Amazon DynamoDB Lecture Notes

In the previous sessions of this course, you learnt about HBase, which is a columnar-type NoSQL datastore service offered by Apache.

In this module, you were introduced to DynamoDB, which is NoSQL database service offered by Amazon AWS and supports both key-value data store and document data store.

Fundamentals of NoSQL Databases

You learnt that unlike relational database, where there is a predefined schema, NoSQL database supports a schemaless architecture, and the design of the database can be changed very easily. There are primarily four different types of NoSQL database architectures:

1. **Key-value data store**
2. **Column-oriented data store**
3. **Document data store**
4. **Graph data store**

In document data stores, a collection of values describe a certain item. The values are unique to a specific item and are compressed together to form a document describing or representing that item. The values stored in the document usually follow a specific industry-relevant format, such as JSON, XML, or BSON.

A relational database supports a schema-on-write operation, where the database has a predefined schema and the data added to this database conforms to this predefined structure, as shown in the table below:

Table 1: A Sample Relational Table

Employee ID	Employee Name	Age	City	Salary
1	Shiva	26	Hyderabad	50,000
2	Karanpreet	25	Mumbai	75,000
3	Salma	22	Mumbai	60,000
4	Jeanona	33	Mumbai	1,00,000

A NoSQL database, on the other hand, doesn't require any predefined schema. This means unstructured data can be easily added to a NoSQL database. As you can see in the table below, different types of products with different components can easily be stored using a single table.

Table 2: A Sample NoSQL Table

Product Category	Product Type	Contents
Electronics	Mobile	Model No.: iPhone X Storage: 64 GB Colour: Space Grey Camera: 12 MP
Electronics	Laptop	Model No.: Dell Vostro Storage: 1 TB OS: Windows 10 RAM: 8 GB
Home Appliances	Microwave	Manufacturer: LG Mode No.: LG-HMW Capacity: 25 L Colour: White
Home Appliances	Television	Manufacturer: Samsung Type: LED Resolution: 4K Size: 19 Inches
Entertainment	Movie	Title: Titanic Cast: Leonardo Di Caprio, Kate Winslet Director: James Cameroon IMDb Rating: 7.8/10

Basic Features of DynamoDB

In DynamoDB, you learnt that data is stored in the form of key-value pairs. A sample table shown below, represents a key-value pair relationship:

Table 3: A Key-Value Pair Example

Key	Value
Country	India
State	Maharashtra
City	Mumbai

Pin code	400018

Every row in such a table is called an item, where each item is made of a key-value pair. The key declared should be unique and should have a value. The key can be a string, a number, or a binary, and the value can be either single-valued or a multivalued set, e.g. JSON or BLOB (Basic Large Object).

Furthermore, the various key features of DynamoDB are as follows:

1. **Auto Scaling**: DynamoDB **scales up** the platform capacities automatically when the databases experience heavy user traffic or a sudden spike in the number of users.

2. **High availability**: DynamoDB **automatically replicates data** stored in the database across the availability zones. This process ensures **high availability** and no loss of data even if one of the zones loses the data because of individual machine failure.

3. **Self partitioning:** DynamoDB allocates additional partitions to a table if the table's provisioned throughput capacities are increased beyond what the existing partitions can support, or if an existing partition fills to its capacity and more storage space is required. Note that partition management occurs automatically in the background and does not require any manual intervention.

4. **Performance monitor**: DynamoDB provides tools to monitor the performance of the database by letting the user compare the consumption of the read-and-write capacity units against the provisioned/set throughput capacities. This feature helps the user **scale down the capacities** in case less units are being utilized, in order to **save on cost**.

Core Components of DynamoDB

In this module, you were introduced to various core components of a DynamoDB table, which included items, attributes, partition keys, etc. We discussed in detail the significance and functioning of each component of a table.

The core components of DynamoDB are as follows:

- **Table**: As in RDBMS systems, DynamoDB stores the data in the form of tables. A table in DynamoDB is a **collection of items**, and every table should have a **unique name** for a specific account and region.

- **Item:** Data in a table is stored in the form of items, where each row represents a unique item. The maximum size of an item in a table can be **400 KB.** In the session videos, you learnt that each row,

i.e. each order in the sample e-commerce database we created in the video, which was stored in a specific row, served as an item in the table.

