For more information, see [How DAX Processes Writes \(p. 678\)](#).

When you write an item, DAX ensures that the cached item is synchronized with the item as it exists in DynamoDB. This is helpful for applications that need to re-read an item immediately after writing it. However, if other applications write directly to a DynamoDB table, the item in the DAX item cache is no longer in sync with DynamoDB.

To illustrate, consider two users (Alice and Bob), who are working with the `ProductCatalog` table. Alice accesses the table using DAX, but Bob bypasses DAX and accesses the table directly in DynamoDB.



1. Alice updates an item in the `ProductCatalog` table. DAX forwards the request to DynamoDB, and the update succeeds. DAX then writes the item to its item cache and returns a successful response to Alice. From that point on, until the item is ultimately evicted from the cache, any user who reads the item from DAX sees the item with Alice's update.
2. A short time later, Bob updates the same `ProductCatalog` item that Alice wrote. However, Bob updates the item directly in DynamoDB. DAX does not automatically refresh its item cache in response to updates via DynamoDB. Therefore, DAX users don't see Bob's update.
3. Alice reads the item from DAX again. The item is in the item cache, so DAX returns it to Alice without accessing the DynamoDB table.

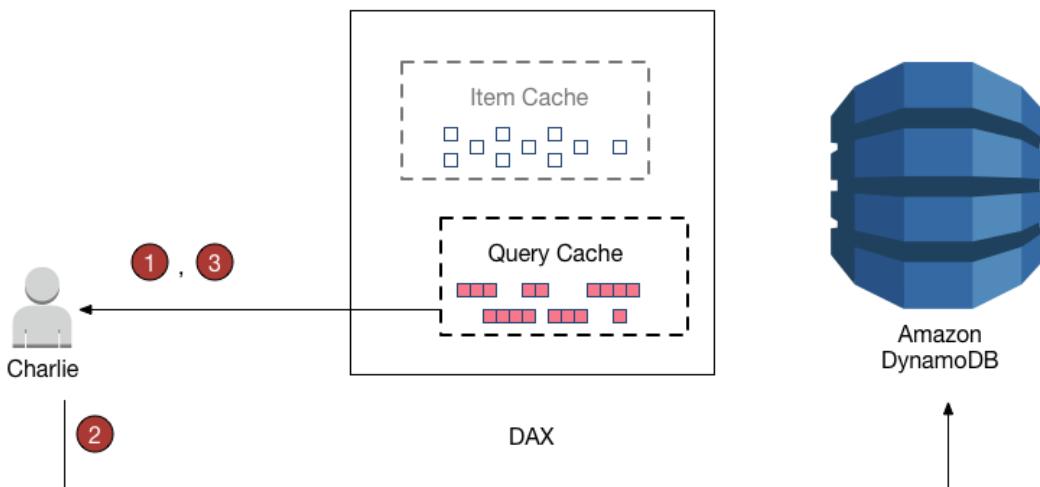
In this scenario, Alice and Bob see different representations of the same `ProductCatalog` item. This is the case until DAX evicts the item from the item cache, or until another user updates the same item again using DAX.

Write-Around

If your application needs to write large quantities of data (such as a bulk data load), it might make sense to bypass DAX and write the data directly to DynamoDB. Such a *write-around* strategy reduces write latency. However, the item cache doesn't remain in sync with the data in DynamoDB.

If you decide to use a write-around strategy, remember that DAX populates its item cache whenever applications use the DAX client to read data. This can be advantageous in some cases because it ensures that only the most frequently read data is cached (as opposed to the most frequently written data).

For example, consider a user (Charlie) who wants to work with a different table, the `GameScores` table, using DAX. The partition key for `GameScores` is `UserId`, so all of Charlie's scores would have the same `UserId`.



1. Charlie wants to retrieve all of his scores, so he sends a `Query` to DAX. Assuming that this query has not been issued before, DAX forwards the query to DynamoDB for processing. It stores the results in the DAX query cache, and then returns the results to Charlie. The result set remains available in the query cache until it is evicted.
2. Now suppose that Charlie plays the Meteor Blasters game and achieves a high score. Charlie sends an `UpdateItem` request to DynamoDB, modifying an item in the `GameScores` table.

3. Finally, Charlie decides to rerun his earlier `Query` to retrieve all of his data from `GameScores`. Charlie does not see his high score for Meteor Blasters in the results. This is because the query results come from the query cache, not the item cache. The two caches are independent from one another, so a change in one cache does not affect the other cache.

DAX does not refresh result sets in the query cache with the most current data from DynamoDB. Each result set in the query cache is current as of the time that the `Query` or `Scan` operation was performed. Thus, Charlie's `Query` results don't reflect his `PutItem` operation. This is the case until DAX evicts the result set from the query cache.

Developing with the DynamoDB Accelerator (DAX) Client

To use DAX from an application, you use the DAX client for your programming language. The DAX client is designed for minimal disruption to your existing Amazon DynamoDB applications—with only a few simple code modifications needed.

Note

DAX clients for various programming languages are available on the following site:

- <http://dax-sdk.s3-website-us-west-2.amazonaws.com>

This section demonstrates how to launch an Amazon EC2 instance in your default Amazon VPC, connect to the instance, and run a sample application. It also provides information about how to modify your existing application so that it can use your DAX cluster.

Topics

- [Tutorial: Running a Sample Application Using DynamoDB Accelerator \(DAX\) \(p. 682\)](#)
- [Modifying an Existing Application to Use DAX \(p. 720\)](#)
- [Querying Global Secondary Indexes \(p. 723\)](#)

Tutorial: Running a Sample Application Using DynamoDB Accelerator (DAX)

This tutorial demonstrates how to launch an Amazon EC2 instance in your default virtual private cloud (VPC), connect to the instance, and run a sample application that uses Amazon DynamoDB Accelerator (DAX).

Note

To complete this tutorial, you must have a DAX cluster running in your default VPC. If you haven't created a DAX cluster, see [Creating a DAX Cluster \(p. 668\)](#) for instructions.

Topics

- [Step 1: Launch an Amazon EC2 Instance \(p. 683\)](#)
- [Step 2: Create an IAM User and Policy \(p. 684\)](#)
- [Step 3: Configure an Amazon EC2 Instance \(p. 685\)](#)
- [Step 4: Run a Sample Application \(p. 685\)](#)

Step 1: Launch an Amazon EC2 Instance

When your Amazon DynamoDB Accelerator (DAX) cluster is available, you can launch an Amazon EC2 instance in your default Amazon VPC. You can then install and run DAX client software on that instance.

To launch an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose **Launch Instance**, and do the following:

Step 1: Choose an Amazon Machine Image (AMI)

1. In the list of AMIs, find the **Amazon Linux AMI**, and choose **Select**.

Step 2: Choose an Instance Type

1. In the list of instance types, choose **t2.micro**.
2. Choose **Next: Configure Instance Details**.

Step 3: Configure Instance Details

1. For **Network**, choose your default VPC.
2. Choose **Next: Add Storage**.

Step 4: Add Storage

1. Skip this step by choosing **Next: Add Tags**.

Step 5: Add Tags

1. Skip this step by choosing **Next: Configure Security Group**.

Step 6: Configure Security Group

1. Choose **Select an existing security group**.
2. In the list of security groups, choose **default**. This is the default security group for your VPC.
3. Choose **Next: Review and Launch**.

Step 7: Review Instance Launch

1. Choose **Launch**.
3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
 - If you don't have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You are asked to download a private key file (.pem file). You need this file later when you log in to your Amazon EC2 instance.
 - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (.pem file) available in order to log in to your Amazon EC2 instance.
4. After configuring your key pair, choose **Launch Instances**.

5. In the console navigation pane, choose **EC2 Dashboard**, and then choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance, for example: ec2-11-22-33-44.us-west-2.compute.amazonaws.com. Make a note of this public DNS name because you need it for [Step 3: Configure an Amazon EC2 Instance \(p. 685\)](#).

Note

It takes a few minutes for your Amazon EC2 instance to become available. In the meantime, proceed to [Step 2: Create an IAM User and Policy \(p. 684\)](#) and follow the instructions there.

Step 2: Create an IAM User and Policy

In this step, you create an AWS Identity and Access Management (IAM) user with a policy that grants access to your Amazon DynamoDB Accelerator (DAX) cluster and to DynamoDB. You can then run applications that interact with your DAX cluster.

To create an IAM user and policy

1. Open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, choose **Users**.
3. Choose **Add user**.
4. On the **Details** page, enter the following information:
 - **User name**—Enter a unique name, for example: MyDAXUser.
 - **Access type**—Choose **Programmatic access**.

Choose **Next: Permissions**.

5. On the **Set permissions** page, choose **Attach existing policies directly**, and then choose **Create policy**.
6. On the **Create policy** page, choose **Create Your Own Policy**.
7. On the **Review policy** page, provide the following information:
 - **Policy Name**—Enter a unique name, for example, MyDAXUserPolicy.
 - **Description**—Enter a short description for the policy.
 - **Policy Document**—Copy and paste the following document.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        },  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

```
}
```

Choose **Create policy**.

8. Return to the **Permissions** page. In the list of policies, choose **Refresh**.
9. To narrow the list of policies, choose **Filter, Customer managed**. Choose the IAM policy that you created in the previous step (for example: `MyDAXUserPolicy`), and then choose **Next: Review**.
10. On the **Review** page, choose **Create user**.
11. On the **Complete** page, go to the **Secret access key** and choose **Show**. After you do this, copy both the **Access key ID** and **Secret access key**. You need both of these identifiers for [Step 3: Configure an Amazon EC2 Instance \(p. 685\)](#).

Step 3: Configure an Amazon EC2 Instance

When your Amazon EC2 instance is available, you can log in to the instance and prepare it for use.

Note

The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see [Connect to Your Linux Instance](#) in the [Amazon EC2 User Guide for Linux Instances](#).

To configure the EC2 instance

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Use the `ssh` command to log in to your Amazon EC2 instance, as shown in the following example.

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

You need to specify your private key file (`.pem` file) and the public DNS name of your instance. (See [Step 1: Launch an Amazon EC2 Instance \(p. 683\)](#).)

The login ID is `ec2-user`. No password is required.

3. After you log in to your EC2 instance, configure your AWS credentials as shown following. Enter your AWS access key ID and secret key (from [Step 2: Create an IAM User and Policy \(p. 684\)](#)), and set the default Region name to your current Region. (In the following example, the default Region name is `us-west-2`.)

```
aws configure

AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJAlrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]:
```

After launching and configuring your Amazon EC2 instance, you can test the functionality of DAX using one of the available sample applications. For more information, see [Step 4: Run a Sample Application \(p. 685\)](#).

Step 4: Run a Sample Application

To help you test Amazon DynamoDB Accelerator (DAX) functionality, you can run one of the available sample applications on your Amazon EC2 instance.

Topics

- [DAX SDK for Go \(p. 686\)](#)
- [Java and DAX \(p. 688\)](#)
- [.NET and DAX \(p. 697\)](#)
- [Node.js and DAX \(p. 706\)](#)
- [Python and DAX \(p. 713\)](#)

DAX SDK for Go

Follow this procedure to run the Amazon DynamoDB Accelerator (DAX) SDK for Go sample application on your Amazon EC2 instance.

To run the SDK for Go sample for DAX

1. Set up the SDK for Go on your Amazon EC2 instance:

- a. Install the Go programming language (Golang).

```
sudo yum install -y golang
```

- b. Set the GOPATH environment variable.

```
# GOROOT is the location where Go package is installed on your system
export GOROOT=/usr/lib/golang

# GOPATH is the location of your work directory
export GOPATH=$HOME/projects

# PATH in order to access go binary system wide
export PATH=$PATH:$GOROOT/bin
```

Note

The preceding commands set the environment variables for your current session only. To make these settings permanent, add the commands in the `~/.bash_profile` file.

- c. Test that Golang is installed and running correctly.

```
go version
```

The following message should appear.

```
go version go1.9.6 linux/amd64/
```

2. Install the DAX Golang client.

```
go get github.com/aws/aws-dax-go
```

3. Install the sample Golang application.

```
go get github.com/aws-samples/aws-dax-go-sample
```

4. Run the following Golang programs. The first program creates a DynamoDB table named `TryDaxGoTable`. The second program writes data to the table.

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command create-table
```

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command put-item
```

- Run the following Golang programs.

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command get-item
```

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command query
```

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command scan
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

- In the previous step, you ran the programs against the DynamoDB endpoint. Now, run the programs again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

- Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output, as in the following example.

```
{  
    "Port": 8111,  
    "Address": "mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com"  
}
```

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dax -command get-item -endpoint mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dax -command query -endpoint mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dax -command scan -endpoint mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for GetItem, Query, and Scan should be significantly lower with DAX than with DynamoDB.

7. Run the following Golang program to delete TryDaxGoTable.

```
go run $GOPATH/src/github.com/aws-samples/aws-dax-go-sample/try_dax.go -service dynamodb -command delete-table
```

Java and DAX

Follow this procedure to run the Java sample for Amazon DynamoDB Accelerator (DAX) on your Amazon EC2 instance.

To run the Java sample for DAX

1. Install the Java Development Kit (JDK).

```
sudo yum install -y java-devel
```

2. Download the AWS SDK for Java (.zip file), and then extract it.

```
wget http://sdk-for-java.amazonwebservices.com/latest/aws-java-sdk.zip
unzip aws-java-sdk.zip
```

3. Download the latest version of the DAX Java client (.jar file).

```
wget http://dax-sdk.s3-website-us-west-2.amazonaws.com/java/DaxJavaClient-latest.jar
```

Note

The client for the DAX SDK for Java is available on Apache Maven. For more information, see [Using the Client as an Apache Maven Dependency](#) (p. 690).

4. Set your CLASSPATH variable. In this example, replace *sdkVersion* with the actual version number of the AWS SDK for Java (for example, 1.11.112).

```
export SDKVERSION=$sdkVersion
export CLASSPATH=.:./DaxJavaClient-latest.jar:aws-java-sdk-$SDKVERSION/lib/aws-java-
sdk-$SDKVERSION.jar:aws-java-sdk-$SDKVERSION/third-party/lib/*
```

5. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

6. Compile the code as follows.

```
javac TryDax*.java
```

7. Run the program.

```
java TryDax
```

You should see output similar to the following.

```
Creating a DynamoDB client

Attempting to create table; please wait...
Successfully created table.  Table status: ACTIVE
Writing data to the table...
Writing 10 items for partition key: 1
Writing 10 items for partition key: 2
Writing 10 items for partition key: 3
Writing 10 items for partition key: 4
Writing 10 items for partition key: 5
Writing 10 items for partition key: 6
Writing 10 items for partition key: 7
Writing 10 items for partition key: 8
Writing 10 items for partition key: 9
Writing 10 items for partition key: 10

Running GetItem, Scan, and Query tests...
First iteration of each test will result in cache misses
Next iterations are cache hits

GetItem test - partition key 1 and sort keys 1-10
Total time: 136.681 ms - Avg time: 13.668 ms
Total time: 122.632 ms - Avg time: 12.263 ms
Total time: 167.762 ms - Avg time: 16.776 ms
Total time: 108.130 ms - Avg time: 10.813 ms
Total time: 137.890 ms - Avg time: 13.789 ms
Query test - partition key 5 and sort keys between 2 and 9
Total time: 13.560 ms - Avg time: 2.712 ms
Total time: 11.339 ms - Avg time: 2.268 ms
Total time: 7.809 ms - Avg time: 1.562 ms
Total time: 10.736 ms - Avg time: 2.147 ms
Total time: 12.122 ms - Avg time: 2.424 ms
Scan test - all items in the table
Total time: 58.952 ms - Avg time: 11.790 ms
Total time: 25.507 ms - Avg time: 5.101 ms
Total time: 37.660 ms - Avg time: 7.532 ms
Total time: 26.781 ms - Avg time: 5.356 ms
Total time: 46.076 ms - Avg time: 9.215 ms

Attempting to delete table; please wait...
Successfully deleted table.
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

8. In the previous step, you ran the program against the DynamoDB endpoint. Now run the program again, but this time, the `GetItem`, `Query`, and `Scan` operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output, as in the following example.

```
{  
    "Port": 8111,  
    "Address": "mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com"  
}
```

Now run the program again, but this time, specify the cluster endpoint as a command line parameter.

```
java TryDax mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

For more information about this program, see the following sections:

- [TryDax.java \(p. 691\)](#)
- [TryDaxHelper.java \(p. 692\)](#)
- [TryDaxTests.java \(p. 695\)](#)

Using the Client as an Apache Maven Dependency

Follow these steps to use the client for the DAX SDK for Java in your application as a dependency.

To use the client as a Maven dependency

1. Download and install Apache Maven. For more information, see [Downloading Apache Maven](#) and [Installing Apache Maven](#).
2. Add the client Maven dependency to your application's Project Object Model (POM) file. In this example, replace `x.x.x.x` with the actual version number of the client (for example, `1.0.200704.0`).

```
<!--Dependency:-->  
<dependencies>  
    <dependency>
```

```
<groupId>com.amazonaws</groupId>
<artifactId>amazon-dax-client</artifactId>
<version>x.x.x.x</version>
</dependency>
</dependencies>
```

TryDax.java

The `TryDax.java` file contains the `main` method. If you run the program with no command line parameters, it creates an Amazon DynamoDB client and uses that client for all API operations. If you specify a DynamoDB Accelerator (DAX) cluster endpoint on the command line, the program also creates a DAX client and uses it for `GetItem`, `Query`, and `Scan` operations.

You can modify the program in several ways:

- Use the DAX client instead of the DynamoDB client. For more information, see [Java and DAX \(p. 688\)](#).
- Choose a different name for the test table.
- Modify the number of items written by changing the `helper.writeData` parameters. The second parameter is the number of partition keys, and the third parameter is the number of sort keys. By default, the program uses 1–10 for partition key values and 1–10 for sort key values, for a total of 100 items written to the table. For more information, see [TryDaxHelper.java \(p. 692\)](#).
- Modify the number of `GetItem`, `Query`, and `Scan` tests, and modify their parameters.
- Comment out the lines containing `helper.createTable` and `helper.deleteTable` (if you don't want to create and delete the table each time you run the program).

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the Client as an Apache Maven Dependency \(p. 690\)](#).

Alternatively, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX \(p. 688\)](#) for an example of setting your `CLASSPATH` variable.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
import com.amazonaws.services.dynamodbv2.document.DynamoDB;

public class TryDax {

    public static void main(String[] args) throws Exception {
        TryDaxHelper helper = new TryDaxHelper();
        TryDaxTests tests = new TryDaxTests();

        DynamoDB ddbClient = helper.getDynamoDBClient();
        DynamoDB daxClient = null;
```

```

        if (args.length >= 1) {
            daxClient = helper.getDaxClient(args[0]);
        }

        String tableName = "TryDaxTable";

        System.out.println("Creating table...");
        helper.createTable(tableName, ddbClient);
        System.out.println("Populating table...");
        helper.writeData(tableName, ddbClient, 10, 10);

        DynamoDB testClient = null;
        if (daxClient != null) {
            testClient = daxClient;
        } else {
            testClient = ddbClient;
        }

        System.out.println("Running GetItem, Scan, and Query tests...");
        System.out.println("First iteration of each test will result in cache misses");
        System.out.println("Next iterations are cache hits\n");

        // GetItem
        tests.getItemTest(tableName, testClient, 1, 10, 5);

        // Query
        tests.queryTest(tableName, testClient, 5, 2, 9, 5);

        // Scan
        tests.scanTest(tableName, testClient, 5);
    }

    helper.deleteTable(tableName, ddbClient);
}
}

```

[TryDaxHelper.java](#)

The `TryDaxHelper.java` file contains utility methods.

The `getDynamoDBClient` and `getDaxClient` methods provide Amazon DynamoDB and DynamoDB Accelerator (DAX) clients. For control plane operations (`CreateTable`, `DeleteTable`) and write operations, the program uses the DynamoDB client. If you specify a DAX cluster endpoint, the main program creates a DAX client for performing read operations (`GetItem`, `Query`, `Scan`).

The other `TryDaxHelper` methods (`createTable`, `writeData`, `deleteTable`) are for setting up and tearing down the DynamoDB table and its data.

You can modify the program in several ways:

- Use different provisioned throughput settings for the table.
- Modify the size of each item written (see the `stringSize` variable in the `writeData` method).
- Modify the number of `GetItem`, `Query`, and `Scan` tests and their parameters.
- Comment out the lines containing `helper.CreateTable` and `helper.DeleteTable` (if you don't want to create and delete the table each time you run the program).

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the Client as an Apache Maven Dependency \(p. 690\)](#).

Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX \(p. 688\)](#) for an example of setting your CLASSPATH variable.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
import java.util.Arrays;

import com.amazonaws.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;
import com.amazonaws.util.EC2MetadataUtils;

public class TryDaxHelper {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDynamoDBClient() {
        System.out.println("Creating a DynamoDB client");
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
            .withRegion(region)
            .build();
        return new DynamoDB(client);
    }

    DynamoDB getDaxClient(String daxEndpoint) {
        System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
        AmazonDynamoDB client = daxClientBuilder.build();
        return new DynamoDB(client);
    }

    void createTable(String tableName, DynamoDB client) {
        Table table = client.getTable(tableName);
        try {
            System.out.println("Attempting to create table; please wait...");

            table = client.createTable(tableName,
                Arrays.asList(
                    new KeySchemaElement("pk", KeyType.HASH), // Partition key
                    new KeySchemaElement("sk", KeyType.RANGE) // Sort key
                )
            );
        } catch (Exception e) {
            System.out.println("Error creating table: " + e.getMessage());
        }
    }
}
```

```

        new KeySchemaElement("sk", KeyType.RANGE)), // Sort key
    Arrays.asList(
        new AttributeDefinition("pk", ScalarAttributeType.N),
        new AttributeDefinition("sk", ScalarAttributeType.N)),
    new ProvisionedThroughput(10L, 10L));
    table.waitForActive();
    System.out.println("Successfully created table. Table status: " +
        table.getDescription().getTableStatus());

} catch (Exception e) {
    System.err.println("Unable to create table: ");
    e.printStackTrace();
}
}

void writeData(String tableName, DynamoDB client, int pkmax, int skmax) {
    Table table = client.getTable(tableName);
    System.out.println("Writing data to the table...");

    int stringSize = 1000;
    StringBuilder sb = new StringBuilder(stringSize);
    for (int i = 0; i < stringSize; i++) {
        sb.append('X');
    }
    String someData = sb.toString();

    try {
        for (Integer ipk = 1; ipk <= pkmax; ipk++) {
            System.out.println("Writing " + skmax + " items for partition key: " +
ipk));
            for (Integer isk = 1; isk <= skmax; isk++) {
                table.putItem(new Item()
                    .withPrimaryKey("pk", ipk, "sk", isk)
                    .withString("someData", someData));
            }
        }
    } catch (Exception e) {
        System.err.println("Unable to write item:");
        e.printStackTrace();
    }
}

void deleteTable(String tableName, DynamoDB client) {
    Table table = client.getTable(tableName);
    try {
        System.out.println("\nAttempting to delete table; please wait...");
        table.delete();
        table.waitForDelete();
        System.out.println("Successfully deleted table.");
    } catch (Exception e) {
        System.err.println("Unable to delete table: ");
        e.printStackTrace();
    }
}
}

```

TryDaxTests.java

The `TryDaxTests.java` file contains methods that perform read operations against a test table in Amazon DynamoDB. These methods are not concerned with how they access the data (using either the DynamoDB client or the DAX client), so there is no need to modify the application logic.

You can modify the program in several ways:

- Modify the `queryTest` method so that it uses a different `KeyConditionExpression`.
- Add a `ScanFilter` to the `scanTest` method so that only some of the items are returned to you.

Note

To run this program, you can set up Maven to use the client for the DAX SDK for Java and the AWS SDK for Java as dependencies. For more information, see [Using the Client as an Apache Maven Dependency \(p. 690\)](#).

Or, you can download and include both the DAX Java client and the AWS SDK for Java in your classpath. See [Java and DAX \(p. 688\)](#) for an example of setting your `CLASSPATH` variable.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
import java.util.HashMap;  
import java.util.Iterator;  
  
import com.amazonaws.services.dynamodbv2.document.DynamoDB;  
import com.amazonaws.services.dynamodbv2.document.Item;  
import com.amazonaws.services.dynamodbv2.document.ItemCollection;  
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;  
import com.amazonaws.services.dynamodbv2.document.ScanOutcome;  
import com.amazonaws.services.dynamodbv2.document.Table;  
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;  
  
public class TryDaxTests {  
  
    void getItemTest(String tableName, DynamoDB client, int pk, int sk, int iterations) {  
        long startTime, endTime;  
        System.out.println("GetItem test - partition key " + pk + " and sort keys 1-" +  
sk);  
        Table table = client.getTable(tableName);  
  
        for (int i = 0; i < iterations; i++) {  
            startTime = System.nanoTime();  
            try {  
                for (Integer ipk = 1; ipk <= pk; ipk++) {  
                    for (Integer isk = 1; isk <= sk; isk++) {  
                        table.getItem("pk", ipk, "sk", isk);  
                    }  
                }  
            } catch (Exception e) {  
                System.err.println("Unable to get item:");  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```

        }
        endTime = System.nanoTime();
        printTime(startTime, endTime, pk * sk);
    }
}

void queryTest(String tableName, DynamoDB client, int pk, int sk1, int sk2, int iterations) {
    long startTime, endTime;
    System.out.println("Query test - partition key " + pk + " and sort keys between " +
sk1 + " and " + sk2);
    Table table = client.getTable(tableName);

    HashMap<String, Object> valueMap = new HashMap<String, Object>();
    valueMap.put(":pkval", pk);
    valueMap.put(":skval1", sk1);
    valueMap.put(":skval2", sk2);

    QuerySpec spec = new QuerySpec()
        .withKeyConditionExpression("pk = :pkval and sk between :skval1
and :skval2")
        .withValueMap(valueMap);

    for (int i = 0; i < iterations; i++) {
        startTime = System.nanoTime();
        ItemCollection<QueryOutcome> items = table.query(spec);

        try {
            Iterator<Item> iter = items.iterator();
            while (iter.hasNext()) {
                iter.next();
            }
        } catch (Exception e) {
            System.err.println("Unable to query table:");
            e.printStackTrace();
        }
        endTime = System.nanoTime();
        printTime(startTime, endTime, iterations);
    }
}

void scanTest(String tableName, DynamoDB client, int iterations) {
    long startTime, endTime;
    System.out.println("Scan test - all items in the table");
    Table table = client.getTable(tableName);

    for (int i = 0; i < iterations; i++) {
        startTime = System.nanoTime();
        ItemCollection<ScanOutcome> items = table.scan();
        try {

            Iterator<Item> iter = items.iterator();
            while (iter.hasNext()) {
                iter.next();
            }
        } catch (Exception e) {
            System.err.println("Unable to scan table:");
            e.printStackTrace();
        }
        endTime = System.nanoTime();
        printTime(startTime, endTime, iterations);
    }
}

public void printTime(long startTime, long endTime, int iterations) {
    System.out.format("\tTotal time: %.3f ms - ", (endTime - startTime) / (1000000.0));
}

```

```
        System.out.format("Avg time: %.3f ms\n", (endTime - startTime) / (iterations *  
1000000.0));  
    }  
}
```

.NET and DAX

Follow these steps to run the .NET sample on your Amazon EC2 instance.

Note

This tutorial uses the .NET Core SDK. It shows how you can run a program in your default Amazon VPC to access your Amazon DynamoDB Accelerator (DAX) cluster. If you prefer, you can use the AWS Toolkit for Visual Studio to write a .NET application and deploy it into your VPC. For more information, see [Creating and Deploying Elastic Beanstalk Applications in .NET Using AWS Toolkit for Visual Studio](#) in the *AWS Elastic Beanstalk Developer Guide*.

To run the .NET sample for DAX

1. Go to the [Microsoft Downloads page](#) and download the latest .NET Core SDK for Linux. The downloaded file is `dotnet-sdk-N.N.N-linux-x64.tar.gz`.
2. Extract the .NET Core files.

```
mkdir dotnet  
tar zxvf dotnet-sdk-N.N.N-linux-x64.tar.gz -C dotnet
```

Replace `N.N.N` with the actual version number of the .NET Core SDK (for example: `2.1.4`).

3. Verify the installation.

```
alias dotnet=$HOME/dotnet/dotnet  
dotnet --version
```

This should print the version number of the .NET Core SDK.

Note

Instead of the version number, you might receive the following error:
`error: libunwind.so.8: cannot open shared object file: No such file or directory`
To resolve the error, install the `libunwind` package.

```
sudo yum install -y libunwind
```

After you do this, you should be able to run the `dotnet --version` command without any errors.

4. Create a new .NET project.

```
dotnet new console -o myApp
```

This requires a few minutes to perform a one-time-only setup. When it is complete, run the sample project.

```
dotnet run --project myApp
```

You should receive the following message: `Hello World!`

5. The `myApp/myApp.csproj` file contains metadata about your project. To use the DAX client in your application, modify the file so that it looks like the following.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
</PropertyGroup>
<ItemGroup>
    <PackageReference Include="AWSSDK.DAX.Client" Version="*" />
</ItemGroup>

</Project>
```

6. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

7. Now run the sample programs, one at a time. For each program, copy its contents into the `myApp/Program.cs`, and then run the `MyApp` project.

Run the following .NET programs. The first program creates a DynamoDB table named `TryDaxTable`. The second program writes data to the table.

```
cp 01-CREATETable.cs myApp/Program.cs
dotnet run --project myApp

cp 02-Write-Data.cs myApp/Program.cs
dotnet run --project myApp
```

8. Next, run some programs to perform `GetItem`, `Query`, and `Scan` operations on your DAX cluster. To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output, as in the following example.

```
{
    "Port": 8111,
    "Address": "mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com"
}
```

Now run the following programs, specifying your cluster endpoint as a command line parameter. (Replace the sample endpoint with your actual DAX cluster endpoint.)

```
cp 03-GetItem-Test.cs myApp/Program.cs
dotnet run --project myApp mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111

cp 04-Query-Test.cs myApp/Program.cs
dotnet run --project myApp mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111

cp 05-Scan-Test.cs myApp/Program.cs
dotnet run --project myApp mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

9. Run the following .NET program to delete `TryDaxTable`.

```
cp 06>DeleteTable.cs myApp/Program.cs
dotnet run --project myApp
```

For more information about these programs, see the following sections:

- [01 CreateTable.cs \(p. 699\)](#)
- [02 Write Data.cs \(p. 700\)](#)
- [03 GetItem-Test.cs \(p. 701\)](#)
- [04 Query-Test.cs \(p. 702\)](#)
- [05 Scan-Test.cs \(p. 704\)](#)
- [06 DeleteTable.cs \(p. 705\)](#)

01-CreateTable.cs

The `01-CreateTable.cs` program creates a table (`TryDaxTable`). The remaining .NET programs in this section depend on this table.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using Amazon.DynamoDBv2.Model;
using System.Collections.Generic;
using System;
using Amazon.DynamoDBv2;

namespace ClientTest
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

        AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        var tableName = "TryDaxTable";

        var request = new CreateTableRequest()
        {
            TableName = tableName,
            KeySchema = new List<KeySchemaElement>()
            {
                new KeySchemaElement{ AttributeName = "pk",KeyType = "HASH" },
                new KeySchemaElement{ AttributeName = "sk",KeyType = "RANGE" }
            },
            AttributeDefinitions = new List<AttributeDefinition>()
            {
                new AttributeDefinition{ AttributeName = "pk",AttributeType = "N" },
                new AttributeDefinition{ AttributeName = "sk",AttributeType = "N" }
            },
            ProvisionedThroughput = new ProvisionedThroughput()
            {
                ReadCapacityUnits = 10,
                WriteCapacityUnits = 10
            }
        };

        var response = client.CreateTableAsync(request).Result;

        Console.WriteLine("Hit <enter> to continue...");
        Console.ReadLine();
    }
}
}

```

02-Write-Data.cs

The 02-Write-Data.cs program writes test data to TryDaxTable.

```


/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
using Amazon.DynamoDBv2.Model;
using System.Collections.Generic;
using System;
using Amazon.DynamoDBv2;

namespace ClientTest
{
    class Program
    {
        static void Main(string[] args)
        {

            AmazonDynamoDBClient client = new AmazonDynamoDBClient();


```

```
var tableName = "TryDaxTable";

string someData = new String('X', 1000);
var pkmax = 10;
var skmax = 10;

for (var ipk = 1; ipk <= pkmax; ipk++)
{
    Console.WriteLine("Writing " + skmax + " items for partition key: " + ipk);
    for (var isk = 1; isk <= skmax; isk++)
    {
        var request = new PutItemRequest()
        {
            TableName = tableName,
            Item = new Dictionary<string, AttributeValue>()
            {
                { "pk", new AttributeValue{N = ipk.ToString()} },
                { "sk", new AttributeValue{N = isk.ToString()} },
                { "someData", new AttributeValue{S = someData} }
            }
        };
        var response = client.PutItemAsync(request).Result;
    }
}

Console.WriteLine("Hit <enter> to continue...");
Console.ReadLine();
}
```

03-GetItem-Test.cs

The 03-GetItem-Test.cs program performs GetItem operations on TryDaxTable.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
using Amazon.Runtime;  
using Amazon.DAX;  
using Amazon.DynamoDBv2.Model;  
using System.Collections.Generic;  
using System;  
using Amazon.DynamoDBv2;  
using Amazon;  
  
namespace ClientTest  
{  
    class Program  
    {
```

```
static void Main(string[] args)
{
    String hostName = args[0].Split(':')[0];
    int port = Int32.Parse(args[0].Split(':')[1]);
    Console.WriteLine("Using DAX client - hostname=" + hostName + ", port=" +
port);

    var clientConfig = new DaxClientConfig(hostName, port)
    {
        AwsCredentials = FallbackCredentialsFactory.GetCredentials()

    };
    var client = new ClusterDaxClient(clientConfig);

    var tableName = "TryDaxTable";

    var pk = 1;
    var sk = 10;
    var iterations = 5;

    var startTime = DateTime.Now;

    for (var i = 0; i < iterations; i++)
    {
        for (var ipk = 1; ipk <= pk; ipk++)
        {
            for (var isk = 1; isk <= sk; isk++)
            {
                var request = new GetItemRequest()
                {
                    TableName = tableName,
                    Key = new Dictionary<string, AttributeValue>() {
                        {"pk", new AttributeValue {N = ipk.ToString()} },
                        {"sk", new AttributeValue {N = isk.ToString()} }
                    }
                };
                var response = client.GetItemAsync(request).Result;
                Console.WriteLine("GetItem succeeded for pk: " + ipk + ", sk: " +
isk);
            }
        }
    }

    var endTime = DateTime.Now;
    TimeSpan timeSpan = endTime - startTime;
    Console.WriteLine("Total time: " + (int)timeSpan.TotalMilliseconds + " milliseconds");

    Console.WriteLine("Hit <enter> to continue...");
    Console.ReadLine();
}
}
```

04-Query-Test.cs

The 04-Query-Test.cs program performs Query operations on TryDaxTable.

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;
using System.Collections.Generic;
using System;
using Amazon.DynamoDBv2;
using Amazon;

namespace ClientTest
{
    class Program
    {
        static void Main(string[] args)
        {

            String hostName = args[0].Split(':')[0];
            int port = Int32.Parse(args[0].Split(':')[1]);
            Console.WriteLine("Using DAX client - hostname=" + hostName + ", port=" +
port);

            var clientConfig = new DaxClientConfig(hostName, port)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials()

            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var pk = 5;
            var sk1 = 2;
            var sk2 = 9;
            var iterations = 5;

            var startTime = DateTime.Now;

            for (var i = 0; i < iterations; i++)
            {
                var request = new QueryRequest()
                {
                    TableName = tableName,
                    KeyConditionExpression = "pk = :pkval and sk between :skval1
and :skval2",
                    ExpressionAttributeValues = new Dictionary<string, AttributeValue>() {
                        {"":pkval", new AttributeValue {N = pk.ToString()} },
                        {"":skval1", new AttributeValue {N = sk1.ToString()} },
                        {"":skval2", new AttributeValue {N = sk2.ToString()} }
                    }
                };
                var response = client.QueryAsync(request).Result;
                Console.WriteLine(i + ": Query succeeded");
            }
        }
    }
}

```

```
        }

        var endTime = DateTime.Now;
        TimeSpan timeSpan = endTime - startTime;
        Console.WriteLine("Total time: " + (int)timeSpan.TotalMilliseconds + " milliseconds");

        Console.WriteLine("Hit <enter> to continue...");
        Console.ReadLine();
    }
}
```

05-Scan-Test.cs

The 05-Scan-Test.cs program performs Scan operations on TryDaxTable.

```
/***
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using Amazon.Runtime;
using Amazon.DAX;
using Amazon.DynamoDBv2.Model;
using System.Collections.Generic;
using System;
using Amazon.DynamoDBv2;
using Amazon;

namespace ClientTest
{
    class Program
    {
        static void Main(string[] args)
        {

            String hostName = args[0].Split(':')[0];
            int port = Int32.Parse(args[0].Split(':')[1]);
            Console.WriteLine("Using DAX client - hostname=" + hostName + ", port=" +
port);

            var clientConfig = new DaxClientConfig(hostName, port)
            {
                AwsCredentials = FallbackCredentialsFactory.GetCredentials(
                );
            };
            var client = new ClusterDaxClient(clientConfig);

            var tableName = "TryDaxTable";

            var iterations = 5;
```

```
var startTime = DateTime.Now;

for (var i = 0; i < iterations; i++)
{
    var request = new ScanRequest()
    {
        TableName = tableName
    };
    var response = client.ScanAsync(request).Result;
    Console.WriteLine(i + ": Scan succeeded");
}

var endTime = DateTime.Now;
TimeSpan timeSpan = endTime - startTime;
Console.WriteLine("Total time: " + (int)timeSpan.TotalMilliseconds + " milliseconds");

Console.WriteLine("Hit <enter> to continue...");
Console.ReadLine();
}
}
```

06-DeleteTable.cs

The 06-DeleteTable.cs program deletes TryDaxTable. Run this program after you have finished testing.

```
/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
using Amazon.DynamoDBv2.Model;
using System;
using Amazon.DynamoDBv2;

namespace ClientTest
{
    class Program
    {
        static void Main(string[] args)
        {

            AmazonDynamoDBClient client = new AmazonDynamoDBClient();

            var tableName = "TryDaxTable";

            var request = new DeleteTableRequest()
            {
                TableName = tableName
            };

            var response = client.DeleteTableAsync(request).Result;
```

```
        Console.WriteLine("Hit <enter> to continue...");  
        Console.ReadLine();  
    }  
}  
}
```

Node.js and DAX

Follow these steps to run the Node.js sample application on your Amazon EC2 instance.

To run the Node.js sample for DAX

1. Set up Node.js on your Amazon EC2 instance, as follows:

- a. Install node version manager (nvm).

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

- b. Use nvm to install Node.js.

```
nvm install 12.16.3
```

- c. Test that Node.js is installed and running correctly.

```
node -e "console.log('Running Node.js ' + process.version)"
```

This should display the following message.

```
Running Node.js v12.16.3
```

2. Install the DAX Node.js client using the node package manager (npm).

```
npm install amazon-dax-client
```

3. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

4. Run the following Node.js programs. The first program creates an Amazon DynamoDB table named TryDaxTable. The second program writes data to the table.

```
node 01-create-table.js  
node 02-write-data.js
```

5. Run the following Node.js programs.

```
node 03-getitem-test.js
```

```
node 04-query-test.js
node 05-scan-test.js
```

Take note of the timing information—the number of milliseconds required for the `GetItem`, `Query`, and `Scan` tests.

6. In the previous step, you ran the programs against the DynamoDB endpoint. Run the programs again, but this time, the `GetItem`, `Query` and `Scan` operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following.

- **Using the DynamoDB console**—Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

- **Using the AWS CLI**—Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output, as in the following example.

```
{
    "Port": 8111,
    "Address": "mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com"
}
```

Now run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
node 03-getitem-test.js mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
node 04-query-test.js mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
node 05-scan-test.js mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

7. Run the following Node.js program to delete `TryDaxTable`.

```
node 06-delete-table
```

For more information about these programs, see the following sections:

- [01-create-table.js \(p. 708\)](#)
- [02-write-data.js \(p. 708\)](#)
- [03-getitem-test.js \(p. 709\)](#)
- [04-query-test.js \(p. 711\)](#)
- [05-scan-test.js \(p. 712\)](#)
- [06-delete-table.js \(p. 713\)](#)

01-create-table.js

The 01-create-table.js program creates a table (`TryDaxTable`). The remaining Node.js programs in this section depend on this table.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
const AmazonDaxClient = require('amazon-dax-client');  
var AWS = require("aws-sdk");  
  
var region = "us-west-2";  
  
AWS.config.update({  
    region: region  
});  
  
var dynamodb = new AWS.DynamoDB() //low-level client  
  
var tableName = "TryDaxTable";  
  
var params = {  
    TableName : tableName,  
    KeySchema: [  
        { AttributeName: "pk", KeyType: "HASH"},   //Partition key  
        { AttributeName: "sk", KeyType: "RANGE" }    //Sort key  
    ],  
    AttributeDefinitions: [  
        { AttributeName: "pk", AttributeType: "N" },  
        { AttributeName: "sk", AttributeType: "N" }  
    ],  
    ProvisionedThroughput: {  
        ReadCapacityUnits: 10,  
        WriteCapacityUnits: 10  
    }  
};  
  
dynamodb.createTable(params, function(err, data) {  
    if (err) {  
        console.error("Unable to create table. Error JSON:", JSON.stringify(err, null, 2));  
    } else {  
        console.log("Created table. Table description JSON:", JSON.stringify(data, null,  
2));  
    }  
});
```

02-write-data.js

The 02-write-data.js program writes test data to `TryDaxTable`.

```
/**
```

```

/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
const AmazonDaxClient = require('amazon-dax-client');
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
    region: region
});

var ddbClient = new AWS.DynamoDB.DocumentClient()

var tableName = "TryDaxTable";

var someData = "X".repeat(1000);
var pkmax = 10;
var skmax = 10;

for (var ipk = 1; ipk <= pkmax; ipk++)  {

    for (var isk = 1; isk <= skmax; isk++) {
        var params = {
            TableName: tableName,
            Item: {
                "pk": ipk,
                "sk": isk,
                "someData": someData
            }
        };

        // 
        //put item

        ddbClient.put(params, function(err, data) {
            if (err) {
                console.error("Unable to write data: ", JSON.stringify(err, null, 2));
            } else {
                console.log("PutItem succeeded");
            }
        });
    }
}

```

03-getitem-test.js

The 03-getitem-test.js program performs GetItem operations on TryDaxTable.

```

/** 
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.

```

```

/*
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
const AmazonDaxClient = require('amazon-dax-client');
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
    region: region
});

var ddbClient = new AWS.DynamoDB.DocumentClient()
var daxClient = null;

if (process.argv.length > 2) {
    var dax = new AmazonDaxClient({endpoints: [process.argv[2]], region: region})
    daxClient = new AWS.DynamoDB.DocumentClient({service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var pk = 1;
var sk = 10;
var iterations = 5;

for (var i = 0; i < iterations; i++) {

    var startTime = new Date().getTime();

    for (var ipk = 1; ipk <= pk; ipk++) {
        for (var isk = 1; isk <= sk; isk++) {

            var params = {
                TableName: tableName,
                Key:{ 
                    "pk": ipk,
                    "sk": isk
                }
            };

            client.get(params, function(err, data) {
                if (err) {
                    console.error("Unable to read item. Error JSON:", JSON.stringify(err, null, 2));
                } else {
                    // GetItem succeeded
                }
            });
        }
    }

    var endTime = new Date().getTime();
    console.log("\tTotal time: ", (endTime - startTime), "ms - Avg time: ", (endTime - startTime) / iterations, "ms");
}

```

04-query-test.js

The 04-query-test.js program performs Query operations on TryDaxTable.

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 *  
 * This file is licensed under the Apache License, Version 2.0 (the "License").  
 * You may not use this file except in compliance with the License. A copy of  
 * the License is located at  
 *  
 * http://aws.amazon.com/apache2.0/  
 *  
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the  
 * specific language governing permissions and limitations under the License.  
 */  
const AmazonDaxClient = require('amazon-dax-client');  
var AWS = require("aws-sdk");  
  
var region = "us-west-2";  
  
AWS.config.update({  
    region: region  
});  
  
var ddbClient = new AWS.DynamoDB.DocumentClient()  
var daxClient = null;  
  
if (process.argv.length > 2) {  
    var dax = new AmazonDaxClient({endpoints: [process.argv[2]], region: region})  
    daxClient = new AWS.DynamoDB.DocumentClient({service: dax });  
}  
  
var client = daxClient != null ? daxClient : ddbClient;  
var tableName = "TryDaxTable";  
  
var pk = 5;  
var sk1 = 2;  
var sk2 = 9;  
var iterations = 5;  
  
var params = {  
    TableName: tableName,  
    KeyConditionExpression: "pk = :pkval and sk between :skval1 and :skval2",  
    ExpressionAttributeValues: {  
        ":pkval":pk,  
        ":skval1":sk1,  
        ":skval2":sk2  
    }  
};  
  
for (var i = 0; i < iterations; i++) {  
    var startTime = new Date().getTime();  
  
    client.query(params, function(err, data) {  
        if (err) {  
            console.error("Unable to read item. Error JSON:", JSON.stringify(err, null,  
2));  
        } else {  
            // Query succeeded  
    
```

```

        }
    });

    var endTime = new Date().getTime();
    console.log("\tTotal time: ", (endTime - startTime) , "ms - Avg time: ", (endTime - startTime) / iterations, "ms");

}

```

05-scan-test.js

The 05-scan-test.js program performs Scan operations on TryDaxTable.

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
*/
const AmazonDaxClient = require('amazon-dax-client');
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
    region: region
});

var ddbClient = new AWS.DynamoDB.DocumentClient()
var daxClient = null;

if (process.argv.length > 2) {
    var dax = new AmazonDaxClient({endpoints: [process.argv[2]], region: region})
    daxClient = new AWS.DynamoDB.DocumentClient({service: dax });
}

var client = daxClient != null ? daxClient : ddbClient;
var tableName = "TryDaxTable";

var iterations = 5;

var params = {
    TableName: tableName
};
var startTime = new Date().getTime();
for (var i = 0; i < iterations; i++) {

    client.scan(params, function(err, data) {
        if (err) {
            console.error("Unable to read item. Error JSON:", JSON.stringify(err, null, 2));
        } else {
            // Scan succeeded
        }
    });
}

```

```
}

var endTime = new Date().getTime();
console.log("\tTotal time: ", (endTime - startTime) , "ms - Avg time: ", (endTime - startTime) / iterations, "ms");
```

06-delete-table.js

The 06-delete-table.js program deletes TryDaxTable. Run this program after you have finished testing.

```
/** 
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
const AmazonDaxClient = require('amazon-dax-client');
var AWS = require("aws-sdk");

var region = "us-west-2";

AWS.config.update({
  region: region
});

var dynamodb = new AWS.DynamoDB(); //low-level client

var tableName = "TryDaxTable";

var params = {
  TableName : tableName
};

dynamodb.deleteTable(params, function(err, data) {
  if (err) {
    console.error("Unable to delete table. Error JSON:", JSON.stringify(err, null, 2));
  } else {
    console.log("Deleted table. Table description JSON:", JSON.stringify(data, null, 2));
  }
});
```

Python and DAX

Follow this procedure to run the Python sample application on your Amazon EC2 instance.

To run the Python sample for DAX

1. Install the DAX Python client using the pip utility.

```
pip install amazon-dax-client
```

2. Download the sample program source code (.zip file).

```
wget http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/TryDax.zip
```

When the download is complete, extract the source files.

```
unzip TryDax.zip
```

3. Run the following Python programs. The first program creates an Amazon DynamoDB table named TryDaxTable. The second program writes data to the table.

```
python 01-create-table.py  
python 02-write-data.py
```

4. Run the following Python programs.

```
python 03-getitem-test.py  
python 04-query-test.py  
python 05-scan-test.py
```

Take note of the timing information—the number of milliseconds required for the GetItem, Query, and Scan tests.

5. In the previous step, you ran the programs against the DynamoDB endpoint. Now run the programs again, but this time, the GetItem, Query, and Scan operations are processed by your DAX cluster.

To determine the endpoint for your DAX cluster, choose one of the following:

- **Using the DynamoDB console** — Choose your DAX cluster. The cluster endpoint is shown on the console, as in the following example.

```
mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

- **Using the AWS CLI** — Enter the following command.

```
aws dax describe-clusters --query "Clusters[*].ClusterDiscoveryEndpoint"
```

The cluster endpoint port and address are shown in the output, as in this example.

```
{  
    "Port": 8111,  
    "Address": "mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com"  
}
```

Run the programs again, but this time, specify the cluster endpoint as a command line parameter.

```
python 03-getitem-test.py mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

```
python 04-query-test.py mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
python 05-scan-test.py mycluster.frfx8h.clustercfg.dax.usw2.amazonaws.com:8111
```

Look at the rest of the output, and take note of the timing information. The elapsed times for `GetItem`, `Query`, and `Scan` should be significantly lower with DAX than with DynamoDB.

6. Run the following Python program to delete `TryDaxTable`.

```
python 06-delete-table.py
```

For more information about these programs, see the following sections:

- [01-create-table.py \(p. 715\)](#)
- [02-write-data.py \(p. 716\)](#)
- [03-getitem-test.py \(p. 716\)](#)
- [04-query-test.py \(p. 717\)](#)
- [05-scan-test.py \(p. 718\)](#)
- [06-delete-table.py \(p. 719\)](#)

01-create-table.py

The `01-create-table.py` program creates a table (`TryDaxTable`). The remaining Python programs in this section depend on this table.

```
import boto3

def create_dax_table(dyn_resource=None):
    """
    Creates a DynamoDB table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    :return: The newly created table.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource('dynamodb')

    table_name = 'TryDaxTable'
    params = {
        'TableName': table_name,
        'KeySchema': [
            {'AttributeName': 'partition_key', 'KeyType': 'HASH'},
            {'AttributeName': 'sort_key', 'KeyType': 'RANGE'}
        ],
        'AttributeDefinitions': [
            {'AttributeName': 'partition_key', 'AttributeType': 'N'},
            {'AttributeName': 'sort_key', 'AttributeType': 'N'}
        ],
        'ProvisionedThroughput': {
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    }
    table = dyn_resource.create_table(**params)
    print(f"Creating {table_name}...")
    table.wait_until_exists()
    return table
```

```
if __name__ == '__main__':
    dax_table = create_dax_table()
    print(f"Created table.")
```

02-write-data.py

The 02-write-data.py program writes test data to TryDaxTable.

```
import boto3

def write_data_to_dax_table(key_count, item_size, dyn_resource=None):
    """
    Writes test data to the demonstration table.

    :param key_count: The number of partition and sort keys to use to populate the
                      table. The total number of items is key_count * key_count.
    :param item_size: The size of non-key data for each test item.
    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource('dynamodb')

    table = dyn_resource.Table('TryDaxTable')
    some_data = 'X' * item_size

    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.put_item(Item={
                'partition_key': partition_key,
                'sort_key': sort_key,
                'some_data': some_data
            })
            print(f"Put item ({partition_key}, {sort_key}) succeeded.")

if __name__ == '__main__':
    write_key_count = 10
    write_item_size = 1000
    print(f"Writing {write_key_count}*{write_key_count} items to the table. "
          f"Each item is {write_item_size} characters.")
    write_data_to_dax_table(write_key_count, write_item_size)
```

03-getitem-test.py

The 03-getitem-test.py program performs GetItem operations on TryDaxTable.

```
import argparse
import sys
import time
import amazondax
import boto3

def get_item_test(key_count, iterations, dyn_resource=None):
    """
    Gets items from the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

    :param key_count: The number of items to get from the table in each iteration.
    """
    pass
```

```

:param iterations: The number of iterations to run.
:param dyn_resource: Either a Boto3 or DAX resource.
:return: The start and end times of the test.
"""
if dyn_resource is None:
    dyn_resource = boto3.resource('dynamodb')

table = dyn_resource.Table('TryDaxTable')
start = time.perf_counter()
for _ in range(iterations):
    for partition_key in range(1, key_count + 1):
        for sort_key in range(1, key_count + 1):
            table.get_item(Key={
                'partition_key': partition_key,
                'sort_key': sort_key
            })
            print('.', end='')
            sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
    args = parser.parse_args()

    test_key_count = 10
    test_iterations = 50
    if args.endpoint_url:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazonadax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = get_item_test(
                test_key_count, test_iterations, dyn_resource=dax)
    else:
        print(f"Getting each item from the table {test_iterations} times, "
              f"using the Boto3 client.")
        test_start, test_end = get_item_test(
            test_key_count, test_iterations)
    print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
          f"{{(test_end - test_start)/ test_iterations}}")

```

04-query-test.py

The 04-query-test.py program performs Query operations on TryDaxTable.

```

import argparse
import time
import sys
import amazonadax
import boto3
from boto3.dynamodb.conditions import Key

def query_test(partition_key, sort_keys, iterations, dyn_resource=None):
    """
    Queries the table a specified number of times. The time before the
    first iteration and the time after the last iteration are both captured
    and reported.

```

```

:param partition_key: The partition key value to use in the query. The query
    returns items that have partition keys equal to this value.
:param sort_keys: The range of sort key values for the query. The query returns
    items that have sort key values between these two values.
:param iterations: The number of iterations to run.
:param dyn_resource: Either a Boto3 or DAX resource.
:return: The start and end times of the test.
"""
if dyn_resource is None:
    dyn_resource = boto3.resource('dynamodb')

table = dyn_resource.Table('TryDaxTable')
key_condition_expression = \
    Key('partition_key').eq(partition_key) & \
    Key('sort_key').between(*sort_keys)

start = time.perf_counter()
for _ in range(iterations):
    table.query(KeyConditionExpression=key_condition_expression)
    print('.', end='')
    sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
    args = parser.parse_args()

    test_partition_key = 5
    test_sort_keys = (2, 9)
    test_iterations = 100
    if args.endpoint_url:
        print(f"Querying the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazonadax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = query_test(
                test_partition_key, test_sort_keys, test_iterations, dyn_resource=dax)
    else:
        print(f"Querying the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = query_test(
            test_partition_key, test_sort_keys, test_iterations)

    print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
          f"{{(test_end - test_start)/test_iterations}}")

```

05-scan-test.py

The 05-scan-test.py program performs Scan operations on TryDaxTable.

```

import argparse
import time
import sys
import amazonadax
import boto3

def scan_test(iterations, dyn_resource=None):
    """

```

```

Scans the table a specified number of times. The time before the
first iteration and the time after the last iteration are both captured
and reported.

:param iterations: The number of iterations to run.
:param dyn_resource: Either a Boto3 or DAX resource.
:return: The start and end times of the test.
"""
if dyn_resource is None:
    dyn_resource = boto3.resource('dynamodb')

table = dyn_resource.Table('TryDaxTable')
start = time.perf_counter()
for _ in range(iterations):
    table.scan()
    print('.', end='')
    sys.stdout.flush()
print()
end = time.perf_counter()
return start, end

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        'endpoint_url', nargs='?',
        help="When specified, the DAX cluster endpoint. Otherwise, DAX is not used.")
    args = parser.parse_args()

    test_iterations = 100
    if args.endpoint_url:
        print(f"Scanning the table {test_iterations} times, using the DAX client.")
        # Use a with statement so the DAX client closes the cluster after completion.
        with amazondax.AmazonDaxClient.resource(endpoint_url=args.endpoint_url) as dax:
            test_start, test_end = scan_test(test_iterations, dyn_resource=dax)
    else:
        print(f"Scanning the table {test_iterations} times, using the Boto3 client.")
        test_start, test_end = scan_test(test_iterations)
    print(f"Total time: {test_end - test_start:.4f} sec. Average time: "
          f"{(test_end - test_start)/test_iterations}.")

```

06-delete-table.py

The 06-delete-table.py program deletes TryDaxTable. Run this program after you have finished testing Amazon DynamoDB Accelerator (DAX) functionality.

```

import boto3

def delete_dax_table(dyn_resource=None):
    """
    Deletes the demonstration table.

    :param dyn_resource: Either a Boto3 or DAX resource.
    """
    if dyn_resource is None:
        dyn_resource = boto3.resource('dynamodb')

    table = dyn_resource.Table('TryDaxTable')
    table.delete()

    print(f"Deleting {table.name}...")
    table.wait_until_not_exists()

```

```
if __name__ == '__main__':
    delete_dax_table()
    print("Table deleted!")
```

Modifying an Existing Application to Use DAX

If you already have a Java application that uses Amazon DynamoDB, you have to modify it so that it can access your DynamoDB Accelerator (DAX) cluster. You don't have to rewrite the entire application because the DAX Java client is similar to the DynamoDB low-level client included in the AWS SDK for Java.

Suppose that you have a DynamoDB table named `Music`. The partition key for the table is `Artist`, and its sort key is `SongTitle`. The following program reads an item directly from the `Music` table.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {

    public static void main(String[] args) throws Exception {

        // Create a DynamoDB client
        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        try {
            System.out.println("Attempting to read the item...");
            GetItemResult result = client.getItem(request);
            System.out.println("GetItem succeeded: " + result);

        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }
    }
}
```

To modify the program, replace the DynamoDB client with a DAX client.

```
import java.util.HashMap;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class GetMusicItem {
```

```

public static void main(String[] args) throws Exception {
    //Create a DAX client

    AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
    daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111");
    AmazonDynamoDB client = daxClientBuilder.build();

    /*
    ** ...
    ** Remaining code omitted (it is identical)
    ** ...
    */

}
}

```

Using the DynamoDB Document API

The AWS SDK for Java provides a document interface for DynamoDB. The document API acts as a wrapper around the low-level DynamoDB client. For more information, see [Document Interfaces](#).

The document interface can also be used with the low-level DAX client, as shown in the following example.

```

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazon.dax.client.dynamodbv2.AmazonDaxClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.GetItemOutcome;
import com.amazonaws.services.dynamodbv2.document.Table;

public class GetMusicItemWithDocumentApi {

    public static void main(String[] args) throws Exception {
        //Create a DAX client

        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();
        daxClientBuilder.withRegion("us-
east-1").withEndpointConfiguration("mydaxcluster.2cmrw1.clustercfg.dax.use1.cache.amazonaws.com:8111");
        AmazonDynamoDB client = daxClientBuilder.build();

        // Document client wrapper
        DynamoDB docClient = new DynamoDB(client);

        Table table = docClient.getTable("Music");

        try {
            System.out.println("Attempting to read the item...");
            GetItemOutcome outcome = table.tgetItemOutcome(
                "Artist", "No One You Know",
                "SongTitle", "Scared of My Shadow");
            System.out.println(outcome.getItem());
            System.out.println("GetItem succeeded: " + outcome);
        } catch (Exception e) {
            System.err.println("Unable to read item");
            System.err.println(e.getMessage());
        }
    }
}

```

DAX Async Client

The `AmazonDaxClient` is synchronous. For a long-running DAX API operation, such as a Scan of a large table, this can block program execution until the operation is complete. If your program needs to perform other work while a DAX API operation is in progress, you can use `ClusterDaxAsyncClient` instead.

The following program shows how to use `ClusterDaxAsyncClient`, along with Java `Future`, to implement a non-blocking solution.

```
/*
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */
import java.util.HashMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

import com.amazonaws.dax.client.dynamodbv2.ClientConfig;
import com.amazonaws.dax.client.dynamodbv2.ClusterDaxAsyncClient;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.handlers.AsyncHandler;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBAsync;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.dynamodbv2.model.GetItemRequest;
import com.amazonaws.services.dynamodbv2.model.GetItemResult;

public class DaxAsyncClientDemo {
    public static void main(String[] args) throws Exception {

        ClientConfig daxConfig = new ClientConfig().withCredentialsProvider(new
        ProfileCredentialsProvider())
            .withEndpoints("mydaxcluster.2cmrw1.clustercfg.dax.us-east-1.cache.amazonaws.com:8111");

        AmazonDynamoDBAsync client = new ClusterDaxAsyncClient(daxConfig);

        HashMap<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("Artist", new AttributeValue().withS("No One You Know"));
        key.put("SongTitle", new AttributeValue().withS("Scared of My Shadow"));

        GetItemRequest request = new GetItemRequest()
            .withTableName("Music").withKey(key);

        // Java Futures
        Future<GetItemResult> call = client.getItemAsync(request);
        while (!call.isDone()) {
            // Do other processing while you're waiting for the response
            System.out.println("Doing something else for a few seconds...");
            Thread.sleep(3000);
        }
        // The results should be ready by now

        try {
            call.get();
        }
```

```
        } catch (ExecutionException ee) {
            // Futures always wrap errors as an ExecutionException.
            // The *real* exception is stored as the cause of the
            // ExecutionException
            Throwable exception = ee.getCause();
            System.out.println("Error getting item: " + exception.getMessage());
        }

        // Async callbacks
        call = client.getItemAsync(request, new AsyncHandler<GetItemRequest, GetItemResult>() {

            @Override
            public void onSuccess(GetItemRequest request, GetItemResult getItemResult) {
                System.out.println("Result: " + getItemResult);
            }

            @Override
            public void onError(Exception e) {
                System.out.println("Unable to read item");
                System.err.println(e.getMessage());
                // Callers can also test if exception is an instance of
                // AmazonServiceException or AmazonClientException and cast
                // it to get additional information
            }
        });
        call.get();
    }
}
```

Querying Global Secondary Indexes

You can use Amazon DynamoDB Accelerator (DAX) to query [global secondary indexes](#) using DynamoDB programmatic interfaces.

The following example demonstrates how to use DAX to query the `CreateDateIndex` global secondary index that is created in [Example: Global Secondary Indexes Using the AWS SDK for Java Document API](#).

The `DAXClient` class instantiates the client objects that are needed to interact with the DynamoDB programming interfaces.

```
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.util.EC2MetadataUtils;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;

public class DaxClient {

    private static final String region = EC2MetadataUtils.getEC2InstanceRegion();

    DynamoDB getDaxDocClient(String daxEndpoint) {
        System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
        AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

        daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
        AmazonDynamoDB client = daxClientBuilder.build();

        return new DynamoDB(client);
    }
}
```

```

DynamoDBMapper getDaxMapperClient(String daxEndpoint) {
    System.out.println("Creating a DAX client with cluster endpoint " + daxEndpoint);
    AmazonDaxClientBuilder daxClientBuilder = AmazonDaxClientBuilder.standard();

    daxClientBuilder.withRegion(region).withEndpointConfiguration(daxEndpoint);
    AmazonDynamoDB client = daxClientBuilder.build();

    return new DynamoDBMapper(client);
}
}

```

You can query a global secondary index in the following ways:

- Use the `queryIndex` method on the `QueryIndexDax` class defined in the following example. The `QueryIndexDax` takes as a parameter the client object that is returned by the `getDaxDocClient` method on the `DaxClient` class.
- If you are using the [object persistence interface](#), use the `queryIndexMapper` method on the `QueryIndexDax` class defined in the following example. The `queryIndexMapper` takes as a parameter the client object that is returned by the `getDaxMapperClient` method defined on the `DaxClient` class.

```

import java.util.Iterator;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import java.util.List;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBQueryExpression;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import java.util.HashMap;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.utils.ValueMap;
import com.amazonaws.services.dynamodbv2.document.spec.QuerySpec;
import com.amazonaws.services.dynamodbv2.document.QueryOutcome;
import com.amazonaws.services.dynamodbv2.document.ItemCollection;
import com.amazonaws.services.dynamodbv2.document.Index;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;

public class QueryIndexDax {

    //This is used to query Index using the low-level interface.
    public static void queryIndex(DynamoDB client, String tableName, String indexName) {
        Table table = client.getTable(tableName);

        System.out.println("\n*****\n");
        System.out.print("Querying index " + indexName + "...");

        Index index = table.getIndex(indexName);

        ItemCollection<QueryOutcome> items = null;

        QuerySpec querySpec = new QuerySpec();

        if (indexName == "CreateDateIndex") {
            System.out.println("Issues filed on 2013-11-01");
            querySpec.withKeyConditionExpression("CreateDate = :v_date and
begins_with(IssueId, :v_issue)")
                .WithValueMap(new ValueMap().withString(":v_date",
"2013-11-01").withString(":v_issue", "A-"));
            items = index.query(querySpec);
        } else {
            System.out.println("\nNo valid index name provided");
        }
    }
}

```

```

    }

    Iterator<Item> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        System.out.println(iterator.next().toJSONPretty());
    }

}

//This is used to query Index using the high-level mapper interface.
public static void queryIndexMapper(DynamoDBMapper mapper, String tableName, String
indexName) {
    HashMap<String, AttributeValue> eav = new HashMap<String, AttributeValue>();
    eav.put(":v_date", new AttributeValue().withS("2013-11-01"));
    eav.put(":v_issue", new AttributeValue().withS("A-"));
    DynamoDBQueryExpression<CreateDate> queryExpression = new
DynamoDBQueryExpression<CreateDate>()
        .withIndexName("CreateDateIndex").withConsistentRead(false)
        .withKeyConditionExpression("CreateDate = :v_date and begins_with(IssueId, :v_issue)")
        .withExpressionAttributeValues(eav);

    List<CreateDate> items = mapper.query(CreateDate.class, queryExpression);
    Iterator<CreateDate> iterator = items.iterator();

    System.out.println("Query: printing results...");

    while (iterator.hasNext()) {
        CreateDate iterObj = iterator.next();
        System.out.println(iterObj.getCreateDate());
        System.out.println(iterObj.getIssueId());
    }
}
}

```

The class definition below represents the Issues table and is used in the `queryIndexMapper` method.

```

import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;

@DynamoDBTable(tableName = "Issues")
public class CreateDate {
    private String createDate;
    @DynamoDBHashKey(attributeName = "IssueId")
    private String issueId;

    @DynamoDBIndexHashKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
"CreateDate")
    public String getCreateDate() {
        return createDate;
    }

    public void setCreateDate(String createDate) {
        this.createDate = createDate;
    }

    @DynamoDBIndexRangeKey(globalSecondaryIndexName = "CreateDateIndex", attributeName =
"IssueId")
    public String getIssueId() {
        return issueId;
    }
}

```

```
public void setIssueId(String issueId) {  
    this.issueId = issueId;  
}  
}
```

Managing DAX Clusters

This section addresses some of the common management tasks for Amazon DynamoDB Accelerator (DAX) clusters.

Topics

- [IAM Permissions for Managing a DAX Cluster \(p. 726\)](#)
- [Scaling a DAX Cluster \(p. 728\)](#)
- [Customizing DAX Cluster Settings \(p. 729\)](#)
- [Configuring TTL Settings \(p. 729\)](#)
- [Tagging Support for DAX \(p. 730\)](#)
- [AWS CloudTrail Integration \(p. 731\)](#)
- [Deleting a DAX Cluster \(p. 731\)](#)

IAM Permissions for Managing a DAX Cluster

When you administer a DAX cluster using the AWS Management Console or the AWS Command Line Interface (AWS CLI), we strongly recommend that you narrow the scope of actions that users can perform. By doing so, you help mitigate risk while following the principle of least privilege.

The following discussion focuses on access control for the DAX management APIs. For more information, see [Amazon DynamoDB Accelerator](#) in the *Amazon DynamoDB API Reference*.

Note

For more detailed information about managing AWS Identity and Access Management (IAM) permissions, see the following:

- IAM and creating DAX clusters: [Creating a DAX Cluster \(p. 668\)](#).
- IAM and DAX data plane operations: [DAX Access Control \(p. 745\)](#).

For the DAX management APIs, you can't scope API actions to a specific resource. The `Resource` element must be set to `"*"`. This is different from DAX data plane API operations, such as `GetItem`, `Query`, and `Scan`. Data plane operations are exposed through the DAX client, and those operations *can* be scoped to specific resources.

To illustrate, consider the following IAM policy document.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dax:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "*"  
            ]  
        }  
    ]  
}
```

```

        "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
    ]
}
]
```

Suppose that the intent of this policy is to allow DAX management API calls for the cluster `DAXCluster01`—and only that cluster.

Now suppose that a user issues the following AWS CLI command.

```
aws dax describe-clusters
```

This command fails with a Not Authorized exception because the underlying `DescribeClusters` API call can't be scoped to a specific cluster. Even though the policy is syntactically valid, the command fails because the `Resource` element must be set to `"*"`. However, if the user runs a program that sends DAX data plane calls (such as `GetItem` or `Query`) to `DAXCluster01`, those calls *do* succeed. This is because DAX data plane APIs can be scoped to specific resources (in this case, `DAXCluster01`).

If you want to write a single comprehensive IAM policy to encompass both DAX management APIs and DAX data plane APIs, we suggest that you include two distinct statements in the policy document. One of these statements should address the DAX data plane APIs, while the other statement addresses the DAX management APIs.

The following example policy shows this approach. Note how the `DAXDataAPIs` statement is scoped to the `DAXCluster01` resource, but the resource for `DAXManagementAPIs` must be `"*"`. The actions shown in each statement are for illustration only. You can customize them as needed for your application.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXDataAPIs",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:DefineKeySchema",
                "dax:Endpoints"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
            ],
            "Condition": {
                "StringLike": {
                    "dax:ReplicationFactor": "1"
                }
            }
        },
        {
            "Sid": "DAXManagementAPIs",
            "Action": [
                "dax>CreateParameterGroup",
                "dax>CreateSubnetGroup",
                "dax:DecreaseReplicationFactor",
                "dax>DeleteCluster",
                "dax>DeleteParameterGroup",
                "dax:ModifyParameterGroup"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

```
        "dax:DeleteSubnetGroup",
        "dax:DescribeClusters",
        "dax:DescribeDefaultParameters",
        "dax:DescribeEvents",
        "dax:DescribeParameterGroups",
        "dax:DescribeParameters",
        "dax:DescribeSubnetGroups",
        "dax:IncreaseReplicationFactor",
        "dax>ListTags",
        "dax:RebootNode",
        "dax:TagResource",
        "dax:UntagResource",
        "dax:UpdateCluster",
        "dax:UpdateParameterGroup",
        "dax:UpdateSubnetGroup"
    ],
    "Effect": "Allow",
    "Resource": [
        "*"
    ]
}
]
```

Scaling a DAX Cluster

There are two options available for scaling a DAX cluster. The first option is *horizontal scaling*, where you add read replicas to the cluster. The second option is *vertical scaling*, where you select different node types. For advice on how to approach choosing an appropriate cluster size and node type for your application, see [DAX Cluster Sizing Guide \(p. 766\)](#).

Horizontal Scaling

With horizontal scaling, you can improve throughput for read operations by adding more read replicas to the cluster. A single DAX cluster supports up to 10 read replicas, and you can add or remove replicas while the cluster is running.

The following AWS CLI examples show how to increase or decrease the number of nodes. The `--new-replication-factor` argument specifies the total number of nodes in the cluster. One of the nodes is the primary node, and the other nodes are read replicas.

```
aws dax increase-replication-factor \
--cluster-name MyNewCluster \
--new-replication-factor 5
```

```
aws dax decrease-replication-factor \
--cluster-name MyNewCluster \
--new-replication-factor 3
```

Note

The cluster status changes to `modifying` when you modify the replication factor. The status changes to `available` when the modification is complete.

Vertical Scaling

If you have a large working set of data, your application might benefit from using larger node types. Larger nodes can enable the cluster to store more data in memory, reducing cache misses and improving

overall application performance of the application. (All of the nodes in a DAX cluster must be of the same type.)

If your DAX cluster has a high rate of write operations or cache misses, your application might also benefit from using larger node types. Write operations and cache misses consume resources on the cluster's primary node. Therefore, using larger node types might increase the performance of the primary node and thereby allow a higher throughput for these types of operations.

You can't modify the node types on a running DAX cluster. Instead, you must create a new cluster with the desired node type. For a list of supported node types, see [Nodes \(p. 664\)](#).

You can create a new DAX cluster using the AWS Management Console, [AWS CloudFormation](#), the AWS CLI, or the [AWS SDK](#). (For the AWS CLI, use the `--node-type` parameter to specify the node type.)

Customizing DAX Cluster Settings

When you create a DAX cluster, the following default settings are used:

- Automatic cache eviction enabled with Time to Live (TTL) of 5 minutes
- No preference for Availability Zones
- No preference for maintenance windows
- Notifications disabled

For new clusters, you can customize the settings at creation time. To do this in the AWS Management Console, clear **Use default settings** to modify the following settings:

- **Network and Security**—Allows you to run individual DAX cluster nodes in different Availability Zones within the current AWS Region. If you choose **No Preference**, the nodes are distributed among Availability Zones automatically.
- **Parameter Group**—A named set of parameters that are applied to every node in the cluster. You can use a parameter group to specify cache TTL behavior. You can change the value of any given parameter within a parameter group (except default parameter group `default.dax.1.0`) at any time.
- **Maintenance Window**—A weekly time period during which software upgrades and patches are applied to the nodes in the cluster. You can choose the start day, start time, and duration of the maintenance window. If you choose **No Preference**, the maintenance window is selected at random from an 8-hour block of time per Region. For more information, see [Maintenance Window \(p. 667\)](#).

Note

Parameter Group and **Maintenance Window** can also be changed at any time on a running cluster.

When a maintenance event occurs, DAX can notify you using Amazon Simple Notification Service (Amazon SNS). To configure notifications, choose an option from the **Topic for SNS notification** selector. You can create a new Amazon SNS topic, or use an existing topic.

For more information about setting up and subscribing to an Amazon SNS topic, see [Getting Started with Amazon SNS](#) in the *Amazon Simple Notification Service Developer Guide*.

Configuring TTL Settings

DAX maintains two caches for data that it reads from DynamoDB:

- **Item cache**—For items retrieved using `GetItem` or `BatchGetItem`.
- **Query cache**—For result sets retrieved using `Query` or `Scan`.

For more information, see [Item Cache \(p. 663\)](#) and [Query Cache \(p. 663\)](#).

The default TTL for each of these caches is 5 minutes. If you want to use different TTL settings, you can launch a DAX cluster using a custom parameter group. To do this on the console, choose **DAX | Parameter groups** in the navigation pane.

You can also perform these tasks using the AWS CLI. The following example shows how to launch a new DAX cluster using a custom parameter group. In this example, the item cache TTL is set to 10 minutes, and the query cache TTL is set to 3 minutes.

1. Create a new parameter group.

```
aws dax create-parameter-group \
--parameter-group-name custom-ttl
```

2. Set the item cache TTL to 10 minutes (600000 milliseconds).

```
aws dax update-parameter-group \
--parameter-group-name custom-ttl \
--parameter-name-values "ParameterName=record-ttl-millis,ParameterValue=600000"
```

3. Set the query cache TTL to 3 minutes (180000 milliseconds).

```
aws dax update-parameter-group \
--parameter-group-name custom-ttl \
--parameter-name-values "ParameterName=query-ttl-millis,ParameterValue=180000"
```

4. Verify that the parameters have been set correctly.

```
aws dax describe-parameters --parameter-group-name custom-ttl \
--query "Parameters[*].[ParameterName,Description,ParameterValue]"
```

You can now launch a new DAX cluster with this parameter group.

```
aws dax create-cluster \
--cluster-name MyNewCluster \
--node-type dax.r3.large \
--replication-factor 3 \
--iam-role-arn arn:aws:iam::123456789012:role/DAXServiceRole \
--parameter-group custom-ttl
```

Note

You can't modify a parameter group that is being used by a running DAX instance.

Tagging Support for DAX

Many AWS services, including DynamoDB, support *tagging*—the ability to label resources with user-defined names. You can assign tags to DAX clusters, allowing you to quickly identify all of your AWS resources that have the same tag, or to categorize your AWS bills by the tags you assign.

For more information, see [Adding Tags and Labels to Resources \(p. 359\)](#).

Using the AWS Management Console

To manage DAX cluster tags

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.

2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose the cluster that you want to work with.
4. Choose the **Tags** tab. You can add, list, edit, or delete your tags here.

When the settings are as you want them, choose **Apply Changes**.

Using the AWS CLI

When you use the AWS CLI to manage DAX cluster tags, you must first determine the Amazon Resource Name (ARN) for the cluster. The following example shows how to determine the ARN for a cluster named `MyDAXCluster`.

```
aws dax describe-clusters \
--cluster-name MyDAXCluster \
--query "Clusters[*].ClusterArn"
```

In the output, the ARN will look similar to this: `arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster`

The following example shows how to tag the cluster.

```
aws dax tag-resource \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
--tags="Key=ClusterUsage,Value=prod"
```

List all the tags for a cluster.

```
aws dax list-tags \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster
```

To remove a tag, specify its key.

```
aws dax untag-resource \
--resource-name arn:aws:dax:us-west-2:123456789012:cache/MyDAXCluster \
--tag-keys ClusterUsage
```

AWS CloudTrail Integration

DAX is integrated with AWS CloudTrail, allowing you to audit DAX cluster activities. You can use CloudTrail logs to view all the changes that have been made at the cluster level. You can also see changes to cluster components such as nodes, subnet groups, and parameter groups. For more information, see [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 877\)](#).

Deleting a DAX Cluster

If you are no longer using a DAX cluster, you should delete it to avoid being charged for unused resources.

You can delete a DAX cluster using the console or the AWS CLI. The following is an example.

```
aws dax delete-cluster --cluster-name mydaxcluster
```

Monitoring DAX

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon DynamoDB Accelerator (DAX) and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs.

Before you start monitoring DAX, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal DAX performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DAX, you should consider storing historical monitoring data. This stored data gives you a baseline from which to compare current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline, you should, at a minimum, monitor the following items both during load testing and in production:

- CPU utilization and throttled requests, so that you can determine whether you might need to use a larger node type in your cluster. The CPU utilization of your cluster is available through the [CPUUtilization CloudWatch metric \(p. 734\)](#).
- Operation latency (as measured on the client side) should remain consistently within your application's latency requirements.
- Error rates should remain low, as seen from the [ErrorRequestCount](#), [FaultRequestCount](#), and [FailedRequestCount CloudWatch metrics \(p. 734\)](#).

Apart from the preceding items, you should, at a minimum, monitor the following additional items in production:

- Estimated database size and evicted size, so that you can determine whether the cluster's node type has sufficient memory to hold your working set.
- Client connections, so that you can monitor for any unexplained spikes in connections to the cluster.

Topics

- [Monitoring Tools \(p. 732\)](#)
- [Monitoring with Amazon CloudWatch \(p. 733\)](#)
- [Logging DAX Operations Using AWS CloudTrail \(p. 745\)](#)

Monitoring Tools

AWS provides tools that you can use to monitor Amazon DynamoDB Accelerator (DAX). You can configure some of these tools to do the monitoring for you, and some require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Topics

- [Automated Monitoring Tools \(p. 733\)](#)
- [Manual Monitoring Tools \(p. 733\)](#)

Automated Monitoring Tools

You can use the following automated monitoring tools to watch DAX and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see [Monitoring with Amazon CloudWatch \(p. 857\)](#).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see [Monitoring Log Files](#) in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see [What Is Amazon CloudWatch Events](#) in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see [Working with CloudTrail Log Files](#) in the *AWS CloudTrail User Guide*.

Manual Monitoring Tools

Another important part of monitoring DAX involves manually monitoring those items that the CloudWatch alarms don't cover. The DAX, CloudWatch, Trusted Advisor, and other AWS Management Console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on DAX.

- The DAX dashboard shows the following:
 - Service health
- The CloudWatch home page shows the following:
 - Current alarms and status
 - Graphs of alarms and resources
 - Service health status

In addition, you can use CloudWatch to do the following:

- Create [customized dashboards](#) to monitor the services that you care about.
- Graph metric data to troubleshoot issues and discover trends.
- Search and browse all of your AWS resource metrics.
- Create and edit alarms to be notified of problems.

Monitoring with Amazon CloudWatch

You can monitor Amazon DynamoDB Accelerator (DAX) using Amazon CloudWatch, which collects and processes raw data from DAX into readable, near real-time metrics. These statistics are recorded for a period of two weeks. You can then access historical information for a better perspective on how

your web application or service is performing. By default, DAX metric data is sent to CloudWatch automatically. For more information, see [What Is Amazon CloudWatch?](#) in the *Amazon CloudWatch User Guide*.

Topics

- [How Do I Use DAX Metrics? \(p. 734\)](#)
- [Viewing DAX Metrics and Dimensions \(p. 734\)](#)
- [Creating CloudWatch Alarms to Monitor DAX \(p. 743\)](#)

How Do I Use DAX Metrics?

The metrics reported by DAX provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, and not a comprehensive list.

| How Can I? | Relevant Metrics |
|---|---|
| Determine if any system errors occurred | Monitor <code>FaultRequestCount</code> to determine if any requests resulted in an HTTP 500 (server error) code. This can indicate a DAX internal service error or an HTTP 500 in the underlying table's SystemErrors metric . |
| Determine if any user errors occurred | Monitor <code>ErrorRequestCount</code> to determine if any requests resulted in an HTTP 400 (client error) code. If you see the error count growing, you might want to investigate and make sure you are sending correct client requests. |
| Determine if any cache misses occurred | Monitor <code>ItemCacheMisses</code> to determine the number of times an item was not found in the cache, and <code>QueryCacheMisses</code> and <code>ScanCacheMisses</code> to determine the number of times a query or scan result was not found in the cache. |
| Monitor cache hit rates | Use CloudWatch Metric Math to define a cache hit rate metric using math expressions. For example, for the item cache, you can use the expression $m1 / \text{SUM}([m1, m2]) * 100$, where <code>m1</code> is the <code>ItemCacheHits</code> metric and <code>m2</code> is the <code>ItemCacheMisses</code> metric for your cluster. For the query and scan caches, you can follow the same pattern using the corresponding query and scan cache metric. |

Viewing DAX Metrics and Dimensions

When you interact with Amazon DynamoDB, it sends metrics and dimensions to Amazon CloudWatch. You can use the following procedures to view the metrics for DynamoDB Accelerator (DAX).

To view metrics (console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Metrics**.

3. Select the **DAX** namespace.

To view metrics (AWS CLI)

- At a command prompt, use the following command.

```
aws cloudwatch list-metrics --namespace "AWS/DAX"
```

DAX Metrics and Dimensions

The following sections contain the metrics and dimensions that DAX sends to CloudWatch.

DAX Metrics

The following metrics are available from DAX. DAX sends metrics to CloudWatch only when they have a non-zero value.

Note

CloudWatch aggregates the following DAX metrics at one-minute intervals:

- CPUUtilization
- NetworkBytesIn
- NetworkBytesOut
- NetworkPacketsIn
- NetworkPacketsOut
- GetItemRequestCount
- BatchGetItemRequestCount
- BatchWriteItemRequestCount
- DeleteItemRequestCount
- PutItemRequestCount
- UpdateItemRequestCount
- TransactWriteItemsCount
- TransactGetItemsCount
- ItemCacheHits
- ItemCacheMisses
- QueryCacheHits
- QueryCacheMisses
- ScanCacheHits
- ScanCacheMisses
- TotalRequestCount
- ErrorRequestCount
- FaultRequestCount
- FailedRequestCount
- QueryRequestCount
- ScanRequestCount
- ClientConnections
- EstimatedDbSize

- `EvictedSize`

Not all statistics, such as Average or Sum, are applicable for every metric. However, all of these values are available through the DAX console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics. In the following table, each metric has a list of valid statistics that are applicable to that metric.

| Metric | Description |
|-------------------|---|
| CPUUtilization | <p>The percentage of CPU utilization of the node or cluster.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |
| NetworkBytesIn | <p>The number of bytes received on all network interfaces by the node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |
| NetworkBytesOut | <p>The number of bytes sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of bytes on a single node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |
| NetworkPacketsIn | <p>The number of packets received on all network interfaces by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |
| NetworkPacketsOut | <p>The number of packets sent out on all network interfaces by the node or cluster. This metric identifies the volume of outgoing traffic in terms of the number of packets on a single node or cluster.</p> |

| Metric | Description |
|----------------------------|--|
| | <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |
| GetItemRequestCount | <p>The number of <code>GetItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| BatchGetItemRequestCount | <p>The number of <code>BatchGetItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| BatchWriteItemRequestCount | <p>The number of <code>BatchWriteItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|-------------------------|--|
| DeleteItemRequestCount | <p>The number of <code>DeleteItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| PutItemRequestCount | <p>The number of <code>PutItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| UpdateItemRequestCount | <p>The number of <code>UpdateItem</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| TransactWriteItemsCount | <p>The number of <code>TransactWriteItems</code> requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|----------------------|--|
| TransactGetItemCount | <p>The number of TransactGetItems requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ItemCacheHits | <p>The number of times an item was returned from the cache by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ItemCacheMisses | <p>The number of times an item was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| QueryCacheHits | <p>The number of times a query result was returned from the node or cluster cache.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|-------------------|---|
| QueryCacheMisses | <p>The number of times a query result was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ScanCacheHits | <p>The number of times a scan result was returned from the node or cluster cache.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ScanCacheMisses | <p>The number of times a scan result was not in the node or cluster cache, and had to be retrieved from DynamoDB.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| TotalRequestCount | <p>Total number of requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|-----------------------|---|
| ErrorRequestCount | <p>Total number of requests that resulted in a user error reported by the node or cluster. Requests that were throttled by the node or cluster are included.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ThrottledRequestCount | <p>Total number of requests throttled by the node or cluster. Requests that were throttled by DynamoDB are not included, and can be monitored using DynamoDB Metrics (p. 857).</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| FaultRequestCount | <p>Total number of requests that resulted in an internal error reported by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| FailedRequestCount | <p>Total number of requests that resulted in an error reported by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|-------------------|--|
| QueryRequestCount | <p>The number of query requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ScanRequestCount | <p>The number of scan requests handled by the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ClientConnections | <p>The number of simultaneous connections made by clients to the node or cluster.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| EstimatedDbSize | <p>An approximation of the amount of data cached in the item cache and the query cache by the node or cluster.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average |

| Metric | Description |
|-------------|---|
| EvictedSize | <p>The amount of data that was evicted by the node or cluster to make room for newly requested data. If the hit rates goes up, and you see this metric also growing, it probably means that your working set has increased. You should consider switching to a cluster with a larger node type.</p> <p>Units: Bytes</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • Sum |

Dimensions for DAX Metrics

The metrics for DAX are qualified by the values for the account, cluster ID, or cluster ID and node ID combination. You can use the CloudWatch console to retrieve DAX data along any of the dimensions in the following table.

| Dimension | Description |
|-------------------|---|
| Account | DAX provides aggregated statistics across all clusters in an account. |
| ClusterId | Limits the data to a cluster. |
| ClusterId, NodeId | Limits the data to a node within a cluster |

Creating CloudWatch Alarms to Monitor DAX

You can create an Amazon CloudWatch alarm that sends an Amazon Simple Notification Service (Amazon SNS) message when the alarm changes state. An alarm watches a single metric over a time period that you specify. It performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification that is sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state. The state must have changed and been maintained for a specified number of periods.

How Can I Be Notified of Query Cache Misses?

1. Create an Amazon SNS topic, `arn:aws:sns:us-west-2:522194210714:QueryMissAlarm`.

For more information, see [Set Up Amazon Simple Notification Service](#) in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
Prompt>aws cloudwatch put-metric-alarm \
--alarm-name QueryCacheMissesAlarm \
--alarm-description "Alarm over query cache misses" \
--namespace AWS/DAX \
--metric-name QueryCacheMisses \
--dimensions Name=ClusterID,Value=myCluster \
--statistic Sum \
--threshold 8 \
--comparison-operator GreaterThanOrEqualToThreshold \
--period 60 \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-west-2:522194210714:QueryMissAlarm
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-reason
"initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name QueryCacheMissesAlarm --state-
reason "initializing" --state-value ALARM
```

Note

You can increase or decrease the threshold to one that makes sense for your application. You can also use [CloudWatch Metric Math](#) to define a cache miss rate metric and set an alarm over that metric.

How Can I Be Notified If Requests Cause Any Internal Error in the Cluster?

1. Create an Amazon SNS topic, arn:aws:sns:us-west-2:123456789012:notify-on-system-errors.

For more information, see [Set Up Amazon Simple Notification Service](#) in the *Amazon CloudWatch User Guide*.

2. Create the alarm.

```
Prompt>aws cloudwatch put-metric-alarm \
--alarm-name FaultRequestCountAlarm \
--alarm-description "Alarm when a request causes an internal error" \
--namespace AWS/DAX \
--metric-name FaultRequestCount \
--dimensions Name=ClusterID,Value=myCluster \
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 60 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-
reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name FaultRequestCountAlarm --state-
reason "initializing" --state-value ALARM
```

Logging DAX Operations Using AWS CloudTrail

Amazon DynamoDB Accelerator (DAX) is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DAX.

To learn more about DAX and CloudTrail, see the DynamoDB Accelerator (DAX) section in [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 877\)](#).

DAX Access Control

DynamoDB Accelerator (DAX) is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB have separate access control mechanisms. Both services use AWS Identity and Access Management (IAM) to implement their respective security policies, but the security models for DAX and DynamoDB are different.

We highly recommend that you understand both security models, so that you can implement proper security measures for your applications that use DAX.

This section describes the access control mechanisms provided by DAX and provides sample IAM policies that you can tailor to your needs.

With DynamoDB, you can create IAM policies that limit the actions a user can perform on individual DynamoDB resources. For example, you can create a user role that only allows the user to perform read-only actions on a particular DynamoDB table. (For more information, see [the section called "Identity and Access Management" \(p. 824\)](#).) By comparison, the DAX security model focuses on cluster security, and the ability of the cluster to perform DynamoDB API actions on your behalf.

Warning

If you are currently using IAM roles and policies to restrict access to DynamoDB tables data, then the use of DAX can **subvert** those policies. For example, a user could have access to a DynamoDB table via DAX but not have explicit access to the same table accessing DynamoDB directly. For more information, see [the section called "Identity and Access Management" \(p. 824\)](#).

DAX does not enforce user-level separation on data in DynamoDB. Instead, users inherit the permissions of the DAX cluster's IAM policy when they access that cluster. Thus, when accessing DynamoDB tables via DAX, the only access controls that are in effect are the permissions in the DAX cluster's IAM policy. No other permissions are recognized.

If you require isolation, we recommend that you create additional DAX clusters and scope the IAM policy for each cluster accordingly. For example, you could create multiple DAX clusters and allow each cluster to access only a single table.

IAM Service Role for DAX

When you create a DAX cluster, you must associate the cluster with an IAM role. This is known as the *service role* for the cluster.

Suppose that you wanted to create a new DAX cluster named *DAXCluster01*. You could create a service role named *DAXServiceRole*, and associate the role with *DAXCluster01*. The policy for *DAXServiceRole* would define the DynamoDB actions that *DAXCluster01* could perform, on behalf of the users who interact with *DAXCluster01*.

When you create a service role, you must specify a trust relationship between *DAXServiceRole* and the DAX service itself. A trust relationship determines which entities can assume a role and make use of its permissions. The following is an example trust relationship document for *DAXServiceRole*:

```
{  
    "Version": "2012-10-17",
```

```

"Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "Service": "dax.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }
]
}

```

This trust relationship allows a DAX cluster to assume *DAXServiceRole* and perform DynamoDB API calls on your behalf.

The DynamoDB API actions that are allowed are described in an IAM policy document, which you attach to *DAXServiceRole*. The following is an example policy document.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DaxAccessPolicy",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:PutItem",
                "dynamodb:GetItem",
                "dynamodb:UpdateItem",
                "dynamodb>DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
            ]
        }
    ]
}

```

This policy allows DAX to perform necessary DynamoDB API actions on a DynamoDB table. The `dynamodb:DescribeTable` action is required for DAX to maintain metadata about the table, and the others are read and write actions performed on items in the table. The table, named `Books`, is in the us-west-2 Region and is owned by AWS account ID 123456789012.

IAM Policy to Allow DAX Cluster Access

After you create a DAX cluster, you need to grant permissions to an IAM user so that the user can access the DAX cluster.

For example, suppose that you want to grant access to *DAXCluster01* to an IAM user named Alice. You would first create an IAM policy (*AliceAccessPolicy*) that defines the DAX clusters and DAX API actions that the recipient can access. You would then confer access by attaching this policy to user Alice.

The following policy document gives the recipient full access on *DAXCluster01*.

