Thu Anh Le
CS327E

TECHNICAL REPORT

PROJECT OVERVIEW

The project aimed to build a data warehouse in RedShift to collect information about contemporary popular music. The datasets that were used were Million Song and Music Brainz. These two datasets have different formats with a few common attributes that allow opportunities for joining. In addition, these datasets also have several unique attributes that will help to expand the amount of information collected.

The two Million Song and Music Brainz datasets will be brought into a single database. From there, data tables within each of these datasets will be collected and joined within a unified schema. Once a unified schema is created, interesting questions about contemporary popular music can then be answered.

I. DESIGNING THE DATABASE

a. Process Summary

Datasets used: MusicBrainz, Million Song

End-products: physical diagrams, unified schema diagrams, data dictionary, English queries

b. Approach and Challenges

- **Prioritizing columns**

It was difficult to think about what columns were needed and what were not. There were a lot of columns, and it was required for us to read through the data description to understand how they were related to the tables. There were two approaches that I considered:

- ***Approach #1:*** write out the English queries and determine exactly what would be needed
- ***Approach #2:*** Think about the most common business problems and determine what columns will likely be used in most circumstances.

I decided to go with approach #2. My reasons were that it would allow the database to be more robust, and it would be easier for me to change queries in the future if needed with the least amount of time and effort.

- **Unifying data from different sources**

This is perhaps the most interesting challenge in the final project. Since the two datasets selected were from different sources, it was fun trying to form connections between them. At first, I mistakenly thought that the global id for all tables should match. However, I realized that it was important to read the descriptions and see if the joins made sense. A few column ids matched well with each other, whereas the other ones did not.

A few tables also seemed to only work in couple with another. Therefore, I had to pick out a main table with all the data points needed and connect the smaller tables through it.

A sub-challenge was also choosing a "bridge" table that will allow joining the two different datasets together. Luckily, the "Songs_Summary" table from Million Songs had most of the data needed with ids, and it could be joined with Music Brainz by track names.

- **Writing English queries**

Since my design process was somewhat different from the original instruction, I encountered this dilemma of trying to be creative with the technical details at the back of my head. I think this limited my ability to come up with better queries and analysis.

II. LOADING THE DATA INTO REDSHIFT

a. Process Summary

We were given a generate_DDL.py script to generate `create table` statements. Then we created a separate schema for each dataset. Within each schema, there will be tables that we selected from the physical diagram.

Each dataset had their own DDL script for reproducibility. The scripts worked as follow:

+ Check if the schema exists, if not, drop the schema

```
DROP SCHEMA IF EXISTS millionsong;
Create SCHEMA millionsong;
```

+ Create tables in the schema by pasting results from the python script over

Once the tables have been created, the copy command was used to load data tables from the given .csv files. The copy statements were put together into each dataset's script, then pasted into the terminal.

After all the staging tables were loaded, records from each table were sampled to check for validity. The output of each query was stored in a text file.

End-products: DDL scripts, copy scripts, check scripts, data tables within Redshift

b. Approach and Challenges

- **Changing the python script to generate DDL scripts**

Originally, the python script we were given output each `create table` statement into a separate .sql file. How the file path was coded in the script also made it difficult to find the .csv files correctly. Therefore, a few tweaks to the script were made so that all `create table` statements were printed nicely into one .sql file according to their dataset, as well as having the encoded paths point to the correct file paths. These small changes have increased work efficiency by about 50% for this specific task.

- ***Lines changed:***

[27] sql_file = path + slashes + 'create_' + name + '.sql' to

sql_file = 'create_' + 'dataset_name' + '.sql'

[36] with open(sql_file, 'w') as out_file: to

With open(sql_file, 'a') as out_file

- **Cleaning up the `create table` statements**

Before setting up the tables in Redshift, the .sql scripts were given a run-through to make sure that column names and types were generated correctly. A lot of the column types were understood as `boolean` or `int` instead of `varchar`. Most column names were correct.

- **Loading the tables**

At first, all the `create table` statements worked well. However, when transferring the data from the .csv files over, a few columns exceeded the character length limit and could not be loaded successfully. When this happened, I ran the diagnostic query, dropped the table, set the length limit higher, ran the `create table` statement again, and loaded the data until the error no longer persists.

- **Miscellaneous issues**

The first tables were created in the wrong schema at first because I forgot to set search_path. Headers needed to be switched to `FALSE` when copy over.

III. TRANSFORMING THE DATA

To prepare the data for cross-dataset analysis reporting, data needed to be transformed to join together and ensure consistency.

a. Process Summary

Analysis queries were reviewed to identify columns that will be used in dataset joins and query operations. Then, uncleansed columns within the tables needed were substituted with cleansed columns in the unified schema.

After that, the analysis queries were translated to SQL using the data from the unified schema. The queries were reviewed to make sure that the output made sense and were suitable for reporting.

b. Approach and Challenges

- **Standardizing the data**

A lot of our data points contained punctuation characters such as ;, /, (, [, :, -, ..., with, Vs. and needed to be removed so that they can be joined with other columns across datasets. To see if the columns contained punctuation characters, 100-200 records from the staging tables were sampled randomly. If the columns that contained punctuation characters, a new ccolumn with cleansed data were added. This column was first created as a copy of the original column.

