Amazon DynamoDB Scans, Queries, and Indexes

**SPL-TF-200-DBDYL2-10-EN - Version 1.0.4**

© 2024 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited. All trademarks are the property of their owners.

Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at [*AWS Training and Certification*](https://support.aws.amazon.com/#/contacts/aws-training).

**Lab overview**

AnyCompany is developing a new movie database application. The application will allow users to search for movies by title, year they were released, actors in the movie, rating, and so on. AnyCompany chose Amazon DynamoDB as the database service to store the information about all of the movies that the application will reference.

Amazon DynamoDB is a serverless NoSQL, also known as nonrelational, database that supports key-value and document data models. With DynamoDB, you can start small and quickly scale globally as your application and user base grows, with the ability to support petabytes of data and tens of millions of read and write requests per second. AWS manages the DynamoDB service, so there are no servers to update or maintain. For more information, refer to *Amazon DynamoDB Features* and *Core Components of Amazon DynamoDB* in the **Additional resources** section.

As a newly hired junior database engineer, you are tasked with creating the appropriate Local Secondary Indexes (LSIs) and/or Global Secondary Indexes (GSIs) to improve the search functionality of your application. For this lab, you will work with a sample database named *movies* . Remember that DynamoDB tables are incredibly flexible, and do not have a traditional “schema” like relational databases. This movies table has a partition (HASH) key of “year” (Number), and a sort (RANGE) key of “title” (String). The full list of attributes populating the database for this lab match these characteristics:

* year – Number (Partition or HASH key)
* title – String (Sort or RANGE key)
* actors – String
* directors – String
* genre – String
* image\_url – String
* plot – String
* rank – Number
* rating – Number
* release date – String
* running\_time\_secs – Number

Note that in practice, the data for the actors and directors attributes has been formatted so as to be readable as python *dict* type, and therefore represent more than one actual performer. For example, the *actors* attribute for a given item is a string such as [{“S”: “John Doe”}, {“S”: “Jane Doe”}].

TOPICS COVERED

By the end of this lab, you will be able to:

* Use the SDK to create Local Secondary Indexes (LSIs) and Global Secondary Indexes (GSIs)
* Use the SDK to scan your data and return filtered results
* Use the SDK to perform advanced queries to fetch data, noting performance differences between methods

TECHNICAL KNOWLEDGE PREREQUISITES

To successfully complete this lab, you should be familiar with basic navigation of the AWS Management Console, NoSQL database concepts, the Python programming language, and be comfortable editing scripts using a text editor.

ICON KEY

Various icons are used throughout this lab to call attention to certain aspects of the guide. The following list explains the purpose for each one:

* The keyboard icon specifies that you must run a command.
* The clipboard icon indicates that you can verify the output of a command or edited file by comparing it to the provided example.
* The note icon specifies important hints, tips, guidance, or advice.
* The “i” circle icon specifies where to find more information.
* The person with a check mark icon indicates an opportunity to check your knowledge and test what you have learned.

**Start lab**

1. To launch the lab, at the top of the page, choose **Start lab**.

**Caution:** You must wait for the provisioned AWS services to be ready before you can continue.

1. To open the lab, choose **Open Console**.

You are automatically signed in to the AWS Management Console in a new web browser tab.

**WARNING:** **Do not change the Region unless instructed.**

COMMON SIGN-IN ERRORS

**Error: You must first sign out**



If you see the message, **You must first log out before logging into a different AWS account:**

* Choose the **click here** link.
* Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
* Choose **Open Console** again.

**Error: Choosing Start Lab has no effect**

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

* Add the lab domain name to your pop-up or script blocker’s allow list or turn it off.
* Refresh the page and try again.

**Task 1: Prepare the environment**

The AWS SDK for Python (Boto3) is the method you will use to interact with DynamoDB. You will install the SDK and then verify the status of the table.

TASK 1.1: CONNECT TO THE AWS CLOUD9 IDE AND INSTALL THE AWS SDK FOR PYTHON (BOTO3)

In this task, you install the AWS SDK for Python (Boto3). You then use Python scripts to access the table and verify that it is populated with some sample data.

1. In the AWS Management Console, on the **Services** menu, choose **Cloud9**.
2. On the **Your environments** page, on the **Lab\_Cloud9\_IDE** card, choose **Open IDE**

The AWS Cloud9 environment opens in a new browser tab.

