Code walk-through:

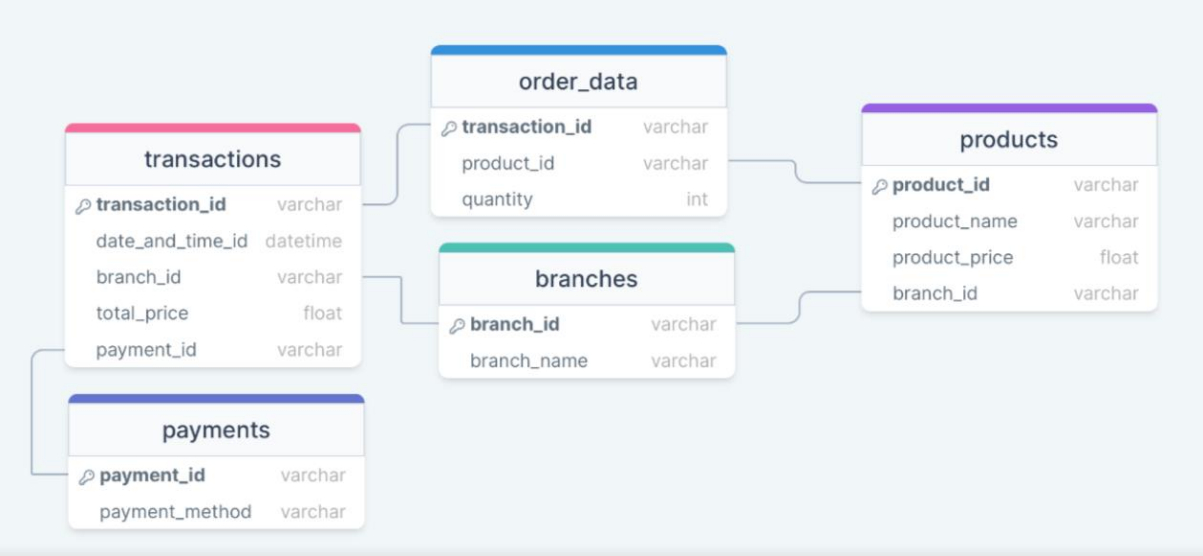
Our objective:

To take data from a CSV file, reformat it into lists that have the information we need and then use those individual lists to fill data tables in our database.

In order to reach this goal, we must work backwards and understand what sorts of tables we would need, what information we would need for each table and then create specific lists that cater to these requirements.

One example would be the products table. On this table we would want EACH item we have from the CSV file, it's respective price and of course a unique product id number that could be used to reference it in other tables. Additionally, we would want to know which branch sold this item, so the id number for that specific branch included (especially if the same product is only sold at specific stores or at different prices in each store).

(A visual representation of this I stole from Christina 😊)



This details all info. that each table will display and the relationships/dependencies of each one.

Now the end goal has been established, we can get to the nitty gritty of things.

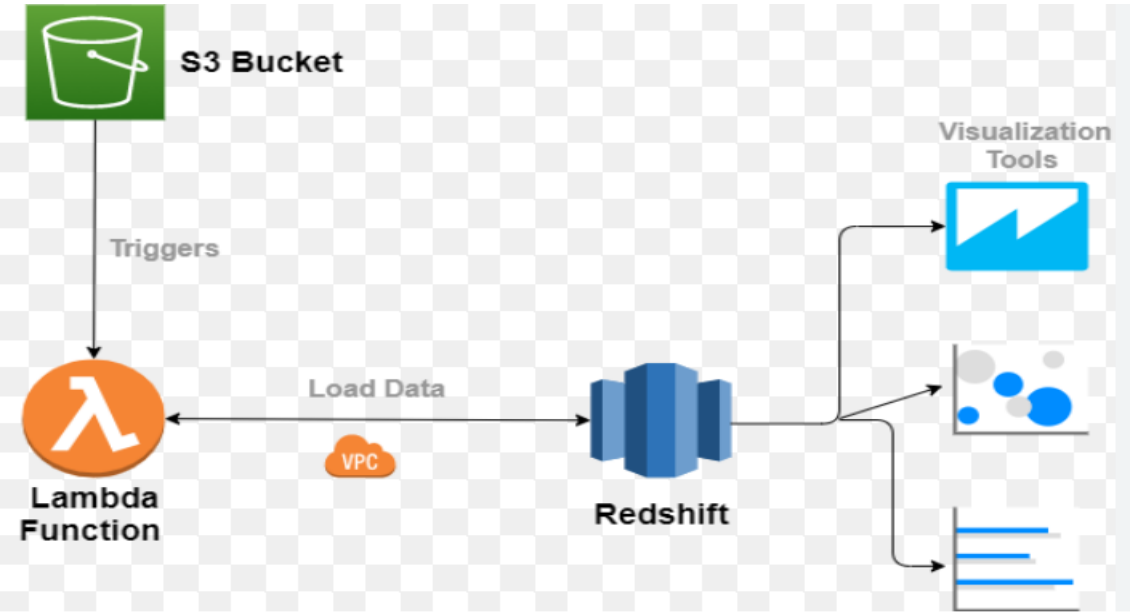
We are using AWS services so need to implement a method that utilizes it's features (Buckets, Lambdas, Cloud Monitoring).

The main construct is

A) An Event occurs in our file holder (the S3 Bucket)

B) Next the file holder is associate with a Lambda (this lambda containing the code we want to run to extract our data from the CSV and then format it, form lists from it and then put them lists into tables.) which is triggered to get to work (I.e run our code)

C) Lastly after the code is run, we will have our data in the AWS Redshift database ready for use (querying/making tables.)



(Only thing missing here is an error to indicate that an Actual event occurred first **[*which was CSV files being placed in our File holder - the S3 Bucket*]** that triggered this whole domino effect.)

In order to use a Lambda function, we need to have the code that will do the reformatting and creating lists **READY** to put inside it and then we employ the use of our good friend **"lamb_handler"**, which is ACTUALLY the function which is activated when the lambda is triggered. So, if you want anything to occur during the lambda function activation, you better have that piece of code written in the lambda_handler.

```
def lambda_handler(event, context): #To extract data from CSV and create a list from it.
    try:
        print(f'lambda_handler: started: event=${event}')

        bucket = event["Records"][0]['s3']['bucket']['name']
        #code to find bucket the event took place in
        print("Bucket located")
        s3file= event["Records"][0]['s3']['object']['key']
        #code to find file the event that triggered event

        print(f'lambda_handler: bucket={bucket}, key={s3file}')
        #code to print file/bucket that triggered event

        response = s3.get_object(Bucket=bucket, Key=s3file)
        #code to get this file from bucket

        #Next step is to actually extract data from file
        file = response['Body'].read().decode('utf-8').split('\n')
        #this will read the file and decode so is in correct format
```

This is just a small excerpt from the code that's present in our lambda_handler, but shows you how it's written etc. Also, we must make sure that it has the "def" statement before it and the arguments (event, context) otherwise it won't run correctly. ***-NOT SURE WHY YET BUT WILL TRY TO FIND OUT***

