Final Report - Music Database Project

Team Databased

Tolan Nguyen and Blake Chicoine

Introduction:

To thoroughly describe our experience through this project, it is necessary to describe the challenges we faced, recount our approaches to the challenges, the issues we encountered, and the final solutions we came up with. In a nutshell, we were challenged to face several hundred .csv files, tasked to design a database to store the data in Redshift, assigned to query the data, and then challenged to construct meaningful visualizations for our data. With the given music data from MusicBrainz, Discog, and Million Songs, we found a wealth of information that could be gleaned from the data, as well as several issues that presented themselves before us.

M1:

Before starting the project, we had to first create AWS accounts to use the tools we needed to build our databases and run our queries. This section was the simplest to complete, and gave us a chance to mentally prepare for the challenges ahead. In setting up our instances, we decided to set up the Redshift instance on one account, and shared the connection information through Stache. We set the IP range to match the University's IP range so that either of us could connect to Redshift from anywhere on campus without having to change IP settings every time. This saved us from

headache later on during the rest of the project, and we encountered no other issues related to set up.

M2:

At the beginning of the project, we had to first sort through and determine which datasets would be interesting for us to develop a database for and to analyze. Several issues popped up for us in determining which datasets we wanted to use. First, we had to find the data we wanted use, which was basically Artists, Releases, and Songs. Then another determining factor for us was the future ease of inserting the dataset into a database. One of the main problems we wanted to avoid was unnecessary complexity in the database design, and the Million Songs dataset had several .csvs that would fold into a single database, which we feared would complicate the database, without adding value to the data we were interested in. This led us to choosing the Discog and MusicBrainz dataset for our final choices.

From the datasets we chose, we had to then narrow down the exact files we wanted to use in our database. To do this, we picked from the file names that seemed interesting to the queries we wanted to perform. After picking 75% of the files we wanted, we went back to determine if there were files that were necessary to join tables together, and included these in our selection. Unbeknownst to us, we had copied the selected subset files to another directory to keep track of the data we wanted to use, which turned out to be a smart choice for later milestones in automatically creating the tables and loading the data.

After selecting the data, we spent a large amount of time designing the database structure in the form of diagrams. While we did not stumble on any major issues in this section, we did struggle to determine relationships between datasets. Some datasets were obvious in how they connected together, such as some tables having join tables in between that indicated a multiple to multiple relationship, which made things somewhat simpler. Some were more difficult, such as us not knowing if multiple aliases could belong to multiple artists. Connections like this we determined to only be determinable from the full dataset, which we did not have yet. In these cases, we left them as single to multiple relationships.

Once the individual staging diagrams were done, we copy and pasted them together to determine how we wanted to create a unified schema. In doing so, we determined that IDs from the datasets would not be able to connect the Discog and MusicBrainz datasets, so we would need to connect between data sets by name. After the unified schema was completed, we entered the data in a data dictionary, along with notes on the connections between pieces of data to help us later on in later milestones.

From milestone 2, we learned a lot about being careful with diagram and database design. By having a solid foundation in our original design, we streamlined the rest of the process for creating our database. In addition, we accidentally learned about the value of organization as well, where we sorted the files we planned to use in another directory to later use that allowed us to come back and easily sort through what we needed.

M3:

For milestone 3, we need to generate tables for each .csv that we selected, and then load the data into the tables. To do this, we were provided a Python script to generate DDLs that captured the column names and data types from each .csv to create a table, and code to generate schemas. However, the first problem we saw was that we would have to run the script 30 times and analyze the results every time. To remedy this, we edited the generate_ddl.py to run across every file in the directory that we had saved our .csvs in. This created many smaller .sql files, one for each .csv. In order to scan the directory, we used the glob library in Python. This allowed us to list the contents of the full directory. From there, we scanned the directory again to find the smaller .sql files to combine in one large create_tables.sql file. At the beginning of the main file, we inserted the search_path based on file names and the create schema code. In the end, this saved us about 50% of the time we would've needed to manually run the generate_ddl.py scripts for each table we needed, though the final script needed additional editing to ensure the data would fit. By batching together the generate_ddl.py and separating the .csvs at the beginning when we were deciding which ones to use, we managed to save a lot of time and headache on creating tables for this milestone.

After saving time through aggregated Python scripts, we still needed to manually edit the large file by padding the character lengths in the DDL. The original pulled character lengths were not long enough which would have caused problems in loading data, so we used a simple find and replace in a text editor to add 0s. Another thing we needed to edit in the final DDL scripts were the data types that the generate_ddl.py

gave us. Many of the ones generated were incorrect, such as labelling integer values in ids as boolean. For these, it was also a simple find and replace to change them to integer. Preemptively scanning through these files helped us save some time later on when loading data, so that we had fewer errors after waiting for data to load.

The next step in this milestone was to load the data into the tables and schemas we created. For this part, we were only given code to load tables, but not a script like the generate_ddl.py. This time, we created another Python script that used also the glob library to identify our directory and files again, and created load statements to load data into our tables. By using regular expressions, we were able to extract the file name and table name to insert them into the the copy scripts. For example, we extracted the “artists” from artists.csv, and then located the folder name as Discog, which set the S3 bucket that we pulled the data from. The rest of the copy command was then pasted in. After every file was parsed, we inserted it into the final loading file and began to load data.

When we started to load our data, we found an error in the S3 bucket permissions where we could not load the data at all. Professor Cohen had to fix it real quick and we were up and running again shortly after. Thankfully we only found a few errors in loading data afterwards. To resolve these errors, we had to go back and fix errors in our table creation code. Since we automatically generated the DDL, the only errors we found were the ones we put in by editing the code, like when we were padding 0s to character values. Occasionally we lost a ; or dropped a word which caused the errors we had. It just goes to show that by automating and coding the

generation of the script, we had fewer errors than if we had hand coded it, and by hand coding the edits we introduced new errors.

