DynamoDB

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

DynamoDB is a fully-managed NoSQL database service designed to deliver fast and predictable performance. It uses the Dynamo model in the essence of its design, and improves those features. It began as a way to manage website scalability challenges presented by the holiday season load.

This tutorial introduces you to key DynamoDB concepts necessary for creating and deploying a highly-scalable and performance-focused database.

Audience

This tutorial targets IT professionals, students, and management professionals who want a solid grasp of essential DynamoDB concepts.

After completing this tutorial, you will achieve intermediate expertise in DynamoDB, and easily build on your knowledge to solve more challenging problems.

Prerequisites

This tutorial assumes general knowledge of database technology, programming, Java or Java-like programming languages, and querying languages. It also assumes familiarity with typical database operations in an application.

Copyright and Disclaimer

Table of Contents

1. DynamoDB – Overview

DynamoDB allows users to create databases capable of storing and retrieving any amount of data, and serving any amount of traffic. It automatically distributes data and traffic over servers to dynamically manage each customer's requests, and also maintains fast performance.

DynamoDB vs. RDBMS

DynamoDB uses a NoSQL model, which means it uses a non-relational system. The following table highlights the differences between DynamoDB and RDBMS:

Common Tasks	RDBMS	DynamoDB
Connect to the Source	It uses a persistent connection and SQL commands.	It uses HTTP requests and API operations.
Create a Table	Its fundamental structures are tables, and must be defined.	It only uses primary keys, and no schema on creation. It uses various data sources.
Get Table Info	All table info remains accessible.	Only primary keys are revealed.
Load Table Data	It uses rows made of columns.	In tables, it uses items made of attributes.
Read Table Data	It uses SELECT statements and filtering statements.	It uses GetItem, Query, and Scan.
Manage Indexes	It uses standard indexes created through SQL statements. Modifications to it occur automatically on table changes.	It uses a secondary index to achieve the same function. It requires specifications (partition key and sort key).
Modify Table Data	It uses an UPDATE statement.	It uses an UpdateItem operation.
Delete Table Data	It uses a DELETE statement.	It uses a DeleteItem operation.
Delete a Table	It uses a DROP TABLE statement.	It uses a DeleteTable operation.

Advantages

The two main advantages of DynamoDB are scalability and flexibility. It does not force the use of a particular data source and structure, allowing users to work with virtually anything, but in a uniform way.

Its design also supports a wide range of use from lighter tasks and operations to demanding enterprise functionality. It also allows simple use of multiple languages: Ruby, Java, Python, C#, Erlang, PHP, and Perl.

Limitations

DynamoDB does suffer from certain limitations, however, these limitations do not necessarily create huge problems or hinder solid development.

You can review them from the following points:

- **Capacity Unit Sizes –** A read capacity unit is a single consistent read per second for items no larger than 4KB. A write capacity unit is a single write per second for items no bigger than 1KB.

- **Provisioned Throughput Min/Max –** All tables and global secondary indices have a minimum of one read and one write capacity unit. Maximums depend on region. In the US, 40K read and write remains the cap per table (80K per account), and other regions have a cap of 10K per table with a 20K account cap.

- **Provisioned Throughput Increase and Decrease –** You can increase this as often as needed, but decreases remain limited to no more than four times daily per table.

- **Table Size and Quantity Per Account –** Table sizes have no limits, but accounts have a 256 table limit unless you request a higher cap.

- **Secondary Indexes Per Table –** Five local and five global are permitted.

- **Projected Secondary Index Attributes Per Table –** DynamoDB allows 20 attributes.

- **Partition Key Length and Values –** Their minimum length sits at 1 byte, and maximum at 2048 bytes, however, DynamoDB places no limit on values.

- **Sort Key Length and Values –** Its minimum length stands at 1 byte, and maximum at 1024 bytes, with no limit for values unless its table uses a local secondary index.

- **Table and Secondary Index Names –** Names must conform to a minimum of 3 characters in length, and a maximum of 255. They use the following characters: A-Z, a-z, 0-9, "_", "-", and ".".

- **Attribute Names –** One character remains the minimum, and 64KB the maximum, with exceptions for keys and certain attributes.

- **Reserved Words –** DynamoDB does not prevent the use of reserved words as names.

- **Expression Length –** Expression strings have a 4KB limit. Attribute expressions have a 255-byte limit. Substitution variables of an expression have a 2MB limit.

2. DynamoDB – Basic Concepts

Before using DynamoDB, you must familiarize yourself with its basic components and ecosystem. In the DynamoDB ecosystem, you work with tables, attributes, and items. A table holds sets of items, and items hold sets of attributes. An attribute is a fundamental element of data requiring no further decomposition, i.e., a field.

Primary Key

The Primary Keys serve as the means of unique identification for table items, and secondary indexes provide query flexibility. DynamoDB streams record events by modifying the table data.

The Table Creation requires not only setting a name, but also the primary key; which identifies table items. No two items share a key. DynamoDB uses two types of primary keys:

- **Partition Key –** This simple primary key consists of a single attribute referred to as the "partition key." Internally, DynamoDB uses the key value as input for a hash function to determine storage.

- **Partition Key and Sort Key –** This key, known as the "Composite Primary Key", consists of two attributes:

    - The partition key and

    - The sort key.

    DynamoDB applies the first attribute to a hash function, and stores items with the same partition key together; with their order determined by the sort key. Items can share partition keys, but not sort keys.

The Primary Key attributes only allow scalar (single) values; and string, number, or binary data types. The non-key attributes do not have these constraints.

Secondary Indexes

These indexes allow you to query table data with an alternate key. Though DynamoDB does not force their use, they optimize querying.

DynamoDB uses two types of secondary indexes:

- **Global Secondary Index –** This index possesses partition and sort keys, which can differ from table keys.

- **Local Secondary Index –** This index possesses a partition key identical to the table, however, its sort key differs.

API

The API operations offered by DynamoDB include those of the control plane, data plane (e.g., creation, reading, updating, and deleting), and streams. In control plane operations, you create and manage tables with the following tools:

- CreateTable
- DescribeTable
- ListTables
- UpdateTable
- DeleteTable

In the data plane, you perform CRUD operations with the following tools:

Create	Read	Update	Delete
PutItem BatchWriteItem	GetItem BatchGetItem Query Scan	UpdateItem	DeleteItem BatchWriteItem

The stream operations control table streams. You can review the following stream tools:

- ListStreams
- DescribeStream
- GetShardIterator
- GetRecords

Provisioned Throughput

In table creation, you specify provisioned throughput, which reserves resources for reads and writes. You use capacity units to measure and set throughput.

When applications exceed the set throughput, requests fail. The DynamoDB GUI console allows monitoring of set and used throughput for better and dynamic provisioning.

Read Consistency

DynamoDB uses ***eventually consistent*** and ***strongly consistent*** reads to support dynamic application needs. Eventually consistent reads do not always deliver current data.

The strongly consistent reads always deliver current data (with the exception of equipment failure or network problems). Eventually consistent reads serve as the default setting, requiring a setting of true in the **ConsistentRead** parameter to change it.

Partitions

DynamoDB uses partitions for data storage. These storage allocations for tables have SSD backing and automatically replicate across zones. DynamoDB manages all the partition tasks, requiring no user involvement.

In table creation, the table enters the CREATING state, which allocates partitions. When it reaches ACTIVE state, you can perform operations. The system alters partitions when its capacity reaches maximum or when you change throughput.

3. DynamoDB – Environment

The DynamoDB Environment only consists of using your Amazon Web Services account to access the DynamoDB GUI console, however, you can also perform a local install.

Navigate to the following website: https://aws.amazon.com/dynamodb/

Click the "Get Started with Amazon DynamoDB" button, or the "Create an AWS Account" button if you do not have an Amazon Web Services account. The simple, guided process will inform you of all the related fees and requirements.

After performing all the necessary steps of the process, you will have the access. Simply sign in to the AWS console, and then navigate to the DynamoDB console.

Be sure to delete unused or unnecessary material to avoid associated fees.

Local Install

The AWS (Amazon Web Service) provides a version of DynamoDB for local installations. It supports creating applications without the web service or a connection. It also reduces provisioned throughput, data storage, and transfer fees by allowing a local database. This guide assumes a local install.

When ready for deployment, you can make a few small adjustments to your application to convert it to AWS use.

The install file is a **.jar executable**. It runs in Linux, Unix, Windows, and any other OS with Java support. Download the file by using one of the following links:

- **Tarball –** http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.tar.gz
- **Zip archive –** http://dynamodb-local.s3-website-us-west-2.amazonaws.com/dynamodb_local_latest.zip

Note: *Other repositories offer the file, but not necessarily the latest version. Use the links above for up-to-date install files. Also, ensure you have Java Runtime Engine (JRE) version 6.x or a newer version. DynamoDB cannot run with older versions.*

After downloading the appropriate archive, extract its directory (DynamoDBLocal.jar) and place it in the desired location.

You can then start DynamoDB by opening a command prompt, navigating to the directory containing DynamoDBLocal.jar, and entering the following command:

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

You can also stop the DynamoDB by closing the command prompt used to start it.

Working Environment

You can use a JavaScript shell, a GUI console, and multiple languages to work with DynamoDB. The languages available include Ruby, Java, Python, C#, Erlang, PHP, and Perl.

In this tutorial, we use Java and GUI console examples for conceptual and code clarity. Install a Java IDE, the AWS SDK for Java, and setup AWS security credentials for the Java SDK in order to utilize Java.

Conversion from Local to Web Service Code

When ready for deployment, you will need to alter your code. The adjustments depend on code language and other factors. The main change merely consists of changing the ***endpoint*** from a local point to an AWS region. Other changes require deeper analysis of your application.

A local install differs from the web service in many ways including, but not limited to the following key differences:

- The local install creates tables immediately, but the service takes much longer.
- The local install ignores throughput.
- The deletion occurs immediately in a local install.
- The reads/writes occur quickly in local installs due to the absence of network overhead.

4. DynamoDB – Operations Tools

DynamoDB provides three options for performing operations: a web-based GUI console, a JavaScript shell, and a programming language of your choice.

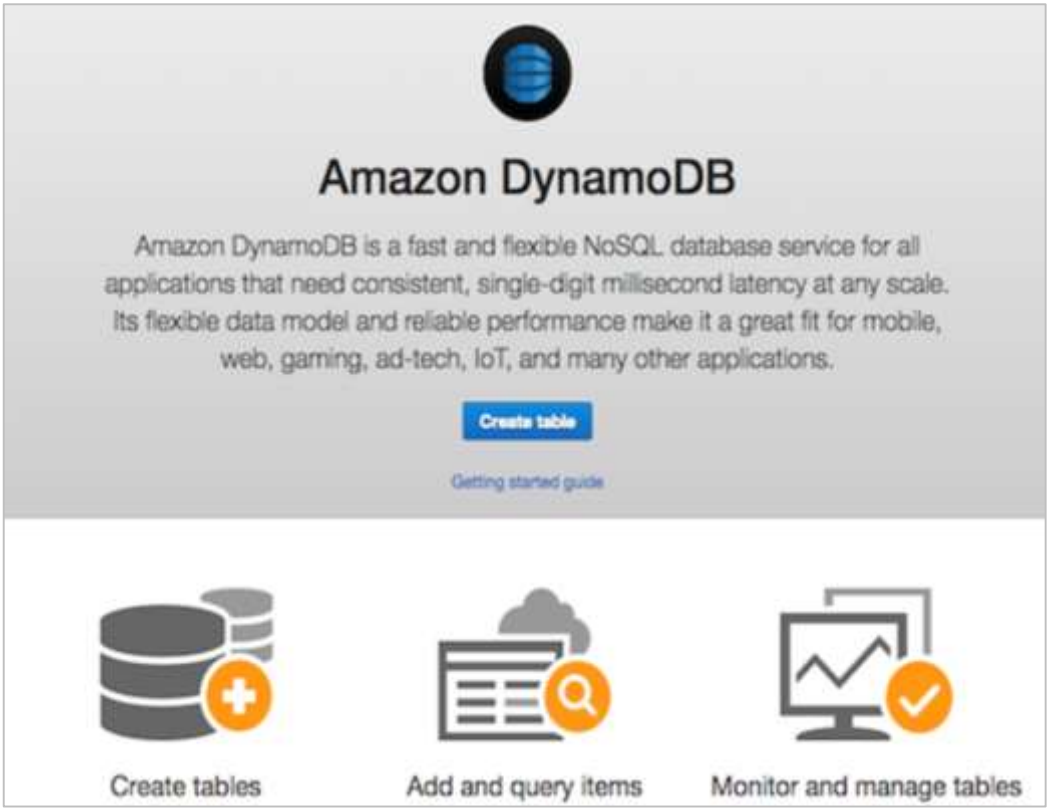
In this tutorial, we will focus on using the GUI console and Java language for clarity and conceptual understanding.

GUI Console

The GUI console or the AWS Management Console for Amazon DynamoDB can be found at the following address: https://console.aws.amazon.com/dynamodb/home

It allows you to perform the following tasks:

- CRUD
- View Table Items
- Perform Table Queries
- Set Alarms for Table Capacity Monitoring
- View Table Metrics in Real-Time
- View Table Alarms

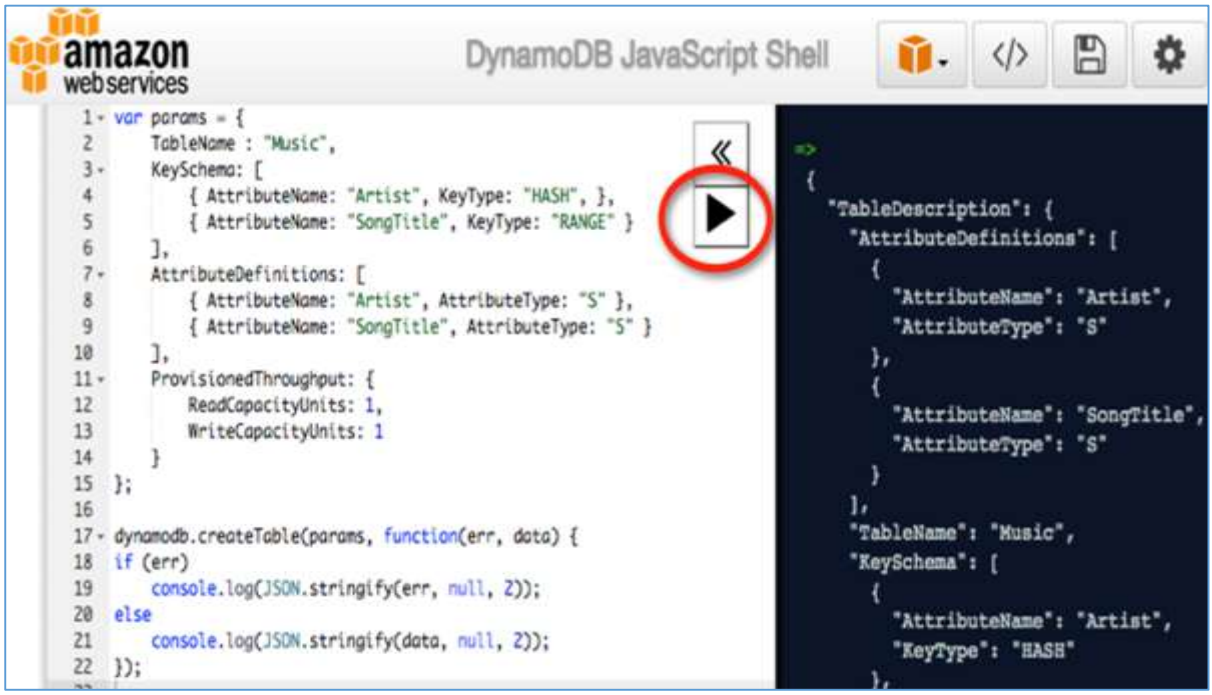


If your DynamoDB account has no tables, on access, it guides you through creating a table. Its main screen offers three shortcuts for performing common operations:

- Create Tables
- Add and Query Tables
- Monitor and Manage Tables

The JavaScript Shell

DynamoDB includes an interactive JavaScript shell. The shell runs inside a web browser, and the recommended browsers include Firefox and Chrome.



Note: *Using other browsers may result in errors.*

Access the shell by opening a web browser and entering the following address: http://localhost:8000/shell

Use the shell by entering JavaScript in the left pane, and clicking the "Play" icon button in the top right corner of the left pane, which runs the code. The code results display in the right pane.

DynamoDB and Java

Use Java with DynamoDB by utilizing your Java development environment. Operations confirm to normal Java syntax and structure.

5. DynamoDB – Data Types

Data types supported by DynamoDB include those specific to attributes, actions, and your coding language of choice.

Attribute Data Types

DynamoDB supports a large set of data types for table attributes. Each data type falls into one of the three following categories:

- **Scalar –** These types represent a single value, and include number, string, binary, Boolean, and null.
- **Document –** These types represent a complex structure possessing nested attributes, and include lists and maps.
- **Set –** These types represent multiple scalars, and include string sets, number sets, and binary sets.

Remember DynamoDB as a schemaless, NoSQL database that does not need attribute or data type definitions when creating a table. It only requires a primary key attribute data types in contrast to RDBMS, which require column data types on table creation.

Scalars

- **Numbers –** They are limited to 38 digits, and are either positive, negative, or zero.
- **String –** They are Unicode using UTF-8, with a minimum length of >0 and maximum of 400KB.
- **Binary –** They store any binary data, e.g., encrypted data, images, and compressed text. DynamoDB views its bytes as unsigned.
- **Boolean –** They store true or false.
- **Null –** They represent an unknown or undefined state.

Document

- **List –** It stores ordered value collections, and uses square ([...]) brackets.
- **Map –** It stores unordered name-value pair collections, and uses curly ({...}) braces.

Set

Sets must contain elements of the same type whether number, string, or binary. The only limits placed on sets consist of the 400KB item size limit, and each element being unique.

Action Data Types

DynamoDB API holds various data types used by actions. You can review a selection of the following key types:

- **AttributeDefinition –** It represents key table and index schema.
- **Capacity –** It represents the quantity of throughput consumed by a table or index.
- **CreateGlobalSecondaryIndexAction –** It represents a new global secondary index added to a table.
- **LocalSecondaryIndex –** It represents local secondary index properties.
- **ProvisionedThroughput –** It represents the provisioned throughput for an index or table.
- **PutRequest –** It represents PutItem requests.
- **TableDescription –** It represents table properties.

Supported Java Datatypes

DynamoDB provides support for primitive data types, Set collections, and arbitrary types for Java.

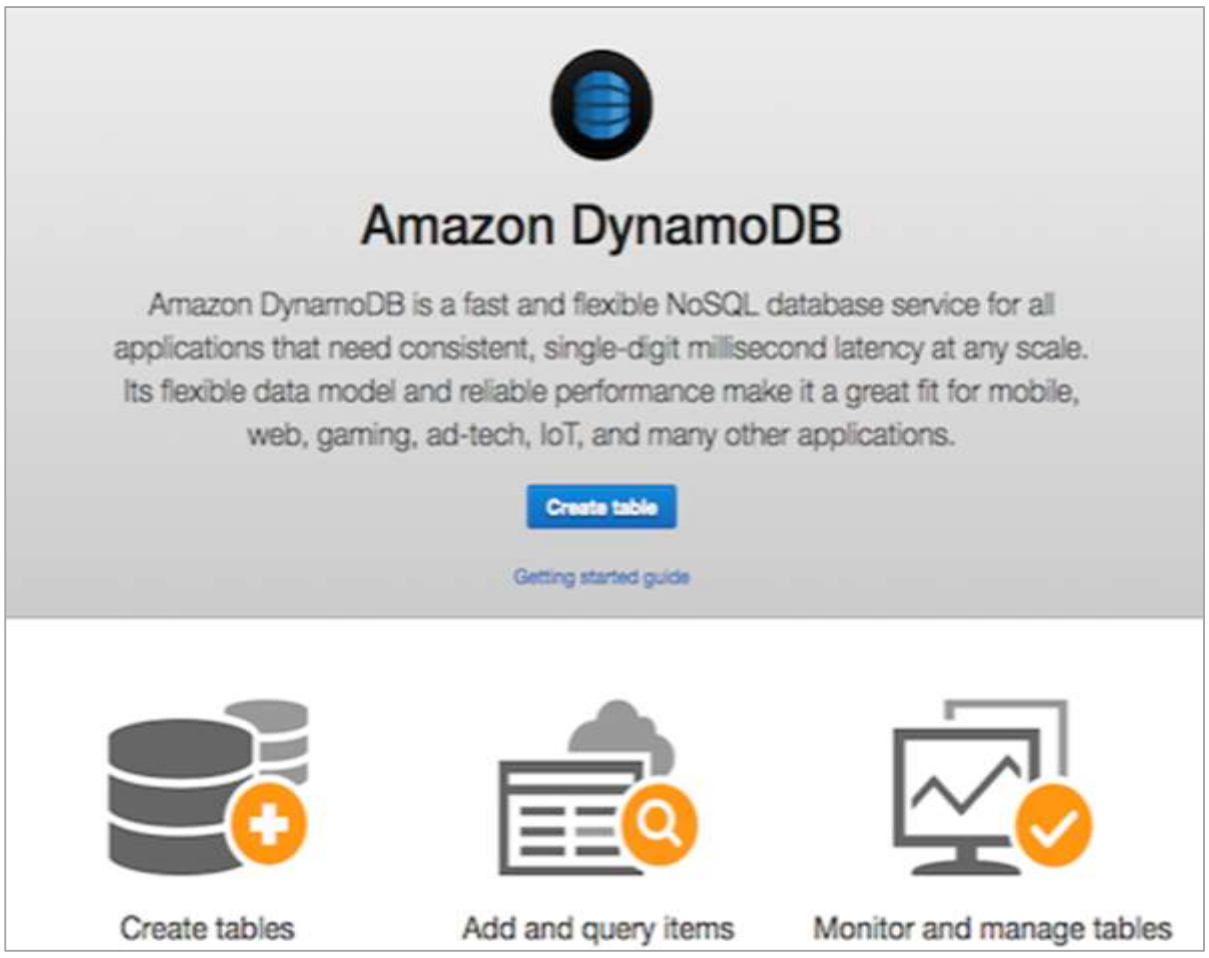
6. DynamoDB – Create Table

Creating a table generally consists of spawning the table, naming it, establishing its primary key attributes, and setting attribute data types.

