23. To find detailed feedback on your work, choose Details followed by **View Submission Report**.

Lab complete

Congratulations! You have completed the lab.

24. Choose End Lab at the top of this page, and then select Yes to confirm that you want to end the lab.

    A panel indicates that *DELETE has been initiated... You may close this message box now.*

25. Select the **X** in the top-right corner to close the panel.

LAB 5.1

[version_1.0.5]

Lab 5.1: Working with Amazon DynamoDB

Lab overview and objectives

In this lab, you use Amazon DynamoDB to store and manage menu information. Using databases, such as DynamoDB, simplifies data management because you can easily query, sort, edit, and index data. You will use both the AWS Command Line Interface (AWS CLI) and the AWS SDK for Python (Boto3) to work with DynamoDB.

In upcoming labs, you will use application programming interface (API) calls from the café website to dynamically retrieve and update data that's stored in a DynamoDB table.

After completing this lab, you should be able to:

- Create a new DynamoDB table
- Add data to the table
- Modify table items based on conditions
- Query the table
- Add a global secondary index to the table

When you start the lab, the following resources are already created for you in the AWS account:

- AWS Cloud9 integrated development environment (IDE) instance
- Amazon Elastic Compute Cloud (Amazon EC2) instance for authoring code and running commands through the AWS CLI

starting resources

At the end of this lab, your architecture should look like the following example:

starting resources

Duration

This lab requires approximately **90 minutes** to complete.

AWS service restrictions

In this lab environment, access to AWS services and service actions might be restricted to the ones that are needed to complete the lab instructions. You might encounter errors if you attempt to access other services or perform actions beyond those that this lab describes.

Accessing the AWS Management Console

1. At the top of these instructions, choose Start Lab to launch your lab.

   A **Start Lab** panel opens, and it displays the lab status.

   Tip: If you need more time to complete the lab, choose the **Start Lab** button again to restart the timer for the environment.

2. Wait until you see the message **Lab status: ready**, and then close the **Start Lab** panel by choosing the **X**.

3. At the top of these instructions, choose AWS

   This opens the AWS Management Console in a new browser tab. The system automatically logs you in.

   Tip: If a new browser tab does not open, a banner or icon is usually at the top of your browser with a message that your browser is preventing the website from opening pop-up windows. Choose the banner or icon, and then choose **Allow pop ups**.

4. Arrange the **AWS Management Console** tab so that it displays along side these instructions. Ideally, you will be able to see both browser tabs at the same time so that you can follow the lab steps more easily.

Scenario

The café website is up and running, and the café staff noticed a significant increase in new customer visits. Multiple customers also mentioned that it would be helpful if the website had an up-to-date menu. They could then use the menu to check the availability of food items before going to the café.

Frank and Martha ask Sofía to explore whether she can implement this feature for customers. Sofía is feeling more confident in her coding skills and has also been learning about different ways to store information in AWS. She knows that before they can dynamically update data on the website, she must first choose a data storage service to hold the data. She also needs to learn how to manage table data, load the product records, and create scripts to retrieve information from the data platform.

A business request from the café: Store menu information in the cloud

Frank and Martha mentioned to Sofía that they want the website to dynamically update its menu information. To prepare for this new functionality, Sofía decides to store this information in DynamoDB.

Café staff must be able to retrieve information from the table. Sofía decides to create one script that retrieves all inventory items from the table and another script (as a proof of concept) that uses a product name to retrieve a single record.

For this first challenge, you take on the role of Sofía. You use the AWS CLI and the SDK for Python to configure and create a DynamoDB table, load records into the table, and extract data from the table.

Task 1: Preparing the lab

Before you can start this lab, you must import some files and install some packages in the AWS Cloud9 environment that was prepared for you.

5. Connect to the AWS Cloud9 IDE.
    - From the **Services** menu, search for and select **Cloud9**. You should see an existing IDE named **Cloud9 Instance**.
    - In the **Cloud9 Instance** pane, choose **Open IDE**.

      The AWS Cloud9 IDE loads in a new browser tab.

6. Download and extract the files that you need for this lab.
    - Run the following command in the same terminal:

      ```
      xxxxxxxxxx
      wget https://aws-tc-largeobjects.s3.us-west-2.amazonaws.com/CUR-TF-200-ACCDEV-2-91558/03-lab-dynamo/code.zip -P /home/ec2-user/environment
      ```

    - You should see that the **code.zip** file was downloaded to the AWS Cloud9 instance and is now in the left navigation pane.
    - Extract the file by running the following command:

      ```
      xxxxxxxxxx
      unzip code.zip
      ```

      Tip: This action extracts the files from code.zip. In the **Environment** pane on the left, you should now see a new folder that is named **resources**. You use the files that you downloaded and extracted later in this lab.