```

{
    "Version": "2012-10-17",
    "Statement": [

```

```
{
    "Action": [
        "dax:*"
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
    ]
}
}
```

The policy document allows access to the DAX cluster, but it does not grant any DynamoDB permissions. (The DynamoDB permissions are conferred by the DAX service role.)

For user Alice, you would first create `AliceAccessPolicy` with the policy document shown previously. You would then attach the policy to Alice.

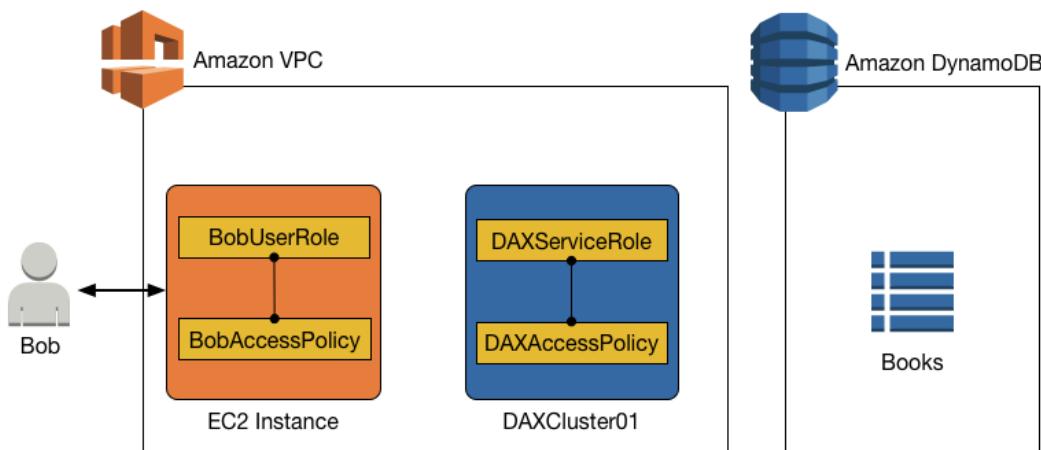
Note

Instead of attaching the policy to an IAM user, you could attach it to an IAM role. That way, all of the users who assume that role would have the permissions that you defined in the policy.

The user policy, together with the DAX service role, determine the DynamoDB resources and API actions that the recipient can access via DAX.

Case Study: Accessing DynamoDB and DAX

The following scenario can help further your understanding of IAM policies for use with DAX. (This scenario is referred to throughout the rest of this section.) The following diagram shows a high-level overview of the scenario.



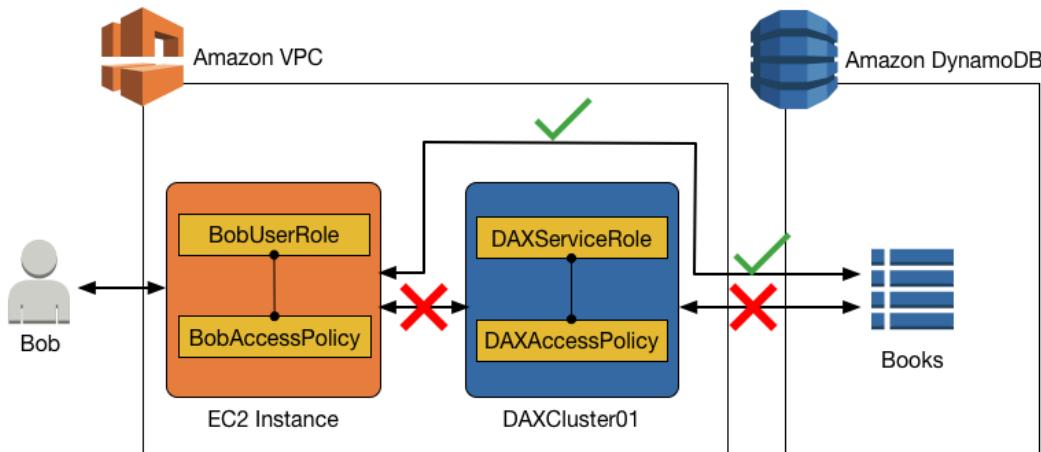
In this scenario, there are the following entities:

- An IAM user (Bob).
- An IAM role (`BobUserRole`). Bob assumes this role at runtime.
- An IAM policy (`BobAccessPolicy`). This policy is attached to `BobUserRole`. `BobAccessPolicy` defines the DynamoDB and DAX resources that `BobUserRole` is allowed to access.
- A DAX cluster (`DAXCluster01`).

- An IAM service role (**DAXServiceRole**). This role allows **DAXCluster01** to access DynamoDB.
 - An IAM policy (**DAXAccessPolicy**). This policy is attached to **DAXServiceRole**. **DAXAccessPolicy** defines the DynamoDB APIs and resources that **DAXCluster01** is allowed to access.
 - A DynamoDB table (**Books**).

The combination of policy statements in `BobAccessPolicy` and `DAXAccessPolicy` determine what Bob can do with the `Books` table. For example, Bob might be able to access `Books` directly (using the DynamoDB endpoint), indirectly (using the DAX cluster), or both. Bob might also be able to read data from `Books`, write data to `Books`, or both.

Access to DynamoDB, But No Access with DAX



It is possible to allow direct access to a DynamoDB table, while preventing indirect access using a DAX cluster. For direct access to DynamoDB, the permissions for `BobUserRole` are determined by `BobAccessPolicy` (which is attached to the role).

Read-Only Access to DynamoDB (Only)

Bob can access DynamoDB with `BobUserRole`. The IAM policy attached to this role (`BobAccessPolicy`) determines the DynamoDB tables that `BobUserRole` can access, and what APIs that `BobUserRole` can invoke.

Consider the following policy document for BobAccessPolicy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DynamoDBAccessStmt",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:Scan"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

```
        ]
    }
```

When this document is attached to `BobAccessPolicy`, it allows `BobUserRole` to access the DynamoDB endpoint and perform read-only operations on the `Books` table.

DAX does not appear in this policy, so access via DAX is denied.

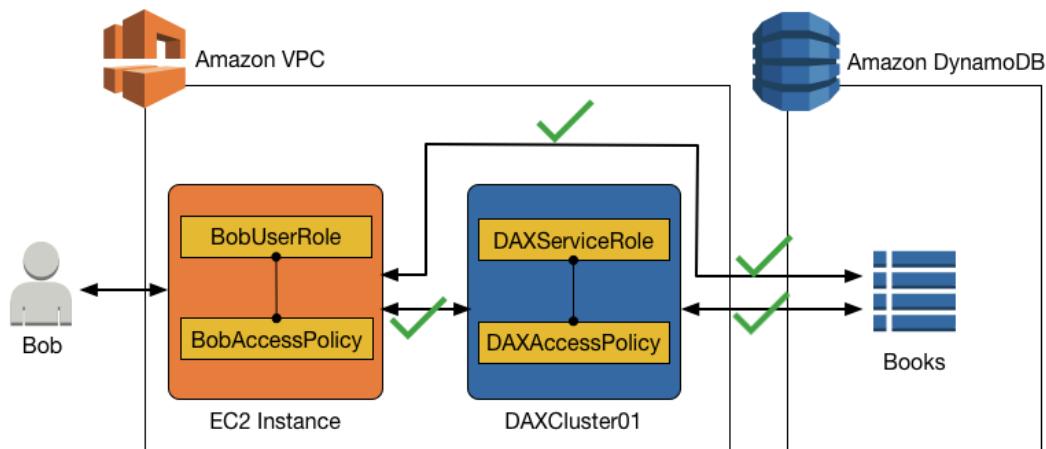
Read/Write Access to DynamoDB (Only)

If `BobUserRole` requires read/write access to DynamoDB, the following policy would work.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Again, DAX does not appear in this policy, so access via DAX is denied.

Access to DynamoDB and to DAX



To allow access to a DAX cluster, you must include DAX-specific actions in an IAM policy.

The following DAX-specific actions correspond to their similarly named counterparts in the DynamoDB API:

- `dax:.GetItem`
 - `dax:BatchGetItem`
 - `dax:Query`
 - `dax:Scan`
 - `dax:PutItem`
 - `dax:UpdateItem`
 - `dax:DeleteItem`
 - `dax:BatchWriteItem`
 - `dax:ConditionCheckItem`

The same is true for the `dax:EnclosingOperation` condition key.

In addition, there are four other DAX-specific actions that do not correspond to any DynamoDB APIs:

- `dax:DefineAttributeList`
 - `dax:DefineAttributeListId`
 - `dax:DefineKeySchema`
 - `dax:Endpoints`

You must specify all four of these actions in any IAM policy that allows access to a DAX cluster. These actions are specific to the low-level DAX data transport protocol. Your application does not need to concern itself with these actions—they are only used in IAM policies.

Read-Only Access to DynamoDB and Read-Only Access to DAX

Suppose that Bob requires read-only access to the Books table, from DynamoDB and from DAX. The following policy (attached to `BobUserRole`) confers this access.

```

        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:Scan"
    ],
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
}
}

```

The policy has a statement for DAX access (`DAXAccessStmt`) and another statement for DynamoDB access (`DynamoDBAccessStmt`). These statements allow Bob to send `GetItem`, `BatchGetItem`, `Query`, and `Scan` requests to `DAXCluster01`.

However, the service role for `DAXCluster01` would also require read-only access to the `Books` table in DynamoDB. The following IAM policy, attached to `DAXServiceRole`, would fulfill this requirement.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}

```

ReadWrite Access to DynamoDB and Read-Only with DAX

For a given user role, you can provide read/write access to a DynamoDB table, while also allowing read-only access via DAX.

For Bob, the IAM policy for `BobUserRole` would need to allow DynamoDB read and write actions on the `Books` table, while also supporting read-only actions via `DAXCluster01`.

The following example policy document for `BobUserRole` confers this access.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:.GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:DefineKeySchema",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:Endpoints"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        },
        {

```

```

        "Sid": "DynamoDBAccessStmt",
        "Effect": "Allow",
        "Action": [
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:Scan",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem",
            "dynamodb:DescribeTable",
            "dynamodb:ConditionCheckItem"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
    }
]
}

```

In addition, `DAXServiceRole` would require an IAM policy that allows `DAXCluster01` to perform read-only actions on the `Books` table.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}

```

ReadWrite Access to DynamoDB and Read/Write Access to DAX

Now suppose that Bob required read/write access to the `Books` table, directly from DynamoDB or indirectly from `DAXCluster01`. The following policy document, attached to `BobAccessPolicy`, confers this access.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:DefineKeySchema",
                "dax:ListTables"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cluster/DAXCluster01"
        }
    ]
}

```

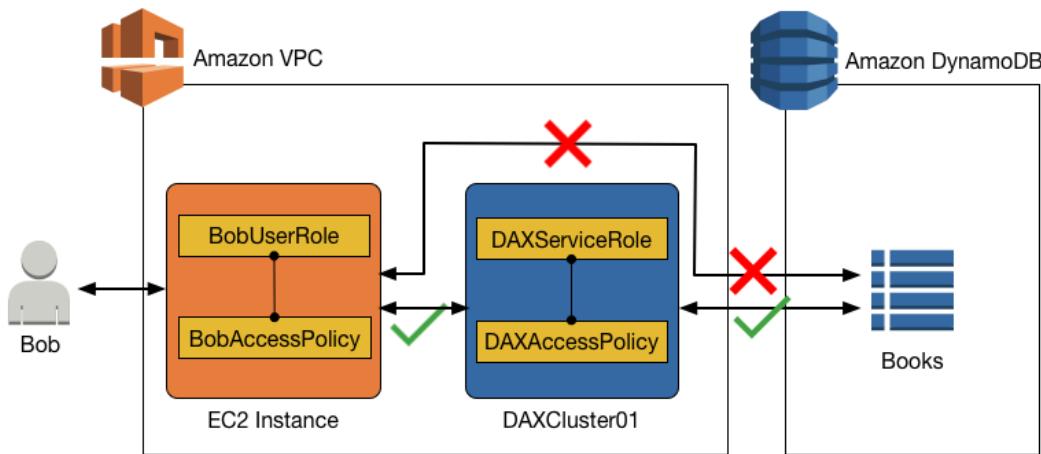
```
"dax:DefineAttributeList",
"dax:DefineAttributeListId",
"dax:Endpoints",
"dax:ConditionCheckItem"
],
"Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
},
{
"Sid": "DynamoDBAccessStmt",
"Effect": "Allow",
>Action": [
    "dynamodb:GetItem",
    "dynamodb:BatchGetItem",
    "dynamodb:Query",
    "dynamodb:Scan",
    "dynamodb:PutItem",
    "dynamodb:UpdateItem",
    "dynamodb:DeleteItem",
    "dynamodb:BatchWriteItem",
    "dynamodb:DescribeTable",
    "dynamodb:ConditionCheckItem"
],
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
}
]
```

In addition, `DAXServiceRole` would require an IAM policy that allows `DAXCluster01` to perform read/write actions on the `Books` table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Access to DynamoDB Via DAX, But No Direct Access to DynamoDB

In this scenario, Bob can access the `Books` table via DAX, but he does not have direct access to the `Books` table in DynamoDB. Thus, when Bob gains access to DAX, he also gains access to a DynamoDB table that he otherwise might not be able to access. When you configure an IAM policy for the DAX service role, remember that any user that is given access to the DAX cluster via the user access policy gains access to the tables specified in that policy. In this case, `BobAccessPolicy` gains access to the tables specified in `DAXAccessPolicy`.



If you are currently using IAM roles and policies to restrict access to DynamoDB tables and data, using DAX can subvert those policies. In the following policy, Bob has access to a DynamoDB table via DAX but does not have explicit direct access to the same table in DynamoDB.

The following policy document (**BobAccessPolicy**), attached to **BobUserRole**, would confer this access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DAXAccessStmt",
            "Effect": "Allow",
            "Action": [
                "dax:GetItem",
                "dax:BatchGetItem",
                "dax:Query",
                "dax:Scan",
                "dax:PutItem",
                "dax:UpdateItem",
                "dax:DeleteItem",
                "dax:BatchWriteItem",
                "dax:DefineKeySchema",
                "dax:DefineAttributeList",
                "dax:DefineAttributeListId",
                "dax:Endpoints",
                "dax:ConditionCheckItem"
            ],
            "Resource": "arn:aws:dax:us-west-2:123456789012:cache/DAXCluster01"
        }
    ]
}
```

In this access policy, there are no permissions to access DynamoDB directly.

Together with **BobAccessPolicy**, the following **DAXAccessPolicy** gives **BobUserRole** access to the **DynamoDB** table **Books** even though **BobUserRole** cannot directly access the **Books** table.

```
{
    "Version": "2012-10-17",
```

```

"Statement": [
    {
        "Sid": "DynamoDBAccessStmt",
        "Effect": "Allow",
        "Action": [
            "dynamodb:GetItem",
            "dynamodb:BatchGetItem",
            "dynamodb:Query",
            "dynamodb:Scan",
            "dynamodb:PutItem",
            "dynamodb:UpdateItem",
            "dynamodb:DeleteItem",
            "dynamodb:BatchWriteItem",
            "dynamodb:DescribeTable",
            "dynamodb:ConditionCheckItem"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
    }
]
}

```

As this example shows, when you configure access control for the user access policy and the DAX cluster access policy, you must fully understand the end-to-end access to ensure that the principle of least privilege is observed. Also ensure that giving a user access to a DAX cluster does not subvert previously established access control policies.

DAX Encryption at Rest

Amazon DynamoDB Accelerator (DAX) encryption at rest provides an additional layer of data protection by helping secure your data from unauthorized access to the underlying storage. Organizational policies, industry or government regulations, and compliance requirements might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of your applications that are deployed in the cloud.

With encryption at rest, the data persisted by DAX on disk is encrypted using 256-bit Advanced Encryption Standard, also known as AES-256 encryption. DAX writes data to disk as part of propagating changes from the primary node to read replicas.

DAX encryption at rest automatically integrates with AWS Key Management Service (AWS KMS) for managing the single service default key that is used to encrypt your clusters. If a service default key doesn't exist when you create your encrypted DAX cluster, AWS KMS automatically creates a new key for you. This key is used with encrypted clusters that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

After your data is encrypted, DAX handles the decryption of your data transparently with minimal impact on performance. You don't need to modify your applications to use encryption.

Note

DAX does not call AWS KMS for every single DAX operation. DAX only uses the key at the cluster launch. Even if access is revoked, DAX can still access the data until the cluster is shut down. Customer-specified AWS KMS keys are not supported.

DAX encryption at rest is available for the following cluster node types.

| Family | Node Type |
|-----------------------|---------------|
| Memory-optimized (R4) | dax.r4.large |
| | dax.r4.xlarge |

| Family | Node Type |
|----------------------|-----------------|
| General purpose (T2) | dax.r4.2xlarge |
| | dax.r4.4xlarge |
| | dax.r4.8xlarge |
| | dax.r4.16xlarge |
| General purpose (T2) | dax.t2.small |
| | dax.t2.medium |

Important

DAX encryption at rest is not supported for `dax.r3.*` node types.

You cannot enable or disable encryption at rest after a cluster has been created. You must re-create the cluster to enable encryption at rest if it was not enabled at creation.

DAX encryption at rest is offered at no additional cost (AWS KMS encryption key usage charges apply). For information about pricing, see [Amazon DynamoDB Pricing](#).

Enabling Encryption at Rest Using the AWS Management Console

Follow these steps to enable DAX encryption at rest on a table using the console.

To enable DAX encryption at rest

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, under **DAX**, choose **Clusters**.
3. Choose **Create cluster**.
4. For **Cluster name**, enter a short name for your cluster. Choose the **node type** for all of the nodes in the cluster, and for the cluster size, use **3** nodes.
5. In **Encryption**, make sure that **Enable encryption** is selected.

Encryption **Enable encryption**

You may enable encryption for your DAX cluster to help protect data at rest. [Learn more](#)

6. After choosing the IAM role, subnet group, security groups, and cluster settings, choose **Launch cluster**.

To confirm that the cluster is encrypted, check the cluster details under the **Clusters** pane. Encryption should be **ENABLED**.

Using Service-Linked IAM Roles for DAX

Amazon DynamoDB Accelerator (DAX) uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to DAX. Service-linked roles

are predefined by DAX and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up DAX easier because you don't have to manually add the necessary permissions. DAX defines the permissions of its service-linked roles, and unless defined otherwise, only DAX can assume its roles. The defined permissions include the trust policy and the permissions policy. That permissions policy can't be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your DAX resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) in the *IAM User Guide*. Look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** link to view the service-linked role documentation for that service.

Topics

- [Service-Linked Role Permissions for DAX \(p. 757\)](#)
- [Creating a Service-Linked Role for DAX \(p. 758\)](#)
- [Editing a Service-Linked Role for DAX \(p. 758\)](#)
- [Deleting a Service-Linked Role for DAX \(p. 758\)](#)

Service-Linked Role Permissions for DAX

DAX uses the service-linked role named `AWSServiceRoleForDAX`. This role allows DAX to call AWS services on behalf of your DAX cluster.

Important

The `AWSServiceRoleForDAX` service-linked role makes it easier for you to set up and maintain a DAX cluster. However, you must still grant each cluster access to DynamoDB before you can use it. For more information, see [DAX Access Control \(p. 745\)](#).

The `AWSServiceRoleForDAX` service-linked role trusts the following services to assume the role:

- `dax.amazonaws.com`

The role permissions policy allows DAX to complete the following actions on the specified resources:

- Actions on `ec2`:
 - `AuthorizeSecurityGroupIngress`
 - `CreateNetworkInterface`
 - `CreateSecurityGroup`
 - `DeleteNetworkInterface`
 - `DeleteSecurityGroup`
 - `DescribeAvailabilityZones`
 - `DescribeNetworkInterfaces`
 - `DescribeSecurityGroups`
 - `DescribeSubnets`
 - `DescribeVpcs`
 - `ModifyNetworkInterfaceAttribute`
 - `RevokeSecurityGroupIngress`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

To allow an IAM entity to create `AWSServiceRoleForDAX` service-linked roles

Add the following policy statement to the permissions for that IAM entity.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "iam:CreateServiceLinkedRole"  
    ],  
    "Resource": "*",  
    "Condition": {"StringLike": {"iam:AWSServiceName": "dax.amazonaws.com"}}  
}
```

Creating a Service-Linked Role for DAX

You don't need to manually create a service-linked role. When you create a cluster, DAX creates the service-linked role for you.

Important

If you were using the DAX service before February 28, 2018, when it began supporting service-linked roles, DAX created the `AWSServiceRoleForDAX` role in your account. For more information, see [A New Role Appeared in My AWS Account](#) in the *IAM User Guide*.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create an instance or a cluster, DAX creates the service-linked role for you again.

Editing a Service-Linked Role for DAX

DAX does not allow you to edit the `AWSServiceRoleForDAX` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a Service-Linked Role for DAX

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your DAX clusters before you can delete the service-linked role.

Cleaning Up a Service-Linked Role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the `AWSServiceRoleForDAX` role.

3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

Note

If you are unsure whether DAX is using the `AWSServiceRoleForDAX` role, you can try to delete the role. If the service is using the role, the deletion fails, and you can view the Regions where the role is being used. If the role is being used, you must delete your DAX clusters before you can delete the role. You can't revoke the session for a service-linked role.

If you want to remove the `AWSServiceRoleForDAX` role, you must first delete all of your DAX clusters.

Deleting All of Your DAX Clusters

Use one of these procedures to delete each of your DAX clusters.

To delete a DAX cluster (console)

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, under **DAX**, choose **Clusters**.
3. Choose **Actions**, and then choose **Delete**.
4. In the **Delete cluster confirmation** box, choose **Delete**.

To delete a DAX cluster (AWS CLI)

See [delete-cluster](#) in the *AWS CLI Command Reference*.

To delete a DAX cluster (API)

See [DeleteCluster](#) in the *Amazon DynamoDB API Reference*.

Deleting the Service-Linked Role

To manually delete the service-linked role using IAM

Use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForDAX` service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Accessing DAX Across AWS Accounts

Imagine that you have a DynamoDB Accelerator (DAX) cluster running in one AWS account (account A), and the DAX cluster needs to be accessible from an Amazon Elastic Compute Cloud (Amazon EC2) instance in another AWS account (account B). In this tutorial, you accomplish this by launching an EC2 instance in account B with an IAM role from account B. You then use temporary security credentials from the EC2 instance to assume an IAM role from account A. Finally, you use the temporary security credentials from assuming the IAM role in account A to make application calls over an Amazon VPC peering connection to the DAX cluster in account A. In order to perform these tasks you will need administrative access in both AWS accounts.

Topics

- [Set Up IAM \(p. 760\)](#)
- [Set Up a VPC \(p. 761\)](#)
- [Modify the DAX Client to Allow Cross-account Access \(p. 763\)](#)

Set Up IAM

1. Create a text file named `AssumeDaxRoleTrust.json` with the following content, which allows Amazon EC2 to work on your behalf.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "ec2.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

2. In account B, create a role that Amazon EC2 can use when launching instances.

```
aws iam create-role \  
    --role-name AssumeDaxRole \  
    --assume-role-policy-document file://AssumeDaxRoleTrust.json
```

3. Create a text file named `AssumeDaxRolePolicy.json` with the following content, which allows code running on the EC2 instance in account B to assume an IAM role in account A. Replace `accountA` with the actual ID of account A.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "sts:AssumeRole",  
            "Resource": "arn:aws:iam::accountA:role/DaxCrossAccountRole"  
        }  
    ]  
}
```

4. Add that policy to the role you just created.

```
aws iam put-role-policy \  
    --role-name AssumeDaxRole \  
    --policy-name AssumeDaxRolePolicy \  
    --policy-document file://AssumeDaxRolePolicy.json
```

5. Create an instance profile to allow instances to use the role.

```
aws iam create-instance-profile \  
    --instance-profile-name AssumeDaxInstanceProfile
```

6. Associate the role with the instance profile.

```
aws iam add-role-to-instance-profile \  
    --instance-profile-name AssumeDaxInstanceProfile \  
    --role-name AssumeDaxRole
```

7. Create a text file named `DaxCrossAccountRoleTrust.json` with the following content, which allows account B to assume an account A role. Replace `accountB` with the actual ID of account B.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::accountB:role/AssumeDaxRole"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

8. In account A, create the role that account B can assume.

```
aws iam create-role \  
    --role-name DaxCrossAccountRole \  
    --assume-role-policy-document file://DaxCrossAccountRoleTrust.json
```

9. Create a text file named `DaxCrossAccountPolicy.json` that allows access to the DAX cluster. Replace `dax-cluster-arn` with the correct Amazon Resource Name (ARN) of your DAX cluster.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dax:GetItem",  
                "dax:BatchGetItem",  
                "dax:Query",  
                "dax:Scan",  
                "dax:PutItem",  
                "dax:UpdateItem",  
                "dax:DeleteItem",  
                "dax:BatchWriteItem",  
                "dax:ConditionCheckItem",  
                "dax:DefineAttributeList",  
                "dax:DefineAttributeListId",  
                "dax:DefineKeySchema",  
                "dax:Endpoints"  
            ],  
            "Resource": "dax-cluster-arn"  
        }  
    ]  
}
```

10. In account A, add the policy to the role.

```
aws iam put-role-policy \  
    --role-name DaxCrossAccountRole \  
    --policy-name DaxCrossAccountPolicy \  
    --policy-document file://DaxCrossAccountPolicy.json
```

Set Up a VPC

1. Find the subnet group of account A's DAX cluster. Replace `cluster-name` with the name of the DAX cluster that account B must access.

```
aws dax describe-clusters \
--cluster-name cluster-name \
--query 'Clusters[0].SubnetGroup'
```

2. Using that *subnet-group*, find the cluster's VPC.

```
aws dax describe-subnet-groups \
--subnet-group-name subnet-group \
--query 'SubnetGroups[0].VpcId'
```

3. Using that *vpc-id*, find the VPC's CIDR.

```
aws ec2 describe-vpcs \
--vpc vpc-id \
--query 'Vpcs[0].CidrBlock'
```

4. From account B, create a VPC using a different, non-overlapping CIDR than the one found in the previous step. Then, create at least one subnet. You can use either the [VPC creation wizard](#) in the AWS Management Console or the [AWS CLI](#).
5. From account B, request a peering connection to the account A VPC as described in [Creating and Accepting a VPC Peering Connection](#). From account A, accept the connection.
6. From account B, find the new VPC's routing table. Replace *vpc-id* with the ID of the VPC you created in account B.

```
aws ec2 describe-route-tables \
--filters 'Name=vpc-id,Values=vpc-id' \
--query 'RouteTables[0].RouteTableId'
```

7. Add a route to send traffic destined for account A's CIDR to the VPC peering connection. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \
--route-table-id accountB-route-table-id \
--destination-cidr accountA-vpc-cidr \
--vpc-peering-connection-id peering-connection-id
```

8. From account A, find the DAX cluster's route table using the *vpc-id* you found previously.

```
aws ec2 describe-route-tables \
--filters 'Name=vpc-id, Values=accountA-vpc-id' \
--query 'RouteTables[0].RouteTableId'
```

9. From account A, add a route to send traffic destined for account B's CIDR to the VPC peering connection. Replace each *user input placeholder* with the correct values for your accounts.

```
aws ec2 create-route \
--route-table-id accountA-route-table-id \
--destination-cidr accountB-vpc-cidr \
--vpc-peering-connection-id peering-connection-id
```

10. From account B, launch an EC2 instance in the VPC that you created earlier. Give it the `AssumeDaxInstanceProfile`. You can use either the [launch wizard](#) in the AWS Management Console or the [AWS CLI](#). Take note of the instance's security group.
11. From account A, find the security group used by the DAX cluster. Remember to replace *cluster-name* with the name of your DAX cluster.

```
aws dax describe-clusters \
```

```
--cluster-name cluster-name \  
--query 'Clusters[0].SecurityGroups[0].SecurityGroupIdentifier'
```

12. Update the DAX cluster's security group to allow inbound traffic from the security group of the EC2 instance you created in account B. Remember to replace the *user input placeholders* with the correct values for your accounts.

```
aws ec2 authorize-security-group-ingress \  
--group-id accountA-security-group-id \  
--protocol tcp \  
--port 8111 \  
--source-group accountB-security-group-id \  
--group-owner accountB-id
```

At this point, an application on account B's EC2 instance is able to use the instance profile to assume the `arn:aws:iam::accountA-id:role/DaxCrossAccountRole` role and use the DAX cluster.

Modify the DAX Client to Allow Cross-account Access

Note

AWS Security Token Service (AWS STS) credentials are temporary credentials. Some clients handle refreshing automatically, while others require additional logic to refresh the credentials. We recommend that you follow the guidance of the appropriate documentation.

Java

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Java and DAX \(p. 688\)](#) tutorial.

1. Add the following imports.

```
import com.amazonaws.auth.STSSummarySessionCredentialsProvider;  
import com.amazonaws.services.securitytoken.AWSIdentityTokenService;  
import com.amazonaws.services.securitytoken.AWSIdentityTokenServiceClientBuilder;
```

2. Get a credentials provider from AWS STS and create a DAX client object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```
AWSIdentityTokenService awsIdentityTokenService =  
    AWSIdentityTokenServiceClientBuilder  
        .standard()  
        .withRegion(region)  
        .build();  
  
STSSummarySessionCredentialsProvider credentials = new  
    STSSummarySessionCredentialsProvider.Builder("arn:aws:iam::accountA:role/  
    RoleName", "TryDax")  
        .withSTSClient(awsIdentityTokenService)  
        .build();  
  
DynamoDB client = AmazonDAXClientBuilder.standard()  
    .withRegion(region)  
    .withEndpointConfiguration(dax_endpoint)  
    .withCredentials(credentials)  
    .build();
```

.NET

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [.NET and DAX \(p. 697\)](#) tutorial.

1. Add the [AWSSDK.SecurityToken](#) NuGet package to the solution.

```
<PackageReference Include="AWSSDK.SecurityToken" Version="latest version" />
```

2. Use the `SecurityToken` and `SecurityToken.Model` packages.

```
using Amazon.SecurityToken;
using Amazon.SecurityToken.Model;
```

3. Get temporary credentials from `AmazonSimpleTokenService` and create a `ClusterDaxClient` object. Remember to replace each `user input placeholder` with the correct values for your accounts.

```
IAmazonSecurityTokenService sts = new AmazonSecurityTokenServiceClient();

var assumeRoleResponse = sts.AssumeRole(new AssumeRoleRequest
{
    RoleArn = "arn:aws:iam::accountA:role/RoleName",
    RoleSessionName = "TryDax"
});

Credentials credentials = assumeRoleResponse.Credentials;

var clientConfig = new DaxClientConfig(dax_endpoint, port)
{
    AwsCredentials = assumeRoleResponse.Credentials
};

var client = new ClusterDaxClient(clientConfig);
```

Go

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find [working code examples on GitHub](#).

1. Import the AWS STS and session packages.

```
import (
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/sts"
    "github.com/aws/aws-sdk-go/aws/credentials/stscreds"
)
```

2. Get temporary credentials from `AmazonSimpleTokenService` and create a DAX client object. Remember to replace each `user input placeholder` with the correct values for your accounts.

```
sess, err := session.NewSession(&aws.Config{
    Region: aws.String(region)),
}
if err != nil {
    return nil, err
}
```

```

stsClient := sts.New(sess)
arp := &stscreds.AssumeRoleProvider{
    Duration:    900 * time.Second,
    ExpiryWindow: 10 * time.Second,
    RoleARN:      "arn:aws:iam::accountA:role/role_name",
    Client:       stsClient,
    RoleSessionName: "session_name",
}
cfg := dax.DefaultConfig()

cfg.HostPorts = []string{dax_endpoint}
cfg.Region = region
cfg.Credentials = credentials.NewCredentials(arp)
daxClient := dax.New(cfg)

```

Python

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Python and DAX \(p. 713\)](#) tutorial.

1. Import boto3.

```
import boto3
```

2. Get temporary credentials from sts and create an AmazonDaxClient object. Remember to replace each *user input placeholder* with the correct values for your accounts.

```

sts = boto3.client('sts')
stsresponse =
    sts.assume_role(RoleArn='arn:aws:iam::accountA:role/RoleName',RoleSessionName='tryDax')
credentials = botocore.session.get_session()['Credentials']

dax = amazondax.AmazonDaxClient(session, region_name=region,
    endpoints=[dax_endpoint], aws_access_key_id=credentials['AccessKeyId'],
    aws_secret_access_key=credentials['SecretAccessKey'],
    aws_session_token=credentials['SessionToken'])
client = dax

```

Node.js

This section helps you modify your existing DAX client code to allow cross-account DAX access. If you don't have DAX client code already, you can find working code examples in the [Node.js and DAX \(p. 706\)](#) tutorial. Remember to replace each *user input placeholder* with the correct values for your accounts.

```

const AmazonDaxClient = require('amazon-dax-client');
const AWS = require('aws-sdk');
const region = 'region';
const endpoints = [daxEndpoint1, ...];

const getCredentials = async() => {
    return new Promise((resolve, reject) => {
        const sts = new AWS.STS();
        const roleParams = {
            RoleArn: 'arn:aws:iam::accountA:role/RoleName',
            RoleSessionName: 'tryDax',
        };
        sts.assumeRole(roleParams, (err, session) => {

```

```
        if(err) {
          reject(err);
        } else {
          resolve({
            accessKeyId: session.Credentials.AccessKeyId,
            secretAccessKey: session.Credentials.SecretAccessKey,
            sessionToken: session.Credentials.SessionToken,
          });
        }
      });
    });
  };
}

const createDaxClient = async() => {
  const credentials = await getCredentials();
  const daxClient = new AmazonDaxClient({endpoints: endpoints, region: region,
  accessKeyId: credentials.accessKeyId, secretAccessKey: credentials.secretAccessKey,
  sessionToken: credentials.sessionToken});
  return new AWS.DynamoDB.DocumentClient({service: daxClient});
};

createDaxClient().then((client) => {
  client.get(...);
  ...
}).catch((error) => {
  console.log('Caught an error: ' + error);
});
```

DAX Cluster Sizing Guide

This guide provides advice for choosing an appropriate Amazon DynamoDB Accelerator (DAX) cluster size and node type for your application. These instructions guide you through the steps of estimating your application's DAX traffic, selecting a cluster configuration, and testing it.

If you have an existing DAX cluster and want to evaluate whether it has the appropriate number and size of nodes, please refer to [Scaling a DAX Cluster \(p. 728\)](#).

Topics

- [Overview \(p. 766\)](#)
- [Estimating Traffic \(p. 767\)](#)
- [Load Testing \(p. 767\)](#)

Overview

It's important to scale your DAX cluster appropriately for your workload, whether you're creating a new cluster or maintaining an existing cluster. As time goes on and your application's workload changes, you should periodically revisit your scaling decisions to make sure that they are still appropriate.

The process typically follows these steps:

1. **Estimating traffic.** In this step, you make predictions about the volume of traffic that your application will send to DAX, the nature of the traffic (read vs. write operations), and the expected cache hit rate.
2. **Load testing.** In this step, you create a cluster and send traffic to it mirroring your estimates from the previous step. Repeat this step until you find a suitable cluster configuration.
3. **Production monitoring.** While your application is using DAX in production, you should [monitor the cluster \(p. 732\)](#) to continuously validate that it is still scaled correctly as your workload changes over time.

Estimating Traffic

There are three main factors that characterize a typical DAX workload:

- Cache hit rate
- [Read capacity units \(p. 341\)](#) (RCUs) per second
- [Write capacity units \(p. 342\)](#) (WCUs) per second

Estimating Cache Hit Rate

If you already have a DAX cluster, you can use the `ItemCacheHits` and `ItemCacheMisses` [Amazon CloudWatch metrics \(p. 734\)](#) to determine the cache hit rate. The cache hit rate is equal to $\text{ItemCacheHits} / (\text{ItemCacheHits} + \text{ItemCacheMisses})$. If your workload includes `Query` or `Scan` operations, you should also look at the `QueryCacheHits`, `QueryCacheMisses`, `ScanCacheHits`, and `ScanCacheMisses` metrics. Cache hit rates vary from one application to another and are heavily influenced by the cluster's Time to Live (TTL) setting. Typical hit rates for applications using DAX are 85–95 percent.

Estimating Read and Write Capacity Units

If you already have DynamoDB tables for your application, look at the `ConsumedReadCapacityUnits` and `ConsumedWriteCapacityUnits` [CloudWatch metrics \(p. 734\)](#). Use the `Sum` statistic and divide by the number of seconds in the period.

If you also already have a DAX cluster, remember that the DynamoDB `ConsumedReadCapacityUnits` metric only accounts for cache misses. So, to get an idea of the read capacity units per second handled by your DAX cluster, divide the number by your cache miss rate (that is, $1 - \text{cache hit rate}$).

If you don't already have a DynamoDB table, see the documentation about [read capacity units \(p. 341\)](#) and [write capacity units \(p. 342\)](#) to estimate your traffic based on your application's estimated request rate, items accessed per request, and item size.

When making traffic estimates, plan for future growth and for expected and unexpected peaks to ensure that your cluster has enough headroom for traffic increases.

Load Testing

The next step after estimating traffic is to test the cluster configuration under load.

1. For your initial load test, we recommend that you start with the `dax.r4.large` node type, the lowest-cost fixed performance, memory-optimized node type.
2. A fault-tolerant cluster requires at least three nodes, spread across three Availability Zones. In this case, if an Availability Zone becomes unavailable, the effective number of Availability Zones is reduced by one-third. For your initial load test, we recommend that you start with a two-node cluster, which simulates the failure of one Availability Zone in a three-node cluster.
3. Drive sustained traffic (as estimated in the previous step) to your test cluster for the duration of the load test.
4. Monitor the performance of the cluster during the load test.

Ideally, the traffic profile that you drive during the load test should be as similar as possible to your application's real traffic. This includes the distribution of operations (for example, 70 percent `GetItem`, 25 percent `Query`, and 5 percent `PutItem`), the request rate for each operation, the number of items accessed per request, and the distribution of item sizes. To achieve a cache hit rate similar to your application's expected cache hit rate, pay close attention to the distribution of keys in your test traffic.

Note

Be careful when load testing T2 node types (`dax.t2.small` and `dax.t2.medium`). T2 node types provide [burstable CPU performance](#) that varies over time depending on the node's CPU credit balance. A DAX cluster running on T2 nodes might appear to be operating normally, but if any node is bursting above the [baseline performance](#) of its instance, the node is spending its accrued CPU credit balance. When the credit balance runs low, [performance is gradually lowered](#) to the baseline performance level.

[Monitor your DAX cluster \(p. 732\)](#) during the load test to determine whether the node type that you're using for the load test is the right node type for you. In addition, during a load test, you should monitor your request rate and cache hit rate to ensure that your test infrastructure is actually driving the amount of traffic you intend.

If load testing indicates that the selected cluster configuration can't sustain your application's workload, we recommend [switching to a larger node type \(p. 728\)](#), especially if you see high CPU utilization on the primary node in the cluster or high eviction rates. If hit rates are consistently high, and the ratio of read to write traffic is high, you may want to consider [adding more nodes to your cluster \(p. 728\)](#). Refer to [Scaling a DAX Cluster \(p. 728\)](#) for additional guidance on when to use a larger node type (vertical scaling) or add more nodes (horizontal scaling).

You should repeat your load test after making changes to your cluster configuration.

DAX API Reference

For more information about Amazon DynamoDB Accelerator (DAX) APIs, see [Amazon DynamoDB Accelerator](#) in the Amazon DynamoDB API Reference.

NoSQL Workbench for Amazon DynamoDB

NoSQL Workbench for Amazon DynamoDB is a cross-platform client-side application for modern database development and operations and is available for Windows and macOS. NoSQL Workbench is a unified visual tool that provides data modeling, data visualization, and query development features to help you design, create, query, and manage DynamoDB tables.

Data Modeling

With NoSQL Workbench for DynamoDB, you can build new data models from, or design models based on, existing data models that satisfy your application's data access patterns. You can also import and export the designed data model at the end of the process. For more information, see [Building Data Models with NoSQL Workbench \(p. 770\)](#).

Data Visualization

The data model visualizer provides a canvas where you can map queries and visualize the access patterns (facets) of the application without having to write code. Every facet corresponds to a different access pattern in DynamoDB. You can manually add data to your data model or import data from MySQL. For more information, see [Visualizing Data Access Patterns \(p. 779\)](#).

Operation Building

NoSQL Workbench provides a rich graphical user interface for you to develop and test queries. You can use the *operation builder* to view, explore, and query datasets. You can also use the structured operation builder to build and perform data plane operations. It supports projection and condition expression, and lets you generate sample code in multiple languages. For more information, see [Exploring Datasets and Building Operations with NoSQL Workbench \(p. 785\)](#).

Topics

- [Download NoSQL Workbench \(p. 769\)](#)
- [Building Data Models with NoSQL Workbench \(p. 770\)](#)
- [Visualizing Data Access Patterns \(p. 779\)](#)
- [Exploring Datasets and Building Operations with NoSQL Workbench \(p. 785\)](#)
- [Sample Data Models for NoSQL Workbench \(p. 798\)](#)
- [Release History for NoSQL Workbench \(p. 800\)](#)

Download NoSQL Workbench

Follow these instructions to download and install NoSQL Workbench for Amazon DynamoDB.

To download and install NoSQL Workbench

1. Use one of the following links to download NoSQL Workbench for free.

| Operating system | Download link | Checksum link |
|------------------|------------------------------------|--------------------------|
| macOS | Download for macOS | Checksum |

| Operating system | Download link | Checksum link |
|------------------|--------------------------------------|--------------------------|
| Windows | Download for Windows | Checksum |
| Linux* | Download for Linux | Checksum |

* NoSQL Workbench supports Ubuntu 12.04 , Fedora 21, and Debian 8 or any newer versions of these Linux distributions.

2. Start the application that you downloaded, and follow the onscreen instructions to install NoSQL Workbench.

Building Data Models with NoSQL Workbench

You can use the data modeler tool in NoSQL Workbench for Amazon DynamoDB to build new data models, or to design models based on existing data models that satisfy your applications' data access patterns. The data modeler includes a few sample data models to help you get started.

Topics

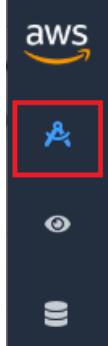
- [Creating a New Data Model \(p. 770\)](#)
- [Importing an Existing Data Model \(p. 774\)](#)
- [Exporting a Data Model \(p. 775\)](#)
- [Editing an Existing Data Model \(p. 776\)](#)

Creating a New Data Model

Follow these steps to create a new data model in Amazon DynamoDB using NoSQL Workbench.

To create a new data model

1. Open NoSQL Workbench, and in the navigation pane on the left side, choose the **Data modeler** icon.



2. Choose **Create data model**.



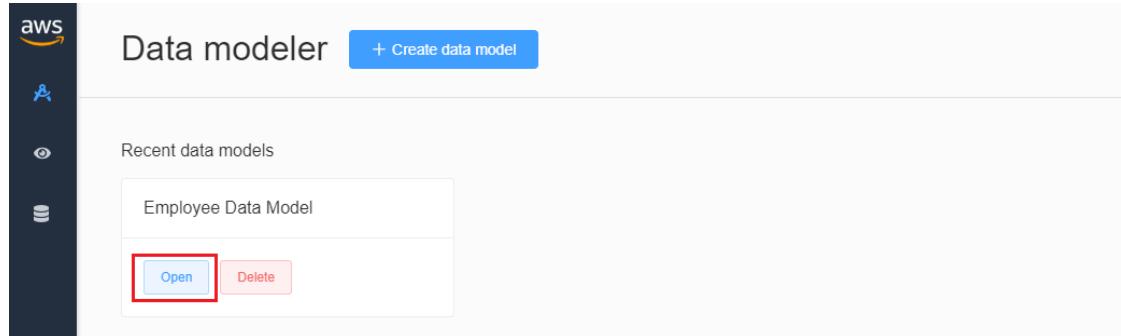
3. Enter a name for the data model, and then choose **Create**.

Create data model

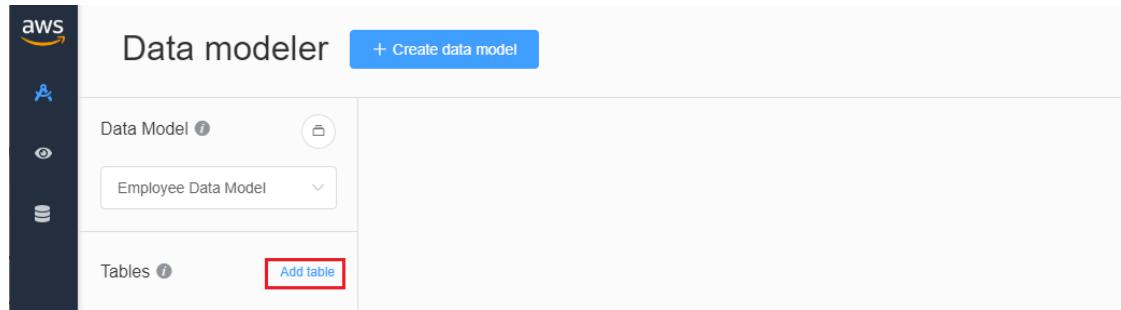
* Name Employee Data Model

Cancel Create

4. On the newly created model, choose **Open**.



5. Choose **Add table**.



For more information about tables, see [Working with Tables in DynamoDB](#).

6. Specify the following:

- **Table name** — Enter a unique name for the table.
- **Partition key** — Enter a partition key name and specify its type.
- If you want to add a sort key:
 1. Select **Add sort key**.
 2. Specify the sort key name and its type.
 - 3.

* Table name

Employee



Primary key attributes

* Partition key

LoginAlias

String



Add sort key

7. To add other attributes, do the following for each attribute:

1. Choose **Add other attribute**.
2. Specify the attribute name and type.

Other attributes

* Attribute name

FirstName

String



* Attribute name

LastName

String



* Attribute name

ManagerLoginAlias

String



* Attribute name

Designation

String



* Attribute name

Skills

String Set



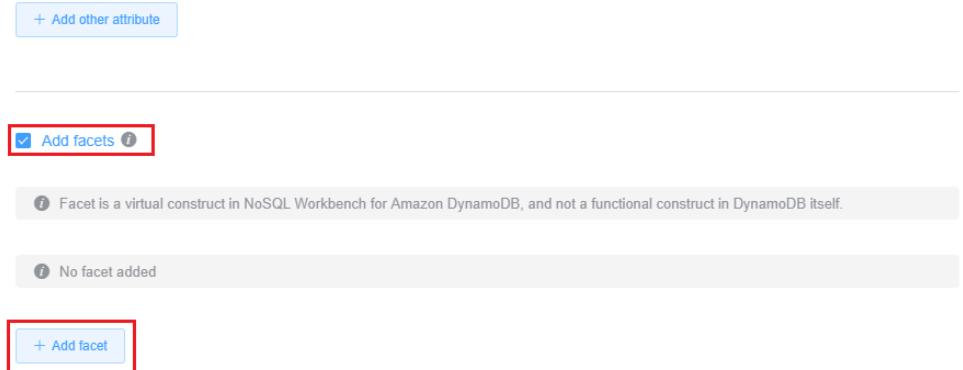
+ Add other attribute

8. Add a facet:

Note

Facets represent an application's different data access patterns for DynamoDB.

- Select **Add facets**.
- Choose **Add facet**.



- Specify the following:
 - The **Facet name**.
 - A **Partition key alias**.
 - A **Sort key alias**.
 - Choose the **Other attributes** that are part of this facet.

Choose **Add facet**.

Add facet

* Facet name: SongDetails

* Partition key alias: SongId

* Sort key alias: Metadata

Other attributes: Title, Artist, TotalDownloads

Cancel Add facet

Repeat step 8 if you want to add more facets.

9. If you want to add a global secondary index, choose **Add global secondary index**.

Specify the **Global secondary index name**, **Partition key attribute**, and **Projection type**.

Global secondary indexes

Global secondary index name

* Partition key FirstName

Add sort key LastName

Projection type ALL

+ Add global secondary index

For more information about working with global secondary indexes in DynamoDB, see [Global Secondary Indexes](#).

10. Choose **Add table definition**.

Global secondary indexes

+ Add global secondary index

Cancel

Add table definition

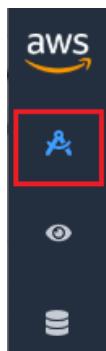
For more information about the `CreateTable` API operation, see [CreateTable](#) in the *Amazon DynamoDB API Reference*.

Importing an Existing Data Model

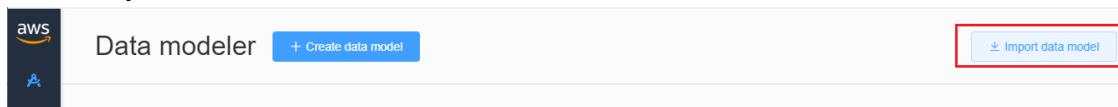
You can use NoSQL Workbench for Amazon DynamoDB to build a data model by importing and modifying an existing model.

To import a data model

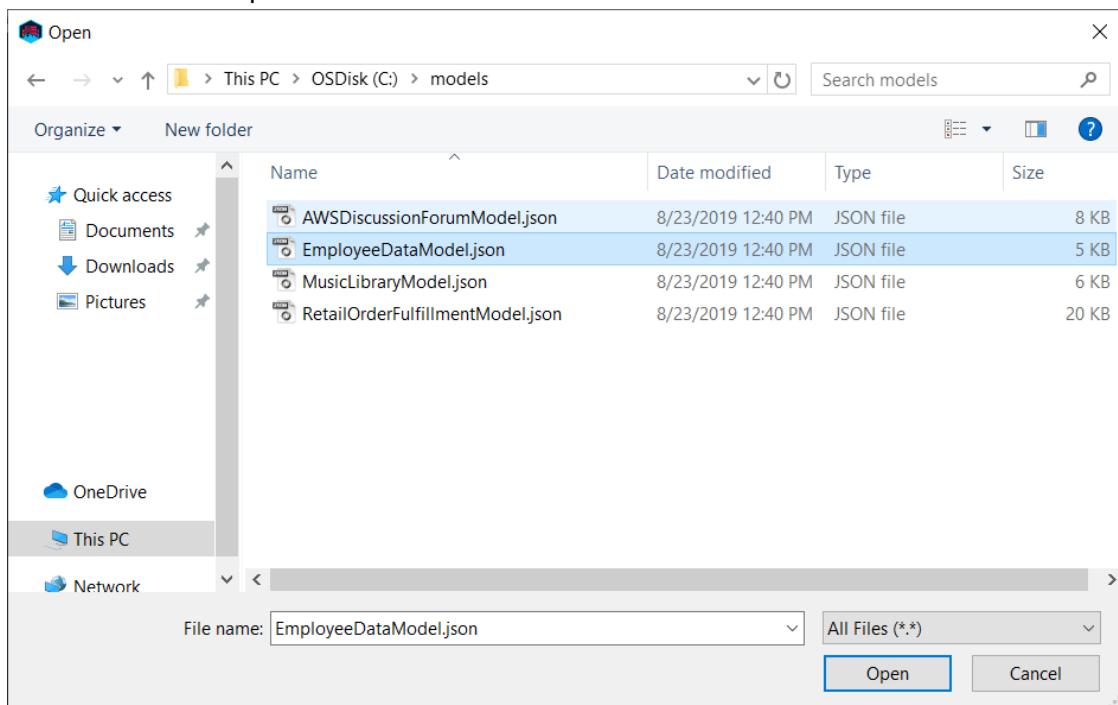
1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** icon.



2. Choose **Import data model**.



3. Choose a model to import.

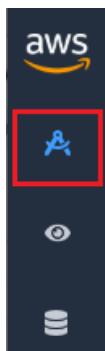


Exporting a Data Model

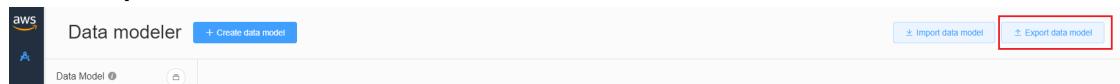
After you create a data model using NoSQL Workbench for Amazon DynamoDB, you can save and export the model.

To export a data model

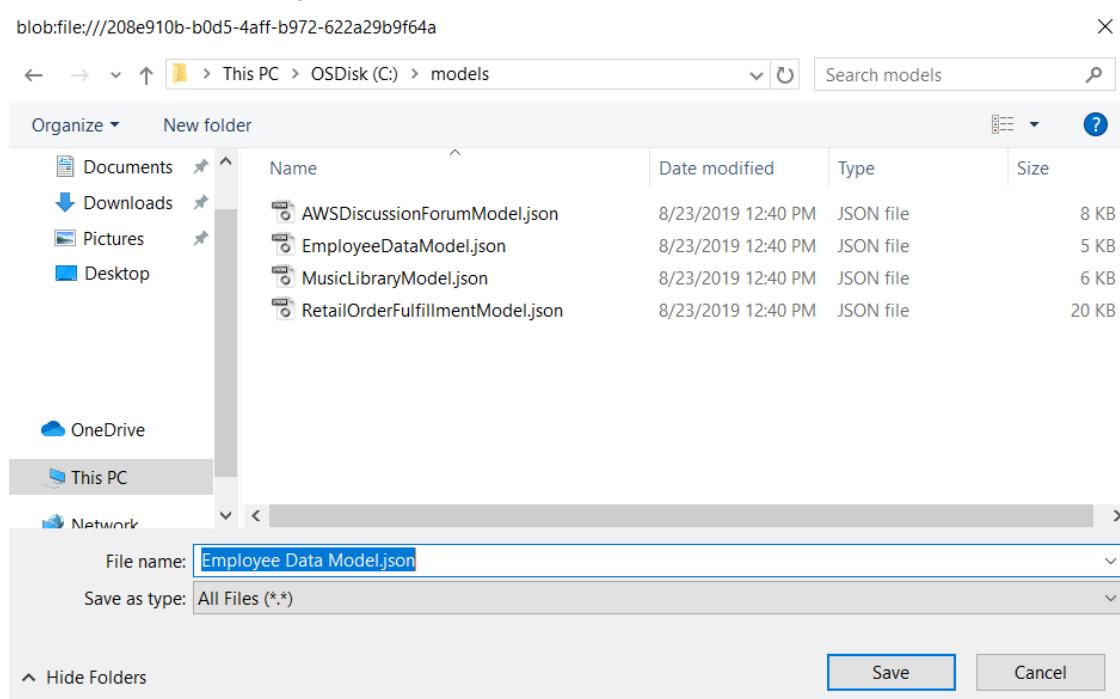
1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** icon.



2. Choose **Export data model**.



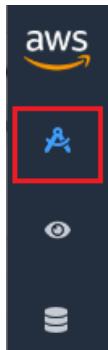
3. Choose a location to save your model.



Editing an Existing Data Model

To edit an existing model

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Data modeler** button.

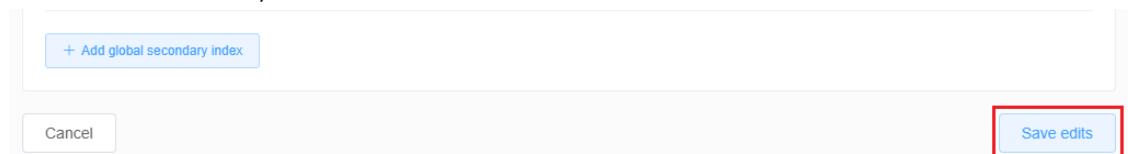


2. Choose the table you wish to edit.

Choose **Edit**.



3. Make the needed edits, and then choose **Save edits**.



To manually edit an existing model and add a facet

1. Export your model. For more information, see [Exporting a Data Model \(p. 775\)](#).
2. Open the exported file in an editor.
3. Locate the `DataModel` Object for the table you want to create a facet for.

Add a `TableFacets` array representing all the facets for the table.

For each facet add an object to the `TableFacets` array. Each array element has the following properties:

- `FacetName` — A name for your facet. This value must be unique across the model.
- `PartitionKeyAlias` — A friendly name for the table's partition key. This alias is displayed when you view the facet in NoSQL Workbench.
- `SortKeyAlias` — A friendly name for the table's sort key. This alias is displayed when you view the facet in NoSQL Workbench. This property is not needed if the table has no sort key defined.
- `NonKeyAttributes` — An array of attribute names that are needed for the access pattern. These names must map to the attribute names that are defined for the table.

```
{  
  "modelName": "Music Library Data Model",  
  "dataModel": [  
    {  
      "tableName": "Songs",  
      "partitionKey": "SongID",  
      "sortKey": "SongTitle",  
      "nonKeyAttributes": ["ArtistName", "SongLength"]  
    }  
  ]  
}
```

```
"KeyAttributes": {
    "PartitionKey": {
        "AttributeName": "Id",
        "AttributeType": "S"
    },
    "SortKey": {
        "AttributeName": "Metadata",
        "AttributeType": "S"
    }
},
"NonKeyAttributes": [
    {
        "AttributeName": "DownloadMonth",
        "AttributeType": "S"
    },
    {
        "AttributeName": "TotalDownloadsInMonth",
        "AttributeType": "S"
    },
    {
        "AttributeName": "Title",
        "AttributeType": "S"
    },
    {
        "AttributeName": "Artist",
        "AttributeType": "S"
    },
    {
        "AttributeName": "TotalDownloads",
        "AttributeType": "S"
    },
    {
        "AttributeName": "DownloadTimestamp",
        "AttributeType": "S"
    }
],
"TableFacets": [
    {
        "FacetName": "SongDetails",
        "KeyAttributeAlias": {
            "PartitionKeyAlias": "SongId",
            "SortKeyAlias": "Metadata"
        },
        "NonKeyAttributes": [
            "Title",
            "Artist",
            "TotalDownloads"
        ]
    },
    {
        "FacetName": "Downloads",
        "KeyAttributeAlias": {
            "PartitionKeyAlias": "SongId",
            "SortKeyAlias": "Metadata"
        },
        "NonKeyAttributes": [
            "DownloadTimestamp"
        ]
    }
]
```

4. You can now import the modified model into NoSQL Workbench. For more information, see [Importing an Existing Data Model \(p. 774\)](#).

Visualizing Data Access Patterns

You can use the visualizer tool in NoSQL Workbench for Amazon DynamoDB to map queries and visualize different access patterns (known as *facets*) of an application. Every facet corresponds to a different access pattern in DynamoDB. You can also manually add data to your data model or import data from MySQL.

Topics

- [Adding Sample Data to a Data Model \(p. 779\)](#)
- [Viewing Data Access Patterns \(p. 780\)](#)
- [Viewing All Tables in a Data Model Using Aggregate View \(p. 781\)](#)
- [Committing a Data Model to DynamoDB \(p. 782\)](#)

Adding Sample Data to a Data Model

By adding sample data to your model, you can display data when visualizing the model and its various data access patterns, or *facets*.

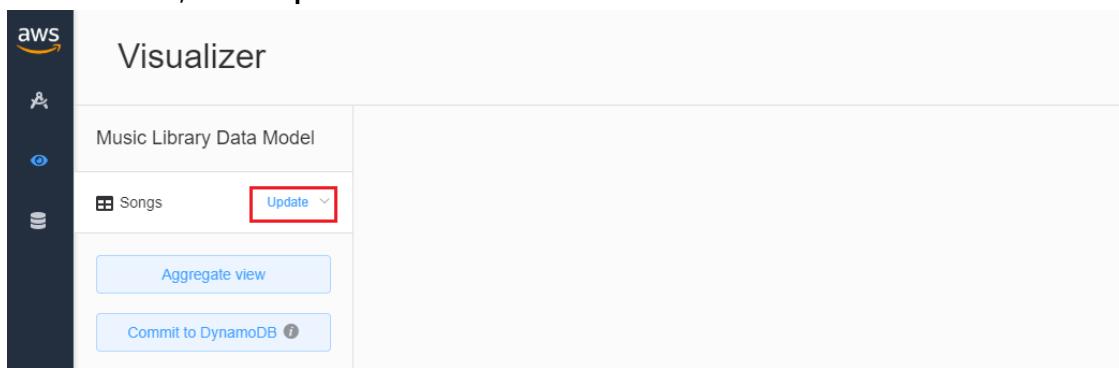
Follow these steps to add sample data to a data model using NoSQL Workbench for Amazon DynamoDB.

To add sample data

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the visualizer, choose **Update** next to the table name.



3. Choose **Add Data**.

[Import data model](#)
[Export data model](#)

FACET
Songs
Add data
Edit data

| Id (Partition key) : String ▾ | Metadata (Sort key) : String ▾ | DownloadMonth : String ▾ | TotalDownloadsIn Month : String ▾ | Title : String ▾ | Artist : String ▾ | To | Str |
|----------------------------------|--------------------------------|--------------------------|-----------------------------------|------------------|-------------------|----|-----|
| | | | | | | | |

4. Enter the sample data into the empty textboxes, and then choose **Save**.

[Import data model](#)
[Export data model](#)

FACET
Songs
Save
Cancel

| Id (Partition key) : String ▾ | Metadata (Sort key) : String ▾ | DownloadMonth : String ▾ | TotalDownloadsIn Month : String ▾ | Title : String ▾ | Artist : String ▾ | TotalDownloads : String ▾ | DownloadTimestamp : String ▾ |
|----------------------------------|--------------------------------|--------------------------|-----------------------------------|------------------|-------------------|---------------------------|------------------------------|
| 1 | ACME Album | 01-2018 | 3 | Attribute value | Attribute value | Attribute value | Attribute value |

Viewing Data Access Patterns

In NoSQL Workbench, *facets* represent an application's different data access patterns for Amazon DynamoDB.

To view information about facets in NoSQL Workbench

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the data model on the left side, choose a table to view.
 3. Choose the **Facets** drop-down arrow for the selected table.
 4. In the list, choose a facet to view.

| SongId (PK) ▾ | Metadata (SK) ▾ | Title ▾ | Artist ▾ | TotalDownloads ▾ |
|---------------|-----------------|---------------|-------------|------------------|
| 1 | Details | Wild Love | Argyboots | 3 |
| 2 | Details | Funky Drummer | James Brown | 4 |

You can also edit the facet definitions using the Data Modeler. For more information, see [Editing an Existing Data Model \(p. 776\)](#).

Viewing All Tables in a Data Model Using Aggregate View

The aggregate view in NoSQL Workbench for Amazon DynamoDB represents all the tables in a data model. For each table, the following information appears:

- Table column names.
- Sample data.
- All global secondary indexes that are associated with the table. The following information is displayed for each index:
 - Index column names
 - Sample data

To view all table information

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. In the visualizer, choose **Aggregate view**.

The screenshot shows the AWS Visualizer interface for a 'Music Library Data Model'. On the left, there's a navigation pane with icons for AWS, a magnifying glass, and a circular icon. The circular icon is highlighted with a red box. The main area displays an 'AGGREGATED VIEW' for the 'Songs' table. It shows a primary key structure with 'Partition key: Id' and 'Sort key: Metadata'. Below this, there are three rows of data: 'Wild Love' (DownloadTimestamp: 2018-01-01T00:00:07), 'Did-9349823681' (DownloadTimestamp: 2018-01-01T00:01:08), and 'Did-9349823682' (DownloadTimestamp: 2018-01-01T00:01:08). At the bottom of the interface, there are 'Import data model' and 'Export Data Model' buttons, and a 'Commit to DynamoDB' button which is also highlighted with a red box.

Committing a Data Model to DynamoDB

When you are satisfied with your data model, you can commit the model to Amazon DynamoDB.

Note

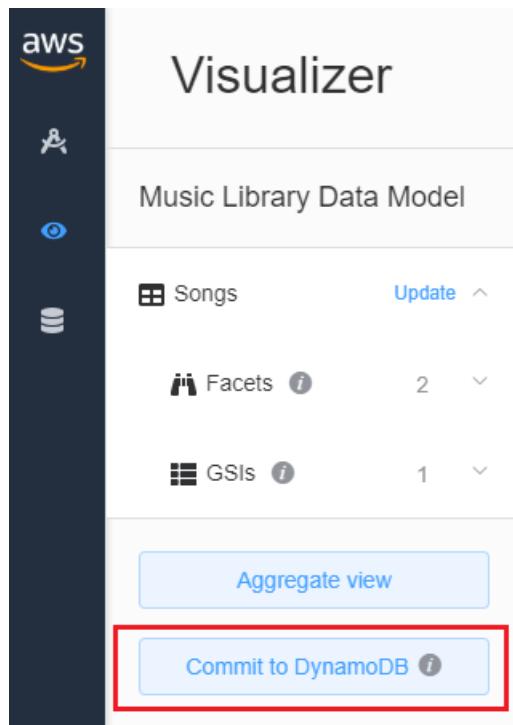
- This action results in the creation of server-side resources in AWS for the tables and global secondary indexes represented in the data model.
- Tables are created with the following characteristics:
 - Auto scaling is set to 70 percent target utilization.
 - Provisioned capacity is set to 5 read capacity units and 5 write capacity units.
- Global secondary indexes are created with provisioned capacity of 10 read capacity units and 5 write capacity units.

To commit the data model to DynamoDB

1. In the navigation pane on the left side, choose the **visualizer** icon.



2. Choose **Commit to DynamoDB**.



3. Choose an already existing connection, or create a new connection by choosing the **Add new remote connection** tab.
 - To add a new connection, specify the following information:
 - **Account Alias**
 - **AWS Region**
 - **Access key ID**
 - **Secret access key**
 - For more information about how to obtain the access keys, see [Getting an AWS Access Key](#).
 - You can optionally specify the following:
 - **Session token**
 - **IAM role ARN**
 - If you don't want to sign up for a free tier account, and prefer to use [DynamoDB Local \(Downloadable Version\)](#):
 1. Choose the **Add a new DynamoDB local connection** tab
 2. Specify the **Connection name** and **Port**.
4. Choose **Commit**.

Commit to DynamoDB

i On this page, you create server-side resources such as tables and global secondary indexes for the selected data model. By default, all tables and indexes are created with a read capacity of five read capacity units and a write capacity of five write capacity units.

< Add connections [Add a new remote connection](#) [Add a new DynamoDB local connection](#) >

Connection name

Connection name

* AWS Region

AWS Region

* Access key ID

AWS access key ID

* Secret access key

AWS secret access key

Session token

AWS session token

i

IAM role ARN

IAM role ARN

i

Persist connection

If you select this check box, AWS connection secrets will be persisted in

C:\Users\michael.m\aws\credentials

[Cancel](#)

[Reset](#)

[Commit](#)

Exploring Datasets and Building Operations with NoSQL Workbench

NoSQL Workbench for Amazon DynamoDB provides a rich graphical user interface for developing and testing queries. You can use the operation builder in NoSQL Workbench to view, explore, and query datasets. You can also use the structured operation builder to build and perform data plane operations, with support for projection and condition expression, and generate sample code in multiple languages.

Topics

- [Exploring Datasets \(p. 785\)](#)
- [Building Complex Operations \(p. 789\)](#)

Exploring Datasets

To explore your Amazon DynamoDB tables, you first need to connect to your AWS account.

To add a connection to your database

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Operation builder** icon.



2. Choose **Add connection**.



3. Specify the following information:

- **Account alias**
- **AWS Region**
- **Access key ID**
- **Secret access key**

For more information about how to obtain the access keys, see [Getting an AWS Access Key](#).

You can optionally, specify the following:

- **Session token**
- **IAM role ARN**

4. Choose **Connect**.

Add a new database connection



A remote connection lets you access the DynamoDB web service in different AWS Regions.

A DynamoDB local connection lets you access the DynamoDB local server running on your computer.

[Remote](#)

DynamoDB local

* Connection name

Connection 3

* Default AWS Region

Default AWS Region

* Access key ID

AWS Access key ID

* Secret access key

AWS secret access key

Session token

AWS session token



IAM role ARN

IAM role ARN



Persist connection

If you select this check box, AWS connection secrets will be persisted in

C:\Users\m*****ml.aws\credentials

[Cancel](#)

[Reset](#)

[Connect](#)

If you don't want to sign up for a free tier account, and prefer to use [DynamoDB Local \(Downloadable Version\)](#):

- a. Choose the **Local** tab on the connection screen
- b. Specify the following information:
 - **Connection name**
 - **Port**
- c. Choose the **connect** button.

Add a new database connection

i A remote connection can let you interact with DynamoDB servers in different regions.
A local connection can let you interact with the DynamoDB Local server on your machine.

Remote **Local**

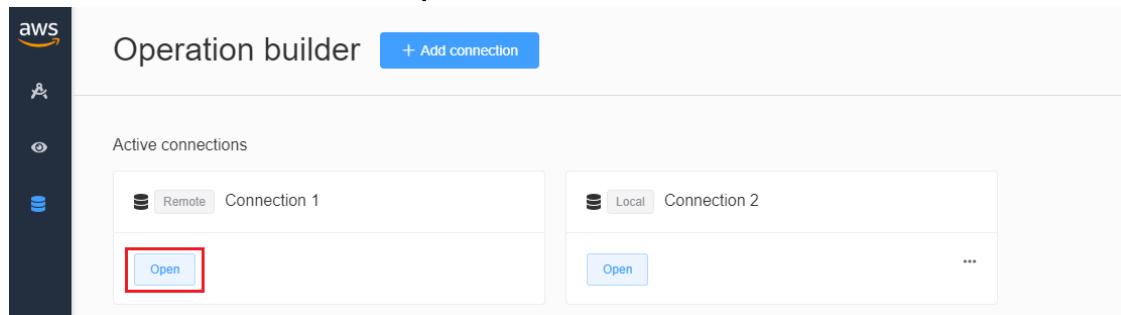
i Please setup the dynamoDB local server on your machine before adding a connection.
[Setting Up DynamoDB Local](#)

* Connection name

* Hostname

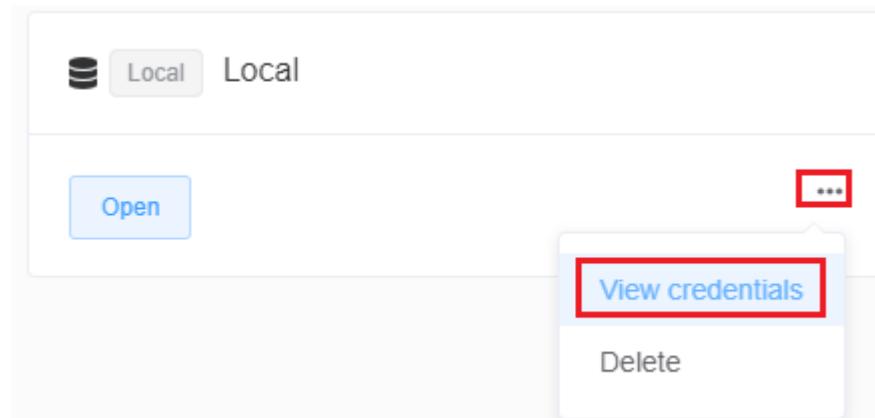
* Port

5. On the created connection, choose **Open**.



Note

- When adding a connection to [DynamoDB Local \(Downloadable Version\)](#), NoSQL Workbench generates a random AWS access key ID and secret access key. To retrieve the values of these keys, choose **View credentials** from the connection context menu.



- When adding a connection to [DynamoDB Local \(Downloadable Version\)](#), the NoSQL Workbench uses localhost as the Region for the DynamoDB Local connection.
- If [DynamoDB Local \(Downloadable Version\)](#) is launched without the `-sharedDb` command line option, a new database file is created in DynamoDB Local for each connection from NoSQL Workbench.
- You can't use an existing [DynamoDB Local \(Downloadable Version\)](#) database (created via AWS CLI/SDK) via NoSQL Workbench, unless the DynamoDB Local instance was started with the `-sharedDb` command line option.

After connecting to your DynamoDB database, the list of available tables appears in the left pane. Choose one of the tables to return a sample of the data stored in the table.

You can now execute queries against the selected table.

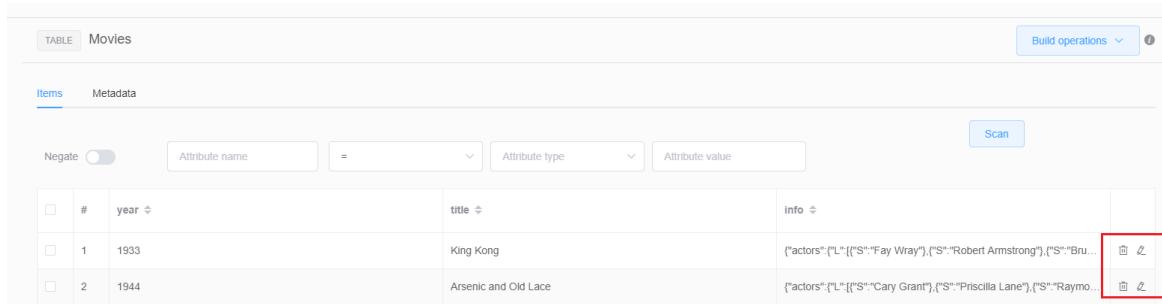
To execute queries on a table

- In the **Attribute name** list, choose the attribute that you want to query on.
- Specify the comparison operator.
- Specify the data type of the value.
- Specify the value to query for.
- Choose **Scan**.

| year | info | title |
|------|-----------|--|
| 1933 | King Kong | {"actors": [{"S": "Fay Wray"}, {"S": "Robert ..."}]} |

For more information about this operation, see [Scan](#) in the *Amazon DynamoDB API Reference*.

You can update and delete any row that is returned. To update, choose the **Edit** (pencil) icon on the right side of the row. To delete, choose the **Delete** (trash can) icon.



The screenshot shows the 'Movies' table in the Amazon DynamoDB console. The table has columns: #, year, title, and info. There are three items listed:

- # 1, year 1933, title King Kong, info (actors: ["L": {"S": "Fay Wray"}, {"S": "Robert Armstrong"}, {"S": "Bruce Cabot"}])
- # 2, year 1944, title Arsenic and Old Lace, info (actors: ["L": {"S": "Cary Grant"}, {"S": "Priscilla Lane"}, {"S": "Raymond Massey"}])

Choosing the Delete icon starts a [Delete Item \(p. 793\)](#) operation.

DeleteItem operation builder

| | |
|---|-----------|
| * Partition key | 1933 |
| * Sort key | King Kong |
| Condition expression | |
| + Condition + Child expression | |
| <input type="button" value="Cancel"/> <input type="button" value="Clear form"/> <input type="button" value="Execute"/> <input type="button" value="Generate code"/> | |

Choosing the Edit icon starts an [Update Item \(p. 792\)](#) operation.

UpdateItem operation builder

| | |
|---|-----------|
| * Partition key | 1933 |
| * Sort key | King Kong |
| Update expression | |
| Select Operation + | |
| Condition expression | |
| + Condition + Child expression | |
| <input type="button" value="Cancel"/> <input type="button" value="Clear form"/> <input type="button" value="Execute"/> <input type="button" value="Generate code"/> | |

Choosing the + (plus sign) in the bottom left of the row list starts a [Put Item \(p. 790\)](#) operation.

PutItem operation builder

| | |
|--|---|
| * Partition key | Attribute value for partition key: year |
| * Sort key | Attribute value for sort key: title |
| Other attributes | |
| + | |
| Condition expression | |
| + Condition + Child expression | |
| <input type="button" value="Cancel"/> <input type="button" value="Clear"/> <input type="button" value="Execute"/> <input type="button" value="Generate code"/> | |

Building Complex Operations

The operation builder in NoSQL Workbench for Amazon DynamoDB provides a visual interface where you can perform complex data plane operations. It includes support for projection expressions and condition expressions. You can also choose to generate sample code for these operations, in multiple languages.

To build DynamoDB operations

1. In NoSQL Workbench, in the navigation pane on the left side, choose the **Operation builder** icon.



2. Choose **Build operations**, and then choose the operation that you want.

The screenshot shows the AWS NoSQL Workbench interface. On the left is a dark sidebar with icons for AWS Lambda, CloudWatch Metrics, CloudWatch Logs, and the Operation builder (which is highlighted with a red box). The main area is titled 'Operation builder' and has a 'Build operations' button with a dropdown menu. The dropdown menu lists several operations: Update Item, Put Item, Delete Item, Query, Scan, and TransactWriteItems. The 'Movies' table is selected in the center.

You can perform the following operations in the operation builder.

Topics

- [Put Item \(p. 790\)](#)
- [Update Item \(p. 792\)](#)
- [Delete Item \(p. 793\)](#)
- [Query \(p. 794\)](#)
- [Scan \(p. 795\)](#)
- [TransactWriteItems \(p. 796\)](#)

Put Item

To execute or generate code for a `Put Item` operation, do the following.

1. Specify the partition key value.
2. Specify the sort key value, if one exists.
3. If you want to add non-key attributes, do the following:
 - a. Choose the + (plus sign) next to **Other attributes**.
 - b. Specify the **Attribute name**, **Type**, and **Value**.
4. If a condition expression must be satisfied for the `Put Item` operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition Expressions \(p. 392\)](#).

PutItem operation builder

The screenshot shows the PutItem operation builder interface. It includes fields for 'Partition key' (value: year) and 'Sort key' (value: title). Under 'Other attributes', there is a field for 'Name' (info), 'Type' (String), and 'Value' (none). The 'Condition expression' section contains buttons for '+ Condition', '+ Child expression', 'And', 'Or', and 'Negate'. Below this is a condition template with fields for 'Attribute name', 'Comparison operator', 'Attribute type', and 'Attribute value'. A 'Child expression' section with similar controls is also present. At the bottom are 'Cancel', 'Clear', 'Execute', and 'Generate code' buttons.

5. If you want to generate code, choose **Generate code.**

Select your desired language from the displayed tabs. You can now copy this code and use it in your application.

PutItem operation builder

The screenshot shows the PutItem operation builder interface with generated code for the Python tab. The code is as follows:

```

# Before running the code below, please follow these steps to setup your workspace if you have not
# set it up already:
#
# 1. Setup credentials for DynamoDB access. One of the ways to setup credentials is to add them to
#    ~/.aws/credentials file (C:\Users\USER_NAME\.aws\credentials file for Windows users) in
#    following format:
#
#    [profile_name]
#    aws_access_key_id = YOUR_ACCESS_KEY_ID
#    aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
#
#    If <profile_name> is specified as "default" then AWS SDKs and CLI will be able to read the credentials
#    without any additional configuration. But if a different profile name is used then it needs to be
#    specified while initializing DynamoDB client via AWS SDKs or while configuring AWS CLI.
#
#    Please refer following guide for more details on credential configuration:

```

At the bottom are 'Cancel', 'Clear', 'Execute', and 'Generate code' buttons.

6. If you want the operation to be executed immediately, choose **Execute.**

For more information about this operation, see [PutItem](#) in the *Amazon DynamoDB API Reference*.

Update Item

To execute or generate code for an `Update Item` operation, do the following:

1. Enter the partition key value.
2. Enter the sort key value, if one exists.
3. In **Update expression**, choose the expression in the list.
4. Choose the **+** (plus sign) for the expression.
5. Enter the attribute name and attribute value for the selected expression.
6. If you want to add more expressions, choose another expression in the **Update Expression** drop-down list, and then select the **+**.
7. If a condition expression must be satisfied for the `Update Item` operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition Expressions \(p. 392\)](#).

UpdateItem operation builder

The screenshot shows the 'UpdateItem operation builder' interface. It has fields for 'Partition key' (value: year) and 'Sort key' (value: title). Below these is an 'Update expression' field with a dropdown menu showing 'Select Operation'. Underneath the expression field are tabs for '+ Condition' (selected), '+ Child expression', 'And', 'Or', and 'Negate'. At the bottom are buttons for 'Cancel', 'Clear form', 'Execute', and 'Generate code'.

8. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

UpdateItem operation builder

* Partition key: 1933

* Sort key: King Kong

Update expression: Set

* Set: info = String additional information

Condition expression: + Condition + Child expression

Generated code:

Python

```
# Load the AWS SDK for Python
import boto3
from botocore.exceptions import ClientError

def create_dynamodb_client(region="us-east-1"):
    return boto3.client("dynamodb", region_name=region)

def create_update_item_input():
    return {
        "TableName": "Movies",
        "Key": {
            "year": {"N": "1933"},
            "title": {"S": "King Kong"}
        },
        "UpdateExpression": "SET #10320 = :10320",
        "ExpressionAttributeNames": {"#10320": "info"},
        "ConditionExpression": "#year = 1933 AND #title = King Kong"
    }
```

JavaScript (Node.js)

```
# Load the AWS SDK for Node.js
var AWS = require('aws-sdk');

// Set the region
AWS.config.update({region: 'us-east-1'});

var dynamoDB = new AWS.DynamoDB();
var params = {
    TableName: "Movies",
    Key: {
        year: "1933",
        title: "King Kong"
    },
    UpdateExpression: "SET #10320 = :10320",
    ExpressionAttributeNames: {"#10320": "info"},
    ConditionExpression: "#year = 1933 AND #title = King Kong"
};

dynamoDB.updateItem(params, function(err, data) {
    if (err) {
        console.error("Error", err);
    } else {
        console.log("Success", data);
    }
});
```

Java

```
# Load the AWS SDK for Java
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.model.UpdateItemRequest;
import com.amazonaws.services.dynamodbv2.model.UpdateItemResult;

public class UpdateItemExample {
    public static void main(String[] args) {
        // Create a new AmazonDynamoDBClient
        AmazonDynamoDB client = new AmazonDynamoDBClient();

        // Create a new UpdateItemRequest
        UpdateItemRequest request = new UpdateItemRequest()
            .withTableName("Movies")
            .withKey(new AttributeValue().withS("1933"))
            .withUpdateExpression("SET #10320 = :10320")
            .withConditionExpression("#year = 1933 AND #title = King Kong")
            .withExpressionAttributeNames(new ExpressionAttributeNames()
                .withEntry("#10320", "info"))
            .withExpressionAttributeValues(new ExpressionAttributeValues()
                .withEntry(":10320", new AttributeValue().withS("King Kong")));
    }
}
```

Cancel Clear form Execute Generate code

- If you want the operation to be executed immediately, choose **Execute**.

For more information about this operation, see [UpdateItem](#) in the *Amazon DynamoDB API Reference*.

Delete Item

To execute or generate code for a `Delete Item` operation, do the following.

- Enter the partition key value.
- Enter the sort key value, if one exists.
- If a condition expression must be satisfied for the `Delete Item` operation to succeed, do the following:
 - Choose **Condition**.
 - Specify the attribute name, comparison operator, attribute type, and attribute value.
 - If other conditions are needed, choose **Condition** again.

For more information, see [Condition Expressions \(p. 392\)](#).

DeleteItem operation builder

* Partition key: Attribute value for partition key: year

* Sort key: Attribute value for sort key: title

Condition expression: + Condition + Child expression

And Or Negate

* Condition Negate Attribute name = Attribute type Attribute value

Cancel Clear form Execute Generate code

4. If you want to generate code, choose **Generate code.**

Choose the tab for the language that you want. You can now copy this code and use it in your application.



The screenshot shows the 'DeleteItem operation builder' interface. At the top, there are fields for 'Partition key' (1933) and 'Sort key' (King Kong). Below these are tabs for 'Condition expression', '+ Condition', '+ Child expression', and 'Generated code'. The 'Generated code' tab is selected, showing Python code:

```

# coding=utf-8
# AWS SDK for Python - AWS SDK for Python (PySDK)
# Copyright 2013 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"). You may
# not use this file except in compliance with the License. A copy of the
# License is located at
#
# http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.

import boto3
from botocore.exceptions import ClientError

def create_dynamodb_client(region='us-east-1'):
    return boto3.client("dynamodb", region_name=region)

def create_delete_item_input():
    return {
        "tableName": "Movies",
        "key": {
            "year": {"N": "1933"},
            "title": {"S": "King Kong"}
        }
    }

def execute_delete_item(dynamodb_client, input):
    dynamodb_client.delete_item(
        ...
    )

```

At the bottom right are buttons for 'Clear form', 'Execute', and 'Generate code'.

5. If you want the operation to be executed immediately, choose **Execute.**

For more information about this operation, see [DeleteItem](#) in the *Amazon DynamoDB API Reference*.

Query

To execute or generate code for a **Query** operation, do the following.

1. Specify the partition key value.
2. If a sort key is needed for the **Query** operation:
 - a. Select **Sort key**.
 - b. Specify the comparison operator, attribute type, and attribute value.
3. If not all the attributes should be returned with the operation result, select **Projection expression**.
4. Choose the + (plus sign).
5. Enter the attribute to return with the query result.
6. If more attributes are needed, choose the + .
7. If a condition expression must be satisfied for the **Query** operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition Expressions \(p. 392\)](#).

Query operation builder

* Partition key Attribute value for partition key: year

Sort key

Projection expression

* Projected attribute Attribute name

Filter expression Condition Child expression
 And Or Negate

* Condition Negate Attribute name = Attribute type Attribute value

Other parameters

Cancel Clear form Execute Generate code

8. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

Query operation builder

* Partition key Vertigo

Sort key

Projection expression

* Projected attribute info

Filter expression Condition Child expression

Other parameters

Generated code

Python JavaScript (Node.js) Java

```
# Load the AWS SDK for Python
import boto3
from botocore.exceptions import ClientError

def create_dynamodb_client(region='us-east-1'):
    return boto3.client("dynamodb", region_name=region)

def create_query_input():
    return {
        "TableName": "Movies",
        "KeyConditionExpression": "#3c9e1 = :3c9e1",
        "ProjectionExpression": "#3c9e0",
        "ExpressionAttributeNames": {"#3c9e0": "info", "#3c9e1": "year"},
        "ExpressionAttributeValues": {":3c9e1": {"N": "Vertigo")}
```

Cancel Clear form Execute Generate code

9. If you want the operation to be executed immediately, choose **Execute**.

For more information about this operation, see [Query](#) in the *Amazon DynamoDB API Reference*.

Scan

To execute or generate code for a Scan operation, do the following.

1. If not all the attributes should be returned with the operation result, select **Projection expression**.

2. Choose the **+** (plus sign).
3. Specify the attribute to return with the query result.
4. If more attributes are needed, choose the **+** again.
5. If a condition expression must be satisfied for the scan operation to succeed, do the following:
 - a. Choose **Condition**.
 - b. Specify the attribute name, comparison operator, attribute type, and attribute value.
 - c. If other conditions are needed, choose **Condition** again.

For more information, see [Condition Expressions \(p. 392\)](#).

Scan operation builder

The screenshot shows the 'Scan operation builder' interface. At the top, there are buttons for 'Filter expression', '+ Condition', '+ Child expression', and a help icon. Below this, under 'Projection expression', two attributes are selected: 'year' and 'title'. In the 'Other parameters' section, there is a 'Generated code' tab. The 'Python' tab is selected, displaying the following code:

```
# Load the AWS SDK for Python
import boto3
from botocore.exceptions import ClientError

def create_dynamodb_client(region="us-east-1"):
    return boto3.client("dynamodb", region_name=region)

def create_scan_input():
    return {
        "TableName": "Movies",
        "ProjectionExpression": "#d3cc0,#d3cc1",
        "ExpressionAttributeNames": {"#d3cc0": "year", "#d3cc1": "title"}
    }

def execute_scan(dynamodb_client, input):
    try:
```

At the bottom of the interface are buttons for 'Cancel', 'Clear form', 'Execute', and 'Generate code'.

6. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

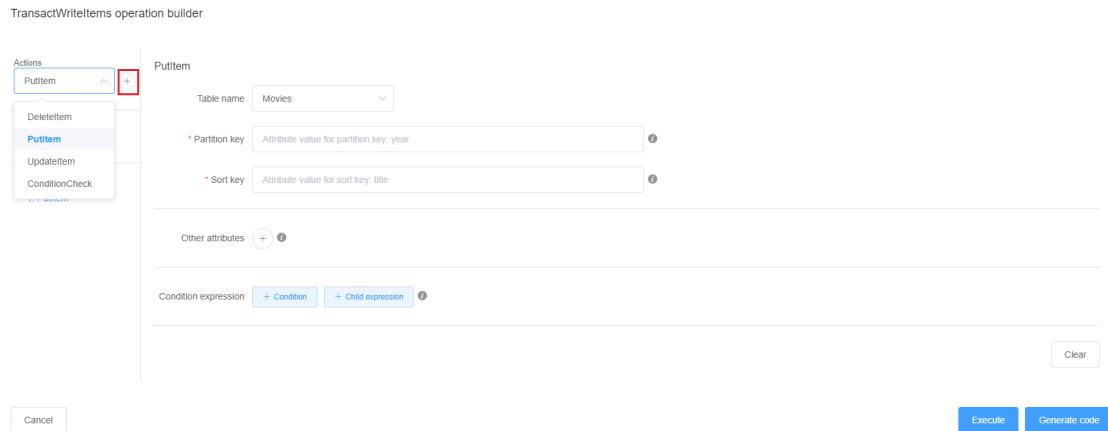
7. If you want the operation to be executed immediately, choose **Execute**.

TransactWriteItems

To execute or generate code for a `TransactWriteItems` operation, do the following.

1. In the **Actions** drop-down list, choose the operation that you want.
 - For `DeleteItem`, follow the instructions for the [Delete Item \(p. 793\)](#) operation.
 - For `PutItem`, follow the instructions for the [Put Item \(p. 790\)](#) operation.
 - For `UpdateItem`, follow the instructions for the [Update Item \(p. 792\)](#) operation.

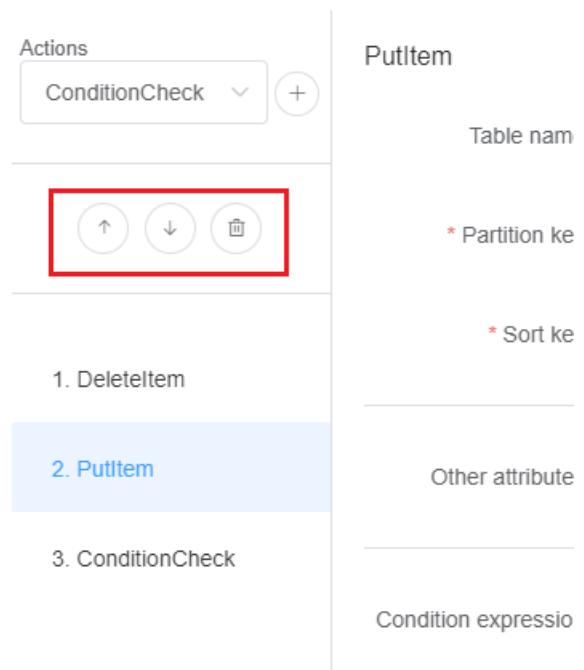
When you are done specifying the details of the operation, choose the **+** button.



To change the order of actions, choose an action in the list on the left side, and then choose the up or down arrows to move it up or down in the list.

To delete an action, choose the action in the list, and then choose the **Delete** (trash can) icon.

TransactWriteItems operation builder



2. If you want to generate code, choose **Generate code**.

Choose the tab for the language that you want. You can now copy this code and use it in your application.

The screenshot shows the Amazon DynamoDB Workbench interface. At the top, there's a header with '2. PutItem' and tabs for 'Python', 'JavaScript (Node.js)', and 'Java'. Below the tabs is a code editor containing Python code for creating a DynamoDB client and performing a transact write item operation. The code includes a 'Select' statement for the 'Movies' table with a 'Year' range key from '2010' to '2011'. At the bottom of the code editor are 'Cancel', 'Execute', and 'Generate code' buttons.