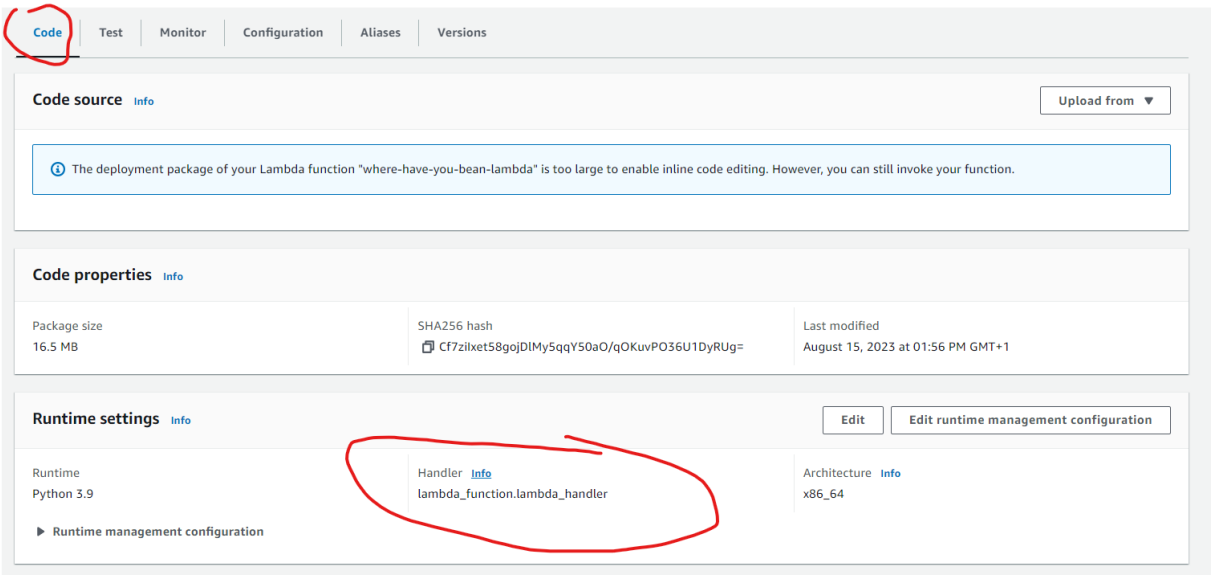
This lambda_handler is a function (As you can see. It needs to be defined like functions usually do and have the code it wants to run when called specified inside it with the correct indentation).

N.B: We do not need to call the function afterwards by writing lambda_handler(). The function will activate itself. We simply need to know which .py file it is in and then specify this in the settings on AWS making sure that the .py file and function are correctly listed.

This needs to be stored in our lambdas settings on the AWS site in the following format: **"file.nameoflambdahandlerfunction"**

Now, in our case, the lambda handler function is called **"lambda_handler"** and the .py file it's located in is called **"lambda_function"** >>> Thus, it becomes: **"lambda_function.lambda_handler"**

The names for both **can be changed** as long as we make sure that the settings match that on the AWS site, so the system knows what to run and where to find it when the event occurs. This can be found when going to the AWS site >> searching for your specific lambda created, then going down to the "code" menu, and then looking up the "Runtime settings".

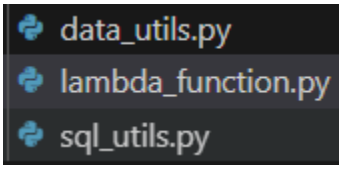


Now specifically for our code. The tools we'll be using have been separated into 3 files.

1)**Data_utils.py** : Where all the functions we will be using have been defined and can be called in our lambda_handler function.

2)**Sql_utils.py:** Where all functions related to creation of tables, establishing a connection with redshift and loading the tables have been defined and can be called from into our lambda_handler function

3)**Lambda_function.py**: Where the lambda_handler (thing that will execute everything) lives and all the processes will actually take place.



To make sure the lambda_handler has access to all of these functions in other files and the dependencies it needs, we write this at the top of the lambda_function.py

```
import json
import boto3 # library used to access AWS API
import csv
import psycopg2 as psy
from datetime import datetime
from data_utils import *
from sql_utils import *
```

So it imports all the functions from both **data_utils** and **sql_utils** and then can use them easily.

N.B: Import * = import everything within that specific file.

Step by step walk-through of code:

Phase 1:

The code begins with us acknowledging that an event occurred in our S3 bucket i.e., someone putting something in the bucket (which will produce an **EVENT log** that is not viewable by us directly unfortunately) and will be in this dictionary within a list format:

```
{
    'Records': [
        {
            'eventVersion': '2.1',
            'eventSource': 'aws:s3',
            'awsRegion': 'us-east-1',
            'eventTime': '2023-08-20T12:00:00.000Z',
            'eventName': 'ObjectCreated:Put',
            's3': {
                's3SchemaVersion': '1.0',
                'bucket': {
                    'name': 'your-bucket-name',
                    'arn': 'arn:aws:s3:::your-bucket-name'
                },
                'object': {
                    'key': 'path/to/your/file.csv',
                    'size': 12345,
                    'eTag': 'abcdefg1234567890',
                    'sequencer': '0A1B2C3D4E5F678901'
                }
            }
        },
        # Additional records for other S3 events
    ]
}
```

As you can see here, the bucket it occurred in is mentioned under **“bucket: name”** and so is the type of event under **“eventName:”** which is **“ObjectCreated.Put”** (meaning an object was put into our bucket) and finally also what object was put inside under **“object:key”** which was **“path/to/your/file.csv.”** (Basically, they put it in a series of folders and the file was called **“file.csv”**. This is a list of dictionaries, and we would have had more if other events occurred at the same time.!!! So, we can look up this dictionary and find the info we want.)

Using this event info, we can tell the lambda what to look for and where (although it **should already know** since It’s only associated with one bucket) AND THEN use this info to get the **file** and extract data from it ..or do whatever else you want to do with it. **The world is your oyster!**

```
def lambda_handler(event, context): #To extract data from CSV and crea
    try:
        print(f'lambda_handler: started: event=${event}')

        bucket = event["Records"][0]['s3']['bucket']['name']
        #code to find bucket the event took place in
        print("Bucket located")
        s3file= event["Records"][0]['s3']['object']['key']
        #code to find file the event that triggered event

        print(f'lambda_handler: bucket={bucket}, key={s3file}')
        #code to print file/bucket that triggered event

        response = s3.get_object(Bucket=bucket, Key=s3file)
        #code to get this file from bucket

        #Next step is to actually extract data from file
        file = response['Body'].read().decode('utf-8').split('\n')
        #this will read the file and decode so is in correct format
```

Now that we know this event log will be produced when the event occurs, we can look it up and extract the information we need to get our CSV file that was dropped into the bucket and then start working on it.

- **First** we print a statement showing the **lambda_handler** has started **working**.

- **Second** we find the bucket that this event occurred in and save this information to a variable (we have called it **“bucket”** here.) by looking into the **“event”** list of dictionaries and specifying the exact path to get to our info… which was bucket name. Thus we look in the Records dictionary, then take the first entry (hence why it says 0) then we check the s3 dictionary, then bucket dictionary, then finally look up the field that says “name”.