7. Run a script that upgraded the version of Python installed on the Cloud9 instance. It will also upgrade the version of the AWS CLI installed.
    - To set permissions on the script and then run it, run the following commands:

      ```
      xxxxxxxxxx
      chmod +x ./resources/setup.sh && ./resources/setup.sh
      ```

8. Verify the AWS CLI version and also verify that the SDK for Python is installed.
    - Confirm that the AWS CLI is now at version 2 by running the `aws --version` command.
    - In the AWS Cloud9 Bash terminal (at the bottom of the IDE), run the following command:

      ```
      xxxxxxxxxx
      pip show boto3
      ```

      Note: If you see a message about not using the latest version of pip, ignore the message.

Task 2: Creating a DynamoDB table by using the SDK for Python

To store and dynamically manage the café's menu items, Sofía decides to create a new DynamoDB table.

In this task, you take on the role of Sofía to create and define the new DynamoDB table.

Initially, you create this table with only one *attribute*. Because every DynamoDB table requires a *primary key*, this attribute becomes the primary key for the table. Each value used as a primary key must be unique.

The **product_name** is the first attribute that you define in the table. The **product_name** attribute works well because the café's product names should not be duplicated. Also, the café wants to use the product names to query details about each record.

Tip: Choose your primary keys carefully. After a table has been created, you cannot change the primary key to use a different attribute or set of attributes. If you must change the primary key, you must create a new table and migrate the data from your existing table. For more details, refer to the [Amazon DynamoDB Developer Guide](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html) at https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

Like the previous labs, you use the SDK for Python so that you can use the Python script with the DynamoDB service.

9. First, verify that no tables exist in the environment by using the AWS Management Console:
   - In the top-left corner of the AWS Cloud9 IDE, choose the **AWS Cloud9** icon, and choose **Go To Your Dashboard**.

     A new tab opens in your browser.
   - In the new tab, open the DynamoDB console by choosing the **Services** menu and then choosing **DynamoDB**.

     Note: If you see a message that offers a preview of the new console at the top of the page, choose this option. Choose the try it and let us know what you think link.

     Note: When the DynamoDB console opens, close any informational messages at the top of the page by choosing the **X** (in the top-right area of each message).
   - On the left, expand the **DynamoDB** navigation pane by choosing the menu icon.
   - From the **DynamoDB** menu, choose **Tables**.
   - Review the **Tables** pane.

     starting resources

     Notice that no tables are listed.

10. Edit the script that will create the table:
    - Return to the AWS Cloud9 IDE browser tab.
    - In the navigation pane of the AWS Cloud9 IDE, expand the **python_3** directory.
    - Open the **create_table.py** script by double-clicking it.
    - Replace the *<FMI_1>* placeholder with the table name, which is:

      ```
      xxxxxxxxxx
      FoodProducts
      ```
    - In the upper left, choose **File > Save** to save your changes.

11. To understand what the script does, review the code:
    - The line that defines the **DDB** variable also configures the SDK for Python resource.
    - It sets both the AWS service and the AWS Region that the Python script will call.

      ```
      xxxxxxxxxx
      DDB = boto3.resource('dynamodb', region_name='us-east-1')
      ```
    - The following section provides the details that describe the new table. Note the values for **TableName**, **KeySchema**, and **AttributeDefinitions**. The **AttributeDefinitions** parameter contains only one value, which is **product_name**. You can add more attributes later at runtime.

      ```
      xxxxxxxxxx
      params = {
             'TableName': '<FMI_1>',
             'KeySchema': [
                 {'AttributeName': 'product_name', 'KeyType': 'HASH'}
             ],
             'AttributeDefinitions': [
                 {'AttributeName': 'product_name', 'AttributeType': 'S'}
             ],
             'ProvisionedThroughput': {
                 'ReadCapacityUnits': 1,
                 'WriteCapacityUnits': 1
             }
         }
      ```

- The line that defines the **table** variable also creates the table:

  💁 Boto requires key values arguments rather than the object literal format, so you use `**params` to pass the parameters to the **create_table** operation.

  ```
  xxxxxxxxxx
  table = DDB.create_table(**params)
  ```

12. In the AWS Cloud9 terminal window, go to the **python_3** directory, and run the following code:

    ```
    x
    cd python_3
    python3 create_table.py
    ```

    This command can take several minutes to run because in the code you are waiting for the table to be ready before exiting the function. It might look like it hangs, but be patient. Once the command completes successfully, the terminal output should show the following message:

    ```
    xxxxxxxxxx
    Done
    ```

    Once done (and not before), run the following command to make sure the table was successfully created:

    ```
    x
     aws dynamodb list-tables --region us-east-1
    ```

    The output should be similar to the following example:

    ```
    xxxxxxxxxx
    {
      "TableNames": [
          "FoodProducts"
      ]
    }
    ```

13. Return to the browser tab with the DynamoDB console. Use the refresh icon on the far right, choose FoodProducts, and verify that the table has a created state of **Active**.

    In the next few tasks, you learn how to add data to the table that you created.