```
# Load the AWS SDK for Python
import boto3
from botocore.exceptions import ClientError

def create_dynamodb_client(region="us-east-1"):
    return boto3.client("dynamodb", region_name=region)

def create_transact_write_items_input():
    return {
        "TransactItems": [
            {
                "Delete": {
                    "TableName": "Movies",
                    "Key": {
                        "Year": {"N": "2010"},
                        "Title": {"S": "A Touch of Evil"}
                    }
                }
            }
        ]
    }

# Create the DynamoDB client
client = create_dynamodb_client()

# Create the transact write items input
transact_write_items_input = create_transact_write_items_input()

# Create the transact write items
response = client.transact_write_items(
    TransactItems=transact_write_items_input
)
```

3. If you want the operation to be executed immediately, choose **Execute**.

For more information about transactions, see [Amazon DynamoDB Transactions](#).

Sample Data Models for NoSQL Workbench

The home page for the modeler and visualizer display a number of sample models that ship with the NoSQL Workbench. This section describes these models and their potential uses.

Topics

- [Employee Data Model \(p. 798\)](#)
- [Discussion Forum Data Model \(p. 799\)](#)
- [Music Library Data Model \(p. 799\)](#)
- [Ski Resort Data Model \(p. 799\)](#)
- [Credit Card Offers Data Model \(p. 799\)](#)
- [Bookmarks Data Model \(p. 800\)](#)

Employee Data Model

This data model is an introductory model. It represents an employee's basic details such as a unique alias, first name, last name, designation, manager, and skills.

This data model depicts a few techniques such as handling complex attribute such as having more than one skill. This model is also an example of one-to-many relationship through the manager and thier reporting employees that has been achieved by the secondary index DirectReports.

The access patterns facilitated by this data model are:

- Retrieval of an employee record using the employee's login alias, facilitated by a table called Employee.
- Search for employees by name, facilitated by the Employee table's global secondary index called Name.
- Retrieval of all direct reports of a manager using the manager's login alias, facilitated by the Employee table's global secondary index called DirectReports.

Discussion Forum Data Model

This data model represents a discussion forums. Using this model customers can engage with the developer community, ask questions, and respond to other customers' posts. Each AWS service has a dedicated forum. Anyone can start a new discussion thread by posting a message in a forum, and each thread receives any number of replies.

The access patterns facilitated by this data model are:

- Retrieval of a forum record using the forum's name, facilitated by a table called `Forum`.
- Retrieval of a specific thread or all threads for a forum, facilitated by a table called `Thread`.
- Search for replies using the posting user's email address, facilitated by the `Reply` table's global secondary index called `PostedBy-Message-Index`.

Music Library Data Model

This data model represents a music library that has a large collection of songs and showcases its most downloaded songs in near-real time.

The access patterns facilitated by this data model are:

- Retrieval of a song record, facilitated by a table called `Songs`.
- Retrieval of a specific download record or all download records for a song, facilitated by a table called `Songs`.
- Retrieval of a specific monthly download count record or all monthly download count records for a song, facilitated by a table called `Song`.
- Retrieval of all records (including song record, download records, and monthly download count records) for a song, facilitated by a table called `Songs`.
- Search for most downloaded songs, facilitated by the `Songs` table's global secondary index called `DownloadsByMonth`.

Ski Resort Data Model

This data model represents a ski resort that has a large collection of data for each ski lift collected daily.

The access patterns facilitated by this data model are:

- Retrieval of all data for a given ski lift or overall resort, dynamic and static, facilitated by a table called `SkiLifts`.
- Retrieval of all dynamic data (including unique lift riders, snow coverage, avalanche danger, and lift status) for a ski lift or the overall resort on a specific date, facilitated by a table called `SkiLifts`.
- Retrieval of all static data (including if the lift is for experienced riders only, vertical feet the lift rises, and lift riding time) for a specific ski lift, facilitated by a table called `SkiLifts`.
- Retrieval of date of data recorded for a specific ski lift or the overall resort sorted by total unique riders, facilitated by the `SkiLifts` table's global secondary index called `SkiLiftsByRiders`.

Credit Card Offers Data Model

This data model is used by a Credit Card Offers Application.

A credit card provider produces offers over time. These offers include balance transfers without fees, increased credit limits, lower interest rates, cash back, and airline miles. After a customer accepts or declines these offers, the respective offer status is updated accordingly.

The access patterns facilitated by this data model are:

- Retrieval of account records using `AccountId`, as facilitated by the main table.
- Retrieval of all the accounts with few projected items, as facilitated by the secondary index `AccountIndex`.
- Retrieval of accounts and all the offer records associated with those accounts by using `AccountId`, as facilitated by the main table.
- Retrieval of accounts and specific offer records associated with those accounts by using `AccountId` and `OfferId`, as facilitated by the main table.
- Retrieval of all ACCEPTED/DECLINED offer records of specific `OfferType` associated with accounts using `AccountId`, `OfferType`, and `Status`, as facilitated by the secondary index `GSI1`.
- Retrieval of offers and associated offer item records using `OfferId`, as facilitated by the main table.

Bookmarks Data Model

This data model is used store bookmarks for customers.

A customer can have many bookmarks and a bookmark can belong to many customers. This data model represents a many-to-many relationship.

The access patterns facilitated by this data model are:

- A single query by `customerId` can now return customer data as well as bookmarks.
- A query `ByEmail` index returns customer data by email address. Note that bookmarks are not retrieved by this index.
- A query `ByUrl` index gets bookmarks data by URL. Note that we have `customerId` as the sort key for the index because the same URL can be bookmarked by multiple customers.
- A query `ByCustomerFolder` index gets bookmarks by folder for each customer.

Release History for NoSQL Workbench

The following table describes the important changes in each release of the *NoSQL Workbench* client tool.

| Change | Description | Date |
|---|--|--------------------|
| NoSQL Workbench preview released. | This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables. For more information, see NoSQL Workbench for Amazon DynamoDB (Preview) . | September 16, 2019 |
| Support for DynamoDB Local (Downloadable Version) . | The NoSQL Workbench now supports connecting to DynamoDB Local (Downloadable Version) to design, create, query, and manage DynamoDB tables. | November 8, 2019 |

| Change | Description | Date |
|---|---|-------------------|
| Support for IAM roles and temporary security credentials. | NoSQL Workbench for Amazon DynamoDB adds support for AWS Identity and Access Management (IAM) roles and temporary security credentials. | December 19, 2019 |
| NoSQL Workbench for Amazon DynamoDB – GA. | NoSQL Workbench for Amazon DynamoDB is generally available. | March 2, 2020 |
| Support for Linux. | NoSQL Workbench for Amazon DynamoDB is supported on Linux- Ubuntu , Fedora and Debian. | May 4, 2020 |

Analyzing Data Access Using CloudWatch Contributor Insights for DynamoDB

Amazon CloudWatch Contributor Insights for Amazon DynamoDB is a diagnostic tool for identifying the most frequently accessed and throttled keys in your table or index at a glance. This tool uses [CloudWatch Contributor Insights](#).

By enabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, you can view the most accessed and throttled items in those resources.

Note

CloudWatch charges apply for Contributor Insights for DynamoDB. For more information about pricing, see [Amazon CloudWatch pricing](#).

Topics

- [CloudWatch Contributor Insights for DynamoDB: How It Works \(p. 802\)](#)
- [Getting Started with CloudWatch Contributor Insights for DynamoDB \(p. 806\)](#)
- [Using IAM with CloudWatch Contributor Insights for DynamoDB \(p. 810\)](#)

CloudWatch Contributor Insights for DynamoDB: How It Works

Amazon DynamoDB integrates with [Amazon CloudWatch Contributor Insights](#) to provide information about the most accessed and throttled items in a table or global secondary index. DynamoDB delivers this information to you via CloudWatch Contributor Insights [rules](#), [reports](#), and graphs of report data.

For more information about CloudWatch Contributor Insights, see [Using Contributor Insights to Analyze High-Cardinality Data](#) in the [Amazon CloudWatch User Guide](#).

The following sections describe the core concepts and behavior of CloudWatch Contributor Insights for DynamoDB.

Topics

- [CloudWatch Contributor Insights for DynamoDB Rules \(p. 802\)](#)
- [Understanding CloudWatch Contributor Insights for DynamoDB Graphs \(p. 803\)](#)
- [Interactions with Other DynamoDB Features \(p. 805\)](#)
- [CloudWatch Contributor Insights for DynamoDB Billing \(p. 806\)](#)

CloudWatch Contributor Insights for DynamoDB Rules

When you enable CloudWatch Contributor Insights for DynamoDB on a table or global secondary index, DynamoDB creates the following [rules](#) on your behalf:

- **Most accessed items (partition key)** — Identifies the partition keys of the most accessed items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-PKC-[resource_name]-[creationtimestamp]

- **Most throttled keys (partition key)** — Identifies the partition keys of the most throttled items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-PKT-[resource_name]-[creationtimestamp]

If your table or global secondary index has a sort key, DynamoDB also creates the following rules specific to sort keys:

- **Most accessed keys (partition and sort keys)** — Identifies the partition and sort keys of the most accessed items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-SKC-[resource_name]-[creationtimestamp]

- **Most throttled keys (partition and sort keys)** — Identifies the partition and sort keys of the most throttled items in your table or global secondary index.

CloudWatch rule name format: DynamoDBContributorInsights-SKT-[resource_name]-[creationtimestamp]

Note

- You can't use the CloudWatch console or APIs to directly modify or delete the rules created by CloudWatch Contributor Insights for DynamoDB. Disabling CloudWatch Contributor Insights for DynamoDB on a table or global secondary index automatically deletes the rules created for that table or global secondary index.
- When you use the [GetInsightRuleReport](#) operation with CloudWatch Contributor Insights rules that are created by DynamoDB, only `MaxContributorValue` and `Maximum` return useful statistics. The other statistics in this list don't return meaningful values.

You can create CloudWatch Alarms using the CloudWatch Contributor Insights for DynamoDB [rules](#). This allows you to be notified when any item exceed or meets a specific threshold for `ConsumedThroughputUnits` or `ThrottleCount`. For more information, see [Setting an Alarm on Contributor Insights Metric Data](#).

Understanding CloudWatch Contributor Insights for DynamoDB Graphs

CloudWatch Contributor Insights for DynamoDB displays two types of graphs on both the DynamoDB and CloudWatch consoles: *Most Accessed Items* and *Most Throttled Items*.

Most Accessed Items

Use this graph to identify the most accessed items in the table or global secondary index. The graph displays `ConsumedThroughputUnits` on the y-axis and time on the x-axis. Each of the top N keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures key access frequency by using `ConsumedThroughputUnits`, which measures combined read and write traffic. `ConsumedThroughputUnits` is defined as the following:

- Provisioned — $(3 \times \text{consumed write capacity units}) + \text{consumed read capacity units}$
- On-demand — $(3 \times \text{write request units}) + \text{read request units}$

On the DynamoDB console, each data point in the graph represents the maximum of ConsumedThroughputUnits over a 1-minute period. For example, a graph value of 180,000 ConsumedThroughputUnits indicates that the item was accessed continuously at the per-item maximum throughput of 1,000 write request units or 3,000 read request units for a 60-second span within that 1-minute period ($3,000 \times 60$ seconds). In other words, the graphed values represent the highest-traffic minute within each 1-minute period. You can change the time granularity of the ConsumedThroughputUnits metric (for example, to view 5-minute metrics instead of 1-minute) on the CloudWatch console.

If you see several closely clustered lines without any obvious outliers, it indicates that your workload is relatively balanced across items over the given time window. If you see isolated points in the graph instead of connected lines, it indicates an item that was frequently accessed only for a brief period.

If your table or global secondary index has a sort key, DynamoDB creates two graphs: one for the most accessed partition keys and one for the most accessed partition + sort key pairs. You can see traffic at the partition key level in the partition key-only graph. You can see traffic at the item level in the partition + sort key graphs.

Most Throttled Items

Use this graph to identify the most throttled items in the table or global secondary index. The graph displays ThrottleCount on the y-axis and time on the x-axis. Each of the top N keys is displayed in its own color, with a legend displayed below the x-axis.

DynamoDB measures throttle frequency using ThrottleCount, which is the count of ProvisionedThroughputExceededException, ThrottlingException, and RequestLimitExceeded errors.

On the DynamoDB console, each data point in the graph represents the count of throttle events over a 1-minute period.

If you see no data in this graph, it indicates that your requests are not being throttled. If you see isolated points in the graph instead of connected lines, it indicates that an item was frequently throttled for a brief period.

If your table or global secondary index has a sort key, DynamoDB creates two graphs: one for most throttled partition keys and one for most throttled partition + sort key pairs. You can see throttle count at the partition key level in the partition key-only graph, and throttle count at the item-level in the partition + sort key graphs.

Report Examples

The following are examples of the reports generated for a table with both a partition key and sort key.



Interactions with Other DynamoDB Features

The following sections describe how CloudWatch Contributor Insights for DynamoDB behaves and interacts with several other features in DynamoDB.

Global Tables

CloudWatch Contributor Insights for DynamoDB monitors global table replicas as distinct tables. The Contributor Insights graphs for a replica in one AWS Region might not show the same patterns as another Region. This is because write data is replicated across all replicas in a global table, but each replica can serve Region-bound read traffic.

DynamoDB Accelerator (DAX)

CloudWatch Contributor Insights for DynamoDB doesn't show DAX cache responses. It only shows responses to accessing a table or a global secondary index.

Encryption at Rest

CloudWatch Contributor Insights for DynamoDB doesn't affect how encryption works in DynamoDB. The primary key data that is published in CloudWatch is encrypted with the AWS owned customer master key (CMK). However, DynamoDB also supports the AWS managed CMK and a customer managed CMK.

If you require your primary key data to be encrypted with the AWS managed CMK or a customer managed CMK, you should not enable CloudWatch Contributor Insights for DynamoDB for that table.

Fine-Grained Access Control

CloudWatch Contributor Insights for DynamoDB doesn't function differently for tables with fine-grained access control (FGAC). In other words, any user who has the appropriate CloudWatch permissions can view FGAC-protected primary keys in CloudWatch Contributor Insights graphs.

If the table's primary key contains FGAC-protected data that you don't want published to CloudWatch, you should not enable CloudWatch Contributor Insights for DynamoDB for that table.

Access Control

You control access to CloudWatch Contributor Insights for DynamoDB using AWS Identity and Access Management (IAM) by limiting DynamoDB control plane permissions and CloudWatch data plane permissions. For more information see, [Using IAM with CloudWatch Contributor Insights for DynamoDB](#).

CloudWatch Contributor Insights for DynamoDB Billing

Charges for CloudWatch Contributor Insights for DynamoDB appear in the [CloudWatch](#) section of your monthly bill. These charges are calculated based on the number of DynamoDB events that are processed. For tables and global secondary indexes with CloudWatch Contributor Insights for DynamoDB enabled, each item that is written or read via a [data plane](#) operation represents one event.

If a table or global secondary index includes a sort key, each item that is read or written represents two events. This is because DynamoDB is identifying top contributors from separate time series: one for partitions keys only, and one for partition and sort key pairs.

For example, assume that your application performs the following DynamoDB operations: a `GetItem`, a `PutItem`, and a `BatchWriteItem` that puts five items

- If your table or global secondary index has only a partition key, it results in 7 events (1 for the `GetItem`, 1 for the `PutItem`, and 5 for the `BatchWriteItem`).
- If your table or global secondary index has a partition key and sort key, it results in 14 events (2 for the `GetItem`, 2 for the `PutItem`, and 10 for the `BatchWriteItem`).
- A `Query` operation always results in 1 event, regardless of the number of items returned.

Unlike other DynamoDB features, CloudWatch Contributor Insights for DynamoDB billing *does not* vary based on the following:

- The [capacity mode](#) (provisioned vs. on-demand)
- Whether you perform read or write requests
- The size (KB) of the items read or written

Getting Started with CloudWatch Contributor Insights for DynamoDB

This section describes how to use Amazon CloudWatch Contributor Insights with the Amazon DynamoDB console or the AWS Command Line Interface (AWS CLI).

In the following examples, you use the DynamoDB table that is defined in the [Getting Started with DynamoDB](#) tutorial.

Topics

- [Using Contributor Insights \(Console\) \(p. 807\)](#)
- [Using Contributor Insights \(AWS CLI\) \(p. 810\)](#)

Using Contributor Insights (Console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the **Music** table.
4. Choose the **Contributor Insights** tab.
5. Choose **Manage Contributor Insights**.

The screenshot shows the 'Contributor Insights' tab for the 'Music' table. The 'Manage Contributor Insights' button is highlighted with a red box. Below it, there is a table showing resource names, types, and current contributor insights status:

| Resource Name | Type | Contributor Insights Status |
|------------------|------------|-----------------------------|
| Music | Base Table | Disabled |
| AlbumTitle-index | GSI | Disabled |

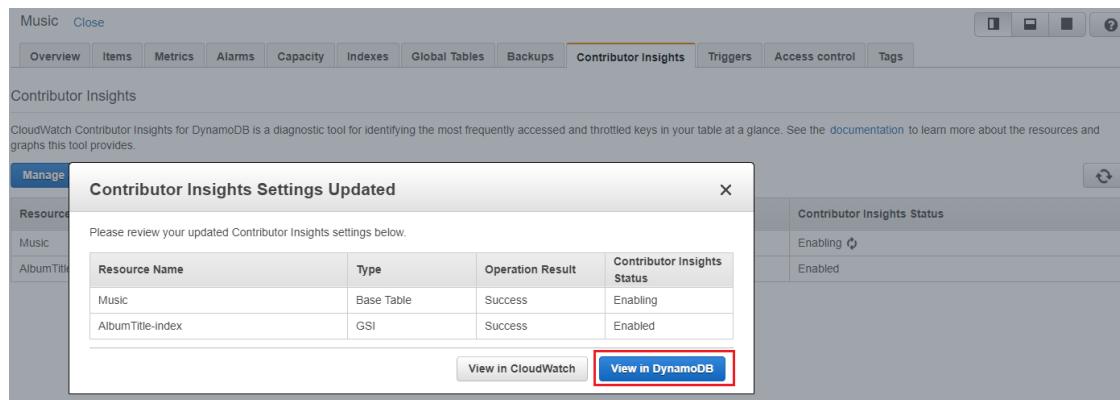
A message at the bottom states: "Contributor Insights is not enabled on this table or any of its indexes."

6. In the **Manage Contributor Insights** dialog box, under **Contributor Insights Status**, choose **Enabled** for both the **Music** base table and the **AlbumTitle-index** global secondary index. Then choose **Confirm**.

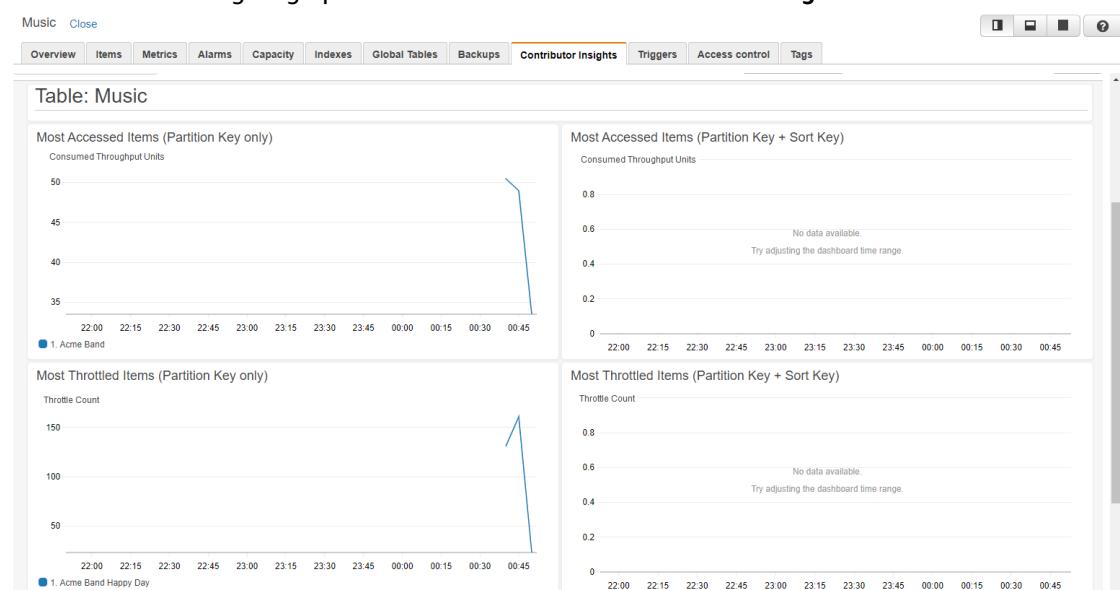
The screenshot shows the 'Manage Contributor Insights' dialog box. It displays the same table from the previous screenshot, but with the 'Contributor Insights Status' dropdown set to 'Enabled' for both rows. A note below the table states: "Users who have the appropriate CloudWatch permissions will be able to view primary keys protected by fine-grained access control (FGAC) in Contributor Insight graphs. If the primary key contains FGAC-protected data that you do not want published to CloudWatch, then you should not enable Contributor Insights for this table." At the bottom, there is a note about additional charges and a 'Confirm' button highlighted with a red box.

If the operation fails, see [DescribeContributorInsights FailureException](#) in the *Amazon DynamoDB API Reference* for possible reasons.

7. Choose **View in DynamoDB**.



8. The Contributor Insights graphs are now visible on the **Contributor Insights** tab for the **Music** table.



Creating CloudWatch Alarms

Follow these steps to create a CloudWatch alarm and be notified when any partition key consumes more than 50,000 [ConsumedThroughputUnits](#).

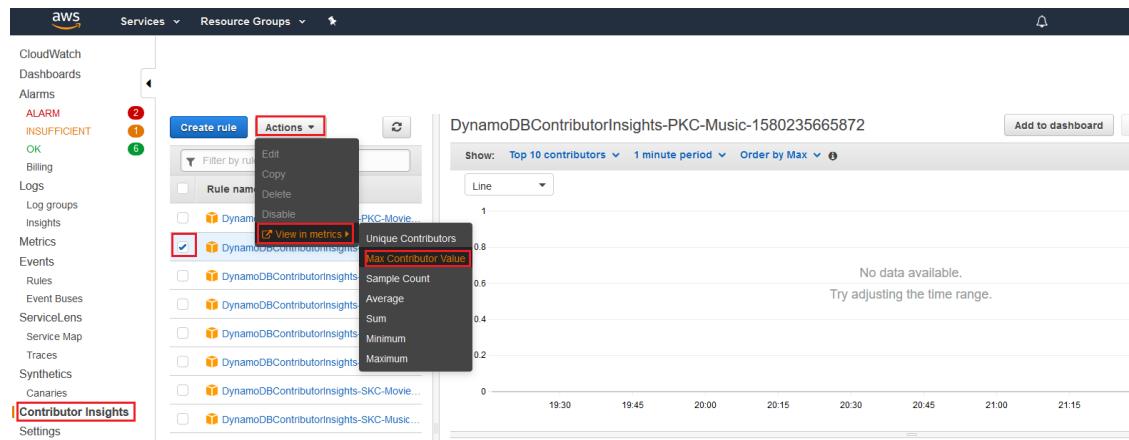
1. Sign in to the AWS Management Console and open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>
2. In the navigation pane on the left side of the console, choose **Contributor Insights**.
3. Choose the **DynamoDBContributorInsights-PKC-Music** rule.
4. Choose the **Actions** drop down.
5. Choose **View in metrics**.
6. Choose **Max Contributor Value**.

Note

Only **Max Contributor Value** and **Maximum** return useful statistics. The other statistics in this list don't return meaningful values.

Amazon DynamoDB Developer Guide

Using Contributor Insights (Console)



7. On the Actions column, Choose Create Alarm.



8. Enter a value of 50000 for threshold and choose Next.

This screenshot shows the 'Step 3: Add name and description' screen for creating a CloudWatch Alarm. The 'Label' field contains 'DynamoDBContributorInsights-PKC-Music-158749'. The 'Math expression' field contains 'INSIGHT_RULE_METRIC(DynamoDBContributorInsights...'. The 'Period' is set to '1 minute'. In the 'Conditions' section, the 'Threshold type' is set to 'Static' (radio button selected). The 'Greater > threshold' option is selected under 'Whenever DynamoDBContributorInsights-PKC-Music-1587490256272 MaxContributorValue is...'. The 'than...' field contains '50000'. The 'Next' button is visible at the bottom right.

9. See [Using Amazon CloudWatch Alarms](#) for details on how to configure the notification for the alarm.

Using Contributor Insights (AWS CLI)

1. Enable CloudWatch Contributor Insights for DynamoDB on the `Music` base table.

```
aws dynamodb update-contributor-insights --table-name Music --contributor-insights-action=ENABLE
```

2. Enable Contributor Insights for DynamoDB on the `AlbumTitle-index` global secondary index.

```
aws dynamodb update-contributor-insights --table-name Music --index-name AlbumTitle-index --contributor-insights-action=ENABLE
```

3. Get the status and rules for the `Music` table and all its indexes.

```
aws dynamodb describe-contributor-insights --table-name Music
```

4. Disable CloudWatch Contributor Insights for DynamoDB on the `AlbumTitle-index` global secondary index.

```
aws dynamodb update-contributor-insights --table-name Music --index-name AlbumTitle-index --contributor-insights-action=DISABLE
```

5. Get the status of the `Music` table and all its indexes.

```
aws dynamodb list-contributor-insights --table-name Music
```

Using IAM with CloudWatch Contributor Insights for DynamoDB

The first time that you enable Amazon CloudWatch Contributor Insights for Amazon DynamoDB, DynamoDB automatically creates an AWS Identity and Access Management (IAM) service-linked role for you. This role, `AWSServiceRoleForDynamoDBCloudWatchContributorInsights`, allows DynamoDB to manage CloudWatch Contributor Insights rules on your behalf. Don't delete this service-linked role. If you delete it, all your managed rules will no longer be cleaned up when you delete your table or global secondary index.

For more information about service-linked roles, see [Using Service-Linked Roles](#) in the *IAM User Guide*.

The following permissions are required:

- To enable or disable CloudWatch Contributor Insights for DynamoDB, you must have `dynamodb:UpdateContributorInsights` permission on the table or index.
- To view CloudWatch Contributor Insights for DynamoDB graphs, you must have `cloudwatch:GetInsightRuleReport` permission.
- To describe CloudWatch Contributor Insights for DynamoDB for a given DynamoDB table or index, you must have `dynamodb:DescribeContributorInsights` permission.
- To list CloudWatch Contributor Insights for DynamoDB statuses for each table and global secondary index, you must have `dynamodb>ListContributorInsights` permission.

Example: Enable or Disable CloudWatch Contributor Insights for DynamoDB

The following IAM policy grants permissions to enable or disable CloudWatch Contributor Insights for DynamoDB.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iam:CreateServiceLinkedRole",  
                "dynamodb:UpdateContributorInsights"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Example: Retrieve CloudWatch Contributor Insights Rule Report

The following IAM policy grants permissions to retrieve CloudWatch Contributor Insights rule report.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "cloudwatch:GetInsightRuleReport"  
            ],  
            "Resource": "arn:aws:cloudwatch:*::*:insight-rule/DynamoDBContributorInsights*"  
        }  
    ]  
}
```

Example: Selectively Apply CloudWatch Contributor Insights for DynamoDB Permissions Based on Resource

The following IAM policy grants permissions to allow the `ListContributorInsights` and `DescribeContributorInsights` actions and denies the `UpdateContributorInsights` action for a specific global secondary index.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:ListContributorInsights",  
                "dynamodb:DescribeContributorInsights"  
            ],  
            "Resource": "arn:aws:dynamodb:  
            "Effect": "Deny",  
            "Action": "dynamodb:UpdateContributorInsights",  
            "Resource": "arn:aws:dynamodb:  
        }  
    ]  
}
```

```
        "Action": [
            "dynamodb>ListContributorInsights",
            "dynamodb>DescribeContributorInsights"
        ],
        "Resource": "*"
    },
    {
        "Effect": "Deny",
        "Action": [
            "dynamodb>UpdateContributorInsights"
        ],
        "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/Author-
index"
    }
]
```

Using Service-Linked Roles for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to CloudWatch Contributor Insights for DynamoDB. Service-linked roles are predefined by CloudWatch Contributor Insights for DynamoDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up CloudWatch Contributor Insights for DynamoDB easier because you don't have to manually add the necessary permissions. CloudWatch Contributor Insights for DynamoDB defines the permissions of its service-linked roles, and unless defined otherwise, only CloudWatch Contributor Insights for DynamoDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-Linked Role Permissions for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB uses the service-linked role named **AWSServiceRoleForDynamoDBCloudWatchContributorInsights**. The purpose of the service-linked role is to allow Amazon DynamoDB to manage Amazon CloudWatch Contributor Insights rules created for DynamoDB tables and global secondary indexes, on your behalf.

The **AWSServiceRoleForDynamoDBCloudWatchContributorInsights** service-linked role trusts the following services to assume the role:

- `contributorinsights.dynamodb.amazonaws.com`

The role permissions policy allows CloudWatch Contributor Insights for DynamoDB to complete the following actions on the specified resources:

- Action: `Create and manage Insight Rules on DynamoDBContributorInsights`

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

Creating a Service-Linked Role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually create a service-linked role. When you enable Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you enable Contributor Insights, CloudWatch Contributor Insights for DynamoDB creates the service-linked role for you again.

Editing a Service-Linked Role for CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB does not allow you to edit the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a Service-Linked Role for CloudWatch Contributor Insights for DynamoDB

You don't need to manually delete the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` role. When you disable Contributor Insights in the AWS Management Console, the AWS CLI, or the AWS API, CloudWatch Contributor Insights for DynamoDB cleans up the resources.

You can also use the IAM console, the AWS CLI or the AWS API to manually delete the service-linked role. To do this, you must first manually clean up the resources for your service-linked role and then you can manually delete it.

Note

If the CloudWatch Contributor Insights for DynamoDB service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the `AWSServiceRoleForDynamoDBCloudWatchContributorInsights` service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Security and Compliance in Amazon DynamoDB

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security of the cloud and security *in the cloud*:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the [AWS compliance programs](#). To learn about the compliance programs that apply to DynamoDB, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using DynamoDB. The following topics show you how to configure DynamoDB to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your DynamoDB resources.

Topics

- [Data Protection in DynamoDB \(p. 814\)](#)
- [Identity and Access Management \(p. 823\)](#)
- [Logging and Monitoring \(p. 855\)](#)
- [Compliance Validation by Industry for DynamoDB \(p. 885\)](#)
- [Resilience and Disaster Recovery in Amazon DynamoDB \(p. 886\)](#)
- [Infrastructure Security in Amazon DynamoDB \(p. 886\)](#)
- [Configuration and Vulnerability Analysis in Amazon DynamoDB \(p. 893\)](#)
- [Security Best Practices for Amazon DynamoDB \(p. 893\)](#)

Data Protection in DynamoDB

Amazon DynamoDB provides a highly durable storage infrastructure designed for mission-critical and primary data storage. Data is redundantly stored on multiple devices across multiple facilities in an Amazon DynamoDB Region.

DynamoDB protects user data stored at rest and also data in transit between on-premises clients and DynamoDB, and between DynamoDB and other AWS resources within the same AWS Region.

Topics

- [DynamoDB Encryption at Rest \(p. 815\)](#)
- [Data Protection in DynamoDB Accelerator \(p. 823\)](#)

- [Internetwork Traffic Privacy \(p. 823\)](#)

DynamoDB Encryption at Rest

All user data stored in Amazon DynamoDB is fully encrypted at rest. DynamoDB encryption at rest provides enhanced security by encrypting all your data at rest using encryption keys stored in [AWS Key Management Service \(AWS KMS\)](#). This functionality helps reduce the operational burden and complexity involved in protecting sensitive data. With encryption at rest, you can build security-sensitive applications that meet strict encryption compliance and regulatory requirements.

DynamoDB encryption at rest provides an additional layer of data protection by securing your data in an encrypted table—including its primary key, local and global secondary indexes, streams, global tables, backups, and DynamoDB Accelerator (DAX) clusters whenever the data is stored in durable media. Organizational policies, industry or government regulations, and compliance requirements often require the use of encryption at rest to increase the data security of your applications.

Encryption at rest integrates with AWS KMS for managing the encryption key that is used to encrypt your tables. For more information, see [AWS Key Management Service Concepts](#) in the [AWS Key Management Service Developer Guide](#).

When creating a new table, you can choose one of the following customer master keys (CMK) to encrypt your table:

- AWS owned CMK – Default encryption type. The key is owned by DynamoDB (no additional charge).
- AWS managed CMK – The key is stored in your account and is managed by AWS KMS (AWS KMS charges apply).
- Customer managed CMK – The key is stored in your account and is created, owned, and managed by you. You have full control over the CMK (AWS KMS charges apply).

When you access an encrypted table, DynamoDB decrypts the table data transparently. You can switch between the AWS owned CMK, AWS managed CMK, and customer managed CMK at any given time. You don't have to change any code or applications to use or manage encrypted tables. DynamoDB continues to deliver the same single-digit millisecond latency that you have come to expect, and all DynamoDB queries work seamlessly on your encrypted data.

You can specify an encryption key when you create a new table or switch the encryption keys on an existing table by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API. To learn how, see [Managing Encrypted Tables in DynamoDB \(p. 818\)](#).

Encryption at rest using the AWS owned CMK is offered at no additional charge. However, AWS KMS charges apply for an AWS managed CMK and for a customer managed CMK. For more information about pricing, see [AWS KMS pricing](#).

DynamoDB encryption at rest is available in all AWS Regions, including the AWS China (Beijing) and AWS China (Ningxia) Regions and the AWS GovCloud (US) Regions. Encryption at rest support for customer managed CMKs is available in all AWS Regions except the Asia Pacific (Osaka-Local) Region. For more information, see [Encryption at Rest: How It Works \(p. 815\)](#) and [DynamoDB Encryption at Rest Usage Notes \(p. 817\)](#).

Encryption at Rest: How It Works

Amazon DynamoDB encryption at rest encrypts your data using 256-bit Advanced Encryption Standard (AES-256), which helps secure your data from unauthorized access to the underlying storage.

Encryption at rest integrates with AWS Key Management Service (AWS KMS) for managing the encryption key that is used to encrypt your tables.

When creating a new table or switching the encryption keys on an existing table, you can choose one of the following customer master keys (CMK):

- AWS owned CMK – Default encryption type. The key is owned by DynamoDB (no additional charge).
- AWS managed CMK – The key is stored in your account and is managed by AWS KMS (AWS KMS charges apply).
- Customer managed CMK – The key is stored in your account and is created, owned, and managed by you. You have full control over the CMK (AWS KMS charges apply).

AWS Owned CMK

AWS owned CMKs are not stored in your AWS account. They are part of a collection of CMKs that AWS owns and manages for use in multiple AWS accounts. AWS services can use AWS owned CMKs to protect your data.

You cannot view, manage, or use AWS owned CMKs, or audit their use. However, you do not need to do any work or change any programs to protect the keys that encrypt your data.

You are not charged a monthly fee or a usage fee for use of AWS owned CMKs, and they do not count against AWS KMS limits for your account.

AWS Managed CMK

AWS managed CMKs are CMKs in your account that are created, managed, and used on your behalf by an AWS service that is integrated with AWS KMS. You can view the AWS managed CMKs in your account, view their key policies, and audit their use in AWS CloudTrail logs. However, you cannot manage these CMKs or change their permissions.

Encryption at rest automatically integrates with AWS KMS for managing the AWS managed CMK for DynamoDB (`aws/dynamodb`) that is used to encrypt your tables. If an AWS managed CMK doesn't exist when you create your encrypted DynamoDB table, AWS KMS automatically creates a new key for you. This key is used with encrypted tables that are created in the future. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

For more information about managing permissions of the AWS managed CMK, see [Authorizing Use of the AWS Managed CMK](#) in the *AWS Key Management Service Developer Guide*.

Customer Managed CMK

Customer managed CMKs are CMKs in your AWS account that you create, own, and manage. You have full control over these CMKs, including establishing and maintaining their key policies, IAM policies, and grants; enabling and disabling them; rotating their cryptographic material; adding tags; creating aliases that refer to them; and scheduling them for deletion. For more information about managing permissions of a customer managed CMK, see [Customer Managed CMK Key Policy](#).

When you specify a customer managed CMK as the table-level encryption key, the DynamoDB table, local and global secondary indexes, and streams are encrypted with the same customer managed CMK. On-demand backups are encrypted with the table-level encryption key that is specified at the time the backup is created. Updating the table-level encryption key does not change the encryption key that is associated with existing on-demand backups.

Setting the state of the customer managed CMK to disabled or scheduling it for deletion prevents all users and the DynamoDB service from being able to encrypt or decrypt data and to perform read and write operations on the table. DynamoDB must have access to your encryption key to ensure that you can continue to access your table and to prevent data loss.

If you disable your customer managed CMK or schedule it for deletion, your table status becomes **Inaccessible**. To ensure that you can continue working with the table, you must provide DynamoDB

access to the specified encryption key within seven days. As soon as the service detects that your encryption key is inaccessible, DynamoDB sends you an email notification to alert you.

Note

If your customer managed CMK remains inaccessible to the DynamoDB service for longer than seven days, the table is archived and can no longer be accessed. DynamoDB creates an on-demand backup of your table, and you are billed for it. You can use this on-demand backup to restore your data to a new table. To initiate the restore, the last customer managed CMK on the table must be enabled, and DynamoDB must have access to it.

Notes on Using Managed CMKs

Amazon DynamoDB can't read your table data unless it has access to the CMK stored in your AWS KMS account. DynamoDB uses envelope encryption and key hierarchy to encrypt data. Your AWS KMS encryption key is used to encrypt the root key of this key hierarchy. For more information, see [Envelope Encryption in the AWS Key Management Service Developer Guide](#).

You can use AWS CloudTrail and Amazon CloudWatch Logs to track the requests that DynamoDB sends to AWS KMS on your behalf. For more information, see [Monitoring DynamoDB Interaction with AWS KMS in the AWS Key Management Service Developer Guide](#).

DynamoDB doesn't call AWS KMS for every DynamoDB operation. The key is refreshed once every 5 minutes per client connection with active traffic.

Ensure that you have configured the SDK to reuse connections. Otherwise, you will experience latencies from DynamoDB having to reestablish new AWS KMS cache entries for each DynamoDB operation. In addition, you might potentially have to face higher AWS KMS and CloudTrail costs. For example, to do this using the Node.js SDK, you can create a new HTTPS agent with `keepAlive` turned on. For more information, see [Configuring maxSockets in Node.js in the AWS SDK for JavaScript Developer Guide](#).

DynamoDB Encryption at Rest Usage Notes

Consider the following when you are using encryption at rest in Amazon DynamoDB.

All Table Data Is Encrypted

Server-side encryption at rest is enabled on all DynamoDB table data and cannot be disabled. You cannot encrypt only a subset of items in a table. DynamoDB has encrypted all existing tables that were previously unencrypted by using the AWS owned customer master key (CMK).

Encryption at rest only encrypts data while it is static (at rest) on a persistent storage media. If data security is a concern for data in transit or data in use, you might need to take additional measures:

- **Data in transit:** All your data in DynamoDB is encrypted in transit (except the data in DAX). By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) encryption.
- **Data in use:** Protect your data before sending it to DynamoDB using client-side encryption. For more information, see [Client-Side and Server-Side Encryption in the Amazon DynamoDB Encryption Client Developer Guide](#).

You can use streams with encrypted tables. DynamoDB streams are always encrypted with a table-level encryption key. For more information, see [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#).

DynamoDB backups are encrypted, and the table that is restored from a backup also has encryption enabled. You can use the AWS owned CMK, AWS managed CMK, or customer managed CMK to encrypt your backup data. For more information, see [On-Demand Backup and Restore for DynamoDB \(p. 603\)](#).

Local secondary indexes and global secondary indexes are encrypted using the same key as the base table.

Global tables can be encrypted using an AWS owned CMK or AWS managed CMK.

Encryption Types

On the AWS Management Console, the encryption type is `KMS` when you use the AWS managed CMK or customer managed CMK to encrypt your data. The encryption type is `DEFAULT` when you use the AWS owned CMK. In the Amazon DynamoDB API, the encryption type is `KMS` when you use the AWS managed CMK or customer managed CMK. In the absence of encryption type, your data is encrypted using the AWS owned CMK. You can switch between the AWS owned CMK, AWS managed CMK, and customer managed CMK at any given time. You can use the console, the AWS Command Line Interface (AWS CLI), or the Amazon DynamoDB API to switch the encryption keys.

Note the following limitations when using customer managed CMKs:

- You cannot use a customer managed CMK to encrypt global tables. For more information, see [Global Tables: Multi-Region Replication with DynamoDB \(p. 624\)](#).
- You cannot use a customer managed CMK with DynamoDB Accelerator (DAX) clusters. For more information, see [DAX Encryption at Rest \(p. 755\)](#).
- You can use a customer managed CMK to encrypt tables that use transactions. However, during transaction propagation, data is encrypted using an AWS owned CMK. Data at rest is still encrypted using your customer managed CMK.
- You can use a customer managed CMK to encrypt tables that use Contributor Insights. However, data that is transmitted to Amazon CloudWatch is encrypted with an AWS owned CMK.
- If you disable your customer managed CMK or schedule it for deletion, any data in DynamoDB Streams is still subject to a 24-hour lifetime. Any unretrieved activity data is eligible for trimming when it is older than 24 hours.
- If you disable your customer managed CMK or schedule it for deletion, Time to Live (TTL) deletes continue for 30 minutes. These TTL deletes continue to be emitted to DynamoDB Streams and are subject to the standard trimming/retention interval.

Managing Encrypted Tables in DynamoDB

You can use the AWS Management Console or the AWS Command Line Interface (AWS CLI) to specify the encryption key on new tables and update the encryption keys on existing tables in Amazon DynamoDB.

Topics

- [Specifying the Encryption Key for a New Table \(p. 818\)](#)
- [Updating an Encryption Key \(p. 821\)](#)

Specifying the Encryption Key for a New Table

Follow these steps to specify the encryption key on a new table using the Amazon DynamoDB console or the AWS CLI.

[Creating an Encrypted Table \(Console\)](#)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose **Create Table**. For the **Table name**, enter **Music**. For the primary key, enter **Artist**, and for the sort key, enter **SongTitle**, both as strings.
4. In **Table settings**, make sure that **Use default settings** is not selected.

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

Use default settings

Note

If **Use default settings** is selected, tables are encrypted at rest with the AWS owned customer master key (CMK) at no additional cost.

5. Under **Encryption at rest**, choose an encryption type:

- **Default** – AWS owned CMK. The key is owned by DynamoDB (no additional charge).
- **KMS** – Customer managed CMK. The key is stored in your account and is created, owned, and managed by you (AWS KMS charges apply).
- **KMS** – AWS managed CMK. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS charges apply).

Encryption At Rest

Select Server-side encryption settings for your DynamoDB table to help protect data at rest. [Learn more](#)

DEFAULT

The key is owned by Amazon DynamoDB. You are not charged any fee for using these CMKs.

KMS - Customer managed CMK

The key is stored in your account that you create, own, and manage. AWS Key Management Service (KMS) charges apply. [Learn more](#)

KMS - AWS managed CMK

The key is stored in your account and is managed by AWS Key Management Service (KMS). AWS KMS charges apply.

6. Choose **Create** to create the encrypted table. To confirm the encryption type, check the table details on the **Overview** tab.

[Creating an Encrypted Table \(AWS CLI\)](#)

Use the AWS CLI to create a table with the default AWS owned CMK, the AWS managed CMK, or a customer managed CMK for Amazon DynamoDB.

To create an encrypted table with the default AWS owned CMK

- Create the encrypted Music table as follows.

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

```

```
AttributeName=Artist,KeyType=HASH \
AttributeName=SongTitle,KeyType=RANGE \
--provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5
```

Note

This table is now encrypted using the default AWS owned CMK in the DynamoDB service account.

To create an encrypted table with the AWS managed CMK for DynamoDB

- Create the encrypted Music table as follows.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
    AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
  --key-schema \
    AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --sse-specification Enabled=true,SSEType=KMS
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234alb234",
}
```

To create an encrypted table with a customer managed CMK for DynamoDB

- Create the encrypted Music table as follows.

```
aws dynamodb create-table \
  --table-name Music \
  --attribute-definitions \
    AttributeName=Artist,AttributeType=S \
    AttributeName=SongTitle,AttributeType=S \
  --key-schema \
    AttributeName=Artist,KeyType=HASH \
    AttributeName=SongTitle,KeyType=RANGE \
  --provisioned-throughput \
    ReadCapacityUnits=10,WriteCapacityUnits=5 \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-a123-
ab1234alb234
```

The SSEDescription status of the table description is set to ENABLED and the SSEType is KMS.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234alb234",
```

}

Updating an Encryption Key

You can also use the DynamoDB console or the AWS CLI to update the encryption keys of an existing table between an AWS owned CMK, AWS managed CMK, and customer managed CMK at any time.

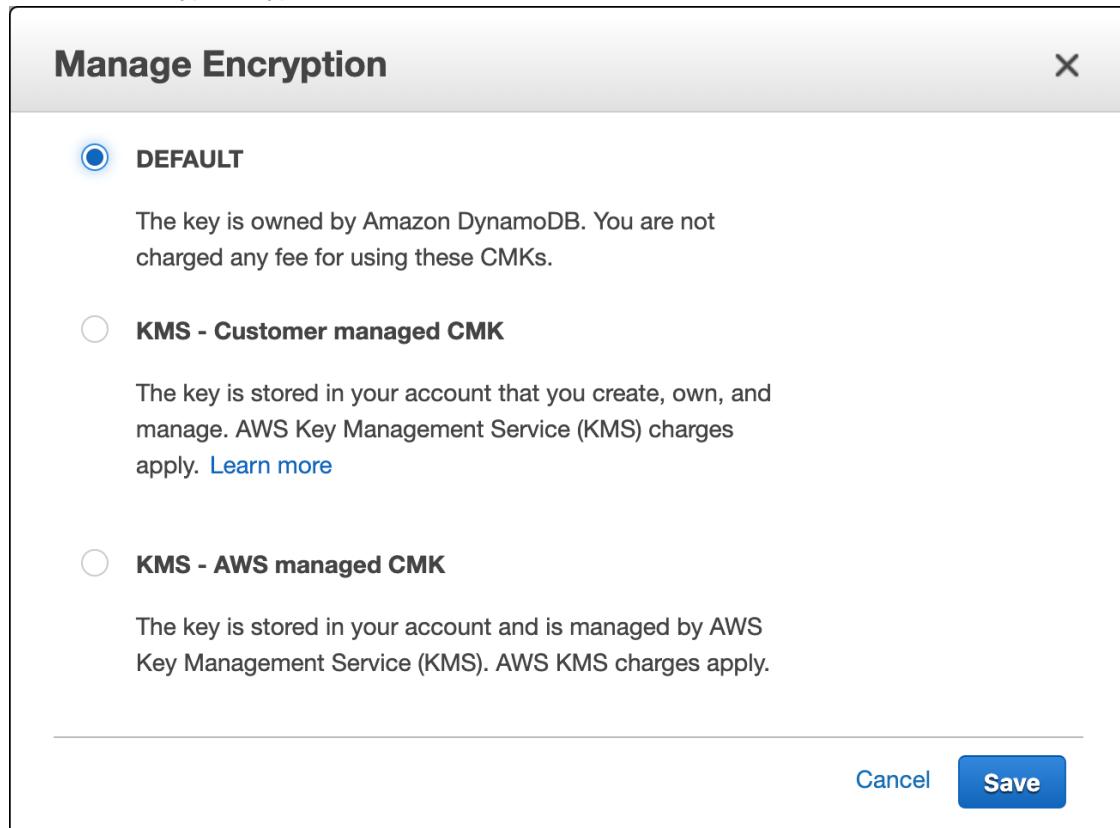
Updating an Encryption Key (Console)

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Tables**.
3. Choose the table that you want to update, and then choose the **Overview** tab.
4. Choose **Manage Encryption**.

Table details

| | |
|------------------------|---|
| Table name | Music |
| Primary partition key | Artist (String) |
| Primary sort key | SongTitle (String) |
| Point-in-time recovery | ENABLED Disable |
| Encryption Type | DEFAULT Manage Encryption |

5. Choose an encryption type.



- **Default** – AWS owned CMK. The key is owned by DynamoDB (no additional charge).

- **KMS** – Customer managed CMK. The key is stored in your account and is created, owned, and managed by you (AWS KMS charges apply).
- **KMS** – AWS managed CMK. The key is stored in your account and is managed by AWS Key Management Service (AWS KMS charges apply).

Then choose **Save** to update the encrypted table. To confirm the encryption type, check the table details under the **Overview** tab.

Updating an Encryption Key (AWS CLI)

The following examples show how to update an encrypted table using the AWS CLI.

To update an encrypted table with the default AWS owned CMK

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=false
```

Note

This table is now encrypted using the default AWS owned CMK in the DynamoDB service account.

To update an encrypted table with the AWS managed CMK for DynamoDB

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true
```

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
  "SSEType": "KMS",
  "Status": "ENABLED",
  "KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234a1b234",
}
```

To update an encrypted table with a customer managed CMK for DynamoDB

- Update the encrypted `Music` table, as in the following example.

```
aws dynamodb update-table \
  --table-name Music \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=abcd1234-abcd-1234-a123-
ab1234a1b234
```

The `SSEDescription` status of the table description is set to `ENABLED` and the `SSEType` is `KMS`.

```
"SSEDescription": {
```

```
"SSEType": "KMS",
"Status": "ENABLED",
"KMSMasterKeyArn": "arn:aws:kms:us-east-1:123456789012:key/abcd1234-abcd-1234-a123-
ab1234a1b234",
}
```

Data Protection in DynamoDB Accelerator

Amazon DynamoDB Accelerator (DAX) encryption at rest provides an additional layer of data protection by helping secure your data from unauthorized access to the underlying storage. Organizational policies, industry or government regulations, and compliance requirements might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of your applications that are deployed in the cloud.

For more information about data protection in DAX, see [DAX Encryption at Rest \(p. 755\)](#).

Internet Traffic Privacy

Connections are protected both between Amazon DynamoDB and on-premises applications and between DynamoDB and other AWS resources within the same AWS Region.

Traffic Between Service and On-Premises Clients and Applications

You have two connectivity options between your private network and AWS:

- An AWS Site-to-Site VPN connection. For more information, see [What is AWS Site-to-Site VPN?](#) in the [AWS Site-to-Site VPN User Guide](#).
- An AWS Direct Connect connection. For more information, see [What is AWS Direct Connect?](#) in the [AWS Direct Connect User Guide](#).

Access to DynamoDB via the network is through AWS published APIs. Clients must support Transport Layer Security (TLS) 1.0. We recommend TLS 1.2 or above. Clients must also support cipher suites with Perfect Forward Secrecy (PFS), such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, you must sign requests using an access key ID and a secret access key that are associated with an IAM principal, or you can use the [AWS Security Token Service \(STS\)](#) to generate temporary security credentials to sign requests.

Traffic Between AWS Resources in the Same Region

An Amazon Virtual Private Cloud (Amazon VPC) endpoint for DynamoDB is a logical entity within a VPC that allows connectivity only to DynamoDB. The Amazon VPC routes requests to DynamoDB and routes responses back to the VPC. For more information, see [VPC Endpoints](#) in the [Amazon VPC User Guide](#). For example policies that you can use to control access from VPC endpoints, see [Using IAM Policies to Control Access to DynamoDB](#).

Identity and Access Management

IAM administrators control who can be authenticated (signed in) and authorized (have permissions) to use Amazon DynamoDB resources. You can use AWS Identity and Access Management (IAM) to

manage access permissions and implement security policies for both Amazon DynamoDB and DynamoDB Accelerator (DAX).

Topics

- [Identity and Access Management in Amazon DynamoDB \(p. 824\)](#)
- [Identity and Access Management in DynamoDB Accelerator \(p. 855\)](#)

Identity and Access Management in Amazon DynamoDB

Access to Amazon DynamoDB requires credentials. Those credentials must have permissions to access AWS resources, such as an Amazon DynamoDB table or an Amazon Elastic Compute Cloud (Amazon EC2) instance. The following sections provide details on how you can use [AWS Identity and Access Management \(IAM\)](#) and DynamoDB to help secure access to your resources.

- [Authentication \(p. 824\)](#)
- [Access Control \(p. 825\)](#)

Authentication

You can access AWS as any of the following types of identities:

- **AWS account root user** – When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the [best practice of using the root user only to create your first IAM user](#). Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.
- **IAM user** – An [IAM user](#) is an identity within your AWS account that has specific custom permissions (for example, permissions to create a table in DynamoDB). You can use an IAM user name and password to sign in to secure AWS webpages like the [AWS Management Console](#), [AWS Discussion Forums](#), or the [AWS Support Center](#).

In addition to a user name and password, you can also generate [access keys](#) for each user. You can use these keys when you access AWS services programmatically, either through [one of the several SDKs](#) or by using the [AWS Command Line Interface \(CLI\)](#). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don't use AWS tools, you must sign the request yourself. DynamoDB supports *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see [Signature Version 4 Signing Process](#) in the [AWS General Reference](#).

- **IAM role** – An [IAM role](#) is an IAM identity that you can create in your account that has specific permissions. An IAM role is similar to an IAM user in that it is an AWS identity with permissions policies that determine what the identity can and cannot do in AWS. However, instead of being uniquely associated with one person, a role is intended to be assumable by anyone who needs it. Also, a role does not have standard long-term credentials such as a password or access keys associated with it. Instead, when you assume a role, it provides you with temporary security credentials for your role session. IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an [identity provider](#). For more information about federated users, see [Federated Users and Roles](#) in the *IAM User Guide*.
- **AWS service access** – A service role is an IAM role that a service assumes to perform actions in your account on your behalf. When you set up some AWS service environments, you must define a role for the service to assume. This service role must include all the permissions that are required for the service to access the AWS resources that it needs. Service roles vary from service to service, but many allow you to choose your permissions as long as you meet the documented requirements for that service. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. You can create, modify, and delete a service role from within IAM. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see [Creating a Role to Delegate Permissions to an AWS Service](#) in the *IAM User Guide*.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances](#) in the *IAM User Guide*.

Access Control

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access Amazon DynamoDB resources. For example, you must have permissions to create an Amazon DynamoDB table.

The following sections describe how to manage permissions for Amazon DynamoDB. We recommend that you read the overview first.

- [Overview of Managing Access \(p. 825\)](#)
- [Using Identity-Based Policies \(IAM Policies\) \(p. 828\)](#)
- [DynamoDB API Permissions Reference \(p. 835\)](#)
- [Using Conditions \(p. 840\)](#)

Overview of Managing Access Permissions to Your Amazon DynamoDB Resources

Every AWS resource is owned by an AWS account, and permissions to create or access a resource are governed by permissions policies. An account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles), and some services (such as AWS Lambda) also support attaching permissions policies to resources.

Note

An *account administrator* (or administrator user) is a user with administrator privileges. For more information, see [IAM Best Practices](#) in the *IAM User Guide*.

When granting permissions, you decide who is getting the permissions, the resources they get permissions for, and the specific actions that you want to allow on those resources.

Topics

- [DynamoDB Resources and Operations \(p. 826\)](#)
- [Understanding Resource Ownership \(p. 826\)](#)
- [Managing Access to Resources \(p. 826\)](#)
- [Specifying Policy Elements: Actions, Effects, and Principals \(p. 828\)](#)
- [Specifying Conditions in a Policy \(p. 828\)](#)

DynamoDB Resources and Operations

In DynamoDB, the primary resources are *tables*. DynamoDB also supports additional resource types, *indexes*, and *streams*. However, you can create indexes and streams only in the context of an existing DynamoDB table. These are referred to as *subresources*.

These resources and subresources have unique Amazon Resource Names (ARNs) associated with them, as shown in the following table.

| Resource Type | ARN Format |
|---------------|--|
| Table | <code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i></code> |
| Index | <code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i>/index/<i>index-name</i></code> |
| Stream | <code>arn:aws:dynamodb:<i>region</i>:<i>account-id</i>:table/<i>table-name</i>/stream/<i>stream-label</i></code> |

DynamoDB provides a set of operations to work with DynamoDB resources. For a list of available operations, see [Amazon DynamoDB Actions](#).

Understanding Resource Ownership

The AWS account owns the resources that are created in the account, regardless of who created the resources. Specifically, the resource owner is the AWS account of the [principal entity](#) (that is, the AWS account root user, an IAM user, or an IAM role) that authenticates the resource creation request. The following examples illustrate how this works:

- If you use your AWS account root user credentials to create a table, your AWS account is the owner of the resource (in DynamoDB, the resource is a table).
- If you create an IAM user in your AWS account and grant permissions to create a table to that user, the user can create a table. However, your AWS account, to which the user belongs, owns the table resource.
- If you create an IAM role in your AWS account with permissions to create a table, anyone who can assume the role can create a table. Your AWS account, to which the role belongs, owns the table resource.

Managing Access to Resources

A *permissions policy* describes who has access to what. The following section explains the available options for creating permissions policies.

Note

This section discusses using IAM in the context of DynamoDB. It doesn't provide detailed information about the IAM service. For complete IAM documentation, see [What Is IAM?](#) in the

IAM User Guide. For information about IAM policy syntax and descriptions, see [AWS IAM Policy Reference](#) in the *IAM User Guide*.

Policies attached to an IAM identity are referred to as *identity-based* policies (IAM policies). Policies attached to a resource are referred to as *resource-based* policies. DynamoDB supports only identity-based policies (IAM policies).

Topics

- [Identity-Based Policies \(IAM Policies\) \(p. 827\)](#)
- [Resource-Based Policies \(p. 828\)](#)

Identity-Based Policies (IAM Policies)

You can attach policies to IAM identities. For example, you can do the following:

- **Attach a permissions policy to a user or a group in your account** – To grant a user permissions to create an Amazon DynamoDB resource, such as a table, you can attach a permissions policy to a user or group that the user belongs to.
- **Attach a permissions policy to a role (grant cross-account permissions)** – You can attach an identity-based permissions policy to an IAM role to grant cross-account permissions. For example, the administrator in account A can create a role to grant cross-account permissions to another AWS account (for example, account B) or an AWS service as follows:
 1. Account A administrator creates an IAM role and attaches a permissions policy to the role that grants permissions on resources in account A.
 2. Account A administrator attaches a trust policy to the role identifying account B as the principal who can assume the role.
 3. Account B administrator can then delegate permissions to assume the role to any users in account B. Doing this allows users in account B to create or access resources in account A. The principal in the trust policy can also be an AWS service principal if you want to grant an AWS service permissions to assume the role.

For more information about using IAM to delegate permissions, see [Access Management](#) in the *IAM User Guide*.

The following is an example policy that grants permissions for one DynamoDB action (`dynamodb>ListTables`). The wildcard character (*) in the Resource value means that you can use this action to obtain the names of all the tables owned by the AWS account in the current AWS Region.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ListTables",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb>ListTables"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

For more information about using identity-based policies with DynamoDB, see [Using Identity-Based Policies \(IAM Policies\) for Amazon DynamoDB \(p. 828\)](#). For more information about users, groups, roles, and permissions, see [Identities \(Users, Groups, and Roles\)](#) in the *IAM User Guide*.

Resource-Based Policies

Other services, such as Amazon S3, also support resource-based permissions policies. For example, you can attach a policy to an S3 bucket to manage access permissions to that bucket. DynamoDB doesn't support resource-based policies.

Specifying Policy Elements: Actions, Effects, and Principals

For each DynamoDB resource, the service defines a set of API operations. To grant permissions for these API operations, DynamoDB defines a set of actions that you can specify in a policy. Some API operations can require permissions for more than one action in order to perform the API operation. For more information about resources and API operations, see [DynamoDB Resources and Operations \(p. 826\)](#) and [DynamoDB Actions](#).

The following are the most basic policy elements:

- **Resource** – You use an Amazon Resource Name (ARN) to identify the resource that the policy applies to. For more information, see [DynamoDB Resources and Operations \(p. 826\)](#).
- **Action** – You use action keywords to identify resource operations that you want to allow or deny. For example, `dynamodb:Query` allows the user permissions to perform the DynamoDB `Query` operation.
- **Effect** – You specify the effect, either allow or deny, when the user requests the specific action. If you don't explicitly grant access to (allow) a resource, access is implicitly denied. You can also explicitly deny access to a resource, which you might do to make sure that a user cannot access it, even if a different policy grants access.
- **Principal** – In identity-based policies (IAM policies), the user that the policy is attached to is the implicit principal. For resource-based policies, you specify the user, account, service, or other entity that you want to receive permissions (applies to resource-based policies only). DynamoDB doesn't support resource-based policies.

To learn more about IAM policy syntax and descriptions, see [AWS IAM Policy Reference](#) in the *IAM User Guide*.

For a list showing all of the Amazon DynamoDB API operations and the resources that they apply to, see [DynamoDB API Permissions: Actions, Resources, and Conditions Reference \(p. 835\)](#).

Specifying Conditions in a Policy

When you grant permissions, you can use the access policy language to specify the conditions when a policy should take effect. For example, you might want a policy to be applied only after a specific date. For more information about specifying conditions in a policy language, see [Condition](#) in the *IAM User Guide*.

To express conditions, you use predefined condition keys. There are AWS-wide condition keys and DynamoDB-specific keys that you can use as appropriate. For a complete list of AWS-wide keys, see [Available Keys for Conditions](#) in the *IAM User Guide*. For a complete list of DynamoDB-specific keys, see [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 840\)](#).

Using Identity-Based Policies (IAM Policies) for Amazon DynamoDB

This topic provides examples of identity-based policies that demonstrate how an account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on Amazon DynamoDB resources.

Important

We recommend that you first review the introductory topics that explain the basic concepts and options available to manage access to your Amazon DynamoDB resources. For more

information, see [Overview of Managing Access Permissions to Your Amazon DynamoDB Resources \(p. 825\)](#).

The sections in this topic cover the following:

- [Permissions Required to Use the Amazon DynamoDB Console \(p. 829\)](#)
- [AWS Managed \(Predefined\) Policies for Amazon DynamoDB \(p. 829\)](#)
- [Customer Managed Policy Examples \(p. 830\)](#)

The following shows an example of a permissions policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DescribeQueryScanBooksTable",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:DescribeTable",  
                "dynamodb:Query",  
                "dynamodb:Scan"  
            ],  
            "Resource": "arn:aws:dynamodb:us-west-2:account-id:table/Books"  
        }  
    ]  
}
```

The policy has one statement that grants permissions for three DynamoDB actions (`dynamodb:DescribeTable`, `dynamodb:Query`, and `dynamodb:Scan`) on a table in the `us-west-2` Region, which is owned by the AWS account specified by `account-id`. The *Amazon Resource Name (ARN)* in the `Resource` value specifies the table to which the permissions apply.

Permissions Required to Use the Amazon DynamoDB Console

For a user to work with the DynamoDB console, that user must have a minimum set of permissions that allow the user to work with the DynamoDB resources for their AWS account. In addition to these DynamoDB permissions, the console requires permissions from the following services:

- Amazon CloudWatch permissions to display metrics and graphs.
- AWS Data Pipeline permissions to export and import DynamoDB data.
- AWS Identity and Access Management permissions to access roles necessary for exports and imports.
- Amazon Simple Notification Service permissions to notify you whenever a CloudWatch alarm is triggered.
- AWS Lambda permissions to process DynamoDB Streams records.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console won't function as intended for users with that IAM policy. To ensure that those users can still use the DynamoDB console, also attach the `AmazonDynamoDBReadOnlyAccess` managed policy to the user, as described in [AWS Managed \(Predefined\) Policies for Amazon DynamoDB \(p. 829\)](#).

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the Amazon DynamoDB API.

AWS Managed (Predefined) Policies for Amazon DynamoDB

AWS addresses many common use cases by providing standalone IAM policies that are created and administered by AWS. These AWS managed policies grant necessary permissions for common use cases

so that you can avoid having to investigate what permissions are needed. For more information, see [AWS Managed Policies](#) in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to DynamoDB and are grouped by use case scenario:

- **AmazonDynamoDBReadOnlyAccess** – Grants read-only access to DynamoDB resources by using the AWS Management Console.
- **AmazonDynamoDBFullAccess** – Grants full access to DynamoDB resources by using the AWS Management Console.
- **AmazonDynamoDBFullAccesswithDataPipeline** – Grants full access to DynamoDB resources, including export and import using AWS Data Pipeline, by using AWS Management Console.

Note

You can review these permissions policies by signing in to the IAM console and searching for specific policies there.

We do not recommend using these AWS managed policies for production databases. Instead, you should create your own custom IAM policies to allow permissions for DynamoDB actions and resources following the [least privilege model](#). You can attach these custom policies to the IAM users or groups that require those permissions.

Customer Managed Policy Examples

In this section, you can find example user policies that grant permissions for various DynamoDB actions. These policies work when you are using AWS SDKs or the AWS CLI. When you are using the console, you need to grant additional permissions specific to the console, which is discussed in [Permissions Required to Use the Amazon DynamoDB Console \(p. 829\)](#).

Note

All examples use the us-west-2 Region and contain fictitious account IDs.

Examples

- [Example 1: Allow a User to Perform Any DynamoDB Actions on a Table \(p. 830\)](#)
- [Example 2: Allow Read-Only Access on Items in a Table \(p. 831\)](#)
- [Example 3: Allow Put, Update, and Delete Operations on a Specific Table \(p. 831\)](#)
- [Example 4: Allow Access to a Specific Table and All of Its Indexes \(p. 831\)](#)
- [Example 5: Set Up Permissions Policies for Separate Test and Production Environments \(p. 832\)](#)
- [Example 6: Prevent a User from Purchasing Reserved Capacity Offerings \(p. 833\)](#)
- [Example 7: Allow Read Access for a DynamoDB Stream Only \(Not for the Table\) \(p. 834\)](#)
- [Example 8: Allow an AWS Lambda Function to Process DynamoDB Stream Records \(p. 835\)](#)

Example 1: Allow a User to Perform Any DynamoDB Actions on a Table

The following permissions policy grants permissions for all DynamoDB actions on a table. The ARN value specified in the `Resource` identifies a table in a specific Region.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllAPIActionsOnBooks",  
            "Effect": "Allow",  
            "Action": "dynamodb:*",  
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"  
        }  
    ]  
}
```

```
    ]
}
```

Note

If you replace the table name in the resource ARN (`Books`) with a wildcard character (*), you allow any DynamoDB actions on *all* tables in the account. Carefully consider the security implications if you decide to do this.

Example 2: Allow Read-Only Access on Items in a Table

The following permissions policy grants permissions for the `GetItem` and `BatchGetItem` DynamoDB actions only and thereby sets read-only access to a table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAPIActionsOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Example 3: Allow Put, Update, and Delete Operations on a Specific Table

The following permissions policy grants permissions for the `PutItem`, `UpdateItem`, and `DeleteItem` actions on a specific DynamoDB table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PutUpdateDeleteOnBooks",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/Books"
        }
    ]
}
```

Example 4: Allow Access to a Specific Table and All of Its Indexes

The following permissions policy grants permissions for all of the DynamoDB actions on a table (`Book`) and all of the table's indexes. For more information about how indexes work, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AccessAllIndexesOnBooks",

```

```

        "Effect": "Allow",
        "Action": [
            "dynamodb:*"
        ],
        "Resource": [
            "arn:aws:dynamodb:us-west-2:123456789012:table/Books",
            "arn:aws:dynamodb:us-west-2:123456789012:table/Books/index/*"
        ]
    }
}

```

Example 5: Set Up Permissions Policies for Separate Test and Production Environments

Suppose that you have separate test and production environments where each environment maintains its own version of a table named `ProductCatalog`. If you create these `ProductCatalog` tables from the same AWS account, testing work might affect the production environment because of the way that permissions are set up. (For example, the limits on concurrent create and delete actions are set at the AWS account level.) As a result, each action in the test environment reduces the number of actions that are available in your production environment. There is also a risk that the code in your test environment might accidentally access tables in the production environment. To prevent these issues, consider creating separate AWS accounts for your production and test environments.

Suppose further that you have two developers, Bob and Alice, who are testing the `ProductCatalog` table. Instead of creating a separate AWS account for every developer, your developers can share the same test account. In this test account, you can create a copy of the same table for each developer to work on, such as `Alice_ProductCatalog` and `Bob_ProductCatalog`. In this case, you can create IAM users Alice and Bob in the AWS account that you created for the test environment. You can then grant permissions to these users to perform DynamoDB actions on the tables that they own.

To grant these user permissions, you can do either of the following:

- Create a separate policy for each user and then attach each policy to its user separately. For example, you can attach the following policy to user Alice to allow her access to all DynamoDB actions on the `Alice_ProductCatalog` table:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnAliceTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/
Alice_ProductCatalog"
        }
    ]
}

```

Then, you can create a similar policy with a different resource (`Bob_ProductCatalog` table) for user Bob.

- Instead of attaching policies to individual users, you can use IAM policy variables to write a single policy and attach it to a group. You need to create a group and, for this example, add both users Alice and user Bob to the group. The following example grants permissions to perform all DynamoDB actions on the `$(aws:username)_ProductCatalog` table. The policy variable `$(aws:username)` is replaced by the requester's user name when the policy is evaluated. For example, if Alice sends a request to add an item, the action is allowed only if Alice is adding items to the `Alice_ProductCatalog` table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnUserSpecificTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_ProductCatalog"
        },
        {
            "Sid": "AdditionalPrivileges",
            "Effect": "Allow",
            "Action": [
                "dynamodb>ListTables",
                "dynamodb:DescribeTable",
                "cloudwatch:*",
                "sns:)"
            ],
            "Resource": "*"
        }
    ]
}
```

Note

When using IAM policy variables, you must explicitly specify the 2012-10-17 version of the access policy language in the policy. The default version of the access policy language (2008-10-17) does not support policy variables.

Instead of identifying a specific table as a resource, you can use a wildcard character (*) to grant permissions on all tables where the name is prefixed with the name of the IAM user that is making the request, as shown following.

```
"Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/${aws:username}_*"
```

Example 6: Prevent a User from Purchasing Reserved Capacity Offerings

DynamoDB customers can purchase reserved capacity, as described at [Amazon DynamoDB Pricing](#). With reserved capacity, you pay a one-time upfront fee and commit to paying for a minimum usage level, at significant savings, over a period of time. You can use the AWS Management Console to view and purchase reserved capacity. However, you might not want all of the users in your organization to have the same levels of access.

DynamoDB provides the following API operations for controlling access to reserved capacity management:

- `dynamodb:DescribeReservedCapacity` – returns the reserved capacity purchases that are currently in effect.
- `dynamodb:DescribeReservedCapacityOfferings` – returns details about the reserved capacity plans that are currently offered by AWS.
- `dynamodb:PurchaseReservedCapacityOfferings` – performs an actual purchase of reserved capacity.

The AWS Management Console uses these API operations to display reserved capacity information and to make purchases. You cannot call these operations from an application program, because they can be

accessed only from the console. However, you can allow or deny access to these operations in an IAM permissions policy.

The following policy allows users to view reserved capacity offerings and current purchases using the AWS Management Console—but new purchases are denied.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReservedCapacityDescriptions",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeReservedCapacity",
                "dynamodb:DescribeReservedCapacityOfferings"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:)"
        },
        {
            "Sid": "DenyReservedCapacityPurchases",
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:)"
        }
    ]
}
```

Example 7: Allow Read Access for a DynamoDB Stream Only (Not for the Table)

When you enable DynamoDB Streams on a table, it captures information about every modification to data items in the table. For more information, see [Capturing Table Activity with DynamoDB Streams \(p. 573\)](#).

In some cases, you might want to prevent an application from reading data from a DynamoDB table, while still allowing access to that table's stream. For example, you can configure AWS Lambda to poll the stream and invoke a Lambda function when item updates are detected, and then perform additional processing.

The following actions are available for controlling access to DynamoDB Streams:

- dynamodb:DescribeStream
- dynamodb:GetRecords
- dynamodb:GetShardIterator
- dynamodb>ListStreams

The following example creates a policy that grants users permissions to access the streams on a table named `GameScores`. The final wildcard character (*) in the ARN matches any stream ID associated with that table.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AccessGameScoresStreamOnly",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeStream",
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:ListStreams"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:stream/GameScores*"
        }
    ]
}
```

```

        "dynamodb>ListStreams"
    ],
    "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/stream/*"
}
]
}
}

```

Note that this policy permits access to the streams on the `GameScores` table, but not to the table itself.

Example 8: Allow an AWS Lambda Function to Process DynamoDB Stream Records

If you want certain actions to be performed based on new events in a DynamoDB stream, you can write an AWS Lambda function that is triggered by these new events. For more information about using Lambda with stream events, see [DynamoDB Streams and AWS Lambda Triggers \(p. 594\)](#). A Lambda function such as this needs permissions to read data from the DynamoDB stream.

To grant permissions to Lambda, you use the permissions policy that is associated with the Lambda function's IAM role (execution role), which you specify when you create the Lambda function.

For example, you can associate the following permissions policy with the execution role to grant Lambda permissions to perform the DynamoDB Streams actions listed.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowLambdaFunctionInvocation",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "AllAPIAccessForDynamoDBStreams",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetRecords",
                "dynamodb:GetShardIterator",
                "dynamodb:DescribeStream",
                "dynamodb>ListStreams"
            ],
            "Resource": "*"
        }
    ]
}

```

For more information, see [AWS Lambda Permission Model](#) in the *AWS Lambda Developer Guide*.

DynamoDB API Permissions: Actions, Resources, and Conditions Reference

When you are setting up [Access Control \(p. 825\)](#) and writing a permissions policy that you can attach to an IAM identity (identity-based policies), you can use the following list as a reference. The list includes each DynamoDB API operation, the corresponding actions for which you can grant permissions to perform the action, and the AWS resource for which you can grant the permissions. You specify the actions in the policy's `Action` field, and you specify the resource value in the policy's `Resource` field.

You can use AWS-wide condition keys in your DynamoDB policies to express conditions. For a complete list of AWS-wide keys, see [Available Keys](#) in the *IAM User Guide*.

In addition to the AWS-wide condition keys, DynamoDB has its own specific keys that you can use in conditions. For more information, see [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 840\)](#).

Note

To specify an action, use the dynamodb: prefix followed by the API operation name (for example, dynamodb:CreateTable).

DynamoDB API Permissions: Actions, Resources, and Condition Keys Reference

BatchGetItem

Action(s): dynamodb:BatchGetItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

BatchWriteItem

Action(s): dynamodb:BatchWriteItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

ConditionCheck

Action(s): dynamodb:ConditionCheck

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

CreateTable

Action(s): dynamodb:CreateTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

DeleteItem

Action(s): dynamodb:DeleteItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

DeleteTable

Action(s): dynamodb:DeleteTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

DescribeReservedCapacity

Action(s): dynamodb:DeleteTable

Resource:

`arn:aws:dynamodb:region:account-id:*`

DescribeReservedCapacityOfferings

Action(s): dynamodb:DescribeReservedCapacityOfferings

Resource:

`arn:aws:dynamodb:region:account-id:*`

DescribeStream

Action(s): dynamodb:DescribeStream

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

DescribeTable

Action(s): dynamodb:DescribeTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

EnclosingOperation

Action(s): dynamodb:EnclosingOperation

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[GetItem](#)

Action(s): dynamodb:GetItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

[GetRecords](#)

Action(s): dynamodb:GetRecords

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

[GetShardIterator](#)

Action(s): dynamodb:GetShardIterator

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label`

or

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

[ListStreams](#)

Action(s): dynamodb>ListStreams

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name/stream/*`

or

`arn:aws:dynamodb:region:account-id:table/*/stream/*`

[ListTables](#)

Action(s): dynamodb>ListTables

Resource:

*

PurchaseReservedCapacityOfferings

Action(s): dynamodb:PurchaseReservedCapacityOfferings

Resource:

arn:aws:dynamodb:*region*:*account-id*:*

PutItem

Action(s): dynamodb:PutItem

Resource:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*

or

arn:aws:dynamodb:*region*:*account-id*:table/*

Query

Action(s): dynamodb:Query

Resource:

To query a table:arn:aws:dynamodb:*region*:*account-id*:table/*table-name*

or:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*

To query an index:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*index-name*

or:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*

Scan

Action(s): dynamodb:Scan

Resource:

To scan a table:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*

or:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*

To scan an index:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*index-name*

or:

arn:aws:dynamodb:*region*:*account-id*:table/*table-name*/index/*

UpdateItem

Action(s): dynamodb:UpdateItem

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

UpdateTable

Action(s): dynamodb:UpdateTable

Resource:

`arn:aws:dynamodb:region:account-id:table/table-name`

or

`arn:aws:dynamodb:region:account-id:table/*`

Related Topics

- [Access Control \(p. 825\)](#)
- [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 840\)](#)

Using IAM Policy Conditions for Fine-Grained Access Control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect.

Overview

In DynamoDB, you have the option to specify conditions when granting permissions using an IAM policy (see [Access Control \(p. 825\)](#)). For example, you can:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

In DynamoDB, you can specify conditions in an IAM policy using condition keys, as illustrated in the use case in the following section.

Note

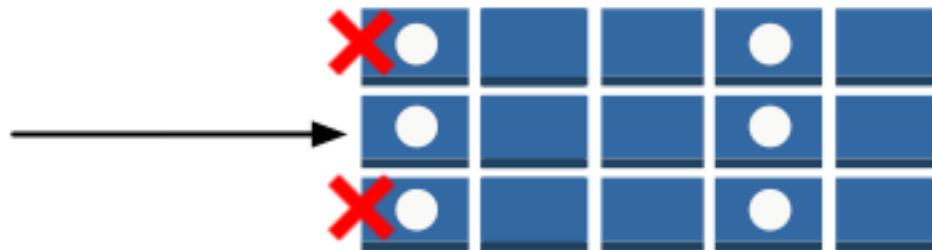
Some AWS services also support tag-based conditions; however, DynamoDB does not.

Permissions Use Case

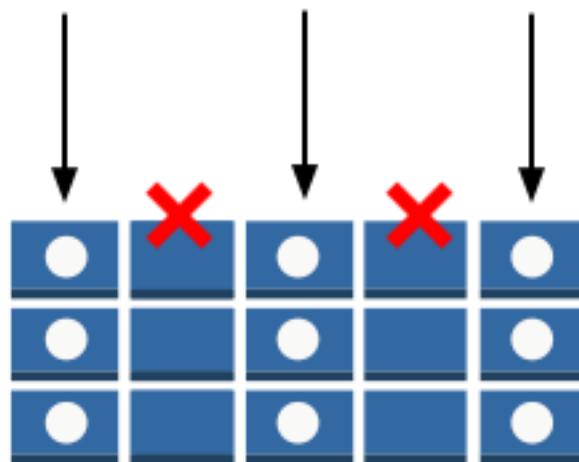
In addition to controlling access to DynamoDB API actions, you can also control access to individual data items and attributes. For example, you can do the following:

- Grant permissions on a table, but restrict access to specific items in that table based on certain primary key values. An example might be a social networking app for games, where all users' saved game data

is stored in a single table, but no users can access data items that they do not own, as shown in the following illustration:



- Hide information so that only a subset of attributes is visible to the user. An example might be an app that displays flight data for nearby airports, based on the user's location. Airline names, arrival and departure times, and flight numbers are all displayed. However, attributes such as pilot names or the number of passengers are hidden, as shown in the following illustration:



To implement this kind of fine-grained access control, you write an IAM permissions policy that specifies conditions for accessing security credentials and the associated permissions. You then apply the policy to IAM users, groups, or roles that you create using the IAM console. Your IAM policy can restrict access to individual items in a table, access to the attributes in those items, or both at the same time.

You can optionally use web identity federation to control access by users who are authenticated by Login with Amazon, Facebook, or Google. For more information, see [Using Web Identity Federation \(p. 850\)](#).

You use the IAM Condition element to implement a fine-grained access control policy. By adding a Condition element to a permissions policy, you can allow or deny access to items and attributes in DynamoDB tables and indexes, based upon your particular business requirements.

As an example, consider a mobile gaming app that lets players select from and play a variety of different games. The app uses a DynamoDB table named GameScores to keep track of high scores and other user data. Each item in the table is uniquely identified by a user ID and the name of the game that the user played. The GameScores table has a primary key consisting of a partition key (`UserId`) and sort key

(GameTitle). Users only have access to game data associated with their user ID. A user who wants to play a game must belong to an IAM role named GameRole, which has a security policy attached to it.

To manage user permissions in this app, you could write a permissions policy such as the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${www.amazon.com:user_id}"
                    ],
                    "dynamodb:Attributes": [
                        "UserId",
                        "GameTitle",
                        "Wins",
                        "Losses",
                        "TopScore",
                        "TopScoreDateTime"
                    ]
                },
                "StringEqualsIfExists": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

In addition to granting permissions for specific DynamoDB actions (**Action** element) on the GameScores table (**Resource** element), the **Condition** element uses the following condition keys specific to DynamoDB that limit the permissions as follows:

- **dynamodb:LeadingKeys** – This condition key allows users to access only the items where the partition key value matches their user ID. This ID, \${www.amazon.com:user_id}, is a substitution variable. For more information about substitution variables, see [Using Web Identity Federation \(p. 850\)](#).
- **dynamodb:Attributes** – This condition key limits access to the specified attributes so that only the actions listed in the permissions policy can return values for these attributes. In addition, the **StringEqualsIfExists** clause ensures that the app must always provide a list of specific attributes to act upon and that the app can't request all attributes.

When an IAM policy is evaluated, the result is always either true (access is allowed) or false (access is denied). If any part of the **Condition** element is false, the entire policy evaluates to false and access is denied.

Important

If you use `dynamodb:Attributes`, you must specify the names of all of the primary key and index key attributes for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

IAM policy documents can contain only the following Unicode characters: horizontal tab (U+0009), linefeed (U+000A), carriage return (U+000D), and characters in the range U+0020 to U+00FF.

Specifying Conditions: Using Condition Keys

AWS provides a set of predefined condition keys (AWS-wide condition keys) for all AWS services that support IAM for access control. For example, you can use the `aws:SourceIp` condition key to check the requester's IP address before allowing an action to be performed. For more information and a list of the AWS-wide keys, see [Available Keys for Conditions](#) in the IAM User Guide.

The following table shows the DynamoDB service-specific condition keys that apply to DynamoDB.

| DynamoDB Condition Key | Description |
|-----------------------------------|---|
| <code>dynamodb:LeadingKeys</code> | Represents the first key attribute of a table—in other words, the partition key. The key name <code>LeadingKeys</code> is plural, even if the key is used with single-item actions. In addition, you must use the <code>ForAllValues</code> modifier when using <code>LeadingKeys</code> in a condition. |
| <code>dynamodb:Select</code> | Represents the <code>Select</code> parameter of a <code>Query</code> or <code>Scan</code> request. <code>Select</code> can be any of the following values: <ul style="list-style-type: none"> • <code>ALL_ATTRIBUTES</code> • <code>ALL_PROJECTED_ATTRIBUTES</code> • <code>SPECIFIC_ATTRIBUTES</code> • <code>COUNT</code> |
| <code>dynamodb:Attributes</code> | Represents a list of the attribute names in a request, or the attributes that are returned from a request. <code>Attributes</code> values are named the same way and have the same meaning as the parameters for certain DynamoDB API actions, as shown following: <ul style="list-style-type: none"> • <code>AttributesToGet</code> <ul style="list-style-type: none"> Used by: <code>BatchGetItem</code>, <code>.GetItem</code>, <code>Query</code>, <code>Scan</code> • <code>AttributeUpdates</code> <ul style="list-style-type: none"> Used by: <code>UpdateItem</code> • <code>Expected</code> <ul style="list-style-type: none"> Used by: <code>DeleteItem</code>, <code>PutItem</code>, <code>UpdateItem</code> • <code>Item</code> <ul style="list-style-type: none"> Used by: <code>PutItem</code> • <code>ScanFilter</code> <ul style="list-style-type: none"> Used by: <code>Scan</code> |
| <code>dynamodb:ReturnValue</code> | Represents the <code>ReturnValues</code> parameter of a request. <code>ReturnValues</code> can be any of the following values: <ul style="list-style-type: none"> • <code>ALL_OLD</code> |

| DynamoDB Condition Key | Description |
|---------------------------------|--|
| | <ul style="list-style-type: none"> • UPDATED_OLD • ALL_NEW • UPDATED_NEW • NONE |
| dynamodb:ReturnConsumedCapacity | Represents the <code>ReturnConsumedCapacity</code> parameter of a request. <code>ReturnConsumedCapacity</code> can be one of the following values: <ul style="list-style-type: none"> • TOTAL • NONE |

Limits User Access

Many IAM permissions policies allow users to access only those items in a table where the partition key value matches the user identifier. For example, the game app preceding limits access in this way so that users can only access game data that is associated with their user ID. The IAM substitution variables `${www.amazon.com:user_id}`, `${graph.facebook.com:id}`, and `${accounts.google.com:sub}` contain user identifiers for Login with Amazon, Facebook, and Google. To learn how an application logs in to one of these identity providers and obtains these identifiers, see [Using Web Identity Federation \(p. 850\)](#).

Note

Each of the examples in the following section sets the `Effect` clause to `Allow` and specifies only the actions, resources, and parameters that are allowed. Access is permitted only to what is explicitly listed in the IAM policy.

In some cases, it is possible to rewrite these policies so that they are deny-based (that is, setting the `Effect` clause to `Deny` and inverting all of the logic in the policy). However, we recommend that you avoid using deny-based policies with DynamoDB because they are difficult to write correctly, compared to allow-based policies. In addition, future changes to the DynamoDB API (or changes to existing API inputs) can render a deny-based policy ineffective.

Example Policies: Using Conditions for Fine-Grained Access Control

This section shows several policies for implementing fine-grained access control on DynamoDB tables and indexes.

Note

All examples use the us-west-2 Region and contain fictitious account IDs.

1: Grant Permissions That Limit Access to Items with a Specific Partition Key Value

The following permissions policy grants permissions that allow a set of DynamoDB actions on the `GamesScore` table. It uses the `dynamodb:LeadingKeys` condition key to limit user actions only on the items whose `UserID` partition key value matches the Login with Amazon unique user ID for this app.

Important

The list of actions does not include permissions for `Scan` because `Scan` returns all items regardless of the leading keys.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "FullAccessToUserItems",
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:Query"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2::table/GamesScore/*"
      ],
      "Condition": {
        "dynamodb:LeadingKeys": {
          "Fn::Equals": {
            "S": "12345678901234567890123456789012"
          }
        }
      }
    }
  ]
}
```

```

        "dynamodb:BatchGetItem",
        "dynamodb:Query",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem",
        "dynamodb:BatchWriteItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "dynamodb:LeadingKeys": [
                "${www.amazon.com:user_id}"
            ]
        }
    }
}
]
}
}

```

Note

When using policy variables, you must explicitly specify version 2012-10-17 in the policy. The default version of the access policy language, 2008-10-17, does not support policy variables.

To implement read-only access, you can remove any actions that can modify the data. In the following policy, only those actions that provide read-only access are included in the condition.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyAccessToUserItems",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${www.amazon.com:user_id}"
                    ]
                }
            }
        }
    ]
}

```

Important

If you use `dynamodb:Attributes`, you must specify the names of all of the primary key and index key attributes, for the table and any secondary indexes that are listed in the policy. Otherwise, DynamoDB can't use these key attributes to perform the requested action.

2: Grant Permissions That Limit Access to Specific Attributes in a Table

The following permissions policy allows access to only two specific attributes in a table by adding the `dynamodb:Attributes` condition key. These attributes can be read, written, or evaluated in a conditional write or scan filter.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LimitAccessToSpecificAttributes",
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:BatchGetItem",
                "dynamodb:Scan"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:Attributes": [
                        "UserId",
                        "TopScore"
                    ]
                },
                "StringEqualsIfExists": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES",
                    "dynamodb:ReturnValues": [
                        "NONE",
                        "UPDATED_OLD",
                        "UPDATED_NEW"
                    ]
                }
            }
        }
    ]
}
```

Note

The policy takes an *allow list* (sometimes known as *whitelist*) approach, which allows access to a named set of attributes. You can write an equivalent policy that denies access to other attributes instead. We don't recommend this *deny list* (or *blacklist*) approach. Users can determine the names of these denied attributes by repeatedly issuing requests for all possible attribute names, eventually finding an attribute that they aren't allowed to access. To avoid this, follow the *principle of least privilege*, as explained in Wikipedia at http://en.wikipedia.org/wiki/Principle_of_least_privilege, and use an *allow list* approach to enumerate all of the allowed values, rather than specifying the denied attributes.

This policy doesn't permit `PutItem`, `DeleteItem`, or `BatchWriteItem`. These actions always replace the entire previous item, which would allow users to delete the previous values for attributes that they are not allowed to access.

The `StringEqualsIfExists` clause in the permissions policy ensures the following:

- If the user specifies the `Select` parameter, then its value must be `SPECIFIC_ATTRIBUTES`. This requirement prevents the API action from returning any attributes that aren't allowed, such as from an index projection.
- If the user specifies the `ReturnValues` parameter, then its value must be `NONE`, `UPDATED_OLD`, or `UPDATED_NEW`. This is required because the `UpdateItem` action also performs implicit read operations to check whether an item exists before replacing it, and so that previous attribute values can be returned if requested. Restricting `ReturnValues` in this way ensures that users can only read or write the allowed attributes.

- The `StringEqualsIfExists` clause assures that only one of these parameters — `Select` or `ReturnValues` — can be used per request, in the context of the allowed actions.

The following are some variations on this policy:

- To allow only read actions, you can remove `UpdateItem` from the list of allowed actions. Because none of the remaining actions accept `ReturnValues`, you can remove `ReturnValues` from the condition. You can also change `StringEqualsIfExists` to `StringEquals` because the `Select` parameter always has a value (`ALL_ATTRIBUTES`, unless otherwise specified).
- To allow only write actions, you can remove everything except `UpdateItem` from the list of allowed actions. Because `UpdateItem` does not use the `Select` parameter, you can remove `Select` from the condition. You must also change `StringEqualsIfExists` to `StringEquals` because the `ReturnValues` parameter always has a value (`NONE` unless otherwise specified).
- To allow all attributes whose name matches a pattern, use `StringLike` instead of `StringEquals`, and use a multi-character pattern match wildcard character (*).

[3: Grant Permissions to Prevent Updates on Certain Attributes](#)

The following permissions policy limits user access to updating only the specific attributes identified by the `dynamodb:Attributes` condition key. The `StringNotLike` condition prevents an application from updating the attributes specified using the `dynamodb:Attributes` condition key.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PreventUpdatesOnCertainAttributes",
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
            "Condition": {
                "ForAllValues:StringNotLike": {
                    "dynamodb:Attributes": [
                        "FreeGamesAvailable",
                        "BossLevelUnlocked"
                    ]
                },
                "StringEquals": {
                    "dynamodb:ReturnValues": [
                        "NONE",
                        "UPDATED_OLD",
                        "UPDATED_NEW"
                    ]
                }
            }
        }
    ]
}
```

Note the following:

- The `UpdateItem` action, like other write actions, requires read access to the items so that it can return values before and after the update. In the policy, you limit the action to accessing only the attributes that are allowed to be updated by specifying the `dynamodb:ReturnValues` condition key. The condition key restricts `ReturnValues` in the request to specify only `NONE`, `UPDATED_OLD`, or `UPDATED_NEW` and doesn't include `ALL_OLD` or `ALL_NEW`.

- The `PutItem` and `DeleteItem` actions replace an entire item, and thus allows applications to modify any attributes. So when limiting an application to updating only specific attributes, you should not grant permission for these APIs.

4: Grant Permissions to Query Only Projected Attributes in an Index

The following permissions policy allows queries on a secondary index (`TopScoreDateTimeIndex`) by using the `dynamodb:Attributes` condition key. The policy also limits queries to requesting only specific attributes that have been projected into the index.

To require the application to specify a list of attributes in the query, the policy also specifies the `dynamodb:Select` condition key to require that the `Select` parameter of the DynamoDB `query` action is `SPECIFIC_ATTRIBUTES`. The list of attributes is limited to a specific list that is provided using the `dynamodb:Attributes` condition key.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "QueryOnlyProjectedIndexAttributes",
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/TopScoreDateTimeIndex"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:Attributes": [
                        "TopScoreDateTime",
                        "GameTitle",
                        "Wins",
                        "Losses",
                        "Attempts"
                    ]
                },
                "StringEquals": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

The following permissions policy is similar, but the query must request all of the attributes that have been projected into the index.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "QueryAllIndexAttributes",
            "Effect": "Allow",
            "Action": [
                "dynamodb:Query"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/TopScoreDateTimeIndex"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:Attributes": [
                        "TopScoreDateTime",
                        "GameTitle",
                        "Wins",
                        "Losses",
                        "Attempts"
                    ]
                }
            }
        }
    ]
}
```

```

        "Condition": {
            "StringEquals": {
                "dynamodb>Select": "ALL_PROJECTED_ATTRIBUTES"
            }
        }
    ]
}

```

5: Grant Permissions to Limit Access to Certain Attributes and Partition Key Values

The following permissions policy allows specific DynamoDB actions (specified in the Action element) on a table and a table index (specified in the Resource element). The policy uses the dynamodb:LeadingKeys condition key to restrict permissions to only the items whose partition key value matches the user's Facebook ID.

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LimitAccessToCertainAttributesAndKeyValues",
            "Effect": "Allow",
            "Action": [
                "dynamodb:UpdateItem",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:BatchGetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores",
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores/index/
TopScoreDateTimeIndex"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${graph.facebook.com:id}"
                    ],
                    "dynamodb:Attributes": [
                        "attribute-A",
                        "attribute-B"
                    ]
                },
                "StringEqualsIfExists": {
                    "dynamodb>Select": "SPECIFIC_ATTRIBUTES",
                    "dynamodb:ReturnValues": [
                        "NONE",
                        "UPDATED_OLD",
                        "UPDATED_NEW"
                    ]
                }
            }
        }
    ]
}

```

Note the following:

- Write actions allowed by the policy (`UpdateItem`) can only modify `attribute-A` or `attribute-B`.
- Because the policy allows `UpdateItem`, an application can insert new items, and the hidden attributes will be null in the new items. If these attributes are projected into `TopScoreDateTimeIndex`, the policy has the added benefit of preventing queries that cause fetches from the table.

- Applications cannot read any attributes other than those listed in `dynamodb:Attributes`. With this policy in place, an application must set the `Select` parameter to `SPECIFIC_ATTRIBUTES` in read requests, and only whitelisted attributes can be requested. For write requests, the application cannot set `ReturnValues` to `ALL_OLD` or `ALL_NEW` and it cannot perform conditional write operations based on any other attributes.

Related Topics

- [Access Control \(p. 825\)](#)
- [DynamoDB API Permissions: Actions, Resources, and Conditions Reference \(p. 835\)](#)

Using Web Identity Federation

If you are writing an application targeted at large numbers of users, you can optionally use *web identity federation* for authentication and authorization. Web identity federation removes the need for creating individual IAM users. Instead, users can sign in to an identity provider and then obtain temporary security credentials from AWS Security Token Service (AWS STS). The app can then use these credentials to access AWS services.

Web identity federation supports the following identity providers:

- Login with Amazon
- Facebook
- Google

Additional Resources for Web Identity Federation

The following resources can help you learn more about web identity federation:

- The post [Web Identity Federation using the AWS SDK for .NET](#) on the AWS Developer blog walks through how to use web identity federation with Facebook. It includes code snippets in C# that show how to assume an IAM role with web identity and how to use temporary security credentials to access an AWS resource.
- The [AWS SDK for iOS](#) and the [AWS SDK for Android](#) contain sample apps. They include code that shows how to invoke the identity providers, and then how to use the information from these providers to get and use temporary security credentials.
- The article [Web Identity Federation with Mobile Applications](#) discusses web identity federation and shows an example of how to use web identity federation to access an AWS resource.

Example Policy for Web Identity Federation

To show how you can use web identity federation with DynamoDB, revisit the `GameScores` table that was introduced in [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 840\)](#). Here is the primary key for `GameScores`.

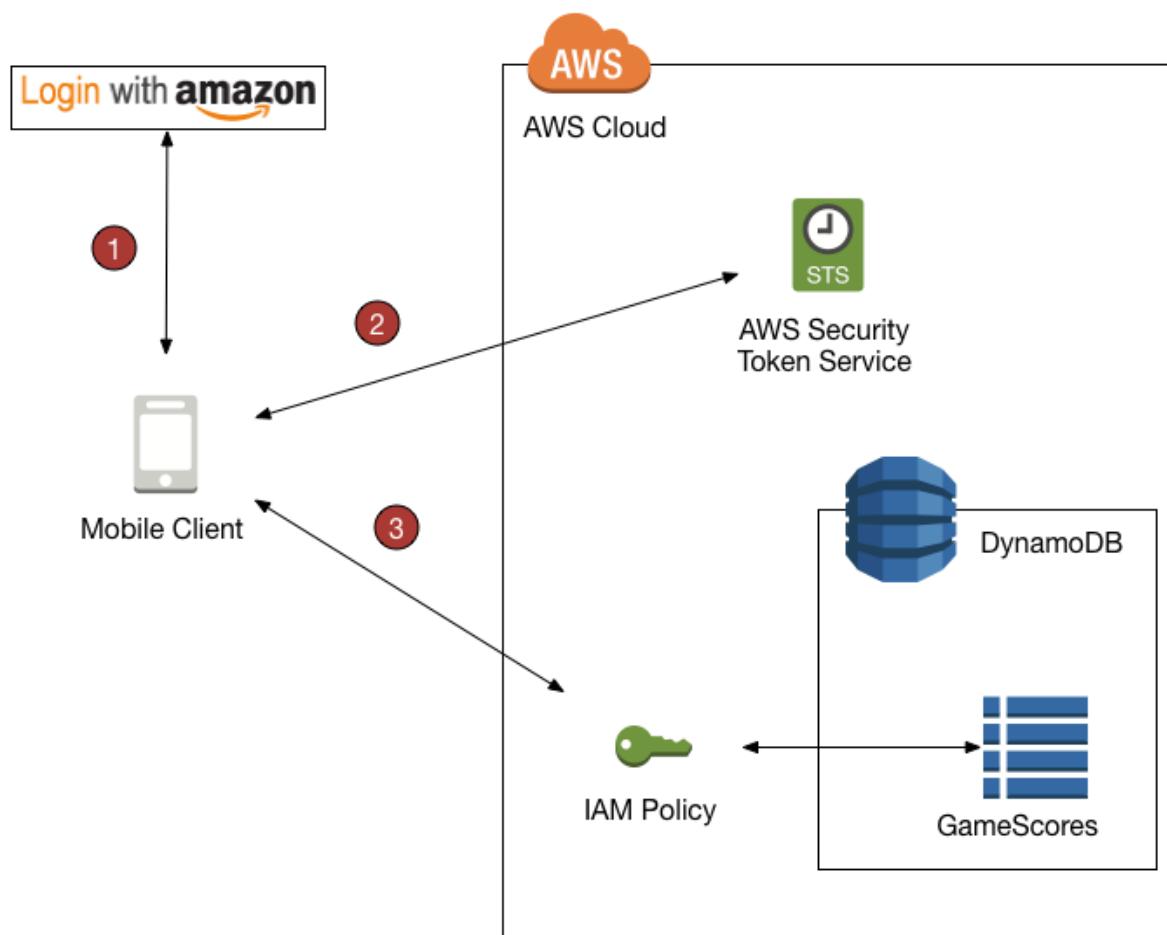
| Table Name | Primary Key Type | Partition Key Name and Type | Sort Key Name and Type |
|---|------------------|---|--|
| GameScores (<code>UserId</code> , <code>GameTitle</code> , ...) | Composite | Attribute Name: <code>UserId</code> Type: String | Attribute Name: <code>GameTitle</code> Type: String |

Now suppose that a mobile gaming app uses this table, and that app needs to support thousands, or even millions, of users. At this scale, it becomes very difficult to manage individual app users, and to guarantee that each user can only access their own data in the *GameScores* table. Fortunately, many users already have accounts with a third-party identity provider, such as Facebook, Google, or Login with Amazon. So it makes sense to use one of these providers for authentication tasks.

To do this using web identity federation, the app developer must register the app with an identity provider (such as Login with Amazon) and obtain a unique app ID. Next, the developer needs to create an IAM role. (For this example, this role is named *GameRole*.) The role must have an IAM policy document attached to it, specifying the conditions under which the app can access *GameScores* table.

When a user wants to play a game, they sign in to their Login with Amazon account from within the gaming app. The app then calls AWS Security Token Service (AWS STS), providing the Login with Amazon app ID and requesting membership in *GameRole*. AWS STS returns temporary AWS credentials to the app and allows it to access the *GameScores* table, subject to the *GameRole* policy document.

The following diagram shows how these pieces fit together.



Web Identity Federation Overview

1. The app calls a third-party identity provider to authenticate the user and the app. The identity provider returns a web identity token to the app.
2. The app calls AWS STS and passes the web identity token as input. AWS STS authorizes the app and gives it temporary AWS access credentials. The app is allowed to assume an IAM role (*GameRole*) and access AWS resources in accordance with the role's security policy.

3. The app calls DynamoDB to access the *GameScores* table. Because it has assumed the *GameRole*, the app is subject to the security policy associated with that role. The policy document prevents the app from accessing data that does not belong to the user.

Once again, here is the security policy for *GameRole* that was shown in [Using IAM Policy Conditions for Fine-Grained Access Control \(p. 840\)](#):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${www.amazon.com:user_id}"
                    ],
                    "dynamodb:Attributes": [
                        "UserId",
                        "GameTitle",
                        "Wins",
                        "Losses",
                        "TopScore",
                        "TopScoreDateTime"
                    ]
                },
                "StringEqualsIfExists": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

The Condition clause determines which items in *GameScores* are visible to the app. It does this by comparing the Login with Amazon ID to the UserId partition key values in *GameScores*. Only the items belonging to the current user can be processed using one of DynamoDB actions that are listed in this policy. Other items in the table cannot be accessed. Furthermore, only the specific attributes listed in the policy can be accessed.