Which gives us this code: **bucket = event["Records"][0]['s3']['bucket']['name']**

- **Third** we do the same to find the file name by looking up the same path (**Records> 0 for first entry in dictionary> S3**) but then check the **object** dictionary instead and then specify the **“key”** field to find the name of the file. Then assign this info to a variable called **s3file** (or whatever you want to call it).

s3file= event["Records"][0]['s3']['object']['key']

Then log this information into a print statement so we can track the progress of our function as it works through the instructions, we've given it!

print(f'lambda_handler: bucket={bucket}, key={s3file}')

- **Fourth** we read the content of the CSV file found and then decode it from bytes to a UTF-8 string, splitting it into lines using the newline character ('\n') as the delimiter. This is typically done when you're processing text-based files, such as CSV files, stored in an S3 bucket.

So the code is written as: **file = response['Body'].read().decode('utf-8').split('\n')**

- **Fifth** we take this **file** variable (which is the CSV decoded so we can make sense of its contents) and then actually put its information into a nice list of dictionaries. We assign this list of dictionaries a variable name (We've chosen **"data"** in this case)

So, we will need to use **csv.DictReader** to complete this step (**Something you've probably used in your mini project previously**)

```
36      print("Data extracted!")
37      source_file =csv.DictReader(file, fieldnames=['purchase_date', 'branch', 'customer_name',
38                                                    'items', 'total', 'mode_of_payment', 'cardnumber'], delimiter=',')
39      data = []
40
41      for row in source_file: #the following lines correct the date format so it can be uploaded to a database later
42              date_str = row["purchase_date"]
43              datetime_obj = datetime.strptime(date_str, "%d/%m/%Y %H:%M")
44              row["purchase_date"] = datetime_obj.strftime("%Y-%m-%d %H:%M:%S")
45              data.append(row)
46      print("Data placed into a list.")
```

N.B:

Steps that take place from line 42 to 44 are included to reformat the data extracted from the CSV in the **"date and time"** column specifically from format:

DAY, MONTH, YEAR, HOUR, MINUTE into the format: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND

Because later our database will only accept it in this structure. Thus, it is better to do now!

This is all put inside a **"try"** and **"except"** blocks of code to allow us to find any errors and then print them as messages so we can know where to look then fix them should they occur.

Phase 1 complete. = Data extracted and placed into the variable "data" which should look like this (these are just 2 entries from the data list of dictionaries that.)

```
Data placed into a list.
{'purchase_date': '2021-08-25 09:00:00', 'branch': 'Chesterfield', 'customer_name': 'Richard Copeland', 'items': 'Regular Flavoured iced latte - Hazelnut - 2.75, Large La
tte - 2.45', 'total': '5.2', 'mode_of_payment': 'CARD', 'cardnumber': '5494173772652516'}

{'purchase_date': '2021-08-25 09:02:00', 'branch': 'Chesterfield', 'customer_name': 'Scott Owens', 'items': 'Large Flavoured iced latte - Caramel - 3.25, Regular Flavoure
d iced latte - Hazelnut - 2.75, Regular Flavoured iced latte - Caramel - 2.75, Large Flavoured iced latte - Hazelnut - 3.25, Regular Flavoured latte - Hazelnut - 2.55, Re
gular Flavoured iced latte - Hazelnut - 2.75', 'total': '17.3', 'mode_of_payment': 'CARD', 'cardnumber': '6844802140812058'}
```

Phase 2: Reformatting Data.

Now we must make the lists we want to use with specific info (previously mentioned), so we can take them and fill in the tables in our database!

The first thing that is done however is establish a connection with our database because soon we will need to query it (ask it questions and investigate) to see if certain values or information already exist in my Database and then use them instead of creating new values. For example check if a certain product already exists on the products table (so I don't end up inserting it a second time) and then find out it's unique product id so I can use that value within the lists we're creating and insert that information correctly (I.e. not assign a new product id number to "Large latte" when it already has one and then put that wrong id on the orders table when doing queries or referencing it it.)

The lines of code below accomplish this and allow us to access the redshift database. Creating a connection and cursor variable that we use each time we try to access the database with any of the functions we use. We use two functions here **"get_ssm_param"** and "**open_sql_database_connection_and_cursor"**. Both can be found in the **sql_utils.py** file but they just specify the credentials we use when logging in.

```
52      redshift_details = get_ssm_param(param_name)
53      print("")
54      connection, cursor  = open_sql_database_connection_and_cursor(redshift_details)
55      print("Processed complete! Setup connection with redshift")
param_name = 'where_have_you_bean_redshift_settings'
```

Then we run the code to create the tables we need present in the database and have them ready to insert things into and query (to check for values already existing as in the situation I mentioned above that could cause errors in our data).

```
create_db_tables(connection, cursor)
```

Steps for Reformatting:

1) If you recall above, the items field was filled with a variety of different items all together and their prices but only separated by a comma. This will make it difficult for us to work with because it's not organized so we can't ask python to insert the information into our tables until it's reorganized (diff. items in their own dictionaries with prices etc.). Thus, we use a few functions to slowly break this list down into something workable. First the function **"separate_items_into_list_within_a_list ()"** with arguments list **name1 = data** AND **col_name = "items".**

```
def separate_items_into_list_within_a_list(list_name1,col_name):
    for d in list_name1:
        if col_name not in d:
            print('The column name inserted is not in the list in
            list_name1 = 1
            break
        else:
            d[col_name] = d[col_name].split(",")
    return list_name1
```

This function checks **firstly** if the list has the column we're looking for ("items" in this case) and then **secondly** once it finds it, creates a list with square brackets around all the entries, deciding where one entry starts and ends based on the delimiting factor " , ".

So, wherever we have a comma it decides that this is a new entry and then separates it in the list its creating. (**See below for example**)

After we have applied the function to our raw data set it goes from this:

{'purchase_date': '2023-08-11 09:10:00', 'branch': 'London Camden', 'customer_name': 'Elizabeth Browder', 'items': 'Large Iced latte - 2.85, Large Iced latte - 2.85, Regular Flavoured hot chocolate - Hazelnut - 2.60', 'total': '8.3', 'mode_of_payment': 'CARD', 'cardnumber': '3459479702703356'}

To this:

{'purchase_date': '2023-08-11 09:10:00', 'branch': 'London Camden', 'customer_name': 'Elizabeth Browder', 'items': ['Large Iced latte - 2.85', ' Large Iced latte - 2.85', ' Regular Flavoured hot chocolate - Hazelnut - 2.60'], 'total': '8.3', 'mode_of_payment': 'CARD', 'cardnumber': '3459479702703356'}

(**Notice the square brackets around the items column/key in the dictionary and how the function has placed quotation marks around each entry alone 'Large Iced latte – 2.85' instead of all the items Large ice latte and Regular flavored hot chocolate etc. at once**)

We now assign this new list of dictionaries to the variable **"reformatted_data"**.