1. In the left folder tree, choose the “lab\_folder” folder. This should ensure that the scripts you will save throughout this lab are saved in that location.
2. To the right of the **Welcome** tab, choose the plus  icon, and then select **New Terminal**.
3. In the AWS Cloud9 terminal, to install Boto3, the AWS SDK for Python, run the following command:

pip install boto3

 For more information, refer to *AWS SDK for Python (Boto3)* in the **Additional resources** section.

1. In the AWS Cloud9 terminal, to operate in the lab folder, run the following command:

cd lab\_folder

TASK 1.2: VERIFY CONNECTION TO DYNAMODB

Your environment should already have a sample DynamoDB table named “movies” for you to access.

1. To verify access to the database, run the following command:

python CheckTableStatus.py movies

The status should show as ACTIVE.

**Task 2: Fetch data from a standard table**

The simplest way to request data from the database is a scan, which will view and return all items (up to 1MB at a time), or optionally those that meet your filter criteria. A query, however, is a much more efficient way to request data, since it limits the number of entries that are read. In this task, you will create a scan and a query to compare the operations.

TASK 2.1: SCAN THE DATABASE FOR A SPECIFIC YEAR AND GENRE

Say that you want to find all movies from a specific year that were labeled as comedies. Here you will build a script to scan for that information.

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to scan and filter items:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

from decimal import \*

def scan\_movies(YearToFind,GenreToFind):

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

recordcount = 0

recordscannedcount = 0

scanreturn = table.scan( # perform first scan

FilterExpression=Key('year').eq(YearToFind) & Attr("genre").eq(GenreToFind)

)

recordcount += scanreturn['Count']

recordscannedcount += scanreturn['ScannedCount']

while 'LastEvaluatedKey' in scanreturn.keys(): # if lastevaluatedkey is present, we need to keep scanning and adding to our counts until everything is scanned

scanreturn = table.scan(

FilterExpression=Key('year').eq(YearToFind) & Attr("genre").eq(GenreToFind),

ExclusiveStartKey = scanreturn['LastEvaluatedKey']

)

recordcount += scanreturn['Count']

recordscannedcount += scanreturn['ScannedCount']

return [recordcount, recordscannedcount]

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument("Qyear", help="Search by year and genre.. will return number of movies with that year and genre")

parser.add\_argument("Qgenre", help="Search by year and genre.. will return number of movies with that year and genre")

args = parser.parse\_args()

queryyear = Decimal(args.Qyear)

querygenre = (args.Qgenre) #section to collect argument from command line

start = time.time()

movies = scan\_movies(queryyear, querygenre) #scan\_movies returns our total counts as two items of a list

end = time.time()

print("Count is ", movies[0]) # print the count of items returned by the scan

print("ScannedCount is ", movies[1]) # print the count of items that had to be scanned to process the scan

print('Total time: {} sec'.format(end - start))

 Note that since this script scans all items and then filters them, it will likely reach the 1MB limit for data transactions on a scan operation. Therefore, the script has an additional loop to use the LastEvaluatedKey and ExclusiveStartKey to work through the entire dataset.

 For more information, refer to *Scan Operation* in the **Additional resources** section.

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

MoviesScanYG.py

1. Close the **MoviesScanYG.py** tab.
2. To run the **MoviesScanYG.py** script and view the total, run the following command:

python MoviesScanYG.py 1980 Comedy

1. Record the values returned by this command.

The output should list a number of items found that matched the criteria, as well as a count of all the items scanned and the time consumed by the search. The count of items scanned should equal the total number of items in the database.

Note that this simple scan requires inefficiently scanning the entire database to find the entries that matched the filter. A query operation that only reads a particular partition (in this case, a particular year) will be much more efficient.

TASK 2.2 QUERY THE DATABASE FOR A SPECIFIC YEAR AND GENRE

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to query and filter items:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

from decimal import \*

def scan\_movies(YearToFind,GenreToFind):

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

scanreturn = table.query( # perform query

# IndexName="genre-index",

KeyConditionExpression=Key('year').eq(YearToFind),

FilterExpression=Attr("genre").eq(GenreToFind)