Preparing to Use Web Identity Federation

If you are an application developer and want to use web identity federation for your app, follow these steps:

1. **Sign up as a developer with a third-party identity provider.** The following external links provide information about signing up with supported identity providers:
 - [Login with Amazon Developer Center](#)

- [Registration](#) on the Facebook site
 - [Using OAuth 2.0 to Access Google APIs](#) on the Google site
2. **Register your app with the identity provider.** When you do this, the provider gives you an ID that's unique to your app. If you want your app to work with multiple identity providers, you need to obtain an app ID from each provider.
 3. **Create one or more IAM roles.** You need one role for each identity provider for each app. For example, you might create a role that can be assumed by an app where the user signed in using Login with Amazon, a second role for the same app where the user has signed in using Facebook, and a third role for the app where users sign in using Google.

As part of the role creation process, you need to attach an IAM policy to the role. Your policy document should define the DynamoDB resources required by your app, and the permissions for accessing those resources.

For more information, see [About Web Identity Federation](#) in *IAM User Guide*.

Note

As an alternative to AWS Security Token Service, you can use Amazon Cognito. Amazon Cognito is the preferred service for managing temporary credentials for mobile apps. For more information, see the following pages:

- [How to Authenticate Users \(AWS Mobile SDK for iOS\)](#)
- [How to Authenticate Users \(AWS Mobile SDK for Android\)](#)

Generating an IAM Policy Using the DynamoDB Console

The DynamoDB console can help you create an IAM policy for use with web identity federation. To do this, you choose a DynamoDB table and specify the identity provider, actions, and attributes to be included in the policy. The DynamoDB console then generates a policy that you can attach to an IAM role.

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane, choose **Tables**.
3. In the list of tables, choose the table for which you want to create the IAM policy.
4. Choose the **Access control** tab.
5. Choose the identity provider, actions, and attributes for the policy.

When the settings are as you want them, click **Create policy**. The generated policy appears.

6. Click **Attach policy instructions**, and follow the steps required to attach the generated policy to an IAM role.

Writing Your App to Use Web Identity Federation

To use web identity federation, your app must assume the IAM role that you created. From that point on, the app honors the access policy that you attached to the role.

At runtime, if your app uses web identity federation, it must follow these steps:

1. **Authenticate with a third-party identity provider.** Your app must call the identity provider using an interface that they provide. The exact way in which you authenticate the user depends on the provider and on what platform your app is running. Typically, if the user is not already signed in, the identity provider takes care of displaying a sign-in page for that provider.

After the identity provider authenticates the user, the provider returns a web identity token to your app. The format of this token depends on the provider, but is typically a very long string of characters.

2. **Obtain temporary AWS security credentials.** To do this, your app sends a `AssumeRoleWithWebIdentity` request to AWS Security Token Service (AWS STS). This request contains the following:

- The web identity token from the previous step
- The app ID from the identity provider
- The Amazon Resource Name (ARN) of the IAM role that you created for this identity provider for this app

AWS STS returns a set of AWS security credentials that expire after a certain amount of time (3,600 seconds, by default).

The following is a sample request and response from a `AssumeRoleWithWebIdentity` action in AWS STS. The web identity token was obtained from the Login with Amazon identity provider.

```
GET / HTTP/1.1
Host: sts.amazonaws.com
Content-Type: application/json; charset=utf-8
URL: https://sts.amazonaws.com/?ProviderId=www.amazon.com
&DurationSeconds=900&Action=AssumeRoleWithWebIdentity
&Version=2011-06-15&RoleSessionName=web-identity-federation
&RoleArn=arn:aws:iam::123456789012:role/GameRole
&WebIdentityToken=Atza|IQEBLjAsAhQluyKqyBiYZ8-kclvGTYM81e...(remaining characters
omitted)
```

```
<AssumeRoleWithWebIdentityResponse
  xmlns="https://sts.amazonaws.com/doc/2011-06-15/">
  <AssumeRoleWithWebIdentityResult>
    <SubjectFromWebIdentityToken>amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE</
    SubjectFromWebIdentityToken>
    <Credentials>
      <SessionToken>AQoDYXdzEMf//////////wEa8AP6nNDwcSLnf+cHupC...(remaining characters
omitted)</SessionToken>
      <SecretAccessKey>8Jhi60+EWUubbUShTEsjTxqQtM8UKvsM6XAjdA==</SecretAccessKey>
      <Expiration>2013-10-01T22:14:35Z</Expiration>
      <AccessKeyId>06198791C436IEXAMPLE</AccessKeyId>
    </Credentials>
    <AssumedRoleUser>
      <Arn>arn:aws:sts::123456789012:assumed-role/GameRole/web-identity-federation</Arn>
      <AssumedRoleId>AROAJU4SA2VW5ZRF2YMG:web-identity-federation</AssumedRoleId>
    </AssumedRoleUser>
  </AssumeRoleWithWebIdentityResult>
  <ResponseMetadata>
    <RequestId>c265ac8e-2ae4-11e3-8775-6969323a932d</RequestId>
  </ResponseMetadata>
</AssumeRoleWithWebIdentityResponse>
```

3. **Access AWS resources.** The response from AWS STS contains information that your app requires in order to access DynamoDB resources:

- The `AccessKeyId`, `SecretAccessKey`, and `SessionToken` fields contain security credentials that are valid for this user and this app only.
- The `Expiration` field signifies the time limit for these credentials, after which they are no longer valid.
- The `AssumedRoleId` field contains the name of a session-specific IAM role that has been assumed by the app. The app honors the access controls in the IAM policy document for the duration of this session.

- The `SubjectFromWebIdentityToken` field contains the unique ID that appears in an IAM policy variable for this particular identity provider. The following are the IAM policy variables for supported providers, and some example values for them:

| Policy Variable | Example Value |
|---|--|
| <code>#{www.amazon.com:user_id}</code> | amzn1.account.AGJZDKHJKAUUSW6C44CHPEXAMPLE |
| <code>#{graph.facebook.com:id}</code> | 123456789 |
| <code>#{accounts.google.com:sub}</code> | 123456789012345678901 |

For example IAM policies where these policy variables are used, see [Example Policies: Using Conditions for Fine-Grained Access Control \(p. 844\)](#).

For more information about how AWS STS generates temporary access credentials, see [Requesting Temporary Security Credentials in IAM User Guide](#).

Identity and Access Management in DynamoDB Accelerator

DynamoDB Accelerator (DAX) is designed to work together with DynamoDB, to seamlessly add a caching layer to your applications. However, DAX and DynamoDB have separate access control mechanisms. Both services use AWS Identity and Access Management (IAM) to implement their respective security policies, but the security models for DAX and DynamoDB are different.

For more information about Identity and Access Management in DAX, see [DAX Access Control \(p. 745\)](#).

Logging and Monitoring

Monitoring is an important part of maintaining the reliability, availability, and performance of DynamoDB and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your DynamoDB resources and responding to potential incidents:

Topics

- [Logging and Monitoring in DynamoDB \(p. 855\)](#)
- [Logging and Monitoring in DynamoDB Accelerator \(p. 885\)](#)

Logging and Monitoring in DynamoDB

Monitoring is an important part of maintaining the reliability, availability, and performance of DynamoDB and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. Before you start monitoring DynamoDB, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- What resources will you monitor?
- How often will you monitor these resources?
- What monitoring tools will you use?

- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal DynamoDB performance in your environment, by measuring performance at various times and under different load conditions. As you monitor DynamoDB, you should consider storing historical monitoring data. This stored data will give you a baseline from which to compare current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

To establish a baseline you should, at a minimum, monitor the following items:

- The number of read or write capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used.
- Requests that exceeded a table's provisioned write or read capacity during the specified time period, so you can determine which requests exceed the provisioned throughput limits of a table.
- System errors, so you can determine if any requests resulted in an error.

Topics

- [Monitoring Tools \(p. 856\)](#)
- [Monitoring with Amazon CloudWatch \(p. 857\)](#)
- [Logging DynamoDB Operations by Using AWS CloudTrail \(p. 877\)](#)

Monitoring Tools

AWS provides tools that you can use to monitor DynamoDB. You can configure some of these tools to do the monitoring for you; some require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated Monitoring Tools

You can use the following automated monitoring tools to watch DynamoDB and report when something is wrong:

- **Amazon CloudWatch Alarms** – Watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon EC2 Auto Scaling policy. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods. For more information, see [Monitoring with Amazon CloudWatch \(p. 857\)](#).
- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see [Monitoring Log Files](#) in the *Amazon CloudWatch User Guide*.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see [What is Amazon CloudWatch Events](#) in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see [Working with CloudTrail Log Files](#) in the *AWS CloudTrail User Guide*.

Manual Monitoring Tools

Another important part of monitoring DynamoDB involves manually monitoring those items that the CloudWatch alarms don't cover. The DynamoDB, CloudWatch, Trusted Advisor, and other AWS console

dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on Amazon DynamoDB.

- DynamoDB dashboard shows:
 - Recent alerts
 - Total capacity
 - Service health
- CloudWatch home page shows:
 - Current alarms and status
 - Graphs of alarms and resources
 - Service health status

In addition, you can use CloudWatch to do the following:

- Create [customized dashboards](#) to monitor the services you care about
- Graph metric data to troubleshoot issues and discover trends
- Search and browse all of your AWS resource metrics
- Create and edit alarms to be notified of problems

Monitoring with Amazon CloudWatch

You can monitor Amazon DynamoDB using CloudWatch, which collects and processes raw data from DynamoDB into readable, near real-time metrics. These statistics are retained for a period of time, so that you can access historical information for a better perspective on how your web application or service is performing. By default, DynamoDB metric data is sent to CloudWatch automatically. For more information, see [What Is Amazon CloudWatch?](#) and [Metrics Retention](#) in the *Amazon CloudWatch User Guide*.

Topics

- [DynamoDB Metrics and Dimensions \(p. 857\)](#)
- [How Do I Use DynamoDB Metrics? \(p. 875\)](#)
- [Creating CloudWatch Alarms to Monitor DynamoDB \(p. 876\)](#)

DynamoDB Metrics and Dimensions

When you interact with DynamoDB, it sends the following metrics and dimensions to CloudWatch. You can use the following procedures to view the metrics for DynamoDB.

To view metrics (console)

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Metrics**.
3. Select the **DynamoDB** namespace.

To view metrics (CLI)

- At a command prompt, use the following command:

```
aws cloudwatch list-metrics --namespace "AWS/DynamoDB"
```

CloudWatch displays the following metrics for DynamoDB:

DynamoDB Dimensions and Metrics

The metrics and dimensions that DynamoDB sends to Amazon CloudWatch are listed here.

DynamoDB Metrics

Note

Amazon CloudWatch aggregates the following DynamoDB metrics at one-minute intervals:

- ConditionalCheckFailedRequests
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ReadThrottleEvents
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledRequests
- TransactionConflict
- UserErrors
- WriteThrottleEvents

For all other DynamoDB metrics, the aggregation granularity is five minutes.

Not all statistics, such as *Average* or *Sum*, are applicable for every metric. However, all of these values are available through the Amazon DynamoDB console, or by using the CloudWatch console, AWS CLI, or AWS SDKs for all metrics. In the following table, each metric has a list of valid statistics that are applicable to that metric.

| Metric | Description |
|---------------------------|--|
| AccountMaxReads | <p>The maximum number of read capacity units that can be used by an account. This limit does not apply to on-demand tables or global secondary indexes.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum number of read capacity units that can be used by an account. |
| AccountMaxTableLevelReads | <p>The maximum number of read capacity units that can be used by a table or global secondary index of an account. For on-demand tables this limit caps the maximum read request units a table or a global secondary index can use.</p> <p>Units: Count</p> <p>Valid Statistics:</p> |

| Metric | Description |
|---|---|
| | <ul style="list-style-type: none"> • Maximum – The maximum number of read capacity units that can be used by a table or global secondary index of the account. |
| AccountMaxTableLevelWrites | <p>The maximum number of write capacity units that can be used by a table or global secondary index of an account. For on-demand tables this limit caps the maximum write request units a table or a global secondary index can use.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum number of write capacity units that can be used by a table or global secondary index of the account. |
| AccountMaxWrites | <p>The maximum number of write capacity units that can be used by an account. This limit does not apply to on-demand tables or global secondary indexes.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum number of write capacity units that can be used by an account. |
| AccountProvisionedReadCapacityUtilization | <p>The percentage of provisioned read capacity units utilized by an account.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum percentage of provisioned read capacity units utilized by the account. • Minimum – The minimum percentage of provisioned read capacity units utilized by the account. • Average – The average percentage of provisioned read capacity units utilized by the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average. |

| Metric | Description |
|---------------------------------|---|
| AccountProvisionedWriteCapacity | <p>This metric represents the percentage of provisioned write capacity units utilized by an account.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum percentage of provisioned write capacity units utilized by the account. • Minimum – The minimum percentage of provisioned write capacity units utilized by the account. • Average – The average percentage of provisioned write capacity units utilized by the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average. |
| ConditionalCheckFailedRequests | <p>The number of failed attempts to perform conditional writes. The <code>PutItem</code>, <code>UpdateItem</code>, and <code>DeleteItem</code> operations let you provide a logical condition that must evaluate to true before the operation can proceed. If this condition evaluates to false, <code>ConditionalCheckFailedRequests</code> is incremented by one.</p> <p>Note A failed conditional write will result in an HTTP 400 error (Bad Request). These events are reflected in the <code>ConditionalCheckFailedRequests</code> metric, but not in the <code>UserErrors</code> metric.</p> <p>Units: Count</p> <p>Dimensions: TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|---------------------------|---|
| ConsumedReadCapacityUnits | <p>The number of read capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used. You can retrieve the total consumed read capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see Read/Write Capacity Mode.</p> <p>Note Use the <code>Sum</code> statistic to calculate the consumed throughput. For example, get the <code>Sum</code> value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average <code>ConsumedReadCapacityUnits</code> per second (recognizing that this average does not highlight any large but brief spikes in read activity that occurred during that minute). You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> – The minimum number of read capacity units consumed by any individual request to the table or index. • <code>Maximum</code> – The maximum number of read capacity units consumed by any individual request to the table or index. • <code>Average</code> – The average per-request read capacity consumed. <p>Note The <code>Average</code> value is influenced by periods of inactivity where the sample value will be zero.</p> <ul style="list-style-type: none"> • <code>Sum</code> – The total read capacity units consumed. This is the most useful statistic for the <code>ConsumedReadCapacityUnits</code> metric. • <code>SampleCount</code> – The number of requests to DynamoDB, even if no read capacity was consumed. <p>Note The <code>SampleCount</code> value is influenced by periods of inactivity where the sample value will be zero.</p> |

| Metric | Description |
|----------------------------|---|
| ConsumedWriteCapacityUnits | <p>The number of write capacity units consumed over the specified time period, so you can track how much of your provisioned throughput is used. You can retrieve the total consumed write capacity for a table and all of its global secondary indexes, or for a particular global secondary index. For more information, see Read/Write Capacity Mode.</p> <p>Note Use the <code>Sum</code> statistic to calculate the consumed throughput. For example, get the <code>Sum</code> value over a span of one minute, and divide it by the number of seconds in a minute (60) to calculate the average <code>ConsumedWriteCapacityUnits</code> per second (recognizing that this average does not highlight any large but brief spikes in write activity that occurred during that minute). You can compare the calculated value to the provisioned throughput value that you provide DynamoDB.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • <code>Minimum</code> – The minimum number of write capacity units consumed by any individual request to the table or index. • <code>Maximum</code> – The maximum number of write capacity units consumed by any individual request to the table or index. • <code>Average</code> – The average per-request write capacity consumed. <p>Note The <code>Average</code> value is influenced by periods of inactivity where the sample value will be zero.</p> <ul style="list-style-type: none"> • <code>Sum</code> – The total write capacity units consumed. This is the most useful statistic for the <code>ConsumedWriteCapacityUnits</code> metric. • <code>SampleCount</code> – The number of requests to DynamoDB, even if no write capacity was consumed. <p>Note The <code>SampleCount</code> value is influenced by periods of inactivity where the sample value will be zero.</p> |

| Metric | Description |
|----------------------------------|---|
| MaxProvisionedTableReadCapacity | <p>This percentage of provisioned read capacity units utilized by the highest provisioned read table or global secondary index of an account.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum percentage of provisioned read capacity units utilized by the highest provisioned read table of the account. • Minimum – The minimum percentage of provisioned read capacity units utilized by the highest provisioned read table of the account. • Average – The average percentage of provisioned read capacity units utilized by the highest provisioned read table of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average. |
| MaxProvisionedTableWriteCapacity | <p>The percentage of provisioned write capacity utilized by the highest provisioned write table or global secondary index of an account.</p> <p>Units: Percent</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Maximum – The maximum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account. • Minimum – The minimum percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of an account. • Average – The average percentage of provisioned write capacity units utilized by the highest provisioned write table or global secondary index of the account. The metric is published for five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average. |

| Metric | Description |
|----------------------------------|---|
| OnlineIndexConsumedWriteCapacity | <p>The number of write capacity units consumed when adding a new global secondary index to a table. If the write capacity of the index is too low, incoming write activity during the backfill phase might be throttled. This can increase the time it takes to create the index. You should monitor this statistic while the index is being built to determine whether the write capacity of the index is underprovisioned.</p> <p>You can adjust the write capacity of the index using the <code>UpdateTable</code> operation, even while the index is still being built.</p> <p>The <code>ConsumedWriteCapacityUnits</code> metric for the index does not include the write throughput consumed during index creation.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| OnlineIndexPercentageProgress | <p>The percentage of completion when a new global secondary index is being added to a table. DynamoDB must first allocate resources for the new index, and then backfill attributes from the table into the index. For large tables, this process might take a long time. You should monitor this statistic to view the relative progress as DynamoDB builds the index.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|---------------------------|---|
| OnlineIndexThrottleEvents | <p>The number of write throttle events that occur when adding a new global secondary index to a table. These events indicate that the index creation will take longer to complete, because incoming write activity is exceeding the provisioned write throughput of the index.</p> <p>You can adjust the write capacity of the index using the <code>UpdateTable</code> operation, even while the index is still being built.</p> <p>The <code>WriteThrottleEvents</code> metric for the index does not include any throttle events that occur during index creation.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| PendingReplicationCount | <p>(This metric is for DynamoDB global tables.) The number of item updates that are written to one replica table, but that have not yet been written to another replica in the global table.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>ReceivingRegion</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Average • Sample Count • Sum |

| Metric | Description |
|------------------------------|--|
| ProvisionedReadCapacityUnits | <p>The number of provisioned read capacity units for a table or a global secondary index.</p> <p>The <code>TableName</code> dimension returns the <code>ProvisionedReadCapacityUnits</code> for the table, but not for any global secondary indexes. To view <code>ProvisionedReadCapacityUnits</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum – The lowest setting for provisioned read capacity. If you use <code>UpdateTable</code> to increase read capacity, this metric shows the lowest value of provisioned <code>ReadCapacityUnits</code> during this time period. • Maximum – The highest setting for provisioned read capacity. If you use <code>UpdateTable</code> to decrease read capacity, this metric shows the highest value of provisioned <code>ReadCapacityUnits</code> during this time period. • Average – The average provisioned read capacity. The <code>ProvisionedReadCapacityUnits</code> metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned read capacity units, this statistic might not reflect the true average. |

| Metric | Description |
|-------------------------------|--|
| ProvisionedWriteCapacityUnits | <p>The number of provisioned write capacity units for a table or a global secondary index.</p> <p>The <code>TableName</code> dimension returns the <code>ProvisionedWriteCapacityUnits</code> for the table, but not for any global secondary indexes. To view <code>ProvisionedWriteCapacityUnits</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum – The lowest setting for provisioned write capacity. If you use <code>UpdateTable</code> to increase write capacity, this metric shows the lowest value of provisioned <code>WriteCapacityUnits</code> during this time period. • Maximum – The highest setting for provisioned write capacity. If you use <code>UpdateTable</code> to decrease write capacity, this metric shows the highest value of provisioned <code>WriteCapacityUnits</code> during this time period. • Average – The average provisioned write capacity. The <code>ProvisionedWriteCapacityUnits</code> metric is published at five-minute intervals. Therefore, if you rapidly adjust the provisioned write capacity units, this statistic might not reflect the true average. |
| ReadThrottleEvents | <p>Requests to DynamoDB that exceed the provisioned read capacity units for a table or a global secondary index.</p> <p>A single request can result in multiple events. For example, a <code>BatchGetItem</code> that reads 10 items is processed as 10 <code>GetItem</code> events. For each event, <code>ReadThrottleEvents</code> is incremented by one if that event is throttled. The <code>ThrottledRequests</code> metric for the entire <code>BatchGetItem</code> is not incremented unless <i>all</i> 10 of the <code>GetItem</code> events are throttled.</p> <p>The <code>TableName</code> dimension returns the <code>ReadThrottleEvents</code> for the table, but not for any global secondary indexes. To view <code>ReadThrottleEvents</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • SampleCount • Sum |

| Metric | Description |
|--------------------|---|
| ReplicationLatency | <p>(This metric is for DynamoDB global tables.) The elapsed time between an updated item appearing in the DynamoDB stream for one replica table, and that item appearing in another replica in the global table.</p> <p>Units: Milliseconds</p> <p>Dimensions: TableName, ReceivingRegion</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Average • Minimum • Maximum |
| ReturnedBytes | <p>The number of bytes returned by GetRecords operations (Amazon DynamoDB Streams) during the specified time period.</p> <p>Units: Bytes</p> <p>Dimensions: Operation, StreamLabel, TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| ReturnedItemCount | <p>The number of items returned by Query or Scan operations during the specified time period.</p> <p>The number of items <i>returned</i> is not necessarily the same as the number of items that were evaluated. For example, suppose that you requested a Scan on a table that had 100 items, but specified a FilterExpression that narrowed the results so that only 15 items were returned. In this case, the response from Scan would contain a ScanCount of 100 and a Count of 15 returned items.</p> <p>Units: Count</p> <p>Dimensions: TableName, Operation</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |

| Metric | Description |
|--------------------------|---|
| ReturnedRecordsCount | <p>The number of stream records returned by <code>GetRecords</code> operations (Amazon DynamoDB Streams) during the specified time period.</p> <p>Units: Count</p> <p>Dimensions: <code>Operation</code>, <code>StreamLabel</code>, <code>TableName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount • Sum |
| SuccessfulRequestLatency | <p>The successful requests to DynamoDB or Amazon DynamoDB Streams during the specified time period. <code>SuccessfulRequestLatency</code> can provide two different kinds of information:</p> <ul style="list-style-type: none"> • The elapsed time for successful requests (<code>Minimum</code>, <code>Maximum</code>, <code>Sum</code>, or <code>Average</code>). • The number of successful requests (<code>SampleCount</code>). <p><code>SuccessfulRequestLatency</code> reflects activity only within DynamoDB or Amazon DynamoDB Streams, and does not take into account network latency or client-side activity.</p> <p>Units: Milliseconds</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Minimum • Maximum • Average • SampleCount |
| SystemErrors | <p>The requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 500 status code during the specified time period. An HTTP 500 usually indicates an internal service error.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount |

| Metric | Description |
|----------------------------|---|
| TimeToLiveDeletedItemCount | <p>The number of items deleted by Time to Live (TTL) during the specified time period. This metric helps you monitor the rate of TTL deletions on your table.</p> <p>Units: Count</p> <p>Dimensions: TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none">• Sum |

| Metric | Description |
|-------------------|--|
| ThrottledRequests | <p>Requests to DynamoDB that exceed the provisioned throughput limits on a resource (such as a table or an index).</p> <p>ThrottledRequests is incremented by one if any event within a request exceeds a provisioned throughput limit. For example, if you update an item in a table with global secondary indexes, there are multiple events—a write to the table, and a write to each index. If one or more of these events are throttled, then ThrottledRequests is incremented by one.</p> <p>Note In a batch request (<code>BatchGetItem</code> or <code>BatchWriteItem</code>), ThrottledRequests is incremented only if <i>every</i> request in the batch is throttled. If any individual request within the batch is throttled, one of the following metrics is incremented:</p> <ul style="list-style-type: none"> • <code>ReadThrottleEvents</code> – For a throttled <code>GetItem</code> event within <code>BatchGetItem</code>. • <code>WriteThrottleEvents</code> – For a throttled <code>PutItem</code> or <code>DeleteItem</code> event within <code>BatchWriteItem</code>. <p>To gain insight into which event is throttling a request, compare ThrottledRequests with the <code>ReadThrottleEvents</code> and <code>WriteThrottleEvents</code> for the table and its indexes.</p> <p>Note A throttled request will result in an HTTP 400 status code. All such events are reflected in the ThrottledRequests metric, but not in the UserErrors metric.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>Operation</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount |

| Metric | Description |
|---------------------|--|
| TransactionConflict | <p>Rejected item-level requests due to transactional conflicts between concurrent requests on the same items. For more information, see Transaction Conflict Handling in DynamoDB.</p> <p>Units: Count</p> <p>Dimensions: TableName</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum – The number of rejected item-level requests due to transaction conflicts. <p>Note If multiple item-level requests within a call to <code>TransactWriteItems</code> or <code>TransactGetItems</code> were rejected, <code>Sum</code> is incremented by one for each item-level <code>Put</code>, <code>Update</code>, <code>Delete</code>, or <code>Get</code> request.</p> <ul style="list-style-type: none"> • SampleCount – The number of rejected requests due to transaction conflicts. <p>Note If multiple item-level requests within a call to <code>TransactWriteItems</code> or <code>TransactGetItems</code> are rejected, <code>SampleCount</code> is only incremented by one.</p> <ul style="list-style-type: none"> • Min – The minimum number of rejected item-level requests within a call to <code>TransactWriteItems</code>, <code>TransactGetItems</code>, <code>PutItem</code>, <code>UpdateItem</code>, or <code>DeleteItem</code>. • Max – The maximum number of rejected item-level requests within a call to <code>TransactWriteItems</code>, <code>TransactGetItems</code>, <code>PutItem</code>, <code>UpdateItem</code>, or <code>DeleteItem</code>. • Average – The average number of rejected item-level requests within a call to <code>TransactWriteItems</code>, <code>TransactGetItems</code>, <code>PutItem</code>, <code>UpdateItem</code>, or <code>DeleteItem</code>. |

| Metric | Description |
|------------|---|
| UserErrors | <p>Requests to DynamoDB or Amazon DynamoDB Streams that generate an HTTP 400 status code during the specified time period. An HTTP 400 usually indicates a client-side error, such as an invalid combination of parameters, an attempt to update a nonexistent table, or an incorrect request signature.</p> <p>All such events are reflected in the <code>UserErrors</code> metric, except for the following:</p> <ul style="list-style-type: none"> • <i>ProvisionedThroughputExceededException</i> – See the <code>ThrottledRequests</code> metric in this section. • <i>ConditionalCheckFailedException</i> – See the <code>ConditionalCheckFailedRequests</code> metric in this section. <p><code>UserErrors</code> represents the aggregate of HTTP 400 errors for DynamoDB or Amazon DynamoDB Streams requests for the current AWS Region and the current AWS account.</p> <p>Units: Count</p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount |

| Metric | Description |
|---------------------|---|
| WriteThrottleEvents | <p>Requests to DynamoDB that exceed the provisioned write capacity units for a table or a global secondary index.</p> <p>A single request can result in multiple events. For example, a <code>PutItem</code> request on a table with three global secondary indexes would result in four events—the table write, and each of the three index writes. For each event, the <code>WriteThrottleEvents</code> metric is incremented by one if that event is throttled. For single <code>PutItem</code> requests, if any of the events are throttled, <code>ThrottledRequests</code> is also incremented by one. For <code>BatchWriteItem</code>, the <code>ThrottledRequests</code> metric for the entire <code>BatchWriteItem</code> is not incremented unless all of the individual <code>PutItem</code> or <code>DeleteItem</code> events are throttled.</p> <p>The <code>TableName</code> dimension returns the <code>WriteThrottleEvents</code> for the table, but not for any global secondary indexes. To view <code>WriteThrottleEvents</code> for a global secondary index, you must specify both <code>TableName</code> and <code>GlobalSecondaryIndex</code>.</p> <p>Units: Count</p> <p>Dimensions: <code>TableName</code>, <code>GlobalSecondaryIndexName</code></p> <p>Valid Statistics:</p> <ul style="list-style-type: none"> • Sum • SampleCount |

Dimensions for DynamoDB Metrics

The metrics for DynamoDB are qualified by the values for the account, table name, global secondary index name, or operation. You can use the CloudWatch console to retrieve DynamoDB data along any of the dimensions in the table below.

| Dimension | Description |
|---------------------------------------|---|
| <code>GlobalSecondaryIndexName</code> | This dimension limits the data to a global secondary index on a table. If you specify <code>GlobalSecondaryIndexName</code> , you must also specify <code>TableName</code> . |
| <code>Operation</code> | This dimension limits the data to one of the following DynamoDB operations: <ul style="list-style-type: none"> • <code>PutItem</code> • <code>DeleteItem</code> • <code>UpdateItem</code> • <code>GetItem</code> • <code>BatchGetItem</code> • <code>Scan</code> • <code>Query</code> • <code>BatchWriteItem</code> |

| Dimension | Description |
|------------------------------|---|
| | In addition, you can limit the data to the following Amazon DynamoDB Streams operation: <ul style="list-style-type: none"> • <code>GetRecords</code> |
| <code>ReceivingRegion</code> | This dimension limits the data to a particular AWS region. It is used with metrics originating from replica tables within a DynamoDB global table. |
| <code>StreamLabel</code> | This dimension limits the data to a specific stream label. It is used with metrics originating from Amazon DynamoDB Streams <code>GetRecords</code> operations. |
| <code>TableName</code> | This dimension limits the data to a specific table. This value can be any table name in the current region and the current AWS account. |

How Do I Use DynamoDB Metrics?

The metrics reported by DynamoDB provide information that you can analyze in different ways. The following list shows some common uses for the metrics. These are suggestions to get you started, not a comprehensive list.

| How can I? | Relevant Metrics |
|---|---|
| How can I monitor the rate of TTL deletions on my table? | You can monitor <code>TimeToLiveDeletedItemCount</code> over the specified time period, to track the rate of TTL deletions on your table. For an example of a server-less application using the <code>TimeToLiveDeletedItemCount</code> metric, see Automatically archive items to S3 using DynamoDB Time to Live (TTL) with AWS Lambda and Amazon Kinesis Firehose . |
| How can I determine how much of my provisioned throughput is being used? | You can monitor <code>ConsumedReadCapacityUnits</code> or <code>ConsumedWriteCapacityUnits</code> over the specified time period, to track how much of your provisioned throughput is being used. |
| How can I determine which requests exceed the provisioned throughput limits of a table? | <code>ThrottledRequests</code> is incremented by one if any event within a request exceeds a provisioned throughput limit. Then, to gain insight into which event is throttling a request, compare <code>ThrottledRequests</code> with the <code>ReadThrottleEvents</code> and <code>WriteThrottleEvents</code> metrics for the table and its indexes. |
| How can I determine if any system errors occurred? | You can monitor <code>SystemErrors</code> to determine if any requests resulted in an HTTP 500 (server error) code. Typically, this metric should be equal to zero. If it isn't, then you might want to investigate. <p style="margin-top: 10px;">Note</p> You might encounter internal server errors while working with items. These are expected during the lifetime of a table. Any failed requests can be retried immediately. |

Creating CloudWatch Alarms to Monitor DynamoDB

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period you specify, and performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Auto Scaling policy. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

How can I be notified before I consume my entire read capacity?

1. Create an Amazon SNS topic, `arn:aws:sns:us-east-1:123456789012:capacity-alarm`.

For more information, see [Set Up Amazon Simple Notification Service](#).

2. Create the alarm. In this example, we assume a provisioned capacity of five read capacity units.

```
Prompt>aws cloudwatch put-metric-alarm \
    --alarm-name ReadCapacityUnitsLimitAlarm \
    --alarm-description "Alarm when read capacity reaches 80% of my provisioned read
    capacity" \
    --namespace AWS/DynamoDB \
    --metric-name ConsumedReadCapacityUnits \
    --dimensions Name=TableName,Value=myTable \
    --statistic Sum \
    --threshold 240 \
    --comparison-operator GreaterThanOrEqualToThreshold \
    --period 60 \
    --evaluation-periods 1 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:capacity-alarm
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name ReadCapacityUnitsLimitAlarm --state-
reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name ReadCapacityUnitsLimitAlarm --state-
reason "initializing" --state-value ALARM
```

Note

The alarm is activated whenever the consumed read capacity is at least 4 units per second (80% of provisioned read capacity of 5) for 1 minute (60 seconds). So the threshold is 240 read capacity units (4 units/sec * 60 seconds). Any time the read capacity is updated you should update the alarm calculations appropriately. You can avoid this process by creating alarms through the DynamoDB Console. In this way, the alarms are automatically updated for you.

How can I be notified if any requests exceed the provisioned throughput limits of a table?

1. Create an Amazon SNS topic, `arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput`.

For more information, see [Set Up Amazon Simple Notification Service](#).

2. Create the alarm.

```
Prompt>aws cloudwatch put-metric-alarm \
    --alarm-name RequestsExceedingThroughputAlarm\
    --alarm-description "Alarm when my requests are exceeding provisioned throughput
    limits of a table" \
```

```
--namespace AWS/DynamoDB \
--metric-name ThrottledRequests \
--dimensions Name=TableName,Value=myTable \
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 300 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-east-1:123456789012:requests-exceeding-throughput
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --state-reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name RequestsExceedingThroughputAlarm --state-reason "initializing" --state-value ALARM
```

How can I be notified if any system errors occurred?

1. Create an Amazon SNS topic, arn:aws:sns:us-east-1:123456789012:notify-on-system-errors.

For more information, see [Set Up Amazon Simple Notification Service](#).

2. Create the alarm.

```
Prompt>aws cloudwatch put-metric-alarm \
--alarm-name SystemErrorsAlarm \
--alarm-description "Alarm when system errors occur" \
--namespace AWS/DynamoDB \
--metric-name SystemErrors \
--dimensions Name=TableName,Value=myTable \
--statistic Sum \
--threshold 0 \
--comparison-operator GreaterThanThreshold \
--period 60 \
--unit Count \
--evaluation-periods 1 \
--alarm-actions arn:aws:sns:us-east-1:123456789012:notify-on-system-errors
```

3. Test the alarm.

```
Prompt>aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason "initializing" --state-value OK
```

```
Prompt>aws cloudwatch set-alarm-state --alarm-name SystemErrorsAlarm --state-reason "initializing" --state-value ALARM
```

Logging DynamoDB Operations by Using AWS CloudTrail

DynamoDB is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in DynamoDB. CloudTrail captures all API calls for DynamoDB as events. The calls captured include calls from the DynamoDB console and code calls to the DynamoDB API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for DynamoDB. If you don't configure a trail, you can still view the most recent

events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to DynamoDB, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

DynamoDB Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#).

For an ongoing record of events in your AWS account, including events for DynamoDB, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- [Overview for Creating a Trail](#)
- [CloudTrail Supported Services and Integrations](#)
- [Configuring Amazon SNS Notifications for CloudTrail](#)
- [Receiving CloudTrail Log Files from Multiple Regions](#) and [Receiving CloudTrail Log Files from Multiple Accounts](#)

The following API actions are logged as events in CloudTrail files:

Amazon DynamoDB

- [CreateBackup](#)
- [CreateGlobalTable](#)
- [CreateTable](#)
- [DeleteBackup](#)
- [DeleteTable](#)
- [DescribeBackup](#)
- [DescribeContinuousBackups](#)
- [DescribeGlobalTable](#)
- [DescribeLimits](#)
- [DescribeTable](#)
- [DescribeTimeToLive](#)
- [ListBackups](#)
- [ListTables](#)
- [ListTagsOfResource](#)
- [ListGlobalTables](#)
- [RestoreTableFromBackup](#)
- [RestoreTableToPointInTime](#)
- [TagResource](#)
- [UntagResource](#)
- [UpdateGlobalTable](#)

- [UpdateTable](#)
- [UpdateTimeToLive](#)
- [DescribeReservedCapacity](#)
- [DescribeReservedCapacityOfferings](#)
- [DescribeScalableTargets](#)
- [RegisterScalableTarget](#)
- [PurchaseReservedCapacityOfferings](#)

DynamoDB Streams

- [DescribeStream](#)
- [ListStreams](#)

DynamoDB Accelerator (DAX)

- [CreateCluster](#)
- [CreateParameterGroup](#)
- [CreateSubnetGroup](#)
- [DecreaseReplicationFactor](#)
- [DeleteCluster](#)
- [DeleteParameterGroup](#)
- [DeleteSubnetGroup](#)
- [DescribeClusters](#)
- [DescribeDefaultParameters](#)
- [DescribeEvents](#)
- [DescribeParameterGroups](#)
- [DescribeParameters](#)
- [DescribeSubnetGroups](#)
- [IncreaseReplicationFactor](#)
- [ListTags](#)
- [RebootNode](#)
- [TagResource](#)
- [UntagResource](#)
- [UpdateCluster](#)
- [UpdateParameterGroup](#)
- [UpdateSubnetGroup](#)

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the [CloudTrail userIdentity Element](#).

Understanding DynamoDB Log File Entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log that demonstrates the `CreateTable`, `DescribeTable`, `UpdateTable`, `ListTables`, `DeleteTable`, and `CreateCluster` actions.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2015-05-28T18:06:01Z"
          },
          "sessionIssuer": {
            "type": "Role",
            "principalId": "AKIAI44QH8DHBEXAMPLE",
            "arn": "arn:aws:iam::444455556666:role/admin-role",
            "accountId": "444455556666",
            "userName": "bob"
          }
        }
      },
      "eventTime": "2015-05-01T07:24:55Z",
      "eventSource": "dynamodb.amazonaws.com",
      "eventName": "CreateTable",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "console.aws.amazon.com",
      "requestParameters": {
        "provisionedThroughput": {
          "writeCapacityUnits": 10,
          "readCapacityUnits": 10
        },
        "tableName": "Music",
        "keySchema": [
          {
            "attributeName": "Artist",
            "keyType": "HASH"
          },
          {
            "attributeName": "SongTitle",
            "keyType": "RANGE"
          }
        ],
        "attributeDefinitions": [
          {
            "attributeType": "S",
            "attributeName": "Artist"
          },
          {
            "attributeType": "S",
            "attributeName": "SongTitle"
          }
        ]
      }
    }
  ]
}
```

```

        }
    ],
},
"responseElements": {"tableDescription": {
    "tableName": "Music",
    "attributeDefinitions": [
        {
            "attributeType": "S",
            "attributeName": "Artist"
        },
        {
            "attributeType": "S",
            "attributeName": "SongTitle"
        }
    ],
    "itemCount": 0,
    "provisionedThroughput": {
        "writeCapacityUnits": 10,
        "numberOfDecreasesToday": 0,
        "readCapacityUnits": 10
    },
    "creationDateTime": "May 1, 2015 7:24:55 AM",
    "keySchema": [
        {
            "attributeName": "Artist",
            "keyType": "HASH"
        },
        {
            "attributeName": "SongTitle",
            "keyType": "RANGE"
        }
    ],
    "tableStatus": "CREATING",
    "tableSizeBytes": 0
}},
"requestID": "KAVGJR1Q0I5VHF8FS8V809EV7FVV4KONSO5AEMVJF66Q9ASUAAJG",
"eventID": "a8b5f864-480b-43bf-bc22-9b6d77910a29",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
},
{
    "eventVersion": "1.03",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "444455556666",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2015-05-28T18:06:01Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AKIAI44QH8DHBEXAMPLE",
                "arn": "arn:aws:iam::444455556666:role/admin-role",
                "accountId": "444455556666",
                "userName": "bob"
            }
        }
    },
    "eventTime": "2015-05-04T02:43:11Z",
    "eventSource": "dynamodb.amazonaws.com",

```

```

        "eventName": "DescribeTable",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "console.aws.amazon.com",
        "requestParameters": {"tableName": "Music"},
        "responseElements": null,
        "requestID": "DISTSH6DQRLCC74L48Q51LRBFVV4KQNSO5AEMVJF66Q9ASUAAJG",
        "eventID": "c07befa7-f402-4770-8c1b-1911601ed2af",
        "eventType": "AwsApiCall",
        "apiVersion": "2012-08-10",
        "recipientAccountId": "111122223333"
    },
    {
        "eventVersion": "1.03",
        "userIdentity": {
            "type": "AssumedRole",
            "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
            "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
            "accountId": "111122223333",
            "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
            "sessionContext": {
                "attributes": {
                    "mfaAuthenticated": "false",
                    "creationDate": "2015-05-28T18:06:01Z"
                },
                "sessionIssuer": {
                    "type": "Role",
                    "principalId": "AKIAI44QH8DHBEXAMPLE",
                    "arn": "arn:aws:iam::444455556666:role/admin-role",
                    "accountId": "444455556666",
                    "userName": "bob"
                }
            }
        },
        "eventTime": "2015-05-04T02:14:52Z",
        "eventSource": "dynamodb.amazonaws.com",
        "eventName": "UpdateTable",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "console.aws.amazon.com",
        "requestParameters": {"provisionedThroughput": {
            "writeCapacityUnits": 25,
            "readCapacityUnits": 25
        }},
        "responseElements": {"tableDescription": {
            "tableName": "Music",
            "attributeDefinitions": [
                {
                    "attributeType": "S",
                    "attributeName": "Artist"
                },
                {
                    "attributeType": "S",
                    "attributeName": "SongTitle"
                }
            ],
            "itemCount": 0,
            "provisionedThroughput": {
                "writeCapacityUnits": 10,
                "numberOfDecreasesToday": 0,
                "readCapacityUnits": 10,
                "lastIncreaseDateTime": "May 3, 2015 11:34:14 PM"
            },
            "creationDateTime": "May 3, 2015 11:34:14 PM",
            "keySchema": [

```

```

        {
            "attributeName": "Artist",
            "keyType": "HASH"
        },
        {
            "attributeName": "SongTitle",
            "keyType": "RANGE"
        }
    ],
    "tableStatus": "UPDATING",
    "tableSizeBytes": 0
},
"requestID": "AALNP0J2L244N5O15PKISJ1KUFVV4KQNSO5AEMVJF66Q9ASUAAJG",
"eventID": "eb834e01-f168-435f-92c0-c36278378b6e",
"eventType": "AwsApiCall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
},
{
    "eventVersion": "1.03",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2015-05-28T18:06:01Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AKIAI44QH8DHBEEXAMPLE",
                "arn": "arn:aws:iam::444455556666:role/admin-role",
                "accountId": "444455556666",
                "userName": "bob"
            }
        }
    },
    "eventTime": "2015-05-04T02:42:20Z",
    "eventSource": "dynamodb.amazonaws.com",
    "eventName": "ListTables",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "console.aws.amazon.com",
    "requestParameters": null,
    "responseElements": null,
    "requestID": "3BGHST5OVHLMTPUMAUTA1RF4M3VV4KQNSO5AEMVJF66Q9ASUAAJG",
    "eventID": "bd5bf4b0-b8a5-4bec-9edf-83605bd5e54e",
    "eventType": "AwsApiCall",
    "apiVersion": "2012-08-10",
    "recipientAccountId": "111122223333"
},
{
    "eventVersion": "1.03",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAIOSFODNN7EXAMPLE:bob",
        "arn": "arn:aws:sts::111122223333:assumed-role/users/bob",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {

```

```

        "mfaAuthenticated": "false",
        "creationDate": "2015-05-28T18:06:01Z"
    },
    "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEEXAMPLE",
        "arn": "arn:aws:iam::444455556666:role/admin-role",
        "accountId": "444455556666",
        "userName": "bob"
    }
}
},
"eventTime": "2015-05-04T13:38:20Z",
"eventSource": "dynamodb.amazonaws.com",
"eventName": "DeleteTable",
"awsRegion": "us-west-2",
"sourceIPAddress": "192.0.2.0",
"userAgent": "console.aws.amazon.com",
"requestParameters": {"tableName": "Music"},
"responseElements": {"tableDescription": {
    "tableName": "Music",
    "itemCount": 0,
    "provisionedThroughput": {
        "writeCapacityUnits": 25,
        "numberOfDecreasesToday": 0,
        "readCapacityUnits": 25
    },
    "tableStatus": "DELETING",
    "tableSizeBytes": 0
}},
"requestID": "4KBNVRGD25RG1KEO9UT4V3FQDJVV4KQNSO5AEMVJF66Q9ASUAAJG",
"eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
"eventType": "AwsApicall",
"apiVersion": "2012-08-10",
"recipientAccountId": "111122223333"
},
{
"eventVersion": "1.05",
"userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDAIAVK7DT3VEXAMPLES",
    "arn": "arn:aws:iam::111122223333:user/bob",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "bob"
},
"eventTime": "2019-12-17T23:17:34Z",
"eventSource": "dax.amazonaws.com",
"eventName": "CreateCluster",
"awsRegion": "us-west-2",
"sourceIPAddress": "205.251.233.48",
"userAgent": "aws-cli/1.16.304 Python/3.6.9
Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.13.40",
"requestParameters": {
    "SSESpecification": {
        "enabled": true
    },
    "clusterName": "daxcluster",
    "nodeType": "dax.r4.large",
    "replicationFactor": 3,
    "iamRoleArn": "arn:aws:iam::111122223333:role/DAXServiceRoleForDynamoDBAccess"
},
"responseElements": {
    "cluster": {
        "securityGroups": [

```

```
{  
    "securityGroupIdentifier": "sg-1af6e36e",  
    "status": "active"  
},  
    "parameterGroup": {  
        "nodeIdsToReboot": [],  
        "parameterGroupName": "default.dax1.0",  
        "parameterApplyStatus": "in-sync"  
},  
    "clusterDiscoveryEndpoint": {  
        "port": 8111  
},  
    "clusterArn": "arn:aws:dax:us-west-2:111122223333:cache/daxcluster",  
    "status": "creating",  
    "subnetGroup": "default",  
    "sSEDescription": {  
        "status": "ENABLED",  
        "kMSMasterKeyArn": "arn:aws:kms:us-west-2:111122223333:key/764898e4-  
adb1-46d6-a762-e2f4225b4fc4"  
    },  
    "iamRoleArn": "arn:aws:iam::111122223333:role/DAXServiceRoleForDynamoDBAccess",  
    "clusterName": "daxcluster",  
    "activeNodes": 0,  
    "totalNodes": 3,  
    "preferredMaintenanceWindow": "thu:13:00-thu:14:00",  
    "nodeType": "dax.r4.large"  
}},  
    "requestID": "585adc5f-ad05-4e27-8804-70ba1315f8fd",  
    "eventID": "29158945-28da-4e32-88e1-56d1b90c1a0c",  
    "eventType": "AwsApiCall",  
    "recipientAccountId": "111122223333"  
}  
]  
}
```

Logging and Monitoring in DynamoDB Accelerator

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon DynamoDB Accelerator (DAX) and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs.

For more information about logging and monitoring in DAX, see [Monitoring DAX \(p. 732\)](#).

Compliance Validation by Industry for DynamoDB

The security and compliance of DynamoDB is assessed by third-party auditors as part of multiple AWS compliance programs, including the following:

- System and Organization Controls (SOC)
- Payment Card Industry (PCI)
- Federal Risk and Authorization Management Program (FedRAMP)
- Health Insurance Portability and Accountability Act (HIPAA)

AWS provides a frequently updated list of AWS services in scope of specific compliance programs at [AWS Services in Scope by Compliance Program](#).

Third-party audit reports are available for you to download using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

For more information about AWS compliance programs, see [AWS Compliance Programs](#).

Your compliance responsibility when using DynamoDB is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of DynamoDB is subject to compliance with standards like HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- [Security and Compliance Quick Start Guides](#) – Deployment guides that discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- [Architecting for HIPAA Security and Compliance Whitepaper](#) – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.
- [AWS Compliance Resources](#) – A collection of workbooks and guides that might apply to your industry and location.
- [AWS Config](#) – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience and Disaster Recovery in Amazon DynamoDB

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

If you need to replicate your data or applications over greater geographic distances, use AWS Local Regions. An AWS Local Region is a single data center designed to complement an existing AWS Region. Like all AWS Regions, AWS Local Regions are completely isolated from other AWS Regions.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

In addition to the AWS global infrastructure, Amazon DynamoDB offers several features to help support your data resiliency and backup needs.

On-demand backup and restore

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival. For more information, see [On-Demand Backup and Restore for DynamoDB](#).

Point-in-time recovery

Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For more information, see [Point-in-Time Recovery for DynamoDB](#).

Infrastructure Security in Amazon DynamoDB

As a managed service, Amazon DynamoDB is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

You use AWS published API calls to access DynamoDB through the network. Clients must support TLS (Transport Layer Security) 1.0. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Diffie-Hellman Ephemeral (ECDHE). Most modern systems such as Java 7 and later support these modes. Additionally, requests must be signed using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

You can also use a virtual private cloud (VPC) endpoint for DynamoDB to enable Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. For more information, see [Using Amazon VPC Endpoints to Access DynamoDB \(p. 887\)](#).

Using Amazon VPC Endpoints to Access DynamoDB

For security reasons, many AWS customers run their applications within an Amazon Virtual Private Cloud environment (Amazon VPC). With Amazon VPC, you can launch Amazon EC2 instances into a virtual private cloud, which is logically isolated from other networks—including the public internet. With an Amazon VPC, you have control over its IP address range, subnets, routing tables, network gateways, and security settings.

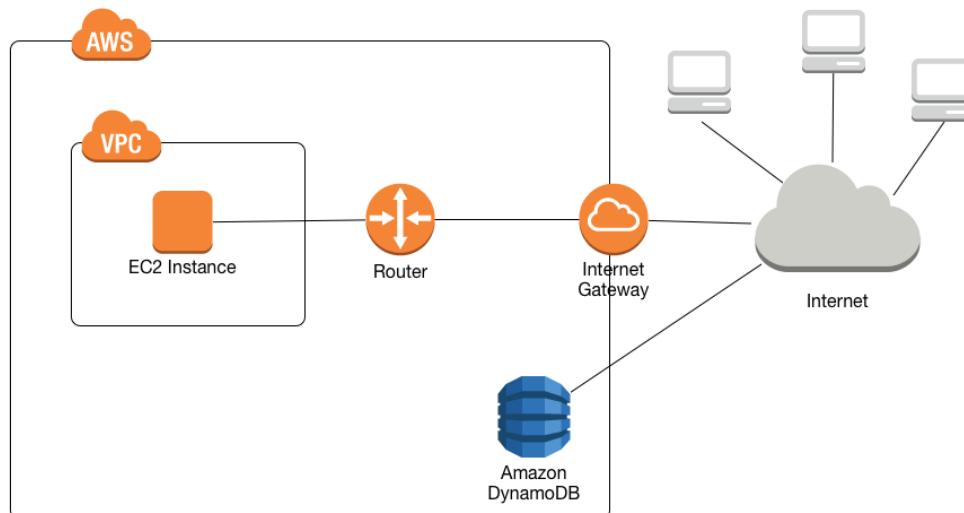
Note

If you created your AWS account after December 4, 2013, then you already have a default VPC in each AWS Region. A default VPC is ready for you to use—you can immediately start using it without having to perform any additional configuration steps.

For more information, see [Default VPC and Default Subnets](#) in the *Amazon VPC User Guide*.

To access the public internet, your VPC must have an internet gateway—a virtual router that connects your VPC to the internet. This allows applications running on Amazon EC2 in your VPC to access internet resources, such as Amazon DynamoDB.

By default, communications to and from DynamoDB use the HTTPS protocol, which protects network traffic by using SSL/TLS encryption. The following diagram shows how an EC2 instance in a VPC accesses DynamoDB:



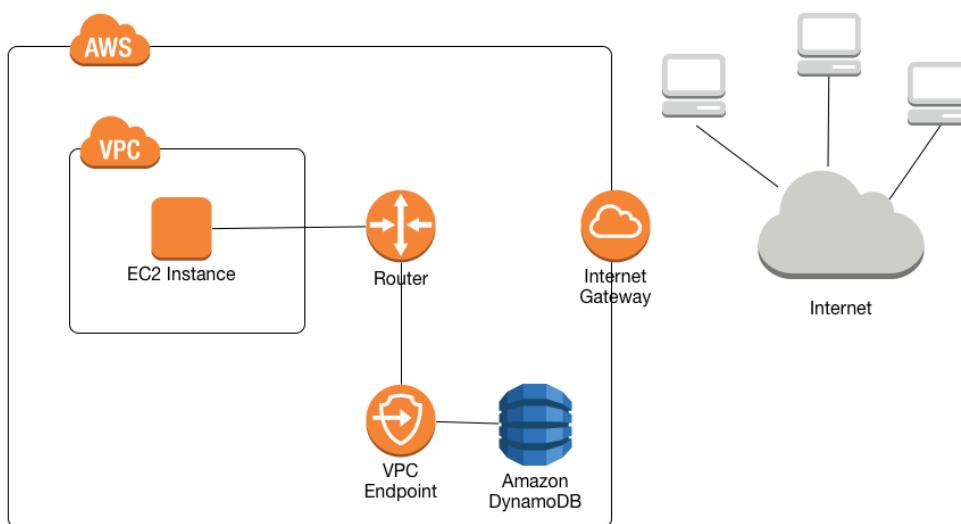
Many customers have legitimate privacy and security concerns about sending and receiving data across the public internet. You can address these concerns by using a virtual private network (VPN) to route all

DynamoDB network traffic through your own corporate network infrastructure. However, this approach can introduce bandwidth and availability challenges.

VPC endpoints for DynamoDB can alleviate these challenges. A *VPC endpoint* for DynamoDB enables Amazon EC2 instances in your VPC to use their private IP addresses to access DynamoDB with no exposure to the public internet. Your EC2 instances do not require public IP addresses, and you don't need an internet gateway, a NAT device, or a virtual private gateway in your VPC. You use endpoint policies to control access to DynamoDB. Traffic between your VPC and the AWS service does not leave the Amazon network.

When you create a VPC endpoint for DynamoDB, any requests to a DynamoDB endpoint within the Region (for example, `dynamodb.us-west-2.amazonaws.com`) are routed to a private DynamoDB endpoint within the Amazon network. You don't need to modify your applications running on EC2 instances in your VPC. The endpoint name remains the same, but the route to DynamoDB stays entirely within the Amazon network, and does not access the public internet.

The following diagram shows how an EC2 instance in a VPC can use a VPC endpoint to access DynamoDB.



For more information, see [the section called “Tutorial: Using a VPC Endpoint for DynamoDB” \(p. 888\)](#).

Tutorial: Using a VPC Endpoint for DynamoDB

This section walks you through setting up and using a VPC endpoint for DynamoDB.

Topics

- [Step 1: Launch an Amazon EC2 Instance \(p. 888\)](#)
- [Step 2: Configure Your Amazon EC2 Instance \(p. 890\)](#)
- [Step 3: Create a VPC Endpoint for DynamoDB \(p. 891\)](#)
- [Step 4: \(Optional\) Clean Up \(p. 892\)](#)

Step 1: Launch an Amazon EC2 Instance

In this step, you launch an Amazon EC2 instance in your default Amazon VPC. You can then create and use a VPC endpoint for DynamoDB.

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose **Launch Instance** and do the following:
 - Step 1: Choose an Amazon Machine Image (AMI)
 - At the top of the list of AMIs, go to **Amazon Linux AMI** and choose **Select**.
 - Step 2: Choose an Instance Type
 - At the top of the list of instance types, choose **t2.micro**.
 - Choose **Next: Configure Instance Details**.
 - Step 3: Configure Instance Details
 - Go to **Network** and choose your default VPC.
 - Choose **Next: Add Storage**.
 - Step 4: Add Storage
 - Skip this step by choosing **Next: Tag Instance**.
 - Step 5: Tag Instance
 - Skip this step by choosing **Next: Configure Security Group**.
 - Step 6: Configure Security Group
 - Choose **Select an existing security group**.
 - In the list of security groups, choose **default**. This is the default security group for your VPC.
 - Choose **Next: Review and Launch**.
 - Step 7: Review Instance Launch
 - Choose **Launch**.
3. In the **Select an existing key pair or create a new key pair** window, do one of the following:
 - If you do not have an Amazon EC2 key pair, choose **Create a new key pair** and follow the instructions. You will be asked to download a private key file (.pem file); you will need this file later when you log in to your Amazon EC2 instance.
 - If you already have an existing Amazon EC2 key pair, go to **Select a key pair** and choose your key pair from the list. You must already have the private key file (.pem file) available in order to log in to your Amazon EC2 instance.
4. When you have configured your key pair, choose **Launch Instances**.
5. Return to the Amazon EC2 console home page and choose the instance that you launched. In the lower pane, on the **Description** tab, find the **Public DNS** for your instance. For example: ec2-00-00-00-00.us-east-1.compute.amazonaws.com.

Make a note of this public DNS name, because you will need it in the next step in this tutorial ([Step 2: Configure Your Amazon EC2 Instance \(p. 890\)](#)).

Note

It will take a few minutes for your Amazon EC2 instance to become available. Before you go on to the next step, ensure that the **Instance State** is running and that all of its **Status Checks** have passed.

Step 2: Configure Your Amazon EC2 Instance

When your Amazon EC2 instance is available, you will be able to log into it and prepare it for first use.

Note

The following steps assume that you are connecting to your Amazon EC2 instance from a computer running Linux. For other ways to connect, see [Connect to Your Linux Instance](#) in the Amazon EC2 User Guide for Linux Instances.

1. You will need to authorize inbound SSH traffic to your Amazon EC2 instance. To do this, you will create a new EC2 security group, and then assign the security group to your EC2 instance.
 - a. In the navigation pane, choose **Security Groups**.
 - b. Choose **Create Security Group**. In the **Create Security Group** window, do the following:
 - **Security group name**—type a name for your security group. For example: my-ssh-access
 - **Description**—type a short description for the security group.
 - **VPC**—choose your default VPC.
 - In the **Security group rules** section, choose **Add Rule** and do the following:
 - **Type**—choose SSH.
 - **Source**—choose My IP.

When the settings are as you want them, choose **Create**.

- c. In the navigation pane, choose **Instances**.
 - d. Choose the Amazon EC2 instance that you launched in [Step 1: Launch an Amazon EC2 Instance \(p. 888\)](#).
 - e. Choose **Actions --> Networking --> Change Security Groups**.
 - f. In the **Change Security Groups**, select the security group that you created earlier in this procedure (for example: my-ssh-access). The existing default security group should also be selected. When the settings are as you want them, choose **Assign Security Groups**.
2. Use the ssh command to log in to your Amazon EC2 instance, as in the following example.

```
ssh -i my-keypair.pem ec2-user@public-dns-name
```

You will need to specify your private key file (.pem file) and the public DNS name of your instance. (See [Step 1: Launch an Amazon EC2 Instance \(p. 888\)](#)).

The login ID is ec2-user. No password is required.

3. Configure your AWS credentials, as shown following. Enter your AWS access key ID, secret key, and default Region name when prompted.

```
aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-east-1
Default output format [None]:
```

You are now ready to create a VPC endpoint for DynamoDB.

Step 3: Create a VPC Endpoint for DynamoDB

In this step, you will create a VPC endpoint for DynamoDB and test it to make sure that it works.

1. Before you begin, verify that you can communicate with DynamoDB using its public endpoint.

```
aws dynamodb list-tables
```

The output will show a list of DynamoDB tables that you currently own. (If you don't have any tables, the list will be empty.).

2. Verify that DynamoDB is an available service for creating VPC endpoints in the current AWS Region. (The command is shown in bold text, followed by example output.)

```
aws ec2 describe-vpc-endpoint-services

{
    "ServiceNames": [
        "com.amazonaws.us-east-1.s3",
        "com.amazonaws.us-east-1.dynamodb"
    ]
}
```

In the example output, DynamoDB is one of the services available, so you can proceed with creating a VPC endpoint for it.

3. Determine your VPC identifier.

```
aws ec2 describe-vpcs

{
    "Vpcs": [
        {
            "VpcId": "vpc-0bbc736e",
            "InstanceTenancy": "default",
            "State": "available",
            "DhcpOptionsId": "dopt-8454b7e1",
            "CidrBlock": "172.31.0.0/16",
            "IsDefault": true
        }
    ]
}
```

In the example output, the VPC ID is vpc-0bbc736e.

4. Create the VPC endpoint. For the `--vpc-id` parameter, specify the VPC ID from the previous step. Use the `--route-table-ids` parameter to associate the endpoint with your route tables.

```
aws ec2 create-vpc-endpoint --vpc-id vpc-0bbc736e --service-name com.amazonaws.us-east-1.dynamodb --route-table-ids rtb-11aa22bb

{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\", \"Statement\":[{\"Effect\": \"Allow\", \"Principal\":\"*\", \"Action\":\"*\", \"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
        "RouteTableIds": [
            "rtb-11aa22bb"
        ]
    }
}
```

```
        ],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

5. Verify that you can access DynamoDB through the VPC endpoint.

```
aws dynamodb list-tables
```

If you want, you can try some other AWS CLI commands for DynamoDB. For more information, see the [AWS CLI Command Reference](#).

Step 4: (Optional) Clean Up

If you want to delete the resources you have created in this tutorial, follow these procedures:

To remove your VPC endpoint for DynamoDB

1. Log in to your Amazon EC2 instance.
2. Determine the VPC endpoint ID.

```
aws ec2 describe-vpc-endpoints

{
    "VpcEndpoint": {
        "PolicyDocument": "{\"Version\":\"2008-10-17\", \"Statement\":[{\"Effect\":
\"Allow\", \"Principal\":\"*\", \"Action\":\"*\", \"Resource\":\"*\"}]}",
        "VpcId": "vpc-0bbc736e",
        "State": "available",
        "ServiceName": "com.amazonaws.us-east-1.dynamodb",
        "RouteTableIds": [],
        "VpcEndpointId": "vpce-9b15e2f2",
        "CreationTimestamp": "2017-07-26T22:00:14Z"
    }
}
```

In the example output, the VPC endpoint ID is vpce-9b15e2f2.

3. Delete the VPC endpoint.

```
aws ec2 delete-vpc-endpoints --vpc-endpoint-ids vpce-9b15e2f2

{
    "Unsuccessful": []
}
```

The empty array [] indicates success (there were no unsuccessful requests).

To terminate your Amazon EC2 instance

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the navigation pane, choose **Instances**.
3. Choose your Amazon EC2 instance.
4. Choose **Actions, Instance State, Terminate**.
5. In the confirmation window, choose **Yes, Terminate**.

Configuration and Vulnerability Analysis in Amazon DynamoDB

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- [Compliance Validation for Amazon DynamoDB](#)
- [Shared Responsibility Model](#)
- [Amazon Web Services: Overview of Security Processes](#)(whitepaper)

The following security best practices also address configuration and vulnerability analysis in Amazon DynamoDB:

- [Monitor DynamoDB compliance with AWS Config Rules](#)
- [Monitor DynamoDB configuration with AWS Config](#)

Security Best Practices for Amazon DynamoDB

Amazon DynamoDB provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Topics

- [DynamoDB Preventative Security Best Practices \(p. 893\)](#)
- [DynamoDB Detective Security Best Practices \(p. 895\)](#)

DynamoDB Preventative Security Best Practices

The following best practices can help you anticipate and prevent security incidents in Amazon DynamoDB.

Encryption at rest

DynamoDB encrypts at rest all user data stored in tables, indexes, streams, and backups using encryption keys stored in [AWS Key Management Service \(AWS KMS\)](#). This provides an additional layer of data protection by securing your data from unauthorized access to the underlying storage .

You can specify whether DynamoDB should use an AWS owned CMK (default encryption type) or an AWS managed CMK to encrypt the user data. For more information, see [Amazon DynamoDB Encryption at Rest](#).

Use IAM roles to authenticate access to DynamoDB

For users, applications, and other AWS services to access DynamoDB, they must include valid AWS credentials in their AWS API requests. You should not store AWS credentials directly in the application or EC2 instance. These are long-term credentials that are not automatically rotated, and therefore could have significant business impact if they are compromised. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources.

For more information, see [Authentication](#).

Use IAM policies for DynamoDB base authorization

When granting permissions, you decide who is getting them, which DynamoDB APIs they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM identities (that is, users, groups, and roles) and thereby grant permissions to perform operations on DynamoDB resources.

You can do this by using the following:

- [AWS Managed \(predefined\) policies](#)
- [Customer managed policies](#)

Use IAM policy conditions for fine-grained access control

When you grant permissions in DynamoDB, you can specify conditions that determine how a permissions policy takes effect. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

You can specify conditions when granting permissions using an IAM policy. For example, you can do the following:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based upon the identity of that user.

For more information, see [Using IAM Policy Conditions for Fine-Grained Access Control](#).

Use a VPC endpoint and policies to access DynamoDB

If you only require access to DynamoDB from within a virtual private cloud (VPC), you should use a VPC endpoint to limit access from only the required VPC. Doing this prevents that traffic from traversing the open internet and being subject to that environment.

Using a VPC endpoint for DynamoDB allows you to control and limit access using the following:

- VPC endpoint policies – These policies are applied on the DynamoDB VPC endpoint. They allow you to control and limit API access to the DynamoDB table.
- IAM policies – By using the `aws:sourceVpc` condition on policies attached to IAM users, groups, or roles, you can enforce that all access to the DynamoDB table is via the specified VPC endpoint.

For more information, see [Endpoints for Amazon DynamoDB](#).

Consider client-side encryption

If you store sensitive or confidential data in DynamoDB, you might want to encrypt that data as close as possible to its origin so that your data is protected throughout its lifecycle. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party.

The [Amazon DynamoDB Encryption Client](#) is a software library that helps you protect your table data before you send it to DynamoDB.

At the core of the DynamoDB Encryption Client is an item encryptor that encrypts, signs, verifies, and decrypts table items. It takes in information about your table items and instructions about which items to encrypt and sign. It gets the encryption materials and instructions on how to use them from a cryptographic material provider that you select and configure.

DynamoDB Detective Security Best Practices

The following best practices for Amazon DynamoDB can help you detect potential security weaknesses and incidents.

Use AWS CloudTrail to monitor AWS managed KMS key usage

If you are using an [AWS managed customer master key \(CMK\)](#) for encryption at rest, usage of this key is logged into AWS CloudTrail. CloudTrail provides visibility into user activity by recording actions taken on your account. CloudTrail records important information about each action, including who made the request, the services used, the actions performed, parameters for the actions, and the response elements returned by the AWS service. This information helps you track changes made to your AWS resources and troubleshoot operational issues. CloudTrail makes it easier to ensure compliance with internal policies and regulatory standards.

You can use CloudTrail to audit key usage. CloudTrail creates log files that contain a history of AWS API calls and related events for your account. These log files include all AWS KMS API requests made using the AWS Management Console, AWS SDKs, and command line tools, in addition to those made through integrated AWS services. You can use these log files to get information about when the CMK was used, the operation that was requested, the identity of the requester, the IP address that the request came from, and so on. For more information, see [Logging AWS KMS API Calls with AWS CloudTrail](#) and the [AWS CloudTrail User Guide](#).

Use CloudTrail to monitor DynamoDB control-plane operations

CloudTrail provides visibility into user activity by recording actions taken on your account. CloudTrail records important information about each action, including who made the request, the services used, the actions performed, parameters for the actions, and the response elements returned by the AWS service. This information helps you to track changes made to your AWS resources and to troubleshoot operational issues. CloudTrail makes it easier to ensure compliance with internal policies and regulatory standards.

Control-plane operations let you create and manage DynamoDB tables. They also let you work with indexes, streams, and other objects that are dependent on tables.

When activity occurs in DynamoDB, that activity is recorded in a CloudTrail event along with other AWS service events in the event history. For more information, see [Logging DynamoDB Operations by Using AWS CloudTrail](#). You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#) in the [AWS CloudTrail User Guide](#).

For an ongoing record of events in your AWS account, including events for DynamoDB, create a [trail](#). A trail enables CloudTrail to deliver log files to an Amazon Simple Storage Service (Amazon S3) bucket. By default, when you create a trail on the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

Consider using DynamoDB Streams to monitor modify/update data-plane operations

AWS CloudTrail does not support logging of DynamoDB data-plane operations, such as `GetItem` and `PutItem`. So you might want to consider using Amazon DynamoDB Streams as a source for these events occurring in your environment.

DynamoDB is integrated with AWS Lambda so that you can create triggers—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon Resource Name (ARN) with a Lambda function that you write. Immediately after an item in the table is modified, a

new record appears in the table's stream. AWS Lambda polls the stream and invokes your Lambda function synchronously when it detects new stream records. The Lambda function can perform any actions that you specify, such as sending a notification or initiating a workflow.

For an example, see [Tutorial: Using AWS Lambda with Amazon DynamoDB Streams](#). This example receives a DynamoDB event input, processes the messages that it contains, and writes some of the incoming event data to Amazon CloudWatch Logs.

Monitor DynamoDB configuration with AWS Config

Using [AWS Config](#), you can continuously monitor and record configuration changes of your AWS resources. AWS Config also enables you to inventory your AWS resources. When a change from a previous state is detected, an Amazon Simple Notification Service (Amazon SNS) notification can be delivered for you to review and take action. Follow the guidance in [Setting Up AWS Config with the Console](#), ensuring that DynamoDB resource types are included.

You can configure AWS Config to stream configuration changes and notifications to an Amazon SNS topic. For example, when a resource is updated, you can get a notification sent to your email, so that you can view the changes. You can also be notified when AWS Config evaluates your custom or managed rules against your resources.

For an example, see [Notifications that AWS Config Sends to an Amazon SNS topic](#) in the *AWS Config Developer Guide*.

Monitor DynamoDB compliance with AWS Config rules

AWS Config continuously tracks the configuration changes that occur among your resources. It checks whether these changes violate any of the conditions in your rules. If a resource violates a rule, AWS Config flags the resource and the rule as noncompliant.

By using AWS Config to evaluate your resource configurations, you can assess how well your resource configurations comply with internal practices, industry guidelines, and regulations. AWS Config provides [AWS managed rules](#), which are predefined, customizable rules that AWS Config uses to evaluate whether your AWS resources comply with common best practices.

Tag your DynamoDB resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they enable you to categorize resources by purpose, owner, environment, or other criteria. The following are some examples:

- Security – Used to determine requirements such as encryption.
- Confidentiality – An identifier for the specific data-confidentiality level a resource supports.
- Environment – Used to distinguish between development, test, and production infrastructure.

For more information, see [AWS Tagging Strategies](#) and [Tagging for DynamoDB](#).

Best Practices for Designing and Architecting with DynamoDB

Use this section to quickly find recommendations for maximizing performance and minimizing throughput costs when working with Amazon DynamoDB.

Contents

- [NoSQL Design for DynamoDB \(p. 898\)](#)
 - Differences Between Relational Data Design and NoSQL (p. 898)
 - Two Key Concepts for NoSQL Design (p. 899)
 - Approaching NoSQL Design (p. 899)
- [Best Practices for Designing and Using Partition Keys Effectively \(p. 900\)](#)
 - Using Burst Capacity Effectively (p. 900)
 - Understanding DynamoDB Adaptive Capacity (p. 900)
 - Boost Throughput Capacity to High-Traffic Partitions (p. 901)
 - Isolate Frequently Accessed Items (p. 901)
 - Designing Partition Keys to Distribute Your Workload Evenly (p. 902)
 - Using Write Sharding to Distribute Workloads Evenly (p. 903)
 - Sharding Using Random Suffixes (p. 903)
 - Sharding Using Calculated Suffixes (p. 903)
 - Distributing Write Activity Efficiently During Data Upload (p. 904)
- [Best Practices for Using Sort Keys to Organize Data \(p. 905\)](#)
 - Using Sort Keys for Version Control (p. 905)
- [Best Practices for Using Secondary Indexes in DynamoDB \(p. 906\)](#)
 - General Guidelines for Secondary Indexes in DynamoDB (p. 907)
 - Use Indexes Efficiently (p. 907)
 - Choose Projections Carefully (p. 907)
 - Optimize Frequent Queries to Avoid Fetches (p. 908)
 - Be Aware of Item-Collection Size Limits When Creating Local Secondary Indexes (p. 908)
 - Take Advantage of Sparse Indexes (p. 908)
 - Examples of Sparse Indexes in DynamoDB (p. 909)
 - Using Global Secondary Indexes for Materialized Aggregation Queries (p. 910)
 - Overloading Global Secondary Indexes (p. 910)
 - Using Global Secondary Index Write Sharding for Selective Table Queries (p. 911)
 - Using Global Secondary Indexes to Create an Eventually Consistent Replica (p. 912)
- [Best Practices for Storing Large Items and Attributes \(p. 913\)](#)
 - Compressing Large Attribute Values (p. 913)
 - Storing Large Attribute Values in Amazon S3 (p. 913)
- [Best Practices for Handling Time Series Data in DynamoDB \(p. 914\)](#)
 - Design Pattern for Time Series Data (p. 914)

- [Time Series Table Examples \(p. 914\)](#)
- [Best Practices for Managing Many-to-Many Relationships \(p. 915\)](#)
 - [Adjacency List Design Pattern \(p. 915\)](#)
 - [Materialized Graph Pattern \(p. 916\)](#)
- [Best Practices for Implementing a Hybrid Database System \(p. 919\)](#)
 - [If You Don't Want to Migrate Everything to DynamoDB \(p. 919\)](#)
 - [How a Hybrid System Can Be Implemented \(p. 920\)](#)
- [Best Practices for Modeling Relational Data in DynamoDB \(p. 920\)](#)
 - [First Steps for Modeling Relational Data in DynamoDB \(p. 922\)](#)
 - [Example of Modeling Relational Data in DynamoDB \(p. 923\)](#)
- [Best Practices for Querying and Scanning Data \(p. 925\)](#)
 - [Performance Considerations for Scans \(p. 925\)](#)
 - [Avoiding Sudden Spikes in Read Activity \(p. 926\)](#)
 - [Taking Advantage of Parallel Scans \(p. 928\)](#)
 - [Choosing TotalSegments \(p. 928\)](#)
- [Best Practices for Using Global Tables \(p. 929\)](#)

NoSQL Design for DynamoDB

NoSQL database systems like Amazon DynamoDB use alternative models for data management, such as key-value pairs or document storage. When you switch from a relational database management system to a NoSQL database system like DynamoDB, it's important to understand the key differences and specific design approaches.

Topics

- [Differences Between Relational Data Design and NoSQL \(p. 898\)](#)
- [Two Key Concepts for NoSQL Design \(p. 899\)](#)
- [Approaching NoSQL Design \(p. 899\)](#)

Differences Between Relational Data Design and NoSQL

Relational database systems (RDBMS) and NoSQL databases have different strengths and weaknesses:

- In RDBMS, data can be queried flexibly, but queries are relatively expensive and don't scale well in high-traffic situations (see [First Steps for Modeling Relational Data in DynamoDB \(p. 922\)](#)).
- In a NoSQL database such as DynamoDB, data can be queried efficiently in a limited number of ways, outside of which queries can be expensive and slow.

These differences make database design different between the two systems:

- In RDBMS, you design for flexibility without worrying about implementation details or performance. Query optimization generally doesn't affect schema design, but normalization is important.
- In DynamoDB, you design your schema specifically to make the most common and important queries as fast and as inexpensive as possible. Your data structures are tailored to the specific requirements of your business use cases.

Two Key Concepts for NoSQL Design

NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can go ahead and create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. You can organize each type of data into its own table.

How NoSQL design is different

- By contrast, you shouldn't start designing your schema for DynamoDB until you know the questions it will need to answer. Understanding the business problems and the application use cases up front is essential.
- You should maintain as few tables as possible in a DynamoDB application.

Approaching NoSQL Design

The first step in designing your DynamoDB application is to identify the specific query patterns that the system must satisfy.

In particular, it is important to understand three fundamental properties of your application's access patterns before you begin:

- **Data size:** Knowing how much data will be stored and requested at one time will help determine the most effective way to partition the data.
- **Data shape:** Instead of reshaping data when a query is processed (as an RDBMS system does), a NoSQL database organizes data so that its shape in the database corresponds with what will be queried. This is a key factor in increasing speed and scalability.
- **Data velocity:** DynamoDB scales by increasing the number of physical partitions that are available to process queries, and by efficiently distributing data across those partitions. Knowing in advance what the peak query loads might help determine how to partition data to best use I/O capacity.

After you identify specific query requirements, you can organize data according to general principles that govern performance:

- **Keep related data together.** Research on routing-table optimization 20 years ago found that "locality of reference" was the single most important factor in speeding up response time: keeping related data together in one place. This is equally true in NoSQL systems today, where keeping related data in close proximity has a major impact on cost and performance. Instead of distributing related data items across multiple tables, you should keep related items in your NoSQL system as close together as possible.

As a general rule, you should maintain as few tables as possible in a DynamoDB application.

Exceptions are cases where high-volume time series data are involved, or datasets that have very different access patterns. A single table with inverted indexes can usually enable simple queries to create and retrieve the complex hierarchical data structures required by your application.

- **Use sort order.** Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.
- **Distribute queries.** It is also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding "hot spots."
- **Use global secondary indexes.** By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

These general principles translate into some common design patterns that you can use to model data efficiently in DynamoDB.

Best Practices for Designing and Using Partition Keys Effectively

The primary key that uniquely identifies each item in an Amazon DynamoDB table can be simple (a partition key only) or composite (a partition key combined with a sort key).

Generally speaking, you should design your application for uniform activity across all logical partition keys in the table and its secondary indexes. You can determine the access patterns that your application requires, and estimate the total read capacity units (RCU) and write capacity units (WCU) that each table and secondary index requires.

DynamoDB supports your access patterns using the throughput that you provisioned as long as the traffic against a given partition does not exceed 3,000 RCU or 1,000 WCUs.

Topics

- [Using Burst Capacity Effectively \(p. 900\)](#)
- [Understanding DynamoDB Adaptive Capacity \(p. 900\)](#)
- [Designing Partition Keys to Distribute Your Workload Evenly \(p. 902\)](#)
- [Using Write Sharding to Distribute Workloads Evenly \(p. 903\)](#)
- [Distributing Write Activity Efficiently During Data Upload \(p. 904\)](#)

Using Burst Capacity Effectively

DynamoDB provides some flexibility in your per-partition throughput provisioning by providing *burst capacity*. Whenever you're not fully using a partition's throughput, DynamoDB reserves a portion of that unused capacity for later *bursts* of throughput to handle usage spikes.

DynamoDB currently retains up to 5 minutes (300 seconds) of unused read and write capacity. During an occasional burst of read or write activity, these extra capacity units can be consumed quickly—even faster than the per-second provisioned throughput capacity that you've defined for your table.

DynamoDB can also consume burst capacity for background maintenance and other tasks without prior notice.

Note that these burst capacity details might change in the future.

Understanding DynamoDB Adaptive Capacity

Adaptive capacity is a feature that enables DynamoDB to run imbalanced workloads indefinitely. It minimizes throttling due to throughput exceptions. It also helps you reduce costs by enabling you to provision only the throughput capacity that you need.

Adaptive capacity is enabled automatically for every DynamoDB table, at no additional cost. You don't need to explicitly enable or disable it.

Topics

- [Boost Throughput Capacity to High-Traffic Partitions \(p. 901\)](#)
- [Isolate Frequently Accessed Items \(p. 901\)](#)

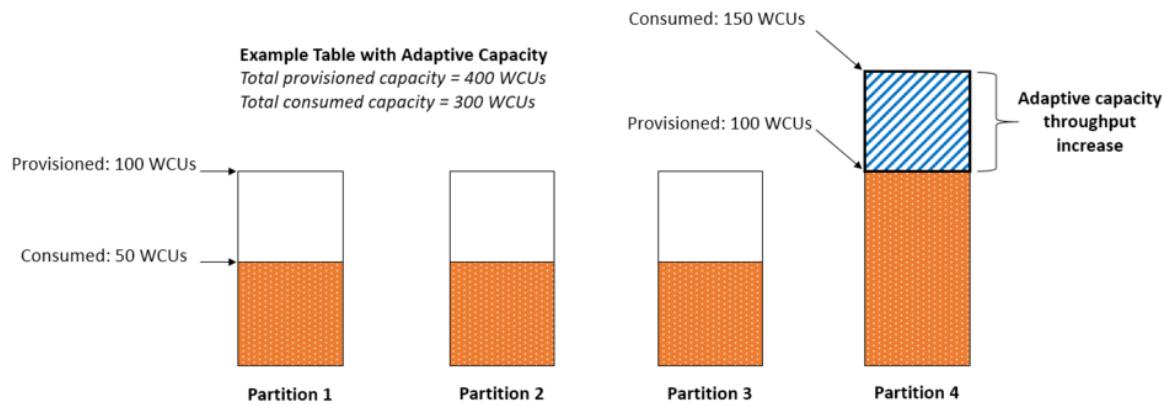
Boost Throughput Capacity to High-Traffic Partitions

It's not always possible to distribute read and write activity evenly. When data access is imbalanced, a "hot" partition can receive a higher volume of read and write traffic compared to other partitions. In extreme cases, throttling can occur if a single partition receives more than 3,000 RCU or 1,000 WCUs.

To better accommodate uneven access patterns, DynamoDB adaptive capacity enables your application to continue reading and writing to hot partitions without being throttled, provided that traffic does not exceed your table's total provisioned capacity or the partition maximum capacity. Adaptive capacity works by automatically and instantly increasing throughput capacity for partitions that receive more traffic.

The following diagram illustrates how adaptive capacity works. The example table is provisioned with 400 WCUs evenly shared across four partitions, allowing each partition to sustain up to 100 WCUs per second. Partitions 1, 2, and 3 each receives write traffic of 50 WCU/sec. Partition 4 receives 150 WCU/sec. This hot partition can accept write traffic while it still has unused burst capacity, but eventually it throttles traffic that exceeds 100 WCU/sec.

DynamoDB adaptive capacity responds by increasing partition 4's capacity so that it can sustain the higher workload of 150 WCU/sec without being throttled.



Isolate Frequently Accessed Items

If your application drives disproportionately high traffic to one or more items, adaptive capacity rebalances your partitions such that frequently accessed items don't reside on the same partition. This isolation of frequently accessed items reduces the likelihood of request throttling due to your workload exceeding the throughput limit on a single partition.

If your application drives consistently high traffic to a single item, adaptive capacity might rebalance your data such that a partition contains only that single, frequently accessed item. In this case, DynamoDB can deliver throughput up to the partition maximum of 3,000 RCU or 1,000 WCUs to that single item's primary key.

Note

This isolation functionality is not available for

- Tables that have a [local secondary index](#) .
- Tables using [Provisioned Read/Write Capacity Mode](#) that have enabled [DynamoDB Streams](#).

Designing Partition Keys to Distribute Your Workload Evenly

The partition key portion of a table's primary key determines the logical partitions in which a table's data is stored. This in turn affects the underlying physical partitions. Provisioned I/O capacity for the table is divided evenly among these physical partitions. Therefore, a partition key design that doesn't distribute I/O requests evenly can create "hot" partitions that result in throttling and use your provisioned I/O capacity inefficiently.

The optimal usage of a table's provisioned throughput depends not only on the workload patterns of individual items, but also on the partition key design. This doesn't mean that you must access all partition key values to achieve an efficient throughput level, or even that the percentage of accessed partition key values must be high. It does mean that the more distinct partition key values that your workload accesses, the more those requests will be spread across the partitioned space. In general, you will use your provisioned throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values increases.

The following is a comparison of the provisioned throughput efficiency of some common partition key schemas.

| Partition key value | Uniformity |
|--|------------|
| User ID, where the application has many users. | Good |
| Status code, where there are only a few possible status codes. | Bad |
| Item creation date, rounded to the nearest time period (for example, day, hour, or minute). | Bad |
| Device ID, where each device accesses data at relatively similar intervals. | Good |
| Device ID, where even if there are many devices being tracked, one is by far more popular than all the others. | Bad |

If a single table has only a small number of partition key values, consider distributing your write operations across more distinct partition key values. In other words, structure the primary key elements to avoid one "hot" (heavily requested) partition key value that slows overall performance.

For example, consider a table with a composite primary key. The partition key represents the item's creation date, rounded to the nearest day. The sort key is an item identifier. On a given day, say 2014-07-09, **all** of the new items are written to that single partition key value (and corresponding physical partition).

If the table fits entirely into a single partition (considering growth of your data over time), and if your application's read and write throughput requirements don't exceed the read and write capabilities of a single partition, your application won't encounter any unexpected throttling as a result of partitioning.

However, if you anticipate your table scaling beyond a single partition, you should architect your application so that it can use more of the table's full provisioned throughput.

Using Write Sharding to Distribute Workloads Evenly

One way to better distribute writes across a partition key space in Amazon DynamoDB is to expand the space. You can do this in several different ways. You can add a random number to the partition key values to distribute the items among partitions. Or you can use a number that is calculated based on something that you're querying on.

Sharding Using Random Suffixes

One strategy for distributing loads more evenly across a partition key space is to add a random number to the end of the partition key values. Then you randomize the writes across the larger space.

For example, for a partition key that represents today's date, you might choose a random number between 1 and 200 and concatenate it as a suffix to the date. This yields partition key values like 2014-07-09.1, 2014-07-09.2, and so on, through 2014-07-09.200. Because you are randomizing the partition key, the writes to the table on each day are spread evenly across multiple partitions. This results in better parallelism and higher overall throughput.

However, to read all the items for a given day, you would have to query the items for all the suffixes and then merge the results. For example, you would first issue a `Query` request for the partition key value 2014-07-09.1. Then issue another `Query` for 2014-07-09.2, and so on, through 2014-07-09.200. Finally, your application would have to merge the results from all those `Query` requests.

Sharding Using Calculated Suffixes

A randomizing strategy can greatly improve write throughput. But it's difficult to read a specific item because you don't know which suffix value was used when writing the item. To make it easier to read individual items, you can use a different strategy. Instead of using a random number to distribute the items among partitions, use a number that you can calculate based upon something that you want to query on.

Consider the previous example, in which a table uses today's date in the partition key. Now suppose that each item has an accessible `OrderId` attribute, and that you most often need to find items by order ID in addition to date. Before your application writes the item to the table, it could calculate a hash suffix based on the order ID and append it to the partition key date. The calculation might generate a number between 1 and 200 that is fairly evenly distributed, similar to what the random strategy produces.

A simple calculation would likely suffice, such as the product of the UTF-8 code point values for the characters in the order ID, modulo 200, + 1. The partition key value would then be the date concatenated with the calculation result.

With this strategy, the writes are spread evenly across the partition key values, and thus across the physical partitions. You can easily perform a `GetItem` operation for a particular item and date because you can calculate the partition key value for a specific `OrderId` value.

To read all the items for a given day, you still must `query` each of the 2014-07-09.N keys (where N is 1–200), and your application then has to merge all the results. The benefit is that you avoid having a single "hot" partition key value taking all of the workload.

Note

For a more efficient strategy specifically designed to handle high-volume time series data, see [Time Series Data \(p. 914\)](#).

Distributing Write Activity Efficiently During Data Upload

Typically, when you load data from other data sources, Amazon DynamoDB partitions your table data on multiple servers. You get better performance if you upload data to all the allocated servers simultaneously.

For example, suppose that you want to upload user messages to a DynamoDB table that uses a composite primary key with `UserID` as the partition key and `MessageID` as the sort key.

When you upload the data, one approach you can take is to upload all message items for each user, one user after another:

| UserID | MessageID |
|--------|---------------|
| U1 | 1 |
| U1 | 2 |
| U1 | ... |
| U1 | ... up to 100 |
| U2 | 1 |
| U2 | 2 |
| U2 | ... |
| U2 | ... up to 200 |

The problem in this case is that you are not distributing your write requests to DynamoDB across your partition key values. You are taking one partition key value at a time and uploading all of its items before going to the next partition key value and doing the same.

Behind the scenes, DynamoDB is partitioning the data in your table across multiple servers. To fully use all the throughput capacity that is provisioned for the table, you must distribute your workload across your partition key values. By directing an uneven amount of upload work toward items that all have the same partition key value, you are not fully using all the resources that DynamoDB has provisioned for your table.

You can distribute your upload work by using the sort key to load one item from each partition key value, then another item from each partition key value, and so on:

| UserID | MessageID |
|--------|-----------|
| U1 | 1 |
| U2 | 1 |
| U3 | 1 |
| ... | ... |
| U1 | 2 |

| UserID | MessageID |
|--------|-----------|
| U2 | 2 |
| U3 | 2 |
| ... | ... |

Every upload in this sequence uses a different partition key value, keeping more DynamoDB servers busy simultaneously and improving your throughput performance.

Best Practices for Using Sort Keys to Organize Data

In an Amazon DynamoDB table, the primary key that uniquely identifies each item in the table can be composed not only of a partition key, but also of a sort key.

Well-designed sort keys have two key benefits:

- They gather related information together in one place where it can be queried efficiently. Careful design of the sort key lets you retrieve commonly needed groups of related items using range queries with operators such as `begins_with`, `between`, `>`, `<`, and so on.
- Composite sort keys let you define hierarchical (one-to-many) relationships in your data that you can query at any level of the hierarchy.

For example, in a table listing geographical locations, you might structure the sort key as follows.

```
[country]#[region]#[state]#[county]#[city]#[neighborhood]
```

This would let you make efficient range queries for a list of locations at any one of these levels of aggregation, from country, to a neighborhood, and everything in between.

Using Sort Keys for Version Control

Many applications need to maintain a history of item-level revisions for audit or compliance purposes and to be able to retrieve the most recent version easily. There is an effective design pattern that accomplishes this using sort key prefixes:

- For each new item, create two copies of the item: One copy should have a version-number prefix of zero (such as `v0_`) at the beginning of the sort key, and one should have a version-number prefix of one (such as `v1_`).
- Every time the item is updated, use the next higher version-prefix in the sort key of the updated version, and copy the updated contents into the item with version-prefix zero. This means that the latest version of any item can be located easily using the zero prefix.

For example, a parts manufacturer might use a schema like the one illustrated below.

| Primary Key | | Data-Item Attributes... | | | | | | | |
|---------------|----------------------|-------------------------|---|-------------|---|-------------|--|-------------|--|
| Partition Key | Sort Key (varies) | Attribute 1 | | Attribute 2 | | Attribute 3 | | Attribute 4 | ... |
| Equipment_1 | Details | Name: | Biphasic Cardiometer <i>(equipment name)</i> | Factory_ID: | S14_Tukwilla <i>(factory where manufactured)</i> | Line_ID | R_7 <i>(assembly-line ID)</i> | | |
| | | Auditor: | Padma <i>(name of the auditor)</i> | Latest: | 3 <i>(most recent audit version)</i> | Time: | 2018-04-15T11:00 <i>(audit date and time)</i> | Result | Passed <i>(audit result)</i> |
| | v0_Audit | Auditor: | Rick <i>(name of the auditor)</i> | Time: | 2018-03-14T11:00 <i>(audit date and time)</i> | Result | Open <i>(audit result)</i> | Report: | 0943922EKG14 <i>(detailed problem report in S3)</i> |
| | v1_Audit | Auditor: | George <i>(name of the auditor)</i> | Time: | 2018-03-18T11:00 <i>(audit date and time)</i> | Result | Open <i>(audit result)</i> | Report: | 0943923EKG15 <i>(detailed problem report in S3)</i> |
| | v3_Audit | Auditor: | Padma <i>(name of the auditor)</i> | Time: | 2018-04-15T11:00 <i>(audit date and time)</i> | Result | Passed <i>(audit result)</i> | Report: | x792 <i>(pass confirmation report)</i> |

The Equipment_1 item goes through a sequence of audits by various auditors. The results of each new audit are captured in a new item in the table, starting with version number one, and then incrementing the number for each successive revision.

When each new revision is added, the application layer replaces the contents of the zero-version item (having sort key equal to v0_Audit) with the contents of the new revision.

Whenever the application needs to retrieve for the most recent audit status, it can query for the sort key prefix of v0_.

If the application needs to retrieve the entire revision history, it can query all the items under the item's partition key and filter out the v0_ item.

This design also works for audits across multiple parts of a piece of equipment, if you include the individual part-IDs in the sort key after the sort key prefix.

Best Practices for Using Secondary Indexes in DynamoDB

Secondary indexes are often essential to support the query patterns that your application requires. At the same time, overusing secondary indexes or using them inefficiently can add cost and reduce performance unnecessarily.

Contents

- [General Guidelines for Secondary Indexes in DynamoDB \(p. 907\)](#)
 - [Use Indexes Efficiently \(p. 907\)](#)
 - [Choose Projections Carefully \(p. 907\)](#)
 - [Optimize Frequent Queries to Avoid Fetches \(p. 908\)](#)
 - [Be Aware of Item-Collection Size Limits When Creating Local Secondary Indexes \(p. 908\)](#)
- [Take Advantage of Sparse Indexes \(p. 908\)](#)
 - [Examples of Sparse Indexes in DynamoDB \(p. 909\)](#)
- [Using Global Secondary Indexes for Materialized Aggregation Queries \(p. 910\)](#)
- [Overloading Global Secondary Indexes \(p. 910\)](#)
- [Using Global Secondary Index Write Sharding for Selective Table Queries \(p. 911\)](#)
- [Using Global Secondary Indexes to Create an Eventually Consistent Replica \(p. 912\)](#)

General Guidelines for Secondary Indexes in DynamoDB

Amazon DynamoDB supports two types of secondary indexes:

- **Global secondary index**—An index with a partition key and a sort key that can be different from those on the base table. A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions. A global secondary index has no size limitations and has its own provisioned throughput settings for read and write activity that are separate from those of the table.
- **Local secondary index**—An index that has the same partition key as the base table, but a different sort key. A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value. As a result, the total size of indexed items for any one partition key value can't exceed 10 GB. Also, a local secondary index shares provisioned throughput settings for read and write activity with the table it is indexing.

Each table in DynamoDB is limited to 20 global secondary indexes (default limit) and 5 local secondary indexes.

For more information about the differences between global secondary indexes and local secondary indexes, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

In general, you should use global secondary indexes rather than local secondary indexes. The exception is when you need strong consistency in your query results, which a local secondary index can provide but a global secondary index cannot (global secondary index queries only support eventual consistency).

The following are some general principles and design patterns to keep in mind when creating indexes in DynamoDB:

Topics

- [Use Indexes Efficiently \(p. 907\)](#)
- [Choose Projections Carefully \(p. 907\)](#)
- [Optimize Frequent Queries to Avoid Fetches \(p. 908\)](#)
- [Be Aware of Item-Collection Size Limits When Creating Local Secondary Indexes \(p. 908\)](#)

Use Indexes Efficiently

Keep the number of indexes to a minimum. Don't create secondary indexes on attributes that you don't query often. Indexes that are seldom used contribute to increased storage and I/O costs without improving application performance.