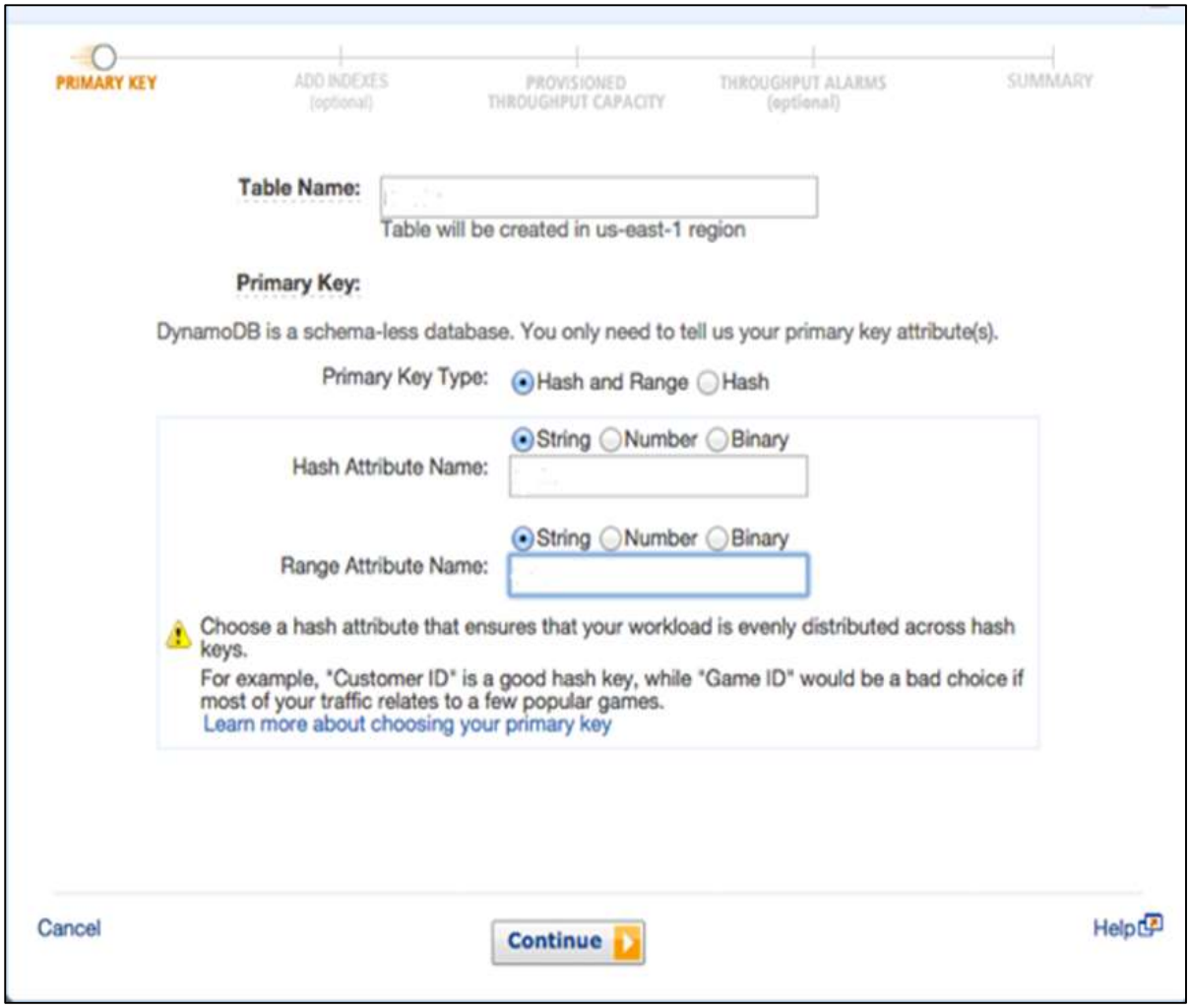
Utilize the GUI Console, Java, or another option to perform these tasks.

Create Table using the GUI Console

Create a table by accessing the console at https://console.aws.amazon.com/dynamodb. Then choose the "Create Table" option.



Our example generates a table populated with product information, with products of unique attributes identified by an ID number (numeric attribute). In the **Create Table** screen, enter the table name within the table name field; enter the primary key (ID) within the partition key field; and enter "Number" for the data type.



After entering all information, select **Create**.

Create Table using Java

Use Java to create the same table. Its primary key consists of the following two attributes:

- **ID –** Use a partition key, and the ScalarAttributeType **N**, meaning number.
- **Nomenclature –** Use a sort key, and the ScalarAttributeType **S**, meaning string.

Java uses the **createTable method** to generate a table; and within the call, table name, primary key attributes, and attribute data types are specified.

You can review the following example:

```
import java.util.Arrays;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 ProductsCreateTable {

    public static void main(String[] args) throws Exception {
        AmazonDynamoDBClient client = new AmazonDynamoDBClient()
            .withEndpoint("http://localhost:8000");

        DynamoDB dynamoDB = new DynamoDB(client);
        String tableName = "Products";

        try {
            System.out.println("Creating the table, wait...");
            Table table = dynamoDB.createTable(tableName,
                Arrays.asList(
                    new KeySchemaElement("ID", KeyType.HASH), // the partition key
                    // the sort key
                    new KeySchemaElement("Nomenclature", KeyType.RANGE)),

                    Arrays.asList(
                        new AttributeDefinition("ID", ScalarAttributeType.N),
                        new AttributeDefinition("Nomenclature", ScalarAttributeType.S)),
                        new ProvisionedThroughput(10L, 10L));
            table.waitForActive();
            System.out.println("Table created successfully.  Status: " +
table.getDescription().getTableStatus());
        } catch (Exception e) {
```

```
            System.err.println("Cannot create the table: ");
            System.err.println(e.getMessage());
        }
    }
}
```

In the above example, note the endpoint: **.withEndpoint**.

It indicates the use of a local install by using the localhost. Also, note the required **ProvisionedThroughput parameter**, which the local install ignores.

7. DynamoDB – Load Table

Loading a table generally consists of creating a source file, ensuring the source file conforms to a syntax compatible with DynamoDB, sending the source file to the destination, and then confirming a successful population.

Utilize the GUI console, Java, or another option to perform the task.

Load Table using GUI Console

Load data using a combination of the command line and console. You can load data in multiple ways, some of which are as follows:

- The Console
- The Command Line
- Code and also
- Data Pipeline (a feature discussed later in the tutorial)

However, for speed, this example uses both the shell and console. First, load the source data into the destination with the following syntax:

```
aws dynamodb batch-write-item --request-items file://[filename]
```

For example:

```
aws dynamodb batch-write-item --request-items file://MyProductData.json
```

Verify the success of the operation by accessing the console at –

https://console.aws.amazon.com/dynamodb

Choose **Tables** from the navigation pane, and select the destination table from the table list.

Select the **Items** tab to examine the data you used to populate the table. Select **Cancel** to return to the table list.

Load Table using Java

Employ Java by first creating a source file. Our source file uses JSON format. Each product has two primary key attributes (ID and Nomenclature) and a JSON map (Stat):

```
[
   {
      "ID" : ... ,
      "Nomenclature" : ... ,
      "Stat" : { ... }
```

```
   },
   {
      "ID" : ... ,
      "Nomenclature" : ... ,
      "Stat" : { ... } }
   },
    ...
]
```

You can review the following example:

```
{
    "ID" : 122,
    "Nomenclature" : "Particle Blaster 5000",
    "Stat" : {
        "Manufacturer" : "XYZ Inc.",
        "sales" : "1M+",
        "quantity" : 500,
        "img_src" : "http://www.xyz.com/manuals/particleblaster5000.jpg",
        "description" : "A laser cutter used in plastic manufacturing."
    }
}
```

The next step is to place the file in the directory used by your application.

Java primarily uses the **putItem** and **path methods** to perform the load.

You can review the following code example for processing a file and loading it:

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

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 ProductsLoadData {

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

        AmazonDynamoDBClient client = new AmazonDynamoDBClient()
            .withEndpoint("http://localhost:8000");

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

        ObjectNode currentNode;

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


            int ID = currentNode.path("ID").asInt();
            String Nomenclature = currentNode.path("Nomenclature").asText();

            try {
                table.putItem(new Item()
                .withPrimaryKey("ID", ID, "Nomenclature", Nomenclature)
                .withJSON("Stat", currentNode.path("Stat").toString()));
                System.out.println("Successful load: " + ID + " " + Nomenclature);

            } catch (Exception e) {
                System.err.println("Cannot add product: " + ID + " " + Nomenclature);
```

```
                System.err.println(e.getMessage());
                break;
            }
        }
        parser.close();
    }
}
```

8. DynamoDB – Query Table

Querying a table primarily requires selecting a table, specifying a partition key, and executing the query; with the options of using secondary indexes and performing deeper filtering through scan operations.

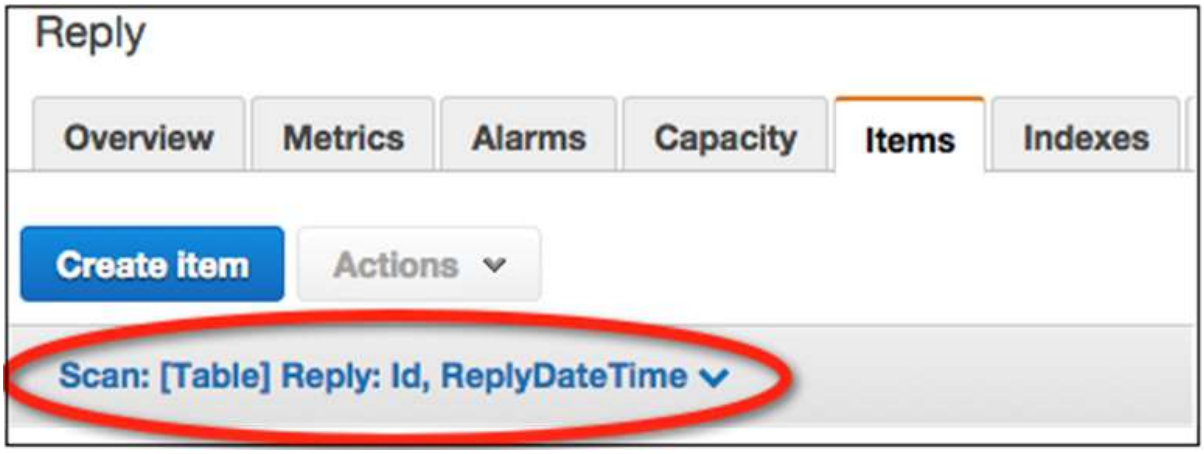
Utilize the GUI Console, Java, or another option to perform the task.

Query Table using the GUI Console

Perform some simple queries using the previously created tables. First, open the console at https://console.aws.amazon.com/dynamodb

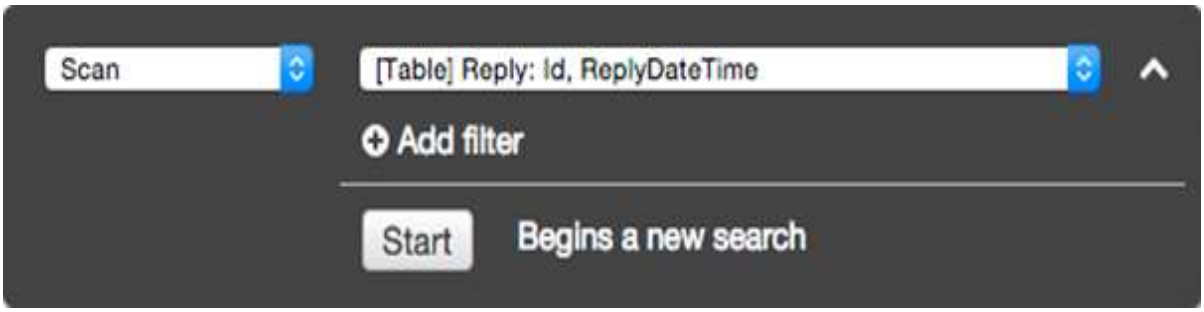
Choose **Tables** from the navigation pane and select **Reply** from the table list. Then select the **Items** tab to see the loaded data.

Select the data filtering link ("Scan: [Table] Reply") beneath the **Create Item** button.



In the filtering screen, select Query for the operation. Enter the appropriate partition key value, and click **Start**.

The **Reply** table then returns matching items.



Query Table using Java

Use the query method in Java to perform data retrieval operations. It requires specifying the partition key value, with the sort key as optional.

Code a Java query by first creating a **querySpec object** describing parameters. Then pass the object to the query method. We use the partition key from the previous examples.

You can review the following example:

```
import java.util.HashMap;
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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;
import com.amazonaws.services.dynamodbv2.document.utils.NameMap;

public class ProductsQuery {

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

        AmazonDynamoDBClient client = new AmazonDynamoDBClient()
            .withEndpoint("http://localhost:8000");

        DynamoDB dynamoDB = new DynamoDB(client);

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

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

        HashMap<String, Object> valueMap = new HashMap<String, Object>();
        valueMap.put(":xxx", 122);
```

```
        QuerySpec querySpec = new QuerySpec()
            .withKeyConditionExpression("#ID = :xxx")
            .withNameMap(new NameMap().with("#ID", "ID"))
            .withValueMap(valueMap);

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

        try {
            System.out.println("Product with the ID 122");
            items = table.query(querySpec);

            iterator = items.iterator();
            while (iterator.hasNext()) {
                item = iterator.next();
                System.out.println(item.getNumber("ID") + ": "
                        + item.getString("Nomenclature"));
            }

        } catch (Exception e) {
            System.err.println("Cannot find products with the ID number 122");
            System.err.println(e.getMessage());
        }
    }
}
```

Note that the query uses the partition key, however, secondary indexes provide another option for queries. Their flexibility allows querying of non-key attributes, a topic which will be discussed later in this tutorial.

The scan method also supports retrieval operations by gathering all the table data. The **optional .withFilterExpression** prevents items outside of specified criteria from appearing in results.

Later in this tutorial, we will discuss **scanning** in detail. Now, take a look at the following example:

```
import java.util.Iterator;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 ProductsScan {

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

        AmazonDynamoDBClient client = new AmazonDynamoDBClient()
            .withEndpoint("http://localhost:8000");

        DynamoDB dynamoDB = new DynamoDB(client);

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

        ScanSpec scanSpec = new ScanSpec()
            .withProjectionExpression("#ID, Nomenclature , stat.sales")
            .withFilterExpression("#ID between :start_id and :end_id")
            .withNameMap(new NameMap().with("#ID",  "ID"))
            .withValueMap(new                ValueMap().withNumber(":start_id",
120).withNumber(":end_id", 129));

        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("Cannot perform a table scan:");
            System.err.println(e.getMessage());
        }
    }
}
```

9. DynamoDB – Delete Table

In this chapter, we will discuss regarding how we can delete a table and also the different ways of deleting a table.

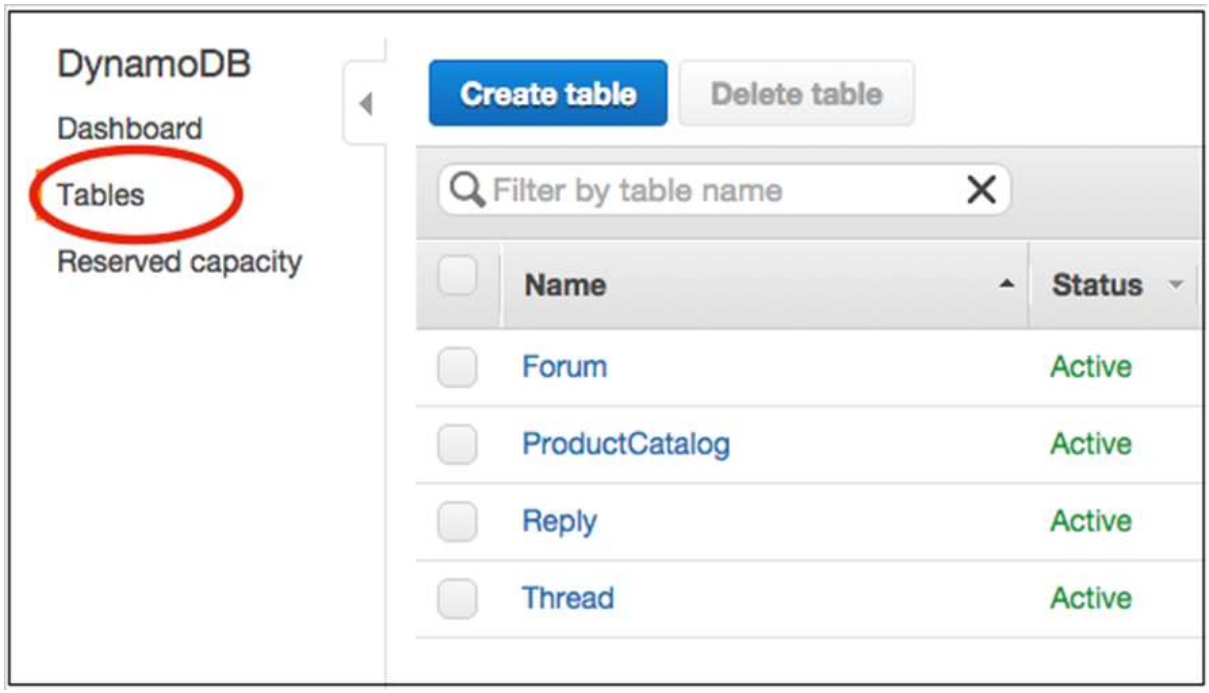
Table deletion is a simple operation requiring little more than the table name. Utilize the GUI console, Java, or any other option to perform this task.

Delete Table using the GUI Console

Perform a delete operation by first accessing the console at –

https://console.aws.amazon.com/dynamodb.

Choose **Tables** from the navigation pane, and choose the table desired for deletion from the table list as shown in the following screeenshot.



Finally, select **Delete Table**. After choosing Delete Table, a confirmation appears. Your table is then deleted.

Delete Table using Java

Use the **delete** method to remove a table. An example is given below to explain the concept better.

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;

public class ProductsDeleteTable {

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

        AmazonDynamoDBClient client = new AmazonDynamoDBClient()
            .withEndpoint("http://localhost:8000");

        DynamoDB dynamoDB = new DynamoDB(client);

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

        try {
            System.out.println("Performing table delete, wait...");
            table.delete();
            table.waitForDelete();
            System.out.print("Table successfully deleted.");

        } catch (Exception e) {
            System.err.println("Cannot perform table delete: ");
            System.err.println(e.getMessage());
        }
    }
}
```

10. DynamoDB – API Interface

DynamoDB offers a wide set of powerful API tools for table manipulation, data reads, and data modification.

Amazon recommends using **AWS SDKs** (e.g., the Java SDK) rather than calling low-level APIs. The libraries make interacting with low-level APIs directly unnecessary. The libraries simplify common tasks such as authentication, serialization, and connections.

Manipulate Tables

DynamoDB offers five low-level actions for Table Management:

- **CreateTable** – This spawns a table and includes throughput set by the user. It requires you to set a primary key, whether composite or simple. It also allows one or multiple secondary indexes.
- **ListTables** – This provides a list of all tables in the current AWS user's account and tied to their endpoint.
- **UpdateTable** – This alters throughput, and global secondary index throughput.
- **DescribeTable** – This provides table metadata; for example, state, size, and indices.
- **DeleteTable** – This simply erases the table and its indices.