- **Primary key (hash key)**: This is a mandatory attribute defined at the time of creating a table. Each primary key should have a **unique value**, and no two items can have the same primary key value in a table, unless we have declared an optional sort key. In the table show below, **CustomerID** can operate as a primary key and **OrderID** as a sort key, to give the desired item.

Table 4: NoSQL Database

Items	**Primary key**	**Sort key**	**Attributes**
	CustomerID	**OrderID**	
Item 1	101	Order001	Model No.: iPhone X Storage: 64 GB Colour: Space Grey Camera: 12 MP
Item 2	101	Order007	Model No.: Dell Vostro Storage: 1 TB OS: Windows 10 RAM: 8 GB
Item 3	102	Order111	Title: Titanic Cast: Leonardo Di Caprio, Kate Winslet Director: James Cameroon IMDb Rating: 7.8/10

Components of DynamoDB Table

- **Sort key (range key)**: This is an optional attribute declared while creating the table. However, declaring the sort key helps in narrowing down the search results while querying a particular set of data.

- **Attribute:** An attribute can be considered a property associated with an item. Each item can have one or more attributes, and it is the **fundamental element of data**, which can't be broken down further.

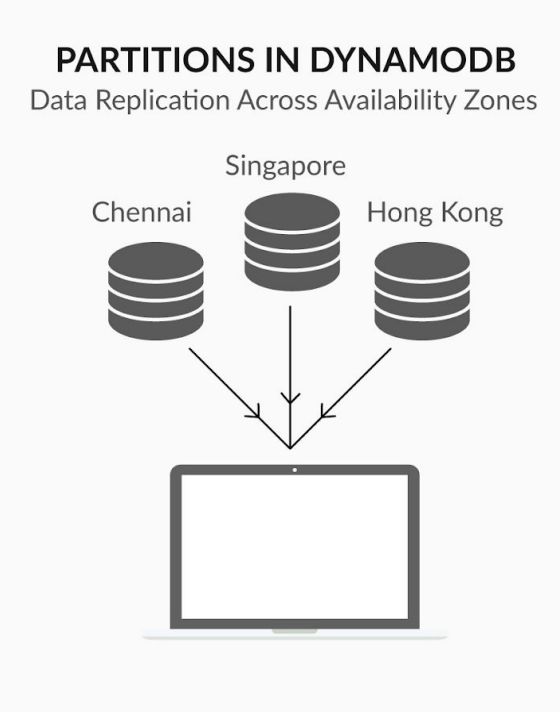
Partitions & Composite Key

In the previously explained concepts, you learnt about various core components, such as tables, items, attributes, etc., of a DynamoDB table. Now let's summarize and learn all about partitions and composite keys in a DynamoDB table

Partitions:

- An important feature of DynamoDB is the way data is stored. Data in DynamoDB is stored in partitions. All the data written into any table is stored in a specific partition, which is an allocation of storage for a table and is backed by **solid-state drives (SSDs).**

- The partition is automatically replicated across multiple **availability zones** within an AWS region to ensure complete availability of data in the table. This ensures that if one availability zone is down, or has lost any data, then that data can be easily fetched from another zone.



- In order to decide which data gets allocated to which partition, DynamoDB makes use of the partition key.

- Partition management is handled entirely by DynamoDB; you never have to manage partitions by yourself.

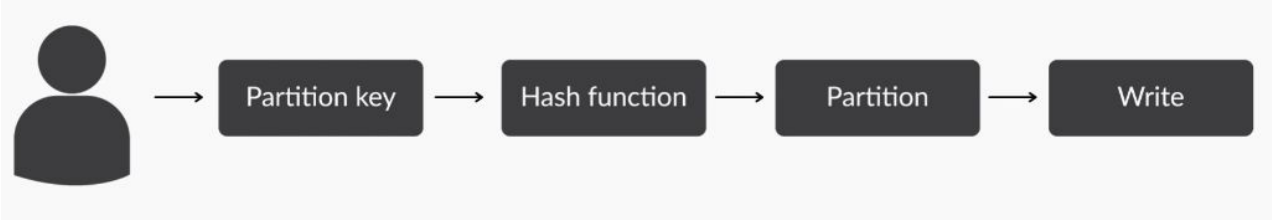
- DynamoDB makes use of the **partition key** to allocate any data to a specific partition.