2) We need to clean this newly acquired list of any useless or personal data (Such as customer info.). So, we create a function that can find a specific column/key and then remove any data associated with it. This function is "**Column_deleter()**." with arguments **list_name = "formatted data"** AND **column_name = "cardnumber"**

We then run this a second time with arguments **list_name = "formatted data"** AND **column_name= "customer_name"** to make sure we have all unwanted info. removed.

```
def coloumn_deleter(list_name, coloumn_name): #removes sensitive data
    for items in list_name:
        if coloumn_name in items:
            del items[coloumn_name]
    return list_name
```

This leaves us with dictionaries that look like this now: (***same example as used above***)

{'purchase_date': '2023-08-11 09:10:00', 'branch': 'London Camden', 'items': ['Large Iced latte - 2.85', ' Large Iced latte - 2.85', ' Regular Flavoured hot chocolate - Hazelnut - 2.60'], 'total': '8.3', 'mode_of_payment': 'CARD'}

This new list is called **"reformatted_cleaned_data".**

3) Next, we create the "**branches_list"** variable, which is a list of all the branches that we have found in the CSV file we're extracting data from and then make sure they are all unique with unique branch ids. This step is taking place now so we can use this information later down the line. (But we will return to reformatting the "reformatted cleaned data" list soon afterwards.)

So, in order to create the **"branches_list"** variable we use this function:

"**branch_list()"** with the arguments **data = "reformatted_cleaned_data"** AND **cursor= cursor.**

```
def branch_list(data, cursor): #Creates an enumerated dictionary of branches
    branches= []
    correct_list= []
    #STEP 1
    for x in data: #create a list of unique branches from the csv file
        y= x["branch"]
        if y not in branches:
            branches.append(y)
    #STEP 2
    count = 1
    for z in branches: #take the list above of branches and form a list of dictionaries composed of each unique branch
        ok = {"branch_id": count, "branch": z}
        correct_list.append(ok)
        count += 1
    #STEP 3
    # Check list of dictionaries made by Querying database to see if this branch already exists in the branches table.
    # If so, then assign it's current product_id (uuid) to this one. If not then give it a new one
    for branches in correct_list:
        # Define the branch_name you want to check
        desired_branch_name = branches["branch"]
        # Query to check if branch_name is present
        query = "SELECT branch_id FROM branches WHERE branch_name = %s"
        cursor.execute(query, (desired_branch_name,))
        result = cursor.fetchone()
        if result:
            branch_id = result[0]
            branches["branch_id"] = branch_id
            print(f"Branch with name '{desired_branch_name}' found. Branch ID: {branch_id}, so this GUID will be used.")
        else:
            new_guid_id_value = str(uuid.uuid4())
            print(f"Branch with name '{desired_branch_name}' not found so will be assigned a new GUID {new_guid_id_value}")
            branches["branch_id"] = new_guid_id_value
    return correct_list
```

<u>This function works in several steps:</u>

A) Creates 2 lists (**branches** and **correct data** and then uses each one for different purposes)

In **step 1**: First we take our list of dictionaries that is inserted (**reformatted_cleaned_data** in this case) and then loop through each of its entries (loop through the dictionaries inside.)

We investigate the "branch" field and then assign this value to the variable **y**. Then check if **y** is in our list called **branches** mentioned at the top. If it is not, we append it to the list (add it to the list). This we repeat for each dictionary inside our list inserted (**reformatted_cleaned_data**) and then at the end of the process will have a list of all branches mentioned in that inserted list (**reformatted_cleaned_data**). So now the branches list inside this function should be a list of all the unique branch names mentioned in our inserted list.

B) In step 2 we want to add the column/key **"Branch_id"** to each dictionary with a unique id number. So, we create a count variable and start it at 1. Then we loop through the branches list (which now is a list

of all the branches mentioned in this CSV file) and for each branch we take this name and then put it in another dictionary with the columns **“branch_id”** and **“branch”** and then assign a unique id to each branch (depending on what the count value is currently at. Check the **“ok”** dictionary as a point of reference). The first branch found during this loop will have id = 1 as that's what the count value is first. Then after each iteration of the loop, we add 1 to the current Count value so the next branch will have an id of (1+1) = 2… and so forth.

Now this gives us a list of dictionaries with each one containing a unique branch id and branch name. This list is saved under the variable **“correct list”**.

C) In Step 3 We wish to verify that the correct list has correct information (**lol**).

More specifically if the branch ids have been already created in our branches table and to use that value if so, otherwise use something called a **GUID (Globally unique identifier)**. This is a string of words and numbers that are unique and difficult to duplicate which makes sure we do not have overlaps when inserting large sets of data. For this we use a function called **uuid4** that python has pre-installed.

So first we loop through our **“correct list”** of branches we’ve created.

We assign a variable **“desired branch name”** as the branch name we’ll be looking for in the database table (**branches**).

We will next run a query using the line:

query = "SELECT branch_id FROM branches WHERE branch_name = %s"

(fyi, the % is specified in the next line)

Which looks into our database table **“branches”**, then checks the **“branch_name”** column specifically to see if the value we specified (**“desired branch name”**) exists there and then if it does locate a field with this name, it scans across into the **“branch_id”** column and selects the value in the **“branch_id”** column and stores this value into a variable result as the first entry.

This is accomplished with the help of the SQL query and **fetchone**() function.

Result variable should look something like this: **('0f71f2fa-790b-4769-b7fa-4d6b071a3936',)**

The first entry being our GUID already assigned to the branch we just looked for. (There are 2 entries in those brackets)

If we discover a result, then the code tells us to assign this value to the **“branch_id”** column.

If we do not, then in the “else” clause, it uses the uuid4() function to assign a new one with the help of the **uuid** module and “str” re-assigner to make sure python takes this value as a string (to avoid any errors.)

After this process has been done for all dictionaries (I.e all branches in the **correct list**) then it will return this list and we can assign it to the variable **“branches_list”** as stated at the very beginning. Thus, completing the branches list formation. (We will later use this list to form and or update our branches table in our Redshift database).

4) We take our **“reformatted_cleaned_data”** and then use that list to create a products list that we will use to fill/update our products table with.

For this we need to start by using the function **“form initial products list()”** with variable list_name1 = **“reformatted_cleaned_data”**

First, we create empty lists/dictionariy templates to fill with information that we’ve gathered from the code that follow: **“product table”, “list101” and “data_dict”.**

In **STEP 1** we verify first that the list has an items column and there are items present:

```
def form_initial_products_list(list_name1):
    products_table = [] #create an empty list
    data_dict = {'product_name': '', 'product_price': '', 'product_id': '', 'number_of_product_ordered': 0, 'branch':''}
    list101 = []
    #Step 1
    for d in list_name1:
        if 'items' not in d:
            print('The list inserted doesn\'t have an "items" column to work with')
            list101 = 1
```

In **STEP 2** we next loop through the ”**items”** column for each dictionary in our **“reformatted_cleaned_data”** list . Then we verify if the items have a hyphen “-”, which usually separates the item from its flavor and price. (e.g Large Flavoured latte - Vanilla – 2.85). If so, we continue and then list each object separated by this hyphen and store these objects in a list called **m**. An example of one iteration of the loop in the **“items”** column of each dictionary is:

“The item Regular Flavoured hot chocolate - Vanilla - 2.60 has m value [' Regular Flavoured hot chocolate ', ' Vanilla ', ' 2.60']”

Now depending on the number of entries in this list **“m”** we will do different things.

