Final project report, CS327E

Carl Jonsson, coj228

Surya Suravaram, srs3975

1 Introduction

1.1 Goal

The goal of the final project in the course *CS327E Elements of Databases* is to create a data warehouse by unifying multiple databases into one, ask interesting queries and create visualizations representing the answer to those queries.

1.2 Data Sets

Three different datasets were given, all containing music-related data. There was the *Discog* dataset, made up of 8 files focused on album releases and related data. We also had the *Millionsong* dataset which was 11 files containing some less fundamental data such as lyrics and song popularity. The final dataset was the *Musicbrainz* dataset which contained 78 files. The information ranged from basic release data to e.g. descriptive tables on instruments and areas (that artists might originate from).

For each of the datasets, we downloaded a smaller subset of the data for initial analysis and planning.

1.3 Tools

A lot of the tools used are associated with Amazon Web Services (AWS). The datasets were stored in S3 buckets, and Amazon Redshift was used for the data warehouse. We made database design diagrams in Lucidchart and a data dictionary in Excel. The queries were written in SQL, and Amazon Quicksight was used to visualize the queries.

Git was used for version control, project planning and issue tracking.

Stache was used for credentials sharing.

2 Analysis

2.1 Setup

The project started with setting up a folder in an already existing Github repository. Most of the files in the project would be placed in this root folder, but in some cases (e.g. with multiple similar scripts) additional subfolders were used. The projects were separated into 5 operational milestones with weekly deadlines, which was incorporated in the Git issue tracking.

After that, we decided on how to set up the teamwork. Although we didn't use the issue assign feature in Github, in most cases we could work separately on different issues. We didn't use the feature because some issues would be better performed by working together. One example is writing the SQL queries, where pair programming in general is viewed as a method to minimize errors. The team strategy was to first use the assigned class hours, and meet up on further occasions based on need.

We created a role in AWS's Identity and Access Management (IAM) to create a Redshift cluster within. After creating the cluster, we opened the security group to traffic from the IP address we were working from.

2.2 Database Design

With the three given data sets, we picked 30 files of data from MusicBrainz and Discog that we found were interesting. We had realized that much of the data was unnecessary, because there were ambiguous or meaningless tables and fields. So, we had deleted unnecessary information from the physical diagram by removing irrelevant fields like "gid", "parent", "child_order", and etc. From this information, we were able to construct conceptual and physical diagrams for each of the two

schemas. Doing so allowed us to see all the tables and specific fields that we could comprise queries from.

Afterwards, we created a unified schema to illustrate cross-schema relations. Surprisingly, we found that there were few connections between cross-schema tables, especially since we expected music data to have lots of like-kind information. For example, tables such as gender, series, and work were in the Musicbrainz dataset while they were not in Discog. Hence, there were a lot of important relationships that could not be made due to the fact that one schema would have missing data while the other had extra tables rife with data and vice versa. Additionally we could not easily connect tables, because certain tables were structured differently in one scheme versus the other. For instance, the Release table in Discog had information regarding the release data and country of release all in one table whereas Musicbrainz also had the same information but was split across three different tables: release, release_country, and release_unknown_country. Nevertheless, we made the limited number of connections between them as seen in the Unified Physical Schema in Exhibit 1. We would like to note that Exhibit 1 does include some of the additional changes we made during the Data Analysis and Visualization process, as four of the original 30 tables were removed since they were not used in the queries.

As we finished our visual diagrams of the unified schema, we had a good idea of how the data was all interconnected. This really helped us formulate our ten queries. We wanted to perform a number of cross-dataset joins, so that we can take advantage of the massive amounts of data from both datasets. The approach we took was to analyze major commonalities among both sets such as artists, releases, countries, labels, mediums, and dates. With these in mind, we considered a range of

intriguing queries to analyze. However, we also wanted to include queries that used unique information in each schema such as works and gender since they are also very relevant that can be used to answer important questions. Our finalized list of queries can be found in Exhibit 2.