Read Data

DynamoDB offers four low-level actions for data reading:

- **GetItem** – It accepts a primary key and returns attributes of the associated item. It permits changes to its default eventually consistent read setting.
- **BatchGetItem** – It executes several GetItem requests on multiple items through primary keys, with the option of one or multiple tables. Its returns no more than 100 items and must remain under 16MB. It permits eventually consistent and strongly consistent reads.
- **Scan** – It reads all the table items and produces an eventually consistent result set. You can filter results through conditions. It avoids the use of an index and scans the entire table, so do not use it for queries requiring predictability.
- **Query** – It returns a single or multiple table items or secondary index items. It uses a specified value for the partition key, and permits the use of comparison operators to narrow scope. It includes support for both types of consistency, and each response obeys a 1MB limit in size.

Modify Data

DynamoDB offers four low-level actions for data modification:

- **PutItem** – This spawns a new item or replaces existing items. On discovery of identical primary keys, by default, it replaces the item. Conditional operators allow you to work around the default, and only replace items under certain conditions.

- **BatchWriteItem** – This executes both multiple PutItem and DeleteItem requests, and over several tables. If one request fails, it does not impact the entire operation. Its cap sits at 25 items, and 16MB in size.

- **UpdateItem** – It changes the existing item attributes, and permits the use of conditional operators to execute updates only under certain conditions.

- **DeleteItem** – It uses the primary key to erase an item, and also allows the use of conditional operators to specify the conditions for deletion.

11. DynamoDB – Creating Items

Creating an item in DynamoDB consists primarily of item and attribute specification, and the option of specifying conditions. Each item exists as a set of attributes, with each attribute named and assigned a value of a certain type.

Value types include scalar, document, or set. Items carry a 400KB size limit, with the possibility of any amount of attributes capable of fitting within that limit. Name and value sizes (binary and UTF-8 lengths) determine item size. Using short attribute names aids in minimizing item size.

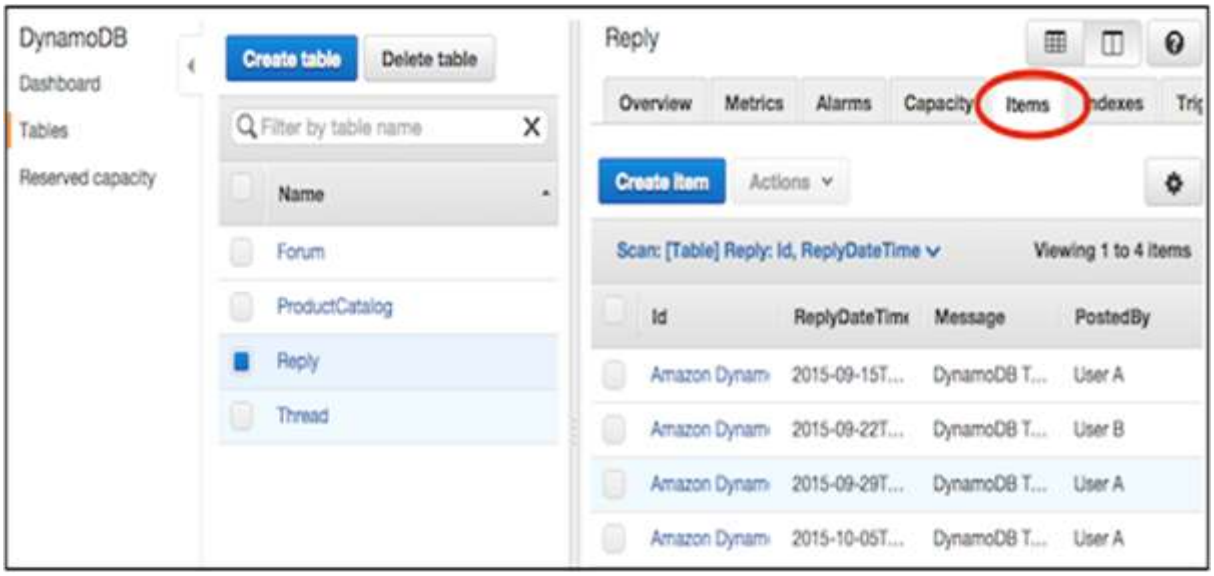
Note: *You must specify all primary key attributes, with primary keys only requiring the partition key; and composite keys requiring both the partition and sort key.*

Also, remember tables possess no predefined schema. You can store dramatically different datasets in one table.

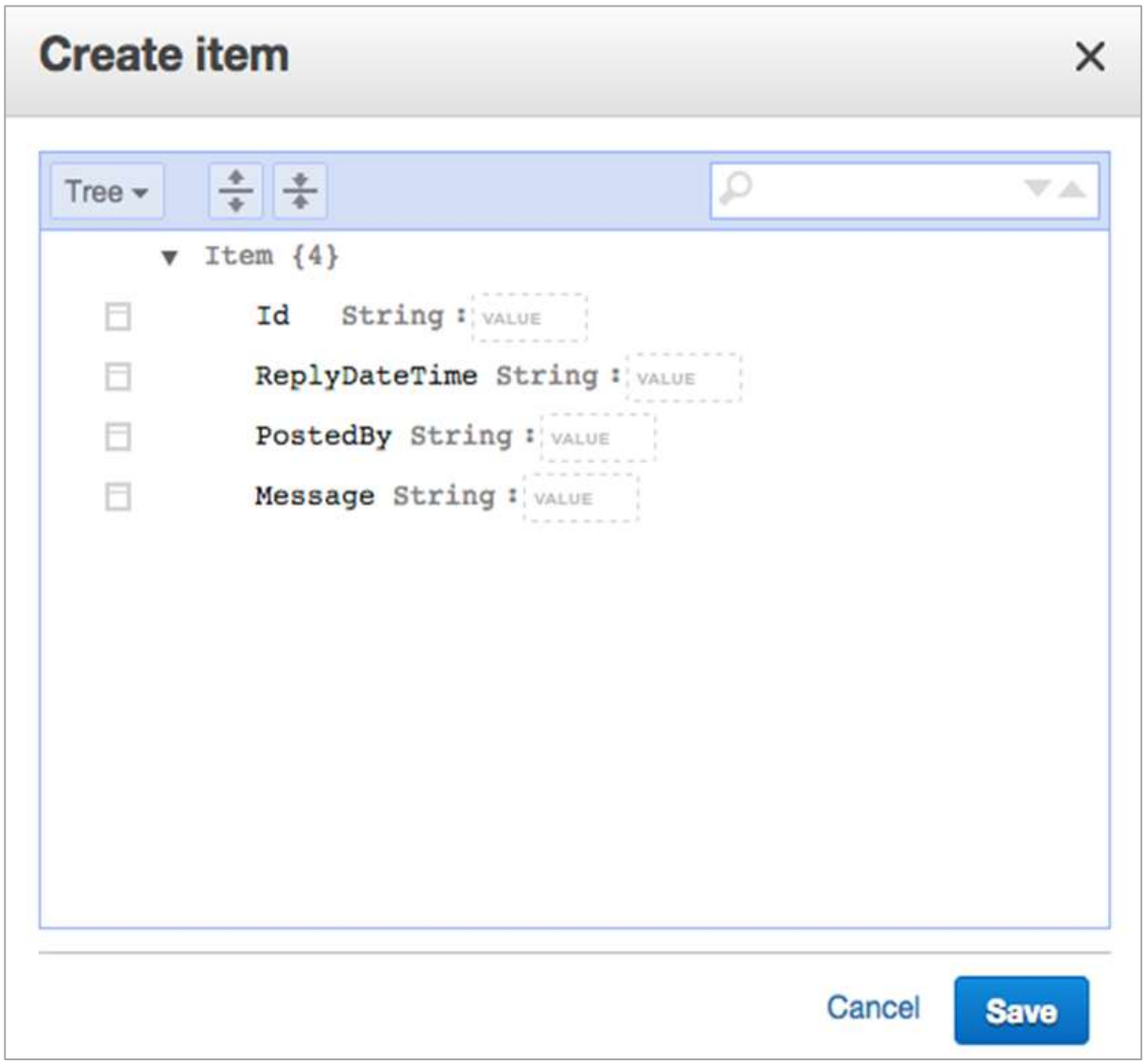
Use the GUI console, Java, or another tool to perform this task.

How to Create an Item Using the GUI Console?

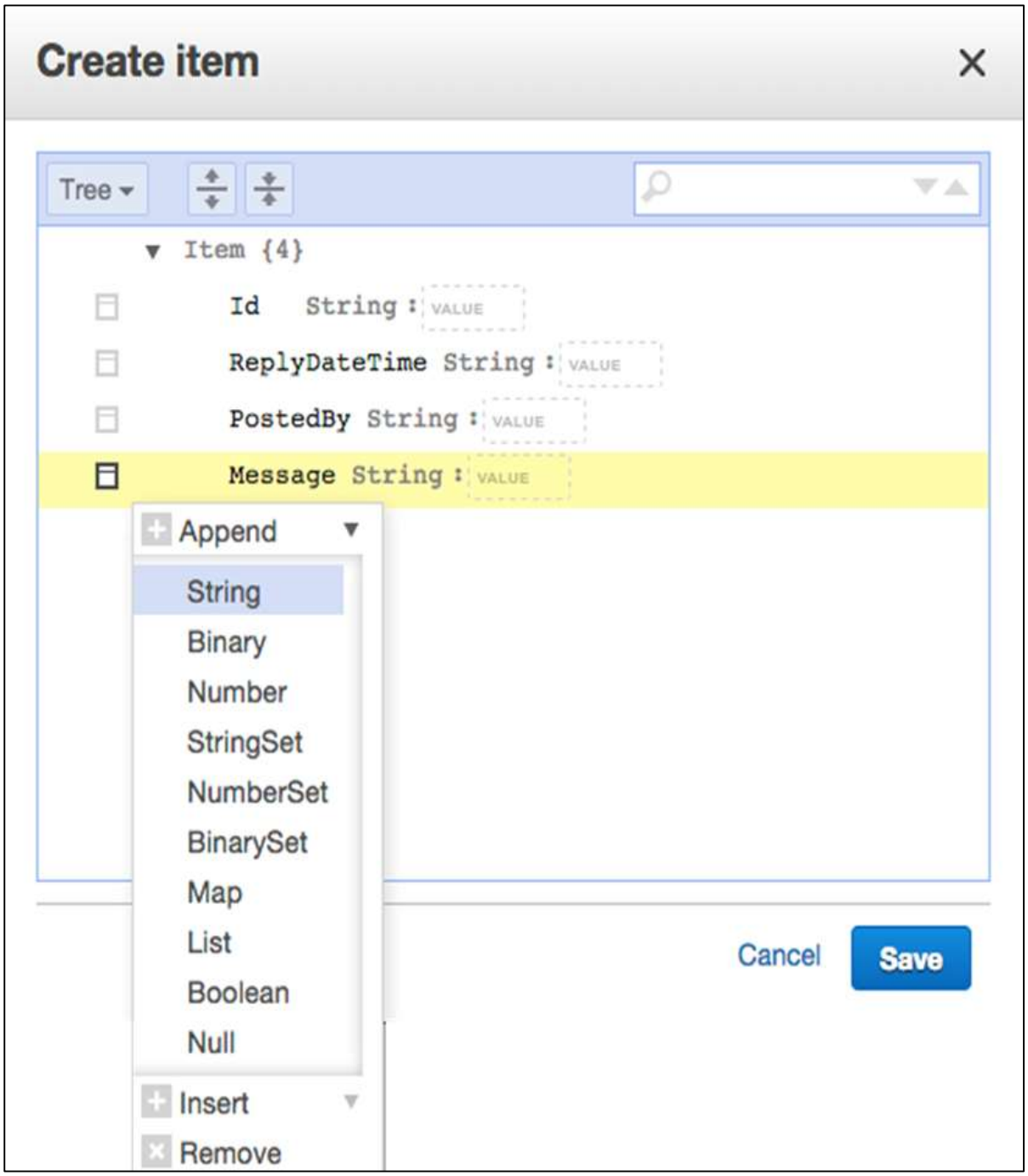
Navigate to the console. In the navigation pane on the left side, select **Tables**. Choose the table name for use as the destination, and then select the **Items** tab as shown in the following screenshot.



Select **Create Item**. The Create Item screen provides an interface for entering the required attribute values. Any secondary indices must also be entered.



If you require more attributes, select the action menu on the left of the **Message**. Then select **Append**, and the desired data type.



After entering all essential information, select **Save** to add the item.

How to Use Java in Item Creation?

Using Java in item creation operations consists of creating a DynamoDB class instance, Table class instance, Item class instance, and specifying the primary key and attributes of the item you will create. Then add your new item with the putItem method.

Example

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

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

// Spawn a related items list
List<Number> RELItems = new ArrayList<Number>();
RELItems.add(123);
RELItems.add(456);
RELItems.add(789);

//Spawn a product picture map
Map<String, String> photos = new HashMap<String, String>();
photos.put("Anterior", "http://xyz.com/products/101_front.jpg");
photos.put("Posterior", "http://xyz.com/products/101_back.jpg");
photos.put("Lateral", "http://xyz.com/products/101_LFTside.jpg");

//Spawn a product review map
Map<String, List<String>> prodReviews = new HashMap<String, List<String>>();

List<String> fiveStarRVW = new ArrayList<String>();
fiveStarRVW.add("Shocking high performance.");
fiveStarRVW.add("Unparalleled in its market.");
prodReviews.put("5 Star", fiveStarRVW);

List<String> oneStarRVW = new ArrayList<String>();
oneStarRVW.add("The worst offering in its market.");
prodReviews.put("1 Star", oneStarRVW);

// Generate the item
Item item = new Item()
```

```
    .withPrimaryKey("Id", 101)
    .withString("Nomenclature", "PolyBlaster 101")
    .withString("Description", "101 description")
    .withString("Category", "Hybrid Power Polymer Cutter")

    .withString("Make", "Brand – XYZ")
    .withNumber("Price", 50000)
    .withString("ProductCategory", "Laser Cutter")
    .withBoolean("Availability", true)
    .withNull("Qty")
    .withList("ItemsRelated", RELItems)
    .withMap("Images", photos)
    .withMap("Reviews", prodReviews);

// Add item to the table
PutItemOutcome outcome = table.putItem(item);
```

You can also look at the following larger example.

Note: *The following sample may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).*

The following sample also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 CreateItemOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    static String tblName = "ProductList";

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

        createItems();

        retrieveItem();

        // Execute updates
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Item deletion
        deleteItem();

    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tblName);
        try {

            Item item = new Item()
                .withPrimaryKey("ID", 303)
```

```
                .withString("Nomenclature", "Polymer Blaster 4000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc.", "LMNOP Inc.")))

                .withNumber("Price", 50000)
                .withBoolean("InProduction", true)
                .withString("Category", "Laser Cutter");
            table.putItem(item);

            item = new Item()
                .withPrimaryKey("ID", 313)
                .withString("Nomenclature", "Agitatatron 2000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc,", "CDE Inc.")))
                .withNumber("Price", 40000)
                .withBoolean("InProduction", true)
                .withString("Category", "Agitator");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Cannot create items.");
            System.err.println(e.getMessage());
        }
    }
}
```

12. DynamoDB – Getting Items

Retrieving an item in DynamoDB requires using GetItem, and specifying the table name and item primary key. Be sure to include a complete primary key rather than omitting a portion.

For example, omitting the sort key of a composite key.

GetItem behavior conforms to three defaults:

- It executes as an eventually consistent read.
- It provides all attributes.
- It does not detail its capacity unit consumption.

These parameters allow you to override the default GetItem behavior.

Retrieve an Item

DynamoDB ensures reliability through maintaining multiple copies of items across multiple servers. Each successful write creates these copies, but takes substantial time to execute; meaning eventually consistent. This means you cannot immediately attempt a read after writing an item.

You can change the default eventually consistent read of GetItem, however, the cost of more current data remains consumption of more capacity units; specifically, two times as much. Note DynamoDB typically achieves consistency across every copy within a second.

You can use the GUI console, Java, or another tool to perform this task.

Item Retrieval Using Java

Using Java in item retrieval operations requires creating a DynamoDB Class Instance, Table Class Instance, and calling the Table instance's getItem method. Then specify the primary key of the item.

You can review the following example:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

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

Item item = table.getItem("IDnum", 109);
```

In some cases, you need to specify the parameters for this operation.

The following example uses **.withProjectionExpression** and **GetItemSpec** for retrieval specifications:

```
GetItemSpec spec = new GetItemSpec()
    .withPrimaryKey("IDnum", 122)
    .withProjectionExpression("IDnum, EmployeeName, Department")
    .withConsistentRead(true);

Item item = table.getItem(spec);

System.out.println(item.toJSONPretty());
```

You can also review a the following bigger example for better understanding.

Note: The following sample may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This sample also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 GetItemOpSample {
```

```
static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
        new ProfileCredentialsProvider()));

static String tblName = "ProductList";

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

    createItems();

    retrieveItem();

    // Execute updates
    updateMultipleAttributes();
    updateAddNewAttribute();
    updateExistingAttributeConditionally();

    // Item deletion
    deleteItem();

}

private static void createItems() {

    Table table = dynamoDB.getTable(tblName);
    try {

        Item item = new Item()
            .withPrimaryKey("ID", 303)
            .withString("Nomenclature", "Polymer Blaster 4000")
            .withStringSet( "Manufacturers",
                new HashSet<String>(Arrays.asList("XYZ Inc.", "LMNOP Inc.")))
            .withNumber("Price", 50000)
            .withBoolean("InProduction", true)
            .withString("Category", "Laser Cutter");

        table.putItem(item);
```

```
            item = new Item()
                .withPrimaryKey("ID", 313)
                .withString("Nomenclature", "Agitatatron 2000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc,", "CDE Inc.")))
                .withNumber("Price", 40000)
                .withBoolean("InProduction", true)
                .withString("Category", "Agitator");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Cannot create items.");
            System.err.println(e.getMessage());
        }
    }

    private static void retrieveItem() {
        Table table = dynamoDB.getTable(tableName);

        try {
            Item item = table.getItem("ID", 303, "ID, Nomenclature,
Manufacturers", null);

            System.out.println("Displaying retrieved items...");
            System.out.println(item.toJSONPretty());

        } catch (Exception e) {
            System.err.println("Cannot retrieve items.");
            System.err.println(e.getMessage());
        }
    }
}
```

13. DynamoDB – Update Items

Updating an item in DynamoDB mainly consists of specifying the full primary key and table name for the item. It requires a new value for each attribute you modify. The operation uses **UpdateItem**, which modifies the existing items or creates them on discovery of a missing item.

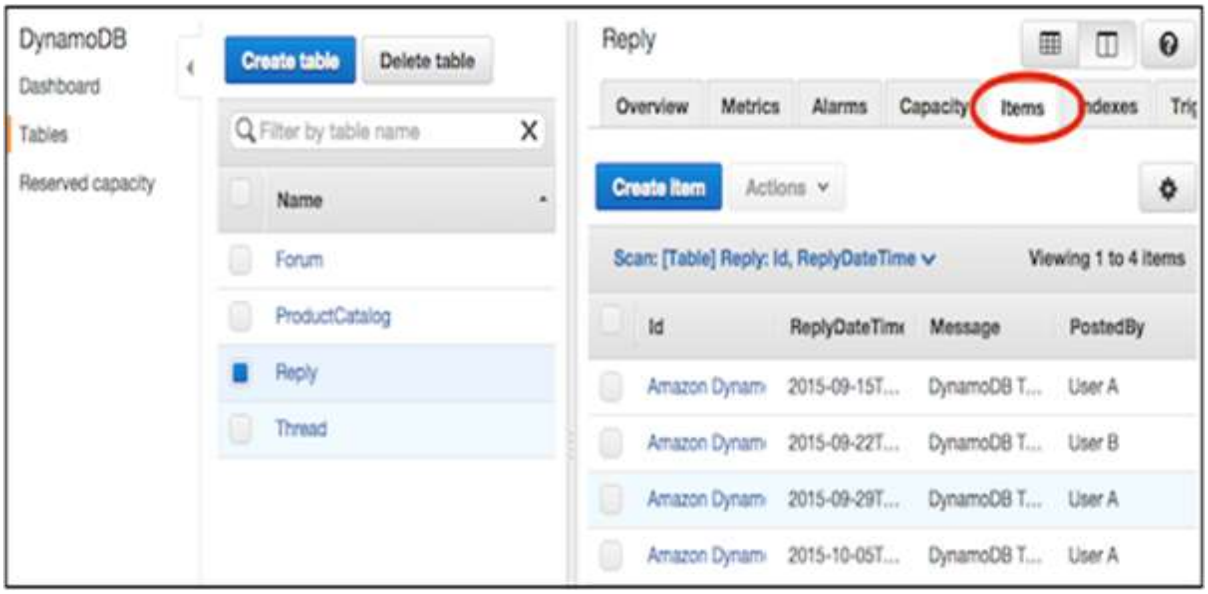
In updates, you might want to track the changes by displaying the original and new values, before and after the operations. UpdateItem uses the **ReturnValues** parameter to achieve this.

