Daniel Tian - dt22249
Matt Peake - mdp2434

1. Introduction

Our database for the final project consisted of using three separate music datasets: Music Brainz, Discogs, and Million Song. The objective was to build a data warehouse by combining at least two of the datasets to create a single database using no less than 30 tables. Specifically we created a unified schema that captures the distinct entities and attributes from each selected dataset and allows for more in depth queries than either dataset could answer on its own.

2. Preliminary Database Design

First, we decided to combine the Music Brainz and Discog datasets to create the database because there is a variety of entities and attributes where we could produce some interesting queries such as finding out which releases were produced in what country and region, what were genres for each of the tracks, and what countries did The Rolling Stones perform in. The questions that we were most interested in answering involved artist information. Since the Million Song dataset primarily contains details about songs, we decided not to it include it in our data warehouse design and instead focus on better understanding the Music Brainz and Discogs data.

For the database, seven tables from Discogs and twenty four tables from Music Brainz were used. The files that were not used in Music Brainz had information about the artists and their songs, which we could not comprehend or did not fit what we wanted in our database. One table from Discogs, “releases_formats”, was excluded because for our purposes we were uninterested in the information it provided. This table

contained only two columns: one for release id values and one for information on what form the product was released in, such as “CD” or “vinyl” record.

Two of the challenges that we encountered when figuring out how to join the two datasets were deciding which tables we needed to answer our questions and which specific columns in each table were actually relevant to us. As we explored MusicBrainz, in most of the CSV files, we decided to not include several of the columns in our database because they were either excessive or could not be deciphered. An example of one of the columns that was difficult to decipher in MusicBrainz is that most of the files had a column, “edits_pending”, that we were unable to understand or make use of the integers after searching for a legend on the MusicBrainz schema website. Separate physical and conceptual diagrams of both datasets needed to be created to see where the two could be joined. For Musicbrainz, initially we went through all the files that were going to be used to figure out how one column of a table linked with another column of another table, which was tedious and nearly impossible to link each table; however, a link to the MusicBrainz schema website was provided for our convenience and was used as a guide to help identify each table’s primary and foreign keys and to gain a better understanding of parent/child and junction table relationships.

The unified schema was created by combining the artist table from each dataset where the artist name columns were equal and not null. In the “combined artist” table, in addition to the artist information included in the musicbrainz artist table, the musicbrainz id and the discogs id associated with an artist were each stored in their own column which allowed us to access information about each artist from either database. We

decided to join on names because, after looking at each database and comparing the columns they had in common, it seemed to be the option that would preserve the most data from the source databases. Although data will inevitably be lost, by joining along artist names, we ensured that artists who survived the inner join would have much more in-depth information associated with them compared to their original associations.