Choose Projections Carefully

Because secondary indexes consume storage and provisioned throughput, you should keep the size of the index as small as possible. Also, the smaller the index, the greater the performance advantage compared to querying the full table. If your queries usually return only a small subset of attributes, and the total size of those attributes is much smaller than the whole item, project only the attributes that you regularly request.

If you expect a lot of write activity on a table compared to reads, follow these best practices:

- Consider projecting fewer attributes to minimize the size of items written to the index. However, this only applies if the size of projected attributes would otherwise be larger than a single write capacity unit (1 KB). For example, if the size of an index entry is only 200 bytes, DynamoDB rounds this up to

1 KB. In other words, as long as the index items are small, you can project more attributes at no extra cost.

- Avoid projecting attributes that you know will rarely be needed in queries. Every time you update an attribute that is projected in an index, you incur the extra cost of updating the index as well. You can still retrieve non-projected attributes in a *Query* at a higher provisioned throughput cost, but the query cost may be significantly lower than the cost of updating the index frequently.
- Specify `ALL` only if you want your queries to return the entire table item sorted by a different sort key. Projecting all attributes eliminates the need for table fetches, but in most cases, it doubles your costs for storage and write activity.

Balance the need to keep your indexes as small as possible against the need to keep fetches to a minimum, as explained in the next section.

Optimize Frequent Queries to Avoid Fetches

To get the fastest queries with the lowest possible latency, project all the attributes that you expect those queries to return. In particular, if you query a local secondary index for attributes that are not projected, DynamoDB automatically fetches those attributes from the table, which requires reading the entire item from the table. This introduces latency and additional I/O operations that you can avoid.

Keep in mind that "occasional" queries can often turn into "essential" queries. If there are attributes that you don't intend to project because you anticipate querying them only occasionally, consider whether circumstances might change and you might regret not projecting those attributes after all.

For more information about table fetches, see [Provisioned Throughput Considerations for Local Secondary Indexes \(p. 542\)](#).

Be Aware of Item-Collection Size Limits When Creating Local Secondary Indexes

An *item collection* is all the items in a table and its local secondary indexes that have the same partition key. No item collection can exceed 10 GB, so it's possible to run out of space for a particular partition key value.

When you add or update a table item, DynamoDB updates all local secondary indexes that are affected. If the indexed attributes are defined in the table, the local secondary indexes grow too.

When you create a local secondary index, think about how much data will be written to it, and how many of those data items will have the same partition key value. If you expect that the sum of table and index items for a particular partition key value might exceed 10 GB, consider whether you should avoid creating the index.

If you can't avoid creating the local secondary index, you must anticipate the item collection size limit and take action before you exceed it. For strategies on working within the limit and taking corrective action, see [Item Collection Size Limit \(p. 546\)](#).

Take Advantage of Sparse Indexes

For any item in a table, DynamoDB writes a corresponding index entry **only if the index sort key value is present in the item**. If the sort key doesn't appear in every table item, the index is said to be *sparse*.

Sparse indexes are useful for queries over a small subsection of a table. For example, suppose that you have a table where you store all your customer orders, with the following key attributes:

- Partition key: `CustomerId`
- Sort key: `OrderId`

To track open orders, you can insert a Boolean attribute named `isOpen` in order items that have not already shipped. Then when the order ships, you can delete the attribute. If you then create an index on `CustomerID` (partition key) and `isOpen` (sort key), only those orders with `isOpen` defined appear in it. When you have thousands of orders of which only a small number are open, it's faster and less expensive to query that index for open orders than to scan the entire table.

Instead of using a Boolean type of attribute like `isOpen`, you could use an attribute with a value that results in a useful sort order in the index. For example, you could use an `OrderOpenDate` attribute set to the date on which each order was placed, and then delete it after the order is fulfilled. That way, when you query the sparse index, the items are returned sorted by the date on which each order was placed.

Examples of Sparse Indexes in DynamoDB

Global secondary indexes are sparse by default. When you create a global secondary index, you specify a partition key and optionally a sort key. Only items in the base table that contain those attributes appear in the index.

By designing a global secondary index to be sparse, you can provision it with lower write throughput than that of the base table, while still achieving excellent performance.

For example, a gaming application might track all scores of every user, but generally only needs to query a few high scores. The following design handles this scenario efficiently:

| Table | Primary Key | | Data Attributes... | | | | |
|-------|---------------|----------|--------------------|--------------------------------|-------------|-------------------------------------|--|
| | Partition Key | Sort Key | Attribute 1 | | Attribute 2 | | Attribute 3 |
| | Player_ID | Game_ID | Score: | 36,750 <i>(game score)</i> | Date: | 2017-11-14 <i>(date of game)</i> | |
| Rick | Rick | Game_1 | Score: | 36,750 <i>(game score)</i> | Date: | 2017-11-14 <i>(date of game)</i> | |
| | | Game_2 | Score: | 69,450 <i>(game score)</i> | Date: | 2017-12-31 <i>(date of game)</i> | |
| | | Game_3 | Score: | 135,900 <i>(game score)</i> | Date: | 2018-01-19 <i>(date of game)</i> | Award: Champ <i>(type of award)</i> |
| Padma | Padma | Game_4 | Score: | 25,350 <i>(game score)</i> | Date: | 2018-01-27 <i>(date of game)</i> | |
| | | Game_5 | Score: | 69,450 <i>(game score)</i> | Date: | 2028-01-19 <i>(date of game)</i> | |
| | | Game_6 | Score: | 147,300 <i>(game score)</i> | Date: | 2018-02-02 <i>(date of game)</i> | Award: Champ <i>(type of award)</i> |
| | | Game_7 | Score: | 169,100 <i>(game score)</i> | Date: | 2018-03-10 <i>(date of game)</i> | Award: Champ <i>(type of award)</i> |

Here, Rick has played three games and achieved Champ status in one of them. Padma has played four games and achieved Champ status in two of them. Notice that the `Award` attribute is present only in items where the user achieved an award. The associated global secondary index looks like the following:

| GSI | Primary Key | | Projected Attributes... | | | |
|-----|---------------|-----------|-------------------------|---------|------------|------|
| | Partition Key | Player_ID | Game_ID | Score | Date | |
| | | Award | Player_ID | Game_ID | Score | Date |
| | Champ | Rick | Game_3 | 135,900 | 2018-01-19 | |
| | | Padma | Game_6 | 147,300 | 2018-02-02 | |
| | | Padma | Game_7 | 169,100 | 2018-03-10 | |

The global secondary index contains only the high scores that are frequently queried, which are a small subset of the items in the base table.

Using Global Secondary Indexes for Materialized Aggregation Queries

Maintaining near real-time aggregations and key metrics on top of rapidly changing data is becoming increasingly valuable to businesses for making rapid decisions. For example, a music library might want to showcase its most downloaded songs in near-real time.

Consider the following music library table layout:

| Music Library Table | | | | | | |
|------------------------------|----------------|-------------------------|--|-------------------|---|--|
| Primary Key | | Data-Item Attributes... | | | | |
| Partition Key | Sort Key | Attribute 1 | | Attribute 2 | | Attribute 3 |
| Song-129 <i>(song ID)</i> | Details | Title: | Wild Love <i>(song title)</i> | Artist: | Argyboots <i>(artist or band name)</i> | Downloads: 15,314,822 <i>(lifetime total downloads)</i> |
| | | GSI Primary Key | | GSI Secondary Key | | |
| | Month-2018-01 | Month: | 2018-01 <i>(download month)</i> | MonthTotal: | 1,746,992 <i>(month total downloads)</i> | |
| | | Time: | 2018-01-01T00:00:07 <i>(download timestamp)</i> | | | |
| | | Time: | 2018-01-01T00:00:07 <i>(download timestamp)</i> | | | |
| | Dld-9349823681 | Time: | 2018-01-01T00:00:07 <i>(download timestamp)</i> | | | |

The table in this example stores songs with the `songID` as the partition key. You can enable Amazon DynamoDB Streams on this table and attach a Lambda function to the streams so that as each song is downloaded, an entry is added to the table with `Partition-Key=SongID` and `Sort-Key=DownloadID`. As these updates are made, they trigger a Lambda function in DynamoDB Streams. The Lambda function can aggregate and group the downloads by `songID` and update the top-level item, `Partition-Key=songID`, and `Sort-Key=Month`. Keep in mind that if a lambda execution fails just after writing the new aggregated value, it may be retried and aggregate the value more than once, leaving you with an approximate value.

To read the updates in near-real time, with single-digit millisecond latency, use the global secondary index with query conditions `Month=2018-01`, `ScanIndexForward=False`, `Limit=1`.

Another key optimization used here is that the global secondary index is a sparse index and is available only on the items that need to be queried to retrieve the data in real time. The global secondary index can serve additional workflows that need information on the top 10 songs that were popular, or any song downloaded in that month.

Overloading Global Secondary Indexes

Although Amazon DynamoDB has a default limit of 20 global secondary indexes per table, in practice, you can index across far more than 20 data fields. As opposed to a table in a relational database management system (RDBMS), in which the schema is uniform, a table in DynamoDB can hold many different types of data items at one time. In addition, the same attribute in different items can contain entirely different types of information.

Consider the following example of a DynamoDB table layout that saves a variety of different kinds of data.

| Primary Key | | Data-Item Attributes... | | | | |
|--------------------------------|-----------------|-------------------------|---|-------------|--|---------|
| Partition Key | Sort Key | Attribute 1 | | Attribute 2 | | ... |
| HR-974 <i>(employee ID)</i> | Employee_Name | Data: | Murphy, John <i>(employee name)</i> | Start: | 2008-11-08 <i>(start date)</i> | ...etc. |
| | YYYY-Q1 | Data: | \$5,477 <i>(order totals in USD)</i> | Name: | Murphy, John <i>(employee name)</i> | |
| | HR_confidential | Data: | 2008-11-08 <i>(hire date)</i> | Name: | Murphy, John <i>(employee name)</i> | ...etc. |
| | Warehouse_01 | Data: | Murphy, John <i>(employee name)</i> | | | |
| | v0_Job_title | Data: | Operator-1 <i>(job title)</i> | Start: | 2008-11-08 <i>(start date)</i> | ...etc. |
| | v1_Job_title | Data: | Operator-2 <i>(job title)</i> | Start: | 2016-11-04 <i>(start date)</i> | ...etc. |
| | v2_Job_title | Data: | Supervisor-1 <i>(job title)</i> | Start: | 2017-11-01 <i>(start date)</i> | ...etc. |

The Data attribute, which is common to all the items, has different content depending on its parent item. If you create a global secondary index for the table that uses the table's sort key as its partition key and the Data attribute as its sort key, you can make a variety of different queries using that single global secondary index. These queries might include the following:

- Look up an employee by name in the global secondary index, by searching on the Employee_Name attribute value.
- Use the global secondary index to find all employees working in a particular warehouse by searching on a warehouse ID (such as Warehouse_01).
- Get a list of recent hires, querying the global secondary index on HR_confidential as a partition key value and Data as the sort key value.

Using Global Secondary Index Write Sharding for Selective Table Queries

Applications frequently need to identify a small subset of items in an Amazon DynamoDB table that meet a certain condition. When these items are distributed randomly across the partition keys of the table, you could resort to a table scan to retrieve them. This option can be expensive, but it works well when a large number of items on the table meet the search condition. However, when the key space is large and the search condition is very selective, this strategy can cause a lot of unnecessary processing.

To enable selective queries across the entire key space, you can use write sharding by adding an attribute containing a (0–N) value to every item that you will use for the global secondary index partition key.

The following is an example of a schema that uses this in a Critical-Event workflow:

| Table | Primary Key <i>Partition Key</i> | Data Attributes... | | | | | |
|-----------|-------------------------------------|---------------------|--------------------------------|-------------|---|---|---------|
| | | Attribute 1 | | Attribute 2 | | Attribute 3 | |
| EID_12345 | Time: <i>event timestamp</i> | 2018-02-07T08:42:40 | State: <i>(event state)</i> | INFO | GSI PK: <i>(random: 0-N)</i> <i>(random GSI-PK value)</i> | GSI SK: <i>INFO#2018-02-07T08:42:40</i> <i>(composite state-time)</i> | ...etc. |
| EID_12346 | Time: <i>event timestamp</i> | 2018-02-07T08:32:40 | State: <i>(event state)</i> | CRITICAL | GSI PK: <i>(random: 0-N)</i> <i>(random GSI-PK value)</i> | GSI SK: <i>CRITICAL#2018-02-07T08:32:40</i> <i>(composite state-time)</i> | ...etc. |
| EID_12347 | Time: <i>event timestamp</i> | 2018-02-07T08:22:40 | State: <i>(event state)</i> | WARN | GSI PK: <i>(random: 0-N)</i> <i>(random GSI-PK value)</i> | GSI SK: <i>WARN#2018-02-07T08:22:40</i> <i>(composite state-time)</i> | ...etc. |
| EID_12348 | Time: <i>event timestamp</i> | 2018-02-07T08:12:40 | State: <i>(event state)</i> | INFO | GSI PK: <i>(random: 0-N)</i> <i>(random GSI-PK value)</i> | GSI SK: <i>INFO#2018-02-07T08:12:40</i> <i>(composite state-time)</i> | ...etc. |

| GSI | Primary Key | | Data Attributes... | | |
|--------|---|----------|--------------------|--|--|
| | Partition Key | Sort Key | Data Attributes... | | |
| GSI PK | GSI SK | ... | | | |
| [0-N] | INFO#2018-02-07T08:42:40 <i>(composite state-time)</i> | ...etc. | | | |
| [0-N] | CRITICAL#2018-02-07T08:32:40 <i>(composite state-time)</i> | ...etc. | | | |
| [0-N] | WARN#2018-02-07T08:22:40 <i>(composite state-time)</i> | ...etc. | | | |
| [0-N] | INFO#2018-02-07T08:12:40 <i>(composite state-time)</i> | ...etc. | | | |

Using this schema design, the event items are distributed across 0–N partitions on the GSI, allowing a scatter read using a sort condition on the composite key to retrieve all items with a given state during a specified time period.

This schema pattern delivers a highly selective result set at minimal cost, without requiring a table scan.

Using Global Secondary Indexes to Create an Eventually Consistent Replica

You can use a global secondary index to create an eventually consistent replica of a table. Creating a replica can allow you to do the following:

- **Set different provisioned read capacity for different readers.** For example, suppose that you have two applications: One application handles high-priority queries and needs the highest levels of read performance, whereas the other handles low-priority queries that can tolerate throttling of read activity.

If both of these applications read from the same table, a heavy read load from the low-priority application could consume all the available read capacity for the table. This would throttle the high-priority application's read activity.

Instead, you can create a replica through a global secondary index whose read capacity you can set separate from that of the table itself. You can then have your low-priority app query the replica instead of the table.

- **Eliminate reads from a table entirely.** For example, you might have an application that captures a high volume of clickstream activity from a website, and you don't want to risk having reads interfere with that. You can isolate this table and prevent reads by other applications (see [Using IAM Policy](#))

[Conditions for Fine-Grained Access Control \(p. 840\)](#), while letting other applications read a replica created using a global secondary index.

To create a replica, set up a global secondary index that has the same key schema as the parent table, with some or all of the non-key attributes projected into it. In applications, you can direct some or all read activity to this global secondary index rather than to the parent table. You can then adjust the provisioned read capacity of the global secondary index to handle those reads without changing the parent table's provisioned read capacity.

There is always a short propagation delay between a write to the parent table and the time when the written data appears in the index. In other words, your applications should take into account that the global secondary index replica is only *eventually consistent* with the parent table.

You can create multiple global secondary index replicas to support different read patterns. When you create the replicas, project only the attributes that each read pattern actually requires. An application can then consume less provisioned read capacity to obtain only the data it needs rather than having to read the item from the parent table. This optimization can result in significant cost savings over time.

Best Practices for Storing Large Items and Attributes

Amazon DynamoDB currently limits the size of each item that you store in a table (see [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#)). If your application needs to store more data in an item than the DynamoDB size limit permits, you can try compressing one or more large attributes or breaking the item into multiple items (efficiently indexed by sort keys). You can also store the item as an object in Amazon Simple Storage Service (Amazon S3) and store the Amazon S3 object identifier in your DynamoDB item.

Compressing Large Attribute Values

Compressing large attribute values can let them fit within item limits in DynamoDB and reduce your storage costs. Compression algorithms such as GZIP or LZO produce binary output that you can then store in a `Binary` attribute type.

For example, the `Reply` table in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section stores messages written by forum users. These user replies might consist of long strings of text, which makes them excellent candidates for compression.

For sample code that demonstrates how to compress such messages in DynamoDB, see the following:

- [Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API \(p. 432\)](#)
- [Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API \(p. 455\)](#)

Storing Large Attribute Values in Amazon S3

As mentioned previously, you can also use Amazon S3 to store large attribute values that cannot fit in a DynamoDB item. You can store them as an object in Amazon S3 and then store the object identifier in your DynamoDB item.

You can also use the object metadata support in Amazon S3 to provide a link back to the parent item in DynamoDB. Store the primary key value of the item as Amazon S3 metadata of the object in Amazon S3. Doing this often helps with maintenance of the Amazon S3 objects.

For example, consider the [ProductCatalog](#) table in the [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#) section. Items in this table store information about item price, description, book authors, and dimensions for other products. If you wanted to store an image of each product that was too large to fit in an item, you could store the images in Amazon S3 instead of in DynamoDB.

When implementing this strategy, keep the following in mind:

- DynamoDB doesn't support transactions that cross Amazon S3 and DynamoDB. Therefore, your application must deal with any failures, which could include cleaning up orphaned Amazon S3 objects.
- Amazon S3 limits the length of object identifiers. So you must organize your data in a way that doesn't generate excessively long object identifiers or violate other Amazon S3 constraints.

For more information about how to use Amazon S3, see the [Amazon Simple Storage Service Developer Guide](#).

Best Practices for Handling Time Series Data in DynamoDB

General design principles in Amazon DynamoDB recommend that you keep the number of tables you use to a minimum. For most applications, a single table is all you need. However, for time series data, you can often best handle it by using one table per application per period.

Design Pattern for Time Series Data

Consider a typical time series scenario, where you want to track a high volume of events. Your write access pattern is that all the events being recorded have today's date. Your read access pattern might be to read today's events most frequently, yesterday's events much less frequently, and then older events very little at all. One way to handle this is by building the current date and time into the primary key.

The following design pattern often handles this kind of scenario effectively:

- Create one table per period, provisioned with the required read and write capacity and the required indexes.
- Before the end of each period, prebuild the table for the next period. Just as the current period ends, direct event traffic to the new table. You can assign names to these tables that specify the periods they have recorded.
- As soon as a table is no longer being written to, reduce its provisioned write capacity to a lower value (for example, 1 WCU), and provision whatever read capacity is appropriate. Reduce the provisioned read capacity of earlier tables as they age. You might choose to archive or delete the tables whose contents are rarely or never needed.

The idea is to allocate the required resources for the current period that will experience the highest volume of traffic and scale down provisioning for older tables that are not used actively, therefore saving costs. Depending on your business needs, you might consider write sharding to distribute traffic evenly to the logical partition key. For more information, see [Using Write Sharding to Distribute Workloads Evenly \(p. 903\)](#).

Time Series Table Examples

The following is a time series data example in which the current table is provisioned at a higher read/write capacity and the older tables are scaled down because they are accessed infrequently.

| Current table | | Promised at: WCU=750 and RCU=100 | |
|---------------|--------------|----------------------------------|-----------|
| Primary Key | | Attributes | |
| Partition Key | Sort Key | Attribute 1 | |
| 2018-01-13 | 00:00:00:002 | 17.173 WCW | 713 items |
| 2018-01-13 | 00:00:00:003 | 14.498 WCW | 713 items |
| 2018-01-13 | 00:00:00:005 | 17.479 WCW | 708 items |
| 2018-01-13 | 00:00:00:007 | 18.173 WCW | 674 items |
| *** | *** | *** | *** |

| Previous table | | Promised at: WCU=111 and RCU=100 | |
|----------------|--------------|----------------------------------|-----------|
| Primary Key | | Attributes | |
| Partition Key | Sort Key | Attribute 1 | |
| 2018-01-13 | 00:00:00:002 | 14.498 WCW | 712 items |
| 2018-01-14 | 00:00:00:003 | 14.498 WCW | 818 items |
| 2018-01-14 | 00:00:00:004 | 14.498 WCW | 818 items |
| 2018-01-14 | 00:00:00:006 | 18.173 WCW | 822 items |
| *** | *** | *** | *** |

| Older table | | Promised at: WCU=111 and RCU=100 | |
|---------------|--------------|----------------------------------|-----------|
| Primary Key | | Attributes | |
| Partition Key | Sort Key | Attribute 1 | |
| 2018-01-10 | 00:00:00:001 | 13.448 WCW | 406 items |
| 2018-01-10 | 00:00:00:002 | 13.532 WCW | 409 items |
| 2018-01-10 | 00:00:00:003 | 13.532 WCW | 409 items |
| 2018-01-10 | 00:00:00:005 | 18.173 WCW | 426 items |
| *** | *** | *** | *** |

Best Practices for Managing Many-to-Many Relationships

Adjacency lists are a design pattern that is useful for modeling many-to-many relationships in Amazon DynamoDB. More generally, they provide a way to represent graph data (nodes and edges) in DynamoDB.

Adjacency List Design Pattern

When different entities of an application have a many-to-many relationship between them, the relationship can be modeled as an adjacency list. In this pattern, all top-level entities (synonymous to nodes in the graph model) are represented using the partition key. Any relationships with other entities (edges in a graph) are represented as an item within the partition by setting the value of the sort key to the target entity ID (target node).

The advantages of this pattern include minimal data duplication and simplified query patterns to find all entities (nodes) related to a target entity (having an edge to a target node).

A real-world example where this pattern has been useful is an invoicing system where invoices contain multiple bills. One bill can belong in multiple invoices. The partition key in this example is either an `InvoiceID` or a `BillID`. `BillID` partitions have all attributes specific to bills. `InvoiceID` partitions have an item storing invoice-specific attributes, and an item for each `BillID` that rolls up to the invoice.

The schema looks like the following.

| Table | Primary Key | | Data Attributes... | | |
|---------------|---------------|-------------------------------|--------------------|------------------------------|--|
| | Partition Key | Sort Key (and GSI PK) | Dated: | 2018-02-07 (date created) | More attributes of this invoice... |
| Invoice-92551 | Inv_ID: | Invoice-92551 (invoice ID) | Dated: | 2018-02-07 (date created) | More attributes of this invoice... |
| | Bill_ID: | Bill-4224663 (bill ID) | Dated: | 2017-12-03 (date created) | Attributes of this bill <i>in this invoice..</i> |
| | Bill_ID: | Bill-4224687 (bill ID) | Dated: | 2018-01-09 (date created) | Attributes of this bill <i>in this invoice..</i> |
| Invoice-92552 | Inv_ID: | Invoice-92552 (invoice ID) | Dated: | 2018-03-04 (date created) | More attributes of this invoice... |
| | Bill_ID: | Bill-4224687 (bill ID) | Dated: | 2018-01-09 (date created) | Attributes of this bill <i>in this invoice..</i> |
| Bill-4224663 | Bill_ID: | Bill-4224663 (bill ID) | Dated: | 2017-12-03 (date created) | More attributes of this bill... |
| | Bill_ID: | Bill-4224687 (bill ID) | Dated: | 2018-01-09 (date created) | More attributes of this bill... |

Using the preceding schema, you can see that all bills for an invoice can be queried using the primary key on the table. To look up all invoices that contain a part of a bill, create a global secondary index on the table's sort key.

The projections for the global secondary index look like the following.

| GSI | Primary Key | Projected Attributes... | |
|---------------|---------------|---|---|
| | Partition Key | | |
| Bill-4224663 | Bill_ID: | Bill-4224663 <i>(table primary key)</i> | Attributes of this bill... |
| | Inv_ID: | Invoice-92551 <i>(table primary key)</i> | Attributes of this bill <i>in this invoice</i> .. |
| Bill-4224687 | Bill_ID: | Bill-4224687 <i>(table primary key)</i> | Attributes of this bill... |
| | Inv_ID: | Invoice-92551 <i>(table primary key)</i> | Attributes of this bill <i>in this invoice</i> .. |
| | Inv_ID: | Invoice-92552 <i>(table primary key)</i> | Attributes of this bill <i>in this invoice</i> .. |
| Invoice-92551 | Inv_ID: | Invoice-92551 <i>(table primary key)</i> | Attributes of this invoice... |
| Invoice-92552 | Inv_ID: | Invoice-92552 <i>(table primary key)</i> | Attributes of this invoice... |

Materialized Graph Pattern

Many applications are built around understanding rankings across peers, common relationships between entities, neighbor entity state, and other types of graph style workflows. For these types of applications, consider the following schema design pattern.

| PK (NodeID) | Primary Key | | Attributes | | |
|-------------|---------------------------|--|-----------------------|-------------------|-------------------|
| | SK (TypeTarget, GSI 2 SK) | | Data | GSI PK | Graph Projections |
| 1 | DATE 2 BIRTH | | 1980-12-19 | Hash(Person.Data) | ... |
| | PERSON 1 | | Data (GSI1 SK) | GSI PK | |
| | PERSON 5 FRIEND | | John Doe | Hash(Person.Data) | |
| | PLACE 4 BIRTH | | Data | GSI PK | |
| | SKILL 6 | | Jane Smith | Hash(Person.Data) | |
| | DATE 2 | | Data | GSI PK | |
| | PLACE 3 | | USA Texas Austin | Hash(Person.Data) | |
| | PLACE 4 | | Data | GSI PK | |
| | DATE 2 BIRTH | | Java Developer Senior | Hash(Person.Data) | |
| | PERSON 5 | | Data | GSI PK | |
| 5 | PERSON 1 FRIEND | | 1980-12-19 | 0 | ... |
| | PLACE 3 BIRTH | | Data | GSI PK | |
| | SKILL 7 | | UK England London | 0 | |
| | SKILL 6 | | Data | GSI PK | |
| | SKILL 7 | | USA Texas Austin | 0 | |
| | DATE 2 | | Data | GSI PK | |
| | PLACE 3 | | Jane Smith | Hash(Person.Data) | |
| | PLACE 4 | | Data | GSI PK | |
| | DATE 2 BIRTH | | John Doe | Hash(Person.Data) | |
| | SKILL 7 | | Data | GSI PK | |
| O-N | SKILL 6 | | Guitar Advanced | Hash(Person.Data) | ... |
| | SKILL 7 | | Data | GSI PK | |
| | DATE 2 | | Java Developer | 0 | |
| | PLACE 3 | | Data | GSI PK | |
| | PLACE 4 | | Guitar | 0 | |
| | DATE 2 BIRTH | | Java Developer Senior | GSI PK | |
| | PERSON 5 | | Data | GSI PK | |
| | PERSON 1 FRIEND | | Jane Smith | Hash(Person.Data) | |
| | SKILL 6 | | John Doe | Hash(Person.Data) | |
| | SKILL 7 | | UK England London | Hash(Person.Data) | |
| GSI 1 | SKILL 7 | | USA Texas Austin | Hash(Person.Data) | ... |
| | DATE 2 BIRTH | | Data | GSI PK | |
| | PERSON 5 | | Jane Smith | Hash(Person.Data) | |
| | PERSON 1 FRIEND | | John Doe | Hash(Person.Data) | |
| | SKILL 6 | | UK England London | Hash(Person.Data) | |
| | SKILL 7 | | USA Texas Austin | Hash(Person.Data) | |
| | DATE 2 | | Data | GSI PK | |
| | PLACE 3 | | Guitar | 0 | |
| | PLACE 4 | | Java Developer | 0 | |
| | DATE 2 BIRTH | | PLACE 3 BIRTH | GSI PK | |

| GSI PK | Primary Key GSI 2 SK (TypeTarget) | Attributes | | | |
|--------|--------------------------------------|------------|-----------------------|-------------------|--|
| | | NodeID | Data | Graph Projections | |
| GSI 2 | DATE 2 | 2 | 1980-12-19 | O-N | |
| | DATE 2 BIRTH | 1 | | | |
| | PERSON 1 | 1 | John Doe | | |
| | PERSON 1 FRIEND | 5 | | | |
| | PERSON 5 | 5 | Jane Smith | | |
| | PERSON 5 FRIEND | 1 | | | |
| | PLACE 3 | 3 | UK England London | | |
| | PLACE 3 BIRTH | 5 | | | |
| | PLACE 4 | 4 | USA texas Austin | | |
| | PLACE 4 BIRTH | 1 | | | |
| | SKILL 6 | 6 | Java Developer | ... | |
| | | NodeID | Data | | |
| | | 1 | Java Developer Senior | | |
| | | 7 | Guitar | | |
| | SKILL 7 | NodeID | Data | | |
| | | 5 | Guitar Advanced | | |

The preceding schema shows a graph data structure that is defined by a set of data partitions containing the items that define the edges and nodes of the graph. Edge items contain a `Type` and a `Target` attribute. These attributes are used as part of a composite key name "TypeTarget" to identify the item in a partition in the primary table or in a second global secondary index.

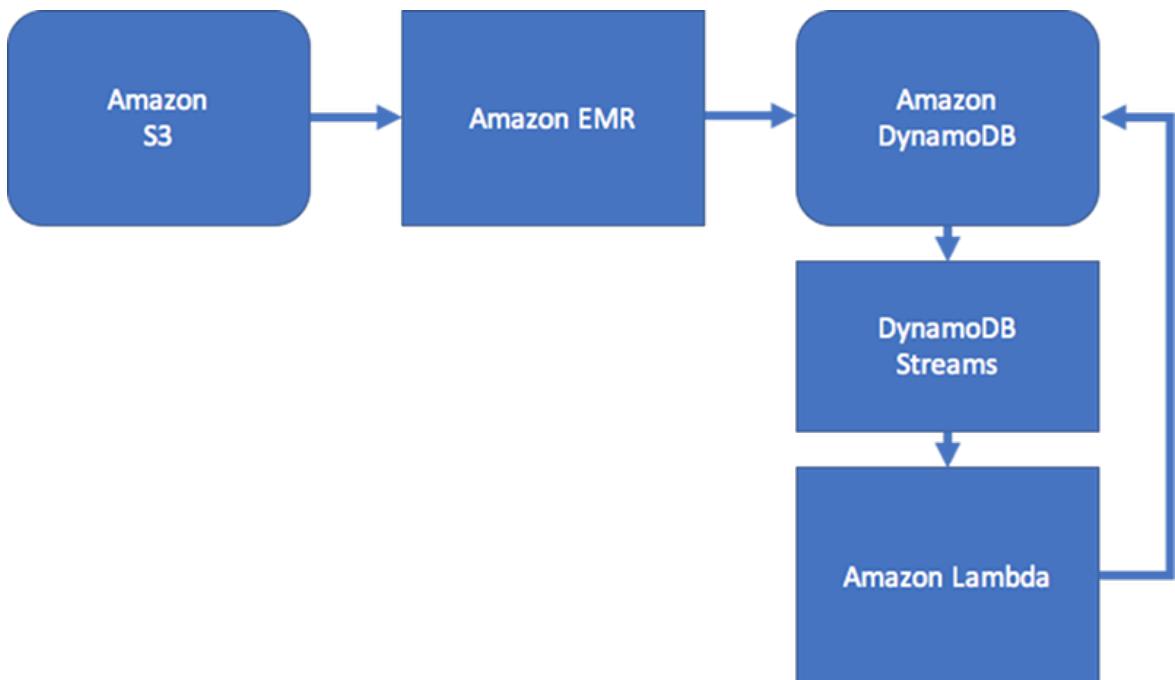
The first global secondary index is built on the `Data` attribute. This attribute uses global secondary index-overloading as described earlier to index several different attribute types, namely Dates, Names, Places, and Skills. Here, one global secondary index is effectively indexing four different attributes.

As you insert items into the table, you can use an intelligent sharding strategy to distribute item sets with large aggregations (birthdate, skill) across as many logical partitions on the global secondary indexes as are needed to avoid hot read/write problems.

The result of this combination of design patterns is a solid datastore for highly efficient real-time graph workflows. These workflows can provide high-performance neighbor entity state and edge aggregation queries for recommendation engines, social-networking applications, node rankings, subtree aggregations, and other common graph use cases.

If your use case isn't sensitive to real-time data consistency, you can use a scheduled Amazon EMR process to populate edges with relevant graph summary aggregations for your workflows. If your application doesn't need to know immediately when an edge is added to the graph, you can use a scheduled process to aggregate results.

To maintain some level of consistency, the design could include Amazon DynamoDB Streams and AWS Lambda to process edge updates. It could also use an Amazon EMR job to validate results on a regular interval. This approach is illustrated by the following diagram. It is commonly used in social networking applications, where the cost of a real-time query is high and the need to immediately know individual user updates is low.



IT service-management (ITSM) and security applications generally need to respond in real time to entity state changes composed of complex edge aggregations. Such applications need a system that can support real-time multiple node aggregations of second- and third-level relationships, or complex edge traversals. If your use case requires these types of real-time graph query workflows, we recommend that you consider using [Amazon Neptune](#) to manage these workflows.

Best Practices for Implementing a Hybrid Database System

In some circumstances, migrating from one or more relational database management systems (RDBMS) to Amazon DynamoDB might not be advantageous. In these cases, it might be preferable to create a hybrid system.

If You Don't Want to Migrate Everything to DynamoDB

For example, some organizations have large investments in the code that produces a multitude of reports needed for accounting and operations. The time it takes to generate a report is not important to them. The flexibility of a relational system is well suited to this kind of task, and re-creating all those reports in a NoSQL context might be prohibitively difficult.

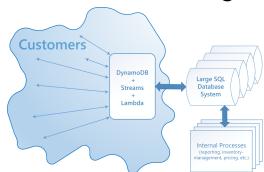
Some organizations also maintain a variety of legacy relational systems that they have acquired or inherited over decades. Migrating data from these systems might be too risky and expensive to justify the effort.

However, the same organizations may now find that their operations depend on high-traffic customer-facing websites, where millisecond response is essential. Relational systems can't scale to meet this requirement except at huge (and often unacceptable) expense.

In these situations, the answer might be to create a hybrid system, in which DynamoDB creates a materialized view of data stored in one or more relational systems and handles high-traffic requests against this view. This type of system can potentially reduce costs by eliminating server hardware, maintenance, and RDBMS licenses that were previously needed to handle customer-facing traffic.

How a Hybrid System Can Be Implemented

DynamoDB can take advantage of DynamoDB Streams and AWS Lambda to integrate seamlessly with one or more existing relational database systems:



A system that integrates DynamoDB Streams and AWS Lambda can provide several advantages:

- It can operate as a persisted cache of materialized views.
- It can be set up to fill gradually with data as that data is queried for, and as data is modified in the SQL system. This means that the entire view does not need to be pre-populated. This in turn means that provisioned throughput capacity is more likely to be used efficiently.
- It has low administrative costs and is highly available and reliable.

For this kind of integration to be implemented, essentially three kinds of interoperation must be provided.



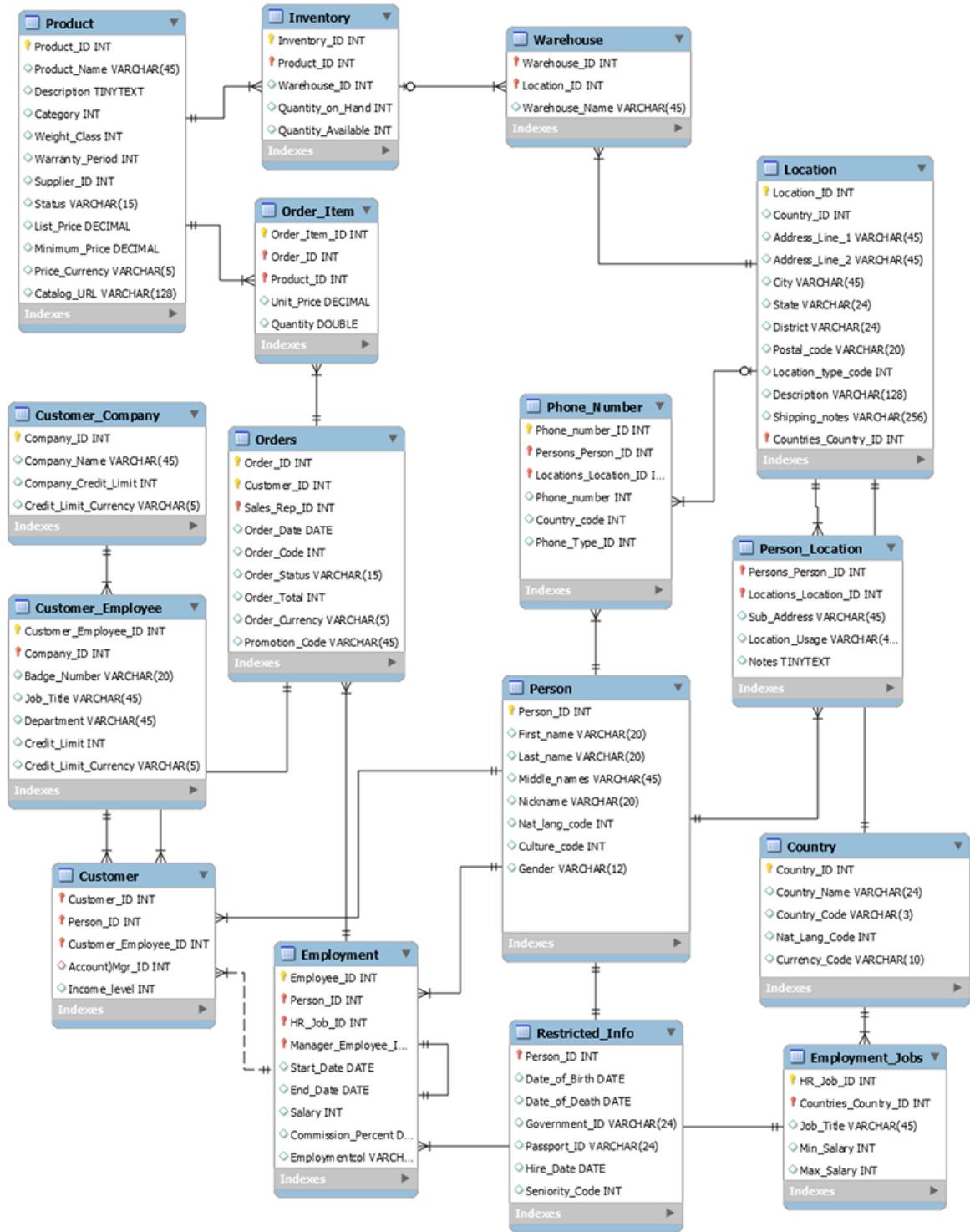
1. **Fill the DynamoDB cache incrementally.** When an item is queried, look for it first in DynamoDB. If it is not there, look for it in the SQL system, and load it into DynamoDB.
2. **Write through a DynamoDB cache.** When a customer changes a value in DynamoDB, a Lambda function is triggered to write the new data back to the SQL system.
3. **Update DynamoDB from the SQL system.** When internal processes such as inventory management or pricing change a value in the SQL system, a stored procedure is triggered to propagate the change to the DynamoDB materialized view.

These operations are straightforward, and not all of them are needed for every scenario.

A hybrid solution can also be useful when you want to rely primarily on DynamoDB, but you also want to maintain a small relational system for one-time queries, or for operations that need special security or that are not time-critical.

Best Practices for Modeling Relational Data in DynamoDB

Traditional relational database management system (RDBMS) platforms store data in a normalized relational structure. This structure reduces hierarchical data structures to a set of common elements that are stored across multiple tables. The following schema is an example of a generic order-entry application with supporting HR schema backing the operational and business support systems of a theoretical manufacturer.



RDBMS platforms use an ad hoc query language (generally a flavor of SQL) to generate or materialize views of the normalized data to support application-layer access patterns.

For example, to generate a list of purchase order items sorted by the quantity in stock at all warehouses that can ship each item, you could issue the following query against the preceding schema.

```
SELECT * FROM Orders
INNER JOIN Order_Items ON Orders.Order_ID = Order_Items.Order_ID
INNER JOIN Products ON Products.Product_ID = Order_Items.Product_ID
INNER JOIN Inventories ON Products.Product_ID = Inventories.Product_ID
ORDER BY Quantity_on_Hand DESC
```

One-time queries of this kind provide a flexible API for accessing data, but they require a significant amount of processing. You must often query the data from multiple locations, and the results must be assembled for presentation. The preceding query initiates complex queries across a number of tables and then sorts and integrates the resulting data.

Another factor that can slow down RDBMS systems is the need to support an ACID-compliant transaction framework. The hierarchical data structures used by most online transaction processing (OLTP) applications must be broken down and distributed across multiple logical tables when they are stored in an RDBMS. Therefore, an ACID-compliant transaction framework is necessary to avoid race conditions that could occur if an application tries to read an object that is in the process of being written. Such a transaction framework necessarily adds significant overhead to the write process.

These two factors are the primary barriers to scale for traditional RDBMS platforms. It remains to be seen whether the NewSQL community can be successful in delivering a distributed RDBMS solution. But it is unlikely that even that would resolve the two limitations described earlier. No matter how the solution is delivered, the processing costs of normalization and ACID transactions must remain significant.

For this reason, when your business requires low-latency response to high-traffic queries, taking advantage of a NoSQL system generally makes technical and economic sense. Amazon DynamoDB helps solve the problems that limit relational system scalability by avoiding them.

A relational database system does not scale well for the following reasons:

- It normalizes data and stores it on multiple tables that require multiple queries to write to disk.
- It generally incurs the performance costs of an ACID-compliant transaction system.
- It uses expensive joins to reassemble required views of query results.

DynamoDB scales well for these reasons:

- Schema flexibility lets DynamoDB store complex hierarchical data within a single item.
- Composite key design lets it store related items close together on the same table.

Queries against the data store become much simpler, often in the following form:

```
SELECT * FROM Table_X WHERE Attribute_Y = "somevalue"
```

DynamoDB does far less work to return the requested data compared to the RDBMS in the earlier example.

First Steps for Modeling Relational Data in DynamoDB

Important

NoSQL design requires a different mindset than RDBMS design. For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise. By contrast, in Amazon DynamoDB, you shouldn't start designing your schema until you know the questions that it needs to answer. Understanding the business problems and the application use cases up front is absolutely essential.

To start designing a DynamoDB table that will scale efficiently, you must take several steps first to identify the access patterns that are required by the operations and business support systems (OSS/BSS) that it needs to support:

- For new applications, review user stories about activities and objectives. Document the various use cases you identify, and analyze the access patterns that they require.
- For existing applications, analyze query logs to find out how people are currently using the system and what the key access patterns are.

After completing this process, you should end up with a list that might look something like the following.

| Most Common/Import Access Patterns in Our Organization | |
|--|---|
| 1 | Look up employee details by employee ID |
| 2 | Query employee details by employee name |
| 3 | Find an employee's phone number(s) |
| 4 | Find a customer's phone number(s) |
| 5 | Get orders for a given customer within a given date range |
| 6 | Show all open orders within a given date range across all customers |
| 7 | See all employees hired recently |
| 8 | Find all employees working in a given warehouse |
| 9 | Get all items on order for a given product |
| 10 | Get current inventories for a given product at all warehouses |
| 11 | Get customers by account representative |
| 12 | Get orders by account representative and date |
| 13 | Get all items on order for a given product |
| 14 | Get all employees with a given job title |
| 15 | Get inventory by product and warehouse |
| 16 | Get total product inventory |
| 17 | Get account representatives ranked by order total and sales period |

In a real application, your list might be much longer. But this collection represents the range of query pattern complexity that you might find in a production environment.

A common approach to DynamoDB schema design is to identify application layer entities and use denormalization and composite key aggregation to reduce query complexity.

In DynamoDB, this means using composite sort keys, overloaded global secondary indexes, partitioned tables/indexes, and other design patterns. You can use these elements to structure the data so that an application can retrieve whatever it needs for a given access pattern using a single query on a table or index. The primary pattern that you can use to model the normalized schema shown in [Relational Modeling \(p. 920\)](#) is the adjacency list pattern. Other patterns used in this design can include global secondary index write sharding, global secondary index overloading, composite keys, and materialized aggregations.

Important

In general, you should maintain as few tables as possible in a DynamoDB application. Exceptions include cases where high-volume time series data are involved, or datasets that have very different access patterns. A single table with inverted indexes can usually enable simple queries to create and retrieve the complex hierarchical data structures required by your application.

Example of Modeling Relational Data in DynamoDB

This example describes how to model relational data in Amazon DynamoDB. A DynamoDB table design corresponds to the relational order entry schema that is shown in [Relational Modeling \(p. 920\)](#). It

follows the [Adjacency List Design Pattern \(p. 915\)](#), which is a common way to represent relational data structures in DynamoDB.

The design pattern requires you to define a set of entity types that usually correlate to the various tables in the relational schema. Entity items are then added to the table using a compound (partition and sort) primary key. The partition key of these entity items is the attribute that uniquely identifies the item and is referred to generically on all items as **PK**. The sort key attribute contains an attribute value that you can use for an inverted index or global secondary index. It is generically referred to as **SK**.

You define the following entities, which support the relational order entry schema.

1. HR-Employee - PK: EmployeeID, SK: Employee Name
 2. HR-Region - PK: RegionID, SK: Region Name
 3. HR-Country - PK: CountryId, SK: Country Name
 4. HR-Location - PK: LocationID, SK: Country Name
 5. HR-Job - PK: JobID, SK: Job Title
 6. HR-Department - PK: DepartmentID, SK: DepartmentID
 7. OE-Customer - PK: CustomerID, SK: AccountRepID
 8. OE-Order - PK OrderID, SK: CustomerID
 9. OE-Product - PK: ProductID, SK: Product Name
- 10OE-Warehouse - PK: WarehouseID, SK: Region Name

After adding these entity items to the table, you can define the relationships between them by adding edge items to the entity item partitions. The following table demonstrates this step.

In this example, the `Employee`, `Order`, and `Product` Entity partitions on the table have additional edge items that contain pointers to other entity items on the table. Next, define a few global secondary indexes (GSIs) to support all the access patterns defined previously. The entity items don't all use the same type of value for the primary key or the sort key attribute. All that is required is to have the primary key and sort key attributes present to be inserted on the table.

The fact that some of these entities use proper names and others use other entity IDs as sort key values allows the same global secondary index to support multiple types of queries. This technique is called *GSI overloading*. It effectively eliminates the default limit of 20 global secondary indexes for tables that contain multiple item types. This is shown in the following diagram as *GSI 1*.

GSI 2 is designed to support a fairly common application access pattern, which is to get all the items on the table that have a certain state. For a large table with an uneven distribution of items across available states, this access pattern can result in a hot key, unless the items are distributed across more than one logical partition that can be queried in parallel. This design pattern is called *write sharding*.

To accomplish this for GSI 2, the application adds the GSI 2 primary key attribute to every Order item. It populates that with a random number in a range of 0–N, where N can generically be calculated using the following formula, unless there is a specific reason to do otherwise.

```
ItemsPerRCU = 4KB / AvgItemSize
PartitionMaxReadRate = 3K * ItemsPerRCU
N = MaxRequiredIO / PartitionMaxReadRate
```

For example, assume that you expect the following:

- Up to 2 million orders will be in the system, growing to 3 million in 5 years.

- Up to 20 percent of these orders will be in an OPEN state at any given time.
- The average order record is around 100 bytes, with three OrderItem records in the order partition that are around 50 bytes each, giving you an average order entity size of 250 bytes.

For that table, the N factor calculation would look like the following.

```
ItemsPerRCU = 4KB / 250B = 16
PartitionMaxReadRate = 3K * 16 = 48K
N = (0.2 * 3M) / 48K = 13
```

In this case, you need to distribute all the orders across at least 13 logical partitions on GSI 2 to ensure that a read of all `Order` items with an OPEN status doesn't cause a hot partition on the physical storage layer. It is a good practice to pad this number to allow for anomalies in the dataset. So a model using `N = 15` is probably fine. As mentioned earlier, you do this by adding the random 0–N value to the GSI 2 PK attribute of each `Order` and `OrderItem` record that is inserted on the table.

This breakdown assumes that the access pattern that requires gathering all OPEN invoices occurs relatively infrequently so that you can use burst capacity to fulfill the request. You can query the following global secondary index using a State and Date Range Sort Key condition to produce a subset or all Orders in a given state as needed.

In this example, the items are randomly distributed across the 15 logical partitions. This structure works because the access pattern requires a large number of items to be retrieved. Therefore, it's unlikely that any of the 15 threads will return empty result sets that could potentially represent wasted capacity. A query always uses 1 read capacity unit (RCU) or 1 write capacity unit (WCU), even if nothing is returned or no data is written.

If the access pattern requires a high velocity query on this global secondary index that returns a sparse result set, it's probably better to use a hash algorithm to distribute the items rather than a random pattern. In this case, you might select an attribute that is known when the query is executed at runtime and hash that attribute into a 0–14 key space when the items are inserted. Then they can be efficiently read from the global secondary index.

Finally, you can revisit the access patterns that were defined earlier. Following is the list of access patterns and the query conditions that you will use with the new DynamoDB version of the application to accommodate them.

Best Practices for Querying and Scanning Data

This section covers some best practices for using `Query` and `Scan` operations in Amazon DynamoDB.

Performance Considerations for Scans

In general, `Scan` operations are less efficient than other operations in DynamoDB. A `Scan` operation always scans the entire table or secondary index. It then filters out values to provide the result you want, essentially adding the extra step of removing data from the result set.

If possible, you should avoid using a `Scan` operation on a large table or index with a filter that removes many results. Also, as a table or index grows, the `Scan` operation slows. The `Scan` operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation. For faster response times, design your tables and indexes so that your

applications can use `Query` instead of `Scan`. (For tables, you can also consider using the `GetItem` and `BatchGetItem` APIs.)

Alternatively, design your application to use `Scan` operations in a way that minimizes the impact on your request rate.

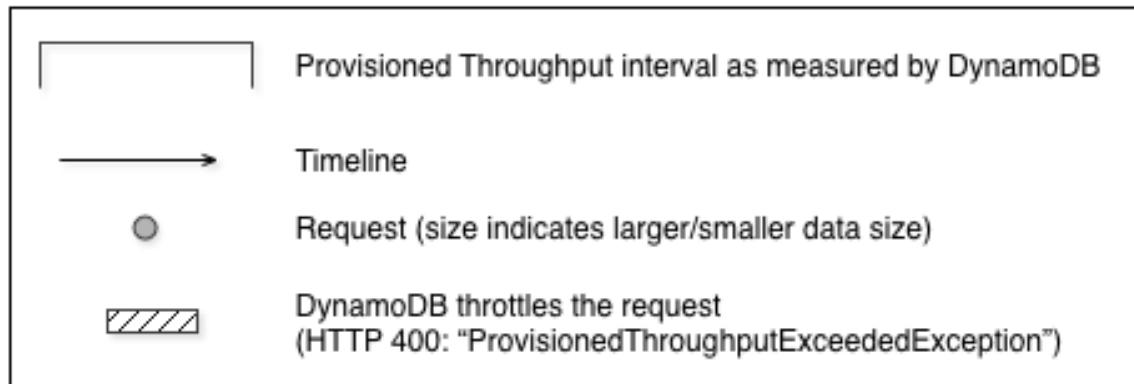
Avoiding Sudden Spikes in Read Activity

When you create a table, you set its read and write capacity unit requirements. For reads, the capacity units are expressed as the number of strongly consistent 4 KB data read requests per second. For eventually consistent reads, a read capacity unit is two 4 KB read requests per second. A `Scan` operation performs eventually consistent reads by default, and it can return up to 1 MB (one page) of data. Therefore, a single `Scan` request can consume $(1 \text{ MB page size} / 4 \text{ KB item size}) / 2$ (eventually consistent reads) = 128 read operations. If you request strongly consistent reads instead, the `Scan` operation would consume twice as much provisioned throughput—256 read operations.

This represents a sudden spike in usage, compared to the configured read capacity for the table. This usage of capacity units by a `scan` prevents other potentially more important requests for the same table from using the available capacity units. As a result, you likely get a `ProvisionedThroughputExceeded` exception for those requests.

The problem is not just the sudden increase in capacity units that the `Scan` uses. The `scan` is also likely to consume all of its capacity units from the same partition because the `scan` requests read items that are next to each other on the partition. This means that the request is hitting the same partition, causing all of its capacity units to be consumed, and throttling other requests to that partition. If the request to read data is spread across multiple partitions, the operation would not throttle a specific partition.

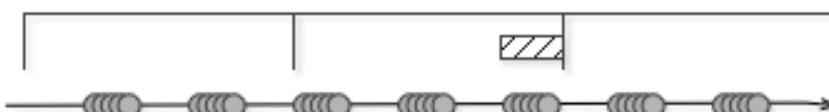
The following diagram illustrates the impact of a sudden spike of capacity unit usage by `Query` and `Scan` operations, and its impact on your other requests against the same table.



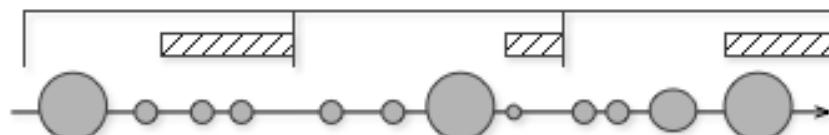
1. Good: Even distribution of requests and size



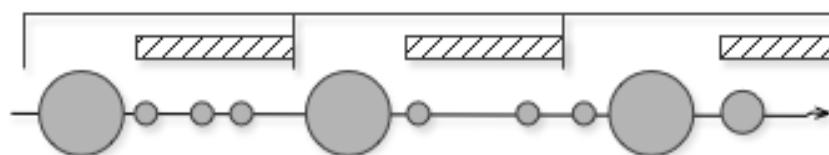
2. Not as Good: Frequent requests in bursts



3. Bad: A few random large requests



4. Bad: Large scan operations



As illustrated here, the usage spike can impact the table's provisioned throughput in several ways:

1. Good: Even distribution of requests and size
2. Not as good: Frequent requests in bursts
3. Bad: A few random large requests
4. Bad: Large scan operations

Instead of using a large Scan operation, you can use the following techniques to minimize the impact of a scan on a table's provisioned throughput.

- **Reduce page size**

Because a Scan operation reads an entire page (by default, 1 MB), you can reduce the impact of the scan operation by setting a smaller page size. The Scan operation provides a *Limit* parameter that you can use to set the page size for your request. Each Query or Scan request that has a smaller page size uses fewer read operations and creates a "pause" between each request. For example, suppose that each item is 4 KB and you set the page size to 40 items. A Query request would then consume only 20 eventually consistent read operations or 40 strongly consistent read operations. A larger number of smaller Query or Scan operations would allow your other critical requests to succeed without throttling.

- **Isolate scan operations**

DynamoDB is designed for easy scalability. As a result, an application can create tables for distinct purposes, possibly even duplicating content across several tables. You want to perform scans on a table that is not taking "mission-critical" traffic. Some applications handle this load by rotating traffic hourly between two tables—one for critical traffic, and one for bookkeeping. Other applications can do this by performing every write on two tables: a "mission-critical" table, and a "shadow" table.

Configure your application to retry any request that receives a response code that indicates you have exceeded your provisioned throughput. Or, increase the provisioned throughput for your table using the `UpdateTable` operation. If you have temporary spikes in your workload that cause your throughput to exceed, occasionally, beyond the provisioned level, retry the request with exponential backoff. For more information about implementing exponential backoff, see [Error Retries and Exponential Backoff \(p. 224\)](#).

Taking Advantage of Parallel Scans

Many applications can benefit from using parallel Scan operations rather than sequential scans. For example, an application that processes a large table of historical data can perform a parallel scan much faster than a sequential one. Multiple worker threads in a background "sweeper" process could scan a table at a low priority without affecting production traffic. In each of these examples, a parallel Scan is used in such a way that it does not starve other applications of provisioned throughput resources.

Although parallel scans can be beneficial, they can place a heavy demand on provisioned throughput. With a parallel scan, your application has multiple workers that are all running Scan operations concurrently. This can quickly consume all of your table's provisioned read capacity. In that case, other applications that need to access the table might be throttled.

A parallel scan can be the right choice if the following conditions are met:

- The table size is 20 GB or larger.
- The table's provisioned read throughput is not being fully used.
- Sequential Scan operations are too slow.

Choosing TotalSegments

The best setting for `TotalSegments` depends on your specific data, the table's provisioned throughput settings, and your performance requirements. You might need to experiment to get it right. We recommend that you begin with a simple ratio, such as one segment per 2 GB of data. For example, for a 30 GB table, you could set `TotalSegments` to 15 (30 GB / 2 GB). Your application would then use 15 workers, with each worker scanning a different segment.

You can also choose a value for `TotalSegments` that is based on client resources. You can set `TotalSegments` to any number from 1 to 1000000, and DynamoDB lets you scan that number of

segments. For example, if your client limits the number of threads that can run concurrently, you can gradually increase `TotalSegments` until you get the best `Scan` performance with your application.

Monitor your parallel scans to optimize your provisioned throughput use, while also making sure that your other applications aren't starved of resources. Increase the value for `TotalSegments` if you don't consume all of your provisioned throughput but still experience throttling in your `Scan` requests. Reduce the value for `TotalSegments` if the `Scan` requests consume more provisioned throughput than you want to use.

Best Practices for Using Global Tables

There are several requirements and best practices that you must consider when using global tables. For more information, see [Best Practices and Requirements for Managing Global Tables \(p. 627\)](#).

DynamoDB Integration with Other AWS Services

Amazon DynamoDB is integrated with other AWS services, letting you automate repeating tasks or build applications that span multiple services. For example:

Topics

- [Configuring AWS Credentials in Your Files Using Amazon Cognito \(p. 930\)](#)
- [Loading Data From DynamoDB Into Amazon Redshift \(p. 932\)](#)
- [Processing DynamoDB Data With Apache Hive on Amazon EMR \(p. 933\)](#)

Configuring AWS Credentials in Your Files Using Amazon Cognito

The recommended way to obtain AWS credentials for your web and mobile applications is to use Amazon Cognito. Amazon Cognito helps you avoid hardcoding your AWS credentials on your files. It uses AWS Identity and Access Management (IAM) roles to generate temporary credentials for your application's authenticated and unauthenticated users.

For example, to configure your JavaScript files to use an Amazon Cognito unauthenticated role to access the Amazon DynamoDB web service, do the following.

To configure credentials to integrate with Amazon Cognito

1. Create an Amazon Cognito identity pool that allows unauthenticated identities.

```
aws cognito-identity create-identity-pool \
    --identity-pool-name DynamoPool \
    --allow-unauthenticated-identities \
    --output json
{
    "IdentityPoolId": "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
    "AllowUnauthenticatedIdentities": true,
    "IdentityPoolName": "DynamoPool"
}
```

2. Copy the following policy into a file named `myCognitoPolicy.json`. Replace the identity pool ID (`us-west-2:12345678-1ab2-123a-1234-a12345ab12`) with your own `IdentityPoolId` obtained in the previous step.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
```

```

        "Federated": "cognito-identity.amazonaws.com"
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
        "StringEquals": {
            "cognito-identity.amazonaws.com:aud": "us-west-2:12345678-1ab2-123a-1234-
a12345ab12"
        },
        "ForAnyValue:StringLike": {
            "cognito-identity.amazonaws.com:amr": "unauthenticated"
        }
    }
}
]
}

```

3. Create an IAM role that assumes the previous policy. In this way, Amazon Cognito becomes a trusted entity that can assume the `Cognito_DynamoPoolUnauth` role.

```
aws iam create-role --role-name Cognito_DynamoPoolUnauth \
--assume-role-policy-document file://PathToFile/myCognitoPolicy.json --output json
```

4. Grant the `Cognito_DynamoPoolUnauth` role full access to DynamoDB by attaching a managed policy (`AmazonDynamoDBFullAccess`).

```
aws iam attach-role-policy --policy-arn arn:aws:iam::aws:policy/
AmazonDynamoDBFullAccess \
--role-name Cognito_DynamoPoolUnauth
```

Note

Alternatively, you can grant fine-grained access to DynamoDB. For more information, see [Using IAM Policy Conditions for Fine-Grained Access Control](#).

5. Obtain and copy the IAM role Amazon Resource Name (ARN).

```
aws iam get-role --role-name Cognito_DynamoPoolUnauth --output json
```

6. Add the `Cognito_DynamoPoolUnauth` role to the `DynamoPool` identity pool. The format to specify is `KeyName=string`, where `KeyName` is `unauthenticated` and the string is the role ARN obtained in the previous step.

```
aws cognito-identity set-identity-pool-roles \
--identity-pool-id "us-west-2:12345678-1ab2-123a-1234-a12345ab12" \
--roles unauthenticated=arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth --output json
```

7. Specify the Amazon Cognito credentials in your files. Modify the `IdentityPoolId` and `RoleArn` accordingly.

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
IdentityPoolId: "us-west-2:12345678-1ab2-123a-1234-a12345ab12",
RoleArn: "arn:aws:iam::123456789012:role/Cognito_DynamoPoolUnauth"
});
```

You can now run your JavaScript programs against the DynamoDB web service using Amazon Cognito credentials. For more information, see [Setting Credentials in a Web Browser](#) in the *AWS SDK for JavaScript Getting Started Guide*.

Loading Data From DynamoDB Into Amazon Redshift

Amazon Redshift complements Amazon DynamoDB with advanced business intelligence capabilities and a powerful SQL-based interface. When you copy data from a DynamoDB table into Amazon Redshift, you can perform complex data analysis queries on that data, including joins with other tables in your Amazon Redshift cluster.

In terms of provisioned throughput, a copy operation from a DynamoDB table counts against that table's read capacity. After the data is copied, your SQL queries in Amazon Redshift do not affect DynamoDB in any way. This is because your queries act upon a copy of the data from DynamoDB, rather than upon DynamoDB itself.

Before you can load data from a DynamoDB table, you must first create an Amazon Redshift table to serve as the destination for the data. Keep in mind that you are copying data from a NoSQL environment into a SQL environment, and that there are certain rules in one environment that do not apply in the other. Here are some of the differences to consider:

- DynamoDB table names can contain up to 255 characters, including '.' (dot) and '-' (dash) characters, and are case-sensitive. Amazon Redshift table names are limited to 127 characters, cannot contain dots or dashes and are not case-sensitive. In addition, table names cannot conflict with any Amazon Redshift reserved words.
- DynamoDB does not support the SQL concept of NULL. You need to specify how Amazon Redshift interprets empty or blank attribute values in DynamoDB, treating them either as NULLs or as empty fields.
- DynamoDB data types do not correspond directly with those of Amazon Redshift. You need to ensure that each column in the Amazon Redshift table is of the correct data type and size to accommodate the data from DynamoDB.

Here is an example COPY command from Amazon Redshift SQL:

```
copy favoritemovies from 'dynamodb://my-favorite-movies-table'
credentials 'aws_access_key_id=<Your-Access-Key-ID>;aws_secret_access_key=<Your-Secret-
Access-Key>';
readratio 50;
```

In this example, the source table in DynamoDB is `my-favorite-movies-table`. The target table in Amazon Redshift is `favoritemovies`. The `readratio 50` clause regulates the percentage of provisioned throughput that is consumed; in this case, the COPY command will use no more than 50 percent of the read capacity units provisioned for `my-favorite-movies-table`. We highly recommend setting this ratio to a value less than the average unused provisioned throughput.

For detailed instructions on loading data from DynamoDB into Amazon Redshift, refer to the following sections in the [Amazon Redshift Database Developer Guide](#):

- [Loading data from a DynamoDB table](#)
- [The COPY command](#)
- [COPY examples](#)

Processing DynamoDB Data With Apache Hive on Amazon EMR

Amazon DynamoDB is integrated with Apache Hive, a data warehousing application that runs on Amazon EMR. Hive can read and write data in DynamoDB tables, allowing you to:

- Query live DynamoDB data using a SQL-like language (HiveQL).
- Copy data from a DynamoDB table to an Amazon S3 bucket, and vice-versa.
- Copy data from a DynamoDB table into Hadoop Distributed File System (HDFS), and vice-versa.
- Perform join operations on DynamoDB tables.

Topics

- [Overview \(p. 933\)](#)
- [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#)
- [Creating an External Table in Hive \(p. 940\)](#)
- [Processing HiveQL Statements \(p. 942\)](#)
- [Querying Data in DynamoDB \(p. 943\)](#)
- [Copying Data to and from Amazon DynamoDB \(p. 945\)](#)
- [Performance Tuning \(p. 956\)](#)

Overview

Amazon EMR is a service that makes it easy to quickly and cost-effectively process vast amounts of data. To use Amazon EMR, you launch a managed cluster of Amazon EC2 instances running the Hadoop open source framework. *Hadoop* is a distributed application that implements the MapReduce algorithm, where a task is mapped to multiple nodes in the cluster. Each node processes its designated work, in parallel with the other nodes. Finally, the outputs are reduced on a single node, yielding the final result.

You can choose to launch your Amazon EMR cluster so that it is persistent or transient:

- A *persistent* cluster runs until you shut it down. Persistent clusters are ideal for data analysis, data warehousing, or any other interactive use.
- A *transient* cluster runs long enough to process a job flow, and then shuts down automatically. Transient clusters are ideal for periodic processing tasks, such as running scripts.

For information about Amazon EMR architecture and administration, see the [Amazon EMR Management Guide](#).

When you launch an Amazon EMR cluster, you specify the initial number and type of Amazon EC2 instances. You also specify other distributed applications (in addition to Hadoop itself) that you want to run on the cluster. These applications include Hue, Mahout, Pig, Spark, and more.

For information about applications for Amazon EMR, see the [Amazon EMR Release Guide](#).

Depending on the cluster configuration, you might have one or more of the following node types:

- Master node — Manages the cluster, coordinating the distribution of the MapReduce executable and subsets of the raw data, to the core and task instance groups. It also tracks the status of each task performed and monitors the health of the instance groups. There is only one master node in a cluster.
- Core nodes — Runs MapReduce tasks and stores data using the Hadoop Distributed File System (HDFS).

- Task nodes (optional) — Runs MapReduce tasks.

Tutorial: Working with Amazon DynamoDB and Apache Hive

In this tutorial, you will launch an Amazon EMR cluster, and then use Apache Hive to process data stored in a DynamoDB table.

Hive is a data warehouse application for Hadoop that allows you to process and analyze data from multiple sources. Hive provides a SQL-like language, *HiveQL*, that lets you work with data stored locally in the Amazon EMR cluster or in an external data source (such as Amazon DynamoDB).

For more information, see to the [Hive Tutorial](#).

Topics

- [Before You Begin \(p. 934\)](#)
- [Step 1: Create an Amazon EC2 Key Pair \(p. 934\)](#)
- [Step 2: Launch an Amazon EMR Cluster \(p. 935\)](#)
- [Step 3: Connect to the Master Node \(p. 936\)](#)
- [Step 4: Load Data into HDFS \(p. 936\)](#)
- [Step 5: Copy Data to DynamoDB \(p. 938\)](#)
- [Step 6: Query the Data in the DynamoDB Table \(p. 939\)](#)
- [Step 7: \(Optional\) Clean Up \(p. 940\)](#)

Before You Begin

For this tutorial, you will need the following:

- An AWS account. If you do not have one, see [Signing Up for AWS \(p. 51\)](#).
- An SSH client (Secure Shell). You use the SSH client to connect to the master node of the Amazon EMR cluster and run interactive commands. SSH clients are available by default on most Linux, Unix, and Mac OS X installations. Windows users can download and install the [PuTTY](#) client, which has SSH support.

Next Step

[Step 1: Create an Amazon EC2 Key Pair \(p. 934\)](#)

Step 1: Create an Amazon EC2 Key Pair

In this step, you will create the Amazon EC2 key pair you need to connect to an Amazon EMR master node and run Hive commands.

1. Sign in to the AWS Management Console and open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. Choose a region (for example, US West (Oregon)). This should be the same region in which your DynamoDB table is located.
3. In the navigation pane, choose **Key Pairs**.
4. Choose **Create Key Pair**.
5. In **Key pair name**, type a name for your key pair (for example, `mykeypair`), and then choose **Create**.

6. Download the private key file. The file name will end with .pem (such as mykeypair.pem). Keep this private key file in a safe place. You will need it to access any Amazon EMR cluster that you launch with this key pair.

Important

If you lose the key pair, you cannot connect to the master node of your Amazon EMR cluster.

For more information about key pairs, see [Amazon EC2 Key Pairs](#) in the *Amazon EC2 User Guide for Linux Instances*.

Next Step

[Step 2: Launch an Amazon EMR Cluster \(p. 935\)](#)

Step 2: Launch an Amazon EMR Cluster

In this step, you will configure and launch an Amazon EMR cluster. Hive and a storage handler for DynamoDB will already be installed on the cluster.

1. Open the Amazon EMR console at <https://console.aws.amazon.com/elasticmapreduce/>.
 2. Choose **Create Cluster**.
 3. On the **Create Cluster - Quick Options** page, do the following:
 - a. In **Cluster name**, type a name for your cluster (for example: My EMR cluster).
 - b. In **EC2 key pair**, choose the key pair you created earlier.
- Leave the other settings at their defaults.
4. Choose **Create cluster**.

It will take several minutes to launch your cluster. You can use the **Cluster Details** page in the Amazon EMR console to monitor its progress.

When the status changes to **Waiting**, the cluster is ready for use.

Cluster Log Files and Amazon S3

An Amazon EMR cluster generates log files that contain information about the cluster status and debugging information. The default settings for **Create Cluster - Quick Options** include setting up Amazon EMR logging.

If one does not already exist, the AWS Management Console creates an Amazon S3 bucket. The bucket name is `aws-logs-account-id-region`, where *account-id* is your AWS account number and *region* is the region in which you launched the cluster (for example, `aws-logs-123456789012-us-west-2`).

Note

You can use the Amazon S3 console to view the log files. For more information, see [View Log Files](#) in the *Amazon EMR Management Guide*.

You can use this bucket for purposes in addition to logging. For example, you can use the bucket as a location for storing a Hive script or as a destination when exporting data from Amazon DynamoDB to Amazon S3.

Next Step

[Step 3: Connect to the Master Node \(p. 936\)](#)

Step 3: Connect to the Master Node

When the status of your Amazon EMR cluster changes to `Waiting`, you will be able to connect to the master node using SSH and perform command line operations.

1. In the Amazon EMR console, choose your cluster's name to view its status.
2. On the **Cluster Details** page, find the **Master public DNS** field. This is the public DNS name for the master node of your Amazon EMR cluster.
3. To the right of the DNS name, choose the **SSH link**.
4. Follow the instructions in [Connect to the Master Node Using SSH](#).

Depending on your operating system, choose the **Windows** tab or the **Mac/Linux** tab, and follow the instructions for connecting to the master node.

After you connect to the master node using either SSH or PuTTY, you should see a command prompt similar to the following:

```
[hadoop@ip-192-0-2-0 ~]$
```

Next Step

[Step 4: Load Data into HDFS \(p. 936\)](#)

Step 4: Load Data into HDFS

In this step, you will copy a data file into Hadoop Distributed File System (HDFS), and then create an external Hive table that maps to the data file.

Download the Sample Data

1. Download the sample data archive (`features.zip`):

```
wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/features.zip
```

2. Extract the `features.txt` file from the archive:

```
unzip features.zip
```

3. View the first few lines of the `features.txt` file:

```
head features.txt
```

The result should look similar to this:

```
1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794  
875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7
```

```
1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10  
26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681  
1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605  
1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558  
1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024  
533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0  
829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671  
541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
```

The data file contains a subset of data provided by the United States Board on Geographic Names (http://geonames.usgs.gov/domestic/download_data.htm).

The `features.txt` file contains a subset of data from the United States Board on Geographic Names (http://geonames.usgs.gov/domestic/download_data.htm). The fields in each line represent the following:

- Feature ID (unique identifier)
- Name
- Class (lake; forest; stream; and so on)
- State
- Latitude (degrees)
- Longitude (degrees)
- Height (in feet)

4. At the command prompt, enter the following command:

```
hive
```

The command prompt changes to this: `hive>`

5. Enter the following HiveQL statement to create a native Hive table:

```
CREATE TABLE hive_features  
(feature_id          BIGINT,  
 feature_name        STRING ,  
 feature_class       STRING ,  
 state_alpha         STRING,  
 prim_lat_dec       DOUBLE ,  
 prim_long_dec      DOUBLE ,  
 elev_in_ft          BIGINT)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '|'  
LINES TERMINATED BY '\n';
```

6. Enter the following HiveQL statement to load the table with data:

```
LOAD DATA  
LOCAL  
INPATH './features.txt'  
OVERWRITE  
INTO TABLE hive_features;
```

7. You now have a native Hive table populated with data from the `features.txt` file. To verify, enter the following HiveQL statement:

```
SELECT state_alpha, COUNT(*)
FROM hive_features
GROUP BY state_alpha;
```

The output should show a list of states and the number of geographic features in each.

Next Step

[Step 5: Copy Data to DynamoDB \(p. 938\)](#)

Step 5: Copy Data to DynamoDB

In this step, you will copy data from the Hive table (`hive_features`) to a new table in DynamoDB.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table**.
3. On the **Create DynamoDB table** page, do the following:
 - a. In **Table**, type **Features**.
 - b. For **Primary key**, in the **Partition key** field, type **Id**. Set the data type to **Number**.

Clear **Use Default Settings**. For **Provisioned Capacity**, type the following:

- **Read Capacity Units**—10
- **Write Capacity Units**—10

Choose **Create**.

4. At the Hive prompt, enter the following HiveQL statement:

```
CREATE EXTERNAL TABLE ddb_features
  (feature_id      BIGINT,
   feature_name    STRING,
   feature_class   STRING,
   state_alpha     STRING,
   prim_lat_dec   DOUBLE,
   prim_long_dec  DOUBLE,
   elev_in_ft      BIGINT)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES(
  "dynamodb.table.name" = "Features",
  "dynamodb.column.mapping"="feature_id:Id,feature_name:Name,feature_class:Class,state_alpha:State,p");

```

You have now established a mapping between Hive and the Features table in DynamoDB.

5. Enter the following HiveQL statement to import data to DynamoDB:

```
INSERT OVERWRITE TABLE ddb_features
SELECT
  feature_id,
```

```
feature_name,  
feature_class,  
state_alpha,  
prim_lat_dec,  
prim_long_dec,  
elev_in_ft  
FROM hive_features;
```

Hive will submit a MapReduce job, which will be processed by your Amazon EMR cluster. It will take several minutes to complete the job.

6. Verify that the data has been loaded into DynamoDB:
 - a. In the DynamoDB console navigation pane, choose **Tables**.
 - b. Choose the Features table, and then choose the **Items** tab to view the data.

Next Step

[Step 6: Query the Data in the DynamoDB Table \(p. 939\)](#)

Step 6: Query the Data in the DynamoDB Table

In this step, you will use HiveQL to query the Features table in DynamoDB. Try the following Hive queries:

1. All of the feature types (`feature_class`) in alphabetical order:

```
SELECT DISTINCT feature_class  
FROM ddb_features  
ORDER BY feature_class;
```

2. All of the lakes that begin with the letter "M":

```
SELECT feature_name, state_alpha  
FROM ddb_features  
WHERE feature_class = 'Lake'  
AND feature_name LIKE 'M%'  
ORDER BY feature_name;
```

3. States with at least three features higher than a mile (5,280 feet):

```
SELECT state_alpha, feature_class, COUNT(*)  
FROM ddb_features  
WHERE elev_in_ft > 5280  
GROUP by state_alpha, feature_class  
HAVING COUNT(*) >= 3  
ORDER BY state_alpha, feature_class;
```

Next Step

[Step 7: \(Optional\) Clean Up \(p. 940\)](#)

Step 7: (Optional) Clean Up

Now that you have completed the tutorial, you can continue reading this section to learn more about working with DynamoDB data in Amazon EMR. You might decide to keep your Amazon EMR cluster up and running while you do this.

If you don't need the cluster anymore, you should terminate it and remove any associated resources. This will help you avoid being charged for resources you don't need.

1. Terminate the Amazon EMR cluster:
 - a. Open the Amazon EMR console at <https://console.aws.amazon.com/elasticmapreduce/>.
 - b. Choose the Amazon EMR cluster, choose **Terminate**, and then confirm.
2. Delete the Features table in DynamoDB:
 - a. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
 - b. In the navigation pane, choose **Tables**.
 - c. Choose the Features table. From the **Actions** menu, choose **Delete Table**.
3. Delete the Amazon S3 bucket containing the Amazon EMR log files:
 - a. Open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
 - b. From the list of buckets, choose `aws-logs- accountID-region`, where `accountID` is your AWS account number and `region` is the region in which you launched the cluster.
 - c. From the **Action** menu, choose **Delete**.

Creating an External Table in Hive

In [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#), you created an external Hive table that mapped to a DynamoDB table. When you issued HiveQL statements against the external table, the read and write operations were passed through to the DynamoDB table.

You can think of an external table as a pointer to a data source that is managed and stored elsewhere. In this case, the underlying data source is a DynamoDB table. (The table must already exist. You cannot create, update, or delete a DynamoDB table from within Hive.) You use the `CREATE EXTERNAL TABLE` statement to create the external table. After that, you can use HiveQL to work with data in DynamoDB, as if that data were stored locally within Hive.

Note

You can use `INSERT` statements to insert data into an external table and `SELECT` statements to select data from it. However, you cannot use `UPDATE` or `DELETE` statements to manipulate data in the table.

If you no longer need the external table, you can remove it using the `DROP TABLE` statement. In this case, `DROP TABLE` only removes the external table in Hive. It does not affect the underlying DynamoDB table or any of its data.

Topics

- [CREATE EXTERNAL TABLE Syntax \(p. 940\)](#)
- [Data Type Mappings \(p. 941\)](#)

CREATE EXTERNAL TABLE Syntax

The following shows the HiveQL syntax for creating an external Hive table that maps to a DynamoDB table:

```

CREATE EXTERNAL TABLE hive_table
  (hive_column1_name hive_column1_datatype, hive_column2_name hive_column2_datatype...)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES (
    "dynamodb.table.name" = "dynamodb_table",
    "dynamodb.column.mapping" =
    "hive_column1_name:dynamodb_attribute1_name,hive_column2_name:dynamodb_attribute2_name..."
);

```

Line 1 is the start of the `CREATE EXTERNAL TABLE` statement, where you provide the name of the Hive table (*hive_table*) you want to create.

Line 2 specifies the columns and data types for *hive_table*. You need to define columns and data types that correspond to the attributes in the DynamoDB table.

Line 3 is the `STORED BY` clause, where you specify a class that handles data management between the Hive and the DynamoDB table. For DynamoDB, `STORED BY` should be set to `'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'`.

Line 4 is the start of the `TBLPROPERTIES` clause, where you define the following parameters for `DynamoDBStorageHandler`:

- `dynamodb.table.name`—the name of the DynamoDB table.
- `dynamodb.column.mapping`—pairs of column names in the Hive table and their corresponding attributes in the DynamoDB table. Each pair is of the form `hive_column_name:dynamodb_attribute_name`, and the pairs are separated by commas.

Note the following:

- The name of the Hive table name does not have to be the same as the DynamoDB table name.
- The Hive table column names do not have to be the same as those in the DynamoDB table.
- The table specified by `dynamodb.table.name` must exist in DynamoDB.
- For `dynamodb.column.mapping`:
 - You must map the key schema attributes for the DynamoDB table. This includes the partition key and the sort key (if present).
 - You do not have to map the non-key attributes of the DynamoDB table. However, you will not see any data from those attributes when you query the Hive table.
 - If the data types of a Hive table column and a DynamoDB attribute are incompatible, you will see `NULL` in these columns when you query the Hive table.

Note

The `CREATE EXTERNAL TABLE` statement does not perform any validation on the `TBLPROPERTIES` clause. The values you provide for `dynamodb.table.name` and `dynamodb.column.mapping` are only evaluated by the `DynamoDBStorageHandler` class when you attempt to access the table.

Data Type Mappings

The following table shows DynamoDB data types and compatible Hive data types:

| DynamoDB Data Type | Hive Data Type |
|--------------------|----------------|
| String | STRING |

| DynamoDB Data Type | Hive Data Type |
|--------------------|--------------------------------|
| Number | BIGINT or DOUBLE |
| Binary | BINARY |
| String Set | ARRAY<STRING> |
| Number Set | ARRAY<BIGINT> or ARRAY<DOUBLE> |
| Binary Set | ARRAY<BINARY> |

Note

The following DynamoDB data types are not supported by the `DynamoDBStorageHandler` class, so they cannot be used with `dynamodb.column.mapping`:

- Map
- List
- Boolean
- Null

If you want to map a DynamoDB attribute of type Number, you must choose an appropriate Hive type:

- The Hive `BIGINT` type is for 8-byte signed integers. It is the same as the `long` data type in Java.
- The Hive `DOUBLE` type is for 8-bit double precision floating point numbers. It is the same as the `double` type in Java.

If you have numeric data stored in DynamoDB that has a higher precision than the Hive data type you choose, then accessing the DynamoDB data could cause a loss of precision.

If you export data of type Binary from DynamoDB to (Amazon S3) or HDFS, the data is stored as a Base64-encoded string. If you import data from Amazon S3 or HDFS into the DynamoDB Binary type, you must ensure the data is encoded as a Base64 string.

Processing HiveQL Statements

Hive is an application that runs on Hadoop, which is a batch-oriented framework for running MapReduce jobs. When you issue a HiveQL statement, Hive determines whether it can return the results immediately or whether it must submit a MapReduce job.

For example, consider the `ddb_features` table (from [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#)). The following Hive query prints state abbreviations and the number of summits in each:

```
SELECT state_alpha, count(*)
FROM ddb_features
WHERE feature_class = 'Summit'
GROUP BY state_alpha;
```

Hive does not return the results immediately. Instead, it submits a MapReduce job, which is processed by the Hadoop framework. Hive will wait until the job is complete before it shows the results from the query:

```
AK 2
AL 2
AR 2
AZ 3
CA 7
CO 2
CT 2
ID 1
KS 1
ME 2
MI 1
MT 3
NC 1
NE 1
NM 1
NY 2
OR 5
PA 1
TN 1
TX 1
UT 4
VA 1
VT 2
WA 2
WY 3
Time taken: 8.753 seconds, Fetched: 25 row(s)
```

Monitoring and Canceling Jobs

When Hive launches a Hadoop job, it prints output from that job. The job completion status is updated as the job progresses. In some cases, the status might not be updated for a long time. (This can happen when you are querying a large DynamoDB table that has a low provisioned read capacity setting.)

If you need to cancel the job before it is complete, you can type **Ctrl+C** at any time.

Querying Data in DynamoDB

The following examples show some ways that you can use HiveQL to query data stored in DynamoDB.

These examples refer to the *ddb_features* table in the tutorial ([Step 5: Copy Data to DynamoDB \(p. 938\)](#)).

Topics

- [Using Aggregate Functions \(p. 943\)](#)
- [Using the GROUP BY and HAVING Clauses \(p. 944\)](#)
- [Joining Two DynamoDB tables \(p. 944\)](#)
- [Joining Tables from Different Sources \(p. 945\)](#)

Using Aggregate Functions

HiveQL provides built-in functions for summarizing data values. For example, you can use the `MAX` function to find the largest value for a selected column. The following example returns the elevation of the highest feature in the state of Colorado.

```
SELECT MAX(elev_in_ft)
FROM ddb_features
WHERE state_alpha = 'CO';
```

Using the GROUP BY and HAVING Clauses

You can use the `GROUP BY` clause to collect data across multiple records. This is often used with an aggregate function such as `SUM`, `COUNT`, `MIN`, or `MAX`. You can also use the `HAVING` clause to discard any results that do not meet certain criteria.

The following example returns a list of the highest elevations from states that have more than five features in the `ddb_features` table.

```
SELECT state_alpha, max(elev_in_ft)
FROM ddb_features
GROUP BY state_alpha
HAVING count(*) >= 5;
```

Joining Two DynamoDB tables

The following example maps another Hive table (`east_coast_states`) to a table in DynamoDB. The `SELECT` statement is a join across these two tables. The join is computed on the cluster and returned. The join does not take place in DynamoDB.

Consider a DynamoDB table named `EastCoastStates` that contains the following data:

| StateName | StateAbbrev |
|----------------|-------------|
| Maine | ME |
| New Hampshire | NH |
| Massachusetts | MA |
| Rhode Island | RI |
| Connecticut | CT |
| New York | NY |
| New Jersey | NJ |
| Delaware | DE |
| Maryland | MD |
| Virginia | VA |
| North Carolina | NC |
| South Carolina | SC |
| Georgia | GA |
| Florida | FL |

Let's assume the table is available as a Hive external table named `east_coast_states`:

```
CREATE EXTERNAL TABLE ddb_east_coast_states (state_name STRING, state_alpha STRING)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "EastCoastStates",
"YNAMODB.column.mapping" = "state_name:StateName,state_alpha:StateAbbrev");
```

The following join returns the states on the East Coast of the United States that have at least three features:

```
SELECT ecs.state_name, f.feature_class, COUNT(*)
```

```
FROM ddb_east_coast_states ecs
JOIN ddb_features f on ecs.state_alpha = f.state_alpha
GROUP BY ecs.state_name, f.feature_class
HAVING COUNT(*) >= 3;
```

Joining Tables from Different Sources

In the following example, `s3_east_coast_states` is a Hive table associated with a CSV file stored in Amazon S3. The `ddb_features` table is associated with data in DynamoDB. The following example joins these two tables, returning the geographic features from states whose names begin with "New."

```
create external table s3_east_coast_states (state_name STRING, state_alpha STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION 's3://bucketname/path/subpath/';
```

```
SELECT ecs.state_name, f.feature_name, f.feature_class
FROM s3_east_coast_states ecs
JOIN ddb_features f
ON ecs.state_alpha = f.state_alpha
WHERE ecs.state_name LIKE 'New%';
```

Copying Data to and from Amazon DynamoDB

In the [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#), you copied data from a native Hive table into an external DynamoDB table, and then queried the external DynamoDB table. The table is external because it exists outside of Hive. Even if you drop the Hive table that maps to it, the table in DynamoDB is not affected.

Hive is an excellent solution for copying data among DynamoDB tables, Amazon S3 buckets, native Hive tables, and Hadoop Distributed File System (HDFS). This section provides examples of these operations.

Topics

- [Copying Data Between DynamoDB and a Native Hive Table \(p. 945\)](#)
- [Copying Data Between DynamoDB and Amazon S3 \(p. 946\)](#)
- [Copying Data Between DynamoDB and HDFS \(p. 950\)](#)
- [Using Data Compression \(p. 955\)](#)
- [Reading Non-Printable UTF-8 Character Data \(p. 955\)](#)

Copying Data Between DynamoDB and a Native Hive Table

If you have data in a DynamoDB table, you can copy the data to a native Hive table. This will give you a snapshot of the data, as of the time you copied it.

You might decide to do this if you need to perform many HiveQL queries, but do not want to consume provisioned throughput capacity from DynamoDB. Because the data in the native Hive table is a copy of the data from DynamoDB, and not "live" data, your queries should not expect that the data is up-to-date.

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#) and have an external table that is mastered in DynamoDB (`ddb_features`).

Example From DynamoDB to Native Hive Table

You can create a native Hive table and populate it with data from *ddb_features*, like this:

```
CREATE TABLE features_snapshot AS
SELECT * FROM ddb_features;
```

You can then refresh the data at any time:

```
INSERT OVERWRITE TABLE features_snapshot
SELECT * FROM ddb_features;
```

In these examples, the subquery `SELECT * FROM ddb_features` will retrieve all of the data from *ddb_features*. If you only want to copy a subset of the data, you can use a `WHERE` clause in the subquery.

The following example creates a native Hive table, containing only some of the attributes for lakes and summits:

```
CREATE TABLE lakes_and_summits AS
SELECT feature_name, feature_class, state_alpha
FROM ddb_features
WHERE feature_class IN ('Lake','Summit');
```

Example From Native Hive Table to DynamoDB

Use the following HiveQL statement to copy the data from the native Hive table to *ddb_features*:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM features_snapshot;
```

Copying Data Between DynamoDB and Amazon S3

If you have data in a DynamoDB table, you can use Hive to copy the data to an Amazon S3 bucket.

You might do this if you want to create an archive of data in your DynamoDB table. For example, suppose you have a test environment where you need to work with a baseline set of test data in DynamoDB. You can copy the baseline data to an Amazon S3 bucket, and then run your tests. Afterward, you can reset the test environment by restoring the baseline data from the Amazon S3 bucket to DynamoDB.

If you worked through [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#), then you already have an Amazon S3 bucket that contains your Amazon EMR logs. You can use this bucket for the examples in this section, if you know the root path for the bucket:

1. Open the Amazon EMR console at <https://console.aws.amazon.com/elasticmapreduce/>.
2. For **Name**, choose your cluster.
3. The URI is listed in **Log URI** under **Configuration Details**.

4. Make a note of the root path of the bucket. The naming convention is:

s3://aws-logs-*accountID-region*

where *accountID* is your AWS account ID and region is the AWS region for the bucket.

Note

For these examples, we will use a subpath within the bucket, as in this example:

s3://aws-logs-123456789012-us-west-2/hive-test

The following procedures are written with the assumption you followed the steps in the tutorial and have an external table that is mastered in DynamoDB (*ddb_features*).

Topics

- [Copying Data Using the Hive Default Format \(p. 947\)](#)
- [Copying Data with a User-Specified Format \(p. 948\)](#)
- [Copying Data Without a Column Mapping \(p. 949\)](#)
- [Viewing the Data in Amazon S3 \(p. 950\)](#)

Copying Data Using the Hive Default Format

Example From DynamoDB to Amazon S3

Use an `INSERT OVERWRITE DIRECTORY` statement to write directly to Amazon S3.

```
INSERT OVERWRITE DIRECTORY 's3://aws-logs-123456789012-us-west-2/hive-test'  
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135  
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260  
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133  
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900  
115905^AChacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as ^A.

Example From Amazon S3 to DynamoDB

1. Create an external table pointing to the unformatted data in Amazon S3.

```
CREATE EXTERNAL TABLE s3_features_unformatted  
(feature_id      BIGINT,  
 feature_name    STRING ,  
 feature_class   STRING ,  
 state_alpha     STRING,  
 prim_lat_dec   DOUBLE ,  
 prim_long_dec  DOUBLE ,  
 elev_in_ft      BIGINT)
```

```
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data to DynamoDB.

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM s3_features_unformatted;
```

Copying Data with a User-Specified Format

If you want to specify your own field separator character, you can create an external table that maps to the Amazon S3 bucket. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to Amazon S3

1. Create a Hive external table that maps to Amazon S3. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

```
CREATE EXTERNAL TABLE s3_features_csv
(feature_id      BIGINT,
feature_name    STRING,
feature_class   STRING,
state_alpha     STRING,
prim_lat_dec   DOUBLE,
prim_long_dec  DOUBLE,
elev_in_ft     BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE s3_features_csv
SELECT * FROM ddb_features;
```

The data file in Amazon S3 looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM s3_features_csv;
```

Copying Data Without a Column Mapping

You can copy data from DynamoDB in a raw format and write it to Amazon S3 without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in Amazon S3.

Note

If your DynamoDB table contains attributes of type Map, List, Boolean or Null, then this is the only way you can use Hive to copy data from DynamoDB to Amazon S3.

Example From DynamoDB to Amazon S3

1. Create an external table associated with your DynamoDB table. (There is no dynamodb.column.mapping in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping
(item MAP<STRING, STRING>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your Amazon S3 bucket.

```
CREATE EXTERNAL TABLE s3_features_no_mapping
(item MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 's3://aws-logs-123456789012-us-west-2/hive-test';
```

3. Copy the data from DynamoDB to Amazon S3.

```
INSERT OVERWRITE TABLE s3_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in Amazon S3 looks like this:

```
Name^C{"s":"Soldiers Farewell
Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":6135}^BLatitude^C{"n":32.3564729}
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BClass^C{"s":Stream"}^BElevation^C{"n":1260}^BLatitude^C{"n":41.2120086}
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":CA"}^BClass^C{"s":Summit"}^BElevation^C{"n":8133}^BLatitude^C{"n":37.7229821}
Name^C{"s":"Neversweet
Gulch"}^BState^C{"s":CA"}^BClass^C{"s":Valley"}^BElevation^C{"n":2900}^BLatitude^C{"n":41.6565269}
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":AL"}^BClass^C{"s":Bay"}^BElevation^C{"n":0}^BLatitude^C{"n":30.6979676}^BId^C
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as ^B and ETX appears as ^C.

Example From Amazon S3 to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from Amazon S3:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM s3_features_no_mapping;
```

Viewing the Data in Amazon S3

If you use SSH to connect to the master node, you can use the AWS Command Line Interface (AWS CLI) to access the data that Hive wrote to Amazon S3.

The following steps are written with the assumption you have copied data from DynamoDB to Amazon S3 using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

```
hive> exit;
```

2. List the contents of the `hive-test` directory in your Amazon S3 bucket. (This is where Hive copied the data from DynamoDB.)

```
aws s3 ls s3://aws-logs-123456789012-us-west-2/hive-test/
```

The response should look similar to this:

```
2016-11-01 23:19:54 81983 000000_0
```

The file name (`000000_0`) is system-generated.

3. (Optional) You can copy the data file from Amazon S3 to the local file system on the master node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

```
aws s3 cp s3://aws-logs-123456789012-us-west-2/hive-test/000000_0 .
```

The response should look similar to this:

```
download: s3://aws-logs-123456789012-us-west-2/hive-test/000000_0
to ./000000_0
```

Note

The local file system on the master node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Copying Data Between DynamoDB and HDFS

If you have data in a DynamoDB table, you can use Hive to copy the data to the Hadoop Distributed File System (HDFS).

You might do this if you are running a MapReduce job that requires data from DynamoDB. If you copy the data from DynamoDB into HDFS, Hadoop can process it, using all of the available nodes in the Amazon EMR cluster in parallel. When the MapReduce job is complete, you can then write the results from HDFS to DDB.

In the following examples, Hive will read from and write to the following HDFS directory: /user/hadoop/hive-test

The examples in this section are written with the assumption you followed the steps in [Tutorial: Working with Amazon DynamoDB and Apache Hive \(p. 934\)](#) and you have an external table that is mastered in DynamoDB (`ddb_features`).

Topics

- [Copying Data Using the Hive Default Format \(p. 951\)](#)
- [Copying Data with a User-Specified Format \(p. 952\)](#)
- [Copying Data Without a Column Mapping \(p. 953\)](#)
- [Accessing the Data in HDFS \(p. 954\)](#)

Copying Data Using the Hive Default Format

Example From DynamoDB to HDFS

Use an `INSERT OVERWRITE` statement to write directly to HDFS.

```
INSERT OVERWRITE DIRECTORY 'hdfs:///user/hadoop/hive-test'  
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709^ASoldiers Farewell Hill^ASummit^ANM^A32.3564729^A-108.33004616135  
1178153^AJones Run^AStream^APA^A41.2120086^A-79.25920781260  
253838^ASentinel Dome^ASummit^ACA^A37.7229821^A-119.584338133  
264054^ANeversweet Gulch^AValley^ACA^A41.6565269^A-122.83614322900  
115905^ACHacaloochee Bay^ABay^AAL^A30.6979676^A-87.97388530
```

Each field is separated by an SOH character (start of heading, 0x01). In the file, SOH appears as ^A.