- Partition allocation is completely a **self-managed** process handled by DynamoDB.

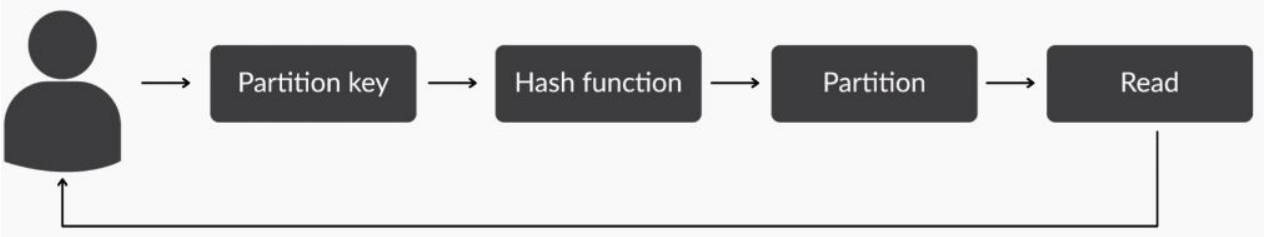
Partition Key:

- Each item getting stored in DynamoDB is required to have a unique primary key, also known as the partition key. This partition key functions as a **unique key identifier** to get access to the desired item.

- When you write an item into the database, the partition for that specific item is decided on the basis of the **partition key value**. DynamoDB passes the partition key as an input to an **input hash function.** The output from this hash function determines the partition in which the item will be stored.



- As in the case of reading an item from the table, DynamoDB uses the partition key value as input to its hash function, yielding the partition in which the item can be found.



Composite Key (Partition and Sort Keys)

- A combination of the partition and sort keys is commonly known as a composite key. Sometimes data items can be segregated on a broad level using a partition key, but they still require an additional key, i.e., a sort key to uniquely identify the data item.
- As mentioned earlier, declaring the sort key helps a user in narrowing down the search results while querying a particular type of data.

Create a DynamoDB Table

In this exercise, you learnt how to create a DynamoDB table by writing a Java program. The name of the table is 'CustomerData', and it contains the data of all the orders placed by some customers on an e-commerce website. It is made up of primary and sort keys, which are 'CustomerID' and 'OrderID', respectively.

The code for creating a table is shown below:

```
package com.amazonaws.samples;
import java.util.Arrays;
import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;

public class Customerdata {

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

      AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
                                                          .withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
          .build();

      DynamoDB dynamoDB = new DynamoDB(client);
      String tableName = "CustomerData";
      try {
         System.out.println("Attempting to create table; please wait...");
         Table table = dynamoDB.createTable(tableName,
```

```
            Arrays.asList(new KeySchemaElement("CustomerID", KeyType.HASH), // Partition Key
               new KeySchemaElement("OrderID", KeyType.RANGE)), // Sort key
            Arrays.asList(new AttributeDefinition("CustomerID", ScalarAttributeType.S),

               new AttributeDefinition("OrderID", ScalarAttributeType.S)),
            new ProvisionedThroughput(10L, 10L));
        table.waitForActive();
        System.out.println("Success.  Table status: " + table.getDescription().getTableStatus());

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

  }
}
```

In the code, it is shown that the first statement calls the java.utils.Arrays package. Along with this, a few more packages need to be called to create a table. All the packages starting with com.amazonaws are used to program DynamoDB.

Next, you will have to create a class, which in this case is 'Customerdata'. Inside the main method of this class, you will have to create an instance of AmazonDynamoDB class, which in this case is 'client'. This instance is used to connect to a data center using the withEndpointConfiguration() method. This means that you need to specify a URL of the data center where the request should be sent and from where the response should be received. In this case, the data center is Oregon (West), which is one of the AWS regions. The URL for Oregon (West) is https://dynamodb.us-west-2.amazonaws.com. Here, us-west-2 refers to the region Oregon (West).

To access the list of all the AWS regions, click here. You can choose any data center or AWS region you wish, but stick only to one; do not change the AWS regions.

