Homework Assignment #2

(Apache Cassandra Exploration)

Your boss was impressed with your previous work on PostgreSQL. As the pandemic continues, and as more and more people try the ShopALot service, ShopALot.com's business is starting to boom. Your boss has heard that PostgreSQL can scale up but not out, so she is getting worried about possibly hitting a wall in terms of the size of the supportable customer base. As a result, your boss wants to explore the possible use of Apache Cassandra instead of PostgreSQL for ShopALot's backend database. Your boss wants you to get the company started down this path technically. Your second assignment is thus to prepare a subset of the data to be migrated from PostgreSQL to Cassandra for a POC (proof-of-concept) project, generating different Cassandra data model designs to answer a handful of queries, and then report back about (1) what you did (and how) and (2) the various implications of what you did.

Get Ready…

Core to ShopALot.com's business model are its customers, orders, items, and products, so your boss has identified that subset of the ShopALot's overall schema as a good target for your POC work. She wants you to test drive Cassandra with some basic tables to explore its key-related functionality, which she's heard takes some getting used to; then she would like you to explore its "no joins" (or more accurately, pre-joined) approach to schema design. For data, she wants you to export tables and query results from PostgreSQL into CSV files and then upload them into the tables of the Cassandra keyspace that you will be designing, defining, and querying in the course of your POC work.

Go...!

It's time to get to work. Here's what you are to do using **cqlsh** (Cassandra's command line interface for executing queries):

1. Cassandra is best known for its high scalability and for the variety of options that it provides for high availability (via replication and eventual consistency) and tunable consistency levels for reads and writes. For better or for worse, Astra hides some of these details - so your boss wants you to see what the story is in Astra. Before you start using the ShopALot keyspace and data, you should have created a sample keyspace called **"Hoofers"** to test out the CQL console (as per the setup document). Use the **DESCRIBE** CQL command to retrieve some information about that keyspace. How many copies of the data does the keyspace maintain? Which cloud region does it reside in? Last but not least, what should the read and write quorum sizes (R & W) be for consistent writes and reads in this case?
2. To get you acquainted with CQL (Cassandra's "SQL"-like query language), you should start by translating the PostgreSQL **CREATE TABLE** statements you constructed in assignment #1 for the **Customers, Orders, OrderItems and Products** tables into the equivalent CQL statements, keeping the same table structures as before. This means

using essentially the same data types for each column that you specify (i.e., **`int, varchar, float, timestamp,`** etc…), as well as the same primary keys.
Note: If you made changes to the table data according to the previous assignment, make sure to do a fresh load of the data from the ShopALot csv folder.

3. In PostgreSQL, use the **\COPY** command to export the four tables into CSV files with headers. Once you have the files, use the Cassandra DSBulk utility to upload the data into the Astra tables you created in step 2. (See the setup document for how to export data from PostgreSQL as well as how to use DSBulk). Include the COPY lines in your submission.
4. With all of your data now loaded, let us (innocently) try a simple filter (WHERE) query. Write a CQL query to find the names and list prices of ***10*** **products that have a category of "Meat & Seafood" using the Astra table you made from the** ***`Products`*** **`dataset`** (i.e. limit the results to 10). Did you run into any problems? (Hint hint: you should.) From the resulting error message, use the appropriately modified command to force Cassandra to execute this query.
5. Unlike a relational database system - where we commonly model our data as tables with queries being somewhat of an afterthought - in Cassandra, queries are brought to the forefront. The "force" command used in step four was a "band-aid" solution; instead, you need to come up with a new table design for the Products data. Create a new table with a primary key of ***`category`*** **and** ***`product_id`***, partitioned only on ***`category`***, and load the ***`Products`*** data into this new table. Perform the same query from step four *without* using the same "force" command. Briefly explain why changing the partitioning key changed whether or not Cassandra discourages you from running your query? Also, briefly explain why you had to include ***`product_id`*** in your primary key and not just use ***`category`***? Drop this table once you are done.
6. In addition to being aware of how your data is partitioned, Cassandra users must also be aware of how their data is *sorted*. Suppose your initial filter query from part four now changes - now you are interested in the names and list prices of **t*he 10* most expensive products with a category of "Meat & Seafood"**. Write the new query in CQL and try to execute it (on the initial table). Similar to part four, you should get an error when trying to run your query; unfortunately there is no command that will force Cassandra to run this query. (You can try the previous solution and see what it says. :-)) The solution, again, is to create a new table design. Create a new table with a partitioning key on the ***`category`*** field and clustering keys on the ***`list_price`*** **and** ***`product_id`*** fields, and load the ***`Products`*** data into this new table. (Note: The ordering of your key fields matters here!) Execute your top-10 query on this new table. Briefly explain why adding a new clustering key changed Cassandra's mind about running your query? Similar to before, drop this table once you are done.
7. Now that you understand the implications of primary and clustering keys, you are ready to tackle some parameterized queries that your boss wants you to try in the POC. Create one table for each of her queries. Use the same column names and types you made in part 2) to initialize the tables you will make for these queries. In parallel, export the