However, before that, it was important to determine the length of the column through a given user-defined function in python to minimize burden on the database memory capacity. When the data points were copied over, punctuation characters were removed using a combination of split_part(), btrim(), and initcap().

Here’s a sample transformation statement that was used:

```
select max(get_utf8_bytes(title)) from songs_summary;
alter table songs_summary add column ctitle varchar(285);
update songs_summary set ctitle= title;
update songs_summary set ctitle = initcap(btrim(split_part(ctitle,'-' , 1)));
```

A few issues that could occur from this method was getting false positives. For example, artist names could be very similar without the punctuation characters and therefore would cause the aggregated results to be much higher that they should be.

Later on, I noticed that removing characters after the '.' in a few columns removed too many characters and in turn, entirely destroyed the purpose of the columns. I had to go back, recreated the columns and thought about my design again.

- **Creating the unified schema**

At this point, it was basically a matter of grunt-work going back to the unified physical diagram and the data dictionary to see what columns were needed for the unified schema. The tables within the unified schema were created based on a subset of the columns using the table-select statement. This statement also pulled the cleansed column and renamed it to the original column's name. The tables were also kept track of by initializing the dataset that they belonged to at the front. Here is an example:

```
create table unified.mb_area_alias as select id,area, cname as name
from musicbrainz.area_alias;
```

These statements were then saved to a create_unified.sql script for reproducibility. Doing this has proved to be necessary for the next steps.

- **Writing the SQL queries**

It was fun writing the English queries and imagined all the possibilities that could be discovered with this wealth of data (an ear-cancer cure based on how many times a song was played, perhaps?). However, due to time and resources (mostly my own limitations and the amount of data points), a few queries had to be simplified and a few queries had the opportunity to be improved.

A few queries seemed simple enough at first, but they actually took a lot of joins and work-arounds to produce successful outputs. For example:

English query: Percent of recordings that are Official, Promotion, Bootleg, Pseudo-Release (release status)

SQL query:

```
select status, count_occurences, cast(count_occurences as decimal) / (select count(*) from (
mb_recording record
join mb_release release on record.id = release.id
join mb_release_status status on release.status = status.id)) * 100 as percent_of_total_occurences
from (
select status.name as status, count(*) as count_occurences
from mb_recording record
join mb_release release on record.id = release.id
join mb_release_status status on release.status = status.id
group by status.name
) as sub_query
group by status, count_occurences;
```

Furthermore, during the process of writing these queries, I discovered that the song_id columns within the Million Song dataset needed to both be uppercase to match. I went ahead and wrote a transform statement to make the song_id columns uppercase.

IV. VISUALIZING THE DATA

a. Process Summary

Virtual views for each analysis query developed were created in the unified schema. Then, runtime of the views were checked to make sure that they will not take too long to render in Amazon Quicksight.

After virtual views were created and optimized, tables will be loaded as a data source into Quicksight.

b. Approach and Challenges

- **Creating the virtual views**

After creating the virtual views, I checked the run-time through all of them and found that a few queries took too long to operate. I had to go back and optimize these queries. The approach I used was breaking down the select statements into smaller tables.

- **Visualizing the data in QuickSight**

It was required to import each table view into a separate data source. The name of the views were initialized with a v_ at the beginning (ex: v_recording_summary) for easy searching in QuickSight.

QuickSight made it very easy to visualize the data in simple graphs. However, the platform does not allow more advanced modifications such as flipping axis, changing field names from the UI, combining datasets, changing colors, or combining bar and line graphs together. It is probably a good resource to create quick visualizations, but it would not be my first choice if I plan to create something more complex.

It was also difficult for me to grasp the concept of QuickSight "Stories" and how to create multiple charts within one story without having to open many tabs at once.

Additionally, a few adjustments had to be made to the analysis queries for visualization purposes. Specifically, the queries that had to be adjusted were average count of words in a song, average duration of songs, and records that come with cassette packaging. For the records that come with cassette packaging, there were too many of them and thus making it impossible to visualize, I went back and altered the query to only display 10 random records with cassette packaging.

V. WRITING THE TECHNICAL REPORT

a. Process Summary

The technical report documents my experiences with the final project milestones and challenges encountered.

b. Approach and Challenges

I decided to break this report down based on the order of milestones. Each milestone will include two sub-parts that are a.) Project Summary and b.) Approach and Challenges. The process summary is intended to briefly summarize what the milestone is about, and the approach and challenges part is intended to dive in-depth into the smaller processes, experiences, and challenges encountered.

VI. CONCLUSION

a. Lessons learned

- Read data description and documentation carefully
- Always perform quality checks
- Optimize when possible
- Take time to develop a thoughtful design before implementation

b. Unexpected results

- A few columns, even though with seemingly similar description, did not match
- There is no correlation between the number of songs produced and the number of times they are played by each artist

c. Unsolved problems and other open issues

- The artist names need to be cleaned by a better method. Right now, a lot of the names are exactly similar due to the cut-off and thus not returning good query results
- A few parts within the database can still be optimized (more derived tables can be created)