Task 3: Working with DynamoDB data – Understanding DynamoDB condition expressions

Now that Sofía created the table, she wants to understand what happens when records are written to it.

In this task, you continue as Sofía to insert the first record into the table.

14. Review the JavaScript Object Notation (JSON) data that defines the new record.

    - In the AWS Cloud9 IDE, expand the **resources** folder.
    - Open the **not_an_existing_product.json** file by double-clicking it.

    Analysis: This file contains one item with two attributes: **product_name** and **product_id**. Both of these attributes are strings. The primary key (**product_name**) was defined when the DynamoDB table was created. Because DynamoDB tables are schemaless (to be exact, not bound by a fixed schema), you can add new attributes to the table when items are inserted or updated. With DynamoDB, you don't need to change the table definition before you add records that contain additional attributes.

15. To insert the new record, run the following command. Ensure that you are still in the python_3 folder.

    ```
    xxxxxxxxxx
    aws dynamodb put-item \
    --table-name FoodProducts \
    ```

starting resources

```
--item file://../resources/not_an_existing_product.json \

--region us-east-1
```

16. Verify that the new record was added to the table by using the DynamoDB console to complete the following tasks:

- Return to the DynamoDB console and choose the **FoodProducts** link.
- Choose **Explore table items**.
- Under **Items returned**, review the information.

You should find one record with two attributes: **product_name** and **product_id**.

Add a second record to the table.

17. Update the JSON data to create a new record:

- Return to the AWS Cloud9 IDE and load the **not_an_existing_product.json** file in the text editor.
- Replace the **product_name** value of *<best cake>* with `best pie`
- Do not change the **product_id** value.
- In the upper left, choose **File > Save** to save your changes.

starting resources

18. To add the new record, run the following command. Notice that this command is the same AWS CLI command that you used to add the first record.

```
x

aws dynamodb put-item \

--table-name FoodProducts \

--item file://../resources/not_an_existing_product.json \

--region us-east-1
```

19. Again, view the new record in the table by using the console:

- Return to the **Item explorer** in the DynamoDB console.
- Confirm that the **FoodProducts** table is selected.
- Choose **Scan**
- Choose **Run**
- Under **Items returned**, review the data.

💁‍♂️ Because the **product_id** attribute is not the primary key of the table, it doesn't need to be unique, and a new record is inserted successfully. If the value of **product_name** is different, a new record is created in the table.

What do you think will happen if you try to insert a duplicate record?

20. Return to the AWS Cloud9 IDE and try re-running the previous AWS CLI command (the `up` key helps here). Don't make any changes to the JSON record.

21. In the DynamoDB console, choose **Run** again and review the **Items returned** list. Do you notice any changes?

💁‍♂️ When a primary key doesn't exist in the table, the DynamoDb **put-item** command inserts a new item. However, if the primary key already exists, this command replaces the existing record with the new record, removing any previous attributes. This behavior is why you don't see a new item in the table: the record was overwritten with identical information. The primary key prevents the same **product_name** values from being added multiple times.

Now, try to insert a record with an existing primary key and a different **product_id** value.

starting resources

22. Update the JSON record:

- Return to the AWS Cloud9 IDE and the **not_an_existing_product.json** file.
- Don't change the value of **product_name**.
- Replace the **product_id** value of <*676767676767*> with `3333333333`
- In the upper left, choose **File > Save** to save your changes.

23. Run the previous AWS CLI **put-item** command again:

```
xxxxxxxxxx
aws dynamodb put-item \
--table-name FoodProducts \
--item file://../resources/not_an_existing_product.json \
--region us-east-1
```

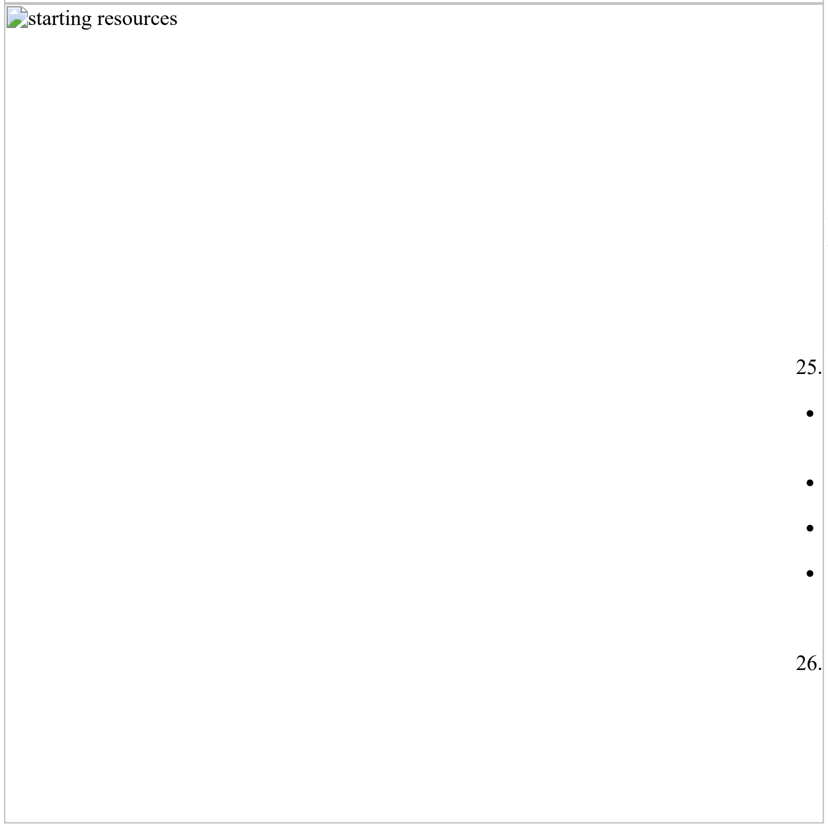
24. View the table data in the DynamoDB item explorer by choosing **Run**.

