Lab 6

Using DynamoDB with EC2 (Python based)

We have set up an EC2 instance and learnt how to write and execute Python code on our instance. In this lab, we will learn to write Python code to connect to a DynamoDB instance on AWS. We will learn how to create tables, add data into these tables, and query these tables, etc. We will write our code in several python scripts residing on the EC2 instance.

In the overall setup, your TCP client (on your local machine) can accept data from your IoT devices and relay it to the TCP server on EC2. The server in turn can invoke methods in the dbmanager to store this data into DynamoDB. Similarly, data may be retrieved from DynamoDB.

The python codes used is this guide are available here.

1. Downloading and installing boto3: the AWS SDK for Python

Boto3 is the AWS SDK for Python. It makes it easy to integrate your Python application, library, or script with AWS services including Amazon S3, Amazon EC2, Amazon DynamoDB, and more. In our case, we will use Boto3 specifically for DynamoDB.

The following two commands should do the job:

$ sudo apt-get update
$ sudo apt-get -y install python3-boto3

Here are some alternative methods.

2. Creating a DynamoDB database

In this section, we'll carry out various operations on a DynamoDB instance. These include table creation, loading sample data, CRUD operations (create, read, update and delete an item), query and scan data, and delete table. The examples below have been taken from AWS's DynamoDB Developer Guide. Note that our focus here is to learn how to carry out these operations from within Python code. This is by no means the only way to access DynamoDB, and you may explore other ways if necessary, such as using JavaScript.

2.1 Create a table

Read the code below for creating a table called Movies. This is available in MoviesCreateTable.py

Creating a table in DynamoDB: an example in Python

```
import boto3

def create_movie_table(dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.create_table(
        TableName='Movies',
        KeySchema=[
            {
                'AttributeName': 'year',
                'KeyType': 'HASH'  # Partition key
            },
            {
                'AttributeName': 'title',
                'KeyType': 'RANGE'  # Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'year',
                'AttributeType': 'N'
            },
            {
                'AttributeName': 'title',
                'AttributeType': 'S'
            },

        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 10,
            'WriteCapacityUnits': 10
        }
    )
    return table

if __name__ == '__main__':
    movie_table = create_movie_table()
    print("Table status:", movie_table.table_status)
```

The table is created in the usual way, as described in class. The line dynamodb = boto3.resource('dynamodb', region_name='us-east-1') provides an object-oriented interface to the DynamoDB service. As you can imagine, we can use the resource method to get interfaces to other AWS services as well. Here is more on boto3 resources and how to access them.

The region_name parameter in resource points to the region containing your EC2 instance and impacts the locations of partitions that will be assigned to DynamoDB items. You can read more on AWS regions and zones here.

In create_table we have a composite primary key, specified by both a Partition Key and a Sort Key. This means that we can insert items which have the same partition key but different sort keys, and these items will be sorted on the same partition in ascending order of the Sort Key.

Letters ‘N’ and ‘S’ stand for numeric and string types respectively. For more data types see here.

Provisioned throughput is the upper-bound on the number of per-second reads and writes your application is allowed to make on this table. Any more read/writes per-second will be throttled. This helps DynamoDB price its services when used outside the free tier. You can read more about provisioned throughput here.

Now we can run the script to create the table:

$ python3 MoviesCreateTable.py

In order to see all existing tables, you can run the following python statement. It is perhaps more convenient to run this statement on the python command prompt rather than adding it to the script.

$ python3
>>> import boto3
>>> db = boto3.resource('dynamodb', region_name='us-east-1')
>>> print(list(db.tables.all()))
>>>exit()

NOTE
Try running the script one more time. You’ll get an error stating that the table ‘Movies’ already exists. If you have multiple python functions, and multiple python scripts running on your EC2 instance, each accessing DynamoDB with resource(‘dynamodb’), they are all accessing the same instance of the database. In fact, your DynamoDB instance is also shared across your EC2 instances. If you created another EC2 instance, assigned it the IAM role with access to DynamoDB, and tried to create a ‘Movies’ table you would get the same error again.

2.2 Load sample data

Often it would be desirable to load a bunch of items into an existing table at once. This can be done by providing the items to boto3 in JSON format.