Note: *The operation does not report capacity unit consumption, but you can use the* ***ReturnConsumedCapacity*** *parameter.*

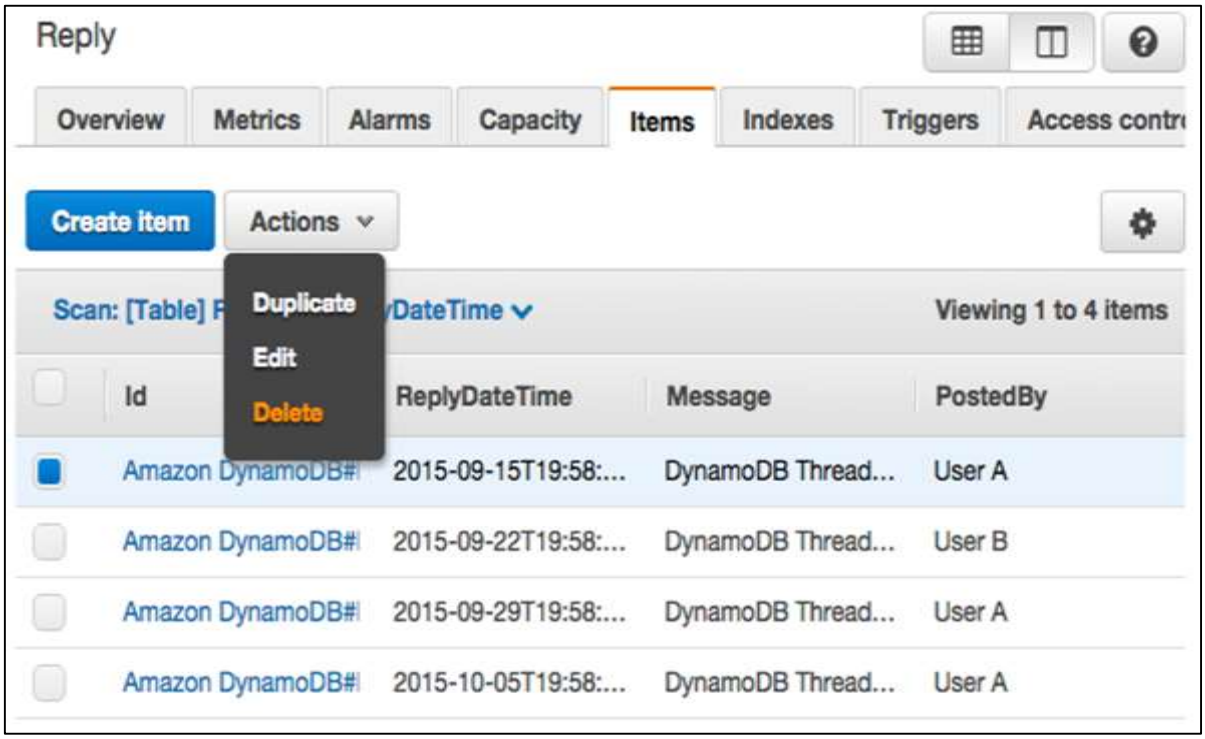
Use the GUI console, Java, or any other tool to perform this task.

How to Update Items Using GUI Tools?

Navigate to the console. In the navigation pane on the left side, select **Tables**. Choose the table needed, and then select the **Items** tab.



Choose the item desired for an update, and select **Actions | Edit**.



Modify any attributes or values necessary in the **Edit Item** window.

Update Items Using Java

Using Java in the item update operations requires creating a Table class instance, and calling its **updateItem** method. Then you specify the item's primary key, and provide an **UpdateExpression** detailing attribute modifications.

The Following is an example of the same:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
                    new ProfileCredentialsProvider()));

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

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#M", "Make");
expressionAttributeNames.put("#P", "Price");

expressionAttributeNames.put("#N", "ID");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
```

```
expressionAttributeValues.put(":val1",
    new HashSet<String>(Arrays.asList("Make1","Make2")));
expressionAttributeValues.put(":val2", 1);   //Price

UpdateItemOutcome outcome =  table.updateItem(
    "internalID",          // key attribute name
    111,           // key attribute value
    "add #M :val1 set #P = #P - :val2 remove #N", // UpdateExpression
    expressionAttributeNames,
    expressionAttributeValues);
```

The **updateItem** method also allows for specifying conditions, which can be seen in the following example:

```
Table table = dynamoDB.getTable("ProductList");

Map<String, String> expressionAttributeNames = new HashMap<String, String>();
expressionAttributeNames.put("#P", "Price");

Map<String, Object> expressionAttributeValues = new HashMap<String, Object>();
expressionAttributeValues.put(":val1", 44);  // change Price to 44
expressionAttributeValues.put(":val2", 15);  // only if currently 15

UpdateItemOutcome outcome = table.updateItem(
    new PrimaryKey("internalID",111),
    "set #P = :val1", // Update
    "#P = :val2",     // Condition
    expressionAttributeNames,
    expressionAttributeValues);
```

Update Items Using Counters

DynamoDB allows atomic counters, which means using UpdateItem to increment/decrement attribute values without impacting other requests; furthermore, the counters always update.

The following is an example that explains how it can be done.

Note: The following sample may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This sample also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 UpdateItemOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    static String tblName = "ProductList";

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

        createItems();

        retrieveItem();

        // Execute updates
        updateMultipleAttributes();
        updateAddNewAttribute();
```

```
        updateExistingAttributeConditionally();

        // Item deletion
        deleteItem();
    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tblName);
        try {
            Item item = new Item()
                .withPrimaryKey("ID", 303)
                .withString("Nomenclature", "Polymer Blaster 4000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc.", "LMNOP Inc.")))
                .withNumber("Price", 50000)
                .withBoolean("InProduction", true)
                .withString("Category", "Laser Cutter");
            table.putItem(item);

            item = new Item()
                .withPrimaryKey("ID", 313)
                .withString("Nomenclature", "Agitatatron 2000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc,", "CDE Inc.")))
                .withNumber("Price", 40000)
                .withBoolean("InProduction", true)
                .withString("Category", "Agitator");

            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Cannot create items.");
            System.err.println(e.getMessage());
        }
    }
```

```
    private static void updateAddNewAttribute() {
        Table table = dynamoDB.getTable(tableName);

        try {

            Map<String, String> expressionAttributeNames = new HashMap<String,
String>();
            expressionAttributeNames.put("#na", "NewAttribute");

            UpdateItemSpec updateItemSpec = new UpdateItemSpec()
            .withPrimaryKey("ID", 303)
            .withUpdateExpression("set #na = :val1")
            .withNameMap(new NameMap()
                .with("#na", "NewAttribute"))
            .withValueMap(new ValueMap()
                .withString(":val1", "A value"))
            .withReturnValues(ReturnValue.ALL_NEW);

            UpdateItemOutcome outcome =  table.updateItem(updateItemSpec);

            // Confirm
            System.out.println("Displaying updated item...");
            System.out.println(outcome.getItem().toJSONPretty());

        }   catch (Exception e) {
            System.err.println("Cannot add an attribute in " + tableName);
            System.err.println(e.getMessage());
        }
    }
}
```

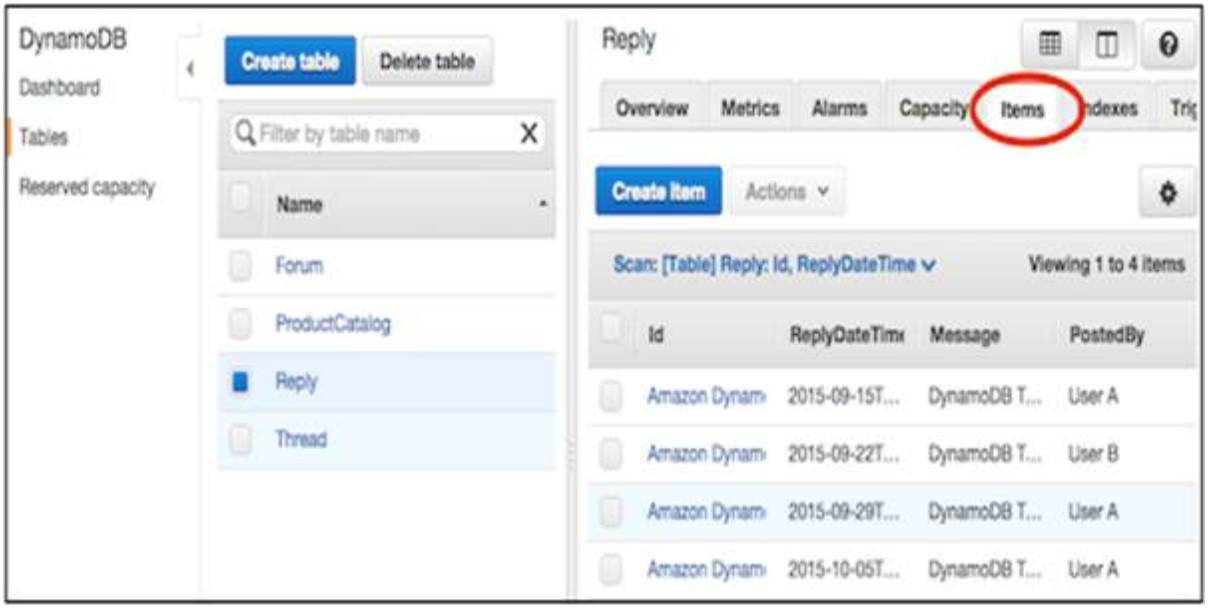
14. DynamoDB – Delete Items

Deleting an item in the DynamoDB only requires providing the table name and the item key. It is also strongly recommended to use of a conditional expression which will be necessary to avoid deleting the wrong items.

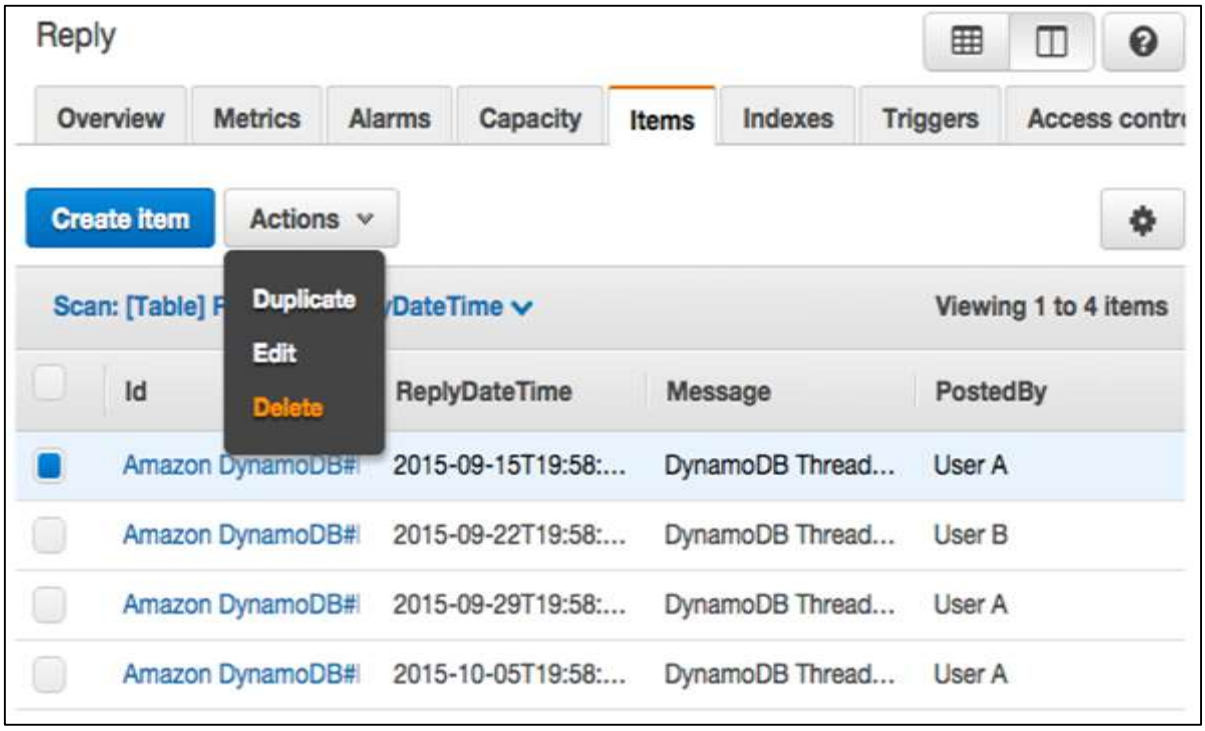
As usual, you can either use the GUI console, Java, or any other needed tool to perform this task.

Delete Items Using the GUI Console

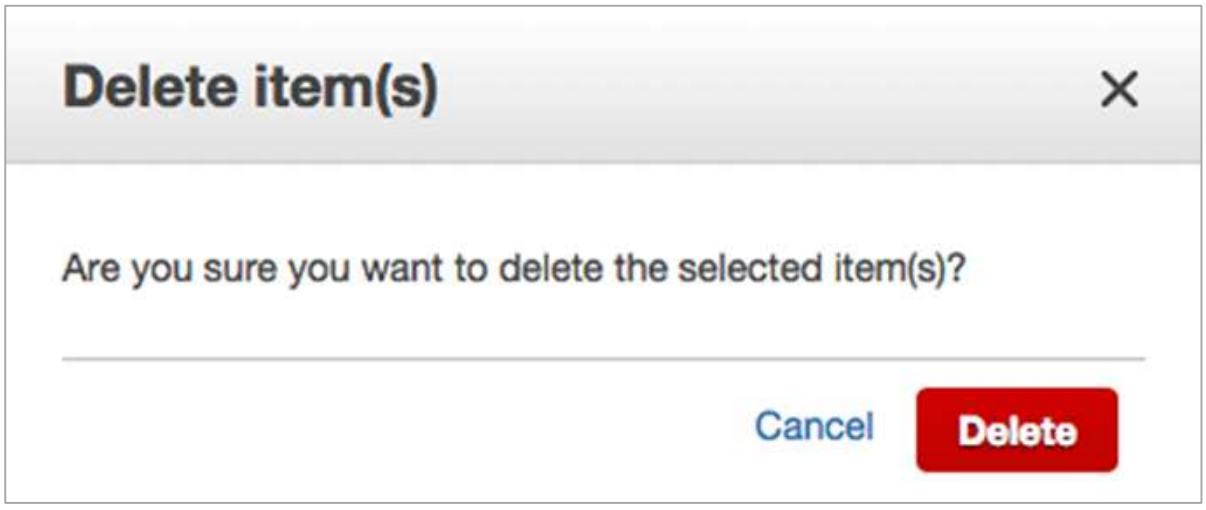
Navigate to the console. In the navigation pane on the left side, select **Tables**. Then select the table name, and the **Items** tab.



Choose the items desired for deletion, and select **Actions | Delete**.



A **Delete Item(s)** dialog box then appears as shown in the following screeshot. Choose "Delete" to confirm.



How to Delete Items Using Java?

Using Java in item deletion operations merely involves creating a DynamoDB client instance, and calling the **deleteItem** method through using the item's key.

You can see the following example, where it has been explained in detail.

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
                    new ProfileCredentialsProvider()));

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

DeleteItemOutcome outcome = table.deleteItem("IDnum", 151);
```

You can also specify the parameters to protect against incorrect deletion. Simply use a **ConditionExpression**.

For example:

```
Map<String,Object> expressionAttributeValues = new HashMap<String,Object>();
expressionAttributeValues.put(":val", false);

DeleteItemOutcome outcome = table.deleteItem("IDnum",151,
    "Ship = :val",
    null, // doesn't use ExpressionAttributeNames
    expressionAttributeValues);
```

The following is a larger example for better understanding.

Note: The following sample may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This sample also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
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.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 DeleteItemOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    static String tblName = "ProductList";

    public static void main(String[] args) throws IOException {
        createItems();
        retrieveItem();

        // Execute updates
        updateMultipleAttributes();
        updateAddNewAttribute();
        updateExistingAttributeConditionally();

        // Item deletion
        deleteItem();

    }

    private static void createItems() {

        Table table = dynamoDB.getTable(tblName);
        try {

            Item item = new Item()

                .withPrimaryKey("ID", 303)
                .withString("Nomenclature", "Polymer Blaster 4000")
```

```
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc.", "LMNOP Inc.")))
                .withNumber("Price", 50000)
                .withBoolean("InProduction", true)
                .withString("Category", "Laser Cutter");
            table.putItem(item);

            item = new Item()
                .withPrimaryKey("ID", 313)
                .withString("Nomenclature", "Agitatatron 2000")
                .withStringSet( "Manufacturers",
                    new HashSet<String>(Arrays.asList("XYZ Inc,", "CDE Inc.")))
                .withNumber("Price", 40000)
                .withBoolean("InProduction", true)
                .withString("Category", "Agitator");
            table.putItem(item);

        } catch (Exception e) {
            System.err.println("Cannot create items.");
            System.err.println(e.getMessage());

        }
    }

    private static void deleteItem() {

        Table table = dynamoDB.getTable(tableName);

        try {

            DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
            .withPrimaryKey("ID", 303)

            .withConditionExpression("#ip = :val")
            .withNameMap(new NameMap()
                .with("#ip", "InProduction"))
```

```
            .withValueMap(new ValueMap()
            .withBoolean(":val", false))
            .withReturnValues(ReturnValue.ALL_OLD);

            DeleteItemOutcome outcome = table.deleteItem(deleteItemSpec);

            // Confirm
            System.out.println("Displaying deleted item...");
            System.out.println(outcome.getItem().toJSONPretty());

        } catch (Exception e) {
            System.err.println("Cannot delete item in " + tableName);
            System.err.println(e.getMessage());
        }
    }
}
```

15. DynamoDB – Batch Writing

Batch writing operates on multiple items by creating or deleting several items. These operations utilize **BatchWriteItem**, which carries the limitations of no more than 16MB writes and 25 requests. Each item obeys a 400KB size limit. Batch writes also cannot perform item updates.

What is Batch Writing?

Batch writes can manipulate items across multiple tables. Operation invocation happens for each individual request, which means operations do not impact each other, and heterogeneous mixes are permitted; for example, one **PutItem** and three **DeleteItem** requests in a batch, with the failure of the PutItem request not impacting the others. Failed requests result in the operation returning information (keys and data) pertaining to each failed request.

Note: If DynamoDB returns any items without processing them, retry them; however, use a back-off method to avoid another request failure based on overloading.

DynamoDB rejects a batch write operation when one or more of the following statements proves to be true:

- The request exceeds the provisioned throughput.
- The request attempts to use **BatchWriteItems** to update an item.
- The request performs several operations on a single item.
- The request tables do not exist.
- The item attributes in the request do not match the target.
- The requests exceed size limits.

Batch writes require certain **RequestItem** parameters:

- Deletion operations need **DeleteRequest** key **subelements** meaning an attribute name and value.
- The **PutRequest** items require an **Item subelement** meaning an attribute and attribute value map.

Response: A successful operation results in an HTTP 200 response, which indicates characteristics like capacity units consumed, table processing metrics, and any unprocessed items.

Batch Writes with Java

Perform a batch write by creating a DynamoDB class instance, a **TableWriteItems** class instance describing all operations, and calling the **batchWriteItem** method to use the TableWriteItems object.

Note: You must create a TableWriteItems instance for every table in a batch write to multiple tables. Also, check your request response for any unprocessed requests.