Next, you need to create an instance of the class DynamoDB, which in this case is 'dynamoDB'. Its constructor takes 'client' as the input parameter. The 'client' variable contains all the required information of the data center situated in Oregon (West).

You now need to declare a string type variable called 'tableName', which will store the name of the table. In this example, the name of the table is 'CustomerData'.

Next, in the try block, you need to create an instance of the 'Table' class with the help of the dynamoDB instance. While creating the instance, use the createTable() method to create the required table. The

input parameter for the createTable() method is the tableName variable, which was declared earlier. Along with tableName, you need to pass the Arrays.asList() method, which indicates that a table is a list of items.

Inside the Arrays.asList() method, you need to pass the KeySchemaElement() method, which is used to declare the primary and sort keys. The keywords to distinguish between primary and sort keys are KeyType.HASH and KeyType.RANGE respectively. Furthermore, you need to specify the attribute type of the primary and sort keys. To do this, you need to use the keyword ScalarAttributeType. You already know that both the keys can have only scalar attributes, which are string, or number, or binary. The word scalar means that the attribute value will be a single value, not a multi-value.

ScalarAttributeType.S indicates that the attribute type for the key is a scalar string.
ScalarAttributeType.N indicates that the attribute type for the key is a scalar number.
ScalarAttributeType.B indicates that the attribute type for the key is a scalar binary.

To define the attribute type, you need to use the AttributeDefinition() method.

Finally, you need to finish the Arrays.asList() method by specifying the throughput values using the ProvisionedThroughput() method. Throughput values indicate the speed at which data is read from the table and the speed at which data is entered in a table.

To measure the reading speed, the RCU metric is used, which stands for Read Capacity Units, and to measure the writing speed, the WCU metric is used, which stands for Write Capacity Units. In this example, the value for both RCU and WCU is 10, which means you can read 10 units of the table per unit time and can write 10 units in the table per unit time.

If you wish to create a table by writing a Java program, you will have to specify the throughput values mandatorily.

Finally, you can finish the try block by writing the table.waitForActive() statement. This statement means that DynamoDB is creating the table requested by the user but is not yet ready to use. Once the status of table turns to ‘active’, which means that the table is successfully created, you can add data into the table or view the table.
Inside the System.out.println() method, the getDescription() and getTableStatus() methods can be used to get the table details.

Loading Data into a Table

To add data into the table, the following code is used:

```
package com.amazonaws.samples;


import java.io.File;
// Imported for reading, writing on other files and other operations.


import java.util.Iterator;
//This package needs to be installed to iterate over different data structure elements like array, list and all.


import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
// This package is imported to load items in DynamoDB tables.


import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
// JsonParser contains methods to parse JSON data using the streaming model
// JsonParser provides forward, read-only access to JSON data using the pull parsing programming model.
// In this model, the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser


import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;
// Jackson is an open source library to process JSON.
```

```
public class LoadData {

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

      AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
           // Creates new instance of builder with all defaults set.
                                                          .withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
          .build();

      DynamoDB dynamoDB = new DynamoDB(client);

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

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

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

      ObjectNode currentNode;

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

         String customerId = currentNode.path("CustomerID").asText();
         String orderId = currentNode.path("OrderID").asText();

         try {
```

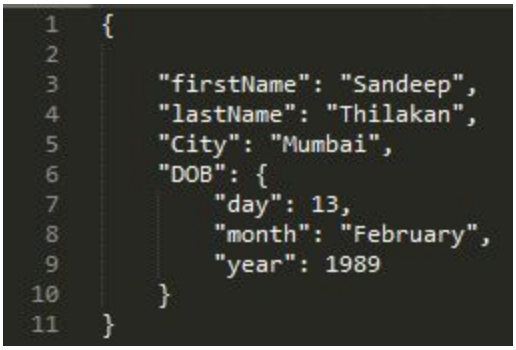
```
                    table.putItem(new Item().withPrimaryKey("CustomerID", customerId, "OrderID",
orderId).withJSON("info",
                currentNode.path("info").toString()));
            System.out.println("PutItem succeeded: " + customerId + " " + orderId);
            // Putting new item in the table


          }
          catch (Exception e) {
            System.err.println("Unable to add item: " + customerId + " " + orderId);
            System.err.println(e.getMessage());
            break;
          }
        }
        parser.close();
    }
}
```

To execute the code above, a new set of libraries needs to be imported in the code, which is as follows:

```
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;
```

These libraries are used to parse a JSON file.