If m has 2 entries, we will take this to mean that only product name and price are present and thus we assign **“product_name”** to = **m[0]** (I.e the first entry in that list) and then **“product_price”** to = **m[1]** (I.e the second entry). And we save this to the empty dictionary we’ve created at the top. We use r.strip/l.strip to remove any excess spaces to the right and left of our **“product name”** so it’s inserted nicely into the dictionary we’re creating. We also use **“float”** to make sure that python remembers this new product price as a float as it will include decimals, and this will prevent errors later.

```
else:
    m = item.split("-")
    print(f'The item: {item} has m value: {m}')
    if len(m) == 2:
        data_dict['product_name'] = m[0].rstrip().lstrip()
        try:
            data_dict['product_price'] = float(m[1])
        except ValueError:
            print('Price is invalid')
            list101 = 3
            break
        data_dict['product_id'] = len(list101) + 1
        data_dict["branch"] = d["branch"]
        list101.append(data_dict)  # Use copy() to create a new dictionary
        data_dict = {'product_name': '', 'product_price': '', 'product_id': '', 'number_of_product_ordered': 0, 'branch':''}
```

If m has 3 entries, this means that the product name, flavor AND price are present and so we assign **“product_name”** to = **m[0] + “” + m[1]** I.e the first and second entry in that list (as we want to group the name and flavour separated by a space) and then **“product_price”** to = **m[2]** I.e the third entry. Once again using r/l.strip and float to make sure its correctly done.

Lastly we assign the product id to simply be the length of the current list of products that we’re forming and + 1 (So it starts at 1 for the first entry and then the next entry will have id 2 etc.) and **“branch”** to be the branch it already has in its current list we’ve picked up in the **“reformatted data list”** then append this dictionary to the **“list101”** list. We also present add the column/key **“number of product ordered”** and present that to 0. (But will potentially use this later).

```
elif len(m) == 3:
    data_dict['product_name'] = m[0].rstrip().lstrip() + ' ' + m[1].rstrip().lstrip()
    try:
        data_dict['product_price'] = float(m[2])
    except ValueError:
        print('Price is invalid')
        list101 = 3
        break
    data_dict['product_id'] = len(list101) + 1
    data_dict["branch"] = d["branch"]
    list101.append(data_dict)  # Use copy() to create a new dictionary
    data_dict = {'product_name': '', 'product_price': '', 'product_id': '', 'number_of_product_ordered': 0, 'branch':''}
```

If m turns out to have more than 3 entries we realise that there is an error and send an error message.

After we’ve done this for every entry in the dictionary, we return this list as **“products table”** and assign it to the **“initial_products_table”** variable.

```
            elif len(m) > 3:
                print('The software isn\'t accustomed to work with this format')
                list101 = 4
                break
    products_table = list101  # Add processed data to list_name2
    return products_table
```

This is one example entry from the **“initial_products_table”** list (that we just created)

{'product_name': 'Regular Speciality Tea Peppermint', 'product_price': 1.3, 'product_id': 5, 'number_of_product_ordered': 0, 'branch': 'London Camden'}

5) We take this newly formed list and of products with their flavors and their prices and then remove duplicate entries using the **“drop_duplicates()”** function with arguments list_name = **“initial_products_table”**, list_name2 = **“initial_products_table”** AND col_name= **“product_name”.**

```
def drop_duplicate(list_name,list_name2,col_name): #will remove duplicates of each product
    # Validity
    unique_list = set()
    result = []
    for dict in list_name:
        if dict[col_name] not in unique_list:
            unique_list.add(dict[col_name])
            result.append(dict)
    list_name2 = result
    return list_name2
```

This function will take the **"initial_products_table"** list and then check each dictionary within it. It will specify the column **"product_name"** within that dictionary and then check if that product name already exists in a list called **"unique_list"**. If not, then it will add it to that list as a reference point (so in short, we're forming a side list **"unique list"** that can hold the names of all products we've cycled through already. And we'll reference it to know if we've already added this product to the main list we're forming of products, including their respective prices etc.). Then this dictionary, if unique and not already present, will be added to our **"result"** list. So slowly but surely, we're creating a list of dictionaries, each unique that include the product_name, price, number of products ordered and branch (as is in our example from the initial products table above. This will remove all duplicate values of products in the **"initial_products_table."**)

We will call this new list of unique products **"new_products_table"**.

6) The next step involves using a function **"create_final_products_list()"** to update the **"number of products ordered"** column to count how many times a specific product appears in our **"reformatted_cleaned_data"** list from **step 3** and then correct this information in our **"new_products_table"** list of unique products.

Then call this new list formed **"final_products_table"**.

```
def create_final_products_list(big_data_list,products_list):
    for d in big_data_list:
        for items in d['items']:
            m = items.split("-")
            if len(m) == 2:
                product_name = m[0].rstrip().lstrip()
                product_price = float(m[1])
            elif len(m) == 3:
                product_name = m[0].rstrip().lstrip() + ' ' + m[1].rstrip().lstrip()
                product_price = float(m[2])
            else:
                continue
            for k in products_list:
                if k['product_name'] == product_name:
                    k['number_of_product_ordered'] = k['number_of_product_ordered'] +1
    return products_list
```

An example entry would be:

{'product_name': 'Regular Flavoured latte Vanilla', 'product_price': 2.55, 'product_id': 2, 'number_of_product_ordered': 63, 'branch': 'London Camden'}

(***I shall skip an explanation of this stage for now.***)

7) We wish to now correct this new list "final products table" by checking the **"product_id"** column. I want this value to be a GUID like we have for branches earlier and to make sure the GUID assigned to each product is new (I.e one hasn't already been given to it when we previously encountered that product on a different CSV file extracted in the past). So, we will need to query our database once more during the process and we use the function **"product_id_checker_and_reassigner"** for this purpose. We take the arguments list = **"final products table"** AND cursor = **cursor**.

```
def product_id_checker_and_reassigner(list, cursor):
    for dict in list:
        # Define the variables you want to check
        # product_id = dict["product_id"]
        product_name = dict["product_name"]
        product_price = dict['product_price']

        # Query to check if already present
        query = "SELECT product_id FROM products WHERE product_name = %s AND product_price = %s"
        cursor.execute(query, (product_name,product_price))
        result = cursor.fetchone()

        if result:
            dict["product_id"]= result[0]
            print(f"product_name '{product_name}' with product_price: {product_price} found, with

        else:
            new_guid_id_value = str(uuid.uuid4())
            print(f"Product '{product_name}' not found so will be assigned a new GUID {new_guid_id
            dict["product_id"] = new_guid_id_value
    return list
```

First step being we investigate our list **"final_products_table"** and then decide which columns we want to use to verify if a product is present. We have chosen **"product name"** and **"product price"** to ensure the product is the same and is the same price (as some products could be priced differently at different branches and so in this case should have a different product i/d).