In this example, the data consists of a few thousand movies from IMDB. Following is the JSON format used:

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

```
    ...
]
```

The year and title part are the primary key we specified while defining the table Movies. info consists of further information about the movies in JSON formation. Following is an example of what a single data item might look like:

```
{
    "year" : 2013,
    "title" : "Turn It Down, Or Else!",
    "info" : {
        "directors" : [
            "Alice Smith",
            "Bob Jones"
        ],
        "release_date" : "2013-01-18T00:00:00Z",
        "rating" : 6.2,
        "genres" : [
            "Comedy",
            "Drama"
        ],
        "image_url" : "http://ia.media-
imdb.com/images/N/O9ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@@._V1_SX400_.jpg",
        "plot" : "A rock band plays their music at high volumes, annoying the neighbors.",
        "rank" : 11,
        "running_time_secs" : 5215,
        "actors" : [
            "David Matthewman",
            "Ann Thomas",
            "Jonathan G. Neff"
       ]
    }
}
```

Read more on the JSON format here.

1. Download the sample data archive: moviedata.zip
2. Extract the data file (moviedata.json) from the archive.
3. Move the moviedata.json file to your EC2 instance using FileZilla.

Read the code below. This is available in MoviesLoadData.py

```
from decimal import Decimal

import json

import boto3

def load_movies(movies, dynamodb=None):
```

```
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
    table = dynamodb.Table('Movies')
    for movie in movies:
        year = int(movie['year'])
        title = movie['title']
        print("Adding movie:", year, title)
        table.put_item(Item=movie)
if __name__ == '__main__':
    with open("moviedata.json") as json_file:
        movie_list = json.load(json_file, parse_float=Decimal)
    load_movies(movie_list)
```

Execute the script on the EC2 instance:

$ python3 MoviesLoadData.py

NOTE: JSON is a good format in which to transmit data from a local computer to the server. The server can create a new record from the received JSON and insert it into a table. See below for how new items are created.

2.3 Crud operations

Now the basic examples of creating, reading, updating and deleting items.

2.3.1 Create a new item

You can add a new data item into an existing table by providing the primary key and the associated data.

Read the code below. This is available in MoviesItemOps01.py

```
from pprint import pprint
import boto3

def put_movie(title, year, plot, rating, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
    table = dynamodb.Table('Movies')
    response = table.put_item(
       Item={
```

```
            'year': year,
            'title': title,
            'info': {
                'plot': plot,
                'rating': rating
            }
        }
    )
    return response

if __name__ == '__main__':
    movie_resp = put_movie("The Big New Movie", 2015,
                           "Nothing happens at all.", 0)
    print("Put movie succeeded:")
    pprint(movie_resp, sort_dicts=False)
```

Run the script to create the item:

$ python3 MoviesItemOps01.py

2.3.2 Reading an item

The previous script added the following item into the Movies table.

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

An item can be read from its table using its primary key value. The method used to accomplish this task is get_item. Read the code below to see how to do this. This code is available in MoviesItemOps02.py.

```
from pprint import pprint
import boto3
from botocore.exceptions import ClientError


def get_movie(title, year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.get_item(Key={'year': year, 'title': title})
    except ClientError as e:
        print(e.response['Error']['Message'])
    else:
        return response['Item']
```

```
if __name__ == '__main__':
    movie = get_movie("The Big New Movie", 2015,)
    if movie:
        print("Get movie succeeded:")
        pprint(movie, sort_dicts=False)
```

Note the use of exception handling. If the item does not exist, a proper error message is displayed. This helps in debugging the code.

Run the script:

$ python3 MoviesItemOps02.py

2.3.3 Update an item

An existing method can be updated using update_item. Once again, the primary key needs to be supplied. With update_item you can: modify the values of existing attributes in item, add new attributes to the item, or remove attributes from the item.

The following table shows the existing item and its required updated version.

```
#existing item
{
   year: 2015,
   title: "The Big New Movie",
   info: {
        plot: "Nothing happens at all.",
        rating: 0
   }
}
#required updated version
{
   year: 2015,
   title: "The Big New Movie",
   info: {
           plot: "Everything happens all at once.",
           rating: 5.5,
           actors: ["Larry", "Moe", "Curly"]
   }
}
```

We have modified the values of plot and rating. We have added a new list attribute, *actors*.

