Homework Assignment #2

(Apache Cassandra Exploration)

As more and more people try the ZotMusic service, ZotMusic'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 ZotMusic'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 ZotMusic's business model are its *users, records, sessions,* and *reviews*, so your boss has identified that subset of ZotMusic'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 was long known for its high scalability as well as for the variety of options that it provides for high availability (via data replication and eventual consistency) and for offering tunable consistency for reads and writes. For better or worse, Astra now hides some of these details – so your boss wants you to see what the story is today in Astra. Before you start using the ZotMusic 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 Hoofers 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 when working with this keyspace?
2. To get you acquainted with CQL (Cassandra's "SQL"-like query language), you should start by translating the PostgreSQL **CREATE TABLE** DDL in the ZotMusic domain from HW1 for **users, records, reviews,** and **sessions** into equivalent CQL statements, keeping the same table structures as before. This means using essentially the same

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

3. In PostgreSQL, use its COPY INTO statement to export the four tables of interest to your boss to CSV files with headers. Once you have the four files, use the Cassandra "DSBulk" utility to upload the data into the Astra tables that you created in question 2. (See the setup document for information on how to export data from PostgreSQL as well as how to use DSBulk). Include these PostgreSQL COPY INTO statements in your submission.
4. With all of your data now loaded into Astra, let us (innocently) try a simple filter (WHERE) query. Write a CQL query to look up the record_id, title, and genre of the records released by the artist with user_id '`user_6ac27408-a0a6-4c57-a025-7b6854f7a8e3`'. Use the Astra table that you made from the ***records*** dataset. Try running your query "as is". Then, based on the resulting error message, use an 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 the queries being somewhat of an afterthought – in Cassandra, queries are brought to the forefront. The **"force"** command used in question 4 was a "band-aid" solution; instead, you need to come up with a *new table design* for the ***records*** table. Create a new table with a primary key of ***`artist_user_id`*** and ***`record_id`***, partitioned only on ***`artist_user_id`***, and load the ***records*** CSV data into this new table. Perform the same query from question 4 *without* using the same "force" command. Briefly explain why changing the partitioning key altered whether or not Cassandra discouraged you from running your query. Also, briefly explain why it was necessary to include ***`record_id`*** in your primary key and not just use ***`artist_user_id`*** for the key.
6. In addition to being very aware of how 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 – suppose now you are interested in the record_id, title, and release_date of the 5 **most recently released** *(release_date)* records by the user with the id '`user_bab3f848-261f-4056-a865-4f01793058a3`'. Write the new query in CQL and try to execute it – on the initial table of records and also on the new table from question 5 – and see what Cassandra says. Similar to question 4, you should get an error when trying to run your query. Unfortunately, this time there is no command that will force Cassandra to run your query. The solution, once again, is to create a new table design. Create a new table with a partitioning key on its ***`artist_user_id`*** field and clustering keys on its ***`record_id`*** and its ***`release_date`*** fields, and load the ***`records`*** data into this new table. (*Note*: The ordering of your key fields matters here, so you will have to order them appropriately.) Execute your top-5 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 new table and the table in question 5 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! **For each of her following four queries, create one table that can support such a query.** Use the same column names and column types as in question 2 above when defining the tables that you create for these queries. In parallel, run PostgreSQL SQL queries to produce the necessary data, export their results as CSV files from PostgreSQL. Note you might need to join multiple tables from PostgreSQL, and you only need to include the relevant columns for each query in your CSV exports. Here are the desired queries:
    a. List the ***review_id*** and ***record_id*** of reviews posted by a particular user with **user_id** and return the records in the descending order of rating.
    b. Count the number of records released with a particular genre of X.
    c. List the review_id, record_id, title of record, and rating of all the reviews posted for records released by an artist with an ***artist_user_id*** of X, ordering your results by the ***posted_at*** time of the review in descending order.
    d. Considering all sessions initiated by a user whose ***user_id*** is X in the (inclusive) time interval between datetimes Y and Z (***initiate_at***), compute the highest ***replay_count*** in these sessions.
8. To see the fruits of your data modeling labor, your boss wants you to run some concrete versions of the parameterized (X, Y, Z) queries above in Astra. Use the tables that you created in question 7 to create the requisite CQL queries to answer her questions. Here are the test queries along with the query parameters to use for each one:
    a. (For 7.a) List the ***review_id*** and ***record_id*** of reviews posted by a particular user with a **user_id** of "`user_9e48cbb4-0bf9-43fc-a578-213fae51068b`"; the results should appear in descending order of ***rating***; print only the first 10 such results.
    b. (For 7.b) Find out how many records have been released in the genre “Folk".
    c. (For 7.c) List the **review_id**, **record_id**, **title of record**, and **rating** of all the reviews posted for records released by the artist with a ***user_id*** of “`user_6f33f39e-7659-4673-bd80-ca11394424b0`”, ordering your results by the ***posted_at*** time of the review in descending order, and show only the first 10 such results.
    d. (For 7.d) Consider all of the sessions that were initiated by a user whose ***user_id*** is ‘`user_05f9132b-47fb-4d2b-992c-17b3c4afb2df`’ where the sessions were initiated between (inclusive) datetimes ‘2024-08-01 00:00:00’ and ‘2024-09-01 00:00:00’ (***initiate_at***), compute the highest ***replay_count*** in these sessions.

    Load the CSV data from question 7 into the new tables using dsbulk and then execute these queries. Include the results in your answer.
9. A new record has come into the application layer of ZotMusic with the following JSON content:

```
{"record_id": "record_d2f498f8-d7ff-4f1c-a967-7090417751f5",
"artist_user_id": "user_38eaa9f8-e8fc-4ce4-a8ae-ffb882c1786c", "title": "Blue
By You", "genre": "Rock",
"release_date": "2024-10-07"}
```

Use CQL's **INSERT** statement to write the DML query (or queries) that the ZotMusic application would need to submit (i.e., from ZotMusic application code on the client side) to add this record into the ZotMusic database as it currently exists (i.e., including the additional table(s) that you created in question 7). Note that you will need to update both the **base table(s)** and the corresponding **new table(s)** created in question 7.
After inserting this new record, verify you have successfully done so by writing the appropriate query (or queries) in CQL and show the output(s) in your answer.

10. **Cassandra Application Development**

You'd like to see what it's like to build a Cassandra application, so next you have decided to create a small external application that talks to your Astra database. To do so, create a Python script with a single function that encapsulates all the logic needed to add a new record. Read the **"Python Driver Quickstart"** documentation on Astra to learn more about connecting to the database through Python APIs.
Having done this, write a function that contains all the INSERT queries needed to add a new record (inspired by question 9) and call your function to add the following record:

```
{"record_id": "record_632fe768-eecb-4596-9780-cc21734feec5",
"artist_user_id": "user_b91cf915-487b-42fc-b6b8-6c17935bb755", "title": "One
Sour Day", "genre": "R&B",
"release_date": "2024-10-07"}
```

When you have succeeded, include your Python script as a part of this assignment. (How hard can that be? :-))

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 of the queries. Include the results of queries where requested. Please follow these instructions in order to generate the PDF file for HW2's submission:

1. Download the HW2 Template file from Canvas and edit it as you go, and once the editing is all done, you can export it as a PDF file and submit it to Gradescope.
2. Stick to the solution template and use the same names for the keyspace (ZotMusic) and for the table names as in the previous assignment. For newly generated tables, you may name them by combining the base table name + "q" + the number of the question, e.g. "*records_q5*".
3. Fill the template in with 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 you may paste the table results directly into your HW2 document from cqlsh.