2.3 Data Load

When we had concluded which 30 files we would need to be able to design a database exhaustive enough to answer our queries, we wanted to load the data from the S3 buckets into our Redshift cluster. We loaded the data into two separate staging tables, to later make a unified while transforming the data to our needs. We created a schema for each of the two datasets, and created one main DDL script for each of the schemas. The main DDL script consisted of a series of commands executing one copy command for each file.

From our experience from the data copy lab in the course, we had to rerun copy commands several time due to unexpectedly long character fields. To prevent this, we chose large field sizes (up to 5 000 characters) right away as to not have to run the copy commands several times due to this error. However, we realized that Redshift's copy command was a lot faster than Postgres due to its node delegation system, so we could actually have tried smaller sizes without losing too much time. In the end, it didn't really matter since we had a lot of space available in the cluster.

We also made some fields as varchar types, even if they were supposed to be integers or even boolean. This was because we noticed some fields contained e.g. "\N" instead of just blank or null values, and we wanted to not have errors in the copy commands for the same reason as above. Instead, we altered this in the data transformation.

Whenever we ran into problems with the copy commands, we queried the table “stl_load_errors” which would tell us for which column the type of the data hadn’t matched the ones we had set. We found several cases where special characters (or rather, letters) appeared in columns that had been exclusively integers in the subset files that we had downloaded to base our copy commands on. In those cases, we instead loaded the columns as varchar and fixed the data types in the transformation step.

2.4 Data Integration

After we had loaded the raw data from the S3 buckets into our Redshift cluster, we wanted to clean up the data for two reasons. The first reason was to standardize formats to be able to unify the datasets on the overlapping data. This enabled us to find answer to queries where data from both datasets were needed. The second reason was to clean up the data with a few unexpected entry data types, to make the final unified tables query friendly.

To enable the cross-dataset queries, we tried a method of removing “punctuations” in name fields. “Punctuations” included special characters such as ‘,’, ‘/’ and ‘-’ but also words such as ‘with’. This would in theory give us more matches between the datasets, but would also give us some false positives. We never made a thorough analysis of the effect of this, which should be a part in a more comprehensive project.

As described in the data loading section above, we had some columns created as varchar when they should in fact be integers. In this step, we corrected that. We created additional columns with similar names apart from adding an initial *c*, so the columns would be called e.g. *ctitle* if the column was called *title*. We then copied the entries from the original column, but casted the data as integers.

Another crucial data transformation was converting our varchar date fields into date type fields. This was a challenge since both data sets had different types of information in their date columns. For instance, Discog had one column for release dates, but there were such varying values ranging from “1993-03-00” to “11/2/1999” and from just the year “2000” to blank values. Transforming such messy data to a clean date type was difficult, as we had to do separate transforms for each of the 4 cases: dashes, slashes, missing day & months, and null values. Functions we learned in class such as btrim(), split_part(), and length() allowed us to make these changes. However, one thing we realized was that the new “creleased” colum with the date type values had null values wherever the month/day was not given. In order to be in the “creleased” column, we needed month, day, and year. Because of this, we were missing thousands of dates, since a lot of the releases only had information about the year. So, in order to retain the useful year values, we included both the filtered years and “creleased” values. After all, our queries need “year” information, so we thought we would have better information if we did not omit all of the releases that were missing a day or month value. On the Musicbrainz side, we performed a very similar process. However, we had to do it twice, because there were date values that needed to be cleaned up in both the “release_country” and “release_unknown_country” tables. What was unique about Musicbrainz is that we had separate columns for date, month, and year, so we did not need to use functions like split_part() that we used in Discog transformations.