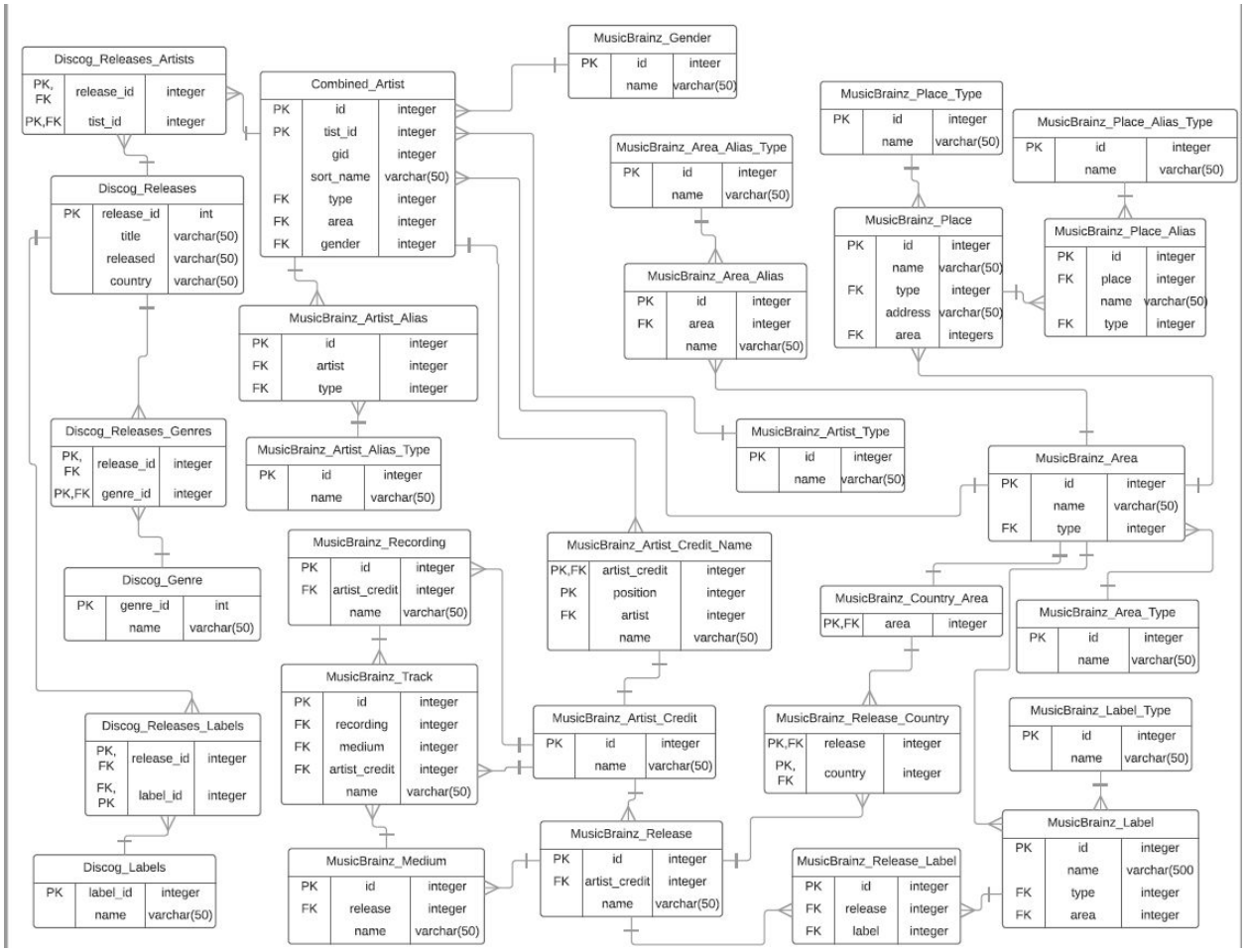

Figure 1: Physical Diagram of Unified Schema

Once we had completed our conceptual and physical design diagrams for both datasets, we combined them with the unified schema. For the most part we stuck to our initial designs, but as we progressed through this project, some minor changes were

made so our database was better suited to answer our questions of interest, the finalized design can be seen in Figure 1 above. With initial designs finished, we created a data dictionary to catalogue which columns we planned to use from each of the CSV files and which relevant information they represented. This ensured that at later stages in the project we would be able to understand the data we were working with in case the column or table name was not immediately obvious.

3. Table Creation and Data Loading

The next step was to create staging tables and load all the data from the CSV files into their respective schema (Discogs or MusicBrainz) from the S3 bucket via DDL (Data Definition Language) SQL script for each dataset. The script to create the schema contains a sequence of actions: create the schema, set the search_path to the schema of interest, and create each table in the schema by calling a separate create table sql file with the \i command in the psql shell command line.

3.1 DDL Scripts

To create each table, there was some set up work that had to be done before running the “create_schema” sql files. In order to expedite the process, we utilized a python script provided by Professor Cohen. The “generate_DDL.py” script, when provided with a csv file (with headings) as input, would scan through the information in the csv file and output an sql file containing DDL that would create tables capable of storing the information read from the CSV files. This script saved our group a great amount of time. Instead of individually writing out the SQL statements for each table, we were able to simply run the python script once for each table. However the create table

statements then needed to be reviewed because some of the data types and lengths were assigned incorrectly. This occurred because the CSV files that we used as input for the python script were only a small sample of the complete datasets in the S3 bucket. Before we were able to load the complete datasets into our database, we went through a trial and error process. If copying data from the S3 bucket failed, we would adjust the DDL files and then attempt to load the data again. This process was repeated until each table was successfully loaded into our schema.