Example From HDFS to DynamoDB

1. Create an external table that maps to the unformatted data in HDFS.

```
CREATE EXTERNAL TABLE hdfs_features_unformatted  
(feature_id      BIGINT,  
 feature_name    STRING ,  
 feature_class   STRING ,  
 state_alpha     STRING,  
 prim_lat_dec   DOUBLE ,  
 prim_long_dec  DOUBLE ,  
 elev_in_ft      BIGINT)  
LOCATION 'hdfs:///user/hadoop/hive-test';
```

2. Copy the data to DynamoDB.

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_unformatted;
```

Copying Data with a User-Specified Format

If you want to use a different field separator character, you can create an external table that maps to the HDFS directory. You might use this technique for creating data files with comma-separated values (CSV).

Example From DynamoDB to HDFS

1. Create a Hive external table that maps to HDFS. When you do this, ensure that the data types are consistent with those of the DynamoDB external table.

```
CREATE EXTERNAL TABLE hdfs_features_csv
(feature_id      BIGINT,
feature_name    STRING ,
feature_class   STRING ,
state_alpha     STRING,
prim_lat_dec   DOUBLE ,
prim_long_dec  DOUBLE ,
elev_in_ft     BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'hdfs:///user/hadoop/hive-test';
```

2. Copy the data from DynamoDB.

```
INSERT OVERWRITE TABLE hdfs_features_csv
SELECT * FROM ddb_features;
```

The data file in HDFS looks like this:

```
920709,Soldiers Farewell Hill,Summit,NM,32.3564729,-108.3300461,6135
1178153,Jones Run,Stream,PA,41.2120086,-79.2592078,1260
253838,Sentinel Dome,Summit,CA,37.7229821,-119.58433,8133
264054,Neversweet Gulch,Valley,CA,41.6565269,-122.8361432,2900
115905,Chacaloochee Bay,Bay,AL,30.6979676,-87.9738853,0
```

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features
SELECT * FROM hdfs_features_csv;
```

Copying Data Without a Column Mapping

You can copy data from DynamoDB in a raw format and write it to HDFS without specifying any data types or column mapping. You can use this method to create an archive of DynamoDB data and store it in HDFS.

Note

If your DynamoDB table contains attributes of type Map, List, Boolean or Null, then this is the only way you can use Hive to copy data from DynamoDB to HDFS.

Example From DynamoDB to HDFS

1. Create an external table associated with your DynamoDB table. (There is no dynamodb.column.mapping in this HiveQL statement.)

```
CREATE EXTERNAL TABLE ddb_features_no_mapping
  (item MAP<STRING, STRING>)
STORED BY 'org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler'
TBLPROPERTIES ("dynamodb.table.name" = "Features");
```

2. Create another external table associated with your HDFS directory.

```
CREATE EXTERNAL TABLE hdfs_features_no_mapping
  (item MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
LOCATION 'hdfs:///user/hadoop/hive-test';
```

3. Copy the data from DynamoDB to HDFS.

```
INSERT OVERWRITE TABLE hdfs_features_no_mapping
SELECT * FROM ddb_features_no_mapping;
```

The data file in HDFS looks like this:

```
Name^C{"s":"Soldiers Farewell
Hill"}^BState^C{"s":"NM"}^BClass^C{"s":"Summit"}^BElevation^C{"n":6135}^BLatitude^C{"n":32.3564729}
Name^C{"s":"Jones
Run"}^BState^C{"s":"PA"}^BClass^C{"s":"Stream"}^BElevation^C{"n":1260}^BLatitude^C{"n":41.2120086}
Name^C{"s":"Sentinel
Dome"}^BState^C{"s":"CA"}^BClass^C{"s":"Summit"}^BElevation^C{"n":8133}^BLatitude^C{"n":37.7229821}
Name^C{"s":"Neversweet
Gulch"}^BState^C{"s":"CA"}^BClass^C{"s":"Valley"}^BElevation^C{"n":2900}^BLatitude^C{"n":41.6565269}
Name^C{"s":"Chacaloochee
Bay"}^BState^C{"s":AL}^BClass^C{"s":Bay}^BElevation^C{"n":0}^BLatitude^C{"n":30.6979676}^BId^C
```

Each field begins with an STX character (start of text, 0x02) and ends with an ETX character (end of text, 0x03). In the file, STX appears as ^B and ETX appears as ^C.

Example From HDFS to DynamoDB

With a single HiveQL statement, you can populate the DynamoDB table using the data from HDFS:

```
INSERT OVERWRITE TABLE ddb_features_no_mapping
SELECT * FROM hdfs_features_no_mapping;
```

Accessing the Data in HDFS

HDFS is a distributed file system, accessible to all of the nodes in the Amazon EMR cluster. If you use SSH to connect to the master node, you can use command line tools to access the data that Hive wrote to HDFS.

HDFS is not the same thing as the local file system on the master node. You cannot work with files and directories in HDFS using standard Linux commands (such as `cat`, `cp`, `mv`, or `rm`). Instead, you perform these tasks using the `hadoop fs` command.

The following steps are written with the assumption you have copied data from DynamoDB to HDFS using one of the procedures in this section.

1. If you are currently at the Hive command prompt, exit to the Linux command prompt.

```
hive> exit;
```

2. List the contents of the `/user/hadoop/hive-test` directory in HDFS. (This is where Hive copied the data from DynamoDB.)

```
hadoop fs -ls /user/hadoop/hive-test
```

The response should look similar to this:

```
Found 1 items
-rw-r--r-- 1 hadoop hadoop 29504 2016-06-08 23:40 /user/hadoop/hive-test/000000_0
```

The file name (`000000_0`) is system-generated.

3. View the contents of the file:

```
hadoop fs -cat /user/hadoop/hive-test/000000_0
```

Note

In this example, the file is relatively small (approximately 29 KB). Be careful when you use this command with files that are very large or contain non-printable characters.

4. (Optional) You can copy the data file from HDFS to the local file system on the master node. After you do this, you can use standard Linux command line utilities to work with the data in the file.

```
hadoop fs -get /user/hadoop/hive-test/000000_0
```

This command will not overwrite the file.

Note

The local file system on the master node has limited capacity. Do not use this command with files that are larger than the available space in the local file system.

Using Data Compression

When you use Hive to copy data among different data sources, you can request on-the-fly data compression. Hive provides several compression codecs. You can choose one during your Hive session. When you do this, the data is compressed in the specified format.

The following example compresses data using the Lempel-Ziv-Oberhumer (LZO) algorithm.

```
SET hive.exec.compress.output=true;
SET io.seqfile.compression.type=BLOCK;
SET mapred.output.compression.codec = com.hadoop.compression.lzo.LzopCodec;

CREATE EXTERNAL TABLE lzo_compression_table (line STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE lzo_compression_table SELECT *
FROM hiveTableName;
```

The resulting file in Amazon S3 will have a system-generated name with .lzo at the end (for example, 8d436957-57ba-4af7-840c-96c2fc7bb6f5-000000.lzo).

The available compression codecs are:

- org.apache.hadoop.io.compress.GzipCodec
- org.apache.hadoop.io.compress.DefaultCodec
- com.hadoop.compression.lzo.LzoCodec
- com.hadoop.compression.lzo.LzopCodec
- org.apache.hadoop.io.compress.BZip2Codec
- org.apache.hadoop.io.compress.SnappyCodec

Reading Non-Printable UTF-8 Character Data

To read and write non-printable UTF-8 character data, you can use the STORED AS SEQUENCEFILE clause when you create a Hive table. A SequenceFile is a Hadoop binary file format. You need to use Hadoop to read this file. The following example shows how to export data from DynamoDB into Amazon S3. You can use this functionality to handle non-printable UTF-8 encoded characters.

```
CREATE EXTERNAL TABLE s3_export(a_col string, b_col bigint, c_col array<string>)
STORED AS SEQUENCEFILE
LOCATION 's3://bucketname/path/subpath/';

INSERT OVERWRITE TABLE s3_export SELECT *
FROM hiveTableName;
```

Performance Tuning

When you create a Hive external table that maps to a DynamoDB table, you do not consume any read or write capacity from DynamoDB. However, read and write activity on the Hive table (such as `INSERT` or `SELECT`) translates directly into read and write operations on the underlying DynamoDB table.

Apache Hive on Amazon EMR implements its own logic for balancing the I/O load on the DynamoDB table and seeks to minimize the possibility of exceeding the table's provisioned throughput. At the end of each Hive query, Amazon EMR returns runtime metrics, including the number of times your provisioned throughput was exceeded. You can use this information, together with CloudWatch metrics on your DynamoDB table, to improve performance in subsequent requests.

The Amazon EMR console provides basic monitoring tools for your cluster. For more information, see [View and Monitor a Cluster](#) in the *Amazon EMR Management Guide*.

You can also monitor your cluster and Hadoop jobs using web-based tools, such as Hue, Ganglia, and the Hadoop web interface. For more information, see [View Web Interfaces Hosted on Amazon EMR Clusters](#) in the *Amazon EMR Management Guide*.

This section describes steps you can take to performance-tune Hive operations on external DynamoDB tables.

Topics

- [DynamoDB Provisioned Throughput \(p. 956\)](#)
- [Adjusting the Mappers \(p. 958\)](#)
- [Additional Topics \(p. 959\)](#)

DynamoDB Provisioned Throughput

When you issue HiveQL statements against the external DynamoDB table, the `DynamoDBStorageHandler` class makes the appropriate low-level DynamoDB API requests, which consume provisioned throughput. If there is not enough read or write capacity on the DynamoDB table, the request will be throttled, resulting in slow HiveQL performance. For this reason, you should ensure that the table has enough throughput capacity.

For example, suppose that you have provisioned 100 read capacity units for your DynamoDB table. This will let you read 409,600 bytes per second (100 × 4 KB read capacity unit size). Now suppose that the table contains 20 GB of data (21,474,836,480 bytes) and you want to use the `SELECT` statement to select all of the data using HiveQL. You can estimate how long the query will take to run like this:

$$21,474,836,480 / 409,600 = 52,429 \text{ seconds} = 14.56 \text{ hours}$$

In this scenario, the DynamoDB table is a bottleneck. It won't help to add more Amazon EMR nodes, because the Hive throughput is constrained to only 409,600 bytes per second. The only way to decrease the time required for the `SELECT` statement is to increase the provisioned read capacity of the DynamoDB table.

You can perform a similar calculation to estimate how long it would take to bulk-load data into a Hive external table mapped to a DynamoDB table. Determine the total number of bytes in the data you want to load, and then divide it by the size of one DynamoDB write capacity unit (1 KB). This will yield the number of seconds it will take to load the table.

You should regularly monitor the CloudWatch metrics for your table. For a quick overview in the DynamoDB console, choose your table and then choose the **Metrics** tab. From here, you can view read and write capacity units consumed and read and write requests that have been throttled.

Read Capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of ProvisionedThroughputExceeded messages in the job output, you can adjust the default read rate. To do this, you can modify the dynamodb.throughput.read.percent configuration variable. You can use the SET command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.read.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, dynamodb.throughput.read.percent will return to its default value.

The value of dynamodb.throughput.read.percent can be between 0.1 and 1.5, inclusively. 0.5 represents the default read rate, meaning that Hive will attempt to consume half of the read capacity of the table. If you increase the value above 0.5, Hive will increase the request rate; decreasing the value below 0.5 decreases the read request rate. (The actual read rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned read capacity of the table, or if your read requests are being throttled too much, try reducing dynamodb.throughput.read.percent below 0.5. If you have sufficient read capacity in the table and want more responsive HiveQL operations, you can set the value above 0.5.

Write Capacity

Amazon EMR manages the request load against your DynamoDB table, according to the table's provisioned throughput settings. However, if you notice a large number of ProvisionedThroughputExceeded messages in the job output, you can adjust the default write rate. To do this, you can modify the dynamodb.throughput.write.percent configuration variable. You can use the SET command to set this variable at the Hive command prompt:

```
SET dynamodb.throughput.write.percent=1.0;
```

This variable persists for the current Hive session only. If you exit Hive and return to it later, dynamodb.throughput.write.percent will return to its default value.

The value of dynamodb.throughput.write.percent can be between 0.1 and 1.5, inclusively. 0.5 represents the default write rate, meaning that Hive will attempt to consume half of the write capacity of the table. If you increase the value above 0.5, Hive will increase the request rate; decreasing the value below 0.5 decreases the write request rate. (The actual write rate will vary, depending on factors such as whether there is a uniform key distribution in the DynamoDB table.)

If you notice that Hive is frequently depleting the provisioned write capacity of the table, or if your write requests are being throttled too much, try reducing dynamodb.throughput.write.percent below 0.5. If you have sufficient capacity in the table and want more responsive HiveQL operations, you can set the value above 0.5.

When you write data to DynamoDB using Hive, ensure that the number of write capacity units is greater than the number of mappers in the cluster. For example, consider an Amazon EMR cluster consisting of 10 *m1.xlarge* nodes. The *m1.xlarge* node type provides 8 mapper tasks, so the cluster would have a total of 80 mappers (10×8). If your DynamoDB table has fewer than 80 write capacity units, then a Hive write operation could consume all of the write throughput for that table.

To determine the number of mappers for Amazon EMR node types, see [Task Configuration](#) in the *Amazon EMR Developer Guide*.

For more information on mappers, see [Adjusting the Mappers \(p. 958\)](#).

Adjusting the Mappers

When Hive launches a Hadoop job, the job is processed by one or more mapper tasks. Assuming that your DynamoDB table has sufficient throughput capacity, you can modify the number of mappers in the cluster, potentially improving performance.

Note

The number of mapper tasks used in a Hadoop job are influenced by *input splits*, where Hadoop subdivides the data into logical blocks. If Hadoop does not perform enough input splits, then your write operations might not be able to consume all the write throughput available in the DynamoDB table.

Increasing the Number of Mappers

Each mapper in an Amazon EMR has a maximum read rate of 1 MiB per second. The number of mappers in a cluster depends on the size of the nodes in your cluster. (For information about node sizes and the number of mappers per node, see [Task Configuration](#) in the *Amazon EMR Developer Guide*.)

If your DynamoDB table has ample throughput capacity for reads, you can try increasing the number of mappers by doing one of the following:

- Increase the size of the nodes in your cluster. For example, if your cluster is using *m1.large* nodes (three mappers per node), you can try upgrading to *m1.xlarge* nodes (eight mappers per node).
- Increase the number of nodes in your cluster. For example, if you have three-node cluster of *m1.xlarge* nodes, you have a total of 24 mappers available. If you were to double the size of the cluster, with the same type of node, you would have 48 mappers.

You can use the AWS Management Console to manage the size or the number of nodes in your cluster. (You might need to restart the cluster for these changes to take effect.)

Another way to increase the number of mappers is to modify the `mapred.tasktracker.map.tasks.maximum` Hadoop configuration parameter. (This is a Hadoop parameter, not a Hive parameter. You cannot modify it interactively from the command prompt.). If you increase the value of `mapred.tasktracker.map.tasks.maximum`, you can increase the number of mappers without increasing the size or number of nodes. However, it is possible for the cluster nodes to run out of memory if you set the value too high.

You set the value for `mapred.tasktracker.map.tasks.maximum` as a bootstrap action when you first launch your Amazon EMR cluster. For more information, see [\(Optional\) Create Bootstrap Actions to Install Additional Software](#) in the *Amazon EMR Management Guide*.

Decreasing the Number of Mappers

If you use the `SELECT` statement to select data from an external Hive table that maps to DynamoDB, the Hadoop job can use as many tasks as necessary, up to the maximum number of mappers in the cluster. In this scenario, it is possible that a long-running Hive query can consume all of the provisioned read capacity of the DynamoDB table, negatively impacting other users.

You can use the `dynamodb.max.map.tasks` parameter to set an upper limit for map tasks:

```
SET dynamodb.max.map.tasks=1
```

This value must be equal to or greater than 1. When Hive processes your query, the resulting Hadoop job will use no more than `dynamodb.max.map.tasks` when reading from the DynamoDB table.

Additional Topics

The following are some more ways to tune applications that use Hive to access DynamoDB.

Retry Duration

By default, Hive will rerun a Hadoop job if it has not returned any results from DynamoDB within two minutes. You can adjust this interval by modifying the `dynamodb.retry.duration` parameter:

```
SET dynamodb.retry.duration=2;
```

The value must be a nonzero integer, representing the number of minutes in the retry interval. The default for `dynamodb.retry.duration` is 2 (minutes).

Parallel Data Requests

Multiple data requests, either from more than one user or more than one application to a single table can drain read provisioned throughput and slow performance.

Process Duration

Data consistency in DynamoDB depends on the order of read and write operations on each node. While a Hive query is in progress, another application might load new data into the DynamoDB table or modify or delete existing data. In this case, the results of the Hive query might not reflect changes made to the data while the query was running.

Request Time

Scheduling Hive queries that access a DynamoDB table when there is lower demand on the DynamoDB table improves performance. For example, if most of your application's users live in San Francisco, you might choose to export daily data at 4:00 A.M. PST when the majority of users are asleep and not updating records in your DynamoDB database.

Service, Account, and Table Limits in Amazon DynamoDB

This section describes current limits within Amazon DynamoDB (or no limit, in some cases). Each limit applies on a per-Region basis unless otherwise specified.

Topics

- [Read/Write Capacity Mode and Throughput \(p. 960\)](#)
- [Tables \(p. 962\)](#)
- [Global Tables \(p. 962\)](#)
- [Secondary Indexes \(p. 963\)](#)
- [Partition Keys and Sort Keys \(p. 963\)](#)
- [Naming Rules \(p. 964\)](#)
- [Data Types \(p. 964\)](#)
- [Items \(p. 965\)](#)
- [Attributes \(p. 965\)](#)
- [Expression Parameters \(p. 966\)](#)
- [DynamoDB Transactions \(p. 966\)](#)
- [DynamoDB Streams \(p. 967\)](#)
- [DynamoDB Accelerator \(DAX\) \(p. 967\)](#)
- [API-Specific Limits \(p. 968\)](#)
- [DynamoDB Encryption at Rest \(p. 969\)](#)

Read/Write Capacity Mode and Throughput

You can switch between read/write capacity modes once every 24 hours.

Capacity Unit Sizes (for Provisioned Tables)

One read capacity unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One write capacity unit = one write per second, for items up to 1 KB in size.

Transactional read requests require two read capacity units to perform one read per second for items up to 4 KB.

Transactional write requests require two write capacity units to perform one write per second for items up to 1 KB.

Request Unit Sizes (for On-Demand Tables)

One read request unit = one strongly consistent read, or two eventually consistent reads, for items up to 4 KB in size.

One write request unit = one write, for items up to 1 KB in size.

Transactional read requests require two read request units to perform one read for items up to 4 KB.

Transactional write requests require two write request units to perform one write for items up to 1 KB.

Throughput Default Limits

AWS places some default limits on the throughput you can provision. These are the limits unless you request a higher amount. To request a service limit increase, see <https://aws.amazon.com/support>.

| | On-Demand | Provisioned |
|--|--|--|
| Per table | 40,000 read request units and 40,000 write request units | 40,000 read capacity units and 40,000 write capacity units |
| Per account | Not applicable | 80,000 read capacity units and 80,000 write capacity units |
| Minimum throughput for any table or global secondary index | Not applicable | 1 read capacity unit and 1 write capacity unit |

Note

All the account's available throughput can be applied to a single table or across multiple tables.

The provisioned throughput limit includes the sum of the capacity of the table together with the capacity of all of its global secondary indexes.

On the AWS Management Console, you can use Amazon CloudWatch to see what your current read and write throughput is in a given AWS Region by looking at the `read capacity` and `write capacity` graphs on the **Metrics** tab. Make sure that you are not too close to the limits.

If you increased your provisioned throughput default limits, you can use the [DescribeLimits](#) operation to see the current limit values.

Increasing or Decreasing Throughput (for Provisioned Tables)

Increasing Provisioned Throughput

You can increase `ReadCapacityUnits` or `WriteCapacityUnits` as often as necessary, using the AWS Management Console or the `UpdateTable` operation. In a single call, you can increase the provisioned throughput for a table, for any global secondary indexes on that table, or for any combination of these. The new settings do not take effect until the `UpdateTable` operation is complete.

You can't exceed your per-account limits when you add provisioned capacity, and DynamoDB doesn't allow you to increase provisioned capacity very rapidly. Aside from these restrictions, you can increase the provisioned capacity for your tables as high as you need. For more information about per-account limits, see the preceding section, [Throughput Default Limits \(p. 961\)](#).

Decreasing Provisioned Throughput

For every table and global secondary index in an `UpdateTable` operation, you can decrease `ReadCapacityUnits` or `WriteCapacityUnits` (or both). The new settings don't take effect until the `UpdateTable` operation is complete. A decrease is allowed up to four times, anytime per day. A day is

defined according to Universal Coordinated Time (UTC). Additionally, if there was no decrease in the past hour, an additional decrease is allowed. This effectively brings the maximum number of decreases in a day to 27 times (4 decreases in the first hour, and 1 decrease for each of the subsequent 1-hour windows in a day).

Important

Table and global secondary index decrease limits are decoupled, so any global secondary indexes for a particular table have their own decrease limits. However, if a single request decreases the throughput for a table and a global secondary index, it is rejected if either exceeds the current limits. Requests are not partially processed.

Example

In the first 4 hours of a day, a table with a global secondary index can be modified as follows:

- Decrease the table's `WriteCapacityUnits` or `ReadCapacityUnits` (or both) four times.
- Decrease the `WriteCapacityUnits` or `ReadCapacityUnits` (or both) of the global secondary index four times.

At the end of that same day, the table and the global secondary index throughput can potentially be decreased a total of 27 times each.

Tables

Table Size

There is no practical limit on a table's size. Tables are unconstrained in terms of the number of items or the number of bytes.

Tables Per Account

For any AWS account, there is an initial limit of 256 tables per AWS Region.

To request a service limit increase, see <https://aws.amazon.com/support>.

Global Tables

AWS places some default limits on the throughput you can provision or utilize when using global tables.

| | On-Demand | Provisioned |
|--------------------|--|--|
| Per table | 40,000 read request units and 40,000 write request units | 40,000 read capacity units and 40,000 write capacity units |
| Per table, per day | 10 TB for all source tables to which a replica was added | 10 TB for all source tables to which a replica was added |

If you are adding a replica for a table that is configured to use more than 40,000 write capacity units (WCU), you must request a service limit increase for your add replica WCU limit. To request a service limit increase see <https://aws.amazon.com/support>.

Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write is made originally. Transactions are not supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East (Ohio) and US West (Oregon) Regions and you perform a `TransactWriteItems` operation in the US East (N. Virginia) Region. In this case, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

Secondary Indexes

Secondary Indexes Per Table

You can define a maximum of 5 local secondary indexes.

There is an initial limit of 20 global secondary indexes per table. To request a service limit increase, see <https://aws.amazon.com/support>.

You can create or delete only one global secondary index per `UpdateTable` operation.

Projected Secondary Index Attributes Per Table

You can project a total of up to 100 attributes into all of a table's local and global secondary indexes. This only applies to user-specified projected attributes.

In a `CreateTable` operation, if you specify a `ProjectionType` of `INCLUDE`, the total count of attributes specified in `NonKeyAttributes`, summed across all of the secondary indexes, must not exceed 100. If you project the same attribute name into two different indexes, this counts as two distinct attributes when determining the total.

This limit does not apply for secondary indexes with a `ProjectionType` of `KEYS_ONLY` or `ALL`.

Partition Keys and Sort Keys

Partition Key Length

The minimum length of a partition key value is 1 byte. The maximum length is 2048 bytes.

Partition Key Values

There is no practical limit on the number of distinct partition key values, for tables or for secondary indexes.

Sort Key Length

The minimum length of a sort key value is 1 byte. The maximum length is 1024 bytes.

Sort Key Values

In general, there is no practical limit on the number of distinct sort key values per partition key value.

The exception is for tables with secondary indexes. With a local secondary index, there is a limit on item collection sizes: For every distinct partition key value, the total sizes of all table and index items cannot exceed 10 GB. This might constrain the number of sort keys per partition key value. For more information, see [Item Collection Size Limit \(p. 546\)](#).

Naming Rules

Table Names and Secondary Index Names

Names for tables and secondary indexes must be at least 3 characters long, but no greater than 255 characters long. The following are the allowed characters:

- A-Z
 - a-z
 - 0-9
 - _ (underscore)
 - - (hyphen)
 - . (dot)

Attribute Names

In general, an attribute name must be at least one character long, but no greater than 64 KB long.

The following are the exceptions. These attribute names must be no greater than 255 characters long:

- Secondary index partition key names.
 - Secondary index sort key names.
 - The names of any user-specified projected attributes (applicable only to local secondary indexes). In a CreateTable operation, if you specify a `ProjectionType` of `INCLUDE`, the names of the attributes in the `NonKeyAttributes` parameter are length-restricted. The `KEYS_ONLY` and `ALL` projection types are not affected.

These attribute names must be encoded using UTF-8, and the total size of each name (after encoding) cannot exceed 255 bytes.

Data Types

String

The length of a String is constrained by the maximum item size of 400 KB.

Strings are Unicode with UTF-8 binary encoding. Because UTF-8 is a variable width encoding, DynamoDB determines the length of a String using its UTF-8 bytes.

Number

A Number can have up to 38 digits of precision, and can be positive, negative, or zero.

DynamoDB uses JSON strings to represent Number data in requests and replies. For more information, see [DynamoDB Low-Level API \(p. 216\)](#).

If number precision is important, you should pass numbers to DynamoDB using strings that you convert from a number type.

Binary

The length of a Binary is constrained by the maximum item size of 400 KB.

Applications that work with Binary attributes must encode the data in base64 format before sending it to DynamoDB. Upon receipt of the data, DynamoDB decodes it into an unsigned byte array and uses that as the length of the attribute.

Items

Item Size

The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value lengths (again binary length). The attribute name counts towards the size limit.

For example, consider an item with two attributes: one attribute named "shirt-color" with value "R" and another attribute named "shirt-size" with value "M". The total size of that item is 23 bytes.

Item Size for Tables with Local Secondary Indexes

For each local secondary index on a table, there is a 400 KB limit on the total of the following:

- The size of an item's data in the table.
 - The size of the local secondary index entry corresponding to that item, including its key values and projected attributes.

Attributes

Attribute Name-Value Pairs Per Item

The cumulative size of attributes per item must fit within the maximum DynamoDB item size (400 KB).

Number of Values in List, Map, or Set

There is no limit on the number of values in a List, a Map, or a Set, as long as the item containing the values fits within the 400 KB item size limit.

Attribute Values

Empty String and Binary attribute values are allowed, if the attribute is not used as a key attribute for a table or index. Empty String and Binary values are allowed inside Set, List, and Map types. An attribute

value cannot be an empty Set (String Set, Number Set, or Binary Set). However, empty Lists and Maps are allowed.

Nested Attribute Depth

DynamoDB supports nested attributes up to 32 levels deep.

Expression Parameters

Expression parameters include `ProjectionExpression`, `ConditionExpression`, `UpdateExpression`, and `FilterExpression`.

Lengths

The maximum length of any expression string is 4 KB. For example, the size of the `ConditionExpression` `a=b` is 3 bytes.

The maximum length of any single expression attribute name or expression attribute value is 255 bytes. For example, `#name` is 5 bytes; `:val` is 4 bytes.

The maximum length of all substitution variables in an expression is 2 MB. This is the sum of the lengths of all `ExpressionAttributeNames` and `ExpressionAttributeValues`.

Operators and Operands

The maximum number of operators or functions allowed in an `UpdateExpression` is 300. For example, the `UpdateExpression SET a = :val1 + :val2 + :val3` contains two "+" operators.

The maximum number of operands for the `IN` comparator is 100.

Reserved Words

DynamoDB does not prevent you from using names that conflict with reserved words. (For a complete list, see [Reserved Words in DynamoDB \(p. 1026\)](#).)

However, if you use a reserved word in an expression parameter, you must also specify `ExpressionAttributeNames`. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

DynamoDB Transactions

DynamoDB transactional API operations have the following constraints:

- A transaction cannot contain more than 25 unique items.
- A transaction cannot contain more than 4 MB of data.
- No two actions in a transaction can work against the same item in the same table. For example, you cannot both `ConditionCheck` and `Update` the same item in one transaction.
- A transaction cannot operate on tables in more than one AWS account or Region.
- Transactional operations provide atomicity, consistency, isolation, and durability (ACID) guarantees only within the AWS Region where the write is made originally. Transactions are not supported across Regions in global tables. For example, suppose that you have a global table with replicas in the US East

(Ohio) and US West (Oregon) Regions and you perform a `TransactWriteItems` operation in the US East (N. Virginia) Region. In this case, you might observe partially completed transactions in the US West (Oregon) Region as changes are replicated. Changes are replicated to other Regions only after they have been committed in the source Region.

DynamoDB Streams

Simultaneous Readers of a Shard in DynamoDB Streams

Do not allow more than two processes to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling.

Maximum Write Capacity for a Table with a Stream Enabled

AWS places some default limits on the write capacity for DynamoDB tables with DynamoDB Streams enabled. These are the limits unless you request a higher amount. To request a service limit increase, see <https://aws.amazon.com/support>.

- US East (N. Virginia), US East (Ohio), US West (N. California), US West (Oregon), South America (São Paulo), Europe (Frankfurt), Europe (Ireland), Asia Pacific (Tokyo), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), China (Beijing) Regions:
 - Per table – 40,000 write capacity units
- All other Regions:
 - Per table – 10,000 write capacity units

Note

The provisioned throughput limits also apply for DynamoDB tables with DynamoDB Streams enabled. For more information, see [Throughput Default Limits \(p. 961\)](#).

DynamoDB Accelerator (DAX)

AWS Region Availability

For a list of AWS Regions in which DAX is available, see [DynamoDB Accelerator \(DAX\)](#) in the *AWS General Reference*.

Nodes

A DAX cluster consists of exactly one primary node, and between zero and nine read replica nodes.

The total number of nodes (per AWS account) cannot exceed 50 in a single AWS Region.

Parameter Groups

You can create up to 20 DAX parameter groups per Region.

Subnet Groups

You can create up to 50 DAX subnet groups per Region.

Within a subnet group, you can define up to 20 subnets.

API-Specific Limits

CreateTable/UpdateTable/DeleteTable

In general, you can have up to 50 `CreateTable`, `UpdateTable`, and `DeleteTable` requests running simultaneously (in any combination). In other words, the total number of tables in the CREATING, UPDATING, or DELETING state cannot exceed 50.

The only exception is when you are creating a table with one or more secondary indexes. You can have up to 25 such requests running at a time. However, if the table or index specifications are complex, DynamoDB might temporarily reduce the number of concurrent operations.

BatchGetItem

A single `BatchGetItem` operation can retrieve a maximum of 100 items. The total size of all the items retrieved cannot exceed 16 MB.

BatchWriteItem

A single `BatchWriteItem` operation can contain up to 25 `PutItem` or `DeleteItem` requests. The total size of all the items written cannot exceed 16 MB.

DescribeTableReplicaAutoScaling

`DescribeTableReplicaAutoScaling` method supports only 10 requests per second.

DescribeLimits

`DescribeLimits` should be called only periodically. You can expect throttling errors if you call it more than once in a minute.

DescribeContributorInsights/ListContributorInsights/UpdateContributorInsights

`DescribeContributorInsights`, `ListContributorInsights` and `UpdateContributorInsights` should be called only periodically. DynamoDB supports up to five requests per second for each of these APIs.

Query

The result set from a `Query` is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the query response to retrieve more results.

Scan

The result set from a `Scan` is limited to 1 MB per call. You can use the `LastEvaluatedKey` from the scan response to retrieve more results.

UpdateTableReplicaAutoScaling

`UpdateTableReplicaAutoScaling` method supports only 10 requests per second.

DynamoDB Encryption at Rest

You can switch from an AWS owned customer master key (CMK) to an AWS managed CMK up to four times, anytime per 24-hour window, starting from when the table was created. And if there was no change in the past 6 hours, an additional change is allowed. This effectively brings the maximum number of changes in a day to eight times (four changes in the first 6 hours, and one change for each of the subsequent 6-hour windows in a day).

You can switch encryption keys to use an AWS owned CMK as often as necessary.

These are the limits unless you request a higher amount. To request a service limit increase, see <https://aws.amazon.com/support>.

Low-Level API Reference

The [Amazon DynamoDB API Reference](#) contains a complete list of operations supported by:

- [DynamoDB](#).
- [DynamoDB Streams](#).
- [DynamoDB Accelerator \(DAX\)](#).

DynamoDB Appendix

Topics

- [Troubleshooting SSL/TLS connection establishment issues \(p. 971\)](#)
- [Example Tables and Data \(p. 973\)](#)
- [Creating Example Tables and Uploading Data \(p. 982\)](#)
- [DynamoDB Example Application Using the AWS SDK for Python \(Boto\): Tic-Tac-Toe \(p. 998\)](#)
- [Exporting and Importing DynamoDB Data Using AWS Data Pipeline \(p. 1017\)](#)
- [Amazon DynamoDB Storage Backend for Titan \(p. 1025\)](#)
- [Reserved Words in DynamoDB \(p. 1026\)](#)
- [Legacy Conditional Parameters \(p. 1035\)](#)
- [Previous Low-Level API Version \(2011-12-05\) \(p. 1051\)](#)

Troubleshooting SSL/TLS connection establishment issues

Amazon DynamoDB is in the process of moving our endpoints to secure certificates signed by the Amazon Trust Services (ATS) Certificate Authority instead of third-party Certificate Authority. In December 2017, we launched the EU-WEST-3 (Paris) Region with the secure certificates issued by the Amazon Trust Services. All new regions launched after December 2017 have endpoints with the certificates issued by the Amazon Trust Services. This guide shows you how to validate and troubleshoot SSL/TLS connection issues.

Testing your application or service

Most AWS SDKs and Command Line Interfaces (CLI's) support the Amazon Trust Services Certificate Authority. If you are using a version of the Python AWS SDK or CLI released before October 29, 2013, you must upgrade. The .NET, Java, PHP, Go, JavaScript, and C++ SDKs and CLI's do not bundle any certificates, their certificates come from the underlying operating system. The Ruby SDK has included at least one of the required CAs since June 10, 2015. Before that date, the Ruby V2 SDK did not bundle certificates. If you use an unsupported, custom, or modified version of the AWS SDK, or if you use custom trust store, you might not have the support needed for Amazon Trust Services Certificate Authority.

To validate access to DynamoDB endpoints, you will need to develop a test that accesses DynamoDB API or DynamoDB Streams API in the EU-WEST-3 region and validate that the TLS handshake succeeds. The specific endpoints you will need to access in such test are:

- DynamoDB: <https://dynamodb.eu-west-3.amazonaws.com>
- DynamoDB Streams: <https://streams.dynamodb.eu-west-3.amazonaws.com>

If your application does not support Amazon Trust Services Certificate Authority you will see one of the following failures:

- SSL/TLS Negotiation errors
- A long delay before your software receives an error indicating SSL/TLS negotiation failure. The delay time depends on the retry strategy and timeout configuration of your client.

Testing your client browser

To verify that your browser can connect to Amazon DynamoDB, open the following URL: <https://dynamodb.eu-west-3.amazonaws.com>. If the test is successful, you will see a message like this:

```
healthy: dynamodb.eu-west-3.amazonaws.com
```

If the test is unsuccessful, it will display an error similar to this: <https://untrusted-root.badssl.com/>.

Updating your software application client

Applications accessing DynamoDB or DynamoDB Streams API endpoints (whether through browsers or programmatically) will need to update the trusted CA list on the client machines if they do not already support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

If the clients already trust ANY of the above three CAs then these will trust certificates used by DynamoDB and no action is required. However, if your clients do not already trust any of the above CAs, the HTTPS connections to the DynamoDB or DynamoDB Streams APIs will fail. For more information, please visit this blog post: <https://aws.amazon.com/blogs/security/how-to-prepare-for-aws-move-to-its-own-certificate-authority/>.

Updating your client browser

You can update the certificate bundle in your browser simply by updating your browser. Instructions for the most common browsers can be found on the browsers' websites:

- Chrome: <https://support.google.com/chrome/answer/95414?hl=en>
- Firefox: <https://support.mozilla.org/en-US/kb/update-firefox-latest-version>
- Safari: <https://support.apple.com/en-us/HT204416>
- Internet Explorer: <https://support.microsoft.com/en-us/help/17295/windows-internet-explorer-which-version#ie=other>

Manually updating your certificate bundle

If you cannot access the DynamoDB API or DynamoDB Streams API then you need to update your certificate bundle. To do this, you need to import at least one of the required CAs. You can find these at <https://www.amazontrust.com/repository/>.

The following operating systems and programming languages support Amazon Trust Services certificates:

- Microsoft Windows versions that have January 2005 or later updates installed, Windows Vista, Windows 7, Windows Server 2008, and newer versions.
- Mac OS X 10.4 with Java for Mac OS X 10.4 Release 5, Mac OS X 10.5 and newer versions.
- Red Hat Enterprise Linux 5 (March 2007), Linux 6, and Linux 7 and CentOS 5, CentOS 6, and CentOS 7
- Ubuntu 8.10
- Debian 5.0

- Amazon Linux (all versions)
- Java 1.4.2_12, Java 5 update 2, and all newer versions, including Java 6, Java 7, and Java 8

If you are still unable to connect please consult your software documentation, OS Vendor or contact AWS Support <https://aws.amazon.com/support> for further assistance.

Example Tables and Data

The *Amazon DynamoDB Developer Guide* uses sample tables to illustrate various aspects of DynamoDB.

| Table Name | Primary Key |
|-----------------------|--|
| <i>ProductCatalog</i> | Simple primary key: <ul style="list-style-type: none">• <code>Id</code> (Number) |
| <i>Forum</i> | Simple primary key: <ul style="list-style-type: none">• <code>Name</code> (String) |
| <i>Thread</i> | Composite primary key: <ul style="list-style-type: none">• <code>ForumName</code> (String)• <code>Subject</code> (String) |
| <i>Reply</i> | Composite primary key: <ul style="list-style-type: none">• <code>Id</code> (String)• <code>ReplyDateTime</code> (String) |

The *Reply* table has a global secondary index named *PostedBy-Message-Index*. This index will facilitate queries on two non-key attributes of the *Reply* table.

| Index Name | Primary Key |
|-------------------------------|---|
| <i>PostedBy-Message-Index</i> | Composite primary key: <ul style="list-style-type: none">• <code>PostedBy</code> (String)• <code>Message</code> (String) |

For more information about these tables, see [Use Case 1: Product Catalog \(p. 325\)](#) and [Use Case 2: Forum Application \(p. 325\)](#).

Sample Data Files

Topics

- [ProductCatalog Sample Data \(p. 974\)](#)
- [Forum Sample Data \(p. 978\)](#)
- [Thread Sample Data \(p. 979\)](#)
- [Reply Sample Data \(p. 981\)](#)

The following sections show the sample data files that are used for loading the *ProductCatalog*, *Forum*, *Thread* and *Reply* tables.

Each data file contains multiple `PutRequest` elements, each of which contain a single item. These `PutRequest` elements are used as input to the `BatchWriteItem` operation, using the AWS Command Line Interface (AWS CLI).

For more information, see [Step 2: Load Data into Tables \(p. 327\)](#) in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#).

ProductCatalog Sample Data

```
{  
    "ProductCatalog": [  
        {  
            "PutRequest": {  
                "Item": {  
                    "Id": {  
                        "N": "101"  
                    },  
                    "Title": {  
                        "S": "Book 101 Title"  
                    },  
                    "ISBN": {  
                        "S": "111-1111111111"  
                    },  
                    "Authors": {  
                        "L": [  
                            {  
                                "S": "Author1"  
                            }  
                        ]  
                    },  
                    "Price": {  
                        "N": "2"  
                    },  
                    "Dimensions": {  
                        "S": "8.5 x 11.0 x 0.5"  
                    },  
                    "PageCount": {  
                        "N": "500"  
                    },  
                    "InPublication": {  
                        "BOOL": true  
                    },  
                    "ProductCategory": {  
                        "S": "Book"  
                    }  
                }  
            }  
        },  
        {  
            "PutRequest": {  
                "Item": {  
                    "Id": {  
                        "N": "102"  
                    },  
                    "Title": {  
                        "S": "Book 102 Title"  
                    },  
                    "ISBN": {  
                        "S": "222-2222222222"  
                    },  
                    "Authors": {  
                }  
            }  
        }  
    ]  
}
```

```
"L": [
    {
        "S": "Author1"
    },
    {
        "S": "Author2"
    }
]
},
"Price": {
    "N": "20"
},
"Dimensions": {
    "S": "8.5 x 11.0 x 0.8"
},
"PageCount": {
    "N": "600"
},
"InPublication": {
    "BOOL": true
},
"ProductCategory": {
    "S": "Book"
}
}
}
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "103"
            },
            "Title": {
                "S": "Book 103 Title"
            },
            "ISBN": {
                "S": "333-3333333333"
            },
            "Authors": {
                "L": [
                    {
                        "S": "Author1"
                    },
                    {
                        "S": "Author2"
                    }
                ]
            },
            "Price": {
                "N": "2000"
            },
            "Dimensions": {
                "S": "8.5 x 11.0 x 1.5"
            },
            "PageCount": {
                "N": "600"
            },
            "InPublication": {
                "BOOL": false
            },
            "ProductCategory": {
                "S": "Book"
            }
        }
    }
}
```

```
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "201"
            },
            "Title": {
                "S": "18-Bike-201"
            },
            "Description": {
                "S": "201 Description"
            },
            "BicycleType": {
                "S": "Road"
            },
            "Brand": {
                "S": "Mountain A"
            },
            "Price": {
                "N": "100"
            },
            "Color": {
                "L": [
                    {
                        "S": "Red"
                    },
                    {
                        "S": "Black"
                    }
                ]
            },
            "ProductCategory": {
                "S": "Bicycle"
            }
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "202"
            },
            "Title": {
                "S": "21-Bike-202"
            },
            "Description": {
                "S": "202 Description"
            },
            "BicycleType": {
                "S": "Road"
            },
            "Brand": {
                "S": "Brand-Company A"
            },
            "Price": {
                "N": "200"
            },
            "Color": {
                "L": [
                    {
                        "S": "Green"
                    },
                    {
                        "S": "Black"
                    }
                ]
            }
        }
    }
}
```

```
        }
    ],
},
"ProductCategory": {
    "S": "Bicycle"
}
}
}
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "203"
            },
            "Title": {
                "S": "19-Bike-203"
            },
            "Description": {
                "S": "203 Description"
            },
            "BicycleType": {
                "S": "Road"
            },
            "Brand": {
                "S": "Brand-Company B"
            },
            "Price": {
                "N": "300"
            },
            "Color": {
                "L": [
                    {
                        "S": "Red"
                    },
                    {
                        "S": "Green"
                    },
                    {
                        "S": "Black"
                    }
                ]
            },
            "ProductCategory": {
                "S": "Bicycle"
            }
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "204"
            },
            "Title": {
                "S": "18-Bike-204"
            },
            "Description": {
                "S": "204 Description"
            },
            "BicycleType": {
                "S": "Mountain"
            },
            "Brand": {
                "S": "Brand-Company B"
            }
        }
    }
}
```

```

        },
        "Price": {
            "N": "400"
        },
        "Color": {
            "L": [
                {
                    "S": "Red"
                }
            ]
        },
        "ProductCategory": {
            "S": "Bicycle"
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "N": "205"
            },
            "Title": {
                "S": "18-Bike-204"
            },
            "Description": {
                "S": "205 Description"
            },
            "BicycleType": {
                "S": "Hybrid"
            },
            "Brand": {
                "S": "Brand-Company C"
            },
            "Price": {
                "N": "500"
            },
            "Color": {
                "L": [
                    {
                        "S": "Red"
                    },
                    {
                        "S": "Black"
                    }
                ]
            },
            "ProductCategory": {
                "S": "Bicycle"
            }
        }
    }
}
]
}

```

Forum Sample Data

```
{
    "Forum": [
        {
            "PutRequest": {
                "Item": {

```

```
        "Name": {"S": "Amazon DynamoDB"},  
        "Category": {"S": "Amazon Web Services"},  
        "Threads": {"N": "2"},  
        "Messages": {"N": "4"},  
        "Views": {"N": "1000"}  
    }  
},  
{  
    "PutRequest": {  
        "Item": {  
            "Name": {"S": "Amazon S3"},  
            "Category": {"S": "Amazon Web Services"}  
        }  
    }  
},  

```

Thread Sample Data

```
{  
    "Thread": [  
        {  
            "PutRequest": {  
                "Item": {  
                    "ForumName": {  
                        "S": "Amazon DynamoDB"  
                    },  
                    "Subject": {  
                        "S": "DynamoDB Thread 1"  
                    },  
                    "Message": {  
                        "S": "DynamoDB thread 1 message"  
                    },  
                    "LastPostedBy": {  
                        "S": "User A"  
                    },  
                    "LastPostedDateTime": {  
                        "S": "2015-09-22T19:58:22.514Z"  
                    },  
                    "Views": {  
                        "N": "0"  
                    },  
                    "Replies": {  
                        "N": "0"  
                    },  
                    "Answered": {  
                        "N": "0"  
                    },  
                    "Tags": {  
                        "L": [  
                            {  
                                "S": "index"  
                            },  
                            {  
                                "S": "primarykey"  
                            },  
                            {  
                                "S": "table"  
                            }  
                        ]  
                    }  
                }  
            }  
        }  
    ]  
}
```

```
        }
    },
{
    "PutRequest": {
        "Item": {
            "ForumName": {
                "S": "Amazon DynamoDB"
            },
            "Subject": {
                "S": "DynamoDB Thread 2"
            },
            "Message": {
                "S": "DynamoDB thread 2 message"
            },
            "LastPostedBy": {
                "S": "User A"
            },
            "LastPostedDateTime": {
                "S": "2015-09-15T19:58:22.514Z"
            },
            "Views": {
                "N": "3"
            },
            "Replies": {
                "N": "0"
            },
            "Answered": {
                "N": "0"
            },
            "Tags": {
                "L": [
                    {
                        "S": "items"
                    },
                    {
                        "S": "attributes"
                    },
                    {
                        "S": "throughput"
                    }
                ]
            }
        }
    },
{
    "PutRequest": {
        "Item": {
            "ForumName": {
                "S": "Amazon S3"
            },
            "Subject": {
                "S": "S3 Thread 1"
            },
            "Message": {
                "S": "S3 thread 1 message"
            },
            "LastPostedBy": {
                "S": "User A"
            },
            "LastPostedDateTime": {
                "S": "2015-09-29T19:58:22.514Z"
            },
            "Views": {
                "N": "0"
            }
        }
    }
}
```

```
        "Replies": {
            "N": "0"
        },
        "Answered": {
            "N": "0"
        },
        "Tags": [
            {
                "L": [
                    {
                        "S": "largeobjects"
                    },
                    {
                        "S": "multipart upload"
                    }
                ]
            }
        ]
    }
}
```

Reply Sample Data

```
{
    "Reply": [
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-15T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 1 text"
                    },
                    "PostedBy": {
                        "S": "User A"
                    }
                }
            }
        },
        {
            "PutRequest": {
                "Item": {
                    "Id": {
                        "S": "Amazon DynamoDB#DynamoDB Thread 1"
                    },
                    "ReplyDateTime": {
                        "S": "2015-09-22T19:58:22.947Z"
                    },
                    "Message": {
                        "S": "DynamoDB Thread 1 Reply 2 text"
                    },
                    "PostedBy": {
                        "S": "User B"
                    }
                }
            }
        },
        {
            "PutRequest": {
```

```
    "Item": {
        "Id": {
            "S": "Amazon DynamoDB#DynamoDB Thread 2"
        },
        "ReplyDateTime": {
            "S": "2015-09-29T19:58:22.947Z"
        },
        "Message": {
            "S": "DynamoDB Thread 2 Reply 1 text"
        },
        "PostedBy": {
            "S": "User A"
        }
    }
},
{
    "PutRequest": {
        "Item": {
            "Id": {
                "S": "Amazon DynamoDB#DynamoDB Thread 2"
            },
            "ReplyDateTime": {
                "S": "2015-10-05T19:58:22.947Z"
            },
            "Message": {
                "S": "DynamoDB Thread 2 Reply 2 text"
            },
            "PostedBy": {
                "S": "User A"
            }
        }
    }
}
]
```

Creating Example Tables and Uploading Data

Topics

- [Creating Example Tables and Uploading Data Using the AWS SDK for Java \(p. 982\)](#)
- [Creating Example Tables and Uploading Data Using the AWS SDK for .NET \(p. 989\)](#)

In [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#), you first create tables using the DynamoDB console and then use the AWS CLI to add data to the tables. This appendix provides code to both create the tables and add data programmatically.

Creating Example Tables and Uploading Data Using the AWS SDK for Java

The following Java code example creates tables and uploads data to the tables. The resulting table structure and data is shown in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). For step-by-step instructions to run this code using Eclipse, see [Java Code Examples \(p. 330\)](#).

```
/**  
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.  
 */
```

```

/*
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.
 */

package com.amazonaws.codesamples;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.HashSet;
import java.util.TimeZone;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.CreateTableRequest;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.LocalSecondaryIndex;
import com.amazonaws.services.dynamodbv2.model.Projection;
import com.amazonaws.services.dynamodbv2.model.ProjectionType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;

public class CreateTablesLoadData {

    static AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard().build();
    static DynamoDB dynamoDB = new DynamoDB(client);

    static SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss.SSS'Z'");

    static String productCatalogTableName = "ProductCatalog";
    static String forumTableName = "Forum";
    static String threadTableName = "Thread";
    static String replyTableName = "Reply";

    public static void main(String[] args) throws Exception {

        try {

            deleteTable(productCatalogTableName);
            deleteTable(forumTableName);
            deleteTable(threadTableName);
            deleteTable(replyTableName);

            // Parameter1: table name
            // Parameter2: reads per second
            // Parameter3: writes per second
            // Parameter4/5: partition key and data type
            // Parameter6/7: sort key and data type (if applicable)

            createTable(productCatalogTableName, 10L, 5L, "Id", "N");
            createTable(forumTableName, 10L, 5L, "Name", "S");
        }
    }
}

```

```

        createTable(threadTableName, 10L, 5L, "ForumName", "S", "Subject", "S");
        createTable(replyTableName, 10L, 5L, "Id", "S", "ReplyDateTime", "S");

        loadSampleProducts(productCatalogTableName);
        loadSampleForums(forumTableName);
        loadSampleThreads(threadTableName);
        loadSampleReplies(replyTableName);

    }
    catch (Exception e) {
        System.err.println("Program failed:");
        System.err.println(e.getMessage());
    }
    System.out.println("Success.");
}

private static void deleteTable(String tableName) {
    Table table = dynamoDB.getTable(tableName);
    try {
        System.out.println("Issuing DeleteTable request for " + tableName);
        table.delete();
        System.out.println("Waiting for " + tableName + " to be deleted...this may take
a while...");
        table.waitForDelete();

    }
    catch (Exception e) {
        System.err.println("DeleteTable request failed for " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
        String partitionKeyName, String partitionKeyType) {

    createTable(tableName, readCapacityUnits, writeCapacityUnits, partitionKeyName,
partitionKeyType, null, null);
}

private static void createTable(String tableName, long readCapacityUnits, long
writeCapacityUnits,
        String partitionKeyName, String partitionKeyType, String sortKeyName, String
sortKeyType) {

    try {

        ArrayList<KeySchemaElement> keySchema = new ArrayList<KeySchemaElement>();
        keySchema.add(new
KeySchemaElement().withAttributeName(partitionKeyName).withKeyType(KeyType.HASH)); // Partition

        // key

        ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();
        attributeDefinitions
            .add(new
AttributeDefinition().withAttributeName(partitionKeyName).withAttributeType(partitionKeyType));

        if (sortKeyName != null) {
            keySchema.add(new
KeySchemaElement().withAttributeName(sortKeyName).withKeyType(KeyType.RANGE)); // Sort

            // key
            attributeDefinitions
    }
}

```



```

        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 102).withString("Title", "Book 102
Title")
            .withString("ISBN", "222-222222222")
            .withStringSet("Authors", new HashSet<String>(Arrays.asList("Author1",
"Author2")))
            .withNumber("Price", 20).withString("Dimensions", "8.5 x 11.0 x
0.8").withNumber("PageCount", 600)
            .withBoolean("InPublication", true).withString("ProductCategory", "Book");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 103).withString("Title", "Book 103
Title")
            .withString("ISBN", "333-333333333")
            .withStringSet("Authors", new HashSet<String>(Arrays.asList("Author1",
"Author2")))
            // Intentional. Later we'll run Scan to find price error. Find
            // items > 1000 in price.
            .withNumber("Price", 2000).withString("Dimensions", "8.5 x 11.0 x
1.5").withNumber("PageCount", 600)
            .withBoolean("InPublication", false).withString("ProductCategory", "Book");
        table.putItem(item);

        // Add bikes.

        item = new Item().withPrimaryKey("Id", 201).withString("Title", "18-Bike-201")
            // Size, followed by some title.
            .withString("Description", "201 Description").withString("BicycleType",
"Road")
            .withString("Brand", "Mountain A")
            // Trek, Specialized.
            .withNumber("Price", 100).withStringSet("Color", new
HashSet<String>(Arrays.asList("Red", "Black")))
            .withString("ProductCategory", "Bicycle");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 202).withString("Title", "21-Bike-202")
            .withString("Description", "202 Description").withString("BicycleType",
"Road")
            .withString("Brand", "Brand-Company A").withNumber("Price", 200)
            .withStringSet("Color", new HashSet<String>(Arrays.asList("Green",
"Black")))
            .withString("ProductCategory", "Bicycle");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 203).withString("Title", "19-Bike-203")
            .withString("Description", "203 Description").withString("BicycleType",
"Road")
            .withString("Brand", "Brand-Company B").withNumber("Price", 300)
            .withStringSet("Color", new HashSet<String>(Arrays.asList("Red", "Green",
"Black")))
            .withString("ProductCategory", "Bicycle");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 204).withString("Title", "18-Bike-204")
            .withString("Description", "204 Description").withString("BicycleType",
"Mountain")
            .withString("Brand", "Brand-Company B").withNumber("Price", 400)
            .withStringSet("Color", new HashSet<String>(Arrays.asList("Red")))
            .withString("ProductCategory", "Bicycle");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", 205).withString("Title", "20-Bike-205")
            .withString("Description", "205 Description").withString("BicycleType",
"Hybrid")
    
```

```

        .withString("Brand", "Brand-Company C").withNumber("Price", 500)
        .withStringSet("Color", new HashSet<String>(Arrays.asList("Red", "Black")));
        .withString("ProductCategory", "Bicycle");
    table.putItem(item);

}

catch (Exception e) {
    System.err.println("Failed to create item in " + tableName);
    System.err.println(e.getMessage());
}

}

private static void loadSampleForums(String tableName) {

    Table table = dynamoDB.getTable(tableName);

    try {

        System.out.println("Adding data to " + tableName);

        Item item = new Item().withPrimaryKey("Name", "Amazon DynamoDB")
            .withString("Category", "Amazon Web Services").withNumber("Threads",
2).withNumber("Messages", 4)
            .withNumber("Views", 1000);
        table.putItem(item);

        item = new Item().withPrimaryKey("Name", "Amazon S3").withString("Category",
"Amazon Web Services")
            .withNumber("Threads", 0);
        table.putItem(item);

    }
    catch (Exception e) {
        System.err.println("Failed to create item in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void loadSampleThreads(String tableName) {
    try {
        long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000); // 7
        // days
        // ago
        long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000); // 14
        // days
        // ago
        long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000); // 21
        // days
        // ago

        Date date1 = new Date();
        date1.setTime(time1);

        Date date2 = new Date();
        date2.setTime(time2);

        Date date3 = new Date();
        date3.setTime(time3);

        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

        Table table = dynamoDB.getTable(tableName);

        System.out.println("Adding data to " + tableName);
    }
}

```

```

        Item item = new Item().withPrimaryKey("ForumName", "Amazon DynamoDB")
            .withString("Subject", "DynamoDB Thread 1").withString("Message", "DynamoDB
thread 1 message")
            .withString("LastPostedBy", "User A").withString("LastPostedDateTime",
dateFormatter.format(date2))
            .withNumber("Views", 0).withNumber("Replies", 0).withNumber("Answered", 0)
            .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"primarykey", "table")));
        table.putItem(item);

        item = new Item().withPrimaryKey("ForumName", "Amazon
DynamoDB").withString("Subject", "DynamoDB Thread 2")
            .withString("Message", "DynamoDB thread 2
message").withString("LastPostedBy", "User A")
            .withString("LastPostedDateTime",
dateFormatter.format(date3)).withNumber("Views", 0)
            .withNumber("Replies", 0).withNumber("Answered", 0)
            .withStringSet("Tags", new HashSet<String>(Arrays.asList("index",
"partitionkey", "sortkey")));
        table.putItem(item);

        item = new Item().withPrimaryKey("ForumName", "Amazon
S3").withString("Subject", "S3 Thread 1")
            .withString("Message", "S3 Thread 3 message").withString("LastPostedBy",
"User A")
            .withString("LastPostedDateTime",
dateFormatter.format(date1)).withNumber("Views", 0)
            .withNumber("Replies", 0).withNumber("Answered", 0)
            .withStringSet("Tags", new HashSet<String>(Arrays.asList("largeobjects",
"multipart upload")));
        table.putItem(item);

    }
    catch (Exception e) {
        System.err.println("Failed to create item in " + tableName);
        System.err.println(e.getMessage());
    }
}

private static void loadSampleReplies(String tableName) {
    try {
        // 1 day ago
        long time0 = (new Date()).getTime() - (1 * 24 * 60 * 60 * 1000);
        // 7 days ago
        long time1 = (new Date()).getTime() - (7 * 24 * 60 * 60 * 1000);
        // 14 days ago
        long time2 = (new Date()).getTime() - (14 * 24 * 60 * 60 * 1000);
        // 21 days ago
        long time3 = (new Date()).getTime() - (21 * 24 * 60 * 60 * 1000);

        Date date0 = new Date();
        date0.setTime(time0);

        Date date1 = new Date();
        date1.setTime(time1);

        Date date2 = new Date();
        date2.setTime(time2);

        Date date3 = new Date();
        date3.setTime(time3);

        dateFormatter.setTimeZone(TimeZone.getTimeZone("UTC"));

        Table table = dynamoDB.getTable(tableName);
    }
}

```

```

        System.out.println("Adding data to " + tableName);

        // Add threads.

        Item item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread
1")
            .withString("ReplyDateTime", (dateFormatter.format(date3)))
            .withString("Message", "DynamoDB Thread 1 Reply 1
text").withString("PostedBy", "User A");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 1")
            .withString("ReplyDateTime", dateFormatter.format(date2))
            .withString("Message", "DynamoDB Thread 1 Reply 2
text").withString("PostedBy", "User B");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
            .withString("ReplyDateTime", dateFormatter.format(date1))
            .withString("Message", "DynamoDB Thread 2 Reply 1
text").withString("PostedBy", "User A");
        table.putItem(item);

        item = new Item().withPrimaryKey("Id", "Amazon DynamoDB#DynamoDB Thread 2")
            .withString("ReplyDateTime", dateFormatter.format(date0))
            .withString("Message", "DynamoDB Thread 2 Reply 2
text").withString("PostedBy", "User A");
        table.putItem(item);

    }
    catch (Exception e) {
        System.err.println("Failed to create item in " + tableName);
        System.err.println(e.getMessage());
    }
}
}

```

Creating Example Tables and Uploading Data Using the AWS SDK for .NET

The following C# code example creates tables and uploads data to the tables. The resulting table structure and data is shown in [Creating Tables and Loading Data for Code Examples in DynamoDB \(p. 325\)](#). For step-by-step instructions to run this code in Visual Studio, see [.NET Code Examples \(p. 332\)](#).

```

/**
 * Copyright 2010-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * This file is licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License. A copy of
 * the License is located at
 *
 * http://aws.amazon.com/apache2.0/
 *
 * This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
 * CONDITIONS OF ANY KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations under the License.

```

```

/*
using System;
using System.Collections.Generic;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.DynamoDBv2.Model;
using Amazon.Runtime;
using Amazon.SecurityToken;

namespace com.amazonaws.codesamples
{
    class CreateTablesLoadData
    {
        private static AmazonDynamoDBClient client = new AmazonDynamoDBClient();

        static void Main(string[] args)
        {
            try
            {
                //DeleteAllTables(client);
                DeleteTable("ProductCatalog");
                DeleteTable("Forum");
                DeleteTable("Thread");
                DeleteTable("Reply");

                // Create tables (using the AWS SDK for .NET low-level API).
                CreateTableProductCatalog();
                CreateTableForum();
                CreateTableThread(); // ForumTitle, Subject */
                CreateTableReply();

                // Load data (using the .NET SDK document API)
                LoadSampleProducts();
                LoadSampleForums();
                LoadSampleThreads();
                LoadSampleReplies();
                Console.WriteLine("Sample complete!");
                Console.WriteLine("Press ENTER to continue");
                Console.ReadLine();
            }
            catch (AmazonServiceException e) { Console.WriteLine(e.Message); }
            catch (Exception e) { Console.WriteLine(e.Message); }
        }

        private static void DeleteTable(string tableName)
        {
            try
            {
                var deleteTableResponse = client.DeleteTable(new DeleteTableRequest()
                {
                    TableName = tableName
                });
                WaitTillTableDeleted(client, tableName, deleteTableResponse);
            }
            catch (ResourceNotFoundException)
            {
                // There is no such table.
            }
        }

        private static void CreateTableProductCatalog()
        {
            string tableName = "ProductCatalog";

            var response = client.CreateTable(new CreateTableRequest()
            {

```

```

        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Id",
                AttributeType = "N"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "Id",
                KeyType = "HASH"
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });
}

WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableForum()
{
    string tableName = "Forum";

    var response = client.CreateTable(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Name",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "Name", // forum Title
                KeyType = "HASH"
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });

    WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableThread()
{
    string tableName = "Thread";

    var response = client.CreateTable(new CreateTableRequest
    {

```

```

        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "ForumName", // Hash attribute
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "Subject",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement
            {
                AttributeName = "ForumName", // Hash attribute
                KeyType = "HASH"
            },
            new KeySchemaElement
            {
                AttributeName = "Subject", // Range attribute
                KeyType = "RANGE"
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });
    WaitTillTableCreated(client, tableName, response);
}

private static void CreateTableReply()
{
    string tableName = "Reply";
    var response = client.CreateTable(new CreateTableRequest
    {
        TableName = tableName,
        AttributeDefinitions = new List<AttributeDefinition>()
        {
            new AttributeDefinition
            {
                AttributeName = "Id",
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "ReplyDateTime",
                AttributeType = "S"
            },
            new AttributeDefinition
            {
                AttributeName = "PostedBy",
                AttributeType = "S"
            }
        },
        KeySchema = new List<KeySchemaElement>()
        {
            new KeySchemaElement()
            {
                AttributeName = "Id",
                KeyType = "HASH"
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });
    WaitTillTableCreated(client, tableName, response);
}

```

```

                KeyType = "HASH"
            },
            new KeySchemaElement()
            {
                AttributeName = "ReplyDateTime",
                KeyType = "RANGE"
            }
        },
        LocalSecondaryIndexes = new List<LocalSecondaryIndex>()
        {
            new LocalSecondaryIndex()
            {
                IndexName = "PostedBy_index",

                KeySchema = new List<KeySchemaElement>() {
                    new KeySchemaElement() {
                        AttributeName = "Id", KeyType = "HASH"
                    },
                    new KeySchemaElement() {
                        AttributeName = "PostedBy", KeyType = "RANGE"
                    }
                },
                Projection = new Projection() {
                    ProjectionType = ProjectionType.KEYS_ONLY
                }
            }
        },
        ProvisionedThroughput = new ProvisionedThroughput
        {
            ReadCapacityUnits = 10,
            WriteCapacityUnits = 5
        }
    });
}

WaitTillTableCreated(client, tableName, response);
}

private static void WaitTillTableCreated(AmazonDynamoDBClient client, string
tableName,
                                         CreateTableResponse response)
{
    var tableDescription = response.TableDescription;

    string status = tableDescription.TableStatus;

    Console.WriteLine(tableName + " - " + status);

    // Let us wait until table is created. Call DescribeTable.
    while (status != "ACTIVE")
    {
        System.Threading.Thread.Sleep(5000); // Wait 5 seconds.
        try
        {
            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });
            Console.WriteLine("Table name: {0}, status: {1}", res.Table.TableName,
                             res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
        // Try-catch to handle potential eventual-consistency issue.
        catch (ResourceNotFoundException)
        {
        }
    }
}

```

```

        }

    private static void WaitTillTableDeleted(AmazonDynamoDBClient client, string
tableName,
                                         DeleteTableResponse response)
{
    var tableDescription = response.TableDescription;

    string status = tableDescription.TableStatus;

    Console.WriteLine(tableName + " - " + status);

    // Let us wait until table is created. Call DescribeTable
    try
    {
        while (status == "DELETING")
        {
            System.Threading.Thread.Sleep(5000); // wait 5 seconds

            var res = client.DescribeTable(new DescribeTableRequest
            {
                TableName = tableName
            });
            Console.WriteLine("Table name: {0}, status: {1}", res.Table.TableName,
                res.Table.TableStatus);
            status = res.Table.TableStatus;
        }
    }
    catch (ResourceNotFoundException)
    {
        // Table deleted.
    }
}

private static void LoadSampleProducts()
{
    Table productCatalogTable = Table.LoadTable(client, "ProductCatalog");
    // ***** Add Books *****
    var book1 = new Document();
    book1["Id"] = 101;
    book1["Title"] = "Book 101 Title";
    book1["ISBN"] = "111-1111111111";
    book1["Authors"] = new List<string> { "Author 1" };
    book1["Price"] = -2; // *** Intentional value. Later used to illustrate scan.
    book1["Dimensions"] = "8.5 x 11.0 x 0.5";
    book1["PageCount"] = 500;
    book1["InPublication"] = true;
    book1["ProductCategory"] = "Book";
    productCatalogTable.PutItem(book1);

    var book2 = new Document();

    book2["Id"] = 102;
    book2["Title"] = "Book 102 Title";
    book2["ISBN"] = "222-222222222";
    book2["Authors"] = new List<string> { "Author 1", "Author 2" } ;
    book2["Price"] = 20;
    book2["Dimensions"] = "8.5 x 11.0 x 0.8";
    book2["PageCount"] = 600;
    book2["InPublication"] = true;
    book2["ProductCategory"] = "Book";
    productCatalogTable.PutItem(book2);

    var book3 = new Document();
    book3["Id"] = 103;
    book3["Title"] = "Book 103 Title";
}

```

```

book3["ISBN"] = "333-333333333";
book3["Authors"] = new List<string> { "Author 1", "Author2", "Author 3" }; ;
book3["Price"] = 2000;
book3["Dimensions"] = "8.5 x 11.0 x 1.5";
book3["PageCount"] = 700;
book3["InPublication"] = false;
book3["ProductCategory"] = "Book";
productCatalogTable.PutItem(book3);

// ***** Add bikes. *****
var bicycle1 = new Document();
bicycle1["Id"] = 201;
bicycle1["Title"] = "18-Bike 201"; // size, followed by some title.
bicycle1["Description"] = "201 description";
bicycle1["BicycleType"] = "Road";
bicycle1["Brand"] = "Brand-Company A"; // Trek, Specialized.
bicycle1["Price"] = 100;
bicycle1["Color"] = new List<string> { "Red", "Black" };
bicycle1["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle1);

var bicycle2 = new Document();
bicycle2["Id"] = 202;
bicycle2["Title"] = "21-Bike 202Brand-Company A";
bicycle2["Description"] = "202 description";
bicycle2["BicycleType"] = "Road";
bicycle2["Brand"] = "";
bicycle2["Price"] = 200;
bicycle2["Color"] = new List<string> { "Green", "Black" };
bicycle2["ProductCategory"] = "Bicycle";
productCatalogTable.PutItem(bicycle2);

var bicycle3 = new Document();
bicycle3["Id"] = 203;
bicycle3["Title"] = "19-Bike 203";
bicycle3["Description"] = "203 description";
bicycle3["BicycleType"] = "Road";
bicycle3["Brand"] = "Brand-Company B";
bicycle3["Price"] = 300;
bicycle3["Color"] = new List<string> { "Red", "Green", "Black" };
bicycle3["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle3);

var bicycle4 = new Document();
bicycle4["Id"] = 204;
bicycle4["Title"] = "18-Bike 204";
bicycle4["Description"] = "204 description";
bicycle4["BicycleType"] = "Mountain";
bicycle4["Brand"] = "Brand-Company B";
bicycle4["Price"] = 400;
bicycle4["Color"] = new List<string> { "Red" };
bicycle4["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle4);

var bicycle5 = new Document();
bicycle5["Id"] = 205;
bicycle5["Title"] = "20-Title 205";
bicycle5["Description"] = "205 description";
bicycle5["BicycleType"] = "Hybrid";
bicycle5["Brand"] = "Brand-Company C";
bicycle5["Price"] = 500;
bicycle5["Color"] = new List<string> { "Red", "Black" };
bicycle5["ProductCategory"] = "Bike";
productCatalogTable.PutItem(bicycle5);
}

```

```

private static void LoadSampleForums()
{
    Table forumTable = Table.LoadTable(client, "Forum");

    var forum1 = new Document();
    forum1["Name"] = "Amazon DynamoDB"; // PK
    forum1["Category"] = "Amazon Web Services";
    forum1["Threads"] = 2;
    forum1["Messages"] = 4;
    forum1["Views"] = 1000;

    forumTable.PutItem(forum1);

    var forum2 = new Document();
    forum2["Name"] = "Amazon S3"; // PK
    forum2["Category"] = "Amazon Web Services";
    forum2["Threads"] = 1;

    forumTable.PutItem(forum2);
}

private static void LoadSampleThreads()
{
    Table threadTable = Table.LoadTable(client, "Thread");

    // Thread 1.
    var thread1 = new Document();
    thread1["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
    thread1["Subject"] = "DynamoDB Thread 1"; // Range attribute.
    thread1["Message"] = "DynamoDB thread 1 message text";
    thread1["LastPostedBy"] = "User A";
    thread1["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0, 0));
    thread1["Views"] = 0;
    thread1["Replies"] = 0;
    thread1["Answered"] = false;
    thread1["Tags"] = new List<string> { "index", "primarykey", "table" };

    threadTable.PutItem(thread1);

    // Thread 2.
    var thread2 = new Document();
    thread2["ForumName"] = "Amazon DynamoDB"; // Hash attribute.
    thread2["Subject"] = "DynamoDB Thread 2"; // Range attribute.
    thread2["Message"] = "DynamoDB thread 2 message text";
    thread2["LastPostedBy"] = "User A";
    thread2["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0, 0));
    thread2["Views"] = 0;
    thread2["Replies"] = 0;
    thread2["Answered"] = false;
    thread2["Tags"] = new List<string> { "index", "primarykey", "rangekey" };

    threadTable.PutItem(thread2);

    // Thread 3.
    var thread3 = new Document();
    thread3["ForumName"] = "Amazon S3"; // Hash attribute.
    thread3["Subject"] = "S3 Thread 1"; // Range attribute.
    thread3["Message"] = "S3 thread 3 message text";
    thread3["LastPostedBy"] = "User A";
    thread3["LastPostedDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0, 0, 0));
    thread3["Views"] = 0;
    thread3["Replies"] = 0;
    thread3["Answered"] = false;
}

```

```

        thread3["Tags"] = new List<string> { "largeobjects", "multipart upload" };
        threadTable.PutItem(thread3);
    }

    private static void LoadSampleReplies()
    {
        Table replyTable = Table.LoadTable(client, "Reply");

        // Reply 1 - thread 1.
        var thread1Reply1 = new Document();
        thread1Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash attribute.
        thread1Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(21, 0, 0)); // Range attribute.
        thread1Reply1["Message"] = "DynamoDB Thread 1 Reply 1 text";
        thread1Reply1["PostedBy"] = "User A";

        replyTable.PutItem(thread1Reply1);

        // Reply 2 - thread 1.
        var thread1Reply2 = new Document();
        thread1Reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash attribute.
        thread1Reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(14, 0, 0)); // Range attribute.
        thread1Reply2["Message"] = "DynamoDB Thread 1 Reply 2 text";
        thread1Reply2["PostedBy"] = "User B";

        replyTable.PutItem(thread1Reply2);

        // Reply 3 - thread 1.
        var thread1Reply3 = new Document();
        thread1Reply3["Id"] = "Amazon DynamoDB#DynamoDB Thread 1"; // Hash attribute.
        thread1Reply3["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0, 0, 0)); // Range attribute.
        thread1Reply3["Message"] = "DynamoDB Thread 1 Reply 3 text";
        thread1Reply3["PostedBy"] = "User B";

        replyTable.PutItem(thread1Reply3);

        // Reply 1 - thread 2.
        var thread2Reply1 = new Document();
        thread2Reply1["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash attribute.
        thread2Reply1["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(7, 0, 0, 0)); // Range attribute.
        thread2Reply1["Message"] = "DynamoDB Thread 2 Reply 1 text";
        thread2Reply1["PostedBy"] = "User A";

        replyTable.PutItem(thread2Reply1);

        // Reply 2 - thread 2.
        var thread2Reply2 = new Document();
        thread2Reply2["Id"] = "Amazon DynamoDB#DynamoDB Thread 2"; // Hash attribute.
        thread2Reply2["ReplyDateTime"] = DateTime.UtcNow.Subtract(new TimeSpan(1, 0, 0, 0)); // Range attribute.
        thread2Reply2["Message"] = "DynamoDB Thread 2 Reply 2 text";
        thread2Reply2["PostedBy"] = "User A";

        replyTable.PutItem(thread2Reply2);
    }
}
}

```

DynamoDB Example Application Using the AWS SDK for Python (Boto): Tic-Tac-Toe

Topics

- [Step 1: Deploy and Test Locally \(p. 998\)](#)
- [Step 2: Examine the Data Model and Implementation Details \(p. 1002\)](#)
- [Step 3: Deploy in Production Using the DynamoDB Service \(p. 1010\)](#)
- [Step 4: Clean Up Resources \(p. 1016\)](#)

The Tic-Tac-Toe game is an example web application built on Amazon DynamoDB. The application uses the AWS SDK for Python (Boto) to make the necessary DynamoDB calls to store game data in a DynamoDB table, and the Python web framework Flask to illustrate end-to-end application development in DynamoDB, including how to model data. It also demonstrates best practices when it comes to modeling data in DynamoDB, including the table you create for the game application, the primary key you define, additional indexes you need based on your query requirements, and the use of concatenated value attributes.

You play the Tic-Tac-Toe application on the web as follows:

1. You log in to the application home page.
2. You then invite another user to play the game as your opponent.

Until another user accepts your invitation, the game status remains as `PENDING`. After an opponent accepts the invite, the game status changes to `IN_PROGRESS`.

3. The game begins after the opponent logs in and accepts the invite.
4. The application stores all game moves and status information in a DynamoDB table.
5. The game ends with a win or a draw, which sets the game status to `FINISHED`.

The end-to-end application building exercise is described in steps:

- **[Step 1: Deploy and Test Locally \(p. 998\)](#)** – In this section, you download, deploy, and test the application on your local computer. You will create the required tables in the downloadable version of DynamoDB.
- **[Step 2: Examine the Data Model and Implementation Details \(p. 1002\)](#)** – This section first describes in detail the data model, including the indexes and the use of the concatenated value attribute. Then the section explains how the application works.
- **[Step 3: Deploy in Production Using the DynamoDB Service \(p. 1010\)](#)** – This section focuses on deployment considerations in production. In this step, you create a table using the Amazon DynamoDB service and deploy the application using AWS Elastic Beanstalk. When you have the application in production, you also grant appropriate permissions so the application can access the DynamoDB table. The instructions in this section walk you through the end-to-end production deployment.
- **[Step 4: Clean Up Resources \(p. 1016\)](#)** – This section highlights areas that are not covered by this example. The section also provides steps for you to remove the AWS resources you created in the preceding steps so that you avoid incurring any charges.

Step 1: Deploy and Test Locally

Topics

- [1.1: Download and Install the Required Packages \(p. 999\)](#)

- [1.2: Test the Game Application \(p. 999\)](#)

In this step you download, deploy, and test the Tic-Tac-Toe game application on your local computer. Instead of using the Amazon DynamoDB web service, you will download DynamoDB to your computer, and create the required table there.

1.1: Download and Install the Required Packages

You will need the following to test this application locally:

- Python
- Flask (a microframework for Python)
- AWS SDK for Python (Boto)
- DynamoDB running on your computer
- Git

To get these tools, do the following:

1. Install Python. For step-by-step instructions, see [Download Python](#).

The Tic-Tac-Toe application has been tested using Python version 2.7.

2. Install Flask and AWS SDK for Python (Boto) using the Python Package Installer (PIP):

- Install PIP.

For instructions, see [Install PIP](#). On the installation page, choose the **get-pip.py** link, and then save the file. Then open a command terminal as an administrator, and enter the following at the command prompt.

```
python.exe get-pip.py
```

On Linux, you don't specify the .exe extension. You only specify `python get-pip.py`.

- Using PIP, install the Flask and Boto packages using the following code.

```
pip install Flask
pip install boto
pip install configparser
```

3. Download DynamoDB to your computer. For instructions on how to run it, see [Setting Up DynamoDB Local \(Downloadable Version\) \(p. 46\)](#).

4. Download the Tic-Tac-Toe application:

- a. Install Git. For instructions, see [git Downloads](#).
- b. Execute the following code to download the application.

```
git clone https://github.com/awslabs/dynamodb-tictactoe-example-app.git
```

1.2: Test the Game Application

To test the Tic-Tac-Toe application, you need to run DynamoDB locally on your computer.

To run the Tic-Tac-Toe application

1. Start DynamoDB.

2. Start the web server for the Tic-Tac-Toe application.

To do so, open a command terminal, navigate to the folder where you downloaded the Tic-Tac-Toe application, and run the application locally using the following code.

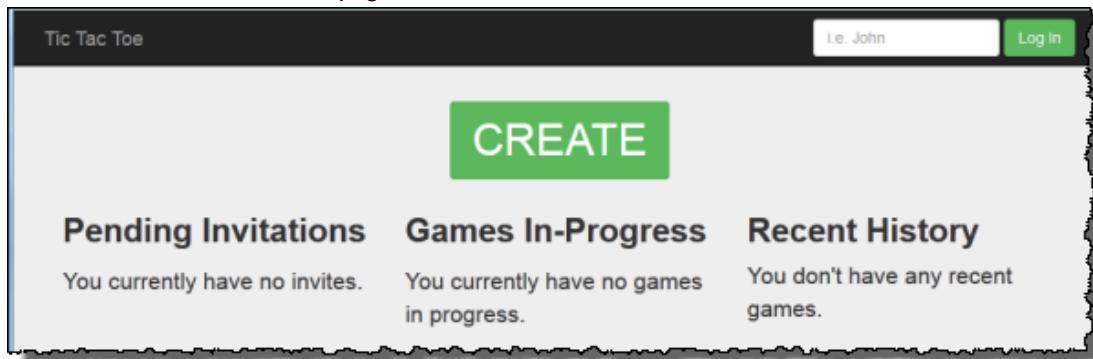
```
python.exe application.py --mode local --serverPort 5000 --port 8000
```

On Linux, you don't specify the .exe extension.

3. Open your web browser, and enter the following.

```
http://localhost:5000/
```

The browser shows the home page.

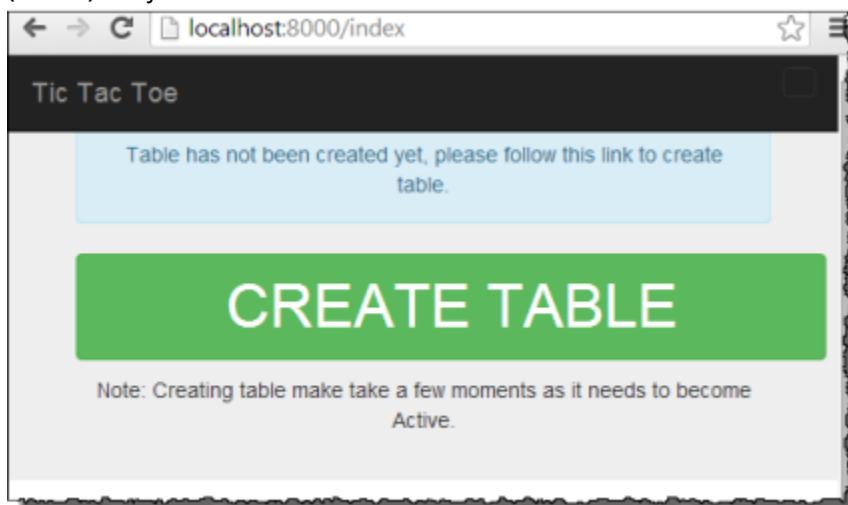


4. Enter **user1** in the **Log in** box to log in as user1.

Note

This example application does not perform any user authentication. The user ID is only used to identify players. If two players log in with the same alias, the application works as if you are playing in two different browsers.

5. If this is your first time playing the game, a page appears requesting you to create the required table (Games) in DynamoDB. Choose **CREATE TABLE**.



6. Choose **CREATE** to create the first tic-tac-toe game.
7. Enter **user2** in the **Choose an Opponent** box, and choose **Create Game!**



Doing this creates the game by adding an item in the Games table. It sets the game status to PENDING.

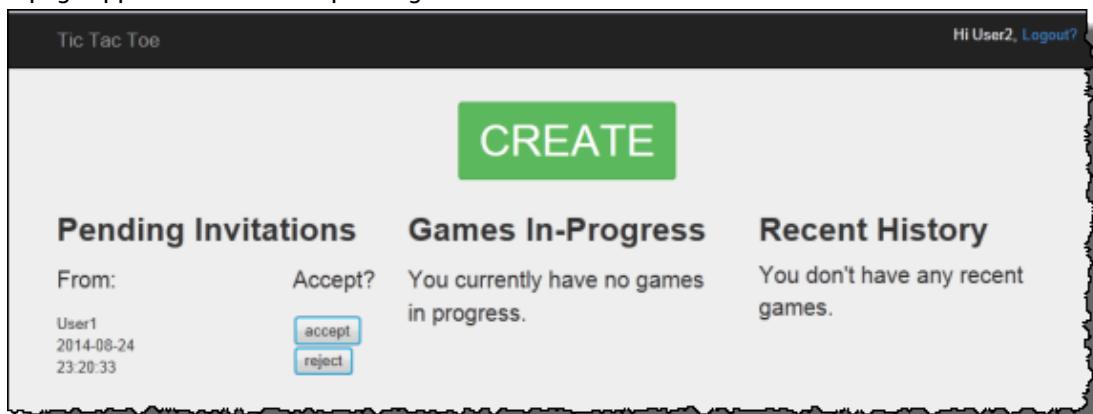
8. Open another browser window, and enter the following.

```
http://localhost:5000/
```

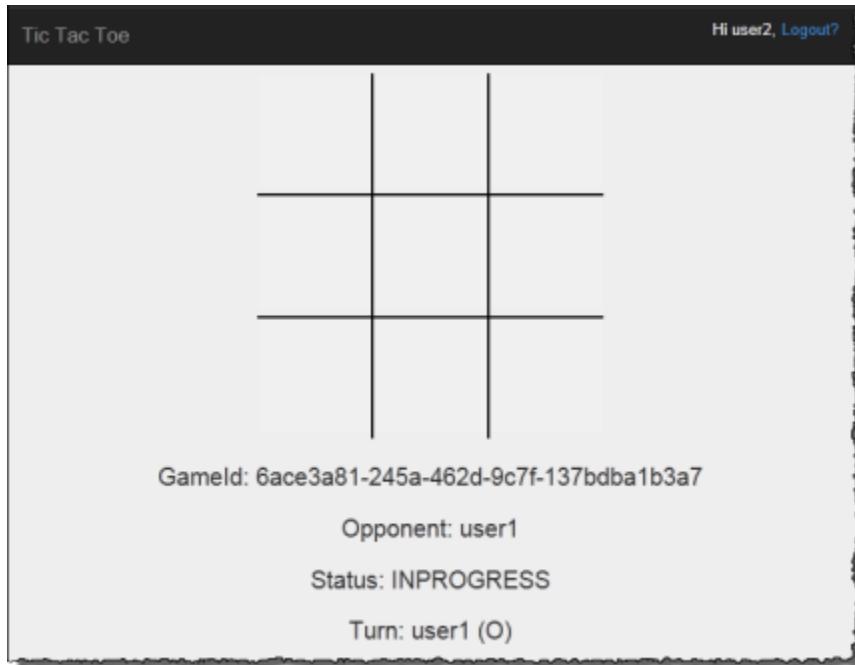
The browser passes information through cookies, so you should use incognito mode or private browsing so that your cookies don't carry over.

9. Log in as user2.

A page appears that shows a pending invitation from user1.



10. Choose **accept** to accept the invitation.



The game page appears with an empty tic-tac-toe grid. The page also shows relevant game information such as the game ID, whose turn it is, and game status.

11. Play the game.

For each user move, the web service sends a request to DynamoDB to conditionally update the game item in the Games table. For example, the conditions ensure that the move is valid, the square that the user chose is available, and that it was the turn of the user who made the move. For a valid move, the update operation adds a new attribute corresponding to the selection on the board. The update operation also sets the value of the existing attribute to the user who can make the next move.

On the game page, the application makes asynchronous JavaScript calls every second, for up to 5 minutes, to check if the game state in DynamoDB has changed. If it has, the application updates the page with new information. After 5 minutes, the application stops making the requests, and you need to refresh the page to get updated information.

Step 2: Examine the Data Model and Implementation Details

Topics

- [2.1: Basic Data Model \(p. 1002\)](#)
- [2.2: Application in Action \(Code Walkthrough\) \(p. 1005\)](#)

2.1: Basic Data Model

This example application highlights the following DynamoDB data model concepts:

- **Table** – In DynamoDB, a table is a collection of items (that is, records), and each item is a collection of name-value pairs called attributes.

In this Tic-Tac-Toe example, the application stores all game data in a table, Games. The application creates one item in the table per game and stores all game data as attributes. A tic-tac-toe game can have up to nine moves. Because DynamoDB tables do not have a schema in cases where only the primary key is the required attribute, the application can store varying number of attributes per game item.

The Games table has a simple primary key made of one attribute, GameId, of string type. The application assigns a unique ID to each game. For more information on DynamoDB primary keys, see [Primary Key \(p. 6\)](#).

When a user initiates a tic-tac-toe game by inviting another user to play, the application creates a new item in the Games table with attributes storing game metadata, such as the following:

- HostId, the user who initiated the game.
- Opponent, the user who was invited to play.
- The user whose turn it is to play. The user who initiated the game plays first.
- The user who uses the O symbol on the board. The user who initiates the games uses the O symbol.

In addition, the application creates a StatusDate concatenated attribute, marking the initial game state as PENDING. The following screenshot shows an example item as it appears in the DynamoDB console:

| Amazon DynamoDB Explore Table Games | | |
|-------------------------------------|--------|---------------------------------------|
| Item Details | | |
| Attribute | Type | Value |
| GameId (Hash Key) | String | "6ffd7f5-e293-4b4a-bacf-6ddde49ef0ae" |
| HostId | String | "user1" |
| O | String | "user1" |
| Opponent | String | "user2" |
| StatusDate | String | "PENDING_2014-07-06 21:28:02.354807" |
| Turn | String | "user1" |

As the game progresses, the application adds one attribute to the table for each game move. The attribute name is the board position, for example TopLeft or BottomRight. For example, a move might have a TopLeft attribute with the value O, a TopRight attribute with the value O, and a BottomRight attribute with the value X. The attribute value is either O or X, depending on which user made the move. For example, consider the following board.



- Concatenated value attributes** – The StatusDate attribute illustrates a concatenated value attribute. In this approach, instead of creating separate attributes to store game status (PENDING, IN_PROGRESS, and FINISHED) and date (when the last move was made), you combine them as single attribute, for example IN_PROGRESS_2014-04-30 10:20:32.

The application then uses the StatusDate attribute in creating secondary indexes by specifying StatusDate as a sort key for the index. The benefit of using the StatusDate concatenated value attribute is further illustrated in the indexes discussed next.

- Global secondary indexes** – You can use the table's primary key, GameId, to efficiently query the table to find a game item. To query the table on attributes other than the primary key attributes, DynamoDB supports the creation of secondary indexes. In this example application, you build the following two secondary indexes:

| Local Secondary Indexes | | | | | | | | | | | | | | | |
|--|-------------------|---------------------|----------------------|---------------------|---------------------|----------------------|--------------------|----------------------------------|---------------------|-------------|--|--|--|--|--|
| Index Name | Hash Key | Range Key | Projected Attributes | Index Size (Bytes)* | Item Count* | | | | | | | | | | |
| This table has no local secondary indexes. | | | | | | | | | | | | | | | |
| Global Secondary Indexes | | | | | | | | | | | | | | | |
| Index Name | Hash Key | Range Key | Projected Attributes | Status | Read Capacity Units | Write Capacity Units | Last Decrease Time | Last Increase Time | Index Size (Bytes)* | Item Count* | | | | | |
| hostStatusDate | HostId (String) | StatusDate (String) | All | Active | 20 | 20 | | Sat May 31 10:35:42 GMT-700 2014 | 20305 | 125 | | | | | |
| oppStatusDate | Opponent (String) | StatusDate (String) | All | Active | 20 | 20 | | Sat May 31 10:35:42 GMT-700 2014 | 20305 | 125 | | | | | |

- HostId-StatusDate-index.** This index has HostId as a partition key and StatusDate as a sort key. You can use this index to query on HostId, for example to find games hosted by a particular user.
- OpponentId-StatusDate-index.** This index has OpponentId as a partition key and StatusDate as a sort key. You can use this index to query on Opponent, for example to find games where a particular user is the opponent.

These indexes are called global secondary indexes because the partition key in these indexes is not the same the partition key (GameId), used in the primary key of the table.

Note that both the indexes specify StatusDate as a sort key. Doing this enables the following:

- You can query using the `BEGINS_WITH` comparison operator. For example, you can find all games with the `IN_PROGRESS` attribute hosted by a particular user. In this case, the `BEGINS_WITH` operator checks for the `StatusDate` value that begins with `IN_PROGRESS`.
- DynamoDB stores the items in the index in sorted order, by sort key value. So if all status prefixes are the same (for example, `IN_PROGRESS`), the ISO format used for the date part will have items sorted from oldest to the newest. This approach enables certain queries to be performed efficiently, for example the following:
 - Retrieve up to 10 of the most recent `IN_PROGRESS` games hosted by the user who is logged in. For this query, you specify the `HostId-StatusDate-index` index.
 - Retrieve up to 10 of the most recent `IN_PROGRESS` games where the user logged in is the opponent. For this query, you specify the `OpponentId-StatusDate-index` index.

For more information about secondary indexes, see [Improving Data Access with Secondary Indexes \(p. 496\)](#).

2.2: Application in Action (Code Walkthrough)

This application has two main pages:

- **Home page** – This page provides the user a simple login, a **CREATE** button to create a new tic-tac-toe game, a list of games in progress, game history, and any active pending game invitations.

The home page is not refreshed automatically; you must refresh the page to refresh the lists.

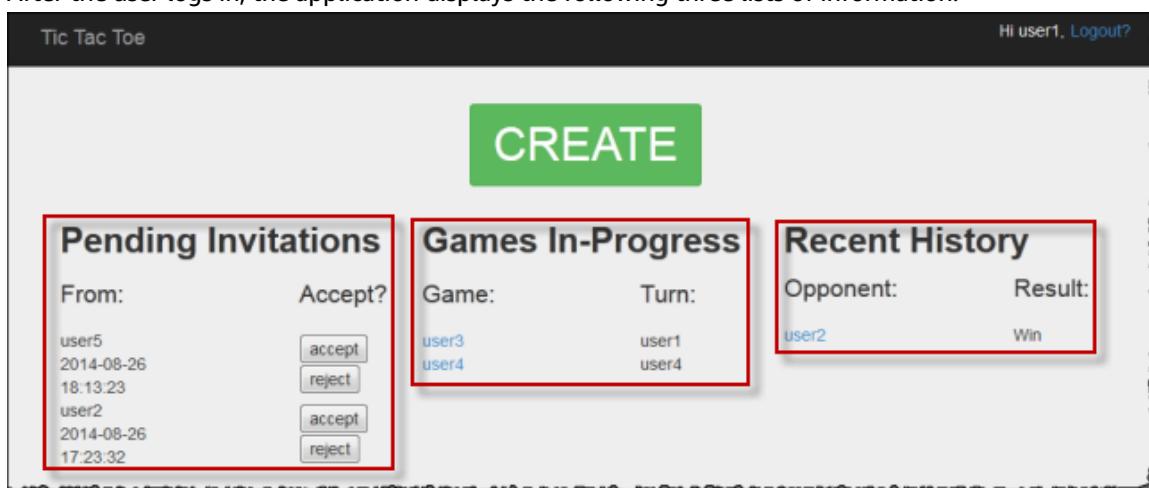
- **Game page** – This page shows the tic-tac-toe grid where users play.

The application updates the game page automatically every second. The JavaScript in your browser calls the Python web server every second to query the Games table whether the game items in the table have changed. If they have, JavaScript triggers a page refresh so that the user sees the updated board.

Let us see in detail how the application works.

Home Page

After the user logs in, the application displays the following three lists of information.



- **Invitations** – This list shows up to the 10 most recent invitations from others that are pending acceptance by the user who is logged in. In the preceding screenshot, user1 has invitations from user5 and user2 pending.
- **Games In-Progress** – This list shows up to the 10 most recent games that are in progress. These are games that the user is actively playing, which have the status IN_PROGRESS. In the screenshot, user1 is actively playing a tic-tac-toe game with user3 and user4.
- **Recent History** – This list shows up to the 10 most recent games that the user finished, which have the status FINISHED. In game shown in the screenshot, user1 has previously played with user2. For each completed game, the list shows the game result.

In the code, the index function (in application.py) makes the following three calls to retrieve game status information:

```
inviteGames      = controller.getGameInvites(session["username"])
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")
finishedGames   = controller.getGamesWithStatus(session["username"], "FINISHED")
```

Each of these calls returns a list of items from DynamoDB that are wrapped by the Game objects. It is easy to extract data from these objects in the view. The index function passes these object lists to the view to render the HTML.

```
return render_template("index.html",
                      user=session["username"],
                      invites=inviteGames,
                      inprogress=inProgressGames,
                      finished=finishedGames)
```

The Tic-Tac-Toe application defines the Game class primarily to store game data retrieved from DynamoDB. These functions return lists of Game objects that enable you to isolate the rest of the application from code related to Amazon DynamoDB items. Thus, these functions help you decouple your application code from the details of the data store layer.