After we had all the cleaned out (transformed) columns, which we denoted by placing a “c” in front of the original column names as described above, we looked at the unified conceptual diagram we created and made the necessary edits. We loaded all the cleaned fields that we needed for answering our queries into the

unified schema on Redshift by writing the appropriate DDL and copying the data from the Musicbrainz and Discog schemas into the Unified schema. After we created our unified schema, we realized that there were certain fields we didn't need from our data dictionary and there were some fields missing that we needed. So we made adjustments, specifically making sure to not include the old fields that were either unnecessary or where they were of varchar type but should have been int.

2.5 Data Analysis (Queries)

Once we had the unified schema created with all of the relevant tables including loaded data that are necessary for our initial ten queries, we were finally able to write the appropriate SELECT statements to analyze the data. Of course, this process had its challenges as well. For instance, all of our queries were cross-table queries that used information from multiple columns through multiple tables. Hence, we ended up using numerous Inner Joins and even a few Outer Joins. The Inner Joins were quite useful anytime we wanted to connect information between two tables and when we only cared about the information that both tables shared. This was the case to a certain extent in pretty much every one of our ten queries. However, some queries had to use Outer Joins like for how many releases are there in each genre. We want to keep the genres that have even zero releases, so we had to think of scenarios like this where Outer Joins would be appropriate.

The next challenge was to figure out how to combine the information from the Musicbrainz tables with Discog tables. We did some research online to see how to combine data from multiple datasets into one entity. We found that Unions would be appropriate, so we looked into Union syntax and usage, which led to a successful integration of the Union keyword. Unions were not only helpful in combining data from across schemas, but they were also useful in combining data across tables in

the same schema. For example, Musicbrainz had two tables: “Releases_Country” and “Releases_Unknown_Country”. Both of these tables had information about the which release was done on which date, but the only difference was that there was just no country value in “Releases_Unknown_Country”. We decided to combine the data in both of the tables for certain queries, so we merged them by using a Union.

Now that we had all the Musicbrainz and Discog data in one table, we had a problem with duplicate information. For instance, if Musicbrainz had a release by the artist, Green Day, and if Discog had the same record, then there would be two instances in the Union output. To solve this issue, we used the distinct() function that we had learned when we were using Postgresql in order to only get the distinct releases by Green Day and not have duplicate results. Another solution we found for working around duplicate data was to do Group Bys, which allow for duplicate data to be grouped into one record.

A combination of unions, distinct() functions, and a number of Group Bys allowed us to successfully perform queries that utilized the data from both data sets. However, when we saw the actual query outputs, we realized that there was another minor problem. A lot of queries had unknown fields when they were grouped by. For instance, when we grouped by countries, we would get that there were a number of releases from an “Unknown” country, which meant that the country was not listed. We did not want these in our queries, so we had included additional WHERE conditions to filter out values such as “Unknown” country or a “Not On Label” label name.

2.6 Visualization

We quickly realized that the Quicksight software wouldn’t offer us a lot of flexibility and that we would have to convert some of our queries to find better ways

to visually represent the gist of the query. Another problem was that it wasn't possible to create dashboard with multiple visuals in the same platform, so we would have to simply create several analyses, which could be viewed independently after sharing access to the account.

During the actual process of converting our query results to visual diagrams/charts, we encountered several difficulties. We began the process by taking our SQL code for the queries and created views to store the results. However, we realized that three of our ten queries were scalar values that could not be graphically represented in any type of chart. Hence, we had to redo many of the previous steps to acquire better output that we can display visually. The first query that gave us trouble with scalar values was "How many female artists are there from the Finland area?". Although a useful query, we could not display this value comparatively to other categories. So, we decided to change it to include both the number of male artists and the number of female artists in Finland. To make this change, we had to refer back to many of our previous deliverables and make alterations. First, we had to change our query that was in a question format, we had to go back to the SQL code and edit the select statement to group by gender, and lastly, we had to recreate the views that store the query outputs. Afterwards, we were able to successfully use AWS Quicksight to display the distribution of genders in a pie chart.