You can review the following example of a batch write:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

TableWriteItems forumTableWriteItems = new TableWriteItems("Forum")
    .withItemsToPut(
        new Item()
            .withPrimaryKey("Title", "XYZ CRM")
            .withNumber("Threads", 0));

TableWriteItems threadTableWriteItems = new TableWriteItems(Thread)
    .withItemsToPut(
        new Item()
            .withPrimaryKey("ForumTitle","XYZ CRM","Topic","Updates")
    .withHashAndRangeKeysToDelete("ForumTitle","A partition key value", "Product
Line 1", "A sort key value");

BatchWriteItemOutcome outcome = dynamoDB.batchWriteItem(forumTableWriteItems,
threadTableWriteItems);
```

The following program is another bigger example for better understanding of how a batch writes with Java.

Note: The following example may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This example also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
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.auth.profile.ProfileCredentialsProvider;
```

```
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 BatchWriteOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    static String forumTableName = "Forum";
    static String threadTableName = "Thread";

    public static void main(String[] args) throws IOException {
        batchWriteMultiItems();
    }

    private static void batchWriteMultiItems() {
        try {

            // Place new item in Forum
            TableWriteItems forumTableWriteItems = new
TableWriteItems(forumTableName) //Forum
                .withItemsToPut(new Item()
                    .withPrimaryKey("Name", "Amazon RDS")
                    .withNumber("Threads", 0));

            // Place one item, delete another in Thread
            // Specify partition key and range key
            TableWriteItems threadTableWriteItems = new
TableWriteItems(threadTableName)
            .withItemsToPut(new Item()
            .withPrimaryKey("ForumName","Product
Support","Subject","Support Thread 1")
            .withString("Message", "New OS Thread 1 message")
```

```
            .withHashAndRangeKeysToDelete("ForumName","Subject", "Polymer
Blaster", "Support Thread 100");


            System.out.println("Processing request...");
            BatchWriteItemOutcome outcome =
dynamoDB.batchWriteItem(forumTableWriteItems, threadTableWriteItems);


            do {


                // Confirm no unprocessed items
                Map<String, List<WriteRequest>> unprocessedItems =
outcome.getUnprocessedItems();


                if (outcome.getUnprocessedItems().size() == 0) {
                    System.out.println("All items processed.");
                } else {
                    System.out.println("Gathering unprocessed items...");
                    outcome =
dynamoDB.batchWriteItemUnprocessed(unprocessedItems);
                }


            } while (outcome.getUnprocessedItems().size() > 0);


        }  catch (Exception e) {
            System.err.println("Could not get items: ");
            e.printStackTrace(System.err);
        }
    }
}
```

16. DynamoDB – Batch Retrieve

Batch Retrieve operations return attributes of a single or multiple items. These operations generally consist of using the primary key to identify the desired item(s). The **BatchGetItem** operations are subject to the limits of individual operations as well as their own unique constraints.

The following requests in batch retrieval operations result in rejection:

- Make a request for more than 100 items.
- Make a request exceeding throughput.

Batch retrieve operations perform partial processing of requests carrying the potential to exceed limits.

For example: a request to retrieve multiple items large enough in size to exceed limits results in part of the request processing, and an error message noting the unprocessed portion. On return of unprocessed items, create a back-off algorithm solution to manage this rather than throttling tables.

The **BatchGet** operations perform eventually with consistent reads, requiring modification for strongly consistent ones. They also perform retrievals in parallel.

Note: The order of the returned items. DynamoDB does not sort the items. It also does not indicate the absence of the requested items. Furthermore, those requests consume capacity units.

All the BatchGet operations require **RequestItems** parameters such as the read consistency, attribute names, and primary keys.

Response: A successful operation results in an HTTP 200 response, which indicates characteristics like capacity units consumed, table processing metrics, and any unprocessed items.

Batch Retrievals with Java

Using Java in BatchGet operations requires creating a DynamoDB class instance, **TableKeysAndAttributes** class instance describing a primary key values list for the items, and passing the TableKeysAndAttributes object to the **BatchGetItem** method.

The following is an example of a BatchGet operation:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

    TableKeysAndAttributes forumTableKeysAndAttributes = new
TableKeysAndAttributes(forumTableName);
    forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Title",
```

```
    "Updates",
    "Product Line 1");


TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumTitle", "Topic",
    "Product Line 1","P1 Thread 1",
    "Product Line 1","P1 Thread 2",
    "Product Line 2","P2 Thread 1");


BatchGetItemOutcome outcome = dynamoDB.batchGetItem(
    forumTableKeysAndAttributes, threadTableKeysAndAttributes);


for (String tableName : outcome.getTableItems().keySet()) {
    System.out.println("Table items " + tableName);
    List<Item> items = outcome.getTableItems().get(tableName);
    for (Item item : items) {
        System.out.println(item);
    }
}
```

You can review the following larger example.

Note: The following program may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This program also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
package com.amazonaws.codesamples.document;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 BatchGetOpSample {
     static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
                 new ProfileCredentialsProvider()));

    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);
            //Create partition key
            forumTableKeysAndAttributes.addHashOnlyPrimaryKeys("Name", "XYZ
Melt-O-tron", "High-Performance Processing");

            TableKeysAndAttributes threadTableKeysAndAttributes = new
TableKeysAndAttributes(threadTableName);
            //Create partition key and sort key

threadTableKeysAndAttributes.addHashAndRangePrimaryKeys("ForumName", "Subject",
                "High-Performance Processing","HP Processing Thread One",
                "High-Performance Processing","HP Processing Thread Two",
                "Melt-O-Tron","MeltO Thread One");



            System.out.println("Processing...");

            BatchGetItemOutcome outcome =
dynamoDB.batchGetItem(forumTableKeysAndAttributes,
```

```
                threadTableKeysAndAttributes);

            Map<String, KeysAndAttributes> unprocessed = null;

            do {
                for (String tableName : outcome.getTableItems().keySet()) {
                    System.out.println("Table items for " + tableName);
                    List<Item> items = outcome.getTableItems().get(tableName);
                    for (Item item : items) {
                        System.out.println(item.toJSONPretty());
                    }
                }

                // Confirm no unprocessed items
                unprocessed = outcome.getUnprocessedKeys();

                if (unprocessed.isEmpty()) {
                    System.out.println("All items processed.");
                } else {
                    System.out.println("Gathering unprocessed items...");
                    outcome = dynamoDB.batchGetItemUnprocessed(unprocessed);
                }

            } while (!unprocessed.isEmpty());

        }  catch (Exception e) {
            System.err.println("Could not get items.");
            System.err.println(e.getMessage());
        }
    }
}
```

17. DynamoDB – Querying

Queries locate items or secondary indices through primary keys. Performing a query requires a partition key and specific value, or a sort key and value; with the option to filter with comparisons. The default behavior of a query consists of returning every attribute for items associated with the provided primary key. However, you can specify the desired attributes with the **ProjectionExpression** parameter.

A query utilizes the **KeyConditionExpression** parameters to select items, which requires providing the partition key name and value in the form of an equality condition. You also have the option to provide an additional condition for any sort keys present.

A few examples of the sort key conditions are:

Condition	Description
x = y	It evaluates to true if the attribute x equals y.
x < y	It evaluates to true if x is less than y.
x <= y	It evaluates to true if x is less than or equal to y.
x > y	It evaluates to true if x is greater than y.
x >= y	It evaluates to true if x is greater than or equal to y.
x BETWEEN y AND z	It evaluates to true if x is both >= y, and <= z.

DynamoDB also supports the following functions: **begins_with (x, substr)**

It evaluates to true if attribute x starts with the specified string.

The following conditions must conform to certain requirements:

- Attribute names must start with a character within the a-z or A-Z set.
- The second character of an attribute name must fall in the a-z, A-Z, or 0-9 set.
- Attribute names cannot use reserved words.

Attribute names out of compliance with the constraints above can define a placeholder.

The query processes by performing retrievals in sort key order, and using any condition and filter expressions present. Queries always return a result set, and on no matches, it returns an empty one.

The results always return in sort key order, and data type based order with the modifiable default as the ascending order.

Querying with Java

Queries in Java allow you to query tables and secondary indices. They require specification of partition keys and equality conditions, with the option to specify sort keys and conditions.

The general required steps for a query in Java include creating a DynamoDB class instance, Table class instance for the target table, and calling the query method of the Table instance to receive the query object.

The response to the query contains an **ItemCollection** object providing all the returned items.

The following example demonstrates detailed querying:

```
DynamoDB dynamoDB = new DynamoDB(
    new AmazonDynamoDBClient(new ProfileCredentialsProvider()));

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

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("ID = :nn")
    .withValueMap(new ValueMap()
        .withString(":nn", "Product Line 1#P1 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());
```

The query method supports a wide variety of optional parameters. The following example demonstrates how to utilize these parameters:

```
Table table = dynamoDB.getTable("Response");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("ID = :nn and ResponseTM > :nn_responseTM")

    .withFilterExpression("Author = :nn_author")
    .withValueMap(new ValueMap()
```

```
        .withString(":nn", "Product Line 1#P1 Thread 1")
        .withString(":nn_responseTM", twoWeeksAgoStr)
        .withString(":nn_author", "Member 123"))
    .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 review the following larger example.

Note: The following program may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This example also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
package com.amazonaws.codesamples.document;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Iterator;
import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 QueryOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new
ProfileCredentialsProvider()));
```

```
    static String tableName = "Reply";

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

        String forumName = "PolyBlaster";
        String threadSubject = "PolyBlaster Thread 1";

        getThreadReplies(forumName, threadSubject);
    }

    private static void getThreadReplies(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("\ngetThreadReplies results:");
        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}
```

18. DynamoDB – Scan

Scan Operations read all table items or secondary indices. Its default function results in returning all data attributes of all items within an index or table. Employ the **ProjectionExpression** parameter in filtering attributes.

Every scan returns a result set, even on finding no matches, which results in an empty set. Scans retrieve no more than 1MB, with the option to filter data.

Note: *The parameters and filtering of scans also apply to querying.*

Types of Scan Operations

Filtering: Scan operations offer fine filtering through filter expressions, which modify data after scans, or queries; before returning results. The expressions use comparison operators. Their syntax resembles condition expressions with the exception of key attributes, which filter expressions do not permit. You cannot use a partition or sort key in a filter expression.

Note: *The 1MB limit applies prior to any application of filtering.*

Throughput Specifications: Scans consume throughput, however, consumption focuses on item size rather than returned data. The consumption remains the same whether you request every attribute or only a few, and using or not using a filter expression also does not impact consumption.

Pagination: DynamoDB paginates results causing division of results into specific pages. The 1MB limit applies to returned results, and when you exceed it, another scan becomes necessary to gather the rest of the data. The **LastEvaluatedKey** value allows you to perform this subsequent scan. Simply apply the value to the **ExclusiveStartkey**. When the **LastEvaluatedKey** value becomes null, the operation has completed all pages of data. However, a non-null value does not automatically mean more data remains. Only a null value indicates status.

The Limit Parameter: The limit parameter manages the result size. DynamoDB uses it to establish the number of items to process before returning data, and does not work outside of the scope. If you set a value of x, DynamoDB returns the first x matching items.

The LastEvaluatedKey value also applies in cases of limit parameters yielding partial results. Use it to complete scans.

Result Count: Responses to queries and scans also include information related to **ScannedCount** and Count, which quantify scanned/queried items and quantify items returned. If you do not filter, their values are identical. When you exceed 1MB, the counts represent only the portion processed.

Consistency: Query results and scan results are eventually consistent reads, however, you can set strongly consistent reads as well. Use the **ConsistentRead** parameter to change this setting.

Note: *Consistent read settings impact consumption by using double the capacity units when set to strongly consistent.*

Performance: Queries offer better performance than scans due to scans crawling the full table or secondary index, resulting in a sluggish response and heavy throughput consumption. Scans work best for small tables and searches with less filters, however, you can design lean scans by obeying a few best practices such as avoiding sudden, accelerated read activity and exploiting parallel scans.

A query finds a certain range of keys satisfying a given condition, with performance dictated by the amount of data it retrieves rather than the volume of keys. The parameters of the operation and the number of matches specifically impact performance.

Parallel Scan

Scan operations perform processing sequentially by default. Then they return data in 1MB portions, which prompts the application to fetch the next portion. This results in long scans for large tables and indices.

This characteristic also means scans may not always fully exploit the available throughput. DynamoDB distributes table data across multiple partitions; and scan throughput remains limited to a single partition due to its single-partition operation.

A solution for this problem comes from logically dividing tables or indices into segments. Then "workers" parallel (concurrently) scan segments. It uses the parameters of Segment and **TotalSegments** to specify segments scanned by certain workers and specify the total quantity of segments processed.

Worker Number

You must experiment with worker values (Segment parameter) to achieve the best application performance.

Note: Parallel scans with large sets of workers impacts throughput by possibly consuming all throughput. Manage this issue with the Limit parameter, which you can use to stop a single worker from consuming all throughput.

The following is a deep scan example.

Note: The following program may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This example also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
package com.amazonaws.codesamples.document;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;

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 ScanOpSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(new
ProfileCredentialsProvider()));
    static String tableName = "ProductList";

    public static void main(String[] args) throws Exception {
        findProductsUnderOneHun(); //finds products under 100 dollars
    }

    private static void findProductsUnderOneHun() {

        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, Nomenclature, ProductCategory, Price", //ProjectionExpression
            null, //No ExpressionAttributeNames
            expressionAttributeValues);

        System.out.println("Scanned " + tableName + " to find items under $100.");
        Iterator<Item> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}
```

19. DynamoDB – Indexes

DynamoDB uses indexes for primary key attributes to improve accesses. They accelerate application accesses and data retrieval, and support better performance by reducing application lag.

Secondary Index

A secondary index holds an attribute subset and an alternate key. You use it through either a query or scan operation, which targets the index.

Its contents include attributes you project or copy. In creation, you define an alternate key for the index, and any attributes you wish to project in the index. DynamoDB then performs a copy of the attributes into the index, including primary key attributes sourced from the table. After performing these tasks, you simply use a query/scan as if performing on a table.

DynamoDB automatically maintains all secondary indices. On item operations, such as adding or deleting, it updates any indexes on the target table.

DynamoDB offers two types of secondary indexes:

- **Global Secondary Index –** This index includes a partition key and sort key, which may differ from the source table. It uses the label "global" due to the capability of queries/scans on the index to span all table data, and over all partitions.

- **Local Secondary Index –** This index shares a partition key with the table, but uses a different sort key. Its "local" nature results from all of its partitions scoping to a table partition with identical partition key value.

The best type of index to use depends on application needs. Consider the differences between the two presented in the following table:

Quality	Global Secondary Index	Local Secondary Index
Key Schema	It uses a simple or composite primary key.	It always uses a composite primary key.
Key Attributes	The index partition key and sort key can consist of string, number, or binary table attributes.	The partition key of the index is an attribute shared with the table partition key. The sort key can be string, number, or binary table attributes.
Size Limits Per Partition Key Value	They carry no size limitations.	It imposes a 10GB maximum limit on total size of indexed items associated with a partition key value.

Online Index Operations	You can spawn them at table creation, add them to existing tables, or delete existing ones.	You must create them at table creation, but cannot delete them or add them to existing tables.
Queries	It allows queries covering the entire table, and every partition.	They address single partitions through the partition key value provided in the query.
Consistency	Queries of these indices only offer the eventually consistent option.	Queries of these offer the options of eventually consistent or strongly consistent.
Throughput Cost	It includes throughput settings for reads and writes. Queries/scans consume capacity from the index, not the table, which also applies to table write updates.	Queries/scans consume table read capacity. Table writes update local indexes, and consume table capacity units.
Projection	Queries/scans can only request attributes projected into the index, with no retrievals of table attributes.	Queries/scans can request those attributes not projected; furthermore, automatic fetches of them occur.

When creating multiple tables with secondary indexes, do it sequentially; meaning make a table and wait for it to reach ACTIVE state before creating another and again waiting. DynamoDB does not permit concurrent creation.

Each secondary index requires certain specifications:

- **Type –** Specify local or global.
- **Name –** It uses naming rules identical to tables.
- **Key Schema –** Only top level string, number, or binary type are permitted, with index type determining other requirements.
- **Attributes for Projection –** DynamoDB automatically projects them, and allows any data type.
- **Throughput –** Specify read/write capacity for global secondary indexes.

The limit for indexes remains 5 global and 5 local per table.

You can access the detailed information about indexes with **DescribeTable**. It returns the name, size, and item count.

Note: These values updates every 6 hours.

In queries or scans used to access index data, provide the table and index names, desired attributes for the result, and any conditional statements. DynamoDB offers the option to return results in either ascending or descending order.

Note: *The deletion of a table also deletes all indexes.*

20. DynamoDB – Global Secondary Indexes

Applications requiring various query types with different attributes can use a single or multiple global secondary indexes in performing these detailed queries.

For example: A system keeping a track of users, their login status, and their time logged in. The growth of the previous example slows queries on its data.

Global secondary indexes accelerate queries by organizing a selection of attributes from a table. They employ primary keys in sorting data, and require no key table attributes, or key schema identical to the table.

All the global secondary indexes must include a partition key, with the option of a sort key. The index key schema can differ from the table, and index key attributes can use any top-level string, number, or binary table attributes.

In a projection, you can use other table attributes, however, queries do not retrieve from parent tables.

Attribute Projections

Projections consist of an attribute set copied from table to secondary index. A Projection always occurs with the table partition key and sort key. In queries, projections allow DynamoDB access to any attribute of the projection; they essentially exist as their own table.

In a secondary index creation, you must specify attributes for projection. DynamoDB offers three ways to perform this task:

- **KEYS_ONLY –** All index items consist of table partition and sort key values, and index key values. This creates the smallest index.
- **INCLUDE –** It includes KEYS_ONLY attributes and specified non-key attributes.
- **ALL –** It includes all source table attributes, creating the largest possible index.

Note the tradeoffs in projecting attributes into a global secondary index, which relate to throughput and storage cost.

Consider the following points:

- If you only need access to a few attributes, with low latency, project only those you need. This reduces storage and write costs.
- If an application frequently accesses certain non-key attributes, project them because the storage costs pale in comparison to scan consumption.
- You can project large sets of attributes frequently accessed, however, this carries a high storage cost.
- Use KEYS_ONLY for infrequent table queries and frequent writes/updates. This controls size, but still offers good performance on queries.

Global Secondary Index Queries and Scans

You can utilize queries for accessing a single or multiple items in an index. You must specify index and table name, desired attributes, and conditions; with the option to return results in ascending or descending order.

You can also utilize scans to get all index data. It requires table and index name. You utilize a filter expression to retrieve specific data.

Table and Index Data Synchronization

DynamoDB automatically performs synchronization on indexes with their parent table. Each modifying operation on items causes asynchronous updates, however, applications do not write to indexes directly.

You need to understand the impact of DynamoDB maintenance on indices. On creation of an index, you specify key attributes and data types, which means on a write, those data types must match key schema data types.

On item creation or deletion, indexes update in an eventually consistent manner, however, updates to data propagate in a fraction of a second (unless system failure of some type occurs). You must account for this delay in applications.

Throughput Considerations in Global Secondary Indexes: Multiple global secondary indexes impact throughput. Index creation requires capacity unit specifications, which exist separate from the table, resulting in operations consuming index capacity units rather than table units.

This can result in throttling if a query or write exceeds provisioned throughput. View throughput settings by using **DescribeTable**.

Read Capacity: Global secondary indexes deliver eventual consistency. In queries, DynamoDB performs provision calculations identical to that used for tables, with a lone difference of using index entry size rather than item size. The limit of a query returns remains 1MB, which includes attribute name size and values across every returned item.

Write Capacity

When write operations occur, the affected index consumes write units. Write throughput costs are the sum of write capacity units consumed in table writes and units consumed in index updates. A successful write operation requires sufficient capacity, or it results in throttling.

Write costs also remain dependent on certain factors, some of which are as follows:

- New items defining indexed attributes or item updates defining undefined indexed attributes use a single write operation to add the item to the index.
- Updates changing indexed key attribute value use two writes to delete an item and write a new one.
- A table write triggering deletion of an indexed attribute uses a single write to erase the old item projection in the index.
- Items absent in the index prior to and after an update operation use no writes.

- Updates changing only projected attribute value in the index key schema, and not indexed key attribute value, use one write to update values of projected attributes into the index.

All these factors assume an item size of less than or equal to 1KB.

Global Secondary Index Storage

On an item write, DynamoDB automatically copies the right set of attributes to any indices where the attributes must exist. This impacts your account by charging it for table item storage and attribute storage. The space used results from the sum of these quantities:

- Byte size of table primary key
- Byte size of index key attribute
- Byte size of projected attributes
- 100 byte-overhead per index item

You can estimate storage needs through estimating average item size and multiplying by the quantity of the table items with the global secondary index key attributes.

DynamoDB does not write item data for a table item with an undefined attribute defined as an index partition or sort key.

Global Secondary Index Crud

Create a table with global secondary indexes by using the **CreateTable** operation paired with the **GlobalSecondaryIndexes** parameter. You must specify an attribute to serve as the index partition key, or use another for the index sort key. All index key attributes must be string, number, or binary scalars. You must also provide throughput settings, consisting of **ReadCapacityUnits** and **WriteCapacityUnits**.

Use **UpdateTable** to add global secondary indexes to existing tables using the GlobalSecondaryIndexes parameter once again.

In this operation, you must provide the following inputs:

- Index name
- Key schema
- Projected attributes
- Throughput settings

By adding a global secondary index, it may take a substantial time with large tables due to item volume, projected attributes volume, write capacity, and write activity. Use **CloudWatch** metrics to monitor the process.

Use **DescribeTable** to fetch status information for a global secondary index. It returns one of four **IndexStatus** for GlobalSecondaryIndexes:

- **CREATING –** It indicates the build stage of the index, and its unavailability.
- **ACTIVE –** It indicates the readiness of the index for use.

- **UPDATING –** It indicates the update status of throughput settings.
- **DELETING –** It indicates the delete status of the index, and its permanent unavailability for use.

Update global secondary index provisioned throughput settings during the loading/backfilling stage (DynamoDB writing attributes to an index and tracking added/deleted/updated items). Use **UpdateTable** to perform this operation.

You should remember that you cannot add/delete other indices during the backfilling stage.

Use UpdateTable to delete global secondary indexes. It permits deletion of only one index per operation, however, you can run multiple operations concurrently, up to five. The deletion process does not affect the read/write activities of the parent table, but you cannot add/delete other indices until the operation completes.

Using Java to Work with Global Secondary Indexes

Create a table with an index through CreateTable. Simply create a DynamoDB class instance, a **CreateTableRequest** class instance for request information, and pass the request object to the CreateTable method.

The following program is a short example:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
                    new ProfileCredentialsProvider()));


// Attributes
ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();


attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("City")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Date")
    .withAttributeType("S"));
attributeDefinitions.add(new AttributeDefinition()
    .withAttributeName("Wind")
    .withAttributeType("N"));
// Key schema of the table
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new KeySchemaElement()
```

```
    .withAttributeName("City")
    .withKeyType(KeyType.HASH));  //Partition key
tableKeySchema.add(new KeySchemaElement()
    .withAttributeName("Date")
    .withKeyType(KeyType.RANGE));  //Sort key

// Wind index
GlobalSecondaryIndex windIndex = new GlobalSecondaryIndex()
    .withIndexName("WindIndex")
    .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("Wind")
    .withKeyType(KeyType.RANGE));  //Sort key

windIndex.setKeySchema(indexKeySchema);

CreateTableRequest createTableRequest = new CreateTableRequest()
    .withTableName("ClimateInfo")
    .withProvisionedThroughput(new ProvisionedThroughput()
        .withReadCapacityUnits((long) 5)
        .withWriteCapacityUnits((long) 1))
    .withAttributeDefinitions(attributeDefinitions)
    .withKeySchema(tableKeySchema)
    .withGlobalSecondaryIndexes(windIndex);
Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

Retrieve the index information with **DescribeTable**. First, create a DynamoDB class instance. Then create a Table class instance to target an index. Finally, pass the table to the describe method.

Here is a short example:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

Table table = dynamoDB.getTable("ClimateInfo");
TableDescription tableDesc = table.describe();

Iterator<GlobalSecondaryIndexDescription> gsiIter =
tableDesc.getGlobalSecondaryIndexes().iterator();
while (gsiIter.hasNext()) {
    GlobalSecondaryIndexDescription gsiDesc = gsiIter.next();
    System.out.println("Index data "
         + gsiDesc.getIndexName() + ":");

    Iterator<KeySchemaElement> kse7Iter = 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("\tProjection type: "
        + projection.getProjectionType());
    if (projection.getProjectionType().toString().equals("INCLUDE")) {
        System.out.println("\t\tNon-key projected attributes: "
            + projection.getNonKeyAttributes());
    }
}
```

Use Query to perform an index query as with a table query. Simply create a DynamoDB class instance, a Table class instance for the target index, an Index class instance for the specific index, and pass the index and query object to the query method.