So, we loop through the **"final_products_table"** list and for each entry in it we store these values and assign these values to the variables **"product_name"** and **"product_price"**.

Next, we write a query statement that looks inside the "products" table and checks if these values are present. (e.g if there the first dictionary during the loop has a product: Large latte, priced at 3.25, the query will check the table to see if these values are present in the **"product_name"** and **"product_price"** columns). If this query locates the values requested, then it checks that rows **"product_id"** column and stores the value in the variable **"result"** as the first entry. And lastly reassigns the **"product_id"** inside our dictionary to match this result.

If it however does NOT find any values matching this in the products table, it means we haven't encountered this product before (it is new) and so we needs to assign a new GUID to that product id column in the dictionary of products that we have (**final_products_table**).

This takes place in the **"else"** clause of the function. ***[We will not go over this step as it has already been covered in the branches_list step 3]***

This is an example entry in the final products table: (***notice the product id has been corrected***)

{'product_name': 'Regular Flavoured latte Vanilla', 'product_price': 2.55, 'product_id': '9688e87a-efc2-4df7-82e4-43b2fd87d55c', 'number_of_product_ordered': 63, 'branch': 'London Camden'}

8) We will now reassign the branch information in our final products table list. We use the **"branch_id_reassigner()"** function for this purpose. We take arguments list_to_change= **"final_products_table"** AND list_to_search = **"branches_list"**. --- (**STEP NOT REQUIRED. SINCE IS DONE AT NEXT STEP. WILL POTENTIALLY CONSIDER REMOVING)**

```
def branch_id_reassigner(list_to_change,list_to_search):
    new_list=[]
    for li in list_to_change:#creates a new list so big_table isnt changed
        d2 = copy.deepcopy(li)
        new_list.append(d2)
    for d in new_list:  # changes the branch name value to its branch id value
        search_value = d["branch"]
        next_list = next(item for item in list_to_search if item["branch"] == search_value)
        id = next_list["branch_id"]
        d["branch"] = id
    for e in new_list: #changes key "branch name"  to be "branch id" in this new list
        e["branch_id"] = e.pop("branch")
    return new_list
```

<u>This function is composed of 3 **for-loops**:</u>

The first one takes the list I want to change (**"final_products_table"**) and simply makes a copy of it (just so I don't affect this list with any commands in case I need it later). We call this new duplicate list **"new_list"**.

Then the second loop will look inside this **new_list**, take each dictionary and look inside for **"branch"** column, find its value and then assign it to a variable **"search_value"**. Afterwards it will use the next function (which is already stored in python so I don't need to define it) and to check my **"branches_list"** (what we created previously, listing all unique branches and their uuids as ids) in the column **"branch"** and check if this **"search_value"** exists anywhere. Then if it is found inside **"branches_list"**, save this specific dictionary that contains it to the variable **"next_list"**.

So, for example, if we were looking for **"Chesterton"** as our branch and thus our **"search value"**, the next function would take the word "Chesterton" and look inside the **"branches_list"** list of dictionaries. Then if has found it, would hold onto that specific dictionary and save it to the variable **"next_item"**.

After this, we look inside our newly acquired dictionary "next_item" and check what value is present in the **"branch_id"** column and save this to the variable "id". Then finally save this value to our **"branch"** column in the first dictionary we were querying I.e the dictionary associated with the **"final_products_table"**.

The last step is also to change the wording from "branch" to **"branch_id"** in our **"final products table"** list so it conforms with our tables scheme later. This is a minor detail but important to prevent errors later on.

This is repeated for all products in the **“final_products_table”** list.

This is an example entry from **“final_products_table”** after we have applied this function to it:

{'product_name': 'Regular Flavoured latte Vanilla', 'product_price': 2.55, 'product_id': '9688e87a-efc2-4df7-82e4-43b2fd87d55c', 'number_of_product_ordered': 63, 'branch_id': '0f71f2fa-790b-4769-b7fa-4d6b071a3936'}

(**“branch”** column has now become “branch_id” and the id has a guuid value.)

9)Next we reorganize this “final_products_table” list in alphabetical order (**upon reflection I’ve realized that this step is not needed but I shall leave it there for good measure**).

We use the **“sorting_alphabetically”** function for this purpose, taking arguments list_to_be_sorted= **“final products_table”** and value= **“product_name”**.

So it will essentially rearrange the products table into alphabetical order.

```
def sorting_alphabetically(list_to_be_sorted, value): #sorts the product
    newlist = sorted(list_to_be_sorted, key=lambda d: d[value])
    return newlist
```

This uses the sorted () function native to python. It basically takes my **“final_products_table”** and then takes each dictionary inside it. Then stores all of their values and rearranges them based on their “product_name” column.

This new list produced will be called “sorted_final_products_table”

10)The final step for the products table is to REASSIGN BRANCHES **but done already so will correct.**

11)Now we write out a standard payments table which should be easy since there are only 2 possible options. Payment by cash or by card. We want to make sure though, that the payments table has the correct payments_ids which will be GUIDS once again. So we need to do 1 of two things:

1) assign a new GUID to the payment id if one hasnt been already assigned

OR

2)find the already assigned GUID to each payment id that we will be able to locate by querying the payments table.

This step must be done before we can proceed so any lists made for the transactions table have the accurate payment id information.

We can use the function “payment_id_checker_and_reassigner” for this purpose.

It will take the arguments list = “payment table” AND cursor = cursor

```
def payment_id_checker_and_reassigner(list, cursor):
    for dict in list:
        # Define the branch_name you want to check
        desired_payment_method = dict["payment_method"]

        # Query to check if branch_name is present
        query = "SELECT payment_id FROM payments WHERE payment_method = %s"
        cursor.execute(query, (desired_payment_method,))
        result = cursor.fetchone()

        if result:
            payment_id = result[0]
            dict["payment_id"] = payment_id
            print(f"Payment method '{desired_payment_method}' found. payment_id: {p

        else:
            new_guid_id_value = str(uuid.uuid4())
            print(f"Payment method '{desired_payment_method}' not found so will be
            dict["payment_id"] = new_guid_id_value
    return list
```

First, we start by looping through each dictionary in the “payments_table” list, and for each one we will look at the column “payment_method” (which will be either card or cash).

Then we save this value to a variable “desired payment method”, after which we will query the database to see if this value already exists.

We use the line:

query = "SELECT payment_id FROM payments WHERE payment_method = %s"

If we find a value in the database (in the payments table) matching this, we then want to check the “payment_id” column for this row and save it to the variable “result”. This will be in the form (value,) and so we assign this value to our “payment_id” column in the “payments_table” list.

If this value isn’t present, then we invoke the “else” clause and assign a new value to it using the GUID function.

This is what our payments table looked like after applying the “payment_id_checker_and_reassigner” function.