Read the code below which accomplishes this task. This code is available in MoviesItemOps03.py

```
from decimal import Decimal
from pprint import pprint
import boto3


def update_movie(title, year, rating, plot, actors, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
```

```
    table = dynamodb.Table('Movies')

    response = table.update_item(
        Key={
            'year': year,
            'title': title
        },
        UpdateExpression="set info.rating=:r, info.plot=:p, info.actors=:a",
        ExpressionAttributeValues={
            ':r': Decimal(rating),
            ':p': plot,
            ':a': actors
        },
        ReturnValues="UPDATED_NEW"
    )
    return response


if __name__ == '__main__':
    update_response = update_movie(
        "The Big New Movie", 2015, 5.5, "Everything happens all at once.",
        ["Larry", "Moe", "Curly"])
    print("Update movie succeeded:")
    pprint(update_response)
```

The important part of the code is the parameter UpdateExpression. The content assigned to UpdateExpression *describes* the changes we wish to see in the specified item. As you can see, there is a specific format to the way we describe these changes. In short, we first write the expression, which includes some place holders (expression attributes), such as :r, :p and :a in this case. In the ExpressionAttributeValues parameter, we specify the values of these place holders. Read more here about this format and the UpdateExpression parameter.

In addition to UpdateExpression, a ConditionExpression may be used to specify a condition that must be satisfied for the update to take place.

Run the script:

$ python3 MoviesItemOps03.py

2.3.4 Delete an Item

We use delete_item to delete an item using its primary key. In the following example, you try to delete a specific movie item if its rating is 5 or less. Notice the use of ConditionExpression in this case. We don't always have to use ConditionExpression, but it's useful in many cases.

```
from decimal import Decimal
from pprint import pprint
import boto3
from botocore.exceptions import ClientError

def delete_underrated_movie(title, year, rating, dynamodb=None):
    if not dynamodb:
```

```
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    try:
        response = table.delete_item(
            Key={
                'year': year,
                'title': title
            },
            ConditionExpression="info.rating <= :val",
            ExpressionAttributeValues={
                ":val": Decimal(rating)
            }
        )
    except ClientError as e:
        if e.response['Error']['Code'] == "ConditionalCheckFailedException":
            print(e.response['Error']['Message'])
        else:
            raise
    else:
        return response


if __name__ == '__main__':
    print("Attempting a conditional delete...")
    delete_response = delete_underrated_movie("The Big New Movie", 2015, 10)
    if delete_response:
        print("Delete movie succeeded:")
        pprint(delete_response)
```

Run the script:

$ python3 MoviesItemOps04.py

2.5 Query the data

To query data from a table, we use the **query** method. We must supply the partition key to the query method. The sort key is optional. For example, only supplying the year will return all movies of that year. Supplying both the year and the title will retrieve that specific movie. Further conditions can be applied to create various kinds of queries.

Let's look at a couple of examples:

Example 1 Get all movies released in a year

Read the following code which accomplishes this task. This is available in the script MoviesQuery01.py.

```
import boto3
from boto3.dynamodb.conditions import Key

def query_movies(year, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")
```

```
    table = dynamodb.Table('Movies')
    response = table.query(
        KeyConditionExpression=Key('year').eq(year)
    )
    return response['Items']


if __name__ == '__main__':
    query_year = 1985
    print(f"Movies from {query_year}")
    movies = query_movies(query_year)
    for movie in movies:
        print(movie['year'], ":", movie['title'])
```

Notice the use of the parameter KeyConditionExpression instead of ConditionExpression. Also notice the use of function Key. The Boto 3 SDK automatically constructs a ConditionExpression for you when you use the Key (or Attr) function imported from boto3.dynamodb.conditions. The Attr function will be useful when applying conditions on attributes rather than keys. You can also specify a ConditionExpression directly as a string instead, as done previously.

Run the script:

$ python3 MoviesQuery01.py

Example 2 Get all movies released in a year with certain titles

Read the following code which accomplishes this task. This is available in the script MoviesQuery02.py.

```
from pprint import pprint
import boto3
from boto3.dynamodb.conditions import Key

def query_and_project_movies(year, title_range, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    print(f"Get year, title, genres, and lead actor")

    # Expression attribute names can only reference items in the projection expression.
    response = table.query(
        ProjectionExpression="#yr, title, info.genres, info.actors[0]",
        ExpressionAttributeNames={"#yr": "year"},
        KeyConditionExpression=
            Key('year').eq(year) & Key('title').between(title_range[0], title_range[1])
    )
    return response['Items']

if __name__ == '__main__':
    query_year = 1992
    query_range = ('A', 'L')
    print(f"Get movies from {query_year} with titles from "
          f"{query_range[0]} to {query_range[1]}")
    movies = query_and_project_movies(query_year, query_range)
    for movie in movies:
        print(f"\n{movie['year']} : {movie['title']}")
```

```
        pprint(movie['info'])
```

The query prints movies from 1992, whose titles begin with letters A – L.