corresponding CSV files from PostgreSQL. You only need to include the relevant columns for each query. Here are the desired queries:

   a. List the ***order_id***'s of all orders that were placed by a customer with a given ***customer_id*** of X; the results should appear in descending price (***total_price***) order.
   b. For each order fulfilled by a shopper with ***shopper_id*** of X, find how many unique items were purchased by that shopper in that order.
   c. List the names and categories of all items purchased in the order with ***order_id*** X, ordering the items by price (***list_price)*** in ascending order.
   d. Considering all orders placed by the customer whose ***customer_id*** is X in the inclusive time interval between datetimes Y and Z (***time_placed***), compute the total amount of money that they spent. (Hint: The presence of clustering columns allows range queries as well as sorting to work.)

8. To see the fruits of your data modeling labor, your boss wants you to run some concrete versions of the parameterized queries above. Use the tables that you created in step 7 to create the requisite CQL queries to answer her questions. Here are the queries:
   a. List the ***order_id***'s of all orders that were placed by a customer with a given ***customer_id*** of '24590'; the results should appear in descending price (***total_price***) order.
   b. For each order fulfilled by a shopper with ***shopper_id*** of '0JKLY', find how many unique items were purchased by that shopper.
   c. List the names and categories of all items purchased in the order with ***order_id*** '005SN', ordering the items by price (***list_price***) in ascending order.
   d. Considering all orders placed by the customer whose customer_id is '32976' in the inclusive time interval between datetimes '2020-03-01' and '2020-09-01' ***(time_placed***), compute the total amount of money that they spent.

   Load the data into the new tables and execute these queries. Include the results in your answer.
9. A new order has come in to the application with the following JSON format:

```
{"order_id": "12MDAE", "total_price": 7.14, "time_placed":
"2021-04-10T19:30:24.000Z", "pickup_time": "2021-04-10T21:01:45.000Z",
"customer_id": "24590", "shopper_id": "MQD30", "state": "CA",
"license_plate": "AKM 554", "store_id": "A7ZNF", "time_fulfilled":
"2021-04-10T23:20:56.000Z", "orderItems": [
{"item_id": "9B317", "qty": "3", "selling_price": 2.38, "product_id":
"GMGO5"}]}
```

   Use the CQL **INSERT** command to write the DML query or queries that the application would need to submit (i.e., from the application code on the client side) to place this order in the ShopALot database as it now exists (i.e., including the additional tables that you created in step 7). So, you will need to update both the **base tables** and the **four new tables** created in step 7.

10. **[Extra credit] Cassandra Application Development**

If you'd like to experience what it's like to build a Cassandra application, you can create a small external application that connects to your Astra database. To do so, create a Python script with a function that encapsulates all the logic needed to place an order. Read the **"Connect"** documentation section on Astra to learn more about connecting to the database through the provided connect bundles with Python. Having done this, write a function containing the INSERT queries needed to place a new order (inspired by step 9) and call it to place the following order:

```
{"order_id": "WEQ174", "total_price": 36.47, "time_placed":
"2021-03-29T15:03:20.000Z", "pickup_time": "2021-03-23T17:54:21.000Z",
"customer_id": "6Z53Z", "shopper_id": "MQD30", "state": "WV",
"license_plate": "0031", "store_id": "ZU9IP", "time_fulfilled":
"2021-03-23T22:43:12.000Z", "orderItems": [
{"item_id": "PO12C", "qty": "7", "selling_price": 5.21, "product_id":
"GMGO5"}]}
```

Submit your Python script as a part of this assignment for extra credit!

What To Turn In

When you have finished your assignment you should use Gradescope to turn in a PDF file that lists all of the CQL statements that you ran - from start to finish - to create the tables and to run all the queries. Include the results of queries where asked. Please follow the following steps in order to generate the file for submission.

1. Open the Google Doc Template file, Click [File] -> [Make a Copy] will create a copy of this template. You can edit it and once the editing is done, you can download it as a PDF file and submit it to Gradescope.
2. Stick to the solution template and use the same name for the keyspace and table names as the previous assignment. For newly generated tables, you may name them the base table name + "q" + the number of the question, e.g. "*customersq5*". If the new table is a JOIN of two tables, you may name it tablename1 + tablename2 + "q" + question number.
3. Fill in your responses. Include the non-query answers to questions where specified. If you did not answer a question, leave the designated space empty.
4. For query results, you can either attach a screenshot in the designated space, or paste the table results directly from cqlsh.