[{'payment_id': 'ac942282-e1e8-4a7a-8deb-65705d59f33a', 'payment_method': 'CASH'},{'payment_id': 'c414f59d-a2f4-4a38-8505-017a071cf0bc', 'payment_method': 'CARD'}]

12)Now we need to create the list we will use for the main “transactions” table in our database. For this we use the function create_main_table_list() with the arguments list_to_be_extracted=reformatted_cleaned_data AND products_table= final_products_table.

This will give us all the info we need to load our transactions table,

We shall do this in a few steps.

Step 1: We create an empty list called “big table” and a template dictionary called “data_dict”, then take each dictionary in the “ reformatted_cleaned_data” list, one example:

{'purchase_date': '2023-08-11 09:02:00', 'branch': 'London Camden', 'items': ['Regular Luxury hot chocolate - 2.40'], 'total': '2.4', 'mode_of_payment': 'CARD'}

```
def payment_id_checker_and_reassigner(list, cursor):
    for dict in list:
        # Define the branch_name you want to check
        desired_payment_method = dict["payment_method"]

        # Query to check if branch_name is present
        query = "SELECT payment_id FROM payments WHERE payment_method = %s"
        cursor.execute(query, (desired_payment_method,))
        result = cursor.fetchone()

        if result:
            payment_id = result[0]
            dict["payment_id"] = payment_id
            print(f"Payment method '{desired_payment_method}' found. payment_id: {p

        else:
            new_guid_id_value = str(uuid.uuid4())
            print(f"Payment method '{desired_payment_method}' not found so will be
            dict["payment_id"] = new_guid_id_value
    return list
```

within each of these dictionaries, we will look at the items column and then count the number of things that are separated by the “-” symbol, so in our example above

['Regular Luxury hot chocolate – 2.40'] gives us 2 objects.

If there are 2 objects, then we assign the product name to the first object and the product price to the second.

If there are 3 objects, we assign product name to object one and two (as the second is the flavor) and the price to the third. This is the same process as we encountered in step 4. (Please look there for more info)

Step 2: Once we have done these things, we loop through the products_table list (which has all the unique products we encountered in our csv and their correct product prices and i/ds(guids) and find the corresponding product in the products_table list, then place its unique product_id in the “products_ordered” list of our template dictionary “data_dict”. Then lastly add it’s price to the “products_sum” column of our template dictionary “data_dict” (Which will start at 0. This will slowly append all products that we encounter for each transaction into the products_ordered column (which is now a list), total their sums and do more steps which are to come.

Step 3: The next thing is we will extract the date and time info from the list we are extracting info from, put that information into the “transaction date and time” column of the “data_dict” and then move onto adding the info for payment method and branch name.

Step 4: Then we finally will append this dictionary that we've filled with information using our "data_dict" template to the big table and then repeat for every other dictionary in the list we're extracting information from (" reformatted_cleaned_data"). The final steps will also be to correct its branch id information using the "guid_reassigner" function nested inside this process.

(We also reset the "data_dict" so it considers each new entry in "reformatted_cleaned_data" and a new transaction and a transaction_id +1 higher than the last.

So in the end we will loop over all dictionaries within the " reformatted_cleaned_data") list and with each one extract its data, put it into a different dictionary that we will consider a new transaction.

We assign this new list of dictionaries with our transactions to a variable called "big_table".

An example entry would be:

'Transaction_id': 'd0fdd48d-8549-42c4-97b3-4c27f2526b35', 'Transaction_date_and_time': '2023-08-11 09:02:00', 'method_of_payment': 'CARD', 'order_id': 2, 'products_ordered': ['a27aa12a-2e5d-4a2c-b7d0-1b134c64c8bc'], 'order_sum': 2.4, 'branch': 'London Camden'}

(As you can see, we still have branch information as its name and not its id so we will change that in step 15)

13)Next we will work on the orders_table_list, which is the list used to fill the database table "orders_table". It includes info such as "transaction id" "product id" and "quantity of product ordered".

We will use the orders_table_func() for this purpose, taking arguments table= big_table

```
def orders_table_func(table):#creates list required for orders tab
    list_to_create =[]
    dict = {}
    for d in table:
        y = d["products_ordered"]
        c = Counter(y)
        for x in c:
            dict["transaction_id"] = d["Transaction_id"]
            dict["product_id"] = x
            dict["quantity"] = c[x]
            list_to_create.append(dict)
            dict = {}
    return list_to_create
```

This function forms an empty list "list_to_create" and dictionary "dict", then starts by looping through all dictionaries in our "big_table"list we just created.

It will then isolate the "products ordered" column and count the number of times each entry is present(i.e the number of times Large latte's id is present, and then the number of times Regular Iced Tea Hazelnut is present etc.) and save this information to a key inside the counter function's output (a function native to python if we call it by specifying the module at the top of the .py file). The output will be a dictionary of products and the number of times its counted this product.

We can find out how many times its counted this product (based on its id) by simply looking inside this output and specifying the product. This will leave us will a value and then we assign that to the "quantity" column inside our template dictionary "dict" . We also assign "Transaction_id" and "product_id" from the dictionary we're currently on in the loop from our "big_table". Then it will add all of these to the empty list "list_to_create" and once done will leave us with the list we shall call "orders_table_list".

An example entry from this list is:

{'transaction_id': '8279c81f-4d80-4d46-8be7-ff77f4be1b63', 'product_id': '9688e87a-efc2-4df7-82e4-43b2fd87d55c', 'quantity': 1}

Representing a unique Order and how many of them were wanted. (Multiple orders belong to the same transaction as one person can buy multiple things but they will be processed as separte orders in the system.

14)Lastly we will amend the "big_table" so the fields "branch id" and "payment_id" are correct and have GUID values instead of numbers such as 1 or 2.

So we will use branch_id_reassigner() and then payment_id_reassigner() functions.

branch_id_reassigner was already discussed in step 8 so we will skip to payment_id_reassigner().

It will use arguments =big_table_with_branch_id AND = payment_table.

```
def payment_id_reassigner(list_to_change,list_to_reference):#changes information in list_to_change dependi
    for d in list_to_change:
        search_value = d["method_of_payment"]
        next_list = next(item for item in list_to_reference if item["payment_method"] == search_value)
        id = next_list["payment_id"]
        d["method_of_payment"] = id
    return list_to_change
```

This function uses a similar process to that of branch_id_reassigner but changes the column names to check if our desired method of payment (cash or card) is in the "payments_table" list and if so take its respective payment id then save that to our current "payment_id" column in the big_table list. This leaves us with an output:

{'Transaction_id': '96aeca7a-2348-4e37-95a5-d86bc65b6004', 'Transaction_date_and_time': '2023-08-11 09:02:00', 'method_of_payment': 'c414f59d-a2f4-4a38-8505-017a071cf0bc', 'order_id': 2, 'products_ordered': ['a27aa12a-2e5d-4a2c-b7d0-1b134c64c8bc'], 'order_sum': 2.4, 'branch_id': '0f71f2fa-790b-4769-b7fa-4d6b071a3936'}

We now assign this new amended list of dictionaries to the variable "big_table_with_branch_id" and then are finished.

We are now ready to move onto the Loading phase!

