Small Artist Discovery (SAD:)

CS 564

Group 16

Anna Kim, Brett Fisher, Hunter Droegkamp, Yewon Lee

[**Introduction**](#_x1c0iwbzivd2) **3**

[Motivation](#_a0dksrxuxpau) 3

[Application Description](#_lfhgitutqzaq) 3

[Project Organization](#_fqhuuaimygg1) 4

[**Implementation**](#_98idaxfqvbw8) **4**

[Architecture](#_5lgzdwtomkp6) 4

[Dataset](#_ntr0lwdqua0) 4

[ER Diagram](#_90zzgmxx52ii) 5

[Relational Model](#_bexr1mqi4bim) 6

[Implementation](#_meothlq425wg) 7

[Evaluation](#_vqm17wzcydrg) 10

[Test SQL Queries](#_t8gif24wkmro) 11

[Slider Search Results Evaluation](#_u1pn6jfri5gb) 11

[Fill Review Table Evaluation](#_v7d266o82ff1) 14

[**Conclusion**](#_k1ho8wwrll5a) **15**

[**References**](#_erqbq2l0nr2j) **16**

# Introduction

Our database application, named Small Artist Discovery, allows users to perform different types of searches for music and discover lesser known songs and artists. We collected songs, artists, albums, and genre information from the Free Music Archive, an online repository of royalty-free music.

## Motivation

The idea for the application stemmed from a mutual interest to work with something music related. After looking at popular recommendation algorithms like those used at Spotify, we noticed that many of these algorithms lack the ability to recommend users much less popular songs. With this problem as our focus, we decided on creating a framework for an application that allows user to discover lesser known songs and artists via different methods of searching.

## Application Description

The Small Artist Discovery application is web-based with an interface built from HTML, JavaScript, and CSS, while the backend is done using PHP. To run server and database in own desktop, Xampp is used, which allows programmers to work on a local server and test a local copies of websites using PHP code and MySQL databases. The application will feature a login system so users must create an account with username and password in order to access the application. This will allow us to store user data to create a more customized experience for each individual.

There will be 2 primary search features of the application: slider search and genre search. Slider search will ask the user to adjust 8 different sliders of different audio features to create a range that the returned song list will be filtered by. There is a possibility that nothing would return for song list, since none of the song matches within the range of all 8 features. Since the slider search result is precise and sensitive, instead of giving a result list in separate page, user can observe the changed list while they move the slider bar. As the user adjusts each slider bars, all the list of songs is returned that displays columns for song name, artist name, album name, and user preference. The user preference column contains a checkbox for each song - if the user likes a song, they can click the checkbox. A marked box is considered a “Like” for that specific song. The genre search begins by allowing the user to select one genre. Once the user presses “Search,” a results list is returned that looks identical to the slider search results, but this time only containing songs from the chosen genre. Songs for both searches will always be ordered in order of least popular to most popular, based on the number of listens of the songs.

The purpose of allowing users to “Like” songs is to help them keep track of what they have already listened to and prefer. This wouldn’t be complete without a functionality for the user to bring up a list of their current liked songs, so this is also implemented into the application. Furthermore, if user wants to remove certain song on User List, the user can mark checkbox that is labeled as “delete” and click the delete button.

## Project Organization

In this report we will cover the system architecture, the data we used to populate the database, the entity relationship diagram and relational model of said database, the implementation process, and finally the evaluation of our queries and functionalities. The ER diagram, relational model, and most of the queries were handled collectively by the group. Individual work was divided as follows: Hunter performed data cleaning, Anna and Yewon created the interface and translated the backend, and Brett worked on backend development.