)

return scanreturn

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument("Qyear", help="Search by year and genre.. will return number of movies with that year and genre")

parser.add\_argument("Qgenre", help="Search by year and genre.. will return number of movies with that year and genre")

args = parser.parse\_args()

queryyear = Decimal(args.Qyear)

querygenre = (args.Qgenre) #section to collect argument from command line

start = time.time()

movies = scan\_movies(queryyear, querygenre) #scan\_movies returns dict, which is dict of each individual database item returned by scan

end = time.time()

print("Count is ", movies['Count']) # print the count of items returned by the query

print("ScannedCount is ", movies['ScannedCount']) # print the count of items that had to be scanned to process the query

print('Total time: {} sec'.format(end - start))

 None of the partitions in this test database are larger than 1MB. Therefore, this script and others in this lab that use the query function do not check for the LastEvaluatedKey status, since the query only reads the data from a particular partition. Remember that in larger datasets, you may hit the 1MB limit even on queries, not just scans. In that case, you would need to check for the LastEvaluatedKey.

 For more information, refer to *Query Operation* in the **Additional resources** section.

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

MoviesQueryYG.py

1. Close the **MoviesQueryYG.py** tab.
2. To run the **MoviesQueryYG.py** script and view the total, run the following command:

python MoviesQueryYG.py 1980 Comedy

1. Record the values returned by this command.

Using a query provides a significant efficiency advantage over doing a scan. Compare the results of the query vs the scan. The count of matches should be equal, but the query had to scan fewer items and therefore took less time. But if you wanted to repeatedly run a query on the genre attribute, there is an even more efficient method. What if we didn’t have to filter by genre, but could query it as a key value? We cannot utilize the query operation to use the genre attribute as a primary criteria the way the table is currently configured, because the movies table has a partition (HASH) key of year, with the title as the sort key. To use the query operation in the manner we want, we need to create a Local Secondary Index (LSI) that includes the genre attribute as a sort key.

TASK 2.3 DELETE THE EXISTING MOVIES TABLE

A Local Secondary Index (LSI) cannot be added to an existing table, so the existing one will need to be removed. Note that this is a critical distinction for Local Secondary Indexes. They can only be created simultaneously with a new table, so it is imperative that the use case for creating one is determined before the table itself is created.

1. To delete the existing “movies” table, run the following command:

python DeleteMoviesTable.py

 The output should show a description of the table with a status of ‘DELETING’, similar to the following:

{'TableDescription': {'TableName': 'movies', 'TableStatus': 'DELETING', 'ProvisionedThroughput': {'NumberOfDecreasesToday': 0, 'ReadCapacityUnits': 1000, 'WriteCapacityUnits': 1000}, 'TableSizeBytes': 0, 'ItemCount': 0, 'TableArn': 'arn:aws:dynamodb:us-east-1:389298664099:table/movies', 'TableId': 'cba78a67-d1e1-4aa8-952f-a8b48f84853e'}, 'ResponseMetadata': {'RequestId': 'A0D5EE41OQMAT7U7L574QQ8SN7VV4KQNSO5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Tue, 26 Jan 2021 16:12:50 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '320', 'connection': 'keep-alive', 'x-amzn-requestid': 'A0D5EE41OQMAT7U7L574QQ8SN7VV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '2041919442'}, 'RetryAttempts': 0}}

**Task 3: Create a table with a Local Secondary Index (LSI)**

Now you will create a new movies table that includes a Local Secondary Index (LSI) based on the genre attribute. This will allow you to use the query operation for the genre attribute and compare the performance to the previous methods.

TASK 3.1 CREATE THE MOVIES TABLE WITH THE GENRE LOCAL SECONDARY INDEX (LSI)

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to create a new DynamoDB table:

import boto3

def create\_movie\_table():

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level client

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'

},

{

'AttributeName': 'genre',

'AttributeType': 'S'

},

],

LocalSecondaryIndexes=[

{

'IndexName': 'genre-index',

'KeySchema': [

{

'AttributeName': 'year',

'KeyType': 'HASH'

},

{

'AttributeName': 'genre',

'KeyType': 'RANGE'

},

],

'Projection': {

'ProjectionType': 'ALL',

}

}

],

ProvisionedThroughput={

'ReadCapacityUnits': 1000,

'WriteCapacityUnits': 1000

}

)

return table

if \_\_name\_\_ == '\_\_main\_\_':

movie\_table = create\_movie\_table()

print("Table status:", movie\_table.table\_status)

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

CreateMovieswithGenreLSI.py

1. Close the **CreateMovieswithGenreLSI.py** tab.
2. To run the **CreateMovieswithGenreLSI.py** script to create the new table, run the following command:

python CreateMovieswithGenreLSI.py

1. To verify that the table was successfully created, run the following command:

python CheckTableStatus.py movies

TASK 3.2 POPULATE THE NEW DATABASE WITH MOVIE INFORMATION

Since the previous table was deleted, all the information it contained was also removed. The new database will need to be populated with data.