Phase 3 Loading tables:

Now that all the data lists we need have been produced:

1)Branches_list : used for the branches table

2)Sorted_final_products_table_with_branches_reassigned : used for the products table

3)Payments_table : used for the payments table

4)Big_table_with_branch_id : used for the transactions table

5)orders_table_list : used for the orders table

We can use them to fill our 5 tables on Redshift. YAY! (THIS IS THE EASY STEP!)

The functions to load these tables are located in the “sql_utils.py” file,

For 1) Branches :

```
def load_branch_tables(connection, cursor, list): #this will fill all our data tables with the data required
    print("Now loading data into tables....")
    #1 to fill branch table
    for data_row in list:
        branch_id = data_row["branch_id"]
        branch_name = data_row["branch"]

        # Check if the branch_id already exists in the table
        select_sql = f"SELECT branch_name FROM branches WHERE branch_name = '{branch_name}'"
        cursor.execute(select_sql)
        existing_branch = cursor.fetchone()

        if existing_branch:
            print(f"Branch with branch_name '{branch_name}' already exists. Skipping insertion.")
        else:
            # Insert the new row if the branch_id doesn't exist
            insert_row_sql = f"""INSERT INTO branches(branch_id, branch_name)
                                 VALUES('{branch_id}', '{branch_name}')"""
            cursor.execute(insert_row_sql)
            print(f"Inserted new branch with branch_name '{branch_name}'.")
    print("Branch table loaded succesfully!")
    connection.commit()
```

We use the function “load_branch_tables” and take arguments connection= connection, cursor= cursor AND list= branches_list.

The first step is to loop through the “branches_list” and for each dictionary inside (which will hold a unique branch and its unique branch_id GUID, we will take the information just mentioned and then query our Redshift database to see if this value is already present.(if it has been input during the loading phase for a previous CSV.)

To do this we create an SQL query:

select_sql = f"SELECT branch_name FROM branches WHERE branch_name = '{branch_name}'"

Which searches our “branches” table in the database and checks to see if the branch name we have is currently present in the table. It assigns this value to the variable “existing_branch” and then

If it is, then we print a statement to inform us so, I

if it isn’t present then we proceed and use an sql statement to insert the values “branch_id” and “branch_name” into the fields “branch_id” and “branch_name” in our “branches” table.

This is contained in the “else” clause and is followed by a cursor.execute statement to act out this action. We then commit the connection to end that functions job.

We repeat this process for the other tables too!

2)products_table

```
def load_products_tables(connection, cursor, list): #this will fill all our data tables with the data required
    #2 to fill products table
    for data_row in list:
        product_id = data_row["product_id"]
        product_name = data_row["product_name"]
        product_price = data_row['product_price']
        branch_id = data_row['branch_id']

        # Check if the values already exists in the table
        query = "SELECT 1 FROM products WHERE product_name = %s AND product_price = %s"
        cursor.execute(query, (product_name,product_price))
        result = cursor.fetchone()
        print(f"Searching for '{product_name}', with price '{product_price}' and result is '{result}'")
        if result:
            print(f"Product with product_name '{product_name}' and price '{product_price}' already exists. Skipping insertion.")
        else:
            # Insert the new row if the product_name doesn't exist
            insert_row_sql =f"""INSERT INTO products(product_id,product_name,product_price,branch_id)
                    VALUES('{product_id}', '{product_name}','{product_price}', '{branch_id}')"""
            cursor.execute(insert_row_sql)
            print(f"Inserted new product with product name '{product_name}'.")
    print("Products table loaded succesfully!")
    connection.commit()
```

This is similar to the fucnction just used above, but we have more variable that we are inserting into this table. We also query the product_name and product_price to make sure the product is the same and not two of the same but with different prices as this shioud also be noted in our products table if it were the case.

Once we have established our product is not present in the table, we follow the same steps and insert the data into the table. We end this process with the cursor.execute and connection.commit statements to push our requests through and execute them.

3) Payments_table

```python
def load_payments_tables(connection, cursor, list): #this will fill all our data tables with the data requi
    #3 to fill payments table
    for data_row in list:
        payment_id = data_row["payment_id"]
        payment_method = data_row["payment_method"]

        # Check if the payments already exists in the table
        select_sql = f"SELECT payment_method FROM payments WHERE payment_method = '{payment_method}'"
        cursor.execute(select_sql)
        existing_payment = cursor.fetchone()

        if existing_payment:
            print(f"Payment method '{payment_method}' already exists. Skipping insertion.")
        else:
            # Insert the new row if the payments doesn't exist
            insert_row_sql =f"""INSERT INTO payments(payment_id,payment_method)
                    VALUES('{payment_id}', '{payment_method}')"""
            cursor.execute(insert_row_sql)
            print(f"Inserted new Payment method '{payment_method}'.")
    print("Payments table loaded succesfully!")
    connection.commit()
```

Similarity to the steps above, we simply query the database and if the value is already present we skip it. Otherwise, it is inserted.

4)Transactions table:

```python
def load_Transactions_tables(connection, cursor, list): #this will fill all our data tables with the data required
    #4 to fill main table - Transactions
    for data_row in list:
        insert_row_sql = f"""INSERT INTO transactions(transaction_id,date_and_time_id,branch_id,total_price,payment_id)
                VALUES('{data_row['Transaction_id']}', '{data_row['Transaction_date_and_time']}',
                '{data_row['branch_id']}','{data_row['order_sum']}','{data_row['method_of_payment']}')"""
        cursor.execute(insert_row_sql)
    print("Transactions table loaded succesfully!")
```

We take each entry in our Big_table_with_branch_id list which looked like this:

{'Transaction_id': '96aeca7a-2348-4e37-95a5-d86bc65b6004', 'Transaction_date_and_time': '2023-08-11 09:02:00', 'method_of_payment': 'c414f59d-a2f4-4a38-8505-017a071cf0bc', 'order_id': 2, 'products_ordered': ['a27aa12a-2e5d-4a2c-b7d0-1b134c64c8bc'], 'order_sum': 2.4, 'branch_id': '0f71f2fa-790b-4769-b7fa-4d6b071a3936'}

Then we simply insert each value in the dictionary into its respective column using the sql code above.

5) Orders_table:

```
def load_orders_tables(connection, cursor, list): #this will fill all our data tables with the data required
    #5 to fill orders table
    for data_row in list:
        insert_row_sql = f"""INSERT INTO order_data(transaction_id,product_id,quantity)
                VALUES('{data_row['transaction_id']}', '{data_row['product_id']}', '{data_row['quantity']}')"""
        cursor.execute(insert_row_sql)
    print("Orders table loaded succesfully!")
    connection.commit()
```

Similarly, as the above table was loaded, we simply take our "orders_table_list" and use each column to fill a different column in our "orders_table" on Redshift using a singl SQL commands

N.B: There is no need to query the database ahead of time for duplicate values as these lists all produce unique entries of orders and transactions and other CSV files processed before this will be for different days and orders.

Once these steps are done, we simply close the cursor and connection.

With this, the lambda has completed its task and we will be able to see all its print statements and metrics in our Cloud Watch logs!!