The architectural pattern described here is also referred as the model-view-controller (MVC) UI pattern. In this case, the Game object instances (representing data) are the model, and the HTML page is the view. The controller is divided into two files. The application.py file has the controller logic for the Flask framework, and the business logic is isolated in the gameController.py file. That is, the application stores everything that has to do with DynamoDB SDK in its own separate file in the dynamodb folder.

Let us review the three functions and how they query the Games table using global secondary indexes to retrieve relevant data.

Using getGameInvites to Get the List of Pending Game Invitations

The getGameInvites function retrieves the list of the 10 most recent pending invitations. These games have been created by users, but the opponents have not accepted the game invitations. For these games, the status remains PENDING until the opponent accepts the invite. If the opponent declines the invite, the application removes the corresponding item from the table.

The function specifies the query as follows:

- It specifies the OpponentId-StatusDate-index index to use with the following index key values and comparison operators:
 - The partition key is OpponentId and takes the index key **user ID**.
 - The sort key is StatusDate and takes the comparison operator and index key value beginswith="PENDING_".

You use the `OpponentId-StatusDate-index` index to retrieve games to which the logged-in user is invited—that is, where the logged-in user is the opponent.

- The query limits the result to 10 items.

```
gameInvitesIndex = self.cm.getGamesTable().query(  
    Opponent__eq=user,  
    StatusDate__beginswith="PENDING_",  
    index="OpponentId-StatusDate-index",  
    limit=10)
```

In the index, for each `OpponentId` (the partition key) DynamoDB keeps items sorted by `StatusDate` (the sort key). Therefore, the games that the query returns will be the 10 most recent games.

Using `getGamesWithStatus` to Get the List of Games with a Specific Status

After an opponent accepts a game invitation, the game status changes to `IN_PROGRESS`. After the game completes, the status changes to `FINISHED`.

Queries to find games that are either in progress or finished are the same except for the different status value. Therefore, the application defines the `getGamesWithStatus` function, which takes the status value as a parameter.

```
inProgressGames = controller.getGamesWithStatus(session["username"], "IN_PROGRESS")  
finishedGames = controller.getGamesWithStatus(session["username"], "FINISHED")
```

The following section discusses in-progress games, but the same description also applies to finished games.

A list of in-progress games for a given user includes both the following:

- In-progress games hosted by the user
- In-progress games where the user is the opponent

The `getGamesWithStatus` function runs the following two queries, each time using the appropriate secondary index.

- The function queries the `Games` table using the `HostId-StatusDate-index` index. For the index, the query specifies primary key values—both the partition key (`HostId`) and sort key (`StatusDate`) values, along with comparison operators.

```
hostGamesInProgress = self.cm.getGamesTable().query(HostId__eq=user,  
    StatusDate__beginswith=status,  
    index="HostId-StatusDate-index",  
    limit=10)
```

Note the Python syntax for comparison operators:

- `HostId__eq=user` specifies the equality comparison operator.
- `StatusDate__beginswith=status` specifies the `BEGINS_WITH` comparison operator.
- The function queries the `Games` table using the `OpponentId-StatusDate-index` index.

```
oppGamesInProgress = self.cm.getGamesTable().query(Opponent__eq=user,  
    StatusDate__beginswith=status,  
    index="OpponentId-StatusDate-index",  
    limit=10)
```

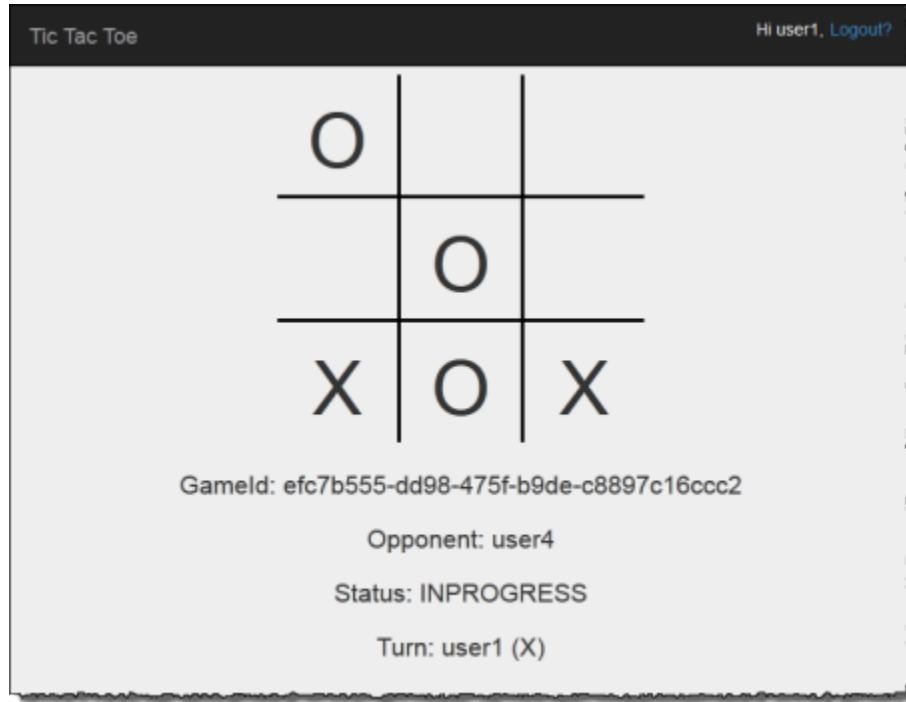
```
limit=10)
```

- The function then combines the two lists, sorts, and for the first 0 to 10 items creates a list of the Game objects and returns the list to the calling function (that is, the index).

```
games = self.mergeQueries(hostGamesInProgress,  
                           oppGamesInProgress)  
return games
```

Game Page

The game page is where the user plays tic-tac-toe games. It shows the game grid along with game-relevant information. The following screenshot shows an example game in progress:



The application displays the game page in the following situations:

- The user creates a game inviting another user to play.

In this case, the page shows the user as host and the game status as PENDING, waiting for the opponent to accept.

- The user accepts one of the pending invitations on the home page.

In this case, the page shows the user as the opponent and game status as IN_PROGRESS.

A user selection on the board generates a form POST request to the application. That is, Flask calls the `selectSquare` function (in `application.py`) with the HTML form data. This function, in turn, calls the `updateBoardAndTurn` function (in `gameController.py`) to update the game item as follows:

- It adds a new attribute specific to the move.
- It updates the `Turn` attribute value to the user whose turn is next.

```
controller.updateBoardAndTurn(item, value, session["username"])
```

The function returns true if the item update was successful; otherwise, it returns false. Note the following about the `updateBoardAndTurn` function:

- The function calls the `update_item` function of the SDK for Python to make a finite set of updates to an existing item. The function maps to the `UpdateItem` operation in DynamoDB. For more information, see [UpdateItem](#).

Note

The difference between the `UpdateItem` and `PutItem` operations is that `PutItem` replaces the entire item. For more information, see [PutItem](#).

For the `update_item` call, the code identifies the following:

- The primary key of the Games table (that is, `ItemId`).

```
key = { "GameId" : { "S" : gameId } }
```

- The new attribute to add, specific to the current user move, and its value (for example, `TopLeft="X"`).

```
attributeUpdates = {
    position : {
        "Action" : "PUT",
        "Value" : { "S" : representation }
    }
}
```

- Conditions that must be true for the update to take place:

- The game must be in progress. That is, the `StatusDate` attribute value must begin with `IN_PROGRESS`.
- The current turn must be a valid user turn as specified by the `Turn` attribute.
- The square that the user chose must be available. That is, the attribute corresponding to the square must not exist.

```
expectations = {"StatusDate" : {"AttributeValueList": [{"S" : "IN_PROGRESS_"}],
    "ComparisonOperator": "BEGINS_WITH"},
    "Turn" : {"Value" : {"S" : current_player}},
    position : {"Exists" : False}}
```

Now the function calls `update_item` to update the item.

```
self.cm.db.update_item("Games", key=key,
    attribute_updates=attributeUpdates,
    expected=expectations)
```

After the function returns, the `selectSquare` function calls `redirect`, as shown in the following example.

```
redirect("/game="+gameId)
```

This call causes the browser to refresh. As part of this refresh, the application checks to see if the game has ended in a win or draw. If it has, the application updates the game item accordingly.

Step 3: Deploy in Production Using the DynamoDB Service

Topics

- [3.1: Create an IAM Role for Amazon EC2 \(p. 1011\)](#)
- [3.2: Create the Games Table in Amazon DynamoDB \(p. 1011\)](#)
- [3.3: Bundle and Deploy the Tic-Tac-Toe Application Code \(p. 1011\)](#)
- [3.4: Set Up the AWS Elastic Beanstalk Environment \(p. 1012\)](#)

In the preceding sections, you deployed and tested the Tic-Tac-Toe application locally on your computer using DynamoDB Local. Now, you deploy the application in production as follows:

- Deploy the application using AWS Elastic Beanstalk, an easy-to-use service for deploying and scaling web applications and services. For more information, see [Deploying a Flask Application to AWS Elastic Beanstalk](#).

Elastic Beanstalk launches one or more Amazon Elastic Compute Cloud (Amazon EC2) instances, which you configure through Elastic Beanstalk, on which your Tic-Tac-Toe application will run.

- Using the Amazon DynamoDB service, create a Games table that exists on AWS rather than locally on your computer.

In addition, you also have to configure permissions. Any AWS resources you create, such as the Games table in DynamoDB, are private by default. Only the resource owner, that is the AWS account that created the Games table, can access this table. Thus, by default your Tic-Tac-Toe application cannot update the Games table.

To grant necessary permissions, you create an AWS Identity and Access Management (IAM) role and grant this role permissions to access the Games table. Your Amazon EC2 instance first assumes this role. In response, AWS returns temporary security credentials that the Amazon EC2 instance can use to update the Games table on behalf of the Tic-Tac-Toe application. When you configure your Elastic Beanstalk application, you specify the IAM role that the Amazon EC2 instance or instances can assume. For more information about IAM roles, see [IAM Roles for Amazon EC2](#) in the *Amazon EC2 User Guide for Linux Instances*.

Note

Before you create Amazon EC2 instances for the Tic-Tac-Toe application, you must first decide the AWS Region where you want Elastic Beanstalk to create the instances. After you create the Elastic Beanstalk application, you provide the same Region name and endpoint in a configuration file. The Tic-Tac-Toe application uses information in this file to create the Games table and send subsequent requests in a specific AWS Region. Both the DynamoDB Games table and the Amazon EC2 instances that Elastic Beanstalk launches must be in the same Region. For a list of available Regions, see [Amazon DynamoDB](#) in the *Amazon Web Services General Reference*.

In summary, you do the following to deploy the Tic-Tac-Toe application in production:

1. Create an IAM role using the IAM service. You attach a policy to this role granting permissions for DynamoDB actions to access the Games table.
2. Bundle the Tic-Tac-Toe application code and a configuration file, and create a .zip file. You use this .zip file to give the Tic-Tac-Toe application code to Elastic Beanstalk to put on your servers. For more information about creating a bundle, see [Creating an Application Source Bundle](#) in the *AWS Elastic Beanstalk Developer Guide*.

In the configuration file (`beanstalk.config`), you provide AWS Region and endpoint information. The Tic-Tac-Toe application uses this information to determine which DynamoDB Region to talk to.

3. Set up the Elastic Beanstalk environment. Elastic Beanstalk launches an Amazon EC2 instance or instances and deploys your Tic-Tac-Toe application bundle on them. After the Elastic Beanstalk environment is ready, you provide the configuration file name by adding the `CONFIG_FILE` environment variable.
4. Create the DynamoDB table. Using the Amazon DynamoDB service, you create the Games table on AWS, rather than locally on your computer. Remember, this table has a simple primary key made of the `GameId` partition key of string type.
5. Test the game in production.

3.1: Create an IAM Role for Amazon EC2

Creating an IAM role of the **Amazon EC2** type allows the Amazon EC2 instance that is running your Tic-Tac-Toe application to assume the correct role and make application requests to access the Games table. When creating the role, choose the **Custom Policy** option and copy and paste the following policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dynamodb>ListTables"  
            ],  
            "Effect": "Allow",  
            "Resource": "*"  
        },  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:922852403271:table/Games",  
                "arn:aws:dynamodb:us-west-2:922852403271:table/Games/index/*"  
            ]  
        }  
    ]  
}
```

For further instructions, see [Creating a Role for an AWS Service \(AWS Management Console\)](#) in the *IAM User Guide*.

3.2: Create the Games Table in Amazon DynamoDB

The Games table in DynamoDB stores game data. If the table does not exist, the application creates the table for you. In this case, let the application create the Games table.

3.3: Bundle and Deploy the Tic-Tac-Toe Application Code

If you followed this example's steps, then you already have the downloaded the Tic-Tac-Toe application. If not, download the application and extract all the files to a folder on your local computer. For instructions, see [Step 1: Deploy and Test Locally \(p. 998\)](#).

After you extract all files, you will have a code folder. To hand off this folder to Elastic Beanstalk, you bundle the contents of this folder as a `.zip` file. First, you add a configuration file to that folder. Your application uses the Region and endpoint information to create a DynamoDB table in the specified Region and make subsequent table operation requests using the specified endpoint.

1. Switch to the folder where you downloaded the Tic-Tac-Toe application.

2. In the root folder of the application, create a text file named `beanstalk.config` with the following content.

```
[dynamodb]
region=<AWS region>
endpoint=<DynamoDB endpoint>
```

For example, you might use the following content.

```
[dynamodb]
region=us-west-2
endpoint=dynamodb.us-west-2.amazonaws.com
```

For a list of available Regions, see [Amazon DynamoDB](#) in the *Amazon Web Services General Reference*.

Important

The Region specified in the configuration file is the location where the Tic-Tac-Toe application creates the Games table in DynamoDB. You must create the Elastic Beanstalk application discussed in the next section in the same Region.

Note

When you create your Elastic Beanstalk application, you request to launch an environment where you can choose the environment type. To test the Tic-Tac-Toe example application, you can choose the **Single Instance** environment type, skip the following, and go to the next step.

However, the **Load balancing, autoscaling** environment type provides a highly available and scalable environment, something you should consider when you create and deploy other applications. If you choose this environment type, you also need to generate a UUID and add it to the configuration file, as shown following.

```
[dynamodb]
region=us-west-2
endpoint=dynamodb.us-west-2.amazonaws.com
[flask]
secret_key= 284e784d-1a25-4a19-92bf-8eeb7a9example
```

In client-server communication, when the server sends a response, for security's sake the server sends a signed cookie that the client sends back to the server in the next request. When there is only one server, the server can locally generate an encryption key when it starts. When there are many servers, they all need to know the same encryption key; otherwise, they won't be able to read cookies set by the peer servers. By adding `secret_key` to the configuration file, you tell all servers to use this encryption key.

3. Zip the content of the root folder of the application (which includes the `beanstalk.config` file)—for example, `TicTacToe.zip`.
4. Upload the `.zip` file to an Amazon Simple Storage Service (Amazon S3) bucket. In the next section, you provide this `.zip` file to Elastic Beanstalk to upload on the server or servers.

For instructions on how to upload to an Amazon S3 bucket, see [Create a Bucket](#) and [Add an Object to a Bucket](#) in the *Amazon Simple Storage Service Getting Started Guide*.

3.4: Set Up the AWS Elastic Beanstalk Environment

In this step, you create an Elastic Beanstalk application, which is a collection of components including environments. For this example, you launch one Amazon EC2 instance to deploy and run your Tic-Tac-Toe application.

1. Enter the following custom URL to set up an Elastic Beanstalk console to set up the environment.

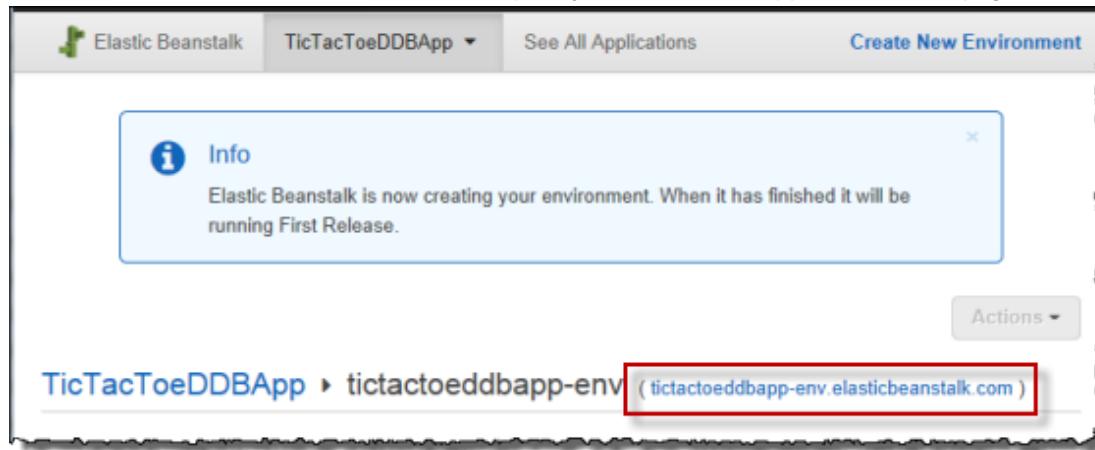
```
https://console.aws.amazon.com/elasticbeanstalk/?region=<AWS-Region>#/newApplication  
?applicationName=TicTacToe<your-name>  
&solutionStackName=Python  
&sourceBundleUrl=https://s3.amazonaws.com/<bucket-name>/TicTacToe.zip  
&environmentType=SingleInstance  
&instanceType=t1.micro
```

For more information about custom URLs, see [Constructing a Launch Now URL in the AWS Elastic Beanstalk Developer Guide](#). For the URL, note the following:

- You must provide an AWS Region name (the same as the one you provided in the configuration file), an Amazon S3 bucket name, and the object name.
- For testing, the URL requests the **SingleInstance** environment type, and **t1.micro** as the instance type.
- The application name must be unique. Thus, in the preceding URL, we suggest you prepend your name to the `applicationName`.

Doing this opens the Elastic Beanstalk console. In some cases, you might need to sign in.

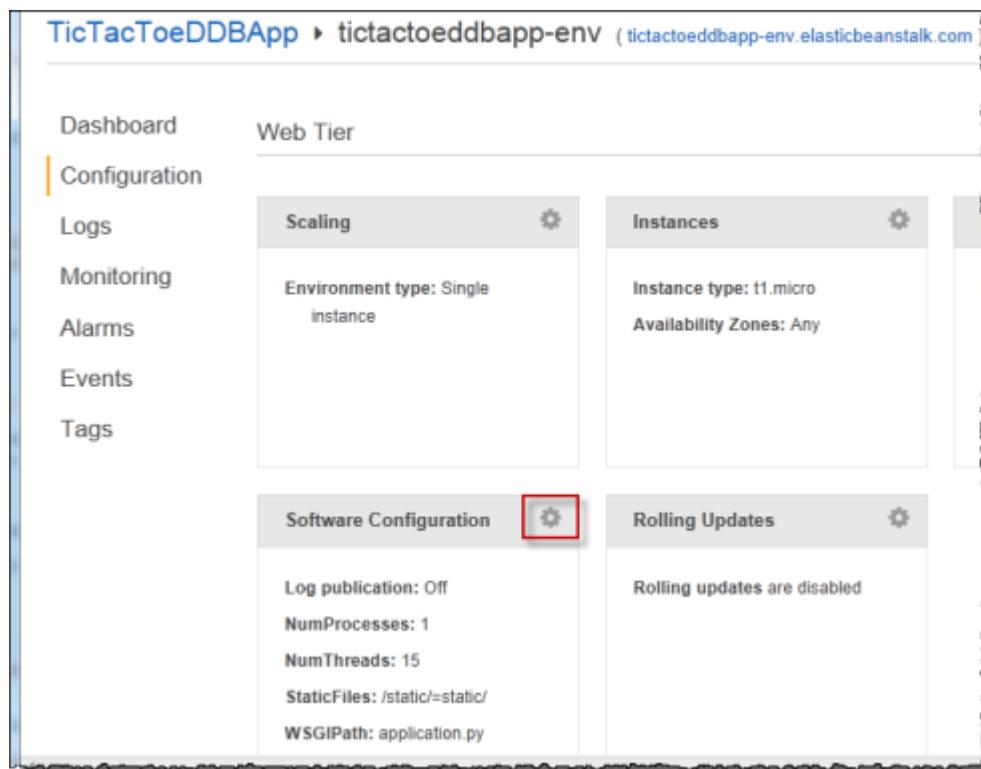
2. In the Elastic Beanstalk console, choose **Review and Launch**, and then choose **Launch**.
3. Note the URL for future reference. This URL opens your Tic-Tac-Toe application home page.



4. Configure the Tic-Tac-Toe application so it knows the location of the configuration file.

After Elastic Beanstalk creates the application, choose **Configuration**.

- a. Choose the gear icon next to **Software Configuration**, as shown in the following screenshot.



- At the end of the **Environment Properties** section, enter **CONFIG_FILE** and its value **beanstalk.config**, and then choose **Save**.

It might take a few minutes for this environment update to complete.

| | |
|---|---------------------------|
| PARAM3 A predefined environment property that will be available to your running application. | |
| PARAM4 A predefined environment property that will be available to your running application. | |
| PARAM5 A predefined environment property that will be available to your running application. | |
| CONFIG_FILE | beanstalk.config + |

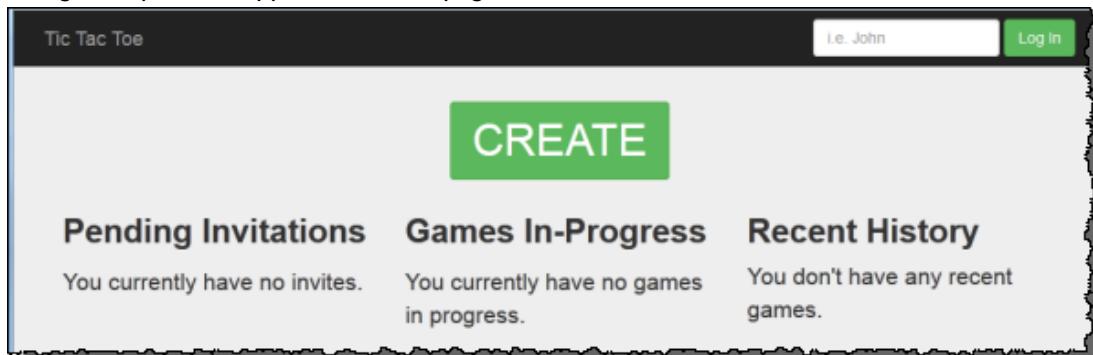
Cancel **Save**

After the update completes, you can play the game.

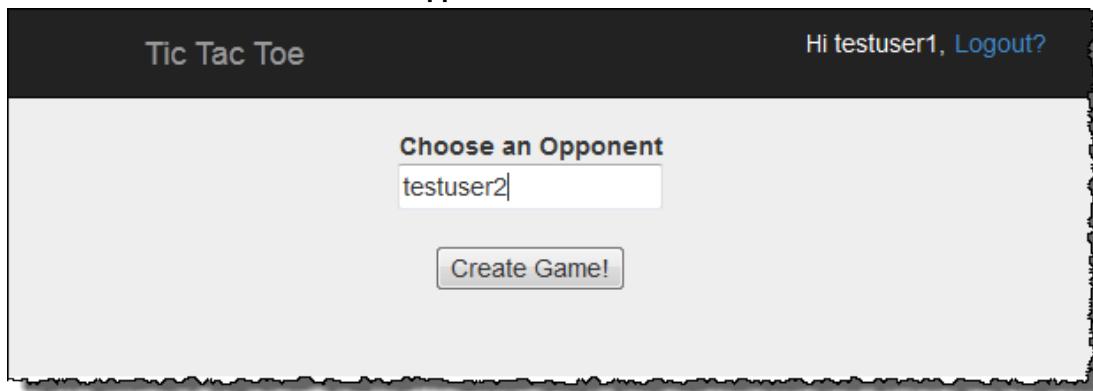
- In the browser, enter the URL you copied in the previous step, as shown in the following example.

`http://<open-name>.elasticbeanstalk.com`

Doing this opens the application home page.



6. Log in as testuser1, and choose **CREATE** to start a new tic-tac-toe game.
7. Enter **testuser2** in the **Choose an Opponent** box.



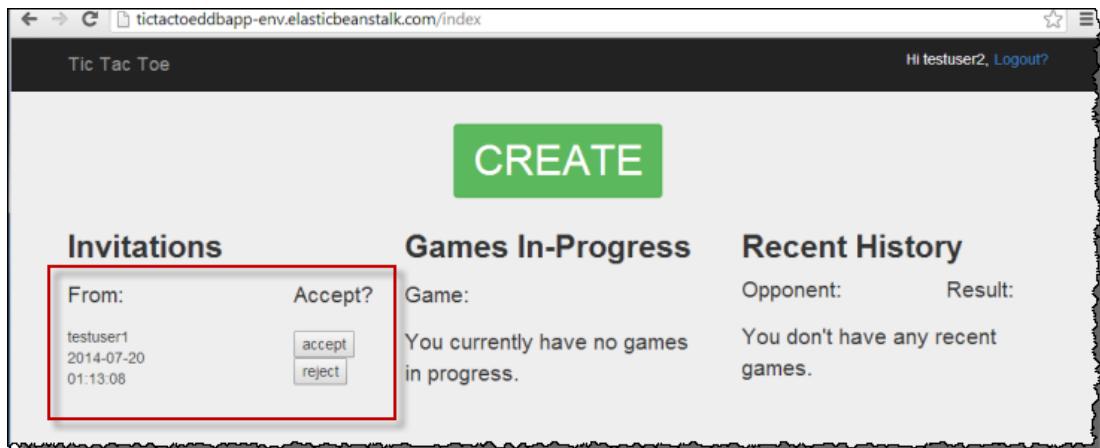
8. Open another browser window.

Make sure that you clear all cookies in your browser window so you won't be logged in as same user.

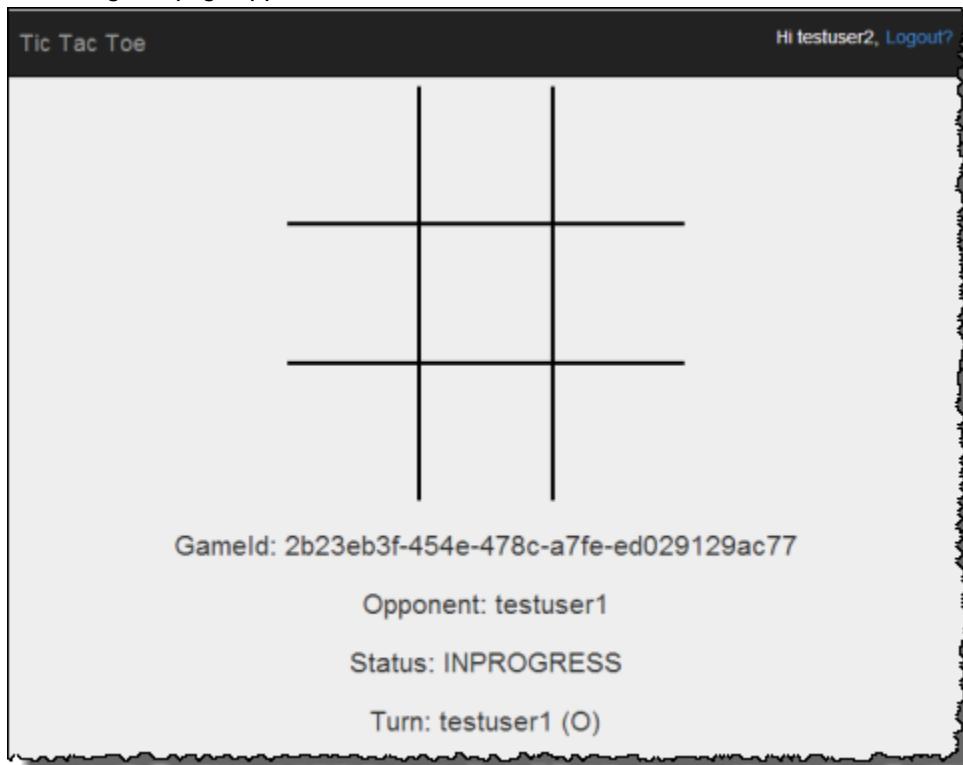
9. Enter the same URL to open the application home page, as shown in the following example.

```
http://<env-name>.elasticbeanstalk.com
```

10. Log in as testuser2.
11. For the invitation from testuser1 in the list of pending invitations, choose **accept**.



12. Now the game page appears.



Both testuser1 and testuser2 can play the game. For each move, the application saves the move in the corresponding item in the Games table.

Step 4: Clean Up Resources

Now you have completed the Tic-Tac-Toe application deployment and testing. The application covers end-to-end web application development on Amazon DynamoDB, except for user authentication. The application uses the login information on the home page only to add a player name when creating a game. In a production application, you would add the necessary code to perform user login and authentication.

If you are done testing, you can remove the resources you created to test the Tic-Tac-Toe application to avoid incurring any charges.

To remove the resources you created

1. Remove the Games table that you created in DynamoDB.
2. Terminate the Elastic Beanstalk environment to free up the Amazon EC2 instances.
3. Delete the IAM role that you created.
4. Remove the object that you created in Amazon S3.

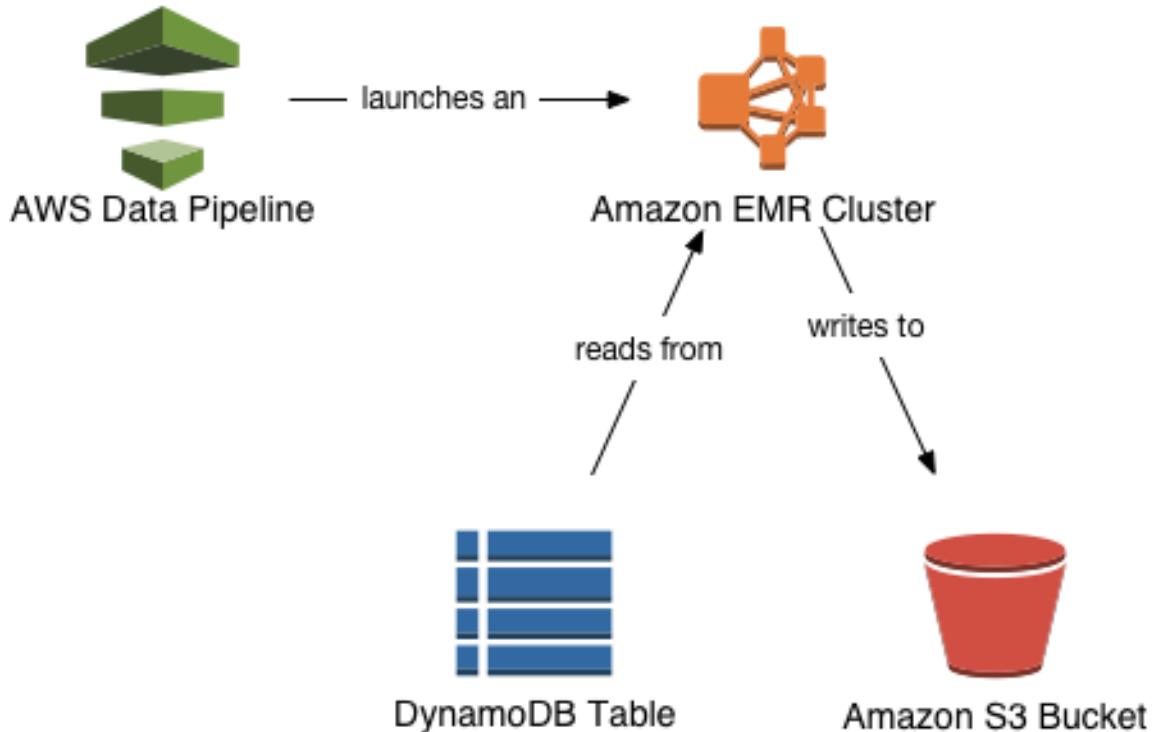
Exporting and Importing DynamoDB Data Using AWS Data Pipeline

You can use AWS Data Pipeline to export data from a DynamoDB table to a file in an Amazon S3 bucket. You can also use the console to import data from Amazon S3 into a DynamoDB table, in the same AWS region or in a different region.

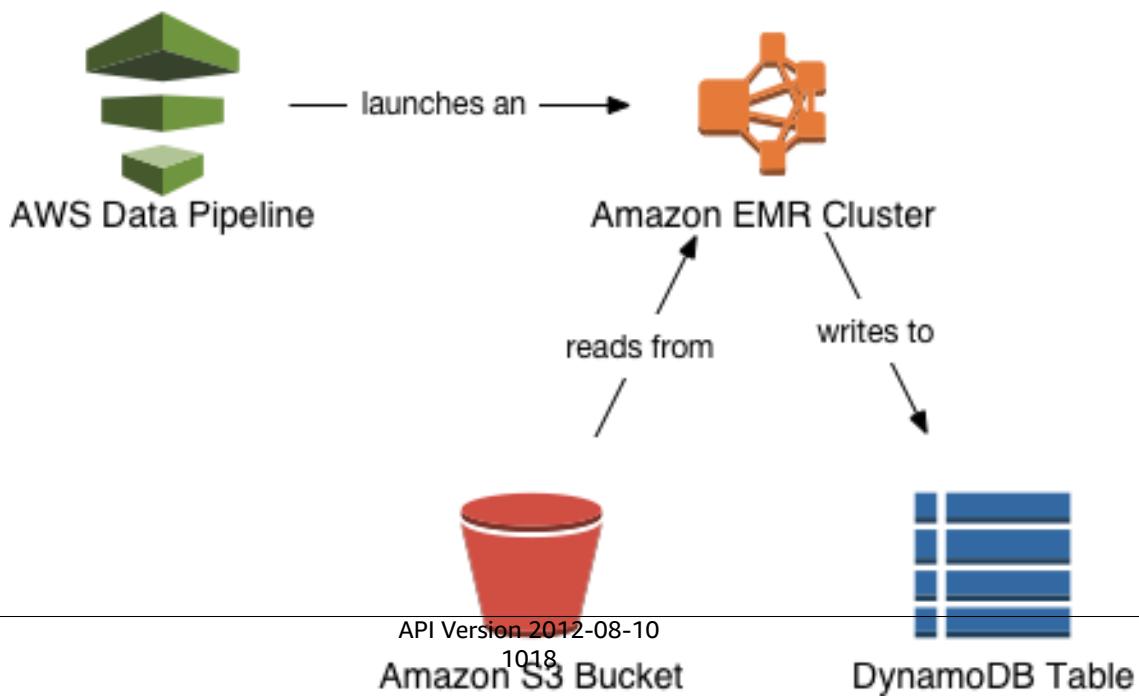
The ability to export and import data is useful in many scenarios. For example, suppose you want to maintain a baseline set of data, for testing purposes. You could put the baseline data into a DynamoDB table and export it to Amazon S3. Then, after you run an application that modifies the test data, you could "reset" the data set by importing the baseline from Amazon S3 back into the DynamoDB table. Another example involves accidental deletion of data, or even an accidental `DeleteTable` operation. In these cases, you could restore the data from a previous export file in Amazon S3. You can even copy data from a DynamoDB table in one AWS region, store the data in Amazon S3, and then import the data from Amazon S3 to an identical DynamoDB table in a second region. Applications in the second region could then access their nearest DynamoDB endpoint and work with their own copy of the data, with reduced network latency.

The following diagram shows an overview of exporting and importing DynamoDB data using AWS Data Pipeline.

Exporting Data from DynamoDB to Amazon S3



Importing Data from Amazon S3 to DynamoDB



To export a DynamoDB table, you use the AWS Data Pipeline console to create a new pipeline. The pipeline launches an Amazon EMR cluster to perform the actual export. Amazon EMR reads the data from DynamoDB, and writes the data to an export file in an Amazon S3 bucket.

The process is similar for an import, except that the data is read from the Amazon S3 bucket and written to the DynamoDB table.

Important

When you export or import DynamoDB data, you will incur additional costs for the underlying AWS services that are used:

- **AWS Data Pipeline**— manages the import/export workflow for you.
- **Amazon S3**— contains the data that you export from DynamoDB, or import into DynamoDB.
- **Amazon EMR**— runs a managed Hadoop cluster to perform reads and writes between DynamoDB to Amazon S3. The cluster configuration is one `m3.xlarge` instance master node and one `m3.xlarge` instance core node.

For more information see [AWS Data Pipeline Pricing](#), [Amazon EMR Pricing](#), and [Amazon S3 Pricing](#).

Prerequisites to Export and Import Data

When you use AWS Data Pipeline for exporting and importing data, you must specify the actions that the pipeline is allowed to perform, and which resources the pipeline can consume. The permitted actions and resources are defined using AWS Identity and Access Management (IAM) roles.

You can also control access by creating IAM policies and attaching them to IAM users or groups . These policies let you specify which users are allowed to import and export your DynamoDB data.

Important

The IAM user that performs the exports and imports must have an *active* AWS Access Key Id and Secret Key. For more information, see [Administering Access Keys for IAM Users](#) in the *IAM User Guide*.

Creating IAM Roles for AWS Data Pipeline

In order to use AWS Data Pipeline, the following IAM roles must be present in your AWS account:

- **DataPipelineDefaultRole** — the actions that your pipeline can take on your behalf.
- **DataPipelineDefaultResourceRole** — the AWS resources that the pipeline will provision on your behalf. For exporting and importing DynamoDB data, these resources include an Amazon EMR cluster and the Amazon EC2 instances associated with that cluster.

If you have never used AWS Data Pipeline before, you will need to create *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole* yourself. Once you have created these roles, you can use them any time you want to export or import DynamoDB data.

Note

If you have previously used the AWS Data Pipeline console to create a pipeline, then *DataPipelineDefaultRole* and *DataPipelineDefaultResourceRole* were created for you at that time. No further action is required; you can skip this section and begin creating pipelines using the DynamoDB console. For more information, see [Exporting Data From DynamoDB to Amazon S3 \(p. 1022\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 1023\)](#).

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Roles**.

3. Click **Create Role** and do the following:
 - a. In the **AWS Service** trusted entity, choose **Data Pipeline**.
 - b. In the **Select your use case** panel, choose **Data Pipeline** and then choose **Next:Permissions**.
 - c. Notice that the **AWSDynamoDBRole** policy is automatically attached. Choose **Next:Review**.
 - d. In the **Role name** field, type **DataPipelineDefaultRole** as the role name and choose **Create role**.
4. Click **Create Role** and do the following:
 - a. In the **AWS Service** trusted entity, choose **Data Pipeline**.
 - b. In the **Select your use case** panel, choose **EC2 Role for Data Pipeline** and then choose **Next:Permissions**.
 - c. Notice that the **AmazonEC2RoleForDataPipelineRole** policy is automatically attached. Choose **Next:Review**.
 - d. In the **Role name** field, type **DataPipelineDefaultResourceRole** as the role name and choose **Create role**.

Now that you have created these roles, you can begin creating pipelines using the DynamoDB console. For more information, see [Exporting Data From DynamoDB to Amazon S3 \(p. 1022\)](#) and [Importing Data From Amazon S3 to DynamoDB \(p. 1023\)](#).

Granting IAM Users and Groups Permission to Perform Export and Import Tasks

If you want to allow other IAM users or groups to export and import your DynamoDB table data, you can create an IAM policy and attach it to the users or groups that you designate. The policy contains only the necessary permissions for performing these tasks.

Granting Full Access Using an AWS Managed Policy

The following procedure describes how to attach the AWS managed policy **AmazonDynamoDBFullAccesswithDataPipeline** to an IAM user. This managed policy provides full access to AWS Data Pipeline and to DynamoDB resources.

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Users** and select the user you want to modify.
3. In the **Permissions** tab, click **Attach Policy**.
4. In the **Attach Policy** panel, select **AmazonDynamoDBFullAccesswithDataPipeline** and click **Attach Policy**.

Note

You can use a similar procedure to attach this managed policy to a group, rather than to a user.

Restricting Access to Particular DynamoDB Tables

If you want to restrict access so that a user can only export or import a subset of your tables, you will need to create a customized IAM policy document. You can use the AWS managed policy **AmazonDynamoDBFullAccesswithDataPipeline** as a starting point for your custom policy, and then modify the policy so that a user can only work with the tables that you specify.

For example, suppose that you want to allow an IAM user to export and import only the *Forum*, *Thread*, and *Reply* tables. This procedure describes how to create a custom policy so that a user can work with those tables, but no others.

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. From the IAM Console Dashboard, click **Policies** and then click **Create Policy**.
3. In the **Create Policy** panel, go to **Copy an AWS Managed Policy** and click **Select**.
4. In the **Copy an AWS Managed Policy** panel, go to **AmazonDynamoDBFullAccesswithDataPipeline** and click **Select**.
5. In the **Review Policy** panel, do the following:
 - a. Review the autogenerated **Policy Name** and **Description**. If you want, you can modify these values.
 - b. In the **Policy Document** text box, edit the policy to restrict access to specific tables. By default, the policy permits all DynamoDB actions on all of your tables:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "cloudwatch:DeleteAlarms",
                "cloudwatch:DescribeAlarmHistory",
                "cloudwatch:DescribeAlarms",
                "cloudwatch:DescribeAlarmsForMetric",
                "cloudwatch:GetMetricStatistics",
                "cloudwatch>ListMetrics",
                "cloudwatch:PutMetricAlarm",
                "dynamodb:*",
                "sns>CreateTopic",
                "sns>DeleteTopic",
                "sns>ListSubscriptions",
                "sns>ListSubscriptionsByTopic",
                "sns>ListTopics",
                "sns:Subscribe",
                "sns:Unsubscribe"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Sid": "DDBConsole"
        },
        ...
    ]
}
```

...remainder of document omitted...

To restrict the policy, first remove the following line:

```
"dynamodb:*
```

Next, construct a new Action that allows access to only the *Forum*, *Thread* and *Reply* tables:

```
{
    "Action": [
        "dynamodb:*
```

```
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dynamodb:us-west-2:123456789012:table/Forum",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Thread",
        "arn:aws:dynamodb:us-west-2:123456789012:table/Reply"
    ]
},
```

Note

Replace `us-west-2` with the region in which your DynamoDB tables reside. Replace `123456789012` with your AWS account number.

Finally, add the new Action to the policy document:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "dynamodb:*"  
            ],  
            "Effect": "Allow",  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Forum",  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Thread",  
                "arn:aws:dynamodb:us-west-2:123456789012:table/Reply"  
            ]  
        },  
        {  
            "Action": [  
                "cloudwatch:DeleteAlarms",  
                "cloudwatch:DescribeAlarmHistory",  
                "cloudwatch:DescribeAlarms",  
                "cloudwatch:DescribeAlarmsForMetric",  
                "cloudwatch:GetMetricStatistics",  
                "cloudwatch>ListMetrics",  
                "cloudwatch:PutMetricAlarm",  
                "sns>CreateTopic",  
                "sns>DeleteTopic",  
                "sns>ListSubscriptions",  
                "sns>ListSubscriptionsByTopic",  
                "sns>ListTopics",  
                "sns:Subscribe",  
                "sns:Unsubscribe"  
            ],  
            "Effect": "Allow",  
            "Resource": "*",  
            "Sid": "DDBConsole"  
        },  
        ...remainder of document omitted...  
    ]  
}
```

- When the policy settings are as you want them, click **Create Policy**.

After you have created the policy, you can attach it to an IAM user.

- From the IAM Console Dashboard, click **Users** and select the user you want to modify.
- In the **Permissions** tab, click **Attach Policy**.
- In the **Attach Policy** panel, select the name of your policy and click **Attach Policy**.

Note

You can use a similar procedure to attach your policy to a group, rather than to a user.

Exporting Data From DynamoDB to Amazon S3

This section describes how to export data from one or more DynamoDB tables to an Amazon S3 bucket. You need to create the Amazon S3 bucket before you can perform the export.

Important

If you have never used AWS Data Pipeline before, you will need to set up two IAM roles before following this procedure. For more information, see [Creating IAM Roles for AWS Data Pipeline \(p. 1019\)](#).

1. Sign in to the AWS Management Console and open the AWS Data Pipeline console at <https://console.aws.amazon.com/datapipeline/>.
2. If you do not already have any pipelines in the current AWS region, choose **Get started now**. Otherwise, if you already have at least one pipeline, choose **Create new pipeline**.
3. On the **Create Pipeline** page, do the following:
 - a. In the **Name** field, type a name for your pipeline. For example: `MyDynamoDBExportPipeline`.
 - b. For the **Source** parameter, select **Build using a template**. From the drop-down template list, choose **Export DynamoDB table to S3**.
 - c. In the **Source DynamoDB table name** field, type the name of the DynamoDB table that you want to export.
 - d. In the **Output S3 Folder** text box, enter an Amazon S3 URI where the export file will be written. For example: `s3://mybucket/exports`

The format of this URI is `s3://bucketname/folder` where:
 - `bucketname` is the name of your Amazon S3 bucket.
 - `folder` is the name of a folder within that bucket. If the folder does not exist, it will be created automatically. If you do not specify a name for the folder, a name will be assigned for it in the form `s3://bucketname/region tablename`.
 - e. In the **S3 location for logs** text box, enter an Amazon S3 URI where the log file for the export will be written. For example: `s3://mybucket/logs/`

The URI format for **S3 Log Folder** is the same as for **Output S3 Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.
4. When the settings are as you want them, click **Activate**.

Your pipeline will now be created; this process can take several minutes to complete. You can monitor the progress in the AWS Data Pipeline console.

When the export has finished, you can go to the [Amazon S3 console](#) to view your export file. The output file name is an identifier value with no extension, such as this example: `ae10f955-fb2f-4790-9b11-fbfea01a871e_000000`. The internal format of this file is described at [Verify Data Export File](#) in the [AWS Data Pipeline Developer Guide](#).

Importing Data From Amazon S3 to DynamoDB

This section assumes that you have already exported data from a DynamoDB table, and that the export file has been written to your Amazon S3 bucket. The internal format of this file is described at [Verify Data Export File](#) in the [AWS Data Pipeline Developer Guide](#). Note that this is the *only* file format that DynamoDB can import using AWS Data Pipeline.

We will use the term *source table* for the original table from which the data was exported, and *destination table* for the table that will receive the imported data. You can import data from an export file in Amazon S3, provided that all of the following are true:

- The destination table already exists. (The import process will not create the table for you.)
- The destination table has the same key schema as the source table.

The destination table does not have to be empty. However, the import process will replace any data items in the table that have the same keys as the items in the export file. For example, suppose you have a *Customer* table with a key of *CustomerId*, and that there are only three items in the table (*CustomerId* 1, 2, and 3). If your export file also contains data items for *CustomerID* 1, 2, and 3, the items in the destination table will be replaced with those from the export file. If the export file also contains a data item for *CustomerId* 4, then that item will be added to the table.

The destination table can be in a different AWS region. For example, suppose you have a *Customer* table in the US West (Oregon) region and export its data to Amazon S3. You could then import that data into an identical *Customer* table in the Europe (Ireland) region. This is referred to as a *cross-region* export and import. For a list of AWS regions, go to [Regions and Endpoints](#) in the *AWS General Reference*.

Note that the AWS Management Console lets you export multiple source tables at once. However, you can only import one table at a time.

1. Sign in to the AWS Management Console and open the AWS Data Pipeline console at <https://console.aws.amazon.com/datapipeline/>.
2. (Optional) If you want to perform a cross region import, go to the upper right corner of the window and choose the destination region.
3. Choose **Create new pipeline**.
4. On the **Create Pipeline** page, do the following:
 - a. In the **Name** field, type a name for your pipeline. For example: `MyDynamoDBImportPipeline`.
 - b. For the **Source** parameter, select **Build using a template**. From the drop-down template list, choose **Import DynamoDB backup data from S3**.
 - c. In the **Input S3 Folder** text box, enter an Amazon S3 URI where the export file can be found. For example: `s3://mybucket/exports`

The format of this URI is `s3://bucketname/folder` where:
 - `bucketname` is the name of your Amazon S3 bucket.
 - `folder` is the name of the folder that contains the export file.

The import job will expect to find a file at the specified Amazon S3 location. The internal format of the file is described at [Verify Data Export File](#) in the *AWS Data Pipeline Developer Guide*.

- d. In the **Target DynamoDB table name** field, type the name of the DynamoDB table into which you want to import the data.
 - e. In the **S3 location for logs** text box, enter an Amazon S3 URI where the log file for the import will be written. For example: `s3://mybucket/logs/`

The URI format for **S3 Log Folder** is the same as for **Output S3 Folder**. The URI must resolve to a folder; log files cannot be written to the top level of the S3 bucket.
5. When the settings are as you want them, click **Activate**.

Your pipeline will now be created; this process can take several minutes to complete. The import job will begin immediately after the pipeline has been created.

Troubleshooting

This section covers some basic failure modes and troubleshooting for DynamoDB exports.

If an error occurs during an export or import, the pipeline status in the AWS Data Pipeline console will display as `ERROR`. If this happens, click the name of the failed pipeline to go to its detail page. This will show details about all of the steps in the pipeline, and the status of each one. In particular, examine any execution stack traces that you see.

Finally, go to your Amazon S3 bucket and look for any export or import log files that were written there.

The following are some common issues that may cause a pipeline to fail, along with corrective actions. To diagnose your pipeline, compare the errors you have seen with the issues noted below.

- For an import, ensure that the destination table already exists, and the destination table has the same key schema as the source table. These conditions must be met, or the import will fail.
- Ensure that the Amazon S3 bucket you specified has been created, and that you have read and write permissions on it.
- The pipeline might have exceeded its execution timeout. (You set this parameter when you created the pipeline.) For example, you might have set the execution timeout for 1 hour, but the export job might have required more time than this. Try deleting and then re-creating the pipeline, but with a longer execution timeout interval this time.
- Update the manifest file if you restore from a Amazon S3 bucket that is not the original bucket that the export was performed with (contains a copy of the export).
- You might not have the correct permissions for performing an export or import. For more information, see [Prerequisites to Export and Import Data \(p. 1019\)](#).
- You might have reached a resource limit in your AWS account, such as the maximum number of Amazon EC2 instances or the maximum number of AWS Data Pipeline pipelines. For more information, including how to request increases in these limits, see [AWS Service Limits](#) in the [AWS General Reference](#).

Note

For more details on troubleshooting a pipeline, go to [Troubleshooting](#) in the [AWS Data Pipeline Developer Guide](#).

Predefined Templates for AWS Data Pipeline and DynamoDB

If you would like a deeper understanding of how AWS Data Pipeline works, we recommend that you consult the [AWS Data Pipeline Developer Guide](#). This guide contains step-by-step tutorials for creating and working with pipelines; you can use these tutorials as starting points for creating your own pipelines. We recommend that you read the DynamoDB tutorial, which walks you through the steps required to create an import and export pipeline that you can customize for your requirements. See [Tutorial: Amazon DynamoDB Import and Export Using AWS Data Pipeline](#) in the [AWS Data Pipeline Developer Guide](#).

AWS Data Pipeline offers several templates for creating pipelines; the following templates are relevant to DynamoDB.

Exporting Data Between DynamoDB and Amazon S3

The AWS Data Pipeline console provides two predefined templates for exporting data between DynamoDB and Amazon S3. For more information about these templates, see the following sections of the [AWS Data Pipeline Developer Guide](#):

- [Export DynamoDB to Amazon S3](#)
- [Export Amazon S3 to DynamoDB](#)

Amazon DynamoDB Storage Backend for Titan

The DynamoDB Storage Backend for Titan project has been superseded by the Amazon DynamoDB Storage Backend for JanusGraph, which is available on [GitHub](#).

For up-to-date instructions on the DynamoDB Storage Backend for JanusGraph, see the [README.md](#) file.

Reserved Words in DynamoDB

The following keywords are reserved for use by DynamoDB. Do not use any of these words as attribute names in expressions.

If you need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word, you can define an expression attribute name to use in the place of the reserved word. For more information, see [Expression Attribute Names in DynamoDB \(p. 389\)](#).

```
ABORT
ABSOLUTE
ACTION
ADD
AFTER
AGENT
AGGREGATE
ALL
ALLOCATE
ALTER
ANALYZE
AND
ANY
ARCHIVE
ARE
ARRAY
AS
ASC
ASCII
ASENSITIVE
ASSERTION
ASYMMETRIC
AT
ATOMIC
ATTACH
ATTRIBUTE
AUTH
AUTHORIZATION
AUTHORIZE
AUTO
AVG
BACK
BACKUP
BASE
BATCH
BEFORE
BEGIN
BETWEEN
BIGINT
BINARY
BIT
BLOB
BLOCK
BOOLEAN
BOTH
BREADTH
BUCKET
BULK
BY
BYTE
```

| |
|---------------|
| CALL |
| CALLED |
| CALLING |
| CAPACITY |
| CASCADE |
| CASCADED |
| CASE |
| CAST |
| CATALOG |
| CHAR |
| CHARACTER |
| CHECK |
| CLASS |
| CLOB |
| CLOSE |
| CLUSTER |
| CLUSTERED |
| CLUSTERING |
| CLUSTERS |
| COALESCE |
| COLLATE |
| COLLATION |
| COLLECTION |
| COLUMN |
| COLUMNS |
| COMBINE |
| COMMENT |
| COMMIT |
| COMPACT |
| COMPILE |
| COMPRESS |
| CONDITION |
| CONFLICT |
| CONNECT |
| CONNECTION |
| CONSISTENCY |
| CONSISTENT |
| CONSTRAINT |
| CONSTRAINTS |
| CONSTRUCTOR |
| CONSUMED |
| CONTINUE |
| CONVERT |
| COPY |
| CORRESPONDING |
| COUNT |
| COUNTER |
| CREATE |
| CROSS |
| CUBE |
| CURRENT |
| CURSOR |
| CYCLE |
| DATA |
| DATABASE |
| DATE |
| DATETIME |
| DAY |
| DEALLOCATE |
| DEC |
| DECIMAL |
| DECLARE |
| DEFAULT |
| DEFERRABLE |
| DEFERRED |
| DEFINE |

```
DEFINED
DEFINITION
DELETE
DELIMITED
DEPTH
DEREF
DESC
DESCRIBE
DESCRIPTOR
DETACH
DETERMINISTIC
DIAGNOSTICS
DIRECTORIES
DISABLE
DISCONNECT
DISTINCT
DISTRIBUTE
DO
DOMAIN
DOUBLE
DROP
DUMP
DURATION
DYNAMIC
EACH
ELEMENT
ELSE
ELSEIF
EMPTY
ENABLE
END
EQUAL
EQUALS
ERROR
ESCAPE
ESCAPED
EVAL
EVALUATE
EXCEEDED
EXCEPT
EXCEPTION
EXCEPTIONS
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXIT
EXPLAIN
EXPLODE
EXPORT
EXPRESSION
EXTENDED
EXTERNAL
EXTRACT
FAIL
FALSE
FAMILY
FETCH
FIELDS
FILE
FILTER
FILTERING
FINAL
FINISH
FIRST
FIXED
```

| |
|-------------|
| FLATTEN |
| FLOAT |
| FOR |
| FORCE |
| FOREIGN |
| FORMAT |
| FORWARD |
| FOUND |
| FREE |
| FROM |
| FULL |
| FUNCTION |
| FUNCTIONS |
| GENERAL |
| GENERATE |
| GET |
| GLOB |
| GLOBAL |
| GO |
| GOTO |
| GRANT |
| GREATER |
| GROUP |
| GROUPING |
| HANDLER |
| HASH |
| HAVE |
| HAVING |
| HEAP |
| HIDDEN |
| HOLD |
| HOUR |
| IDENTIFIED |
| IDENTITY |
| IF |
| IGNORE |
| IMMEDIATE |
| IMPORT |
| IN |
| INCLUDING |
| INCLUSIVE |
| INCREMENT |
| INCREMENTAL |
| INDEX |
| INDEXED |
| INDEXES |
| INDICATOR |
| INFINITE |
| INITIALLY |
| INLINE |
| INNER |
| INNER |
| INOUT |
| INPUT |
| INSENSITIVE |
| INSERT |
| INSTEAD |
| INT |
| INTEGER |
| INTERSECT |
| INTERVAL |
| INTO |
| INVALIDATE |
| IS |
| ISOLATION |
| ITEM |

```
ITEMS
ITERATE
JOIN
KEY
KEYS
LAG
LANGUAGE
LARGE
LAST
LATERAL
LEAD
LEADING
LEAVE
LEFT
LENGTH
LESS
LEVEL
LIKE
LIMIT
LIMITED
LINES
LIST
LOAD
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOCATION
LOCATOR
LOCK
LOCKS
LOG
LOGED
LONG
LOOP
LOWER
MAP
MATCH
MATERIALIZED
MAX
MAXLEN
MEMBER
MERGE
METHOD
METRICS
MIN
MINUS
MINUTE
MISSING
MOD
MODE
MODIFIES
MODIFY
MODULE
MONTH
MULTI
MULTISET
NAME
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NEXT
NO
NONE
```

```
NOT
NULL
NULLIF
NUMBER
NUMERIC
OBJECT
OF
OFFLINE
OFFSET
OLD
ON
ONLINE
ONLY
OPAQUE
OPEN
OPERATOR
OPTION
OR
ORDER
ORDINALITY
OTHER
OTHERS
OUT
OUTER
OUTPUT
OVER
OVERLAPS
OVERRIDE
OWNER
PAD
PARALLEL
PARAMETER
PARAMETERS
PARTIAL
PARTITION
PARTITIONED
PARTITIONS
PATH
PERCENT
PERCENTILE
PERMISSION
PERMISSIONS
PIPE
PIPELINED
PLAN
POOL
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROCEDURE
PROCESSED
PROJECT
PROJECTION
PROPERTY
PROVISIONING
PUBLIC
PUT
QUERY
QUIT
QUORUM
RAISE
```

```
RANDOM
RANGE
RANK
RAW
READ
READS
REAL
REBUILD
RECORD
RECURSIVE
REDUCE
REF
REFERENCE
REFERENCES
REFERENCING
REGEXP
REGION
REINDEX
RELATIVE
RELEASE
REMAINDER
RENAME
REPEAT
REPLACE
REQUEST
RESET
RESIGNAL
RESOURCE
RESPONSE
RESTORE
RESTRICT
RESULT
RETURN
RETURNING
RETURNS
REVERSE
REVOKE
RIGHT
ROLE
ROLES
ROLLBACK
ROLLUP
ROUTINE
ROW
ROWS
RULE
RULES
SAMPLE
SATISFIES
SAVE
SAVEPOINT
SCAN
SCHEMA
SCOPE
SCROLL
SEARCH
SECOND
SECTION
SEGMENT
SEGMENTS
SELECT
SELF
SEMI
SENSITIVE
SEPARATE
SEQUENCE
```

```
SERIALIZABLE
SESSION
SET
SETS
SHARD
SHARE
SHARED
SHORT
SHOW
SIGNAL
SIMILAR
SIZE
SKEWED
SMALLINT
SNAPSHOT
SOME
SOURCE
SPACE
SPACES
SPARSE
SPECIFIC
SPECIFICTYPE
SPLIT
SQL
SQLCODE
SQLERROR
SQLEXCEPTION
SQLSTATE
SQLWARNING
START
STATE
STATIC
STATUS
STORAGE
STORE
STORED
STREAM
STRING
STRUCT
STYLE
SUB
SUBMULTISET
SUBPARTITION
SUBSTRING
SUBTYPE
SUM
SUPER
SYMMETRIC
SYNONYM
SYSTEM
TABLE
TABLESAMPLE
TEMP
TEMPORARY
TERMINATED
TEXT
THAN
THEN
THROUGHPUT
TIME
TIMESTAMP
TIMEZONE
TINYINT
TO
TOKEN
TOTAL
```

| |
|-------------|
| TOUCH |
| TRAILING |
| TRANSACTION |
| TRANSFORM |
| TRANSLATE |
| TRANSLATION |
| TREAT |
| TRIGGER |
| TRIM |
| TRUE |
| TRUNCATE |
| TTL |
| TUPLE |
| TYPE |
| UNDER |
| UNDO |
| UNION |
| UNIQUE |
| UNIT |
| UNKNOWN |
| UNLOGGED |
| UNNEST |
| UNPROCESSED |
| UNSIGNED |
| UNTIL |
| UPDATE |
| UPPER |
| URL |
| USAGE |
| USE |
| USER |
| USERS |
| USING |
| UUID |
| VACUUM |
| VALUE |
| VALUED |
| VALUES |
| VARCHAR |
| VARIABLE |
| VARIANCE |
| VARINT |
| VARYING |
| VIEW |
| VIEWS |
| VIRTUAL |
| VOID |
| WAIT |
| WHEN |
| WHENEVER |
| WHERE |
| WHILE |
| WINDOW |
| WITH |
| WITHIN |
| WITHOUT |
| WORK |
| WRAPPED |
| WRITE |
| YEAR |
| ZONE |

Legacy Conditional Parameters

This section compares the legacy conditional parameters with expression parameters in DynamoDB.

With the introduction of expression parameters (see [Using Expressions in DynamoDB \(p. 385\)](#)), several older parameters have been deprecated. New applications should not use these legacy parameters, but should use expression parameters instead. (For more information, see [Using Expressions in DynamoDB \(p. 385\)](#).)

Note

DynamoDB does not allow mixing legacy conditional parameters and expression parameters in a single call. For example, calling the `Query` operation with `AttributesToGet` and `ConditionExpression` will result in an error.

The following table shows the DynamoDB APIs that still support these legacy parameters, and which expression parameter to use instead. This table can be helpful if you are considering updating your applications so that they use expression parameters instead.

| If You Use This API... | With These Legacy Parameters... | Use This Expression Parameter Instead |
|---------------------------|---------------------------------|---------------------------------------|
| <code>BatchGetItem</code> | <code>AttributesToGet</code> | <code>ProjectionExpression</code> |
| <code>DeleteItem</code> | <code>Expected</code> | <code>ConditionExpression</code> |
| <code>GetItem</code> | <code>AttributesToGet</code> | <code>ProjectionExpression</code> |
| <code>PutItem</code> | <code>Expected</code> | <code>ConditionExpression</code> |
| <code>Query</code> | <code>AttributesToGet</code> | <code>ProjectionExpression</code> |
| | <code>KeyConditions</code> | <code>KeyConditionExpression</code> |
| | <code>QueryFilter</code> | <code>FilterExpression</code> |
| <code>Scan</code> | <code>AttributesToGet</code> | <code>ProjectionExpression</code> |
| | <code>ScanFilter</code> | <code>FilterExpression</code> |
| <code>UpdateItem</code> | <code>AttributeUpdates</code> | <code>UpdateExpression</code> |
| | <code>Expected</code> | <code>ConditionExpression</code> |

The following sections provide more information about legacy conditional parameters.

Topics

- [AttributesToGet \(p. 1036\)](#)
- [AttributeUpdates \(p. 1036\)](#)
- [ConditionalOperator \(p. 1038\)](#)
- [Expected \(p. 1038\)](#)
- [KeyConditions \(p. 1041\)](#)
- [QueryFilter \(p. 1043\)](#)
- [ScanFilter \(p. 1045\)](#)
- [Writing Conditions With Legacy Parameters \(p. 1046\)](#)

AttributesToGet

`AttributesToGet` is an array of one or more attributes to retrieve from DynamoDB. If no attribute names are provided, then all attributes will be returned. If any of the requested attributes are not found, they will not appear in the result.

`AttributesToGet` allows you to retrieve attributes of type List or Map; however, it cannot retrieve individual elements within a List or a Map.

Note that `AttributesToGet` has no effect on provisioned throughput consumption. DynamoDB determines capacity units consumed based on item size, not on the amount of data that is returned to an application.

Use *ProjectionExpression* Instead

Suppose you wanted to retrieve an item from the *Music* table, but that you only wanted to return some of the attributes. You could use a `GetItem` request with an `AttributesToGet` parameter, as in this AWS CLI example:

```
aws dynamodb get-item \  
  --table-name Music \  
  --attributes-to-get '[ "Artist", "Genre" ]' \  
  --key '{  
    "Artist": { "S": "No One You Know"},  
    "SongTitle": { "S": "Call Me Today"}  
}'
```

But you could use a `ProjectionExpression` instead:

```
aws dynamodb get-item \  
  --table-name Music \  
  --projection-expression "Artist, Genre" \  
  --key '{  
    "Artist": { "S": "No One You Know"},  
    "SongTitle": { "S": "Call Me Today"}  
}'
```

AttributeUpdates

In an `UpdateItem` operation, `AttributeUpdates` are the names of attributes to be modified, the action to perform on each, and the new value for each. If you are updating an attribute that is an index key attribute for any indexes on that table, the attribute type must match the index key type defined in the `AttributesDefinition` of the table description. You can use `UpdateItem` to update any non-key attributes.

Attribute values cannot be null. String and Binary type attributes must have lengths greater than zero. Set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException` exception.

Each `AttributeUpdates` element consists of an attribute name to modify, along with the following:

- `Value` - The new value, if applicable, for this attribute.
- `Action` - A value that specifies how to perform the update. This action is only valid for an existing attribute whose data type is Number or is a set; do not use ADD for other data types.

If an item with the specified primary key is found in the table, the following values perform the following actions:

- **PUT** - Adds the specified attribute to the item. If the attribute already exists, it is replaced by the new value.
- **DELETE** - Removes the attribute and its value, if no value is specified for **DELETE**. The data type of the specified value must match the existing value's data type.

If a set of values is specified, then those values are subtracted from the old set. For example, if the attribute value was the set [a, b, c] and the **DELETE** action specifies [a, c], then the final attribute value is [b]. Specifying an empty set is an error.

- **ADD** - Adds the specified value to the item, if the attribute does not already exist. If the attribute does exist, then the behavior of **ADD** depends on the data type of the attribute:
 - If the existing attribute is a number, and if **Value** is also a number, then **Value** is mathematically added to the existing attribute. If **Value** is a negative number, then it is subtracted from the existing attribute.

Note

If you use **ADD** to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses 0 as the initial value.

Similarly, if you use **ADD** for an existing item to increment or decrement an attribute value that doesn't exist before the update, DynamoDB uses 0 as the initial value. For example, suppose that the item you want to update doesn't have an attribute named *itemcount*, but you decide to **ADD** the number 3 to this attribute anyway. DynamoDB will create the *itemcount* attribute, set its initial value to 0, and finally add 3 to it. The result will be a new *itemcount* attribute, with a value of 3.

- If the existing data type is a set, and if **Value** is also a set, then **Value** is appended to the existing set. For example, if the attribute value is the set [1, 2], and the **ADD** action specified [3], then the final attribute value is [1, 2, 3]. An error occurs if an **ADD** action is specified for a set attribute and the attribute type specified does not match the existing set type.

Both sets must have the same primitive data type. For example, if the existing data type is a set of strings, **Value** must also be a set of strings.

If no item with the specified key is found in the table, the following values perform the following actions:

- **PUT** - Causes DynamoDB to create a new item with the specified primary key, and then adds the attribute.
- **DELETE** - Nothing happens, because attributes cannot be deleted from a nonexistent item. The operation succeeds, but DynamoDB does not create a new item.
- **ADD** - Causes DynamoDB to create an item with the supplied primary key and number (or set of numbers) for the attribute value. The only data types allowed are Number and Number Set.

If you provide any attributes that are part of an index key, then the data types for those attributes must match those of the schema in the table's attribute definition.

Use *UpdateExpression* Instead

Suppose you wanted to modify an item in the *Music* table. You could use an **UpdateItem** request with an **AttributeUpdates** parameter, as in this AWS CLI example:

```
aws dynamodb update-item \
  --table-name Music \
  --key '{
    "SongTitle": {"S":"Call Me Today"},
    "Artist": {"S":"No One You Know"}
}' \
  --attribute-updates '{
    "Genre": {
      "N": 1
    }
}'
```

```
        "Action": "PUT",
        "Value": {"S":"Rock"}
    }'
```

But you could use a `UpdateExpression` instead:

```
aws dynamodb update-item \
--table-name Music \
--key '{
    "SongTitle": {"S":"Call Me Today"},
    "Artist": {"S":"No One You Know"}
}' \
--update-expression 'SET Genre = :g' \
--expression-attribute-values '{
    ":g": {"S":"Rock"}
}'
```

ConditionalOperator

A logical operator to apply to the conditions in a `Expected`, `QueryFilter` or `ScanFilter` map:

- AND - If all of the conditions evaluate to true, then the entire map evaluates to true.
- OR - If at least one of the conditions evaluate to true, then the entire map evaluates to true.

If you omit `ConditionalOperator`, then AND is the default.

The operation will succeed only if the entire map evaluates to true.

Note

This parameter does not support attributes of type List or Map.

Expected

`Expected` is a conditional block for an `UpdateItem` operation. `Expected` is a map of attribute/condition pairs. Each element of the map consists of an attribute name, a comparison operator, and one or more values. DynamoDB compares the attribute with the value(s) you supplied, using the comparison operator. For each `Expected` element, the result of the evaluation is either true or false.

If you specify more than one element in the `Expected` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the `ConditionalOperator` parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

If the `Expected` map evaluates to true, then the conditional operation succeeds; otherwise, it fails.

`Expected` contains the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the `ComparisonOperator` being used.

For type Number, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, a is greater than A, and a is greater than B.

For type Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- **ComparisonOperator** - A comparator for evaluating attributes in the **AttributeValueList**. When performing the comparison, DynamoDB uses strongly consistent reads.

The following comparison operators are available:

EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN

The following are descriptions of each comparison operator.

- **EQ** : Equal. EQ is supported for all datatypes, including lists and maps.

AttributeValueList can contain only one **AttributeValue** element of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an **AttributeValue** element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not equal `{"NS": ["6", "2", "1"]}`.

- **NE** : Not equal. NE is supported for all datatypes, including lists and maps.

AttributeValueList can contain only one **AttributeValue** of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains an **AttributeValue** of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not equal `{"NS": ["6", "2", "1"]}`.

- **LE** : Less than or equal.

AttributeValueList can contain only one **AttributeValue** element of type String, Number, or Binary (not a set type). If an item contains an **AttributeValue** element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- **LT** : Less than.

AttributeValueList can contain only one **AttributeValue** of type String, Number, or Binary (not a set type). If an item contains an **AttributeValue** element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- **GE** : Greater than or equal.

AttributeValueList can contain only one **AttributeValue** element of type String, Number, or Binary (not a set type). If an item contains an **AttributeValue** element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- **GT** : Greater than.

AttributeValueList can contain only one **AttributeValue** element of type String, Number, or Binary (not a set type). If an item contains an **AttributeValue** element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- **NOT_NULL** : The attribute exists. NOT_NULL is supported for all datatypes, including lists and maps.

Note

This operator tests for the existence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using NOT_NULL, the result is a Boolean true.

This result is because the attribute "a" exists; its data type is not relevant to the NOT_NULL comparison operator.

- **NULL** : The attribute does not exist. NULL is supported for all datatypes, including lists and maps.

Note

This operator tests for the nonexistence of an attribute, not its data type. If the data type of attribute "a" is null, and you evaluate it using `NULL`, the result is a Boolean false. This is because the attribute "a" exists; its data type is not relevant to the `NULL` comparison operator.

- `CONTAINS` : Checks for a subsequence, or value in a set.

`AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is of type String, then the operator checks for a substring match. If the target attribute of the comparison is of type Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it finds an exact match with any member of the set.

`CONTAINS` is supported for lists: When evaluating "a `CONTAINS` b", "a" can be a list; however, "b" cannot be a set, a map, or a list.

- `NOT_CONTAINS` : Checks for absence of a subsequence, or absence of a value in a set.

`AttributeValueList` can contain only one `AttributeValue` element of type String, Number, or Binary (not a set type). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operator evaluates to true if it does not find an exact match with any member of the set.

`NOT_CONTAINS` is supported for lists: When evaluating "a `NOT CONTAINS` b", "a" can be a list; however, "b" cannot be a set, a map, or a list.

- `BEGINS_WITH` : Checks for a prefix.

`AttributeValueList` can contain only one `AttributeValue` of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- `IN` : Checks for matching elements within two sets.

`AttributeValueList` can contain one or more `AttributeValue` elements of type String, Number, or Binary (not a set type). These attributes are compared against an existing set type attribute of an item. If any elements of the input set are present in the item attribute, the expression evaluates to true.

- `BETWEEN` : Greater than or equal to the first value, and less than or equal to the second value.

`AttributeValueList` must contain two `AttributeValue` elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not compare to `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

The following parameters can be used instead of `AttributeValueList` and `ComparisonOperator`:

- `Value` - A value for DynamoDB to compare with an attribute.
- `Exists` - A Boolean value that causes DynamoDB to evaluate the value before attempting the conditional operation:
 - If `Exists` is true, DynamoDB will check to see if that attribute value already exists in the table. If it is found, then the condition evaluates to true; otherwise the condition evaluate to false.

- If `Exists` is `false`, DynamoDB assumes that the attribute value does not exist in the table. If in fact the value does not exist, then the assumption is valid and the condition evaluates to true. If the value is found, despite the assumption that it does not exist, the condition evaluates to false.

Note that the default value for `Exists` is `true`.

The `Value` and `Exists` parameters are incompatible with `AttributeValueList` and `ComparisonOperator`. Note that if you use both sets of parameters at once, DynamoDB will return a `ValidationException` exception.

Note

This parameter does not support attributes of type List or Map.

Use *ConditionExpression* Instead

Suppose you wanted to modify an item in the `Music` table, but only if a certain condition was true. You could use an `UpdateItem` request with an `Expected` parameter, as in this AWS CLI example:

```
aws dynamodb update-item \
  --table-name Music \
  --key '{
    "Artist": {"S":"No One You Know"},
    "SongTitle": {"S":"Call Me Today"}
}' \
  --attribute-updates '{
    "Price": {
      "Action": "PUT",
      "Value": {"N":"1.98"}
    }
}' \
  --expected '{
    "Price": {
      "ComparisonOperator": "LE",
      "AttributeValueList": [ {"N":"2.00"} ]
    }
}'
```

But you could use a `ConditionExpression` instead:

```
aws dynamodb update-item \
  --table-name Music \
  --key '{
    "Artist": {"S":"No One You Know"},
    "SongTitle": {"S":"Call Me Today"}
}' \
  --update-expression 'SET Price = :p1' \
  --condition-expression 'Price <= :p2' \
  --expression-attribute-values '{
    ":p1": {"N":"1.98"},
    ":p2": {"N":"2.00"}
}'
```

KeyConditions

`KeyConditions` are the selection criteria for a `Query` operation. For a query on a table, you can have conditions only on the table primary key attributes. You must provide the partition key name and value as an `EQ` condition. You can optionally provide a second condition, referring to the sort key.

Note

If you don't provide a sort key condition, all of the items that match the partition key will be retrieved. If a `FilterExpression` or `QueryFilter` is present, it will be applied after the items are retrieved.

For a query on an index, you can have conditions only on the index key attributes. You must provide the index partition key name and value as an `EQ` condition. You can optionally provide a second condition, referring to the index sort key.

Each `KeyConditions` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the `ComparisonOperator` being used.

For type `Number`, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on Unicode with UTF-8 binary encoding. For example, `a` is greater than `A`, and `a` is greater than `B`.

For `Binary`, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

- `ComparisonOperator` - A comparator for evaluating attributes, for example, equals, greater than, less than, and so on.

For `KeyConditions`, only the following comparison operators are supported:

`EQ` | `LE` | `LT` | `GE` | `GT` | `BEGINS_WITH` | `BETWEEN`

The following are descriptions of these comparison operators.

- `EQ` : Equal.

`AttributeValueList` can contain only one `AttributeValue` of type `String`, `Number`, or `Binary` (not a set type). If an item contains an `AttributeValue` element of a different type than the one specified in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not equal `{"NS": ["6", "2", "1"]}`.

- `LE` : Less than or equal.

`AttributeValueList` can contain only one `AttributeValue` element of type `String`, `Number`, or `Binary` (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- `LT` : Less than.

`AttributeValueList` can contain only one `AttributeValue` of type `String`, `Number`, or `Binary` (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- `GE` : Greater than or equal.

`AttributeValueList` can contain only one `AttributeValue` element of type `String`, `Number`, or `Binary` (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- `GT` : Greater than.

`AttributeValueList` can contain only one `AttributeValue` element of type `String`, `Number`, or `Binary` (not a set type). If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not equal `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

- **BEGINS_WITH** : Checks for a prefix.

`AttributeValueList` can contain only one `AttributeValue` of type String or Binary (not a Number or a set type). The target attribute of the comparison must be of type String or Binary (not a Number or a set type).

- **BETWEEN** : Greater than or equal to the first value, and less than or equal to the second value.

`AttributeValueList` must contain two `AttributeValue` elements of the same type, either String, Number, or Binary (not a set type). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an `AttributeValue` element of a different type than the one provided in the request, the value does not match. For example, `{"S": "6"}` does not compare to `{"N": "6"}`. Also, `{"N": "6"}` does not compare to `{"NS": ["6", "2", "1"]}`.

Use `KeyConditionExpression` Instead

Suppose you wanted to retrieve several items with the same partition key from the *Music* table. You could use a `Query` request with a `KeyConditions` parameter, as in this AWS CLI example:

```
aws dynamodb query \
--table-name Music \
--key-conditions '{
    "Artist": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "No One You Know"} ]
    },
    "SongTitle": {
        "ComparisonOperator": "BETWEEN",
        "AttributeValueList": [ {"S": "A"}, {"S": "M"} ]
    }
}'
```

But you could use a `KeyConditionExpression` instead:

```
aws dynamodb query \
--table-name Music \
--key-condition-expression 'Artist = :a AND SongTitle BETWEEN :t1 AND :t2' \
--expression-attribute-values '{
    ":a": {"S": "No One You Know"},
    ":t1": {"S": "A"},
    ":t2": {"S": "M"}
}'
```

QueryFilter

In a `Query` operation, `QueryFilter` is a condition that evaluates the query results after the items are read and returns only the desired values.

This parameter does not support attributes of type List or Map.

Note

A `QueryFilter` is applied after the items have already been read; the process of filtering does not consume any additional read capacity units.

If you provide more than one condition in the `QueryFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the

[ConditionalOperator \(p. 1038\)](#) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Note that `QueryFilter` does not allow key attributes. You cannot define a filter condition on a partition key or a sort key.

Each `QueryFilter` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in `ComparisonOperator`.

For type `Number`, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, `a` is greater than `A`, and `a` is greater than `B`.

For type `Binary`, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

For information on specifying data types in JSON, see [DynamoDB Low-Level API \(p. 216\)](#).

- `ComparisonOperator` - A comparator for evaluating attributes. For example, equals, greater than, less than, etc.

The following comparison operators are available:

```
EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS |
BEGINS_WITH | IN | BETWEEN
```

Use `FilterExpression` Instead

Suppose you wanted to query the `Music` table and apply a condition to the matching items. You could use a `Query` request with a `QueryFilter` parameter, as in this AWS CLI example:

```
aws dynamodb query \
--table-name Music \
--key-conditions '{
    "Artist": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "No One You Know"} ]
    }
}' \
--query-filter '{
    "Price": {
        "ComparisonOperator": "GT",
        "AttributeValueList": [ {"N": "1.00"} ]
    }
}'
```

But you could use a `FilterExpression` instead:

```
aws dynamodb query \
--table-name Music \
--key-condition-expression 'Artist = :a' \
--filter-expression 'Price > :p' \
--expression-attribute-values '{
    ":p": {"N": "1.00"},
    ":a": {"S": "No One You Know"}
}'
```

ScanFilter

In a Scan operation, `ScanFilter` is a condition that evaluates the scan results and returns only the desired values.

Note

This parameter does not support attributes of type List or Map.

If you specify more than one condition in the `ScanFilter` map, then by default all of the conditions must evaluate to true. In other words, the conditions are ANDed together. (You can use the [ConditionalOperator \(p. 1038\)](#) parameter to OR the conditions instead. If you do this, then at least one of the conditions must evaluate to true, rather than all of them.)

Each `ScanFilter` element consists of an attribute name to compare, along with the following:

- `AttributeValueList` - One or more values to evaluate against the supplied attribute. The number of values in the list depends on the operator specified in `ComparisonOperator`.

For type Number, value comparisons are numeric.

String value comparisons for greater than, equals, or less than are based on UTF-8 binary encoding. For example, `a` is greater than `A`, and `a` is greater than `B`.

For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values.

For information on specifying data types in JSON, see [DynamoDB Low-Level API \(p. 216\)](#).

- `ComparisonOperator` - A comparator for evaluating attributes. For example, equals, greater than, less than, etc.

The following comparison operators are available:

`EQ | NE | LE | LT | GE | GT | NOT_NULL | NULL | CONTAINS | NOT_CONTAINS | BEGINS_WITH | IN | BETWEEN`

Use `FilterExpression` Instead

Suppose you wanted to scan the `Music` table and apply a condition to the matching items. You could use a Scan request with a `ScanFilter` parameter, as in this AWS CLI example:

```
aws dynamodb scan \
--table-name Music \
--scan-filter '{
    "Genre": {
        "AttributeValueList": [ {"S": "Rock"} ],
        "ComparisonOperator": "EQ"
    }
}'
```

But you could use a `FilterExpression` instead:

```
aws dynamodb scan \
--table-name Music \
--filter-expression 'Genre = :g' \
--expression-attribute-values '{
    ":g": {"S": "Rock"}
}'
```

Writing Conditions With Legacy Parameters

The following section describes how to write conditions for use with legacy parameters, such as `Expected`, `QueryFilter`, and `ScanFilter`.

Note

New applications should use expression parameters instead. For more information, see [Using Expressions in DynamoDB \(p. 385\)](#).

Simple Conditions

With attribute values, you can write conditions for comparisons against table attributes. A condition always evaluates to true or false, and consists of:

- `ComparisonOperator` — greater than, less than, equal to, and so on.
- `AttributeValueList` (optional) — attribute value(s) to compare against. Depending on the `ComparisonOperator` being used, the `AttributeValueList` might contain one, two, or more values; or it might not be present at all.

The following sections describe the various comparison operators, along with examples of how to use them in conditions.

Comparison Operators with No Attribute Values

- `NOT_NULL` - true if an attribute exists.
- `NULL` - true if an attribute does not exist.

Use these operators to check whether an attribute exists, or doesn't exist. Because there is no value to compare against, do not specify `AttributeValueList`.

Example

The following expression evaluates to true if the `Dimensions` attribute exists.

```
...  
    "Dimensions": {  
        ComparisonOperator: "NOT_NULL"  
    }  
...
```

Comparison Operators with One Attribute Value

- `EQ` - true if an attribute is equal to a value.

`AttributeValueList` can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match. For example, the string "3" is not equal to the number 3. Also, the number 3 is not equal to the number set [3, 2, 1].

- `NE` - true if an attribute is not equal to a value.

`AttributeValueList` can contain only one value of type String, Number, Binary, String Set, Number Set, or Binary Set. If an item contains a value of a different type than the one specified in the request, the value does not match.

- `LE` - true if an attribute is less than or equal to a value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains an `AttributeValue` of a different type than the one specified in the request, the value does not match.

- `LT` - true if an attribute is less than a value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- `GE` - true if an attribute is greater than or equal to a value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- `GT` - true if an attribute is greater than a value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If an item contains a value of a different type than the one specified in the request, the value does not match.

- `CONTAINS` - true if a value is present within a set, or if one value contains another.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for a substring match. If the target attribute of the comparison is Binary, then the operator looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it finds an exact match with any member of the set.

- `NOT_CONTAINS` - true if a value is *not* present within a set, or if one value does not contain another value.

`AttributeValueList` can contain only one value of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operator checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operator checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set, then the operator evaluates to true if it *does not* find an exact match with any member of the set.

- `BEGINS_WITH` - true if the first few characters of an attribute match the provided value. Do not use this operator for comparing numbers.

`AttributeValueList` can contain only one value of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).

Use these operators to compare an attribute with a value. You must specify an `AttributeValueList` consisting of a single value. For most of the operators, this value must be a scalar; however, the `EQ` and `NE` operators also support sets.

Examples

The following expressions evaluate to true if:

- A product's price is greater than 100.

```
...
    "Price": {
        ComparisonOperator: "GT",
        AttributeValueList: [ {"N":100} ]
    }
...
```

- A product category begins with "Bo".

```

...
    "ProductCategory": {
        ComparisonOperator: "BEGINS_WITH",
        AttributeValueList: [ {"S":"Bo"} ]
    }
...

```

- A product is available in either red, green, or black:

```

...
    "Color": {
        ComparisonOperator: "EQ",
        AttributeValueList: [
            {"S":"Black"}, {"S":"Red"}, {"S":"Green"} ]
    }
...

```

Note

When comparing set data types, the order of the elements does not matter. DynamoDB will return only the items with the same set of values, regardless of the order in which you specify them in your request.

Comparison Operators with Two Attribute Values

- **BETWEEN** - true if a value is between a lower bound and an upper bound, endpoints inclusive.

`AttributeValueList` must contain two elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains a value of a different type than the one specified in the request, the value does not match.

Use this operator to determine if an attribute value is within a range. The `AttributeValueList` must contain two scalar elements of the same type - String, Number, or Binary.

Example

The following expression evaluates to true if a product's price is between 100 and 200.

```

...
    "Price": {
        ComparisonOperator: "BETWEEN",
        AttributeValueList: [ {"N":"100"}, {"N":"200"} ]
    }
...

```

Comparison Operators with N Attribute Values

- **IN** - true if a value is equal to any of the values in an enumerated list. Only scalar values are supported in the list, not sets. The target attribute must be of the same type and exact value in order to match.

`AttributeValueList` can contain one or more elements of type String, Number, or Binary (not a set). These attributes are compared against an existing non-set type attribute of an item. If *any* elements of the input set are present in the item attribute, the expression evaluates to true.

`AttributeValueList` can contain one or more values of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.

Use this operator to determine whether the supplied value is within an enumerated list. You can specify any number of scalar values in `AttributeValueList`, but they all must be of the same data type.

Example

The following expression evaluates to true if the value for `Id` is 201, 203, or 205.

```
...
  "Id": {
    ComparisonOperator: "IN",
    AttributeValueList: [ {"N":"201"}, {"N":"203"}, {"N":"205"} ]
  }
...
```

Using Multiple Conditions

DynamoDB lets you combine multiple conditions to form complex expressions. You do this by providing at least two expressions, with an optional [ConditionalOperator \(p. 1038\)](#).

By default, when you specify more than one condition, *all* of the conditions must evaluate to true in order for the entire expression to evaluate to true. In other words, an implicit AND operation takes place.

Example

The following expression evaluates to true if a product is a book which has at least 600 pages. Both of the conditions must evaluate to true, since they are implicitly ANDed together.

```
...
  "ProductCategory": {
    ComparisonOperator: "EQ",
    AttributeValueList: [ {"S":"Book"} ]
  },
  "PageCount": {
    ComparisonOperator: "GE",
    AttributeValueList: [ {"N":600} ]
  }
...
```

You can use [ConditionalOperator \(p. 1038\)](#) to clarify that an AND operation will take place. The following example behaves in the same manner as the previous one.

```
...
  "ConditionalOperator" : "AND",
  "ProductCategory": {
    "ComparisonOperator": "EQ",
    "AttributeValueList": [ {"N":"Book"} ]
  },
  "PageCount": {
    "ComparisonOperator": "GE",
    "AttributeValueList": [ {"N":600} ]
  }
...
```

You can also set `ConditionalOperator` to `OR`, which means that *at least one* of the conditions must evaluate to true.

Example

The following expression evaluates to true if a product is a mountain bike, if it is a particular brand name, or if its price is greater than 100.

```

...
    "ConditionalOperator": "OR",
    "BicycleType": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "Mountain"} ]
    },
    "Brand": {
        "ComparisonOperator": "EQ",
        "AttributeValueList": [ {"S": "Brand-Company A"} ]
    },
    "Price": {
        "ComparisonOperator": "GT",
        "AttributeValueList": [ {"N": "100"} ]
    }
...

```

Note

In a complex expression, the conditions are processed in order, from the first condition to the last.

You cannot use both AND and OR in a single expression.

Other Conditional Operators

In previous releases of DynamoDB, the `Expected` parameter behaved differently for conditional writes. Each item in the `Expected` map represented an attribute name for DynamoDB to check, along with the following:

- `Value` — a value to compare against the attribute.
- `Exists` — determine whether the value exists prior to attempting the operation.

The `Value` and `Exists` options continue to be supported in DynamoDB; however, they only let you test for an equality condition, or whether an attribute exists. We recommend that you use `ComparisonOperator` and `AttributeValueList` instead, because these options let you construct a much wider range of conditions.

Example

A `DeleteItem` can check to see whether a book is no longer in publication, and only delete it if this condition is true. Here is an AWS CLI example using a legacy condition:

```

aws dynamodb delete-item \
--table-name ProductCatalog \
--key '{
    "Id": {"N": "600"}
}' \
--expected '{
    "InPublication": {
        "Exists": true,
        "Value": {"BOOL": false}
    }
}'

```

```
    }  
}'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb delete-item \  
  --table-name ProductCatalog \  
  --key '{  
    "Id": {"N": "600"}  
}' \  
  --expected '{  
    "InPublication": {  
      "ComparisonOperator": "EQ",  
      "AttributeValueList": [ {"BOOL": false} ]  
    }  
}'
```

Example

A PutItem operation can protect against overwriting an existing item with the same primary key attributes. Here is an example using a legacy condition:

```
aws dynamodb put-item \  
  --table-name ProductCatalog \  
  --item '{  
    "Id": {"N": "500"},  
    "Title": {"S": "Book 500 Title"}  
}' \  
  --expected '{  
    "Id": { "Exists": false }  
}'
```

The following example does the same thing, but does not use a legacy condition:

```
aws dynamodb put-item \  
  --table-name ProductCatalog \  
  --item '{  
    "Id": {"N": "500"},  
    "Title": {"S": "Book 500 Title"}  
}' \  
  --expected '{  
    "Id": { "ComparisonOperator": "NULL" }  
}'
```

Note

For conditions in the `Expected` map, do not use the `Value` and `Exists` options together with `ComparisonOperator` and `AttributeValueList`. If you do this, your conditional write will fail.

Previous Low-Level API Version (2011-12-05)

This section documents the operations available in the previous DynamoDB low-level API version (2011-12-05). This version of the low-level API is maintained for backward compatibility with existing applications.

New applications should use the current API version (2012-08-10). For more information, see [Low-Level API Reference \(p. 970\)](#).

Note

We recommend that you migrate your applications to the latest API version (2012-08-10), since new DynamoDB features will not be backported to the previous API version.

Topics

- [BatchGetItem \(p. 1052\)](#)
- [BatchWriteItem \(p. 1057\)](#)
- [CreateTable \(p. 1061\)](#)
- [DeleteItem \(p. 1066\)](#)
- [DeleteTable \(p. 1070\)](#)
- [DescribeTables \(p. 1073\)](#)
- [.GetItem \(p. 1076\)](#)
- [ListTables \(p. 1078\)](#)
- [PutItem \(p. 1080\)](#)
- [Query \(p. 1085\)](#)
- [Scan \(p. 1094\)](#)
- [UpdateItem \(p. 1104\)](#)
- [UpdateTable \(p. 1110\)](#)

BatchGetItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The `BatchGetItem` operation returns the attributes for multiple items from multiple tables using their primary keys. The maximum number of items that can be retrieved for a single operation is 100. Also, the number of items retrieved is constrained by a 1 MB size limit. If the response size limit is exceeded or a partial result is returned because the table's provisioned throughput is exceeded, or because of an internal processing failure, DynamoDB returns an `UnprocessedKeys` value so you can retry the operation starting with the next item to get. DynamoDB automatically adjusts the number of items returned per page to enforce this limit. For example, even if you ask to retrieve 100 items, but each individual item is 50 KB in size, the system returns 20 items and an appropriate `UnprocessedKeys` value so you can get the next page of results. If desired, your application can include its own logic to assemble the pages of results into one set.

If no items could be processed because of insufficient provisioned throughput on each of the tables involved in the request, DynamoDB returns a `ProvisionedThroughputExceededException` error.

Note

By default, `BatchGetItem` performs eventually consistent reads on every table in the request. You can set the `ConsistentRead` parameter to `true`, on a per-table basis, if you want consistent reads instead.

`BatchGetItem` fetches items in parallel to minimize response latencies.

When designing your application, keep in mind that DynamoDB does not guarantee how attributes are ordered in the returned response. Include the primary key values in the `AttributesToGet` for the items in your request to help parse the response by item.

If the requested items do not exist, nothing is returned in the response for those items. Requests for non-existent items consume the minimum read capacity units according to the type of read. For more information, see [DynamoDB Item Sizes and Formats \(p. 345\)](#).

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{"RequestItems":
  {"Table1":
    {"Keys":
      [{"HashKeyElement": {"S": "KeyValue1"}, "RangeKeyElement": {"N": "KeyValue2"}},
       {"HashKeyElement": {"S": "KeyValue3"}, "RangeKeyElement": {"N": "KeyValue4"}},
       {"HashKeyElement": {"S": "KeyValue5"}, "RangeKeyElement": {"N": "KeyValue6"}}],
      "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]},
    {"Table2":
      {"Keys":
        [{"HashKeyElement": {"S": "KeyValue4"}},
         {"HashKeyElement": {"S": "KeyValue5}}],
        "AttributesToGet": ["AttributeName4", "AttributeName5", "AttributeName6"]}
      }
    }
}
```

| Name | Description | Required |
|--------------|---|----------|
| RequestItems | A container of the table name and corresponding items to get by primary key. While requesting items, each table name can be invoked only once per operation. Type: String Default: None | Yes |
| Table | The name of the table containing the items to get. The entry is simply a string specifying an existing table with no label. Type: String Default: None | Yes |
| Table:Keys | The primary key values that define the items in the specified table. For more information about primary keys, see Primary Key (p. 6) . Type: Keys | Yes |

| Name | Description | Required |
|-----------------------|--|----------|
| Table:AttributesToGet | Array of Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result. Type: Array | No |
| Table:ConsistentRead | If set to true, then a consistent read is issued, otherwise eventually consistent is used. Type: Boolean | No |

Responses

Syntax

```

HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 855

{"Responses":
  {"Table1":
    {"Items":
      [{"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"N": "AttributeValue"}, "AttributeName3": {"SS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}, {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"S": "AttributeValue"}, "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}], "ConsumedCapacityUnits": 1}, "Table2":
    {"Items":
      [{"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"N": "AttributeValue"}, "AttributeName3": {"SS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}, {"AttributeName1": {"S": "AttributeValue"}, "AttributeName2": {"S": "AttributeValue"}, "AttributeName3": {"NS": ["AttributeValue", "AttributeValue", "AttributeValue"]}}], "ConsumedCapacityUnits": 1}
  },
  "UnprocessedKeys":
    {"Table3":
      {"Keys":
        [{"HashKeyElement": {"S": "KeyValue1"}, "RangeKeyElement": {"N": "KeyValue2"}}, {"HashKeyElement": {"S": "KeyValue3"}, "RangeKeyElement": {"N": "KeyValue4"}}, {"HashKeyElement": {"S": "KeyValue5"}, "RangeKeyElement": {"N": "KeyValue6"}]}, "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]}}
  }
}

```

| Name | Description |
|---|---|
| Responses | <p>Table names and the respective item attributes from the tables.</p> <p>Type: Map</p> |
| Table | <p>The name of the table containing the items. The entry is simply a string specifying the table with no label.</p> <p>Type: String</p> |
| Items | <p>Container for the attribute names and values meeting the operation parameters.</p> <p>Type: Map of attribute names to and their data types and values.</p> |
| ConsumedCapacityUnits | <p>The number of read capacity units consumed, for each table. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Type: Number</p> |
| UnprocessedKeys | <p>Contains an array of tables and their respective keys that were not processed with the current response, possibly due to reaching a limit on the response size. The UnprocessedKeys value is in the same form as a RequestItems parameter (so the value can be provided directly to a subsequent BatchGetItem operation). For more information, see the above RequestItems parameter.</p> <p>Type: Array</p> |
| UnprocessedKeys: Table: Keys | <p>The primary key attribute values that define the items and the attributes associated with the items. For more information about primary keys, see Primary Key (p. 6).</p> <p>Type: Array of attribute name-value pairs.</p> |
| UnprocessedKeys: Table: AttributesToGet | <p>Attribute names within the specified table. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array of attribute names.</p> |
| UnprocessedKeys: Table: ConsistentRead | <p>If set to true, then a consistent read is used for the specified table, otherwise an eventually consistent read is used.</p> <p>Type: Boolean.</p> |

Special Errors

| Error | Description |
|--|--|
| ProvisionedThroughputExceededException | Your maximum allowed provisioned throughput has been exceeded. |

Examples

The following examples show an HTTP POST request and response using the BatchGetItem operation. For examples using the AWS SDK, see [Working with Items and Attributes \(p. 374\)](#).