Analysis: The **product_id** value of the **best pie** record was replaced with the new value of **product_id**.

However, you don't want this behavior. You want separate operations for adding new products and for updating product attributes.

To implement this feature, you can refine the behavior of the **put-item** command with *condition expressions*. You can use condition expressions to determine which item should be modified. In this case, you must prevent records from being overwritten if they already exist in the table. The **attribute_not_exists()** function provides this capability.

Next, test the condition expression. You try to insert another version of the record for **best pie**.


starting resources

25. Update the JSON record:

- Return to the AWS Cloud9 IDE and the **not_an_existing_product.json** file.
- Don't change the value of **product_name**.
- Replace the **product_id** value of <*3333333333*> with `2222222222`
- Save your changes.

26. In the AWS Cloud9 terminal, run the following AWS CLI **put-item** command:

```
x
aws dynamodb put-item \
--table-name FoodProducts \
--item file://../resources/an_existing_product.json \
--condition-expression "attribute_not_exists(product_name)" \
--region us-east-1
```

The command should return this output:

```
xxxxxxxxxx
An error occurred (ConditionalCheckFailedException) when calling the     PutItem operation: The conditional request failed
```

This behavior is expected because the condition expression prevented an overwrite of the existing item.

If you like, you can also make sure the record was unchanged by viewing the data in the DynamoDB console again.

Next, you apply what you learned about conditional expressions to working with the SDK.

Task 4: Adding and modifying a single item by using the SDK

Sofía now has a good understanding of how to use the AWS CLI to control the data that is inserted into the table. She knows that the behavior for inserting data is similar with the SDK. She decides to write to the table by using Python code.

In this task, you continue as Sofía to add and modify a single item by using the SDK.

27. Update the conditional_put.py script.
    - In the AWS Cloud9 IDE, go to the **python_3** directory.
    - Open the **conditional_put.py** script.
    - Replace the <*FMI*> placeholders as directed in the script. You can also refer to the code analysis in the following step.
    - In the upper left, choose **File > Save** to save your changes.

28. Review the code to understand what it does:
    - Focus on the definition of the **response** variable, which begins on line 49.
    - Notice the call to the **put_item** operation. In the SDK for Python, the **put_item** operation is equivalent to the **put-item** command in the AWS CLI.
    - The **put_item** SDK operation requires a value for **Item**, which defines the record that is inserted into the table.

    This record contains five attributes:

```
xxxxxxxxxx
'''
    You must replace <FMI_1> with the table name FoodProducts
    You must replace <FMI_2> with a product name. apple pie
    You must replace <FMI_3> with a444
    You must replace <FMI_4> with 595
    You must replace <FMI_5> with the description: It is amazing!
    You must replace <FMI_6> with a tag: whole pie
    You must replace <FMI_7> with a tag: apple
'''

import boto3
from botocore.exceptions import ClientError

def conditional_put():

    DDB = boto3.client('dynamodb', region_name='us-east-1')

    try:
        response = DDB.put_item(
            TableName='FoodProducts',
            Item={
                'product_name': {
                    'S': 'apple pie'
                },
                'product_id': {
                    'S': 'a444'
                },
                'price_in_cents':{

                    'N': '595' #number passed in as a string (ie in quotes)

                },

                'description':{

                    'S': "It is amazing!"

                },

                'tags':{

                    'L': [{

                            'S': 'whole pie'

                        },{

                            'S': 'apple'

                        }]

                }

            },

            ConditionExpression='attribute_not_exists(product_name)'

        )
```

29. In the AWS Cloud9 terminal, run the file.

    ```
    xxxxxxxxxx

    python3 conditional_put.py
    ```

    If the command completes successfully, the terminal output should show the message **Done**. If you do not see the expected response, review the previous steps as well as any error messages and try again.