One side step we used during the table creation and data loading was to enter a lot of data as characters for simplicity. While not memory sparing, this saved us time to create our staging tables and allowed us to more easily convert to integer or dates later on through transformation.

During this milestone, we learned a lot about both creating DDLs, and how to save time by batching together Python code to automatically create the DDLs. The automatic creation of code ended up saving us several hours of work by preventing random human errors and by rapidly creating code instead of manually copying and pasting code, and adding in new fields.

M4:

Next, for milestone 4, we were tasked to create a unified schema between the two datasets and translate our queries to sql in order to run them over this combined schema. However, the raw data could not simply be joined together and queried on due to inconsistency issues in the columns, so we first had to deal with these punctuation discrepancies. Finding the columns in need of standardizing proved to be a tedious task as we had to search random values in each column and check if they contained any punctuation characters or words that would prevent the data from being integrated without issue. This was especially a struggle because we needed to query random samples from each table, and then check for odd punctuation. Sometimes we did not

find any errors due to the random sampling, but in later queries we did find punctuation and had to add these columns to our transform list.

Next, in a transform sql script for both datasets, we created and populated new “ccolumns” for each column that needed standardizing. To do this, we first defined a user-defined function which finds the maximum characters in a row for a certain column. Running this function on each column in need of standardizing gave us the appropriate size for each of these new columns. Then, for each column, we used split_part(), btrim(), and initcap() to remove the substring for every possible word or punctuation that would cause inconsistencies. Even though some columns only had one or two rows with punctuations, we ran the trim for every punctuation on every column in need because it doesn’t harm the data and this method is much quicker than combing through all the data and finding which specific punctuations to change. We also had to account for inconsistencies that arise with date columns, as there are many different ways to write the date. In the end, we did not convert the dates to the date data type. Instead, we kept them as characters, and used only the year they popped up as. When we searched through some of the dates, we had some strange entries that would’ve given errors if we had tried to change the type, such as finding “UK” as a year. By leaving the dates as year and char(4), we could still easily query the data and only traded off a small amount of memory.

After the punctuations were removed and the data was standardized, we were then able to create a useful unified schema. In our create_unified.sql script, we added only the tables and columns from musicbrainz and discog that were necessary in order

to answer our queries. We simply copied and pasted the relevant tables from the Discog and Musicbrainz create_tables.sql scripts and combined them into a single create_unified.sql.

Finally, after creating our unified schema, it was time to translate our text queries into sql. Right off the bat we ran into an issue regarding a query involving countries which we soon realized was caused by missing tables in our unified schema. We had created the table for musicbrainz release_country, but had overlooked the tables for musicbrainz_area and musicbrainz_country_area so we updated the create tables sql to include these. Another issue we had was the fact that our text queries were made without much forethought on how they would translate to sql. Due to limitations in the data, we had to change some of the queries that were either too complex to translate, or simply didn't result in any useful data. For example we changed the query ""Which artists have the most diverse use of release fomats/ mediums?" because the single result that it produced was not very interesting and quite difficult to generate from a SQL query.

Milestone 4 showed us the tedious process that is combining multiple datasets into one unified schema in order to query over it. By first having to search each column to see where inconsistencies might arise and then going through and changing each punctuation and date format in order to create a standardized unified schema, we realized the importance of uniform formatting and the implications if there are conflicting formats.

M5:

For milestone 5 we wrapped up our study on the two data sets Musicbrainz and Discog by creating virtual views and representing these views visually. When creating our views, we realized a mistake was made in our sql script on the query “What is the most common name of artists?” where it returned multiple columns. We found the error and updated the queries.sql script accordingly to only return one column.

For each view, it was necessary to check the runtime of the query to ensure it didn’t take too long and cause frustration for the user. Our longest query turned out to take 2735 milliseconds, or 2.735 seconds, which was much quicker than our goal of each query taking less than 30 seconds to run. Our quick speeds were a result of relatively small, specific queries searching our trimmed unified schema. Most of the queries we used had included where or having clauses which kept the queries quite specific. We were very fortunate in this step as we did not need to perform any optimizations in order to speed up the process.

After the virtual views were complete, it was time to represent our data graphically so the results of our queries could be easily interpreted. Thanks to Amazon QuickSight, our virtual views were quickly converted into graphical representations where we then determined which type of graph best fit our query. Many of our queries were simply represented by graphs while others were more difficult to find a meaningful representation. For example, our two queries involving differences amongst genders were easily represented by bar graphs with “male” and “female” being the two variables on the x-axis; however, our query viewing the newest releases in the United Kingdom

was a bit more difficult to represent due to the amount of releases combined with very long names. Another issue we had representing our queries graphically is the query “Which areas have releases in the hip-hop genre?” simply results in a list of areas with no differences in quantity. Because of the lack of variation in the results, it is difficult to create a meaningful graphical representation.

Our work creating views to represent graphically through QuickSight proved worthwhile as our data and queries can now be interpreted by someone with no prior knowledge of the assignment. Milestone 5 taught us the importance of graphical representations and how easily they can be created with Amazon QuickSight.

Conclusions:

In the end, we learned a great deal about database querying and integrating in doing this final project. We discovered many benefits of automating scripts, as well as creating and debugging SQL. It was a very educational experience that allowed us to learn about the AWS ecosystem, and one that we can bring with us into our future careers.