We went through similar processes for the other three queries we had to change, which had previously only yielded scalar outputs. For instance, we wanted to know "how many releases came in a Vinyl format among both the Musicbrainz and Discog datasets?", but we could not just have a scalar number as the result. Instead, we took a similar approach as we did with the gender and decided to look at

the overall fraction of Vinyl releases compared to the releases that had other mediums. Consequently, the query had changed to “What fraction of all releases came in any type of Vinyl format among both the Musicbrainz and Discog datasets?”. The same changes were done to the other query, which we changed to: “What fraction of releases in 2014 were released on 04/23/2014 among both the Musicbrainz and Discog datasets?”. Again, all of the finalized queries are available in Exhibit 2 to view.

Unfortunately, these two queries were a bit more challenging to edit on the SQL side. This is because we could not easily group by the field type. For instance, the medium field could not simply be grouped because there was a lot of bad data where the data was called “7 Vinyl”, “10 Vinyl”, “Vinyl”, “VinylDisc”. We wanted to preserve these values because the different types of Vinyl still were important distinctions. So, we did not create a new cleaned out version of the column. Rather, we did two different queries, one for calculating the number of vinyl formats and another for everything that was not vinyl, and we joined these queries together using a union. This way we would have multiple results to perform visualization analysis on. Again the same was done for the fraction of releases in 2014 that were released on 04/23/2014. We simply did a query on how many releases were on that particular date and another one on how many were during the rest of 2014. Next, we used the union to combine the values together and have non-scalar output. We had thought that all of these extra queries, joins, and unions would really slow down the speed of the queries, but they still executed in a matter of seconds. So, we did not need to run performance tables to increase speed of queries, due to the fact that they were already much quicker than 30 seconds each.

3 Conclusion

Working with a variety of tools such as SQL, Entity Relationship Diagrams, AWS Redshift, GitHub, Command Prompts, and IAM has significantly extended our repertoire of skills and has even improved our ability to debug errors. Using all of these tools in conjunction with each other to develop databases, execute queries, and analyze results has taught us how important integration. We not only learned Postgresql prior to the project, but we also learned how to use those fundamentals and apply them to extremely large datasets that were stored on AWS Redshift.

AWS was much faster and more efficient as a result of the valuable work distribution among the Redshift nodes. The majority of tables were copied and transformed in a matter of seconds for each. Even for the queries, each executed in five seconds or less. We had expected to have to create performance table to speed up long-running views, but we were surprised that we didn't need to since Redshift performed quite efficiently. However, we did face challenges early on, since setup is quite a long and arduous task. We had consulted with the teaching assistants and referred to Redshift guides to ensure that setup was done properly.

Along the way, we had seen many other unexpected results and consequences. One major realization was that github is a valuable resource, but if not done correctly, we could end up losing data through merge conflicts. Initially, we only dealt with a few files, so we could essentially push and pull whenever we wanted to without risking conflicts. However, as we progressed through our milestones, we had dozens of files that each of us were constantly and concurrently working on. We did not communicate well regarding our pushes and pulls, so we had unexpected merge conflicts several times and ended up having to re-clone our git repositories, since we had limited knowledge of how GitHub works. Ever since our

conflicts, we improved our communication as well as the frequency of pushes and pulls to make sure all the files were in check.

Next, we really did not expect to find the results we did within our cross-dataset queries. First, we thought that there would not be much overlap between the Musicbrainz and Discog datasets, because the sample data from Musicbrainz made it seem like all the songs/releases in its dataset were more relating to instrumentals, sounds, and etc. while Discog seemed to have more actual songs with lyrics. However, we found that in the actual dataset with all of the data points, there were thousands of overlaps that gave us really good results for queries that involved common artists, mediums, and dates. Second, we did not expect there to be so much “unknown” data values. All the fields were nullable, so when we did a lot of our queries, we would get blank/null results, so we had to include extra conditions in the select statements to filter the unknown values out.