A JSON file is like a TSV or a CSV file, which stores data in the form of objects, or, to be precise, in the form of key-value pairs. The image shown below demonstrates how data is stored in a JSON file in the form of key-value pairs.



As it is shown in the image, “firstName”, “lastName”, and “City” are keys and “Sandeep”, “Thilakan”, and “Mumbai” are their corresponding values. “DOB” is also a key whose value, i.e. {“day”: 13, “month”: “February”, “year”: 1989}, is a list of key-value pairs.

The keys are always written within double quotes followed by a colon. The values can be strings, numbers, lists, arrays, binary numbers, or boolean. String type values are always written within double quotes, list type values are enclosed within curly brackets, or ‘{}’, such that each bracket-separated entity inside the list is separated by a comma, and array type values are enclosed within square brackets, or ‘[]’, such that each bracket-separated entity inside an array is also separated by a comma.

Coming back to the code, the first few lines of the code are going to remain to the same, which are as follows:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()

.withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
        .build();

DynamoDB dynamoDB = new DynamoDB(client);
```

The code snippet above creates a DynamoDB client and establishes the connection between Eclipse and one of the AWS regions. In that region, DynamoDB will create the table. The following link lists all the AWS regions.

AWS Regions

Now, a user needs to provide the name of the table that needs to be populated through the JSON file. In this example, a table called ‘CustomerData’ needs to be populated through the ‘customerdata.json’ file. All of this is done by writing the following code:

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

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

Now, the data from JSON is received through tree after the createParser() method has parsed the required JSON file. Once a JSON file is parsed, the data in it gets broken down into different components and forms a tree. The tree comprises multiple nodes, and each node corresponds to an item in the JSON file. The following code snippet shows how data is received in the form of a tree.

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

ObjectNode currentNode;
```

Next, to read through the nodes, a while loop is used, as shown in the code snippet below:

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

    String customerId = currentNode.path("CustomerID").asText();
    String orderId = currentNode.path("OrderID").asText();
```

The while loop will iterate through the tree using the hasNext() method. This method will check whether there is any component present in the tree or not. As long as a component is present, the hasNext() method will transfer that particular component to its relevant object node and move on to the next component. This process will keep repeating itself until there is no component present to read further, and then, the while loop will break causing the iteration to stop.

All the components in the tree will be mapped as string type values. This is done using the asText() method. Once the components are mapped, the items will be added to their designated table using the putItem() method.

To read more about JSON, click here.

Adding a New Item to a Table

A new item can be added to a table anytime by simply writing a Java program. A user can add one or more items in the table by following this process:

To add an item to a table, a user must provide the following details:

1. The name of the table in which the required item needs to be added. This is done using the following statement:

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

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the required item. This is done using the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";
```

3. If an item consists of maps or a list of key-value pairs, then all the key-value pairs of that item are specified in the following manner:

```
final Map<String, Object> infoMap = new HashMap<String, Object>();
infoMap.put("city", "Pune.");
infoMap.put("Region", "West");
infoMap.put("State Code", "20");
```

Once all the required details of the item are provided, the item can be added using the putItem() method as shown below:

```
try {
    System.out.println("Adding a new item...");
    PutItemOutcome outcome = table
       .putItem(new Item().withPrimaryKey("CustomerID", customerId, "OrderID", orderId).withMap("info",
infoMap));

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

      }
```

Reading a Table Item

To read an item of a table, a user must provide the following details:

1. The name of the table in which that item exists. This is done by writing the following statement:

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

2. The primary key and sort key (mandatory only if the table also has the sort key) values of that item. This can be done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";

GetItemSpec spec = new GetItemSpec().withPrimaryKey("CustomerID", customerId, "OrderID", orderId);
```

3. Finally, the required item can be retrieved using the getItem() method as shown below:

```
try {
```

```
System.out.println("Attempting to read the item...");
Item outcome = table.getItem(spec);
System.out.println("GetItem succeeded: " + outcome);