Take a look at the following code to understand better:

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
    new ProfileCredentialsProvider()));

Table table = dynamoDB.getTable("ClimateInfo");
Index index = table.getIndex("WindIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("#d = :v_date and Wind = :v_wind")
    .withNameMap(new NameMap()
        .with("#d", "Date"))
    .withValueMap(new ValueMap()
        .withString(":v_date","2016-05-15")
        .withNumber(":v_wind",0));

ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next().toJSONPretty());
}
```

The following program is a bigger example for better understanding:

Note: The following program may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

This example also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

```
import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 GlobalSecondaryIndexSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    public static String tableName = "Bugs";


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

        createTable();

        queryIndex("CreationDateIndex");
        queryIndex("NameIndex");
        queryIndex("DueDateIndex");
   }

    public static void createTable() {

        // Attributes
        ArrayList<AttributeDefinition> attributeDefinitions = new
ArrayList<AttributeDefinition>();

        attributeDefinitions.add(new AttributeDefinition()
```

```
            .withAttributeName("BugID")
            .withAttributeType("S"));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("Name")
            .withAttributeType("S"));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("CreationDate")
            .withAttributeType("S"));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("DueDate")
            .withAttributeType("S"));

        // Table Key schema
        ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
        tableKeySchema.add(new KeySchemaElement()
            .withAttributeName("BugID")
            .withKeyType(KeyType.HASH)); //Partition key
        tableKeySchema.add(new KeySchemaElement()
            .withAttributeName("Name")
            .withKeyType(KeyType.RANGE)); //Sort key

        // Indexes' initial provisioned throughput
        ProvisionedThroughput ptIndex = new ProvisionedThroughput()
            .withReadCapacityUnits(1L)
            .withWriteCapacityUnits(1L);

        // CreationDateIndex
        GlobalSecondaryIndex creationDateIndex = new GlobalSecondaryIndex()
            .withIndexName("CreationDateIndex")
            .withProvisionedThroughput(ptIndex)
            .withKeySchema( new KeySchemaElement()
                .withAttributeName("CreationDate")
                .withKeyType(
                    KeyType.HASH), //Partition key
                    new KeySchemaElement()
```

```
                    .withAttributeName("BugID")
                    .withKeyType(KeyType.RANGE)) //Sort key
                    .withProjection(new Projection()
                        .withProjectionType("INCLUDE")
                        .withNonKeyAttributes("Description", "Status"));

        // NameIndex
        GlobalSecondaryIndex nameIndex = new GlobalSecondaryIndex()
            .withIndexName("NameIndex")
            .withProvisionedThroughput(ptIndex)
            .withKeySchema(new KeySchemaElement()
                .withAttributeName("Name")
                .withKeyType(KeyType.HASH), //Partition key
                    new KeySchemaElement()
                        .withAttributeName("BugID")
                        .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(creationDateIndex, nameIndex, dueDateIndex);

        System.out.println("Creating " + tableName + "...");
        dynamoDB.createTable(createTableRequest);

        // Pause for active table state
        System.out.println("Waiting for ACTIVE state of " + tableName);
        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 == "CreationDateIndex") {
            System.out.println("Issues filed on 2016-05-22");
            querySpec.withKeyConditionExpression("CreationDate  =  :v_date  and
begins_with(BugID, :v_bug)")

                .withValueMap(new ValueMap()
                     .withString(":v_date","2016-05-22")
```

```
                    .withString(":v_bug","A-"));
            items = index.query(querySpec);
        } else if (indexName == "NameIndex") {
            System.out.println("Compile error");
            querySpec.withKeyConditionExpression("Name = :v_name and
begins_with(BugID, :v_bug)")
                .withValueMap(new ValueMap()
                    .withString(":v_name","Compile error")
                    .withString(":v_bug","A-"));
            items = index.query(querySpec);
        } else if (indexName == "DueDateIndex") {
            System.out.println("Items due on 2016-10-15");
            querySpec.withKeyConditionExpression("DueDate = :v_date")
                .withValueMap(new ValueMap()
                    .withString(":v_date","2016-10-15"));
            items = index.query(querySpec);
        } else {
            System.out.println("\nInvalid index name");
            return;
        }

        Iterator<Item> iterator = items.iterator();
        System.out.println("Query: getting result...");
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toJSONPretty());
        }
    }
}
```

21. DynamoDB – Local Secondary Indexes

Some applications only perform queries with the primary key, but some situations benefit from an alternate sort key. Allow your application a choice by creating a single or multiple local secondary indexes.

Complex data access requirements, such as combing millions of items, make it necessary to perform more efficient queries/scans. Local secondary indices provide an alternate sort key for a partition key value. They also hold copies of all or some table attributes. They organize data by table partition key, but use a different sort key.

Using a local secondary index removes the need for a whole table scan, and allows a simple and quick query using a sort key.

All the local secondary indexes must satisfy certain conditions:

- Identical partition key and source table partition key.
- A sort key of only one scalar attribute.
- Projection of the source table sort key acting as a non-key attribute.

All the local secondary indexes automatically hold partition and sort keys from parent tables. In queries, this means efficient gathering of projected attributes, and also retrieval of attributes not projected.

The storage limit for a local secondary index remains 10GB per partition key value, which includes all table items, and index items sharing a partition key value.

Projecting an Attribute

Some operations require excess reads/fetching due to complexity. These operations can consume substantial throughput. Projection allows you to avoid costly fetching and perform rich queries by isolating these attributes. Remember projections consist of attributes copied into a secondary index.

When making a secondary index, you specify the attributes projected. Recall the three options provided by DynamoDB: **KEYS_ONLY, INCLUDE, and ALL**.

When opting for certain attributes in projection, consider the associated cost tradeoffs:

- If you project only a small set of necessary attributes, you dramatically reduce the storage costs.
- If you project frequently accessed non-key attributes, you offset scan costs with storage costs.
- If you project most or all non-key attributes, this maximizes flexibility and reduces throughput (no retrievals); however, storage costs rise.
- If you project KEYS_ONLY for frequent writes/updates and infrequent queries, it minimizes size, but maintains query preparation.

Local Secondary Index Creation

Use the **LocalSecondaryIndex** parameter of CreateTable to make a single or multiple local secondary indexes. You must specify one non-key attribute for the sort key. On table creation, you create local secondary indices. On deletion, you delete these indexes.

Tables with a local secondary index must obey a limit of 10GB in size per partition key value, but can store any amount of items.

Local Secondary Index Queries and Scans

A query operation on local secondary indexes returns all items with a matching partition key value when multiple items in the index share sort key values. Matching items do not return in a certain order. Queries for local secondary indexes use either eventual or strong consistency, with strongly consistent reads delivering the latest values.

A scan operation returns all local secondary index data. Scans require you to provide a table and index name, and allow the use of a filter expression to discard data.

Item Writing

On creation of a local secondary index, you specify a sort key attribute and its data type. When you write an item, its type must match the data type of the key schema if the item defines an attribute of an index key.

DynamoDB imposes no one-to-one relationship requirements on table items and local secondary index items. The tables with multiple local secondary indexes carry higher write costs than those with less.

Throughput Considerations in Local Secondary Indexes

Read capacity consumption of a query depends on the nature of data access. Queries use either eventual or strong consistency, with strongly consistent reads using one unit compared to half a unit in eventually consistent reads.

Result limitations include a 1MB size maximum. Result sizes come from the sum of matching index item size rounded up to the nearest 4KB, and matching table item size also rounded up to the nearest 4KB.

The write capacity consumption remains within provisioned units. Calculate the total provisioned cost by finding the sum of consumed units in table writing and consumed units in updating indices.

You can also consider the key factors influencing cost, some of which can be:

- When you write an item defining an indexed attribute or update an item to define an undefined indexed attribute, a single write operation occurs.
- When a table update changes an indexed key attribute value, two writes occur to delete and then – add an item.
- When a write causes the deletion of an indexed attribute, one write occurs to remove the old item projection.

- When an item does not exist within the index prior to or after an update, no writes occur.

Local Secondary Index Storage

On a table item write, DynamoDB automatically copies the right attribute set to the required local secondary indexes. This charges your account. The space used results from the sum of table primary key byte size, index key attribute byte size, any present projected attribute byte size, and 100 bytes in overhead for each index item.

The estimate storage is got by estimating average index item size and multiplying by table item quantity.

Using Java to Work with Local Secondary Indexes

Create a local secondary index by first creating a DynamoDB class instance. Then, create a CreateTableRequest class instance with necessary request information. Finally, use the createTable method.

Example

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

String tableName = "Tools";

CreateTableRequest createTableRequest = new
CreateTableRequest().withTableName(tableName);

//Provisioned Throughput
createTableRequest.setProvisionedThroughput(new
ProvisionedThroughput().withReadCapacityUnits((long)5).withWriteCapacityUnits((
long)5));

//Attributes
ArrayList<AttributeDefinition> attributeDefinitions= new
ArrayList<AttributeDefinition>();
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Make").withAttributeType("S"));
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Model").withAttributeType("S"));
attributeDefinitions.add(new
AttributeDefinition().withAttributeName("Line").withAttributeType("S"));

createTableRequest.setAttributeDefinitions(attributeDefinitions);
```

```
//Key Schema
ArrayList<KeySchemaElement> tableKeySchema = new ArrayList<KeySchemaElement>();
tableKeySchema.add(new
KeySchemaElement().withAttributeName("Make").withKeyType(KeyType.HASH));
//Partition key
tableKeySchema.add(new
KeySchemaElement().withAttributeName("Model").withKeyType(KeyType.RANGE));
//Sort key

createTableRequest.setKeySchema(tableKeySchema);

ArrayList<KeySchemaElement> indexKeySchema = new ArrayList<KeySchemaElement>();
indexKeySchema.add(new
KeySchemaElement().withAttributeName("Make").withKeyType(KeyType.HASH));
//Partition key
indexKeySchema.add(new
KeySchemaElement().withAttributeName("Line").withKeyType(KeyType.RANGE));
//Sort key

Projection projection = new
Projection().withProjectionType(ProjectionType.INCLUDE);
ArrayList<String> nonKeyAttributes = new ArrayList<String>();
nonKeyAttributes.add("Type");
nonKeyAttributes.add("Year");
projection.setNonKeyAttributes(nonKeyAttributes);

LocalSecondaryIndex localSecondaryIndex = new LocalSecondaryIndex()
    .withIndexName("ModelIndex").withKeySchema(indexKeySchema).withProjection(p
rojection);

ArrayList<LocalSecondaryIndex> localSecondaryIndexes = new
ArrayList<LocalSecondaryIndex>();
localSecondaryIndexes.add(localSecondaryIndex);
createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

Table table = dynamoDB.createTable(createTableRequest);
System.out.println(table.getDescription());
```

Retrieve information about a local secondary index with the describe method. Simply create a DynamoDB class instance, create a Table class instance, and pass the table to the describe method.

Example

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

String tableName = "Tools";

Table table = dynamoDB.getTable(tableName);

TableDescription tableDescription = table.describe();

List<LocalSecondaryIndexDescription> localSecondaryIndexes
    = tableDescription.getLocalSecondaryIndexes();

Iterator<LocalSecondaryIndexDescription>               lsiIter               =
localSecondaryIndexes.iterator();
while (lsiIter.hasNext()) {

     LocalSecondaryIndexDescription lsiDescription = lsiIter.next();
     System.out.println("Index info " + 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("\tProjection type: " + projection.getProjectionType());
     if (projection.getProjectionType().toString().equals("INCLUDE")) {
           System.out.println("\t\tNon-key    projected    attributes:    "    +
projection.getNonKeyAttributes());
     }
}
```

Perform a query by using the same steps as a table query. Merely create a DynamoDB class instance, a Table class instance, an Index class instance, a query object, and utilize the query method.

Example

```
DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

String tableName = "Tools";

Table table = dynamoDB.getTable(tableName);
Index index = table.getIndex("LineIndex");

QuerySpec spec = new QuerySpec()
    .withKeyConditionExpression("Make = :v_make and Line = :v_line")
    .withValueMap(new ValueMap()
        .withString(":v_make", "Depault")
        .withString(":v_line", "SuperSawz"));

ItemCollection<QueryOutcome> items = index.query(spec);

Iterator<Item> itemsIter = items.iterator();

while (itemsIter.hasNext()) {
     Item item = itemsIter.next();
     System.out.println(item.toJSONPretty());
}
```

You can also review the following example.

Note: The following example may assume a previously created data source. Before attempting to execute, acquire supporting libraries and create necessary data sources (tables with required characteristics, or other referenced sources).

The following example also uses Eclipse IDE, an AWS credentials file, and the AWS Toolkit within an Eclipse AWS Java Project.

Example

```
import java.util.ArrayList;
import java.util.Iterator;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
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 LocalSecondaryIndexSample {

    static DynamoDB dynamoDB = new DynamoDB(new AmazonDynamoDBClient(
            new ProfileCredentialsProvider()));

    public static String tableName = "ProductOrders";

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

        createTable();
```

```
        query(null);
        query("IsOpenIndex");
        query("OrderCreationDateIndex");
    }

    public static void createTable() {

        CreateTableRequest createTableRequest = new CreateTableRequest()
            .withTableName(tableName)
            .withProvisionedThroughput(new ProvisionedThroughput()
                .withReadCapacityUnits((long) 1)
                .withWriteCapacityUnits((long) 1));

        // Table partition and sort keys attributes
        ArrayList<AttributeDefinition>      attributeDefinitions      =      new
ArrayList<AttributeDefinition>();
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("CustomerID")
            .withAttributeType("S"));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("OrderID")
            .withAttributeType("N"));

        // Index primary key attributes
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("OrderDate")
            .withAttributeType("N"));
        attributeDefinitions.add(new AttributeDefinition()
            .withAttributeName("OpenStatus")
            .withAttributeType("N"));

        createTableRequest.setAttributeDefinitions(attributeDefinitions);

        // Table key schema
        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>();

        // OrderDateIndex
        LocalSecondaryIndex orderDateIndex = new LocalSecondaryIndex()
           .withIndexName("OrderDateIndex");

        // OrderDateIndex key schema
        ArrayList<KeySchemaElement>         indexKeySchema         =         new
ArrayList<KeySchemaElement>();
        indexKeySchema.add(new KeySchemaElement()
            .withAttributeName("CustomerID")
            .withKeyType(KeyType.HASH)); //Partition key
        indexKeySchema.add(new KeySchemaElement()
            .withAttributeName("OrderDate")
            .withKeyType(KeyType.RANGE)); //Sort key

        orderDateIndex.setKeySchema(indexKeySchema);

        // OrderCreationDateIndex projection w/attributes list
        Projection projection = new Projection()
            .withProjectionType(ProjectionType.INCLUDE);
        ArrayList<String> nonKeyAttributes = new ArrayList<String>();
        nonKeyAttributes.add("ProdCat");
        nonKeyAttributes.add("ProdNomenclature");
        projection.setNonKeyAttributes(nonKeyAttributes);
```

```
        orderCreationDateIndex.setProjection(projection);

        localSecondaryIndexes.add(orderDateIndex);

        // IsOpenIndex
        LocalSecondaryIndex isOpenIndex = new LocalSecondaryIndex()
            .withIndexName("IsOpenIndex");

        // OpenStatusIndex key schema
        indexKeySchema = new ArrayList<KeySchemaElement>();
        indexKeySchema.add(new KeySchemaElement()
            .withAttributeName("CustomerID")
            .withKeyType(KeyType.HASH)); //Partition key
        indexKeySchema.add(new KeySchemaElement()
            .withAttributeName("OpenStatus")
            .withKeyType(KeyType.RANGE)); //Sort key

        // OpenStatusIndex projection
        projection = new Projection()
            .withProjectionType(ProjectionType.ALL);

        OpenStatusIndex.setKeySchema(indexKeySchema);
        OpenStatusIndex.setProjection(projection);

        localSecondaryIndexes.add(OpenStatusIndex);

        // Put definitions in CreateTable request
        createTableRequest.setLocalSecondaryIndexes(localSecondaryIndexes);

        System.out.println("Spawning table " + tableName + "...");
        System.out.println(dynamoDB.createTable(createTableRequest));

        // Pause for ACTIVE status
        System.out.println("Waiting for ACTIVE table:" + tableName);
        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("Executing query on" + tableName);

        QuerySpec querySpec = new QuerySpec()
            .withConsistentRead(true)
            .withScanIndexForward(true)
            .withReturnConsumedCapacity(ReturnConsumedCapacity.TOTAL);

        if (indexName == "OpenStatusIndex") {

            System.out.println("\nEmploying index: '" + indexName
                + "' open orders for this customer.");
            System.out.println(
                "Returns only user-specified attribute list\n");
            Index index = table.getIndex(indexName);

            querySpec.withKeyConditionExpression("CustomerID  =  :v_custmid  and
OpenStatus = :v_openstat")
                .withValueMap(new ValueMap()
                    .withString(":v_custmid", "jane@sample.com")
                    .withNumber(":v_openstat", 1));

            querySpec.withProjectionExpression(

              "OrderDate, ProdCat, ProdNomenclature, OrderStatus");
```

```
            ItemCollection<QueryOutcome> items = index.query(querySpec);
            Iterator<Item> iterator = items.iterator();

            System.out.println("Printing query results...");

            while (iterator.hasNext()) {
                System.out.println(iterator.next().toJSONPretty());
            }

        } else if (indexName == "OrderDateIndex") {
            System.out.println("\nUsing index: '" + indexName
                + "': this customer's orders placed after 05/22/2016.");
            System.out.println("Projected attributes are returned\n");
            Index index = table.getIndex(indexName);

            querySpec.withKeyConditionExpression("CustomerID = :v_custmid and
OrderDate >= :v_ordrdate")
            .withValueMap(new ValueMap()
                .withString(":v_custmid", "jane@sample.com")
                .withNumber(":v_ordrdate", 20160522));

            querySpec.withSelect(Select.ALL_PROJECTED_ATTRIBUTES);

            ItemCollection<QueryOutcome> items = index.query(querySpec);
            Iterator<Item> iterator = items.iterator();

            System.out.println("Printing query results...");

            while (iterator.hasNext()) {
                System.out.println(iterator.next().toJSONPretty());
            }