There are a some minor problems with the current setup of the project, and some improvements we would make if we were to do a more rigorous work with a similar goal and the same data, if the goal was more focused on getting query answers that were truthful (given that the data also was exhaustive and truthful, in that case).

For example, the punctuation removal process described in Section 2.4 would render some false positives. The best practice here would be to experiment with supervised samples to find out both the alignment rate and the ratio of correct alignments compared to false ones, using different methods: no punctuation removed, the trim function as advised by the project guidelines, and replacement using regex. In theory, the used trim function would render the most total alignments but also the worst ratio of positive/false alignments. Using regex, we would also be

able to replace e.g. period signs so that we would get (correct) matches for the alignment “REM”/”R.E.M”, whereas the trim function would not align the strings “REM”/”R”.

In the same area as the previous paragraph, we should conduct studies to see which “punctuations” yielded the best result, and experiment with different sets.

There were also some missing data, which affects the total results and would be seen as an unsolved problem that there isn’t much we can do about.

An improvement that we could make is to look more into the options of the copy commands for the data load, to see if Redshift has functionality supporting us to skip the step where we transformed data because of characters appearing where we expected it to only be integer values. This could help us speed up that process for future projects.

Overall, we felt like the project was valuable, but that there wasn’t much room for us to find our own solutions to problems because we were given very good instructions. However, since some of the milestones were time consuming, the scope of the project was on a reasonable level.