3.1.1 Representative Challenges

For example, in the artist CSV file, the length that was assigned by the script for the name of the artist was too short because the CSV files we used did not contain the longest artist name that existed in the complete dataset. As such, when we later attempted to load the dataset from the S3 bucket into our table, we received an error due to invalid data type since some of the names being loaded were longer varchars than the column was capable of holding. This particular case was one we had anticipated. In order to overcome the issue, we simply altered the create table scripts to set the datatype of the artist name columns to varchar(4000). While varchar(4000) is likely a larger string length than needed, this ensured that all data would be loaded without a problem. Later we will discuss methods used to determine more precise (and efficient) varchar data lengths when creating the unified schema.

Other columns such as “edit_pending” were assigned a boolean data type by the generate_DDL script. We had assumed this was the correct type because the sample data file only showed 0’s and 1’s but when we attempted to copy data into tables with

the “edit_pending” column errors appeared. Unlike with the artist names, we had not expected an error associated with this column. In order to better understand what was wrong, we ran a diagnostic query through the psql command line that showed the most recent rows that had failed to load and the specific error associated with them. This made it clear that the issue was that the data from the S3 bucket had another integers that we did not account for and the edit pending column was not intended to be a boolean value as we had initially thought. To resolve this problem, we edited the DDL in any create_table script containing an “edit pending” column so that the columns that were originally assigned the boolean type were changed instead to an integer type.

3.2 Verification by Data Sampling

In addition to changing data types, we added a “drop table if exists” line to the start of each create_table file. The initial “drop table if exists” line allows for quick and easy edits. In the event that we needed to change the table definitions we would have to write specific “alter table” statements or drop/add columns for each edit. However, since for our project the create table DDL was not very time intensive it was more efficient to simply delete the table and then recreate it with the updated DDL. After all the data had been uploaded, a quality check was performed by retrieving ten random rows of data from both schemas to ensure that the data was uploaded into the appropriate columns. While going through this process, we found that one of our tables was empty. On further inspection we found that while we had included all of the tables in the table creation files we had simply skipped one of the Music Brainz tables when writing the script to copy data from S3. This was an easy fix, and after a couple of cycles in which we edited the

table DDL we were able to successfully load the data into the additional table. Several samples were taken from each table and all the data was shown to be loaded properly.

4. Data Integration

The next step was to create and populate a unified schema between the Discogs and MusicBrainz schemas. The two schemas were joined along the names of the artists from the artist tables but it is not possible to simply copy the existing tables from the two schemas because the raw data has consistency issues which would limit the analysis across the datasets. In order to fix the issue, the data needed to be transformed to improve the integrity of the data. SQL statements that took random samples of 100 - 200 rows were implemented to see where integrity of the unified schema may be compromised. The main integrity issue was the varchar columns had punctuations in the string as the strings were formatted differently between the datasets, especially on the artist names. Without overriding the raw data, the staging tables needed to be altered by adding another column where the newly formatted string would be populated in. By creating a new column for the sole purpose of storing the standardized strings, we were able to “reset” the string in the case that our initial attempts at formatting “over-filtered” the strings. For example, if we were to run a query using the edited data and found that it was no longer suitable for our purposes, we could simply update the copy column by referencing the original again.

The punctuation characters that we generally decided to remove from the columns with varchar data types were **;**, */*, **(**, **[**, **:**, **-**, **...**, **with**, and **Vs.** However, in some cases, due to the manner in which we edited the strings, filtering all of these punctuation

characters was not always logical. This arose from the fact that we edited the original data using the “split_part()” sql function. Since various releases may list the main artist “with” additional contributors and this formatting is not standardized across datasets, we used split_part() to simply remove any part of the string after the character being filtered. However, this also means that in some cases, sections of some artists actual names were likely lost if they happened to use an unusual character in their name.