30. Return to the DynamoDB item explorer, and review the updated data. You should find an entry for **apple pie**:

    DDB_insert_4

31. In the AWS Cloud9 IDE, update the **conditional_put.py** script again. This time, replace the **product_id** value of <*a444*> to `a555` and save the file.

32. Run the script again:

    ```
    xxxxxxxxxx

    python3 conditional_put.py

    #Done
    ```

33. In the DynamoDB item explorer, review the table data.

    As you might expect, the item remains **unchanged**. Because the condition **attribute_not_exists(product_name)** was included in the **put_item** operation, the item was not overwritten. This failure behavior is exactly the behavior that you want.

    What do you think will happen if you change only the product name (primary key) to **cherry pie** but keep the same attributes using that conditional expression?

34. In the AWS Cloud9 IDE, update the **conditional_put.py** script by replacing the **product_name** value of <*apple pie*> to `cherry pie` and saving the file.

35. Run the `python3 conditional_put.py` again.

36. In the DynamoDB item explorer, review the data:

    DDB_insert_5

    As expected, a **new** record was added to the table. Now, only new products will be added to the table. This feature prevents accidental updates to existing records when more records are inserted.

Task 5: Adding multiple items by using the SDK and batch processing

Sofía is glad that the café staff will be able to load individual records into the table. However, she knows that it isn't scalable for café staff to load records into the database one at a time. She knows that it's more efficient to use a batch process for loading a large quantity of records at one time.

In this task, you continue as Sofía to implement batch processing by using the SDK.

Because this batch load contains all product records, you must delete all existing records from the table before you run it.

37. In the DynamoDB **Item explorer**, refresh the view of the data by choosing Run

38. Delete all records:
    - Select the check boxes for all the table records.
    - From the **Actions** menu, choose **Delete item(s)**.
    - In the pop-up window confirmation box, enter `Delete` and choose Delete items

39. In the AWS Cloud9 IDE, open the **resources** > `test.json` file, and review the data.

    This file contains six records that you use to test the batch-load script. Notice that this file contains multiple entries for **apple pie** on purpose.

    Now, update the script that performs the batch load.

40. Update the **test_batch_put.py** script:
    - In the AWS Cloud9 IDE, open the **python_3** > **test_batch_put.py** script.
    - Update the *<FMI_1>* placeholder with the `FoodProducts` table name.
    - Replace the *<FMI_2>* with the `product_name` primary key name.
    - In the upper left, choose **File > Save** to save your changes.

41. To understand what the script does, review the code:
    - The table that will be written to by the script is defined in the **table** variable on line 11.
    - The **with** statement that begins on line 12 calls **batch_writer()**, which opens the connection to the database.
    - Then, the code loops through each record and inserts the new data into the **FoodProducts** table:

    ```
    xxxxxxxxxx
    table = DDB.Table('FoodProducts')
    with table.batch_writer(overwrite_by_pkeys=['product_name']) as batch:
       for food in food_list:
           price_in_cents = food['price_in_cents']
           product_name = food['product_name']
    ```

42. In the AWS Cloud9 terminal, run the file:

    ```
    xxxxxxxxxx
    python3 test_batch_put.py
    ```

    After the script completes, the terminal should show the following output:

    ```
    xxxxxxxxxx
    Adding food item: {'product_name': 'apple pie', 'price_in_cents': 595}
    Adding food item: {'product_name': 'cherry pie', 'price_in_cents': 395}
    Adding food item: {'product_name': 'apple pie', 'price_in_cents': 795}
    Adding food item: {'product_name': 'key lime pie', 'price_in_cents': 195}
    Adding food item: {'product_name': 'apple pie', 'price_in_cents': 195}
    Adding food item: {'product_name': 'apple pie', 'price_in_cents': 4495}
    ```

    What is the **price_in_cents** value that you expect to find for **apple_pie**? Validate your theory by checking the data in the console.

43. In the DynamoDB **Item explorer**, select the **FoodProducts** table, and run the scan again.

    DDB_insert_6

    Instead of keeping the first value of **price_in_cents**, for **apple_pie**, the most recent value in the data file was applied. Why did this behavior happen?

    With single-item PUT requests (**put_item**), you can avoid overwriting duplicate records by including a condition. However, with batch inserts, you have two options for handling duplicate keys. You can either allow the overwrite, or you can cause the entire batch process to fail.

    - Review the **test_batch_put.py** script again. Focus on line 12.
    - The **overwrite_by_pkeys=['product_name']** parameter is included in the **batch_writer** method. This parameter tells DynamoDB to use **last write wins** if the key already exists.
    - **Last write wins** is why the **price_in_cents** attribute was updated for **apple pie**.

    However, you know that the café doesn't want the database to add incorrect values. For this dataset, it's better for the load to fail when duplicate **product_name** values are found instead of allowing the update to add incorrect values.

    ou must change the script so that it fails when duplicates are included in the batch. You can then review and clean up the data. To implement this feature, you remove the **overwrite_by_pkeys** parameter from the **batch_writer** method.