1. To populate the new movies table with movie information, run the following command:

python PopulateDatabase.py moviedata.json

**Note:** This is uploading a lot of data. It usually takes a couple of minutes to complete the operation. When complete, the script will display a total duration for the operation.

**Task 4: Fetch data from a table with a Local Secondary Index (LSI)**

Now that the table has a Local Secondary Index using the genre attribute, we can use the query operation. In this task, you will do a query-only operation and compare the results to previous examples.

TASK 4.1 SCAN FOR DATA AS A BASELINE

1. To run the **MoviesScanYG.py** script and view the total, run the following command:

python MoviesScanYG.py 1980 Comedy

1. Record the values returned by this command (they should be similar to the values seen when running this command in a previous task)

TASK 4.2 QUERY DIRECTLY FOR DATA AND COMPARE TO THE PREVIOUS OPERATIONS

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to query the table for all movies from a specific year with a specific genre:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

from decimal import \*

def scan\_movies(YearToFind,GenreToFind):

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

scanreturn = table.query( # perform query

IndexName="genre-index",

KeyConditionExpression=Key('year').eq(YearToFind) & Key('genre').eq(GenreToFind)

)

return scanreturn

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument("Qyear", help="Search by year and genre.. will return number of movies with that year and genre")

parser.add\_argument("Qgenre", help="Search by year and genre.. will return number of movies with that year and genre")

args = parser.parse\_args()

queryyear = Decimal(args.Qyear)

querygenre = (args.Qgenre) #section to collect argument from command line

start = time.time()

movies = scan\_movies(queryyear, querygenre) #scan\_movies returns dict, which is dict of each individual database item returned by scan

end = time.time()

print("Count is ", movies['Count']) # print the count of items returned by the query

print("ScannedCount is ", movies['ScannedCount']) # print the count of items that had to be scanned to process the query

print('Total time: {} sec'.format(end - start))

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

MoviesQueryWithLSIYG.py

1. Close the **MoviesQueryWithLSIYG.py** tab.
2. To run the **MoviesQueryWithLSIYG.py** script and view the total, run the following command:

python MoviesQueryWithLSIYG.py 1980 Comedy

1. Record the values returned by this command.
2. Compare the results from the Query to the results from the Scan

The query, being able to efficiently leverage the Local Secondary Index, should return the same count of movies matching the year and genre, but should only have needed to scan those exact items rather than the entire database. Therefore, it should have completed more quickly than the scan operation, and even the previous query that only leveraged the year key. Note that while the difference may not be particularly significant for a single search, it quickly adds up when considering thousands of searches that may be performed on a database. Also remember that all the “extra” read operations required by the scan and filter operations affect the capacity and performance of the database handling other requests, and can consume additional Read Capacity Units (RCUs).

 Just to verify that the performance difference applies across different searches, run the **MoviesScanYG.py**, **MoviesQueryYG.py**, and **MoviesQueryWithLSIYG.py** scripts with some different years and genres, such as 1945 and Crime, or 1994 and Drama

**Hint:** The Scan should take about the same time no matter what the parameters are, since it scans all items anyway. The query should consistently be quicker, even for parameters that return more results. The query with the LSI should be best.

**Task 5: Create a Global Secondary Index (GSI)**

A Local Secondary Index (LSI) creates the opportunity to use the query function for additional attributes, but still requires leveraging the primary HASH key of the table. What if you wanted to query for attributes without including the primary key? A Global Secondary Index (GSI) allows this, and can be created on an existing DynamoDB table. In this task, you will create a GSI for the genre attribute, compare the performance to a traditional scan, and consider how using a GSI can improve database performance.