        } else {
            System.out.println("\nNo index: All Jane's orders by OrderID:\n");

            querySpec.withKeyConditionExpression("CustomerID = :v_custmid")
            .withValueMap(new ValueMap()
```

```
                .withString(":v_custmid", "jane@example.com"));

            ItemCollection<QueryOutcome> items = table.query(querySpec);
            Iterator<Item> iterator = items.iterator();

            System.out.println("Printing query results...");

            while (iterator.hasNext()) {
                System.out.println(iterator.next().toJSONPretty());
            }
        }
    }
}
```

22. DynamoDB – Aggregation

DynamoDB does not provide aggregation functions. You must make creative use of queries, scans, indices, and assorted tools to perform these tasks. In all this, the throughput expense of queries/scans in these operations can be heavy.

You also have the option to use libraries and other tools for your preferred DynamoDB coding language. Ensure their compatibility with DynamoDB prior to using it.

Calculate Maximum or Minimum

Utilize the ascending/descending storage order of results, the Limit parameter, and any parameters which set order to find the highest and lowest values.

For example:

```
Map<String, AttributeValue> eaval = new HashMap<>();
eaval.put(":v1", new AttributeValue().withS("hashval"));
queryExpression = new DynamoDBQueryExpression<Table>()
        .withIndexName("yourindexname")
        .withKeyConditionExpression("HK = :v1")
        .withExpressionAttributeValues(values)
        .withScanIndexForward(false);  //descending order
queryExpression.setLimit(1);
QueryResultPage<Lookup> res = dynamoDBMapper.queryPage(Table.class,
queryExpression);
```

Calculate Count

Use **DescribeTable** to get a count of the table items, however, note that it provides stale data. Also, utilize the Java **getScannedCount method**.

Utilize **LastEvaluatedKey** to ensure it delivers all results.

For example:

```
ScanRequest scanRequest = new ScanRequest().withTableName(yourtblName);
 ScanResult yourresult = client.scan(scanRequest);
 System.out.println("#items:" + yourresult.getScannedCount());
```

Calculating Average and Sum

Utilize indices and a query/scan to retrieve and filter values before processing. Then simply operate on those values through an object.

23. DynamoDB – Access Control

DynamoDB uses credentials you provide to authenticate requests. These credentials are required and must include permissions for AWS resource access. These permissions span virtually every aspect of DynamoDB down to the minor features of an operation or functionality.

Types of Permissions

In this section, we will discuss regarding the various permissions and resource access in DynamoDB.

Authenticating Users

On signup, you provided a password and email, which serve as root credentials. DynamoDB associates this data with your AWS account, and uses it to give complete access to all resources.

AWS recommends you use your root credentials only for the creation of an administration account. This allows you to create IAM accounts/users with less privileges. IAM users are other accounts spawned with the IAM service. Their access permissions/privileges include access to secure pages and certain custom permissions like table modification.

The access keys provide another option for additional accounts and access. Use them to grant access, and also to avoid manual granting of access in certain situations. Federated users provide yet another option by allowing access through an identity provider.

Administration

AWS resources remain under ownership of an account. Permissions policies govern the permissions granted to spawn or access resources. Administrators associate permissions policies with IAM identities, meaning roles, groups, users, and services. They also attach permissions to resources.

Permissions specify users, resources, and actions. Note administrators are merely accounts with administrator privileges.

Operation and Resources

Tables remain the main resources in DynamoDB. Subresources serve as additional resources, e.g., streams and indices. These resources use unique names, some of which are mentioned in the following table:

Type	ARN (Amazon Resource Name)
Stream	arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label
Index	arn:aws:dynamodb:region:account-id:table/table-name/index/index-name
Table	arn:aws:dynamodb:region:account-id:table/table-name

Ownership

A resource owner is defined as an AWS account which spawned the resource, or principal entity account responsible for request authentication in resource creation. Consider how this functions within the DynamoDB environment:

- In using root credentials to create a table, your account remains resource owner.
- In creating an IAM user and granting the user permission to create a table, your account remains the resource owner.
- In creating an IAM user and granting the user, and anyone capable of assuming the role, permission to create a table, your account remains the resource owner.

Manage Resource Access

Management of access mainly requires attention to a permissions policy describing users and resource access. You associate policies with IAM identities or resources. However, DynamoDB only supports IAM/identity policies.

Identity-based (IAM) policies allow you to grant privileges in the following ways:

- Attach permissions to users or groups.
- Attach permissions to roles for cross-account permissions.

Other AWS allow resource-based policies. These policies permit access to things like an S3 bucket.

Policy Elements

Policies define actions, effects, resources, and principals; and grant permission to perform these operations.

Note: The API operations may require permissions for multiple actions.

Take a closer look at the following policy elements:

- **Resource –** An ARN identifies this.
- **Action –** Keywords identify these resource operations, and whether to allow or deny.

- **Effect –** It specifies the effect for a user request for an action, meaning allow or deny with denial as the default.
- **Principal –** This identifies the user attached to the policy.

Conditions

In granting permissions, you can specify conditions for when policies become active such as on a particular date. Express conditions with condition keys, which include AWS system-wide keys and DynamoDB keys. These keys are discussed in detail later in the tutorial.

Console Permissions

A user requires certain basic permissions to use the console. They also require permissions for the console in other standard services:

- CloudWatch
- Data Pipeline
- Identity and Access Management
- Notification Service
- Lambda

If the IAM policy proves too limited, the user cannot use the console effectively. Also, you do not need to worry about user permissions for those only calling the CLI or API.

Common Use Iam Policies

AWS covers common operations in permissions with standalone IAM managed policies. They provide key permissions allowing you to avoid deep investigations into what you must grant.

Some of them are as follows:

- **AmazonDynamoDBReadOnlyAccess –** It gives read-only access via the console.
- **AmazonDynamoDBFullAccess –** It gives full access via the console.
- **AmazonDynamoDBFullAccesswithDataPipeline –** It gives full access via the console and permits export/import with Data Pipeline.

You can also ofcourse make custom policies.

Granting Privileges: Using The Shell

You can grant permissions with the Javascript shell. The following program shows a typical permissions policy:

```
{
    "Version": "2016-05-22",
    "Statement": [
        {
            "Sid": "DescribeQueryScanToolsTable",
            "Effect": "Deny",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:account-id:table/Tools"
        }
    ]
}
```

You can review the three examples which are as follows:

Block the user from executing any table action.

```
{
    "Version": "2016-05-23",
    "Statement": [
        {
            "Sid": "AllAPIActionsOnTools",
            "Effect": "Deny",
            "Action": "dynamodb:*",
            "Resource": "arn:aws:dynamodb:us-west-2:155556789012:table/Tools"
        }
    ]
}
```

Block access to a table and its indices.

```
{
    "Version": "2016-05-23",
    "Statement": [
        {
            "Sid": "AccessAllIndexesOnTools",
            "Effect": "Deny",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-2:155556789012:table/Tools",
                "arn:aws:dynamodb:us-west-2:155556789012:table/Tools/index/*"
            ]
        }
    ]
}
```

Block a user from making a reserved capacity offering purchase.

```
{
    "Version": "2016-05-23",
    "Statement": [
        {
            "Sid": "BlockReservedCapacityPurchases",
            "Effect": "Deny",
            "Action": "dynamodb:PurchaseReservedCapacityOfferings",
            "Resource": "arn:aws:dynamodb:us-west-2:155556789012:*"
        }
    ]
}
```

Granting Privileges: Using the GUI Console

You can also use the GUI console to create IAM policies. To begin with, choose **Tables** from the navigation pane. In the table list, choose the target table and follow these steps.

Step 1: Select the **Access control** tab.

Step 2: Select the identity provider, actions, and policy attributes. Select **Create policy** after entering all settings.

Step 3: Choose **Attach policy instructions**, and complete each required step to associate the policy with the appropriate IAM role.

24. DynamoDB – Permissions API

DynamoDB API offers a large set of actions, which require permissions. In setting permissions, you must establish the actions permitted, resources permitted, and conditions of each.

You can specify actions within the Action field of the policy. Specify resource value within the Resource field of the policy. But do ensure that you use the correct syntax containing the Dynamodb: prefix with the API operation.

For example: **dynamodb:CreateTable**

You can also employ condition keys to filter permissions.

Permissions and API Actions

Take a good look at the API actions and associated permissions given in the following table:

API Operation	Necessary Permission
BatchGetItem	dynamodb:BatchGetItem
BatchWriteItem	dynamodb:BatchWriteItem
CreateTable	dynamodb:CreateTable
DeleteItem	dynamodb:DeleteItem
DeleteTable	dynamodb:DeleteTable
DescribeLimits	dynamodb:DescribeLimits
DescribeReservedCapacity	dynamodb:DescribeReservedCapacity
DescribeReservedCapacityOfferings	dynamodb:DescribeReservedCapacityOfferings
DescribeStream	dynamodb:DescribeStream
DescribeTable	dynamodb:DescribeTable
GetItem	dynamodb:GetItem
GetRecords	dynamodb:GetRecords

GetShardIterator	dynamodb:GetShardIterator
ListStreams	dynamodb:ListStreams
ListTables	dynamodb:ListTables
PurchaseReservedCapacityOfferings	dynamodb:PurchaseReservedCapacityOfferings
PutItem	dynamodb:PutItem
Query	dynamodb:Query
Scan	dynamodb:Scan
UpdateItem	dynamodb:UpdateItem
UpdateTable	dynamodb:UpdateTable

Resources

In the following table, you can review the resources associated with each permitted API action:

API Operation	Resource
BatchGetItem	arn:aws:dynamodb:region:account-id:table/table-name
BatchWriteItem	arn:aws:dynamodb:region:account-id:table/table-name
CreateTable	arn:aws:dynamodb:region:account-id:table/table-name
DeleteItem	arn:aws:dynamodb:region:account-id:table/table-name
DeleteTable	arn:aws:dynamodb:region:account-id:table/table-name
DescribeLimits	arn:aws:dynamodb:region:account-id:*
DescribeReservedCapacity	arn:aws:dynamodb:region:account-id:*

DescribeReservedCapacityOfferings	arn:aws:dynamodb:region:account-id:*
DescribeStream	arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label
DescribeTable	arn:aws:dynamodb:region:account-id:table/table-name
GetItem	arn:aws:dynamodb:region:account-id:table/table-name
GetRecords	arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label
GetShardIterator	arn:aws:dynamodb:region:account-id:table/table-name/stream/stream-label
ListStreams	arn:aws:dynamodb:region:account-id:table/table-name/stream/*
ListTables	*
PurchaseReservedCapacityOfferings	arn:aws:dynamodb:region:account-id:*
PutItem	arn:aws:dynamodb:region:account-id:table/table-name
Query	arn:aws:dynamodb:region:account-id:table/table-name or arn:aws:dynamodb:region:account-id:table/table-name/index/index-name
Scan	arn:aws:dynamodb:region:account-id:table/table-name or arn:aws:dynamodb:region:account-id:table/table-name/index/index-name
UpdateItem	arn:aws:dynamodb:region:account-id:table/table-name
UpdateTable	arn:aws:dynamodb:region:account-id:table/table-name

25. DynamoDB – Conditions

In granting permissions, DynamoDB allows specifying conditions for them through a detailed IAM policy with condition keys. This supports settings like access to specific items and attributes.

Note: The DynamoDB does not support any tags.

Detailed Control

Several conditions allow specificity down to items and attributes like granting read-only access to specific items based on user account. Implement this level of control with conditioned IAM policies, which manages the security credentials. Then simply apply the policy to the desired users, groups, and roles. Web Identity Federation, a topic discussed later, also provides a way to control user access through Amazon, Facebook, and Google logins.

The condition element of IAM policy implements access control. You simply add it to a policy. An example of its use consists of denying or permitting access to table items and attributes. The condition element can also employ condition keys to limit permissions.

You can review the following two examples of the condition keys:

- **dynamodb:LeadingKeys –** It prevents the item access by users without an ID matching the partition key value.

- **dynamodb:Attributes –** It prevents users from accessing or operating on attributes outside of those listed.

On evaluation, IAM policies result in a true or false value. If any part evaluates to false, the whole policy evaluates to false, which results in denial of access. Be sure to specify all required information in condition keys to ensure users have appropriate access.

Predefined Condition Keys

AWS offers a collection of predefined condition keys, which apply to all services. They support a broad range of uses and fine detail in examining users and access.

Note: There is case sensitivity in condition keys.

You can review a selection of the following service-specific keys:

- **dynamodb:LeadingKey –** It represents a table's first key attribute; the partition key. Use the **ForAllValues** modifier in conditions.

- **dynamodb:Select –** It represents a query/scan request Select parameter. It must be of the value ALL_ATTRIBUTES, ALL_PROJECTED_ATTRIBUTES, SPECIFIC_ATTRIBUTES, or COUNT.

- **dynamodb:Attributes –** It represents an attribute name list within a request, or attributes returned from a request. Its values and their functions resemble API action parameters, e.g., BatchGetItem uses AttributesToGet.

- **dynamodb:ReturnValues –** It represents a requests' ReturnValues parameter, and can use these values: ALL_OLD, UPDATED_OLD, ALL_NEW, UPDATED_NEW, and NONE.

- **dynamodb:ReturnConsumedCapacity –** It represents a request's ReturnConsumedCapacity parameter, and can use these values: TOTAL and NONE.

26. DynamoDB – Web Identity Federation

Web Identity Federation allows you to simplify authentication and authorization for large user groups. You can skip the creation of individual accounts, and require users to login to an identity provider to get temporary credentials or tokens. It uses AWS Security Token Service (STS) to manage credentials. Applications use these tokens to interact with services.

Web Identity Federation also supports other identity providers such as – Amazon, Google, and Facebook.

Function: In use, Web Identity Federation first calls an identity provider for user and app authentication, and the provider returns a token. This results in the app calling AWS STS and passing the token for input. STS authorizes the app and grants it temporary access credentials, which allow the app to use an IAM role and access resources based on policy.

Implementing Web Identity Federation

You must perform the following three steps prior to use:

- Use a supported third party identity provider to register as a developer.
- Register your application with the provider to obtain an app ID.
- Create a single or multiple IAM roles, including policy attachment. You must use a role per provider per app.

Assume one of your IAM roles to use Web Identity Federation. Your app must then perform a three-step process:

- Authentication
- Credential acquisition
- Resource Access

In the first step, your app uses its own interface to call the provider and then manages the token process.

Then step two manages tokens and requires your app to send an **AssumeRoleWithWebIdentity** request to AWS STS. The request holds the first token, the provider app ID, and the ARN of the IAM role. The STS the provides credentials set to expire after a certain period.

In the final step, your app receives a response from STS containing access information for DynamoDB resources. It consists of access credentials, expiration time, role, and role ID.

27. DynamoDB – Data Pipeline

Data Pipeline allows for exporting and importing data to/from a table, file, or S3 bucket. This of course proves useful in backups, testing, and for similar needs or scenarios.

In an export, you use the Data Pipeline console, which makes a new pipeline and launches an Amazon EMR (Elastic MapReduce) cluster to perform the export. An EMR reads data from DynamoDB and writes to the target. We discuss EMR in detail later in this tutorial.

In an import operation, you use the Data Pipeline console, which makes a pipeline and launches EMR to perform the import. It reads data from the source and writes to the destination.

Note: *Export/import operations carry a cost given the services used, specifically, EMR and S3.*

Using Data Pipeline

You must specify action and resource permissions when using Data Pipeline. You can utilize an IAM role or policy to define them. The users who are performing imports/exports should make a note that they would require an active access key ID and secret key.

IAM Roles for Data Pipeline

You need two IAM roles to use Data Pipeline:

- **DataPipelineDefaultRole –** This has all the actions you permit the pipeline to perform for you.
- **DataPipelineDefaultResourceRole –** This has resources you permit the pipeline to provision for you.

If you are new to Data Pipeline, you must spawn each role. All the previous users possess these roles due to the existing roles.

Use the IAM console to create IAM roles for Data Pipeline, and perform the following four steps:

Step 1: Log in to the IAM console located at https://console.aws.amazon.com/iam/

Step 2: Select **Roles** from the dashboard.

Step 3: Select **Create New Role**. Then enter DataPipelineDefaultRole in the **Role Name** field, and select **Next Step**. In the **AWS Service Roles** list in the **Role Type** panel, navigate to **Data Pipeline**, and choose **Select**. Select **Create Role** in the **Review** panel.

Step 4: Select **Create New Role**.

28. DynamoDB – Data Backup

Utilize Data Pipeline's import/export functionality to perform backups. How you execute a backup depends on whether you use the GUI console, or use Data Pipeline directly (API). Either create separate pipelines for each table when using the console, or import/export multiple tables in a single pipeline if using a direct option.

Exporting and Importing Data

You must create an Amazon S3 bucket prior to performing an export. You can export from one or more tables.

Perform the following four step process to execute an export:

Step 1: Log in to the AWS Management Console and open the Data Pipeline console located at https://console.aws.amazon.com/datapipeline/

Step 2: If you have no pipelines in the AWS region used, select **Get started now**. If you have one or more, select **Create new pipeline**.

Step 3: On the creation page, enter a name for your pipeline. Choose **Build using a template** for the Source parameter. Select **Export DynamoDB table to S3** from the list. Enter the source table in the **Source DynamoDB table name** field.

Enter the destination S3 bucket in the **Output S3 Folder** text box using the following format: s3://nameOfBucket/region/nameOfFolder. Enter an S3 destination for the log file in **S3 location for logs** text box.

Step 4: Select **Activate** after entering all settings.

The pipeline may take several minutes to finish its creation process. Use the console to monitor its status. Confirm successful processing with the S3 console by viewing the exported file.

Importing Data

Successful imports can only happen if the following conditions are true: you created a destination table, the destination and source use identical names, and the destination and source use identical key schema.

You can use a populated destination table, however, imports replace data items sharing a key with source items, and also add excess items to the table. The destination can also use a different region.

Though you can export multiple sources, you can only import one per operation. You can perform an import by adhering to the following steps:

Step 1: Log in to the AWS Management Console, and then open the Data Pipeline console.

Step 2: If you are intending to execute a cross region import, then you should select the destination region.

Step 3: Select **Create new pipeline**.

Step 4: Enter the pipeline name in the **Name** field. Choose **Build using a template** for the Source parameter, and in the template list, select **Import DynamoDB backup data from S3**.

Enter the location of the source file in the **Input S3 Folder** text box. Enter the destination table name in the **Target DynamoDB table name** field. Then enter the location for the log file in the **S3 location for logs** text box.

Step 5: Select **Activate** after entering all settings.

The import starts immediately after the pipeline creation. It may take several minutes for the pipeline to complete the creation process.

Errors

When errors occur, the Data Pipeline console displays ERROR as the pipeline status. Clicking the pipeline with an error takes you to its detail page, which reveals every step of the process and the point at which the failure occurred. Log files within also provide some insight.

You can review the common causes of the errors as follows:

- The destination table for an import does not exist, or does not use identical key schema to the source.
- The S3 bucket does not exist, or you do not have read/write permissions for it.
- The pipeline timed out.
- You do not have the necessary export/import permissions.
- Your AWS account reached its resource limit.

29. DynamoDB – Monitoring

Amazon offers CloudWatch for aggregating and analyzing performance through the CloudWatch console, command line, or CloudWatch API. You can also use it to set alarms and perform tasks. It performs specified actions on certain events.

Cloudwatch Console

Utilize CloudWatch by accessing the Management Console, and then opening the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

You can then perform the following steps:

- Select **Metrics** from the navigation pane.
- Under DynamoDB metrics within the **CloudWatch Metrics by Category** pane, choose **Table Metrics**.
- Use the upper pane to scroll below and examine the entire list of table metrics. The **Viewing** list provides metrics options.

In the results interface, you can select/deselect each metric by selecting the checkbox beside the resource name and metric. Then you would be able to view graphs for each item.

API Integration

You can access CloudWatch with queries. Use metric values to perform CloudWatch actions. Note DynamoDB does not send metrics with a value of zero. It simply skips metrics for time periods where those metrics remain at that value.

The following are some of the most commonly used metrics:

- **ConditionalCheckFailedRequests –** It tracks the quantity of failed attempts at conditional writes such as conditional PutItem writes. The failed writes increment this metric by one on evaluation to false. It also throws an HTTP 400 error.
- **ConsumedReadCapacityUnits –** It quantifies the capacity units used over a certain time period. You can use this to examine individual table and index consumption.
- **ConsumedWriteCapacityUnits –** It quantifies the capacity units used over a certain time period. You can use this to examine individual table and index consumption.
- **ReadThrottleEvents –** It quantifies requests exceeding provisioned capacity units in table/index reads. It increments on each throttle including batch operations with multiple throttles.

- **ReturnedBytes –** It quantifies the bytes returned in retrieval operations within a certain time period.

- **ReturnedItemCount –** It quantifies the items returned in Query and Scan operations over a certain time period. It addresses only items returned, not those evaluated, which are typically totally different figures.

Note: *There are many more metrics that exist, and most of these allow you to calculate averages, sums, maximum, minimum, and count.*

30. DynamoDB – CloudTrail

DynamoDB includes CloudTrail integration. It captures low-level API requests from or for DynamoDB in an account, and sends log files to a specified S3 bucket. It targets calls from the console or API. You can use this data to determine requests made and their source, user, timestamp, and more.

When enabled, it tracks actions in log files, which include other service records. It supports eight actions and two streams:

The eight actions are as follows:

- CreateTable
- DeleteTable
- DescribeTable
- ListTables
- UpdateTable
- DescribeReservedCapacity
- DescribeReservedCapacityOfferings
- PurchaseReservedCapacityOfferings

While, the two streams are:

- DescribeStream
- ListStreams

All the logs contain information about accounts making requests. You can determine detailed information like whether root or IAM users made the request, or whether with temporary credentials or federated.

The log files remain in storage for however long you specify, with settings for archiving and deletion. The default creates encrypted logs. You can set alerts for new logs. You can also organize multiple logs, across regions and accounts, into a single bucket.

Interpreting Log Files

Each file contains a single or multiple entries. Each entry consists of multiple JSON format events. An entry represents a request, and includes associated information; with no guarantee of order.

You can review the following sample log file:

```
{"Records": [
    {
        "eventVersion": "5.05",

        "userIdentity": {
```

```
            "type": "AssumedRole",
            "principalId": "AKTTIOSZODNN8SAMPLE:jane",
            "arn": "arn:aws:sts::155522255533:assumed-role/users/jane",
            "accountId": "155522255533",
            "accessKeyId": "AKTTIOSZODNN8SAMPLE",
            "sessionContext": {
                "attributes": {
                    "mfaAuthenticated": "false",
                    "creationDate": "2016-05-11T19:01:01Z"
                },
                "sessionIssuer": {
                    "type": "Role",
                    "principalId": "AKTTI44ZZ6DHBSAMPLE",
                    "arn": "arn:aws:iam::499955777666:role/admin-role",
                    "accountId": "499955777666",
                    "userName": "jill"
                }
            }
        },
        "eventTime": "2016-05-11T14:33:20Z",
        "eventSource": "dynamodb.amazonaws.com",
        "eventName": "DeleteTable",
        "awsRegion": "us-west-2",
        "sourceIPAddress": "192.0.2.0",
        "userAgent": "console.aws.amazon.com",
        "requestParameters": {"tableName": "Tools"},
        "responseElements": {"tableDescription": {
            "tableName": "Tools",
            "itemCount": 0,
            "provisionedThroughput": {
                "writeCapacityUnits": 25,
                "numberOfDecreasesToday": 0,
                "readCapacityUnits": 25
            },
            "tableStatus": "DELETING",
            "tableSizeBytes": 0
```

```
        }},
        "requestID": "4D89G7D98GF7G8A7DF78FG89AS7GFSO5AEMVJF66Q9ASUAAJG",
        "eventID": "a954451c-c2fc-4561-8aea-7a30ba1fdf52",
        "eventType": "AwsApiCall",
        "apiVersion": "2013-04-22",
        "recipientAccountId": "155522255533"
    }
]}
```

31. DynamoDB – MapReduce

Amazon's Elastic MapReduce (EMR) allows you to quickly and efficiently process big data. EMR runs Apache Hadoop on EC2 instances, but simplifies the process. You utilize Apache Hive to query map reduce job flows through HiveQL, a query language resembling SQL. Apache Hive serves as a way to optimize queries and your applications.

You can use the EMR tab of the management console, the EMR CLI, an API, or an SDK to launch a job flow. You also have the option to run Hive interactively or utilize a script.

The EMR read/write operations impact throughput consumption, however, in large requests, it performs retries with the protection of a backoff algorithm. Also, running EMR concurrently with other operations and tasks may result in throttling.

The DynamoDB/EMR integration does not support binary and binary set attributes.

DynamoDB/EMR Integration Prerequisites

Review this checklist of necessary items before using EMR:

- An AWS account
- A populated table under the same account employed in EMR operations
- A custom Hive version with DynamoDB connectivity
- DynamoDB connectivity support
- An S3 bucket (optional)
- An SSH client (optional)
- An EC2 key pair (optional)

Hive Setup

Before using EMR, create a key pair to run Hive in interactive mode. The key pair allows connection to EC2 instances and master nodes of job flows.

You can perform this by following the subsequent steps:

- Log in to the management console, and open the EC2 console located at https://console.aws.amazon.com/ec2/
- Select a region in the upper, right-hand portion of the console. Ensure the region matches the DynamoDB region.
- In the Navigation pane, select **Key Pairs**.
- Select **Create Key Pair**.
- In the **Key Pair Name** field, enter a name and select **Create**.
- Download the resulting private key file which uses the following format: filename.pem.

Note: *You cannot connect to EC2 instances without the key pair.*

Hive Cluster

Create a hive-enabled cluster to run Hive. It builds the required environment of applications and infrastructure for a Hive-to-DynamoDB connection.

You can perform this task by using the following steps:

- Access the EMR console.
- Select **Create Cluster**.
- In the creation screen, set the cluster configuration with a descriptive name for the cluster, select **Yes** for termination protection and check on **Enabled** for logging, an S3 destination for **log folder S3 location**, and **Enabled** for debugging.
- In the Software Configuration screen, ensure the fields hold **Amazon** for Hadoop distribution, the latest version for AMI version, a default Hive version for Applications to be Installed-Hive, and a default Pig version for Applications to be Installed-Pig.
- In the Hardware Configuration screen, ensure the fields hold **Launch into EC2-Classic** for Network, **No Preference** for EC2 Availability Zone, the default for Master-Amazon EC2 Instance Type, no check for Request Spot Instances, the default for Core-Amazon EC2 Instance Type, **2** for Count, no check for Request Spot Instances, the default for Task-Amazon EC2 Instance Type, **0** for Count, and no check for Request Spot Instances.

Be sure to set a limit providing sufficient capacity to prevent cluster failure.

- In the Security and Access screen, ensure fields hold your key pair in EC2 key pair, **No other IAM users** in IAM user access, and **Proceed without roles** in IAM role.
- Review the Bootstrap Actions screen, but do not modify it.
- Review settings, and select **Create Cluster** when finished.

A **Summary** pane appears on the start of the cluster.

Activate SSH Session

You need an active the SSH session to connect to the master node and execute CLI operations. Locate the master node by selecting the cluster in the EMR console. It lists the master node as **Master Public DNS Name**.

Install PuTTY if you do not have it. Then launch PuTTYgen and select **Load**. Choose your PEM file, and open it. PuTTYgen will inform you of successful import. Select **Save private key** to save in PuTTY private key format (PPK), and choose **Yes** for saving without a pass phrase. Then enter a name for the PuTTY key, hit **Save**, and close PuTTYgen.

Use PuTTY to make a connection with the master node by first starting PuTTY. Choose **Session** from the Category list. Enter hadoop@DNS within the Host Name field. Expand **Connection > SSH** in the Category list, and choose **Auth**. In the controlling options screen, select **Browse** for Private key file for authentication. Then select your private key file and open it. Select **Yes** for the security alert pop-up.

When connected to the master node, a Hadoop command prompt appears, which means you can begin an interactive Hive session.

Hive Table

Hive serves as a data warehouse tool allowing queries on EMR clusters using HiveQL. The previous setups give you a working prompt. Run Hive commands interactively by simply entering "hive," and then any commands you wish. See our Hive tutorial for more information on Hive.

32. DynamoDB – Table Activity

DynamoDB streams enable you to track and respond to table item changes. Employ this functionality to create an application which responds to changes by updating information across sources. Synchronize data for thousands of users of a large, multi-user system. Use it to send notifications to users on updates. Its applications prove diverse and substantial. DynamoDB streams serve as the main tool used to achieve this functionality.

The streams capture time-ordered sequences containing item modifications within a table. They hold this data for a maximum of 24 hours. Applications use them to view the original and modified items, almost in real-time.

Streams enabled on a table capture all modifications. On any CRUD operation, DynamoDB creates a stream record with the primary key attributes of the modified items. You can configure streams for additional information such as before and after images.

The Streams carry two guarantees:

- Each record appears one time in the stream and
- Each item modification results in the stream records of the same order as that of the modifications.

All streams process in real-time to allow you to employ them for related functionality in applications.

Managing Streams

On table creation, you can enable a stream. Existing tables allow stream disabling or settings changes. Streams offer the feature of asynchronous operation, which means no table performance impact.

Utilize the AWS Management console for simple stream management. First, navigate to the console, and choose **Tables**. In the Overview tab, choose **Manage Stream**. Inside the Manage Stream window, select the information added to a stream on table data modifications. After entering all settings, select **Enable**.

If you want to disable any existing streams, select **Manage Stream**, and then **Disable**.

You can also utilize the APIs CreateTable and UpdateTable to enable or change a stream. Use the parameter StreamSpecification to configure the stream. StreamEnabled specifies status, meaning true for enabled and false for disabled.

StreamViewType specifies information added to the stream: KEYS_ONLY, NEW_IMAGE, OLD_IMAGE, and NEW_AND_OLD_IMAGES.

Stream Reading

Read and process streams by connecting to an endpoint and making API requests. Each stream consists of stream records, and every record exists as a single modification which owns the stream. Stream records include a sequence number revealing publishing order. Records belong to groups also known as shards. Shards function as containers for several

records, and also hold information needed for accessing and traversing records. After 24 hours, records automatically delete.

These Shards are generated and deleted as needed, and do not last long. They also divide into multiple new shards automatically, typically in response to write activity spikes. On stream disabling, open shards close. The hierarchical relationship between shards means applications must prioritize the parent shards for correct processing order. You can use Kinesis Adapter to automatically do this.

Note: *The operations resulting in no change do not write stream records.*

Accessing and processing records requires performing the following tasks:

- Determine the ARN of the target stream.
- Determine the shard(s) of the stream holding the target records.
- Access the shard(s) to retrieve the desired records.

Note: *There should be a maximum of 2 processes reading a shard at once. If it exceeds 2 processes, then it can throttle the source.*

The stream API actions available include

- ListStreams
- DescribeStream
- GetShardIterator
- GetRecords

You can review the following example of the stream reading:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClient;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBStreamsClient;
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.Tables;

public class StreamsExample {

    private static AmazonDynamoDBClient dynamoDBClient =
        new AmazonDynamoDBClient(new ProfileCredentialsProvider());

    private static AmazonDynamoDBStreamsClient streamsClient =
        new AmazonDynamoDBStreamsClient(new ProfileCredentialsProvider());

    public static void main(String args[]) {

        dynamoDBClient.setEndpoint("InsertDbEndpointHere");

        streamsClient.setEndpoint("InsertStreamEndpointHere");

        // table creation
        String tableName = "MyTestingTable";

        ArrayList<AttributeDefinition> attributeDefinitions =
            new ArrayList<AttributeDefinition>();

        attributeDefinitions.add(new AttributeDefinition()
```

```
            .withAttributeName("ID")
            .withAttributeType("N"));

        ArrayList<KeySchemaElement> keySchema = new
ArrayList<KeySchemaElement>();
        keySchema.add(new KeySchemaElement()
            .withAttributeName("ID")
            .withKeyType(KeyType.HASH)); //Partition key

        StreamSpecification streamSpecification = new StreamSpecification();
        streamSpecification.setStreamEnabled(true);

streamSpecification.setStreamViewType(StreamViewType.NEW_AND_OLD_IMAGES);

        CreateTableRequest createTableRequest = new CreateTableRequest()
            .withTableName(tableName)
            .withKeySchema(keySchema)
            .withAttributeDefinitions(attributeDefinitions)
            .withProvisionedThroughput(new ProvisionedThroughput()
                .withReadCapacityUnits(1L)
                .withWriteCapacityUnits(1L))
            .withStreamSpecification(streamSpecification);

        System.out.println("Executing CreateTable for " + tableName);
        dynamoDBClient.createTable(createTableRequest);

        System.out.println("Creating " + tableName);
        try {
            Tables.awaitTableToBecomeActive(dynamoDBClient, tableName);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Get the table's stream settings
        DescribeTableResult describeTableResult =
```

```
dynamoDBClient.describeTable(tableName);

        String myStreamArn =
describeTableResult.getTable().getLatestStreamArn();
        StreamSpecification myStreamSpec =
            describeTableResult.getTable().getStreamSpecification();

        System.out.println("Current stream ARN for " + tableName + ": "+
myStreamArn);
        System.out.println("Stream enabled: "+ myStreamSpec.getStreamEnabled());
        System.out.println("Update view type: "+ myStreamSpec.getStreamViewType());

        // Add an item
        int numChanges = 0;
        System.out.println("Making some changes to table data");
        Map<String, AttributeValue> item = new HashMap<String, AttributeValue>();
        item.put("ID", new AttributeValue().withN("222"));
        item.put("Alert", new AttributeValue().withS("item!"));
        dynamoDBClient.putItem(tableName, item);
        numChanges++;

        // Update the item
        Map<String, AttributeValue> key = new HashMap<String, AttributeValue>();
        key.put("ID", new AttributeValue().withN("222"));
        Map<String, AttributeValueUpdate> attributeUpdates =
            new HashMap<String, AttributeValueUpdate>();
        attributeUpdates.put("Alert", new AttributeValueUpdate()
            .withAction(AttributeAction.PUT)
            .withValue(new AttributeValue().withS("modified item")));
        dynamoDBClient.updateItem(tableName, key, attributeUpdates);
        numChanges++;

        // Delete the item
        dynamoDBClient.deleteItem(tableName, key);

        numChanges++;
```

```
        // Get stream shards
        DescribeStreamResult describeStreamResult =
            streamsClient.describeStream(new DescribeStreamRequest()
                .withStreamArn(myStreamArn));
        String streamArn =
            describeStreamResult.getStreamDescription().getStreamArn();
        List<Shard> shards =
            describeStreamResult.getStreamDescription().getShards();

        // Process shards
        for (Shard shard : shards) {
            String shardId = shard.getShardId();
            System.out.println(
                "Processing " + shardId + " in "+ streamArn);

            // Get shard iterator
            GetShardIteratorRequest getShardIteratorRequest = new
GetShardIteratorRequest()
                .withStreamArn(myStreamArn)
                .withShardId(shardId)
                .withShardIteratorType(ShardIteratorType.TRIM_HORIZON);
            GetShardIteratorResult getShardIteratorResult =
                streamsClient.getShardIterator(getShardIteratorRequest);
            String nextItr = getShardIteratorResult.getShardIterator();

            while (nextItr != null && numChanges > 0) {

                // Read data records with iterator
                GetRecordsResult getRecordsResult =
                    streamsClient.getRecords(new GetRecordsRequest().
                        withShardIterator(nextItr));
                List<Record> records = getRecordsResult.getRecords();
                System.out.println("Pulling records...");

                for (Record record : records) {
                    System.out.println(record);
                    numChanges--;
```

```
                }
                nextItr = getRecordsResult.getNextShardIterator();
            }
        }
    }
}
```

33. DynamoDB – Error Handling

On unsuccessful processing of a request, DynamoDB throws an error. Each error consists of the following components: HTTP status code, exception name, and message. Error management rests on your SDK, which propagates errors, or your own code.

Codes and Messages

Exceptions fall into different HTTP header status codes. The 4xx and 5xx hold errors related to request issues and AWS.

A selection of exceptions in the HTTP 4xx category are as follows:

- **AccessDeniedException –** The client failed to sign the request correctly.
- **ConditionalCheckFailedException –** A condition evaluated to false.
- **IncompleteSignatureException –** The request included an incomplete signature.

Exceptions in the HTTP 5xx category are as follows:

- Internal Server Error
- Service Unavailable

Retries and Backoff Algorithms

Errors come from a variety of sources such as servers, switches, load balancers, and other pieces of structures and systems. Common solutions consist of simple retries, which supports reliability. All SDKs include this logic automatically, and you can set retry parameters to suit your application needs.

For example – Java offers a maxErrorRetry value to stop retries.

Amazon recommends using a backoff solution in addition to retries in order to control flow. This consists of progressively increasing wait periods between retries and eventually stopping after a fairly short period. Note SDKs perform automatic retries, but not exponential backoff.

The following program is an example of the retry backoff:

```
public enum Results {
    SUCCESS,
    NOT_READY,
    THROTTLED,
    SERVER_ERROR
}
```

```
public static void DoAndWaitExample() {

    try {
        // asynchronous operation.
        long token = asyncOperation();

        int retries = 0;
        boolean retry = false;

        do {
            long waitTime = Math.min(getWaitTime(retries), MAX_WAIT_INTERVAL);

            System.out.print(waitTime + "\n");

            // Pause for result
            Thread.sleep(waitTime);

            // Get result
            Results result = getAsyncOperationResult(token);

            if (Results.SUCCESS == result) {
                retry = false;
            } else if (Results.NOT_READY == result) {
                retry = true;
            } else if (Results.THROTTLED == result) {
                retry = true;
            } else if (Results.SERVER_ERROR == result) {
                retry = true;
            }
            else {
                // stop on other error
                retry = false;
            }

        } while (retry && (retries++ < MAX_RETRIES));
    }
```

```
    catch (Exception ex) {
    }
}

public static long getWaitTime(int retryCount) {

    long waitTime = ((long) Math.pow(3, retryCount) * 100L);

    return waitTime;
}
```

34. DynamoDB – Best Practices

Certain practices optimize code, prevent errors, and minimize throughput cost when working with various sources and elements.

The following are some of the most important and commonly used best practices in DynamoDB.

Tables

The distribution of tables means the best approaches spread read/write activity evenly across all table items.

Aim for uniform data access on table items. Optimal throughput usage rests on primary key selection and item workload patterns. Spread the workload evenly across partition key values. Avoid things like a small amount of heavily used partition key values. Opt for better choices like large quantities of distinct partition key values.

Gain an understanding of partition behavior. Estimate partitions automatically allocated by DynamoDB.

DynamoDB offers burst throughput usage, which reserves unused throughput for "bursts" of power. Avoid heavy use of this option because bursts consume large amounts of throughput quickly; furthermore, it does not prove a reliable resource.

On uploads, distribute data in order to achieve better performance. Implement this by uploading to all allocated servers concurrently.

Cache frequently used items to offload read activity to the cache rather than the database.

Items

Throttling, performance, size, and access costs remain the biggest concerns with items. Opt for one-to-many tables. Remove attributes and divide tables to match access patterns. You can improve efficiency dramatically through this simple approach.

Compress large values prior to storing them. Utilize standard compression tools. Use alternate storage for large attribute values such as S3. You can store the object in S3, and an identifier in the item.

Distribute large attributes across several items through virtual item pieces. This provides a workaround for the limitations of item size.

Queries and Scans

Queries and scans mainly suffer from throughput consumption challenges. Avoid bursts, which typically result from things like switching to a strongly consistent read. Use parallel scans in a low-resource way (i.e., background function with no throttling). Furthermore, only employ them with large tables, and situations where you do not fully utilize throughput or scan operations offer poor performance.

Local Secondary Indices

Indexes present issues in the areas of throughput and storage costs, and the efficiency of queries. Avoid indexing unless you query the attributes often. In projections, choose wisely because they bloat indexes. Select only those heavily used.

Utilize sparse indexes, meaning indexes in which sort keys do not appear in all table items. They benefit queries on attributes not present in most table items.

Pay attention to the item collection (all table items and their indices) expansion. Add/update operations cause both tables and indexes to grow, and 10GB remains the limit for collections.

Global Secondary Indices

Indexes present issues in the areas of throughput and storage costs, and the efficiency of queries. Opt for key attributes spreading, which like read/write spreading in tables provides workload uniformity. Choose attributes which evenly spread data. Also, utilize sparse indexes.

Exploit global secondary indices for fast searches in queries requesting a modest amount of data.

35. DynamoDB – Useful Resources

The following resources offer more in-depth information on DynamoDB, development, and administration. Consult them to continue your studies.

Useful DynamoDB Websites

- DynamoDB Official Knowledge Base – This resource serves the best and latest DynamoDB information, code examples, tips, tricks, fixes, and a comprehensive collection of related information. Amazon offers tutorials, videos, classes, labs, white papers, and much more.

- LinkedIn Pulse Content – LinkedIn, described as the world's largest professional network, allows over 100 million professionals to network. It opened its publishing platform, Pulse, to the public in 2014, resulting in a wealth of valuable professional and industry information. This information includes rare insight, training media, and more related to DynamoDB.

- Lynda.com DynamoDB Videos – This online learning organization offers thousands of videos on various topics (including DynamoDB) supplied by professionals, organizations, and individuals.

- YouTube DynamoDB Videos – Individuals just like you produce thousands of DynamoDB videos on YT to address particular topics not covered elsewhere, or covered poorly elsewhere. These videos use different languages, styles, and offer unique voices.

If you would like your site or literature listed on this page, contact us at

contact@tutorialspoint.com