Sample Request

The following sample requests attributes from two different tables.

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0
content-length: 409

{"RequestItems":
  {"comp1":
    {"Keys":
      [{"HashKeyElement":{"S":"Casey"}, "RangeKeyElement":{"N":"1319509152"}},
       {"HashKeyElement":{"S":"Dave"}, "RangeKeyElement":{"N":"1319509155"}},
       {"HashKeyElement":{"S":"Riley"}, "RangeKeyElement":{"N":"1319509158"}],
      "AttributesToGet":["user", "status"]},
     "comp2":
    {"Keys":
      [{"HashKeyElement":{"S":"Julie"}}, {"HashKeyElement":{"S":"Mingus"}},
       {"AttributesToGet":["user", "friends"]]}
    }
  }
}
```

Sample Response

The following sample is the response.

```
HTTP/1.1 200 OK
x-amzn-RequestId: GTPQVRM4VJS792J1UFJTKUBVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 373
Date: Fri, 02 Sep 2011 23:07:39 GMT

{"Responses":
  {"comp1":
    {"Items":
      [{"status":{"S":"online"}, "user":{"S":"Casey"}},
       {"status":{"S":"working"}, "user":{"S":"Riley"}},
       {"status":{"S":"running"}, "user":{"S":"Dave"}},
       "ConsumedCapacityUnits":1.5},
     "comp2":
    {"Items":
      [{"friends":{"SS":["Elisabeth", "Peter"]}, "user":{"S":"Mingus"}},
       {"friends":{"SS":["Dave", "Peter"]}, "user":{"S":"Julie"}},
       "ConsumedCapacityUnits":1}
    }
  }
},
```

```
    "UnprocessedKeys":{}  
}
```

BatchWriteItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

This operation enables you to put or delete several items across multiple tables in a single call.

To upload one item, you can use `PutItem`, and to delete one item, you can use `DeleteItem`. However, when you want to upload or delete large amounts of data, such as uploading large amounts of data from Amazon EMR (Amazon EMR) or migrating data from another database in to DynamoDB, `BatchWriteItem` offers an efficient alternative.

If you use languages such as Java, you can use threads to upload items in parallel. This adds complexity in your application to handle the threads. Other languages don't support threading. For example, if you are using PHP, you must upload or delete items one at a time. In both situations, `BatchWriteItem` provides an alternative where the specified put and delete operations are processed in parallel, giving you the power of the thread pool approach without having to introduce complexity in your application.

Note that each individual put and delete specified in a `BatchWriteItem` operation costs the same in terms of consumed capacity units. However, because `BatchWriteItem` performs the specified operations in parallel, you get lower latency. Delete operations on non-existent items consume 1 write capacity unit. For more information about consumed capacity units, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

When using `BatchWriteItem`, note the following limitations:

- **Maximum operations in a single request**—You can specify a total of up to 25 put or delete operations; however, the total request size cannot exceed 1 MB (the HTTP payload).
- You can use the `BatchWriteItem` operation only to put and delete items. You cannot use it to update existing items.
- **Not an atomic operation**—Individual operations specified in a `BatchWriteItem` are atomic; however `BatchWriteItem` as a whole is a "best-effort" operation and not an atomic operation. That is, in a `BatchWriteItem` request, some operations might succeed and others might fail. The failed operations are returned in an `UnprocessedItems` field in the response. Some of these failures might be because you exceeded the provisioned throughput configured for the table or a transient failure such as a network error. You can investigate and optionally resend the requests. Typically, you call `BatchWriteItem` in a loop and in each iteration check for unprocessed items, and submit a new `BatchWriteItem` request with those unprocessed items.
- **Does not return any items**—The `BatchWriteItem` is designed for uploading large amounts of data efficiently. It does not provide some of the sophistication offered by `PutItem` and `DeleteItem`. For example, `DeleteItem` supports the `ReturnValues` field in your request body to request the deleted item in the response. The `BatchWriteItem` operation does not return any items in the response.
- Unlike `PutItem` and `DeleteItem`, `BatchWriteItem` does not allow you to specify conditions on individual write requests in the operation.
- Attribute values must not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests that have empty values will be rejected with a `ValidationException`.

DynamoDB rejects the entire batch write operation if any one of the following is true:

- If one or more tables specified in the `BatchWriteItem` request does not exist.
- If primary key attributes specified on an item in the request does not match the corresponding table's primary key schema.
- If you try to perform multiple operations on the same item in the same `BatchWriteItem` request. For example, you cannot put and delete the same item in the same `BatchWriteItem` request.
- If the total request size exceeds the 1 MB request size (the HTTP payload) limit.
- If any individual item in a batch exceeds the 64 KB item size limit.

Requests

Syntax

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
    "RequestItems" : RequestItems
}

RequestItems
{
    "TableName1" : [ Request, Request, ... ],
    "TableName2" : [ Request, Request, ... ],
    ...
}

Request ::=
PutRequest | DeleteRequest

PutRequest ::=
{
    "PutRequest" : {
        "Item" : {
            "Attribute-Name1" : Attribute-Value,
            "Attribute-Name2" : Attribute-Value,
            ...
        }
    }
}

DeleteRequest ::=
{
    "DeleteRequest" : {
        "Key" : PrimaryKey-Value
    }
}

PrimaryKey-Value ::= HashTypePK | HashAndRangeTypePK

HashTypePK ::=
{
    "HashKeyElement" : Attribute-Value
}

HashAndRangeTypePK
{
    "HashKeyElement" : Attribute-Value,
    ...
}
```

```

        "RangeKeyElement" : Attribute-Value,
    }

Attribute-Value ::= String | Numeric| Binary | StringSet | NumericSet | BinarySet

Numeric ::=
{
    "N": "Number"
}

String ::=
{
    "S": "String"
}

Binary ::=
{
    "B": "Base64 encoded binary data"
}

StringSet ::=
{
    "SS": [ "String1", "String2", ... ]
}

NumberSet ::=
{
    "NS": [ "Number1", "Number2", ... ]
}

BinarySet ::=
{
    "BS": [ "Binary1", "Binary2", ... ]
}

```

In the request body, the `RequestItems` JSON object describes the operations that you want to perform. The operations are grouped by tables. You can use `BatchWriteItem` to update or delete several items across multiple tables. For each specific write request, you must identify the type of request (`PutItem`, `DeleteItem`) followed by detail information about the operation.

- For a `PutRequest`, you provide the item, that is, a list of attributes and their values.
- For a `DeleteRequest`, you provide the primary key name and value.

Responses

Syntax

The following is the syntax of the JSON body returned in the response.

```

{
    "Responses" : ConsumedCapacityUnitsByTable
    "UnprocessedItems" : RequestItems
}

ConsumedCapacityUnitsByTable
{
    "TableName1" : { "ConsumedCapacityUnits" : NumericValue },
    "TableName2" : { "ConsumedCapacityUnits" : NumericValue },
    ...
}

```

RequestItems

This syntax is identical to the one described in the JSON syntax in the request.

Special Errors

No errors specific to this operation.

Examples

The following example shows an HTTP POST request and the response of a `BatchWriteItem` operation. The request specifies the following operations on the Reply and the Thread tables:

- Put an item and delete an item from the Reply table
- Put an item into the Thread table

For examples using the AWS SDK, see [Working with Items and Attributes \(p. 374\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.BatchGetItem
content-type: application/x-amz-json-1.0

{
    "RequestItems": {
        "Reply": [
            {
                "PutRequest": {
                    "Item": {
                        "ReplyDateTime": {
                            "S": "2012-04-03T11:04:47.034Z"
                        },
                        "Id": {
                            "S": "DynamoDB#DynamoDB Thread 5"
                        }
                    }
                }
            },
            {
                "DeleteRequest": {
                    "Key": {
                        "HashKeyElement": {
                            "S": "DynamoDB#DynamoDB Thread 4"
                        },
                        "RangeKeyElement": {
                            "S": "oops - accidental row"
                        }
                    }
                }
            ]
        ],
        "Thread": [
            {
                "PutRequest": {
                    "Item": {
                        "ForumName": {
                            "S": "DynamoDB"
                        }
                    }
                }
            }
        ]
    }
}
```

```
        "Subject":{  
            "S":"DynamoDB Thread 5"  
        }  
    }  
}  
]  
}
```

Sample Response

The following example response shows a put operation on both the Thread and Reply tables succeeded and a delete operation on the Reply table failed (for reasons such as throttling that is caused when you exceed the provisioned throughput on the table). Note the following in the JSON response:

- The `Responses` object shows one capacity unit was consumed on both the `Thread` and `Reply` tables as a result of the successful put operation on each of these tables.
- The `UnprocessedItems` object shows the unsuccessful delete operation on the `Reply` table. You can then issue a new `BatchWriteItem` call to address these unprocessed requests.

```
HTTP/1.1 200 OK  
x-amzn-RequestId: G8M9ANLOE5QA26AEUHJKJE0ASBVV4KQNSO5AEMVJF66Q9ASUAAJG  
Content-Type: application/x-amz-json-1.0  
Content-Length: 536  
Date: Thu, 05 Apr 2012 18:22:09 GMT  
  
{  
    "Responses":{  
        "Thread":{  
            "ConsumedCapacityUnits":1.0  
        },  
        "Reply":{  
            "ConsumedCapacityUnits":1.0  
        }  
    },  
    "UnprocessedItems":{  
        "Reply": [  
            {  
                "DeleteRequest":{  
                    "Key":{  
                        "HashKeyElement":{  
                            "S":"DynamoDB#DynamoDB Thread 4"  
                        },  
                        "RangeKeyElement":{  
                            "S":"oops - accidental row"  
                        }  
                    }  
                }  
            }  
        ]  
    }  
}
```

CreateTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The `CreateTable` operation adds a new table to your account.

The table name must be unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as `dynamodb.us-west-2.amazonaws.com`). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in `dynamodb.us-west-2.amazonaws.com` and one in `dynamodb.us-west-1.amazonaws.com`, they are completely independent and do not share any data.

The `CreateTable` operation triggers an asynchronous workflow to begin creating the table. DynamoDB immediately returns the state of the table (`CREATING`) until the table is in the `ACTIVE` state. Once the table is in the `ACTIVE` state, you can perform data plane operations.

Use the [DescribeTables \(p. 1073\)](#) operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0

{"TableName": "Table1",
 "KeySchema":
    {"HashKeyElement": {"AttributeName": "AttributeName1", "AttributeType": "S"},
     "RangeKeyElement": {"AttributeName": "AttributeName2", "AttributeType": "N"}},
    "ProvisionedThroughput": {"ReadCapacityUnits": 5, "WriteCapacityUnits": 10}
}
```

| Name | Description | Required |
|-----------|--|----------|
| TableName | <p>The name of the table to create. Allowed characters are a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long.</p> <p>Type: String</p> | Yes |
| KeySchema | <p>The primary key (simple or composite) structure for the table. A name-value pair for the <code>HashKeyElement</code> is required, and a name-value pair for the <code>RangeKeyElement</code> is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 6).</p> <p>Primary key element names can be between 1 and 255</p> | Yes |

| Name | Description | Required |
|---|--|----------|
| | <p>characters long with no character restrictions.</p> <p>Possible values for the AttributeType are "S" (string), "N" (numeric), or "B" (binary).</p> <p>Type: Map of HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key.</p> | |
| ProvisionedThroughput | <p>New throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits. For details, see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Note For current maximum/minimum values, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960).</p> <p>Type: Array</p> | Yes |
| ProvisionedThroughput: ReadCapacityUnits | <p>Sets the minimum number of consistent ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent ReadCapacityUnits per second provides 100 eventually consistent ReadCapacityUnits per second.</p> <p>Type: Number</p> | Yes |
| ProvisionedThroughput: WriteCapacityUnits | <p>Sets the minimum number of writeCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Type: Number</p> | Yes |

Responses

Syntax

```

HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR0OKIRLG0HVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{
    "TableDescription": {
        "CreationDateTime": 1.310506263362E9,
        "KeySchema": [
            {"HashKeyElement": {"AttributeName": "AttributeName1", "AttributeType": "S"}, "RangeKeyElement": {"AttributeName": "AttributeName2", "AttributeType": "N"}},
            {"ProvisionedThroughput": {"ReadCapacityUnits": 5, "WriteCapacityUnits": 10}, "TableName": "Table1", "TableStatus": "CREATING"}
        ]
    }
}

```

| Name | Description |
|---|---|
| TableDescription | A container for the table properties. |
| CreationDateTime | Date when the table was created in UNIX epoch time . Type: Number |
| KeySchema | The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key. |
| ProvisionedThroughput | Throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits. See Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . Type: Array |
| ProvisionedThroughput :ReadCapacityUnits | The minimum number of ReadCapacityUnits consumed per second before DynamoDB balances the load with other operations Type: Number |
| ProvisionedThroughput :WriteCapacityUnits | The minimum number of ReadCapacityUnits consumed per second before |

| Name | Description |
|-------------|---|
| | WriteCapacityUnits. balances the load with other operations Type: Number |
| TableName | The name of the created table. Type: String |
| TableStatus | The current state of the table (CREATING). Once the table is in the ACTIVE state, you can put data in it. Use the DescribeTables (p. 1073) API to check the status of the table. Type: String |

Special Errors

| Error | Description |
|------------------------|---|
| ResourceInUseException | Attempt to recreate an already existing table. |
| LimitExceededException | The number of simultaneous table requests (cumulative number of tables in the CREATING, DELETING or UPDATING state) exceeds the maximum allowed. Note For current maximum/minimum values, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960) . |

Examples

The following example creates a table with a composite primary key containing a string and a number. For examples using the AWS SDK, see [Working with Tables and Data in DynamoDB \(p. 335\)](#).

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.CreateTable
content-type: application/x-amz-json-1.0

{
    "TableName": "comp-table",
    "KeySchema": [
        {"HashKeyElement": {"AttributeName": "user", "AttributeType": "S"}, "RangeKeyElement": {"AttributeName": "time", "AttributeType": "N"}},
        {"ProvisionedThroughput": {"ReadCapacityUnits": 5, "WriteCapacityUnits": 10}}
}
```

```
}
```

Sample Response

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"TableDescription":
  {"CreationDateTime":1.310506263362E9,
  "KeySchema":
    {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
     "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
  "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":10},
  "TableName":"comp-table",
  "TableStatus":"CREATING"
}
}
```

Related Actions

- [DescribeTables \(p. 1073\)](#)
- [DeleteTable \(p. 1070\)](#)

DeleteItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Deletes a single item in a table by primary key. You can perform a conditional delete operation that deletes the item if it exists, or if it has an expected attribute value.

Note

If you specify `DeleteItem` without attributes or values, all the attributes for the item are deleted.

Unless you specify conditions, the `DeleteItem` is an idempotent operation; running it multiple times on the same item or attribute does *not* result in an error response.

Conditional deletes are useful for only deleting items and attributes if specific conditions are met. If the conditions are met, DynamoDB performs the delete. Otherwise, the item is not deleted.

You can perform the expected conditional check on one attribute per operation.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
```

```
x-amz-target: DynamoDB_20111205.DeleteItem
content-type: application/x-amz-json-1.0

{
    "TableName": "Table1",
    "Key": {
        "HashKeyElement": {"S": "AttributeValue1"}, "RangeKeyElement": {"N": "AttributeValue2"}},
        "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3"}}, "ReturnValues": "ALL_OLD"}
}
```

| Name | Description | Required |
|---|---|----------|
| TableName | The name of the table containing the item to delete. Type: String | Yes |
| Key | The primary key that defines the item. For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value. | Yes |
| Expected | Designates an attribute for a conditional delete. The Expected parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute has a particular value before deleting it. Type: Map of attribute names. | No |
| Expected:AttributeName | The name of the attribute for the conditional put. Type: String | No |
| Expected:AttributeName:ExpectedAttributeValue | Use this parameter to specify whether or not a value already exists for the attribute name-value pair. The following JSON notation deletes the item if the "Color" attribute doesn't exist for that item: <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <pre>"Expected" : {"Color": {"Exists": false}}</pre> </div> The following JSON notation checks to see if the attribute with name "Color" has an | No |

| Name | Description | Required |
|--------------|---|----------|
| | <p>existing value of "Yellow" before deleting the item:</p> <pre>"Expected" : {"Color":{"Exists":true}, {"Value":{"S":"Yellow"}}}</pre> <p>By default, if you use the <code>Expected</code> parameter and provide a <code>Value</code>, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"Color":{"Value":{"S":"Yellow"}}}</pre> <p>Note If you specify <code>{"Exists":true}</code> without an attribute value to check, DynamoDB returns an error.</p> | |
| ReturnValues | <p>Use this parameter if you want to get the attribute name-value pairs before they were deleted. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>. If <code>ALL_OLD</code> is specified, the content of the old item is returned. If this parameter is not provided or is <code>NONE</code>, nothing is returned.</p> <p>Type: String</p> | No |

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 21:31:03 GMT

{"Attributes":
  {"AttributeName3":{"SS":["AttributeValue3","AttributeValue4","AttributeValue5"]}},
```

```

    "AttributeName2": {"S": "AttributeValue2"},  

    "AttributeName1": {"N": "AttributeValue1"}  

  },  

  "ConsumedCapacityUnits": 1  

}

```

| Name | Description |
|-----------------------|--|
| Attributes | If the <code>ReturnValues</code> parameter is provided as <code>ALL_OLD</code> in the request, DynamoDB returns an array of attribute name-value pairs (essentially, the deleted item). Otherwise, the response contains an empty set. Type: Array of attribute name-value pairs. |
| ConsumedCapacityUnits | The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Delete requests on non-existent items consume 1 write capacity unit. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . Type: Number |

Special Errors

| Error | Description |
|--|--|
| <code>ConditionalCheckFailedException</code> | Conditional check failed. An expected attribute value was not found. |

Examples

Sample Request

```

// This header is abbreviated.  

// For a sample of a complete header, see DynamoDB Low-Level API.  

POST / HTTP/1.1  

x-amz-target: DynamoDB_20111205.DeleteItem  

content-type: application/x-amz-json-1.0

{"TableName": "comp-table",  

 "Key":  

   { "HashKeyElement": {"S": "Mingus"}, "RangeKeyElement": {"N": "200"}},  

 "Expected":  

   { "status": {"Value": {"S": "shopping"}},  

    "ReturnValues": "ALL_OLD"  

}

```

Sample Response

```
HTTP/1.1 200 OK
```

```
x-amzn-RequestId: U9809LI6BBFJA5N2ROTBOP017JVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 353
Date: Tue, 12 Jul 2011 22:31:23 GMT

{"Attributes":
  {"friends": {"SS": ["Dooley", "Ben", "Daisy"]},
   "status": {"S": "shopping"},
   "time": {"N": "200"},
   "user": {"S": "Mingus"}
  },
  "ConsumedCapacityUnits": 1
}
```

Related Actions

- [PutItem \(p. 1080\)](#)

DeleteTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The DeleteTable operation deletes a table and all of its items. After a DeleteTable request, the specified table is in the DELETING state until DynamoDB completes the deletion. If the table is in the ACTIVE state, you can delete it. If a table is in CREATING or UPDATING states, then DynamoDB returns a ResourceInUseException error. If the specified table does not exist, DynamoDB returns a ResourceNotFoundException. If table is already in the DELETING state, no error is returned.

Note

DynamoDB might continue to accept data plane operation requests, such as `GetItem` and `PutItem`, on a table in the DELETING state until the table deletion is complete.

Tables are unique among those associated with the AWS Account issuing the request, and the AWS region that receives the request (such as `dynamodb.us-west-1.amazonaws.com`). Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in `dynamodb.us-west-2.amazonaws.com` and one in `dynamodb.us-west-1.amazonaws.com`, they are completely independent and do not share any data; deleting one does not delete the other.

Use the [DescribeTables \(p. 1073\)](#) operation to check the status of the table.

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0

{"TableName": "Table1"}
```

| Name | Description | Required |
|-----------|--|----------|
| TableName | The name of the table to delete. Type: String | Yes |

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 311
Date: Sun, 14 Aug 2011 22:56:22 GMT

{
    "TableDescription": {
        "CreationDateTime": 1.313362508446E9,
        "KeySchema": [
            {"HashKeyElement": {"AttributeName": "user", "AttributeType": "S"}, "RangeKeyElement": {"AttributeName": "time", "AttributeType": "N"}},
            {"ProvisionedThroughput": {"ReadCapacityUnits": 10, "WriteCapacityUnits": 10}, "TableName": "Table1", "TableStatus": "DELETING"}
        ]
    }
}
```

| Name | Description |
|--|---|
| TableDescription | A container for the table properties. |
| CreationDateTime | Date when the table was created. Type: Number |
| KeySchema | The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key. |
| ProvisionedThroughput | Throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits. See Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . |
| ProvisionedThroughput: ReadCapacityUnits | The minimum number of ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. |

| Name | Description |
|--|--|
| | Type: Number |
| ProvisionedThroughput: WriteCapacityUnits | The minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations. Type: Number |
| TableName | The name of the deleted table. Type: String |
| TableStatus | The current state of the table (DELETING). Once the table is deleted, subsequent requests for the table return resource not found. Use the DescribeTables (p. 1073) operation to check the status of the table. Type: String |

Special Errors

| Error | Description |
|------------------------|--|
| ResourceInUseException | Table is in state CREATING or UPDATING and can't be deleted. |

Examples

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DeleteTable
content-type: application/x-amz-json-1.0
content-length: 40

{"TableName": "favorite-movies-table"}
```

Sample Response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 4HONCKIVH1BFUDQ1U68CTG3N27VV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 160
Date: Sun, 14 Aug 2011 17:20:03 GMT

{"TableDescription":
  {"CreationDateTime": 1.313362508446E9,
   "KeySchema":
```

```
    {"HashKeyElement":{"AttributeName":"name","AttributeType":"S"}},  
    "TableName":"favorite-movies-table",  
    "TableStatus":"DELETING"  
}
```

Related Actions

- [CreateTable \(p. 1061\)](#)
- [DescribeTables \(p. 1073\)](#)

DescribeTables

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Returns information about the table, including the current status of the table, the primary key schema and when the table was created. DescribeTable results are eventually consistent. If you use DescribeTable too early in the process of creating a table, DynamoDB returns a `ResourceNotFoundException`. If you use DescribeTable too early in the process of updating a table, the new values might not be immediately available.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.DescribeTable  
content-type: application/x-amz-json-1.0  
  
{ "TableName": "Table1" }
```

| Name | Description | Required |
|-----------|--|----------|
| TableName | The name of the table to describe. Type: String | Yes |

Responses

Syntax

```
HTTP/1.1 200  
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375  
content-type: application/x-amz-json-1.0
```

Content-Length: 543

```
{"Table":  
    {"CreationDateTime":1.309988345372E9,  
     ItemCount:1,  
     "KeySchema":  
         {"HashKeyElement":{"AttributeName":"AttributeName1","AttributeType":"S"},  
          "RangeKeyElement":{"AttributeName":"AttributeName2","AttributeType":"N"}},  
     "ProvisionedThroughput":{"LastIncreaseDateTime": Date, "LastDecreaseDateTime": Date,  
     "ReadCapacityUnits":10,"WriteCapacityUnits":10},  
     "TableName":"Table1",  
     "TableSizeBytes":1,  
     "TableStatus":"ACTIVE"  
    }  
}
```

| Name | Description |
|-----------------------|--|
| Table | Container for the table being described. Type: String |
| CreationDateTime | Date when the table was created in UNIX epoch time . |
| ItemCount | Number of items in the specified table. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number |
| KeySchema | The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary Key (p. 6) . |
| ProvisionedThroughput | Throughput for the specified table, consisting of values for LastIncreaseDateTime (if applicable), LastDecreaseDateTime (if applicable), ReadCapacityUnits and WriteCapacityUnits. If the throughput for the table has never been increased or decreased, DynamoDB does not return values for those elements. See Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . |
| TableName | The name of the requested table. Type: String |

| Name | Description |
|----------------|--|
| TableSizeBytes | Total size of the specified table, in bytes. DynamoDB updates this value approximately every six hours. Recent changes might not be reflected in this value. Type: Number |
| TableStatus | The current state of the table (CREATING, ACTIVE, DELETING or UPDATING). Once the table is in the ACTIVE state, you can add data. |

Special Errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the DescribeTable operation for a table named "comp-table". The table has a composite primary key.

Sample Request

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.DescribeTable
content-type: application/x-amz-json-1.0

{"TableName": "users"}
```

Sample Response

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 543

{"Table":
  {"CreationDateTime": 1.309988345372E9,
   "ItemCount": 23,
   "KeySchema":
     {"HashKeyElement": {"AttributeName": "user", "AttributeType": "S"},
      "RangeKeyElement": {"AttributeName": "time", "AttributeType": "N"}},
   "ProvisionedThroughput": {"LastIncreaseDateTime": 1.309988345384E9,
    "ReadCapacityUnits": 10, "WriteCapacityUnits": 10},
   "TableName": "users",
   "TableSizeBytes": 949,
   "TableStatus": "ACTIVE"
  }
}
```

Related Actions

- [CreateTable \(p. 1061\)](#)
- [DeleteTable \(p. 1070\)](#)

- [ListTables \(p. 1078\)](#)

GetItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The GetItem operation returns a set of Attributes for an item that matches the primary key. If there is no matching item, GetItem does not return any data.

The GetItem operation provides an eventually consistent read by default. If eventually consistent reads are not acceptable for your application, use ConsistentRead. Although this operation might take longer than a standard read, it always returns the last updated value. For more information, see [Read Consistency \(p. 16\)](#).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
 "Key": {
     "HashKeyElement": {"S":"AttributeValue1"},
     "RangeKeyElement": {"N":"AttributeValue2"}
 },
 "AttributesToGet":["AttributeName3","AttributeName4"],
 "ConsistentRead":Boolean
}
```

| Name | Description | Required |
|-----------------|--|----------|
| TableName | The name of the table containing the requested item. Type: String | Yes |
| Key | The primary key values that define the item. For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value. | Yes |
| AttributesToGet | Array of Attribute names. If attribute names are not specified then all attributes will | No |

| Name | Description | Required |
|----------------|---|----------|
| | <p>be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p> | |
| ConsistentRead | <p>If set to <code>true</code>, then a consistent read is issued, otherwise eventually consistent is used.</p> <p>Type: Boolean</p> | No |

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 144

{"Item": {
    "AttributeName3": {"S": "AttributeValue3"},
    "AttributeName4": {"N": "AttributeValue4"},
    "AttributeName5": {"B": "dmFsdWU="}
},
"ConsumedCapacityUnits": 0.5
}
```

| Name | Description |
|-----------------------|--|
| Item | <p>Contains the requested attributes.</p> <p>Type: Map of attribute name-value pairs.</p> |
| ConsumedCapacityUnits | <p>The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. Requests for non-existent items consume the minimum read capacity units, depending on the type of read. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Type: Number</p> |

Special Errors

No errors specific to this operation.

Examples

For examples using the AWS SDK, see [Working with Items and Attributes \(p. 374\)](#).

Sample Request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.GetItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "comptable",  
 "Key":  
     {"HashKeyElement": {"S": "Julie"},  
      "RangeKeyElement": {"N": "1307654345"}},  
 "AttributesToGet": ["status", "friends"],  
 "ConsistentRead": true  
}
```

Sample Response

Notice the ConsumedCapacityUnits value is 1, because the optional parameter ConsistentRead is set to true. If ConsistentRead is set to false (or not specified) for the same request, the response is eventually consistent and the ConsumedCapacityUnits value would be 0.5.

```
HTTP/1.1 200  
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375  
content-type: application/x-amz-json-1.0  
content-length: 72  
  
{"Item":  
    {"friends": {"SS": ["Lynda, Aaron"]},  
     "status": {"S": "online"}  
    },  
 "ConsumedCapacityUnits": 1  
}
```

ListTables

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Returns an array of all the tables associated with the current account and endpoint.

Each DynamoDB endpoint is entirely independent. For example, if you have two tables called "MyTable," one in dynamodb.us-west-2.amazonaws.com and one in dynamodb.us-east-1.amazonaws.com, they are completely independent and do not share any data. The ListTables operation returns all of the table names associated with the account making the request, for the endpoint that receives the request.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.
```

```
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.ListTables
content-type: application/x-amz-json-1.0

{"ExclusiveStartTableName": "Table1", "Limit": 3}
```

The ListTables operation, by default, requests all of the table names associated with the account making the request, for the endpoint that receives the request.

| Name | Description | Required |
|-------------------------|---|----------|
| Limit | A number of maximum table names to return. Type: Integer | No |
| ExclusiveStartTableName | The name of the table that starts the list. If you already ran a ListTables operation and received an LastEvaluatedTableName value in the response, use that value here to continue the list. Type: String | No |

Responses

Syntax

```
HTTP/1.1 200 OK
x-amzn-RequestId: S1LEK2DPQP8OJNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 81
Date: Fri, 21 Oct 2011 20:35:38 GMT

{"TableNames": ["Table1", "Table2", "Table3"], "LastEvaluatedTableName": "Table3"}
```

| Name | Description |
|------------------------|---|
| TableNames | The names of the tables associated with the current account at the current endpoint. Type: Array |
| LastEvaluatedTableName | The name of the last table in the current list, only if some tables for the account and endpoint have not been returned. This value does not exist in a response if all table names are already returned. Use this value as the ExclusiveStartTableName in a new request to continue the list until all the table names are returned. Type: String |

Special Errors

No errors are specific to this operation.

Examples

The following examples show an HTTP POST request and response using the ListTables operation.

Sample Request

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.ListTables  
content-type: application/x-amz-json-1.0  
  
{"ExclusiveStartTableName":"comp2", "Limit":3}
```

Sample Response

```
HTTP/1.1 200 OK  
x-amzn-RequestId: S1LEK2DPQP80JNHVHL8OU2M7KRVV4KQNSO5AEMVJF66Q9ASUAAJG  
content-type: application/x-amz-json-1.0  
content-length: 81  
Date: Fri, 21 Oct 2011 20:35:38 GMT  
  
{"LastEvaluatedTableName": "comp5", "TableNames": [ "comp3", "comp4", "comp5" ] }
```

Related Actions

- [DescribeTables](#) (p. 1073)
- [CreateTable](#) (p. 1061)
- [DeleteTable](#) (p. 1070)

PutItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Creates a new item, or replaces an old item with a new item (including all the attributes). If an item already exists in the specified table with the same primary key, the new item completely replaces the existing item. You can perform a conditional put (insert a new item if one with the specified primary key doesn't exist), or replace an existing item if it has certain attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

Note

To ensure that a new item does not replace an existing item, use a conditional put operation with `Exists` set to `false` for the primary key attribute, or attributes.

For more information about using `PutItem`, see [Working with Items and Attributes \(p. 374\)](#).

Requests

Syntax

```
// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName":"Table1",
 "Item": {
     "AttributeName1": {"S": "AttributeValue1"},
     "AttributeName2": {"N": "AttributeValue2"},
     "AttributeName5": {"B": "dmFsdWU="}
 },
 "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue"}, "Exists": Boolean}},
 "ReturnValues": "ReturnValuesConstant"}
```

| Name | Description | Required |
|------------------------|---|----------|
| TableName | The name of the table to contain the item. Type: String | Yes |
| Item | A map of the attributes for the item, and must include the primary key values that define the item. Other attribute name-value pairs can be provided for the item. For more information about primary keys, see Primary Key (p. 6) . Type: Map of attribute names to attribute values. | Yes |
| Expected | Designates an attribute for a conditional put. The <code>Expected</code> parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it. Type: Map of an attribute names to an attribute value, and whether it exists. | No |
| Expected:AttributeName | The name of the attribute for the conditional put. Type: String | No |

| Name | Description | Required |
|---|--|----------|
| Expected:AttributeName: ExpectedAttributeValue | <p>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.</p> <p>The following JSON notation replaces the item if the "Color" attribute doesn't already exist for that item:</p> <pre>"Expected" : {"Color":{"Exists":false}}</pre> <p>The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before replacing the item:</p> <pre>"Expected" : {"Color":{"Exists":true, "Value":{"S":"Yellow"}}}</pre> <p>By default, if you use the <code>Expected</code> parameter and provide a <code>Value</code>, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify <code>{"Exists":true}</code>, because it is implied. You can shorten the request to:</p> <pre>"Expected" : {"Color":{"Value":{"S":"Yellow"}}}</pre> <p>Note If you specify <code>{"Exists":true}</code> without an attribute value to check, DynamoDB returns an error.</p> | No |

| Name | Description | Required |
|--------------|--|----------|
| ReturnValues | <p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the PutItem request. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>. If <code>ALL_OLD</code> is specified, and PutItem overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is <code>NONE</code>, nothing is returned.</p> <p>Type: String</p> | No |

Responses

Syntax

The following syntax example assumes the request specified a `ReturnValues` parameter of `ALL_OLD`; otherwise, the response has only the `ConsumedCapacityUnits` element.

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 85

{"Attributes":
 {"AttributeName3":{"S":"AttributeValue3"},
 "AttributeName2":{"SS":"AttributeValue2"},
 "AttributeName1":{"SS":"AttributeValue1"},
 },
 "ConsumedCapacityUnits":1
}
```

| Name | Description |
|-----------------------|---|
| Attributes | <p>Attribute values before the put operation, but only if the <code>ReturnValues</code> parameter is specified as <code>ALL_OLD</code> in the request.</p> <p>Type: Map of attribute name-value pairs.</p> |
| ConsumedCapacityUnits | <p>The number of write capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Type: Number</p> |

Special Errors

| Error | Description |
|---------------------------------|--|
| ConditionalCheckFailedException | Conditional check failed. An expected attribute value was not found. |
| ResourceNotFoundException | The specified item or attribute was not found. |

Examples

For examples using the AWS SDK, see [Working with Items and Attributes \(p. 374\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.PutItem
content-type: application/x-amz-json-1.0

{"TableName": "comp5",
 "Item":
  {"time": {"N": "300"},
   "feeling": {"S": "not surprised"},
   "user": {"S": "Riley"}
  },
 "Expected":
  {"feeling": {"Value": {"S": "surprised"}, "Exists": true}}
 "ReturnValues": "ALL_OLD"
}
```

Sample Response

```
HTTP/1.1 200
x-amzn-RequestId: 8952fa74-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 84

{"Attributes":
 {"feeling": {"S": "surprised"},
  "time": {"N": "300"},
  "user": {"S": "Riley"}},
 "ConsumedCapacityUnits": 1
}
```

Related Actions

- [UpdateItem \(p. 1104\)](#)
- [DeleteItem \(p. 1066\)](#)
- [GetItem \(p. 1076\)](#)
- [BatchGetItem \(p. 1052\)](#)

Query

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

A Query operation gets the values of one or more items and their attributes by primary key (Query is only available for hash-and-range primary key tables). You must provide a specific HashKeyValue, and can narrow the scope of the query using comparison operators on the RangeKeyValue of the primary key. Use the ScanIndexForward parameter to get results in forward or reverse order by range key.

Queries that do not return results consume the minimum read capacity units according to the type of read.

Note

If the total number of items meeting the query parameters exceeds the 1MB limit, the query stops and results are returned to the user with a LastEvaluatedKey to continue the query in a subsequent operation. Unlike a Scan operation, a Query operation never returns an empty result set *and* a LastEvaluatedKey. The LastEvaluatedKey is only provided if the results exceed 1MB, or if you have used the Limit parameter.

The result can be set for a consistent read using the ConsistentRead parameter.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.Query  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",  
 "Limit": 2,  
 "ConsistentRead": true,  
 "HashKeyValue": {"S": "AttributeValue1"},  
 "RangeKeyCondition": {"AttributeValueList":  
 [{"N": "AttributeValue2"}], "ComparisonOperator": "GT"},  
 "ScanIndexForward": true,  
 "ExclusiveStartKey": {  
 "HashKeyElement": {"S": "AttributeName1"},  
 "RangeKeyElement": {"N": "AttributeName2"}  
 },  
 "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]}  
}
```

| Name | Description | Required |
|-----------------|--|----------|
| TableName | The name of the table containing the requested items. Type: String | Yes |
| AttributesToGet | Array of Attribute names. If attribute names are not specified then all attributes will | No |

| Name | Description | Required |
|----------------|---|----------|
| | <p>be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p> | |
| Limit | <p>The maximum number of items to return (not necessarily the number of matching items). If DynamoDB processes the number of items up to the limit while querying the table, it stops the query and returns the matching values up to that point, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue the query. Also, if the result set size exceeds 1MB before DynamoDB hits this limit, it stops the query and returns the matching values, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue the query.</p> <p>Type: Number</p> | No |
| ConsistentRead | <p>If set to <code>true</code>, then a consistent read is issued, otherwise eventually consistent is used.</p> <p>Type: Boolean</p> | No |
| Count | <p>If set to <code>true</code>, DynamoDB returns a total number of items that match the query parameters, instead of a list of the matching items and their attributes. You can apply the <code>Limit</code> parameter to count-only queries.</p> <p>Do not set <code>Count</code> to <code>true</code> while providing a list of <code>AttributesToGet</code>; otherwise, DynamoDB returns a validation error. For more information, see Counting the Items in the Results (p. 462).</p> <p>Type: Boolean</p> | No |

| Name | Description | Required |
|--|--|----------|
| HashKeyValue | <p>Attribute value of the hash component of the composite primary key.</p> <p>Type: String, Number, or Binary</p> | Yes |
| RangeKeyCondition | <p>A container for the attribute values and comparison operators to use for the query. A query request does not require a RangeKeyCondition. If you provide only the HashKeyValue, DynamoDB returns all items with the specified hash key element value.</p> <p>Type: Map</p> | No |
| RangeKeyCondition: AttributeValueList | <p>The attribute values to evaluate for the query parameters. The AttributeValueList contains one attribute value, unless a BETWEEN comparison is specified. For the BETWEEN comparison, the AttributeValueList contains two attribute values.</p> <p>Type: A map of AttributeValue to a ComparisonOperator.</p> | No |

| Name | Description | Required |
|---------------------------------------|---|----------|
| RangeKeyCondition: ComparisonOperator | <p>The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a Query operation.</p> <p>Note String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p> <p>Type: String or Binary</p> | No |
| | <p>EQ : Equal. For EQ, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S" : "6" } does not equal { "N" : "6" }. Also, { "N" : "6" } does not equal { "NS" : ["6", "2", "1"] }.</p> | |

| Name | Description | Required |
|------|--|----------|
| | <p>LE : Less than or equal.</p> <p>For LE, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not compare to <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |
| | <p>LT : Less than.</p> <p>For LT, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not compare to <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |
| | <p>GE : Greater than or equal.</p> <p>For GE, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not compare to <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |

| Name | Description | Required |
|------|---|----------|
| | <p>GT : Greater than.</p> <p>For GT, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S": "6" } does not equal { "N": "6" }. Also, { "N": "6" } does not compare to { "NS": ["6", "2", "1"] }.</p> | |
| | <p>BEGINS_WITH : checks for a prefix.</p> <p>For BEGINS_WITH, AttributeValueList can contain only one AttributeValue of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).</p> | |
| | <p>BETWEEN : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For BETWEEN, AttributeValueList must contain two AttributeValue elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S": "6" } does not compare to { "N": "6" }. Also, { "N": "6" } does not compare to { "NS": ["6", "2", "1"] }.</p> | |

| Name | Description | Required |
|-------------------|--|----------|
| ScanIndexForward | <p>Specifies ascending or descending traversal of the index. DynamoDB returns results reflecting the requested order determined by the range key: If the data type is Number, the results are returned in numeric order; otherwise, the traversal is based on ASCII character code values.</p> <p>Type: Boolean</p> <p>Default is <code>true</code> (ascending).</p> | No |
| ExclusiveStartKey | <p>Primary key of the item from which to continue an earlier query. An earlier query might provide this value as the <code>LastEvaluatedKey</code> if that query operation was interrupted before completing the query; either because of the result set size or the <code>Limit</code> parameter. The <code>LastEvaluatedKey</code> can be passed back in a new query request to continue the operation from that point.</p> <p>Type: <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.</p> | No |

Responses

Syntax

```

HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items":[{"AttributeNames1":{"S":"AttributeValue1"}, "AttributeNames2":{"N":"AttributeValue2"}, "AttributeNames3":{"S":"AttributeValue3"}}, {"AttributeNames1":{"S":"AttributeValue3"}, "AttributeNames2":{"N":"AttributeValue4"}, "AttributeNames3":{"S":"AttributeValue3"}, "AttributeNames5":{"B":"dmFsdWU="}}], "LastEvaluatedKey":{"HashKeyElement":{"AttributeValue3":"S"}, "RangeKeyElement":{"AttributeValue4":"N"}}
}

```

```
    "ConsumedCapacityUnits":1
}
```

| Name | Description |
|-----------------------|---|
| Items | Item attributes meeting the query parameters. Type: Map of attribute names to and their data types and values. |
| Count | Number of items in the response. For more information, see Counting the Items in the Results (p. 462) . Type: Number |
| LastEvaluatedKey | Primary key of the item where the query operation stopped, inclusive of the previous result set. Use this value to start a new operation excluding this value in the new request. The <code>LastEvaluatedKey</code> is null when the entire query result set is complete (i.e. the operation processed the “last page”). Type: <code>HashKeyElement</code> , or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key. |
| ConsumedCapacityUnits | The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . Type: Number |

Special Errors

| Error | Description |
|---------------------------|------------------------------------|
| ResourceNotFoundException | The specified table was not found. |

Examples

For examples using the AWS SDK, see [Working with Queries in DynamoDB \(p. 458\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
```

```
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"John"},
 "ScanIndexForward":false,
 "ExclusiveStartKey": {
   "HashKeyElement":{"S":"John"}, 
   "RangeKeyElement":{"S":"The Matrix"}
 }
}
```

Sample Response

```
HTTP/1.1 200
x-amzn-RequestId: 3647e778-71eb-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 308

{"Count":2,"Items": [
  {"fans":{"SS":["Jody","Jake"]}, 
  "name":{"S":"John"}, 
  "rating":{"S":"***"}, 
  "title":{"S":"The End"} 
  },
  {"fans":{"SS":["Jody","Jake"]}, 
  "name":{"S":"John"}, 
  "rating":{"S":"***"}, 
  "title":{"S":"The Beatles"} 
  ],
  "LastEvaluatedKey": {"HashKeyElement":{"S":"John"}, "RangeKeyElement":{"S":"The Beatles"}},
  "ConsumedCapacityUnits":1
}
```

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Query
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable",
 "Limit":2,
 "HashKeyValue":{"S":"Airplane"}, 
 "RangeKeyCondition": {"AttributeValueList": [{"N":"1980"}], "ComparisonOperator": "EQ"}, 
 "ScanIndexForward":false}
```

Sample Response

```
HTTP/1.1 200
x-amzn-RequestId: 8b9ee1ad-774c-11e0-9172-d954e38f553a
content-type: application/x-amz-json-1.0
content-length: 119

{"Count":1,"Items": [
  {"fans":{"SS":["Dave","Aaron"]}, 
  "name":{"S":"Airplane"}, 
  "rating":{"S":"***"}, 
  ]}
```

```

    "year": {"N": "1980"}
  }],
  "ConsumedCapacityUnits": 1
}

```

Related Actions

- Scan (p. 1094)

Scan

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

The Scan operation returns one or more items and its attributes by performing a full scan of a table. Provide a ScanFilter to get more specific results.

Note

If the total number of scanned items exceeds the 1MB limit, the scan stops and results are returned to the user with a LastEvaluatedKey to continue the scan in a subsequent operation. The results also include the number of items exceeding the limit. A scan can result in no table data meeting the filter criteria.

The result set is eventually consistent.

Requests

Syntax

```

// This header is abbreviated.
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName": "Table1",
 "Limit": 2,
 "ScanFilter": {
   "AttributeName1": {"AttributeValueList": [
     {"S": "AttributeValue"}], "ComparisonOperator": "EQ"}
 },
 "ExclusiveStartKey": {
   "HashKeyElement": {"S": "AttributeName1"},
   "RangeKeyElement": {"N": "AttributeName2"}
 },
 "AttributesToGet": ["AttributeName1", "AttributeName2", "AttributeName3"]
}

```

| Name | Description | Required |
|-----------|---|----------|
| TableName | The name of the table containing the requested items. Type: String | Yes |

| Name | Description | Required |
|-----------------|---|----------|
| AttributesToGet | <p>Array of Attribute names. If attribute names are not specified then all attributes will be returned. If some attributes are not found, they will not appear in the result.</p> <p>Type: Array</p> | No |
| Limit | <p>The maximum number of items to evaluate (not necessarily the number of matching items). If DynamoDB processes the number of items up to the limit while processing the results, it stops and returns the matching values up to that point, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue retrieving items. Also, if the scanned data set size exceeds 1MB before DynamoDB reaches this limit, it stops the scan and returns the matching values up to the limit, and a <code>LastEvaluatedKey</code> to apply in a subsequent operation to continue the scan.</p> <p>Type: Number</p> | No |
| Count | <p>If set to <code>true</code>, DynamoDB returns a total number of items for the Scan operation, even if the operation has no matching items for the assigned filter. You can apply the Limit parameter to count-only scans.</p> <p>Do not set Count to <code>true</code> while providing a list of AttributesToGet, otherwise DynamoDB returns a validation error. For more information, see Counting the Items in the Results (p. 478).</p> <p>Type: Boolean</p> | No |

| Name | Description | Required |
|-------------------------------|---|----------|
| ScanFilter | <p>Evaluates the scan results and returns only the desired values. Multiple conditions are treated as "AND" operations: all conditions must be met to be included in the results.</p> <p>Type: A map of attribute names to values with comparison operators.</p> | No |
| ScanFilter:AttributeValueList | <p>The values and conditions to evaluate the scan results for the filter.</p> <p>Type: A map of AttributeValue to a Condition.</p> | No |
| ScanFilter:ComparisonOperator | <p>The criteria for evaluating the provided attributes, such as equals, greater-than, etc. The following are valid comparison operators for a scan operation.</p> <p>Note String value comparisons for greater than, equals, or less than are based on ASCII character code values. For example, a is greater than A, and aa is greater than B. For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters. For Binary, DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example when evaluating query expressions.</p> <p>Type: String or Binary</p> | No |

| Name | Description | Required |
|------|---|----------|
| | <p>EQ : Equal.</p> <p>For EQ, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not equal <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |
| | <p>NE : Not Equal.</p> <p>For NE, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not equal <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |
| | <p>LE : Less than or equal.</p> <p>For LE, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, <code>{ "S": "6" }</code> does not equal <code>{ "N": "6" }</code>. Also, <code>{ "N": "6" }</code> does not compare to <code>{ "NS": ["6", "2", "1"] }</code>.</p> | |

| Name | Description | Required |
|------|--|----------|
| | <p>LT : Less than.</p> <p>For LT, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S": "6" } does not equal { "N": "6" }. Also, { "N": "6" } does not compare to { "NS": ["6", "2", "1"] }.</p> | |
| | <p>GE : Greater than or equal.</p> <p>For GE, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S": "6" } does not equal { "N": "6" }. Also, { "N": "6" } does not compare to { "NS": ["6", "2", "1"] }.</p> | |
| | <p>GT : Greater than.</p> <p>For GT, AttributeValueList can contain only one AttributeValue of type String, Number, or Binary (not a set). If an item contains an AttributeValue of a different type than the one specified in the request, the value does not match. For example, { "S": "6" } does not equal { "N": "6" }. Also, { "N": "6" } does not compare to { "NS": ["6", "2", "1"] }.</p> | |
| | NOT_NULL : Attribute exists. | |
| | NULL : Attribute does not exist. | |

| Name | Description | Required |
|------|--|----------|
| | <p>CONTAINS : checks for a subsequence, or value in a set.</p> <p>For CONTAINS, AttributeValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for a substring match. If the target attribute of the comparison is Binary, then the operation looks for a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operation checks for a member of the set (not as a substring).</p> | |
| | <p>NOT_CONTAINS : checks for absence of a subsequence, or absence of a value in a set.</p> <p>For NOT_CONTAINS, AttributeValueList can contain only oneAttributeValue of type String, Number, or Binary (not a set). If the target attribute of the comparison is a String, then the operation checks for the absence of a substring match. If the target attribute of the comparison is Binary, then the operation checks for the absence of a subsequence of the target that matches the input. If the target attribute of the comparison is a set ("SS", "NS", or "BS"), then the operation checks for the absence of a member of the set (not as a substring).</p> | |

| Name | Description | Required |
|------|--|----------|
| | <p><code>BEGINS_WITH</code> : checks for a prefix.</p> <p>For <code>BEGINS_WITH</code>, <code>AttributeValueList</code> can contain only one <code>AttributeValue</code> of type String or Binary (not a Number or a set). The target attribute of the comparison must be a String or Binary (not a Number or a set).</p> | |
| | <p><code>IN</code> : checks for exact matches.</p> <p>For <code>IN</code>, <code>AttributeValueList</code> can contain more than one <code>AttributeValue</code> of type String, Number, or Binary (not a set). The target attribute of the comparison must be of the same type and exact value to match. A String never matches a String set.</p> | |
| | <p><code>BETWEEN</code> : Greater than, or equal to, the first value and less than, or equal to, the second value.</p> <p>For <code>BETWEEN</code>, <code>AttributeValueList</code> must contain two <code>AttributeValue</code> elements of the same type, either String, Number, or Binary (not a set). A target attribute matches if the target value is greater than, or equal to, the first element and less than, or equal to, the second element. If an item contains an <code>AttributeValue</code> of a different type than the one specified in the request, the value does not match. For example, <code>{"S": "6"}</code> does not compare to <code>{"N": "6"}</code>. Also, <code>{"N": "6"}</code> does not compare to <code>{"NS": ["6", "2", "1"]}</code>.</p> | |

| Name | Description | Required |
|-------------------|--|----------|
| ExclusiveStartKey | <p>Primary key of the item from which to continue an earlier scan. An earlier scan might provide this value if that scan operation was interrupted before scanning the entire table; either because of the result set size or the <code>Limit</code> parameter. The <code>LastEvaluatedKey</code> can be passed back in a new scan request to continue the operation from that point.</p> <p>Type: <code>HashKeyElement</code>, or <code>HashKeyElement</code> and <code>RangeKeyElement</code> for a composite primary key.</p> | No |

Responses

Syntax

```
HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 229

{"Count":2,"Items":[{
    "AttributeName1":{"S":"AttributeValue1"},
    "AttributeName2":{"S":"AttributeValue2"},
    "AttributeName3":{"S":"AttributeValue3"}
},{
    "AttributeName1":{"S":"AttributeValue4"},
    "AttributeName2":{"S":"AttributeValue5"},
    "AttributeName3":{"S":"AttributeValue6"},
    "AttributeName5":{"B":"dmFsdWU="}
}],
"LastEvaluatedKey":
    {"HashKeyElement":{"S":"AttributeName1"},
     "RangeKeyElement":{"N":"AttributeName2"}},
"ConsumedCapacityUnits":1,
"ScannedCount":2
}
```

| Name | Description |
|-------|---|
| Items | <p>Container for the attributes meeting the operation parameters.</p> <p>Type: Map of attribute names to and their data types and values.</p> |
| Count | Number of items in the response. For more information, see Counting the Items in the Results (p. 478) . |

| Name | Description |
|-----------------------|---|
| | Type: Number |
| ScannedCount | <p>Number of items in the complete scan before any filters are applied. A high ScannedCount value with few, or no, Count results indicates an inefficient Scan operation. For more information, see Counting the Items in the Results (p. 478).</p> <p>Type: Number</p> |
| LastEvaluatedKey | <p>Primary key of the item where the scan operation stopped. Provide this value in a subsequent scan operation to continue the operation from that point.</p> <p>The LastEvaluatedKey is null when the entire scan result set is complete (i.e. the operation processed the “last page”).</p> |
| ConsumedCapacityUnits | <p>The number of read capacity units consumed by the operation. This value shows the number applied toward your provisioned throughput. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Type: Number</p> |

Special Errors

| Error | Description |
|---------------------------|------------------------------------|
| ResourceNotFoundException | The specified table was not found. |

Examples

For examples using the AWS SDK, see [Working with Scans in DynamoDB \(p. 475\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"1-hash-rangetable","ScanFilter":{}}
```

Sample Response

```
HTTP/1.1 200
x-amzn-RequestId: 4e8a5fa9-71e7-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 465
```

```
{
  "Count":4,
  "Items":[{
    "date": {"S":"1980"},  

    "fans": {"SS":["Dave","Aaron"]},  

    "name": {"S":"Airplane"},  

    "rating": {"S":"****"}  

  },  

  {"date": {"S":"1999"},  

    "fans": {"SS":["Ziggy","Laura","Dean"]},  

    "name": {"S":"Matrix"},  

    "rating": {"S":*****"}  

  },  

  {"date": {"S":"1976"},  

    "fans": {"SS":["Riley"]},  

    "name": {"S":"The Shaggy D.A."},  

    "rating": {"S":"**"}  

  },  

  {"date": {"S":"1985"},  

    "fans": {"SS":["Fox","Lloyd"]},  

    "name": {"S":"Back To The Future"},  

    "rating": {"S":"****"}  

  }],
  "ConsumedCapacityUnits":0.5,  

  "ScannedCount":4
}
```

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.  

POST / HTTP/1.1  

x-amz-target: DynamoDB_20111205.Scan  

content-type: application/x-amz-json-1.0  

content-length: 125  

{"TableName":"comp5",  

 "ScanFilter":  

  {"time":  

   {"AttributeValueList":[{"N":"400"}],  

    "ComparisonOperator":"GT"}  

 }  

}
```

Sample Response

```
HTTP/1.1 200 OK  

x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG  

content-type: application/x-amz-json-1.0  

content-length: 262  

Date: Mon, 15 Aug 2011 16:52:02 GMT  

{"Count":2,  

 "Items": [  

  {"friends": {"SS":["Dave","Ziggy","Barrie"]},  

   "status": {"S":"chatting"},  

   "time": {"N":"2000"},  

   "user": {"S":"Casey"}},  

  {"friends": {"SS":["Dave","Ziggy","Barrie"]},  

   "status": {"S":"chatting"},  

   "time": {"N":"2000"},  

   "user": {"S":"Freddy"}  

 ],  

 "ConsumedCapacityUnits":0.5,  

 "ScannedCount":4
}
```

```
}
```

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.Scan
content-type: application/x-amz-json-1.0

{"TableName":"comp5",
 "Limit":2,
 "ScanFilter":
 {"time":
 {"AttributeValueList":[{"N":"400"}],
 "ComparisonOperator":"GT"
 },
 "ExclusiveStartKey":
 {"HashKeyElement":{"S":"Freddy"}, "RangeKeyElement":{"N":"2000"}}
}
```

Sample Response

```
HTTP/1.1 200 OK
x-amzn-RequestId: PD1CQK9QCTERLTJP20VALJ60TRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 232
Date: Mon, 15 Aug 2011 16:52:02 GMT

{"Count":1,
 "Items":[
 {"friends":{"SS":["Jane", "James", "John"]},
 "status":{"S":"exercising"},
 "time":{"N":"2200"},
 "user":{"S":"Roger"}}
 ],
 "LastEvaluatedKey":{"HashKeyElement":{"S":"Riley"}, "RangeKeyElement":{"N":"250"}},
 "ConsumedCapacityUnits":0.5
 "ScannedCount":2
}
```

Related Actions

- [Query \(p. 1085\)](#)
- [BatchGetItem \(p. 1052\)](#)

UpdateItem

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Edits an existing item's attributes. You can perform a conditional update (insert a new attribute name-value pair if it doesn't exist, or replace an existing name-value pair if it has certain expected attribute values).

Note

You cannot update the primary key attributes using UpdateItem. Instead, delete the item and use PutItem to create a new item with new attributes.

The UpdateItem operation includes an `Action` parameter, which defines how to perform the update. You can put, delete, or add attribute values.

Attribute values may not be null; string and binary type attributes must have lengths greater than zero; and set type attributes must not be empty. Requests with empty values will be rejected with a `ValidationException`.

If an existing item has the specified primary key:

- **PUT**— Adds the specified attribute. If the attribute exists, it is replaced by the new value.
- **DELETE**— If no value is specified, this removes the attribute and its value. If a set of values is specified, then the values in the specified set are removed from the old set. So if the attribute value contains [a,b,c] and the delete action contains [a,c], then the final attribute value is [b]. The type of the specified value must match the existing value type. Specifying an empty set is not valid.
- **ADD**— Only use the add action for numbers or if the target attribute is a set (including string sets). ADD does not work if the target attribute is a single string value or a scalar binary value. The specified value is added to a numeric value (incrementing or decrementing the existing numeric value) or added as an additional value in a string set. If a set of values is specified, the values are added to the existing set. For example if the original set is [1,2] and supplied value is [3], then after the add operation the set is [1,2,3], not [4,5]. An error occurs if an Add action is specified for a set attribute and the attribute type specified does not match the existing set type.

If you use ADD for an attribute that does not exist, the attribute and its values are added to the item.

If no item matches the specified primary key:

- **PUT**— Creates a new item with specified primary key. Then adds the specified attribute.
- **DELETE**— Nothing happens.
- **ADD**— Creates an item with supplied primary key and number (or set of numbers) for the attribute value. Not valid for a string or a binary type.

Note

If you use ADD to increment or decrement a number value for an item that doesn't exist before the update, DynamoDB uses 0 as the initial value. Also, if you update an item using ADD to increment or decrement a number value for an attribute that doesn't exist before the update (but the item does) DynamoDB uses 0 as the initial value. For example, you use ADD to add +3 to an attribute that did not exist before the update. DynamoDB uses 0 for the initial value, and the value after the update is 3.

For more information about using this operation, see [Working with Items and Attributes \(p. 374\)](#).

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateItem  
content-type: application/x-amz-json-1.0  
  
{"TableName": "Table1",
```

```

    "Key": {
        "HashKeyElement": {"S": "AttributeValue1"},
        "RangeKeyElement": {"N": "AttributeValue2"}},
        "AttributeUpdates": {"AttributeName3": {"Value": {"S": "AttributeValue3_New"}, "Action": "PUT"}},
        "Expected": {"AttributeName3": {"Value": {"S": "AttributeValue3_Current"}}, "ReturnValues": "ReturnValuesConstant"}
    }
}

```

| Name | Description | Required |
|-------------------------|---|----------|
| TableName | The name of the table containing the item to update. Type: String | Yes |
| Key | The primary key that defines the item. For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement to its value and RangeKeyElement to its value. | Yes |
| AttributeUpdates | Map of attribute name to the new value and action for the update. The attribute names specify the attributes to modify, and cannot contain any primary key attributes. Type: Map of attribute name, value, and an action for the attribute update. | |
| AttributeUpdates:Action | Specifies how to perform the update. Possible values: PUT (default), ADD or DELETE. The semantics are explained in the UpdateItem description. Type: String Default: PUT | No |
| Expected | Designates an attribute for a conditional update. The Expected parameter allows you to provide an attribute name, and whether or not DynamoDB should check to see if the attribute value already exists; or if the attribute value exists and has a particular value before changing it. Type: Map of attribute names. | No |

| Name | Description | Required |
|---|---|----------|
| Expected:AttributeName | <p>The name of the attribute for the conditional put.</p> <p>Type: String</p> | No |
| Expected:AttributeName: ExpectedAttributeValue | <p>Use this parameter to specify whether or not a value already exists for the attribute name-value pair.</p> <p>The following JSON notation updates the item if the "Color" attribute doesn't already exist for that item:</p> <div style="border: 1px solid black; padding: 5px;"> <pre>"Expected" : {"Color":{"Exists":false}}</pre> </div> <p>The following JSON notation checks to see if the attribute with name "Color" has an existing value of "Yellow" before updating the item:</p> <div style="border: 1px solid black; padding: 5px;"> <pre>"Expected" : {"Color":{"Exists":true}, {"Value":{"S":"Yellow"}}}</pre> </div> <p>By default, if you use the Expected parameter and provide a Value, DynamoDB assumes the attribute exists and has a current value to be replaced. So you don't have to specify {"Exists":true}, because it is implied. You can shorten the request to:</p> <div style="border: 1px solid black; padding: 5px;"> <pre>"Expected" : {"Color":{"Value": {"S":"Yellow"}}}</pre> </div> <p>Note If you specify {"Exists":true} without an attribute value to check, DynamoDB returns an error.</p> | No |

| Name | Description | Required |
|--------------|--|----------|
| ReturnValues | <p>Use this parameter if you want to get the attribute name-value pairs before they were updated with the <code>UpdateItem</code> request. Possible parameter values are <code>NONE</code> (default) or <code>ALL_OLD</code>, <code>UPDATED_OLD</code>, <code>ALL_NEW</code> or <code>UPDATED_NEW</code>. If <code>ALL_OLD</code> is specified, and <code>UpdateItem</code> overwrote an attribute name-value pair, the content of the old item is returned. If this parameter is not provided or is <code>NONE</code>, nothing is returned. If <code>ALL_NEW</code> is specified, then all the attributes of the new version of the item are returned. If <code>UPDATED_NEW</code> is specified, then the new versions of only the updated attributes are returned.</p> <p>Type: String</p> | No |

Responses

Syntax

The following syntax example assumes the request specified a `ReturnValues` parameter of `ALL_OLD`; otherwise, the response has only the `ConsumedCapacityUnits` element.

```

HTTP/1.1 200
x-amzn-RequestId: 8966d095-71e9-11e0-a498-71d736f27375
content-type: application/x-amz-json-1.0
content-length: 140

{"Attributes": {
    "AttributeName1": {"S": "AttributeValue1"},
    "AttributeName2": {"S": "AttributeValue2"},
    "AttributeName3": {"S": "AttributeValue3"},
    "AttributeName5": {"B": "dmFsdWU="}
},
"ConsumedCapacityUnits": 1
}
```

| Name | Description |
|-----------------------|--|
| Attributes | <p>A map of attribute name-value pairs, but only if the <code>ReturnValues</code> parameter is specified as something other than <code>NONE</code> in the request.</p> <p>Type: Map of attribute name-value pairs.</p> |
| ConsumedCapacityUnits | The number of write capacity units consumed by the operation. This value shows the number |

| Name | Description |
|------|---|
| | <p>applied toward your provisioned throughput. For more information see Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341).</p> <p>Type: Number</p> |

Special Errors

| Error | Description |
|---------------------------------|---|
| ConditionalCheckFailedException | Conditional check failed. Attribute ("+ name +") value is ("+ value +") but was expected ("+ expValue +") |
| ResourceNotFoundExceptions | The specified item or attribute was not found. |

Examples

For examples using the AWS SDK, see [Working with Items and Attributes \(p. 374\)](#).

Sample Request

```
// This header is abbreviated. For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateItem
content-type: application/x-amz-json-1.0

{"TableName": "comp5",
 "Key": {
     "HashKeyElement": {"S": "Julie"}, "RangeKeyElement": {"N": "1307654350"}},
 "AttributeUpdates": {
     "status": {"Value": {"S": "online"}, "Action": "PUT"}, "Expected": {"status": {"Value": {"S": "offline"}}, "ReturnValues": "ALL_NEW"}
 }
```

Sample Response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 5IMHO7F01Q9P7Q6QMKMMI3R3QRVV4KQNSO5AEMVJF66Q9ASUAAJG
content-type: application/x-amz-json-1.0
content-length: 121
Date: Fri, 26 Aug 2011 21:05:00 GMT

{"Attributes": {
    "friends": {"SS": ["Lynda", "Aaron"]}, "status": {"S": "online"}, "time": {"N": "1307654350"}, "user": {"S": "Julie"}}, "ConsumedCapacityUnits": 1}
```

Related Actions

- [PutItem \(p. 1080\)](#)
- [DeleteItem \(p. 1066\)](#)

UpdateTable

Important

This section refers to API version 2011-12-05, which is deprecated and should not be used for new applications.

For documentation on the current low-level API, see the [Amazon DynamoDB API Reference](#).

Description

Updates the provisioned throughput for the given table. Setting the throughput for a table helps you manage performance and is part of the provisioned throughput feature of DynamoDB. For more information, see [Managing Settings on DynamoDB Provisioned Capacity Tables \(p. 341\)](#).

The provisioned throughput values can be upgraded or downgraded based on the maximums and minimums listed in [Service, Account, and Table Limits in Amazon DynamoDB \(p. 960\)](#).

The table must be in the ACTIVE state for this operation to succeed. UpdateTable is an asynchronous operation; while executing the operation, the table is in the UPDATING state. While the table is in the UPDATING state, the table still has the provisioned throughput from before the call. The new provisioned throughput setting is in effect only when the table returns to the ACTIVE state after the UpdateTable operation.

Requests

Syntax

```
// This header is abbreviated.  
// For a sample of a complete header, see DynamoDB Low-Level API.  
POST / HTTP/1.1  
x-amz-target: DynamoDB_20111205.UpdateTable  
content-type: application/x-amz-json-1.0  
  
{ "TableName": "Table1",  
    "ProvisionedThroughput": { "ReadCapacityUnits": 5, "WriteCapacityUnits": 15 }  
}
```

| Name | Description | Required |
|-----------------------|---|----------|
| TableName | The name of the table to update. Type: String | Yes |
| ProvisionedThroughput | New throughput for the specified table, consisting of values for ReadCapacityUnits and WriteCapacityUnits. See Managing Settings on DynamoDB Provisioned Capacity Tables (p. 341) . | Yes |

| Name | Description | Required |
|--|--|----------|
| | Type: Array | |
| ProvisionedThroughput :ReadCapacityUnits | <p>Sets the minimum number of consistent ReadCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> <p>Eventually consistent read operations require less effort than a consistent read operation, so a setting of 50 consistent ReadCapacityUnits per second provides 100 eventually consistent ReadCapacityUnits per second.</p> | Yes |
| ProvisionedThroughput :WriteCapacityUnits | <p>Sets the minimum number of WriteCapacityUnits consumed per second for the specified table before DynamoDB balances the load with other operations.</p> | Yes |

Responses

Syntax

```

HTTP/1.1 200 OK
x-amzn-RequestId: CSOC7TJPLR00OKIRLGOHVAICUFVV4KQNSO5AEMVJF66Q9ASUAAJG
Content-Type: application/json
Content-Length: 311
Date: Tue, 12 Jul 2011 21:31:03 GMT

{
    "TableDescription": {
        "CreationDateTime": 1.321657838135E9,
        "KeySchema": [
            {"HashKeyElement": {"AttributeName": "AttributeValue1", "AttributeType": "S"}, "RangeKeyElement": {"AttributeName": "AttributeValue2", "AttributeType": "N"}},
        "ProvisionedThroughput": {
            "LastDecreaseDateTime": 1.321661704489E9,
            "LastIncreaseDateTime": 1.321663607695E9,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 10},
        "TableName": "Table1",
        "TableStatus": "UPDATING"}}
}

```

| Name | Description |
|-----------------------|---|
| CreationDateTime | Date when the table was created. Type: Number |
| KeySchema | The primary key (simple or composite) structure for the table. A name-value pair for the HashKeyElement is required, and a name-value pair for the RangeKeyElement is optional (only required for composite primary keys). The maximum hash key size is 2048 bytes. The maximum range key size is 1024 bytes. Both limits are enforced separately (i.e. you can have a combined hash + range 2048 + 1024 key). For more information about primary keys, see Primary Key (p. 6) . Type: Map of HashKeyElement, or HashKeyElement and RangeKeyElement for a composite primary key. |
| ProvisionedThroughput | Current throughput settings for the specified table, including values for LastIncreaseDateTime (if applicable), LastDecreaseDateTime (if applicable), Type: Array |
| TableName | The name of the updated table. Type: String |
| TableStatus | The current state of the table (CREATING, ACTIVE, DELETING or UPDATING), which should be UPDATING. Use the DescribeTables (p. 1073) operation to check the status of the table. Type: String |

Special Errors

| Error | Description |
|---------------------------|---------------------------------------|
| ResourceNotFoundException | The specified table was not found. |
| ResourceInUseException | The table is not in the ACTIVE state. |

Examples

Sample Request

```
// This header is abbreviated.
```

```
// For a sample of a complete header, see DynamoDB Low-Level API.
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.UpdateTable
content-type: application/x-amz-json-1.0

{"TableName":"comp1",
 "ProvisionedThroughput":{"ReadCapacityUnits":5,"WriteCapacityUnits":15}
}
```

Sample Response

```
HTTP/1.1 200 OK
content-type: application/x-amz-json-1.0
content-length: 390
Date: Sat, 19 Nov 2011 00:46:47 GMT

{"TableDescription":
 {"CreationDateTime":1.321657838135E9,
 "KeySchema":
 {"HashKeyElement":{"AttributeName":"user","AttributeType":"S"},
 "RangeKeyElement":{"AttributeName":"time","AttributeType":"N"}},
 "ProvisionedThroughput":
 {"LastDecreaseDateTime":1.321661704489E9,
 "LastIncreaseDateTime":1.321663607695E9,
 "ReadCapacityUnits":5,
 "WriteCapacityUnits":10},
 "TableName":"comp1",
 "TableStatus":"UPDATING"}
}
```

Related Actions

- [CreateTable \(p. 1061\)](#)
- [DescribeTables \(p. 1073\)](#)
- [DeleteTable \(p. 1070\)](#)

Document History for DynamoDB

The following table describes the important changes in each release of the *DynamoDB Developer Guide* from July 3, 2018, onward. For notification about updates to this documentation, you can subscribe to the RSS feed (at the top left corner of this page).

| update-history-change | update-history-description | update-history-date |
|---|--|---------------------|
| Support for DynamoDB Local (Downloadable Version) (p. 800) | The NoSQL Workbench now supports connecting to DynamoDB Local (Downloadable Version) to design, create, query, and manage DynamoDB tables. | November 8, 2019 |
| NoSQL Workbench preview released (p. 800) | This is the initial release of NoSQL Workbench for DynamoDB. Use NoSQL Workbench to design, create, query, and manage DynamoDB tables. For more information, see NoSQL Workbench for Amazon DynamoDB (Preview) . | August 26, 2019 |
| DAX adds support for transactional operations using Python and .NET (p. 1114) | DAX supports the <code>TransactWriteItems</code> and <code>TransactGetItems</code> APIs for applications written in Go, Java, .NET, Node.js, and Python. For more information, see In-Memory Acceleration with DAX . | February 14, 2019 |
| Amazon DynamoDB Local (Downloadable Version) Updates (p. 1114) | DynamoDB Local (Downloadable Version) now supports transactional APIs, on-demand read/write capacity, capacity reporting for read and write operations, and 20 global secondary indexes. For more information, see Differences Between Downloadable DynamoDB and the DynamoDB Web Service . | February 4, 2019 |
| Amazon DynamoDB On-Demand (p. 1114) | DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use. For more information, see Read/Write Capacity Mode . | November 28, 2018 |

| | | |
|--|---|-------------------|
| Amazon DynamoDB Transactions (p. 1114) | DynamoDB transactions make coordinated, all-or-nothing changes to multiple items, both within and across tables, providing atomicity, consistency, isolation, and durability (ACID) in DynamoDB. For more information, see Amazon DynamoDB Transactions . | November 27, 2018 |
| Amazon DynamoDB encrypts all customer data at rest (p. 1114) | DynamoDB encryption at rest provides an additional layer of data protection by securing your data in the encrypted table, including its primary key, local and global secondary indexes, streams, global tables, backups, and DAX clusters whenever the data is stored in durable media. For more information, see Amazon DynamoDB Encryption at Rest . | November 15, 2018 |
| Use Amazon DynamoDB Local More Easily with the New Docker Image (p. 1114) | Now, it's easier to use Amazon DynamoDB local, the downloadable version of DynamoDB, to help you develop and test your DynamoDB applications by using the new DynamoDB local Docker image. For more information, see DynamoDB (Downloadable Version) and Docker . | August 22, 2018 |
| Amazon DynamoDB Accelerator (DAX) Adds Support for Encryption at Rest (p. 1114) | Amazon DynamoDB Accelerator (DAX) now supports encryption at rest for new DAX clusters to help you accelerate reads from Amazon DynamoDB tables in security-sensitive applications that are subject to strict compliance and regulatory requirements. For more information, see DAX Encryption at Rest . | August 9, 2018 |
| DynamoDB point-in-time recovery (PITR) adds support for restoring deleted tables (p. 1114) | If you delete a table with point-in-time recovery enabled, a system backup is automatically created and is retained for 35 days (at no additional cost). For more information, see Before You Begin Using Point In Time Recovery . | August 7, 2018 |

[Updates now available over RSS \(p. 1114\)](#)

You can now subscribe to the [RSS feed](#) (at the top left corner of this page) to receive notifications about updates to the Amazon DynamoDB Developer Guide.

July 3, 2018

Earlier Updates

The following table describes important changes of the *DynamoDB Developer Guide* before July 3, 2018.

| Change | Description | Date Changed |
|---|--|------------------|
| Go support for DAX | Now, you can enable microsecond read performance for Amazon DynamoDB tables in your applications written in the Go programming language by using the new Amazon DynamoDB Accelerator (DAX) SDK for Go. For more information, see DAX SDK for Go (p. 686) . | June 26, 2018 |
| DynamoDB announces SLA | DynamoDB has released a public availability SLA. For more information, see Amazon DynamoDB Service Level Agreement . | June 19, 2018 |
| DynamoDB continuous backups and Point-In-Time Recovery (PITR) | Point-in-time recovery helps protect your Amazon DynamoDB tables from accidental write or delete operations. With point in time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your table. For more information, see Point-in-Time Recovery for DynamoDB (p. 618) . | April 25, 2018 |
| Encryption at rest for DynamoDB | DynamoDB encryption at rest, available for new DynamoDB tables, helps you secure your application data in Amazon DynamoDB tables by using | February 8, 2018 |

| Change | Description | Date Changed |
|-----------------------------|--|-------------------|
| | AWS-managed encryption keys stored in AWS Key Management Service. For more information, see DynamoDB Encryption at Rest (p. 815) . | |
| DynamoDB Backup and restore | On-Demand Backup allows you to create full backups of your DynamoDB tables data for data archival, helping you meet your corporate and governmental regulatory requirements. You can backup tables from a few megabytes to hundreds of terabytes of data, with no impact on performance and availability to your production applications. For more information, see On-Demand Backup and Restore for DynamoDB (p. 603) . | November 29, 2017 |
| DynamoDB Global tables | Global Tables builds upon DynamoDB's global footprint to provide you with a fully managed, multi-region, and multi-master database that provides fast, local, read and write performance for massively scaled, global applications. Global Tables replicates your Amazon DynamoDB tables automatically across your choice of AWS regions. For more information, see Global Tables: Multi-Region Replication with DynamoDB (p. 624) . | November 29, 2017 |
| Node.js support for DAX | Node.js developers can leverage Amazon DynamoDB Accelerator (DAX), using the DAX client for Node.js. For more information, see In-Memory Acceleration with DynamoDB Accelerator (DAX) (p. 659) . | October 5, 2017 |

| Change | Description | Date Changed |
|---|---|-----------------|
| VPC Endpoints for DynamoDB | <p>DynamoDB endpoints allow Amazon EC2 instances in your Amazon VPC to access DynamoDB, without exposure to the public Internet. Network traffic between your VPC and DynamoDB does not leave the Amazon network. For more information, see Using Amazon VPC Endpoints to Access DynamoDB (p. 887).</p> | August 16, 2017 |
| Auto Scaling for DynamoDB | <p>DynamoDB auto scaling eliminates the need for manually defining or adjusting provisioned throughput settings. Instead, DynamoDB auto scaling dynamically adjusts read and write capacity in response to actual traffic patterns. This allows a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, DynamoDB auto scaling decreases the provisioned capacity. For more information, see Managing Throughput Capacity Automatically with DynamoDB Auto Scaling (p. 345).</p> | June 14, 2017 |
| Amazon DynamoDB Accelerator (DAX) | <p>Amazon DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement – from milliseconds to microseconds – even at millions of requests per second. For more information, see In-Memory Acceleration with DynamoDB Accelerator (DAX) (p. 659).</p> | April 19, 2017 |
| DynamoDB now supports automatic item expiration with Time-To-Live (TTL) | <p>Amazon DynamoDB Time-to-Live (TTL) enables you to automatically delete expired items from your tables, at no additional cost. For more information, see Expiring Items By Using DynamoDB Time to Live (TTL) (p. 408).</p> | Feb 27, 2017 |

| Change | Description | Date Changed |
|--|---|-------------------|
| DynamoDB now supports Cost Allocation Tags | <p>You can now add tags to your Amazon DynamoDB tables for improved usage categorization and more granular cost reporting. For more information, see Adding Tags and Labels to Resources (p. 359).</p> | Jan 19, 2017 |
| New DynamoDB <code>DescribeLimits</code> API | <p>The <code>DescribeLimits</code> API returns the current provisioned capacity limits for your AWS account in a region, both for the region as a whole and for any one DynamoDB table that you create there. It lets you determine what your current account-level limits are so that you can compare them to the provisioned capacity that you are currently using, and have plenty of time to apply for an increase before you hit a limit. For more information, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960) and the DescribeLimits in the Amazon DynamoDB API Reference.</p> | March 1, 2016 |
| DynamoDB Console Update and New Terminology for Primary Key Attributes | <p>The DynamoDB management console has been redesigned to be more intuitive and easy to use. As part of this update, we are introducing new terminology for primary key attributes:</p> <ul style="list-style-type: none"> • Partition Key—also known as a <i>hash attribute</i>. • Sort Key—also known as a <i>range attribute</i>. <p>Only the names have changed; the functionality remains the same.</p> <p>When you create a table or a secondary index, you can choose either a simple primary key (partition key only), or a composite primary key (partition key and sort key). The DynamoDB documentation has been updated to reflect these changes.</p> | November 12, 2015 |

| Change | Description | Date Changed |
|---|--|-----------------|
| Amazon DynamoDB Storage Backend for Titan | The DynamoDB Storage Backend for Titan is a storage backend for the Titan graph database implemented on top of Amazon DynamoDB. When using the DynamoDB Storage Backend for Titan, your data benefits from the protection of DynamoDB, which runs across Amazon's high-availability data centers. The plugin is available for Titan version 0.4.4 (primarily for compatibility with existing applications) and Titan version 0.5.4 (recommended for new applications). Like other storage backends for Titan, this plugin supports the Tinkerpop stack (versions 2.4 and 2.5), including the Blueprints API and the Gremlin shell. For more information, see Amazon DynamoDB Storage Backend for Titan (p. 1025) . | August 20, 2015 |

| Change | Description | Date Changed |
|---|--|---------------|
| DynamoDB Streams, Cross-Region Replication, and Scan with Strongly Consistent Reads | <p>DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time. For more information, see Capturing Table Activity with DynamoDB Streams (p. 573) and the DynamoDB Streams API Reference.</p> <p>DynamoDB cross-region replication is a client-side solution for maintaining identical copies of DynamoDB tables across different AWS regions, in near real time. You can use cross region replication to back up DynamoDB tables, or to provide low-latency access to data where users are geographically distributed. For more information, see Cross-Region Replication (p. 594).</p> <p>The DynamoDB Scan operation uses eventually consistent reads, by default. You can use strongly consistent reads instead by setting the <code>ConsistentRead</code> parameter to true. For more information, see Read Consistency for Scan (p. 479) and Scan in the Amazon DynamoDB API Reference.</p> | July 16, 2015 |
| AWS CloudTrail support for Amazon DynamoDB | DynamoDB is now integrated with CloudTrail. CloudTrail captures API calls made from the DynamoDB console or from the DynamoDB API and tracks them in log files. For more information, see Logging DynamoDB Operations by Using AWS CloudTrail (p. 877) and the AWS CloudTrail User Guide . | May 28, 2015 |

| Change | Description | Date Changed |
|---|---|----------------|
| Improved support for Query expressions | <p>This release adds a new <code>KeyConditionExpression</code> parameter to the <code>Query</code> API. A <code>Query</code> reads items from a table or an index using primary key values. The <code>KeyConditionExpression</code> parameter is a string that identifies primary key names, and conditions to be applied to the key values; the <code>Query</code> retrieves only those items that satisfy the expression. The syntax of <code>KeyConditionExpression</code> is similar to that of other expression parameters in DynamoDB, and allows you to define substitution variables for names and values within the expression. For more information, see Working with Queries in DynamoDB (p. 458).</p> | April 27, 2015 |
| New comparison functions for conditional writes | <p>In DynamoDB, the <code>ConditionExpression</code> parameter determines whether a <code>PutItem</code>, <code>UpdateItem</code>, or <code>DeleteItem</code> succeeds: The item is written only if the condition evaluates to true. This release adds two new functions, <code>attribute_type</code> and <code>size</code>, for use with <code>ConditionExpression</code>. These functions allow you to perform a conditional writes based on the data type or size of an attribute in a table. For more information, see Condition Expressions (p. 392).</p> | April 27, 2015 |

| Change | Description | Date Changed |
|--|--|-------------------|
| Scan API for secondary indexes | <p>In DynamoDB, a Scan operation reads all of the items in a table, applies user-defined filtering criteria, and returns the selected data items to the application. This same capability is now available for secondary indexes too. To scan a local secondary index or a global secondary index, you specify the index name and the name of its parent table. By default, an index Scan returns all of the data in the index; you can use a filter expression to narrow the results that are returned to the application. For more information, see Working with Scans in DynamoDB (p. 475).</p> | February 10, 2015 |
| Online operations for global secondary indexes | <p>Online indexing lets you add or remove global secondary indexes on existing tables. With online indexing, you do not need to define all of a table's indexes when you create a table; instead, you can add a new index at any time. Similarly, if you decide you no longer need an index, you can remove it at any time. Online indexing operations are non-blocking, so that the table remains available for read and write activity while indexes are being added or removed. For more information, see Managing Global Secondary Indexes (p. 507).</p> | January 27, 2015 |

| Change | Description | Date Changed |
|----------------------------------|---|---------------------|
| Document model support with JSON | <p>DynamoDB allows you to store and retrieve documents with full support for document models. New data types are fully compatible with the JSON standard and allow you to nest document elements within one another. You can use document path dereference operators to read and write individual elements, without having to retrieve the entire document. This release also introduces new expression parameters for specifying projections, conditions and update actions when reading or writing data items. To learn more about document model support with JSON, see Data Types (p. 13) and Using Expressions in DynamoDB (p. 385).</p> | October 7, 2014 |
| Flexible scaling | <p>For tables and global secondary indexes, you can increase provisioned read and write throughput capacity by any amount, provided that you stay within your per-table and per-account limits. For more information, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960).</p> | October 7, 2014 |
| Larger item sizes | <p>The maximum item size in DynamoDB has increased from 64 KB to 400 KB. For more information, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960).</p> | October 7, 2014 |

| Change | Description | Date Changed |
|----------------------------------|--|----------------|
| Improved conditional expressions | <p>DynamoDB expands the operators that are available for conditional expressions, giving you additional flexibility for conditional puts, updates, and deletes. The newly available operators let you check whether an attribute does or does not exist, is greater than or less than a particular value, is between two values, begins with certain characters, and much more. DynamoDB also provides an optional <i>OR</i> operator for evaluating multiple conditions. By default, multiple conditions in an expression are <i>ANDed</i> together, so the expression is true only if all of its conditions are true. If you specify <i>OR</i> instead, the expression is true if one or more one conditions are true. For more information, see Working with Items and Attributes (p. 374).</p> | April 24, 2014 |
| Query filter | <p>The DynamoDB <code>Query</code> API supports a new <code>QueryFilter</code> option. By default, a <code>Query</code> finds items that match a specific partition key value and an optional sort key condition. A <code>Query</code> filter applies conditional expressions to other, non-key attributes; if a <code>Query</code> filter is present, then items that do not match the filter conditions are discarded before the <code>Query</code> results are returned to the application. For more information, see Working with Queries in DynamoDB (p. 458).</p> | April 24, 2014 |

| Change | Description | Date Changed |
|---|---|------------------|
| Data export and import using the AWS Management Console | <p>The DynamoDB console has been enhanced to simplify exports and imports of data in DynamoDB tables. With just a few clicks, you can set up an AWS Data Pipeline to orchestrate the workflow, and an Amazon Elastic MapReduce cluster to copy data from DynamoDB tables to an Amazon S3 bucket, or vice-versa. You can perform an export or import one time only, or set up a daily export job. You can even perform cross-region exports and imports, copying DynamoDB data from a table in one AWS region to a table in another AWS region. For more information, see Exporting and Importing DynamoDB Data Using AWS Data Pipeline (p. 1017).</p> | March 6, 2014 |
| Reorganized higher-level API documentation | <p>Information about the following APIs is now easier to find:</p> <ul style="list-style-type: none"> • Java: DynamoDBMapper • .NET: Document model and object-persistence model <p>These higher-level APIs are now documented here: Higher-Level Programming Interfaces for DynamoDB (p. 225).</p> | January 20, 2014 |

| Change | Description | Date Changed |
|-----------------------------|--|-------------------|
| Global secondary indexes | <p>DynamoDB adds support for global secondary indexes. As with a local secondary index, you define a global secondary index by using an alternate key from a table and then issuing Query requests on the index. Unlike a local secondary index, the partition key for the global secondary index does not have to be the same as that of the table; it can be any scalar attribute from the table. The sort key is optional and can also be any scalar table attribute. A global secondary index also has its own provisioned throughput settings, which are separate from those of the parent table.</p> <p>For more information, see Improving Data Access with Secondary Indexes (p. 496) and Using Global Secondary Indexes in DynamoDB (p. 499).</p> | December 12, 2013 |
| Fine-grained access control | <p>DynamoDB adds support for fine-grained access control. This feature allows customers to specify which principals (users, groups, or roles) can access individual items and attributes in a DynamoDB table or secondary index. Applications can also leverage web identity federation to offload the task of user authentication to a third-party identity provider, such as Facebook, Google, or Login with Amazon. In this way, applications (including mobile apps) can handle very large numbers of users, while ensuring that no one can access DynamoDB data items unless they are authorized to do so.</p> <p>For more information, see Using IAM Policy Conditions for Fine-Grained Access Control (p. 840).</p> | October 29, 2013 |

| Change | Description | Date Changed |
|------------------------------|---|----------------|
| 4 KB read capacity unit size | <p>The capacity unit size for reads has increased from 1 KB to 4 KB. This enhancement can reduce the number of provisioned read capacity units required for many applications. For example, prior to this release, reading a 10 KB item would consume 10 read capacity units; now that same 10 KB read would consume only 3 units (10 KB / 4 KB, rounded up to the next 4 KB boundary). For more information, see Read/Write Capacity Mode (p. 17).</p> | May 14, 2013 |
| Parallel scans | <p>DynamoDB adds support for parallel Scan operations. Applications can now divide a table into logical segments and scan all of the segments simultaneously. This feature reduces the time required for a Scan to complete, and fully utilizes a table's provisioned read capacity. For more information, see Working with Scans in DynamoDB (p. 475).</p> | May 14, 2013 |
| Local secondary indexes | <p>DynamoDB adds support for local secondary indexes. You can define sort key indexes on non-key attributes, and then use these indexes in Query requests. With local secondary indexes, applications can efficiently retrieve data items across multiple dimensions. For more information, see Local Secondary Indexes (p. 536).</p> | April 18, 2013 |

| Change | Description | Date Changed |
|---|---|------------------|
| New API version | <p>With this release, DynamoDB introduces a new API version (2012-08-10). The previous API version (2011-12-05) is still supported for backward compatibility with existing applications. New applications should use the new API version 2012-08-10. We recommend that you migrate your existing applications to API version 2012-08-10, since new DynamoDB features (such as local secondary indexes) will not be backported to the previous API version. For more information on API version 2012-08-10, see the Amazon DynamoDB API Reference.</p> | April 18, 2013 |
| IAM policy variable support | <p>The IAM access policy language now supports variables. When a policy is evaluated, any policy variables are replaced with values that are supplied by context-based information from the authenticated user's session. You can use policy variables to define general purpose policies without explicitly listing all the components of the policy. For more information about policy variables, go to Policy Variables in the <i>AWS Identity and Access Management Using IAM</i> guide.</p> <p>For examples of policy variables in DynamoDB, see Identity and Access Management in Amazon DynamoDB (p. 824).</p> | April 4, 2013 |
| PHP code examples updated for AWS SDK for PHP version 2 | <p>Version 2 of the AWS SDK for PHP is now available. The PHP code examples in the Amazon DynamoDB Developer Guide have been updated to use this new SDK. For more information on Version 2 of the SDK, see AWS SDK for PHP.</p> | January 23, 2013 |
| New endpoint | <p>DynamoDB expands to the AWS GovCloud (US-West) region. For the current list of service endpoints and protocols, see Regions and Endpoints.</p> | December 3, 2012 |

| Change | Description | Date Changed |
|---|--|--------------------|
| New endpoint | DynamoDB expands to the South America (São Paulo) region. For the current list of supported endpoints, see Regions and Endpoints . | December 3, 2012 |
| New endpoint | DynamoDB expands to the Asia Pacific (Sydney) region. For the current list of supported endpoints, see Regions and Endpoints . | November 13, 2012 |
| DynamoDB implements support for CRC32 checksums, supports strongly consistent batch gets, and removes restrictions on concurrent table updates. | <ul style="list-style-type: none"> • DynamoDB calculates a CRC32 checksum of the HTTP payload and returns this checksum in a new header, <code>x-amz-crc32</code>. For more information, see DynamoDB Low-Level API (p. 216). • By default, read operations performed by the <code>BatchGetItem</code> API are eventually consistent. A new <code>ConsistentRead</code> parameter in <code>BatchGetItem</code> lets you choose strong read consistency instead, for any table(s) in the request. For more information, see Description (p. 1052). • This release removes some restrictions when updating many tables simultaneously. The total number of tables that can be updated at once is still 10; however, these tables can now be any combination of CREATING, UPDATING or DELETING status. Additionally, there is no longer any minimum amount for increasing or reducing the <code>ReadCapacityUnits</code> or <code>WriteCapacityUnits</code> for a table. For more information, see Service, Account, and Table Limits in Amazon DynamoDB (p. 960). | November 2, 2012 |
| Best practices documentation | The Amazon DynamoDB Developer Guide identifies best practices for working with tables and items, along with recommendations for query and scan operations. | September 28, 2012 |

| Change | Description | Date Changed |
|---|--|-----------------|
| Support for binary data type | <p>In addition to the Number and String types, DynamoDB now supports Binary data type.</p> <p>Prior to this release, to store binary data, you converted your binary data into string format and stored it in DynamoDB. In addition to the required conversion work on the client-side, the conversion often increased the size of the data item requiring more storage and potentially additional provisioned throughput capacity.</p> <p>With the binary type attributes you can now store any binary data, for example compressed data, encrypted data, and images. For more information see Data Types (p. 13). For working examples of handling binary type data using the AWS SDKs, see the following sections:</p> <ul style="list-style-type: none"> • Example: Handling Binary Type Attributes Using the AWS SDK for Java Document API (p. 432) • Example: Handling Binary Type Attributes Using the AWS SDK for .NET Low-Level API (p. 455) <p>For the added binary data type support in the AWS SDKs, you will need to download the latest SDKs and you might also need to update any existing applications. For information about downloading the AWS SDKs, see .NET Code Examples (p. 332).</p> | August 21, 2012 |
| DynamoDB table items can be updated and copied using the DynamoDB console | DynamoDB users can now update and copy table items using the DynamoDB Console, in addition to being able to add and delete items. This new functionality simplifies making changes to individual items through the Console. | August 14, 2012 |

| Change | Description | Date Changed |
|---|---|---------------------|
| DynamoDB lowers minimum table throughput requirements | DynamoDB now supports lower minimum table throughput requirements, specifically 1 write capacity unit and 1 read capacity unit. For more information, see the Service, Account, and Table Limits in Amazon DynamoDB (p. 960) topic in the Amazon DynamoDB Developer Guide. | August 9, 2012 |
| Signature Version 4 support | DynamoDB now supports Signature Version 4 for authenticating requests. | July 5, 2012 |
| Table explorer support in DynamoDB Console | The DynamoDB Console now supports a table explorer that enables you to browse and query the data in your tables. You can also insert new items or delete existing items. The Creating Tables and Loading Data for Code Examples in DynamoDB (p. 325) and Using the Console (p. 54) sections have been updated for these features. | May 22, 2012 |
| New endpoints | DynamoDB availability expands with new endpoints in the US West (N. California) region, US West (Oregon) region, and the Asia Pacific (Singapore) region. For the current list of supported endpoints, go to Regions and Endpoints . | April 24, 2012 |
| BatchWriteItem API support | DynamoDB now supports a batch write API that enables you to put and delete several items from one or more tables in a single API call. For more information about the DynamoDB batch write API, see BatchWriteItem (p. 1057) . For information about working with items and using batch write feature using AWS SDKs, see Working with Items and Attributes (p. 374) and .NET Code Examples (p. 332) . | April 19, 2012 |
| Documented more error codes | For more information, see Error Handling with DynamoDB (p. 219) . | April 5, 2012 |

| Change | Description | Date Changed |
|--|--|---------------------|
| New endpoint | DynamoDB expands to the Asia Pacific (Tokyo) region. For the current list of supported endpoints, see Regions and Endpoints . | February 29, 2012 |
| ReturnedItemCount metric added | A new metric, <code>ReturnedItemCount</code> , provides the number of items returned in the response of a Query or Scan operation for DynamoDB is available for monitoring through CloudWatch. For more information, see Logging and Monitoring in DynamoDB (p. 855) . | February 24, 2012 |
| Added examples for incrementing values | DynamoDB supports incrementing and decrementing existing numeric values. Examples show adding to existing values in the "Updating an Item" sections at: Working with Items: Java (p. 415) . Working with Items: .NET (p. 435) . | January 25, 2012 |
| Initial product release | DynamoDB is introduced as a new service in Beta release. | January 18, 2012 |