TASK 5.1 CREATE A GLOBAL SECONDARY INDEX (GSI) USING THE GENRE ATTRIBUTE

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to create a new Global Secondary Index (GSI) for the genre attribute:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

from decimal import \*

def MakeGSI():

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

response = table.update(

AttributeDefinitions=[

{

"AttributeName": "genre",

"AttributeType": "S"

},

],

GlobalSecondaryIndexUpdates=[

{

'Create': {

'IndexName': "genre-globo-index",

'KeySchema': [

{

'AttributeName': "genre",

'KeyType': 'HASH'

}

],

'Projection': {

'ProjectionType': 'ALL'

},

'ProvisionedThroughput': {

'ReadCapacityUnits': 1000,

'WriteCapacityUnits': 1000

}

}

}

],

)

table.reload() # this section to reload the table status every 10 seconds until the GSI finishes creating

tmpreply = table.global\_secondary\_indexes

indexnum = 0;

while tmpreply[indexnum]['IndexName'] != "genre-globo-index" : # in case other indexes exist for this table, find the one we just created

indexnum += 1

while tmpreply[indexnum]['IndexStatus'] != 'ACTIVE': # check to see if the new one is active yet, and if not... repeat until it is

time.sleep(10)

print("Still creating...")

table.reload()

tmpreply = table.global\_secondary\_indexes

return response

if \_\_name\_\_ == '\_\_main\_\_':

start = time.time()

result = MakeGSI()

end = time.time()

print('Total time: {} sec'.format(end - start))

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

CreateGenreGSI.py

1. Close the **CreateGenreGSI.py** tab.
2. To run the **CreateGenreGSI.py** script and view the total, run the following command:

python CreateGenreGSI.py

**Note:** Unlike creating the LSI, which is created with a new table having no data, a GSI created on an existing table must manage partitioning the existing data. This means the creation process can take a few minutes to complete, depending on the size of the database. When complete, the script will display a total duration for the operation.

TASK 5.2 QUERY USING THE GLOBAL SECONDARY INDEX (GSI)

First you need to create a query script that searches only based on genre.

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to query the movies database for all movies of a chosen genre:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

from decimal import \*

def scan\_movies(GenreToFind):

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

scanreturn = table.query( # perform first scan

IndexName="genre-globo-index", # assume the global index name was created using our CreateGSI script that named it genre-globo-index

KeyConditionExpression=Key("genre").eq(GenreToFind)

)

return scanreturn

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument("Genre", help="Search by genre, ex: Action... argument is case sensitive")

args = parser.parse\_args()

query\_direct = (args.Genre) #section to collect argument from command line

start = time.time()

movies = scan\_movies(query\_direct) #scan\_movies returns dict, which is dict of each individual database item returned by scan

end = time.time()

print("Count is ", movies['Count']) # print the count of items returned by the query

print("ScannedCount is ", movies['ScannedCount']) # print the count of items that had to be scanned to process the query

print('Total time: {} sec'.format(end - start))

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

QueryByGenre.py

1. Close the **QueryByGenre.py** tab.
2. To run the **QueryByGenre.py** script and view the total, run the following command:

python QueryByGenre.py Comedy

1. Record the values returned by this script. How do they compare to the previous scans and queries?

TASK 5.3 SCAN FOR ONLY GENRE

The previous tasks used the scan function selecting both year and genre. To get an exact comparison with the query done against the new Global Secondary Index, create a scan that filters only on genre.

1. On the **File** menu, choose **New File**.
2. In the **Untitled1** editor tab, copy and paste the following Python script, which is used to scan the movies database for all movies of a chosen genre:

from pprint import pprint

import boto3

from boto3.dynamodb.conditions import Key, Attr

import argparse

import time

def scan\_movies(dname):

region=boto3.session.Session().region\_name

dynamodb = boto3.resource('dynamodb', region\_name=region) #low-level Client

table = dynamodb.Table('movies') #define which dynamodb table to access

recordcount = 0

recordscannedcount = 0

scanreturn = table.scan( # perform first scan

FilterExpression=Attr("genre").eq(dname)

)

recordcount += scanreturn['Count']

recordscannedcount += scanreturn['ScannedCount']

while 'LastEvaluatedKey' in scanreturn.keys(): # if lastevaluatedkey is present, we need to keep scanning and adding to our counts until everything is scanned

scanreturn = table.scan(

FilterExpression=Attr("genre").eq(dname),

ExclusiveStartKey = scanreturn['LastEvaluatedKey']

)

recordcount += scanreturn['Count']

recordscannedcount += scanreturn['ScannedCount']

return [recordcount, recordscannedcount]

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser()

parser.add\_argument("Genre", help="Search by genre, ex: Action... argument is case sensitive")

args = parser.parse\_args()

query\_direct = (args.Genre) #section to collect argument from command line

start = time.time()

movies = scan\_movies(query\_direct) #scan\_movies returns our total counts as two items of a list

end = time.time()

print("Count is ", movies[0]) # print the count of items returned by the scan

print("ScannedCount is ", movies[1]) # print the count of items that had to be scanned to process the scan

print('Total time: {} sec'.format(end - start))

1. On the **File** menu, choose **Save As…** and save the file in “lab\_folder” as

ScanByGenre.py

1. Close the **ScanByGenre.py** tab.
2. To run the **ScanByGenre.py** script and view the total, run the following command:

python ScanByGenre.py Comedy

1. Record the values returned by this script. How do they compare to the previous scans and queries? It is expected that much like with the previous scans, it will return the same number of matches as the query command, but take longer since it had to scan all entries.