Exhibit 1

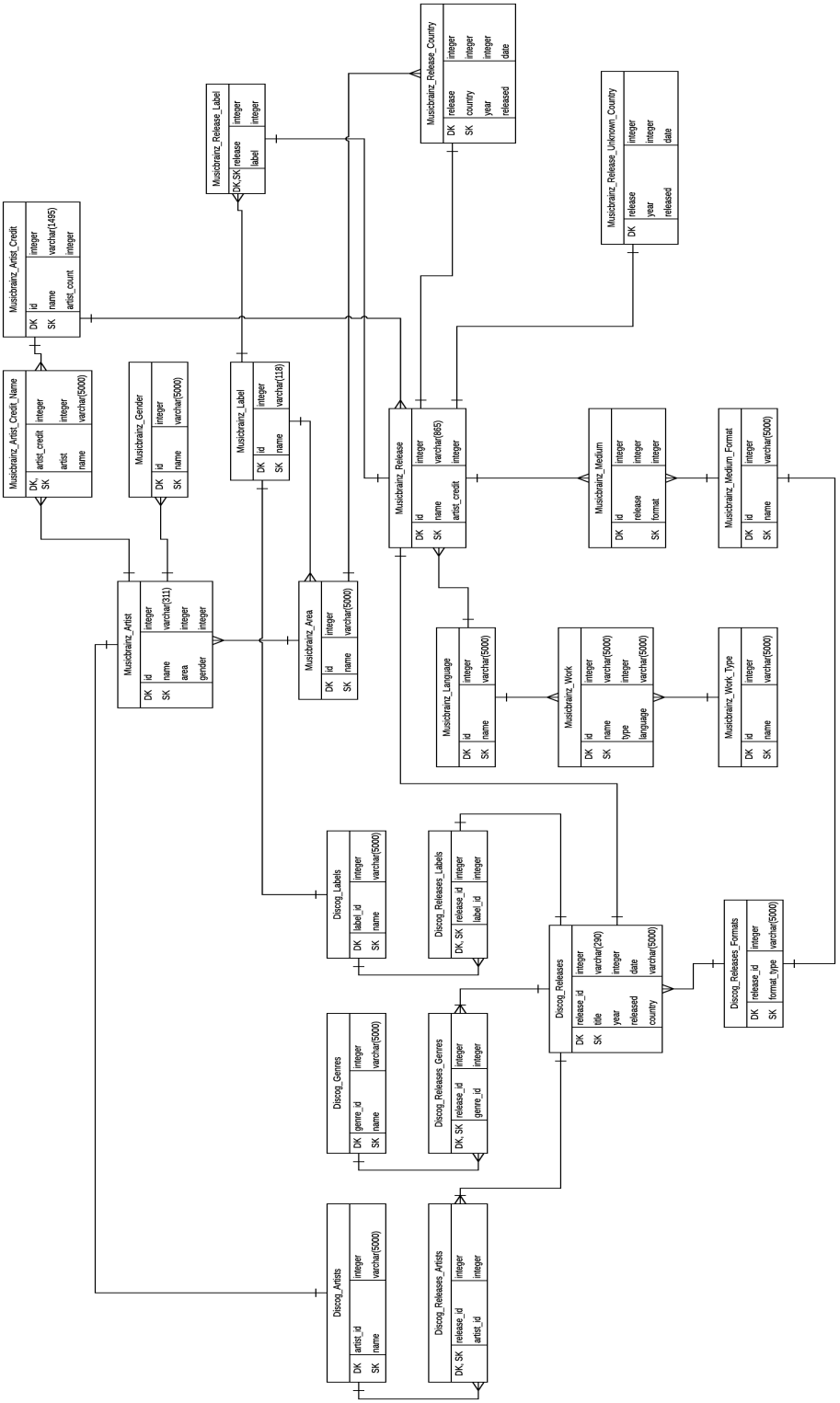
Musicbrainz_Release_Country
DK release integer
integer
SK country integer
year integer
released date
Musicbrainz_Release_Label
DK,SK release integer
label integer
Musicbrainz_Release_Unknown_Country
DK release integer
year integer
released date
Musicbrainz_Artist_Credit
DK id integer
SK name varchar(1495)
artist_count integer
Musicbrainz_Artist_Credit_Name
DK,SK artist_credit integer
artist integer
name varchar(5000)
Musicbrainz_Gender
DK id integer
SK name varchar(5000)
Musicbrainz_Label
DK id integer
SK name varchar(118)
Musicbrainz_Release
DK id integer
SK name varchar(9685)
artist_credit integer
Musicbrainz_Medium
DK id integer
release integer
SK format integer
Musicbrainz_Medium_Format
DK id integer
SK name varchar(5000)
Musicbrainz_Artist
DK id integer
SK name varchar(311)
area integer
gender integer
Musicbrainz_Area
DK id integer
SK name varchar(5000)
Musicbrainz_Language
DK id integer
SK name varchar(5000)
Musicbrainz_Work
DK id integer
SK name varchar(5000)
type integer
language varchar(5000)
Musicbrainz_Work_Type
DK id integer
SK name varchar(5000)
Discog_Labels
DK label_id integer
SK name varchar(5000)
Discog_Releases_Labels
DK,SK release_id integer
label_id integer
Discog_Releases
DK release_id integer
SK title varchar(290)
year integer
released date
country varchar(5000)
Discog_Releases_Formats
DK release_id integer
SK format_type varchar(5000)
Discog_Genres
DK genre_id integer
SK name varchar(5000)
Discog_Releases_Genres
DK,SK release_id integer
genre_id integer
Discog_Artists
DK artist_id integer
SK name varchar(5000)
Discog_Releases_Artists
DK,SK release_id integer
artist_id integer

Exhibit 2

1. Which are the top five countries with the most number of releases among both the Musicbrainz and Discog datasets?

2. How many male and female artists are there from the Finland area?

3. Which labels have more than 5000 releases among both the Musicbrainz and Discog datasets?

4. How many releases are there in each genre?

5. What are all of the releases between 1995 and 2000 by the artist, Green Day?

6. What fraction of all releases came in any type of Vinyl format among both the Musicbrainz and Discog datasets?

7. Which are the 10 most common languages that are used in Operas?

8. Which 5 artists have collaborated on the most number of releases?

9. What fraction of releases in 2014 were released on 04/23/2014 among both the Musicbrainz and Discog datasets?

10. Which are the five most common days of the year for releases among both the Musicbrainz and Discog datasets?