44. To prepare for the production data load, go to the browser tab with the DynamoDB console, and delete all records from the table as you did in the previous steps.

45. You can fix the overwrite behavior by updating the **test_batch_put.py** script and preparing to load the production data.

    - In the AWS Cloud9 IDE, open **python_3** > **test_batch_put.py**.
    - Update line 12 by changing *<with table.batch_writer(overwrite_by_pkeys=['product_name']) as batch>* to the following and saving the file:

    ```
    xxxxxxxxxx

    with table.batch_writer() as batch:
    ```

    💁‍♂️ It's Python, so watch those indentations!

46. Now run the script again:

```
xxxxxxxxxx

python3 test_batch_put.py
```

You will notice errors, which is what what you want this time.

```
xxxxxxxxxx

Adding food item: apple pie 595

Adding food item: cherry pie 395

Adding food item: apple pie 795

Adding food item: key lime pie 195

Adding food item: apple pie 195

Adding food item: apple pie 4495

Traceback (most recent call last):

  File "test_batch_put.py", line 27, in <module>

 batch_put(food_list)

  File "test_batch_put.py", line 21, in batch_put

 batch.put_item(Item=formatted_data)

  File "/usr/local/lib/python3.6/site-packages/boto3/dynamodb/table.py", line 156, in __exit__

 self._flush()

  File "/usr/local/lib/python3.6/site-packages/boto3/dynamodb/table.py", line 137, in _flush

 RequestItems={self._table_name: items_to_send})

  File "/usr/local/lib/python3.6/site-packages/botocore/client.py", line 357, in _api_call

 return self._make_api_call(operation_name, kwargs)

  File "/usr/local/lib/python3.6/site-packages/botocore/client.py", line 676, in _make_api_call

 raise error_class(parsed_response, operation_name)

botocore.exceptions.ClientError: An error occurred (ValidationException) when calling the BatchWriteItem operation: Provided list of item keys contains dup
```

The important feedback in this output is the **ClientError: An error occurred (ValidationException) when calling the BatchWriteItem operation: Provided list of item keys contains duplicates**.

47. In the DynamoDB item explorer, scan the table again, and notice that no items have been added at all.

    This is good; you wish to fail hard and fast if there is a problem. You don't want some items being added and some not. You can leverage this error to catch issues in your JSON data and weed out any duplicate items.

    Using this fail-fast approach, you can now try to load the actual data that the website will use.

48. In AWS Cloud9, review the contents of the **resources/website/all_products.json** file. You will find many items. These items have several attributes, and some include an optional integer attribute called **specials**.

    In order to load the raw JSON used in the website, you use a new script called **batch_put.py**.

    It is very similar to the **test_batch_put.py** script. This script allows for the optional integer **special** attribute and also maps the names of more fields to the correct DynamoDB attribute types.

49. Modify the **python_3/batch_put.py** script.

    - Replace <*FMI*> with `FoodProducts`
    - In the upper left, choose **File > Save** to save your changes.

50. Run the script:

    ```
    xxxxxxxxxx
    python batch_put.py
    ```

    After the script completes, the terminal should show the following output:

    ```
    xxxxxxxxxx
    Adding special food item: apple pie slice 595
    Adding food item: chocolate cake slice 595
    Adding food item: chocolate cake 4095
    Adding special food item: apple pie 4595
    Adding special food item: chocolate chip cupcake 495
    Adding food item: vanilla cupcake 495
    Adding food item: chocolate cupcake 495
    Adding food item: peanutbutter and chocolate cupcake 495
    Adding special food item: strawberry cupcake 495
    Adding food item: vanilla glazed doughnut 295
    Adding food item: cinnamon doughnut 295
    Adding special food item: blueberry jelly doughnut 295
    Adding food item: chocolate doughnut 295
    Adding food item: powdered sugar doughnut 295
    Adding food item: raspberry jelly doughnut 295
    Adding food item: boston cream doughnut 295
    Adding food item: eclair 295
    Adding food item: lemon pie slice 595
    Adding food item: lemon pie 4595
    Adding food item: cherry pie slice 595
    Adding food item: cherry pie 4595
    Adding food item: chocolate iced doughnut 295
    Adding special food item: plain bagel 395
    Adding food item: poppy seed bagel 395
    Adding food item: garlic bagel 395
    Adding food item: blueberry bagel 395
    ```

51. In the DynamoDB **Item explorer**, review the inserted data. You should now see 26 items! (The image below is truncated image for brevity.)

    DDB_batch_load

Task 6: Querying the table by using the SDK

Now that the data is loaded, Sofía needs a way to retrieve, or *query*, the product information from the DynamoDB table.

The SDK has two operations for retrieving data from a DynamoDB table: **scan()** and **query()**.

The **scan** operation reads all records in the table, and unwanted data can then be filtered out. If only a subset of the table data is needed, the **query** operation often provides better performance because it reads only a subset of the records in the table or index.