}
```

Updating a Table Item

To update an existing item in a table, a user must specify the following details:

1. The name of the table in which hat item exists, as shown below:

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

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the item, which need to be updated. This is done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";
```

3. The values that needs to be updated in the item. In this example, a new key 'hits' needs to be added in the 'info' map, whose value is a number, '1'. The following code snippet tells how it is done:

```
UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("CustomerID", customerId,
        "OrderID", orderId)

                .withUpdateExpression("set info.hits = :h")
    .withValueMap(new ValueMap().withNumber(":h", 1))
    .withReturnValues(ReturnValue.UPDATED_NEW);

      try {
      System.out.println("Updating the item...");
      UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
      System.out.println("UpdateItem succeeded:\n" + outcome.getItem().toJSONPretty());

}
```

The key that needs to be added is mentioned in the withUpdateExpression method, and its corresponding value is passed through the withValueMap() constructor.

Finally, the item is updated using the updateItem() method.

Deleting an Item from a Table

A table can be deleted by writing a Java program.

To delete an item from a table, a user must provide the following details:

1. The name of the table in which that item exists. For example,

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

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the item that need to be deleted from a table. This is done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";

DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
    .withPrimaryKey(new PrimaryKey("CustomerID", "IN-80134", "OrderID", "CA-2018-345678"));
```

The item gets deleted using the deleteItem() method, as shown below:

```
    try {
        System.out.println("Attempting to delete an item...");
        table.deleteItem(deleteItemSpec);
        System.out.println("DeleteItem succeeded");
    }
```

Retrieving All Table Items using the Scan Operation

The scan operation is used to retrieve all the items of a table.

To retrieve all the items of a table, a user must provide the name of the table, all the items of which need to be retrieved. After that, the scan() method needs to be used for retrieving all the items of the table, as shown below:

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


    try {
        ItemCollection<ScanOutcome> items = table.scan();

        Iterator<Item> iter = items.iterator();
        while (iter.hasNext()) {
            Item item = iter.next();
```

```
            System.out.println(item.toString());
        }

    }
```

Retrieving All Table Items using the Query Operation

The query operation is used to retrieve some specific, not all, items of the table.

To query a DynamoDB table —

1. A user must specify the name of the table, the items of which need to be retrieved using the query operation.
2. Then, the user needs to specify the attribute value, which will act as a filter to produce the item(s) that correspond to this value. Here's the code snippet:

```
HashMap<String, String> nameMap = new HashMap<String, String>();
nameMap.put("#cu", "CustomerID");
```

In the snippet above, the name of the attribute that needs to be referred to must be specified to make a query successful. In this example, the attribute name is 'CustomerID', which is also the primary key. The attribute name needs to be passed through the nameMap.put() method. Here, '#cu' acts as an alias for 'CustomerID'. Later in the code, it will be addressed as '#cu'. Here's the relevant code snippet:

```
HashMap<String, Object> valueMap = new HashMap<String, Object>();
valueMap.put(":c", "BH-11710");
```

In the code snippet above, the attribute value that acts as a filter needs to be specified. This means that only the items corresponding to this attribute value will be searched for and other items will be ignored. Here, 'BH-11710' is the attribute value that acts as a filter and needs to pass through the valueMap.put() method. So, all the items corresponding to 'BH-11710' will be queried. Here, ':c' acts as an alias for 'BH-11710'. Later in the code, it will be addressed as ':c'.

In a nutshell, the two HashMap statements above mean that in the 'CustomerData' table, a user is looking for all the items that correspond only to 'BH-11710'.

3. Next, the attribute value needs to be assigned to the attribute name using the following statement:

```
QuerySpec querySpec = new QuerySpec().withKeyConditionExpression("#cu =
        :c").withNameMap(nameMap)
.withValueMap(valueMap);
```

Here, '**#cu = :c**' means that the attribute name 'CustomerID' is assigned to the value 'BH-11710'.

4. After this, an item collection variable needs to be declared because the query may fetch either one item or a collection of items. When you declare the variable, set its initial value to null, as follows:

```
ItemCollection<QueryOutcome> items = null;
```

In this example, 'items' is the ItemCollection variable, which will store the scanned values once the query operation is executed.