Run the script:

$ python3 MoviesQuery02.py

Read the output carefully and compare it with the string given to the ProjectionExpression parameter.

Now open MoviesQuery02.py and remove the first two parameters:

```
ProjectionExpression="#yr, title, info.genres, info.actors[0]",
ExpressionAttributeNames={"#yr": "year"},
```

Re-run the script and inspect the output again. What has changed?

2.6 Scan the data

The query method has the limitation that it must be supplied with a specific partition key value. However, what if we wish to write a query that needs to fetch data from multiple partitions? For instance: Print all movies between (and including) the years 1950 and 1959.

For such queries, we use the scan method. It reads all the items of a table, from all the partitions, then applies a filter_expression, which you provide, and returns only the elements matching your criteria.

The following data implements the query, 'Print all movies between (and including) the years 1950 and 1959.' The code is available in MoviesScan.py.

```
from pprint import pprint
import boto3
from boto3.dynamodb.conditions import Key

def scan_movies(year_range, display_movies, dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

    table = dynamodb.Table('Movies')

    #scan and get the first page of results
    response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]));
    data = response['Items']
    display_movies(data)

    #continue while there are more pages of results
    while 'LastEvaluatedKey' in response:
        response = table.scan(FilterExpression=Key('year').between(year_range[0],
year_range[1]), ExclusiveStartKey=response['LastEvaluatedKey'])
        data.extend(response['Items'])
```

```
        display_movies(data)

    return data

if __name__ == '__main__':
    def print_movies(movies):
        for movie in movies:
            #print(f"\n{movie['year']} : {movie['title']}")
            #pprint(movie['info'])
            pprint(movie)

    query_range = (1950, 1959)
    print(f"Scanning for movies released from {query_range[0]} to {query_range[1]}...")
    scan_movies(query_range, print_movies)
```

Run the script:

$ python3 MoviesScan.py

There are a few important things to understand in this code. First is the FilterExpression parameter. Here the Key function has been used in the same way it was used earlier to create KeyConditionExpression.

The other thing to note is how we retrieve the results of the scan function. This has to do with the concept of pagination in DynamoDB. DynamoDB applies and returns the results of queries and scans one page at a time where a page can contain at max 1MB of data. Therefore, the first call to scan returns the first page of results. The while loop then continues to re-call the scan function as along as there are more pages to filter results from. The response object maintains an attribute LastEvaluatedKey which is not None if there are more pages following the current one. The scanning process always begins from the key pointed to by the ExclusiveStartKey parameter. The expression `ExclusiveStartKey=response['LastEvaluatedKey']` points the ExclusiveStartKey to the top of the next page, and so on.

There is a paginator module in Boto3 which provides abstractions for doing pagination in different scenarios.

For more information on queries and scans, visit the links below:

Working with Queries in DynamoDB

Working with Scans in DynamoDB

2.7 Delete table

Read the following code which deletes a table from the database. This code is available in MoviesDeleteTable.py

```
import boto3

def delete_movie_table(dynamodb=None):
    if not dynamodb:
        dynamodb = boto3.resource('dynamodb', region_name="us-east-1")

    table = dynamodb.Table('Movies')
    table.delete()


if __name__ == '__main__':
    delete_movie_table()
    print("Movies table deleted.")
```

Run the script:

$ python3 MoviesDeleteTable.py

2.8 (Optional) Practice Exercises

Recreate the table Movies (if you have deleted it), and load the data into it.

$ python3 MoviesCreateTable.py
$ python3 MoviesLoadData.py

Write python scripts to perform the following operations on the movies database:

1. Print the titles of all movies released in 1994.
2. Print complete information on the movie 'After Hours' released in 1985.
3. Print all movies released before 2000.
4. Print only the years and titles of movies starring Tom Hanks.
5. Remove all movies released before 2000.

Sources

1. Amazon Dynamo DB Developer Guide for Python
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Python.html