Frank and Martha want to show all the menu items on the café website, so Sofía decides to use the **scan()** operation to retrieve all records from the table.

In this task, you continue as Sofía to implement this feature.

Note: Later in the application development process, you use this Python code in an AWS Lambda function. The Lambda function retrieves the table records so that the café website can display all the pastries.

52. Edit the script that selects all records from the table:
    - In AWS Cloud9 IDE, open **python_3** > **get_all_items.py**.

      Note: Do not use the **get_all_items_py** file in the **resources** folder.
    - Update the *<FMI_1>* placeholder with the `FoodProducts` table name.
    - In the upper left, choose **File > Save** to save your changes.
53. Review the **get_all_items.py** script to understand what it does:
    - On line 15, the scan operation is defined in the **response** variable.
    - Notice the **while** loop that begins on line 18.
      - If the result of the scan operation is large, DynamoDB splits the results into 1 MB chunks of information (or the first 25 items if their total is less than 1 MB). These chunks of returned data are called *pages*.
      - When the **while** loop runs, the code processes each page, which is also known as *pagination*. The loop then appends records to the end of the result set until all data has been received:

    ```
    xxxxxxxxxx
     response = table.scan()
     data = response['Items']
     while response.get('LastEvaluatedKey'):
        response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
         data.extend(response['Items'])
    ```

54. In the AWS Cloud9 terminal, run the script:

    ```
    xxxxxxxxxx
    python3 get_all_items.py
    ```

    The returned data should be similar to the following output:

    ```
    xxxxxxxxxx
    [{'price_in_cents': Decimal('295'), 'special': Decimal('1'), 'description': 'so good!', 'product_name': 'blueberry jelly doughnut', 'product_id_str':
    ```

    Notice that Python returns the numeric data for the **price_in_cents** attribute as **Decimal(number as string)**. This is not valid JSON, but that's OK for now. You address this issue in a future lab.

    You have a good start with this query, which returns all records. However, you want to try and search based on a attribute so that Frank and Martha could query for a specific product if they want to by using the **product_name**. You decide as a proof of concept to create a new script that returns a single product instead of returning all products.

55. Update the **get_one_item.py** script.
    - Replace the *<FMI_1>* with the name of the table's primary key.
    - In the upper left, choose **File > Save** to save your changes.

56. Review the code to understand what it does:
    - Focus on lines 13 and 14, which define the **response** variable. The **get_item** operation requires a **TableName** and a **Key**. The **Key** parameter is used to compare the table's primary key, **product_name**, with the value that is passed in from the main module of the script.

- On line 24, note the value that's assigned to the **product** variable. Also observe that this value is passed to the **get_one_item** function.

  ```
  xxxxxxxxxx
  response = DDB.get_item(TableName='FoodProducts',
    Key={
     'product_name': {'S': product}
     }
    )
  data = response['Item']
  print (data)
  if __name__ == '__main__':
    product = "chocolate cake"
    get_one_item(product)
  ```

  Note: The **get_item** operation is a higher level abstraction of the query operation. It is designed to return a single item.

57. In the AWS Cloud9 terminal, run the following command:

    ```
    xxxxxxxxxx
    python3 get_one_item.py
    ```

    If the script completes successfully, the output should be similar to the following example:

    ```
    xxxxxxxxxx
    {'price_in_cents': {'N': '4095'}, 'description': {'S': 'chocolate heaven'}, 'product_name': {'S': 'chocolate cake'}, 'product_id_str': {'S': 'a446'},
    ```

    You will notice that this is now in the DynamoDB object format (unlike the query you did before) and thus includes keys such as **N** and **S**. This is fine for now. You could parse that out and turn it into JSON that the website can understand using Lambda. However, this feature was only for a proof of concept.

    Sofía demonstrates the database features that she developed for the website to Frank, Martha, and the café staff. They are pleased that the website will soon be able to show all the live product information instead of the hard-coded version that they have currently.

Task 7: Adding a global secondary index to the table

Frank and Martha like the proof of concept that can now query the database for a specific product but return to Sofia with a request for a slightly more useful feature. The café staff originally thought that they would like to list all the products when the main page loads, but loading all 26 images would slow down the website.

They considered a pagination feature, but with only 26 items, this option feels unnecessary. Instead, they have asked Sofia to expand on the proof of concept that searched based on a product name. This time, they need to search for items that are both part of the weekly special and also show that they are "on offer" in the **tags** attribute. This way when the website default page loads, it fetches only the featured menu items.

You can also give the users the option to show all items when building the website. This option makes the website much more efficient. You make these website changes in a later lab to accommodate this new loading process.

Sofia knows that searching on a primary key is easy as per her proof of concept. However, in order to search on attributes that are not part of a primary key, she needs to add a **Global Secondary Index** to the existing **FoodProducts** table.

In this task, you continue as Sofia to implement this feature.