5. Next, you need to loop through the table to scan all the items corresponding to the value ':c':

```
try {
   System.out.println("Orders by CustomerID = BH-11710");
   items = table.query(querySpec);
```

In the statements above, the query() method is used to initialise the scan operation on the 'CustomerData' table. The instance 'querySpec' specifies how to filter the items.

```
        iterator = items.iterator();
    while (iterator.hasNext()) {
          item = iterator.next();
          System.out.println(item.getString("CustomerID") + ": " + item.getString("OrderID"));
       }

    }
```

The while loop then reads each item and adds it corresponding to ':c' in the items' list. The output will be received in the form of strings because the getString() method is used.

Initial Partition Allocation

partition, which is an allocation of storage for a table and is backed by **solid-state drives (SSDs).**

The two primary factors affecting the number of initial partitions in a DynamoDB table are —

- The provisioned throughput capacities
- The size of the data stored in the table

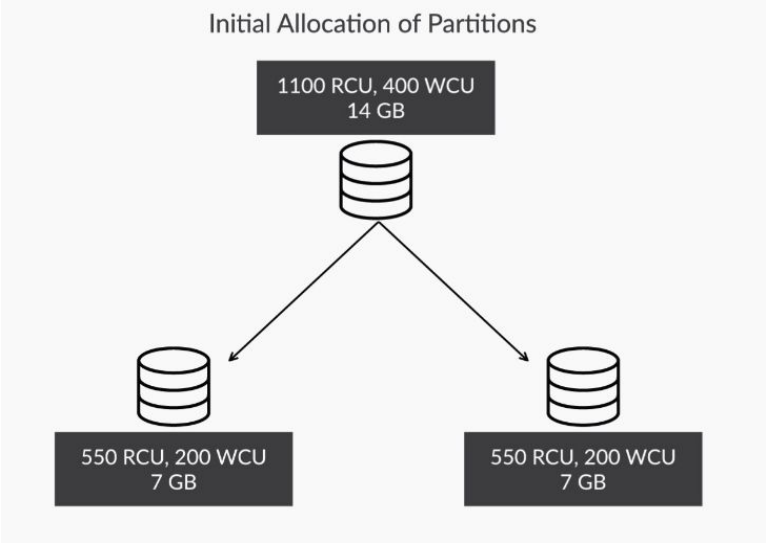
The provisioned throughput capacities, i.e. RCUs and WCUs, determine the number of the initial partitions that are created in a DynamoDB table. Further, DynamoDB divides the throughput capacities evenly among all the available partitions. Each partition can support a maximum of 3000 RCUs and 1000 WCUs. The formula needed to estimate the number of partitions is as follows:

(Desired RCUs/3000) + (Desired WCUs/1000) = Initial partitions (value rounded off to the next integer)

If a table is created with 1100 RCUs and 400 WCUs, then the number of initial partitions created = 1100/3000 + 400/1000 = 0.366 + 0.4 = 0.766. Rounding off this value to the next integer gives a value of 1. Hence, DynamoDB will create 1 partition based on the initial throughput values.

In DynamoDB, each partition can support a maximum data size of 10 GB. The total data is equally distributed among all the initial partitions created by DynamoDB.

If ever size of the data in one partition exceeds 10 GB, DynamoDB will split that partition into two partitions and will divide the data as well as the throughput values evenly across both the partitions, as shown in the image below.



The initial number of allocation of partitions in DynamoDB is the maximum of partitions created based on the throughput requirements and the partitions created based on the size of the data.

For example, if based on throughput values 4 partitions are required and based on size of the data 3 partitions are required then DynamoDB will create 4 partitions.

Secondary Indexing

A secondary index is a data structure created from a subset of attributes from a database and contains an alternate primary key, which can be used to run queries or scan operations. A secondary index helps you access specific data attributes easily, saving both time and money.