 Congratulations! You have successfully created both a Local and Global Secondary Index, and observed the performance advantage those indexes can provide.

**Task 6: Challenge activity**

Applying everything you have learned so far, perform the following tasks:

* Use the SDK to delete a Global Secondary Index
* Use the SDK to create a new Global Secondary Index on an attribute entered by the user when calling the script
* Use the SDK to create a query against genre that also filters for a range of ratings, for example to return a count of all comedies rating between 7 and 8

**Hints:**

* Use the [AWS CLI command reference for DynamoDB](https://awscli.amazonaws.com/v2/documentation/api/latest/reference/dynamodb/index.html) and the [Boto3 documentation](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html) to help inform your solution.
* Use the CreateGenreGSI python script as a starting point for creating a new GSI
* Lookup the FilterExpression option for queries and work with the QueryByGenre python script

 Refer to the **Answer key** section for the answers to this challenge.

**Conclusion**

 Congratulations! You now have successfully:

* Used the SDK to create Local Secondary Indexes (LSIs) and Global Secondary Indexes (GSIs)
* Used the SDK to scan and return filtered results
* Used the SDK to perform advanced queries to fetch data, noting performance differences between methods

**End lab**

Follow these steps to close the console and end your lab.

1. Return to the **AWS Management Console**.
2. At the upper-right corner of the page, choose **AWSLabsUser**, and then choose **Sign out**.
3. Choose **End lab** and then confirm that you want to end your lab.

**Additional resources**

* [Amazon DynamoDB Features](https://aws.amazon.com/dynamodb/features/)
* [Core Components of Amazon DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html)
* [Read/Write Capacity Mode](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html)
* [Provisioned Mode](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html#HowItWorks.ProvisionedThroughput.Manual)
* [Working with Scans in DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html)
* [Working with Queries in DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html)
* [Reserved Words in DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ReservedWords.html)
* [Expression Attribute Names in DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ExpressionAttributeNames.html)
* [AWS SDK for Python (Boto3)](https://aws.amazon.com/sdk-for-python/)
* [CreateTable Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_CreateTable.html)
* [PutItem Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html)
* [Scan Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html)
* [Query Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html)
* [UpdateItem Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_UpdateItem.html)
* [DeleteItem Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DeleteItem.html)
* [DeleteTable Operation](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DeleteTable.html)

For more information about AWS Training and Certification, see [*https://aws.amazon.com/training/*](https://aws.amazon.com/training/).

*Your feedback is welcome and appreciated.*  
If you would like to share any feedback, suggestions, or corrections, please provide the details in our [*AWS Training and Certification Contact Form*](https://support.aws.amazon.com/#/contacts/aws-training).

**Answer key**

TASK 6

The Python script linked [here](https://us-west-2-tcprod.s3.us-west-2.amazonaws.com/courses/SPL-TF-200-DBDYL2/v1.0.4.prod-aa34d5dc/scripts/DeleteGSI.py) can be used to delete a GSI with a name of *attribute*-globo-index for a given attribute from a DynamoDB table named *movies*.

The Python script linked [here](https://us-west-2-tcprod.s3.us-west-2.amazonaws.com/courses/SPL-TF-200-DBDYL2/v1.0.4.prod-aa34d5dc/scripts/CreateAttributeGSI.py) can be used to create a GSI on a DynamoDB table named *movies* for a given attribute and type.

The Python script linked [here](https://us-west-2-tcprod.s3.us-west-2.amazonaws.com/courses/SPL-TF-200-DBDYL2/v1.0.4.prod-aa34d5dc/scripts/QueryByGenreAddRatingRange.py) can be used to query a DynamoDB table named *movies* with a GSI named genre-globo-index and also filter for a range using the *rating* attribute

 Note that all of these scripts require the user to enter one or more parameters when performed