To change the characteristics of a table, such as adding an index, you must use the **UpdateTable** operation. For more information on this operation, refer to the Amazon DynamoDB Developer Guide at https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.Basics.html#WorkingWithTables.Basics.UpdateTable.

58. Update the **add_gsi.py** script.

    - Replace the *<FMI_1>* with the **KeyType** of `HASH`
    - In the upper left, choose **File > Save** to save your changes.

59. To understand what the **add_gsi.py** script does, open this script to review the code.

    - Review the **params** variable on line 12. Focus on the **GlobalSecondaryIndexUpdates** parameter. Notice that a new index named **special_GSI** is created. This new index consists of one attribute: **special**.
    - Similarly to the table creation, the line that defines the **table** variable also updates the table.

```
xxxxxxxxxx
    params = {
        'TableName': 'FoodProducts',
        'AttributeDefinitions': [
            {'AttributeName': 'special', 'AttributeType': 'N'}
        ],
        'GlobalSecondaryIndexUpdates': [
            {
                'Create': {
                    'IndexName': 'special_GSI',
                    'KeySchema': [
                        {
                            'AttributeName': 'special',
                            'KeyType': 'HASH'
                        }
                    ],
                        'Projection': {
                        'ProjectionType': 'ALL'
                    },
                        'ProvisionedThroughput': {
                        'ReadCapacityUnits': 1,
                        'WriteCapacityUnits': 1
                    }
                }
            }
        ]
    }

    table = DDB.update_table(**params)
```

60. In the AWS Cloud9 terminal, run the following command:

    ```
    xxxxxxxxxx
    python3 add_gsi.py
    ```

    If the command completes successfully, the terminal output should display the message **DONE**.

    Note: It can take up to 5 minutes for the index to populate.

61. In the DynamoDB console, monitor the status of the index:
    - Choose **Tables**.
    - Choose **FoodProducts**.
    - Choose the **Indexes** tab.
    - Wait until the **Status** changes from **Creating** to **Active**.

GSI Status

Note: If you are using the new console, the **Status** column may become hidden a few minutes after the index creation and population has completed.

The **special_GSI** index is a sparse index, meaning it does not have as many items as the main table. It is a subset of the data and is more efficient to scan when you want to find only the items that are part of the specials menu.

62. Update the **scan_with_filter.py** script.
    - Change *<FMI_1>* to `special_GSI`
    - Change *<FMI_2>* to `tags`

   - In the upper left, choose **File > Save** to save your changes.

63. Review the code.
   - On line 18, including the **IndexName** option lets the **scan** operator know that it will be going to the index and not the main table to read the data.
   - On line 19, the filter expression processes the records that have been read and shows only records that meet the comparison criteria. In this case, it shows records only if they don't have **out of stock** in the **tags** attribute. This ensures that only items that are available or "on offer" are shown to customers.

     ```
     xxxxxxxxxx
     response = table.scan(
       IndexName='special_GSI',
       FilterExpression=Not(Attr('tags').contains('out of stock')))
     ```

     Note: You may be familiar with the older DynamoDB parameter, ScanFilter. This is a **legacy** parameter. FilterExpression should be used instead. FilterExpression does not include all of the operators that were provided with ScanFilter, for example NOT_CONTAINS. This is why you wrapped the comparison inside the **Not()** function.

64. Save the file and run it.

    ```
    xxxxxxxxxx
    python3 scan_with_filter.py
    ```

    The output should be similar to the following:

    ```
    xxxxxxxxxx
    [{'price_in_cents': Decimal('295'), 'special': Decimal('1'), 'description': 'so good!', 'product_name': 'blueberry jelly doughnut', 'product_id_str':
    ```

    Again, the response is not in the format the website requires. However, when you use this code in a Lambda function in a later lab, you will adjust then.

Update from the café

Sofia is happy with the progress that she has made. The database table is loaded with data, she addressed the café's database backend requirements, and she will soon wire this into the website.

Sofia's next task is to create an API that the website can use.

Sofia decides to stop for the day, but she's looking forward to her next task. She plans to start working on creating a mock API that she can use for testing.

Submitting your work

65. At the top of these instructions, choose **Submit** to record your progress and when prompted, choose **Yes**.
66. If the results don't display after a couple of minutes, return to the top of these instructions and choose **Grades**.

Tip: You can submit your work multiple times. After you change your work, choose **Submit** again. Your last submission is what is recorded for this lab.

67. To find detailed feedback on your work, choose **Details** followed by **View Submission Report**.

Lab complete

Congratulations! You have completed the lab.

68. Choose End Lab at the top of this page, and then select Yes to confirm that you want to end the lab.

    A panel indicates that *DELETE has been initiated... You may close this message box now.*

69. Select the **X** in the top-right corner to close the panel.

LAB 6.1

[Version 1.0.5]

Lab 6.1: Developing REST APIs with Amazon API Gateway