DynamoDB offers two types of secondary indexing, which are as follows:

- Global secondary indexing

- Local secondary indexing

Secondary indexing is performed on any created database table, which is also referred to as the base table. It is important to realise here that **though each secondary index is affiliated to only one base table, a single base table can have any number of secondary indexes.**

Table 5: Base Table

Player ID	Player Name	Match Series	Year	Total Score	Strike Rate
101	Virat Kohli	India vs South Africa	2016	450	96
101	Virat Kohli	India vs Australia	2015	360	92
101	Virat Kohli	India vs Sri Lanka	2016	270	92
102	Rohit Sharma	India vs South Africa	2015	287	106
102	Rohit Sharma	India vs Australia	2016	345	92
103	A. Rahane	India vs South Africa	2016	289	101
103	A. Rahane	India vs Australia	2015	328	86
103	A. Rahane	India vs Sri Lanka	2016	221	89

Global Secondary Index: In global secondary indexing, both the partition and sort keys can be different from those in the base table. In other words, the schema of a global secondary index can be distinct from that of the base table. DynamoDB allows you the flexibility of creating up to **five global secondary indexes.**

The steps needed to create a global secondary index are —

- Define **alternate partition and sort keys**. For the base table shown before, define **Match Series** as an alternate partition key and **Total Score** as an alternate sort key.

- Create a global secondary index. The **primary key of a base table** also gets imported to the new index being created. In this example, primary key, **Player ID**, will also get imported to the global secondary index.

- Define the attributes that need to be copied or imported from the base table to the index being created. DynamoDB then copies the defined attributes into the global secondary index. For this example, set **Player Name** as an attribute that needs to be imported along with the alternate composite key from the base table.

Table 6: Global Secondary Index

Base Table Partition Key	Alternate Partition Key	Alternate Sort Key	Imported Attribute
Player ID	Match Series	Total Score	Player Name
101	India VS South Africa	450	Virat Kohli
102	India VS South Africa	287	Rohit Sharma
103	India VS South Africa	289	A. Rahane
101	India VS Australia	360	Virat Kohli
102	India VS Australia	345	Rohit Sharma
103	India VS Australia	328	A. Rahane
101	India VS Sri Lanka	270	Virat Kohli
103	India VS Sri Lanka	221	A. Rahane

Local Secondary Index: A local secondary index has the **same partition key** as the base table, but its sort key and attributes can be different from those of the base table. DynamoDB allows you the flexibility of creating up to **five local secondary indexes.**

The steps needed to create a local secondary index are —

- Create a local secondary index that you need to keep your partition key the same as in your base table, and define an alternate sort key. For this example, define **Player ID** as the partition key and **Total Score** as an alternate sort key.

- While creating a local secondary index, the **sort key value** of the base table is always projected on to the local secondary index, but it is not part of the index key.

- After defining an alternate sort key, you need to define the attributes that need to be copied or imported from the base table to the index being created. DynamoDB then copies the defined attributes to the global secondary index. If you can recall, in our video, we defined **Player Name** as an attribute, which got imported along with the partition key and alternate sort key from the base table.

Table 7: Local Secondary Index

Base Table Partition Key	Base Table Sort Key	Alternate Sort Key	Imported Attribute
Player ID	Match Series	Total Score	Player Name
101	India VS South Africa	450	Virat Kohli
102	India VS South Africa	287	Rohit Sharma
103	India VS South Africa	289	A. Rahane
101	India VS Australia	360	Virat Kohli
102	India VS Australia	345	Rohit Sharma
103	India VS Australia	328	A. Rahane
101	India VS Sri Lanka	270	Virat Kohli
103	India VS Sri Lanka	221	A. Rahane

The differences between global secondary index and local secondary index are as follows:

Global Secondary Index	**Local Secondary Index**
1. A global secondary index can have different partitions and sort keys from those of the parent table or base table.	A local secondary index must have the same partition key as that of the base table, but it can have a different sort key from that of the base table.
2. The index partition key or the sort key (if present) can be an attribute from the base table, of the type string, number, or binary.	The partition key of the index is the same attribute as that of the base table. The sort key can be any base table attribute of the type string, number, or binary.

3. A global secondary index lets you query the entire table, across all partitions.	A local secondary index lets you query a single partition, as specified by the partition key value in the query.
4. A global secondary index can be added to an existing table.	A local secondary index always needs to be defined when creating the table.
5. A global secondary index has its own provisioned read-and-write throughput values.	A local secondary index has the same provisioned throughput values as those of the base table.
6. For each partition key value in a global secondary index, there is no size restriction.	For each partition key value in a local secondary index, the total size of all the indexed items should be 10 GB or less.