In addition to this method in which we cut off part of the string, we also used the btrim() and initcap() functions to prevent mismatches due to inconsistent data entry in the original datasets. These functions remove blank spaces on either end of a string as well as capitalize the first letter of each word in a string. By applying these filtering functions to each column as we copied them to the “formatted” columns of their respective schema’s table (Discog or Music Brainz) we were able to increase the number of artists that would be appropriately matched when creating the combined artist table. As a side effect of this though, it is also possible that by applying split_part() to artist names we may have caused some artists to be joined that were not actually the same group/person. While not ideal, for our purposes we deemed this an acceptable drawback.

When creating the tables for the unified schema, we no longer needed all of the data copied from S3 into the Discogs and Music Brainz schemas. It was necessary to include every column in the in separate schema DDL files in order to avoid errors from copying the data, but since we did not intend to use several of these columns we simply left them out. In order to do this without error, we created the Unified tables as “selects”

from the relevant tables in the individual schemas. For the most part, this simply involved selecting the columns that we were interested in from either a Discog table or a Music Brainz table. The main exception to this was the unified artist table. As previously mentioned, we connected the two datasets by combining the two separate artist tables into one. To do this, the “create as select” we used had to include an inner join statement between the Discogs and Music Brainz artist tables.

In order to make a more efficient use of memory, the new varchar columns that would be copied into the unified schema needed to have the actual bytes that are required instead of the estimated varchar(4000) used when the data was originally uploaded. Redshift is capable of returning the length of a string, but if a string contains non-standard characters, then the actual number of bytes required may exceed the actual string length. Unfortunately, Redshift does not have a built-in function capable of directly measuring the bytes in a string. Instead, we used a user defined function (UDF) that ran via Python to retrieve the number of bytes.

When using the UDF, some problems we faced were columns missing information (containing null values) which would cause an error from the UDF. If this happened, we would instead use the built-in len() function and attempt to load the data in a varchar of that size. If the data was unable to load, then we knew there must be a foreign character and would raise the size of the varchar in increments until everything was able to copy successfully. While this alternative sounds more involved, since it uses only built-in functions, they are able to run fairly quickly. This is opposed to the UDF function that must run through python and can take 20-30 minutes to run through some

of the larger tables we were working with. After going through each of these processes, we were eventually able to successfully create our unified schema connecting information from both Discogs and Music Brainz.

5. Data Visualizations

With the unified schema completed, we then went on to write several queries that would utilize data from both of the datasets. For example, we did not have any data on genre from the Music Brainz dataset, but since all of the artists in our combined set were linked with a Discog id we were able to access genre information from Discogs while also considering information such as area types in Music Brainz that Discog did not contain. With queries written, we then went to create views that could be accessed by Amazon's Quicksight service to create more intuitive data visualizations.

When creating views from our queries, we ran a simple check on the "quality" of the user experience by ensuring that loading all data from the view did not take an unreasonable amount of time. Unfortunately, some of our more in-depth queries, though they were likely the most interesting, simply took too long to load. The main reason for this is that these queries included multiple subqueries to allow us to look at a break down of genre for the most accomplished (greatest number of releases) artists. In order to make these views load faster we eventually had to simplify the queries by removing some of the additional subqueries and categories we had hoped to visualize.

One example of how the data visualization not only helped showcase our information, but helped us to identify a problem was when we looked into artist names and their associated genres. The query utilized the split_part() function as a "quick and

dirty" method of selecting an artists' first name, [split_part(name, ' ',1)]. However, when we looked at the results of this query visualized in a bar graph, we found that there were over 2 million artists named "Jean" while the next most common name was "Hanz" with about 50 thousand. By looking at the source data we realized that we had over filtered the name data by removing hyphens. After updating the database, the view showed much more reasonable results.

6. Conclusion

Throughout this project we were required to adapt to changing plans as outcomes were not always what we had anticipated. Several times, the importance of "sanity" checks, and validation of data was made clear to us. If we had not run the random select statements of initial schema or made the final visualizations, we likely would have missed major flaws in our database. Looking forward, we wish we had been able to utilize the more in depth analytical queries. It may be possible by pre-computing the multitude of joins required, but we suspect that a better solution overall would be to overhaul the database design. If we were able to connect the data between Discog and Music Brainz in more locations than simply the artist table it would inherently reduce the number of joins required to access the data we need.