# Implementation

## Architecture



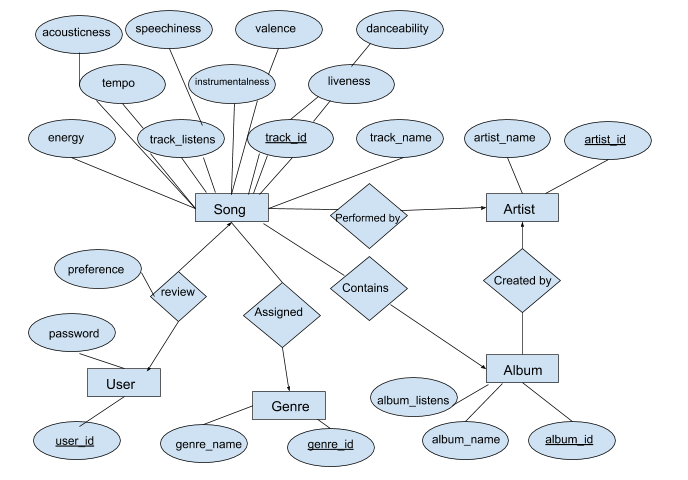


Our application architecture consists of 3 main components: the interface, querying/backend control, and the database. This is set up like a typical model-view-controller pattern that is used in many web-based applications in all different kinds of programming languages. The user interacts with the interface while they log in and while using the application. The interface accesses the queries when the user specifies a search. Then the queries pull data from the database, and the control sends it back to the interface to be displayed. When the user “Likes” a song, the database must be updated to store their preference. Also, when the user “Deletes” or dislikes a song from the user song list, the database must be deleted from Review table.

## Dataset

Data was retrieved from the Free Music Archive whose github repository contained multiple .csv files with information on songs, artists, albums, genres, and audio features among other things (Defferrard, Benzi, Vandergheynst, Bresson, 2016). A total of over 100,000 Creative-Commons licensed tracks are listed in this database. The files we used to extract data were ‘raw\_tracks.csv,’ ‘raw\_echonest.csv,’ ‘raw\_genres.csv,’ ‘raw\_albums.csv,’ and ‘raw\_artists.csv.’ Python, and more specifically the pandas package, was the tool of choice for cleaning this data. It was important to remove any missing data from the dataset, as well as make sure all data types corresponded to our relational model. Even though there were over 100,000 tracks initially, our final total number of tracks was 14,264. Another category that needed clearing up was genre, since many tracks had multiple genres. For those tracks with more than one genre, a single genre was randomly chosen. In total there were 114 different genres. The artist and album tables consisted of 3,391 and 2,861 entries, respectively. Artist, album, and genre tables only contained columns for name and ID (aside from album which also contained a number of listens), while the song table had many more columns due to the need to join tables, have a column for track listens, and for the additional 8 audio features for each song.

## ER Diagram





The ER diagram, despite having many attributes, it relatively simple in design. There are 5 entity sets, Song, Artist, Album, Genre, and User. There are 5 relations connecting these entity sets, Review, Assigned, Performed by, Contains, and Created by.

* Review connects Song and User and represents when a user likes a song in the application, with each user being able to like many songs and each song being able to be liked by more than one user.
* Assigned connects Song and Genre and simply represents what genre belongs to each song, with each genre being able to be assigned to more than one song.
* Performed by connects Song and Artist and represents what artist makes each song, with each artist having made more than one song.
* Contains connects Song and Album and represents how each song will appear on an album, with each album containing more than one song.
* Created by connects Artist and Album and represents how each album is created by one or more artists.

Song has 10 attributes, with track\_id being the primary key.

User has 2 attributes, with user\_id being the primary key.

Genre has 2 attributes, with genre\_id being the primary key.

Album has 3 attributes, with album\_id being the primary key.

Artist has 2 attributes, with artist\_id being the primary key.

In addition, the relation Review has 1 attribute called Preference.

## Relational Model

Song(track\_id: integer, track\_name: string, track\_listens: integer, tempo: real, acousticness: real, speechiness: real, valence: real, danceability: real, energy: real, instrumentalness: real, liveness: real, artist\_id: integer, album\_id: integer, genre\_id: integer)

Album(album\_id: integer, album\_name: string, album\_listens: integer, artist\_id: integer)

Artist(artist\_id: integer, artist\_name: string)

Genre(genre\_id: integer, genre\_name: string)

Username(user\_id: string, pass: string)

Review(user\_id: string, track\_id: integer, preference: boolean)

## Implementation

The application will start by displaying a login page for the user to enter their username and password. If they are not yet a user, there will be a sign up link below that takes the user to the registration page to create a new username and password. Once the user successfully logs in, they are taken to the homepage of the application, which displays 3 buttons: Slider Search, Genre Search, and User List.



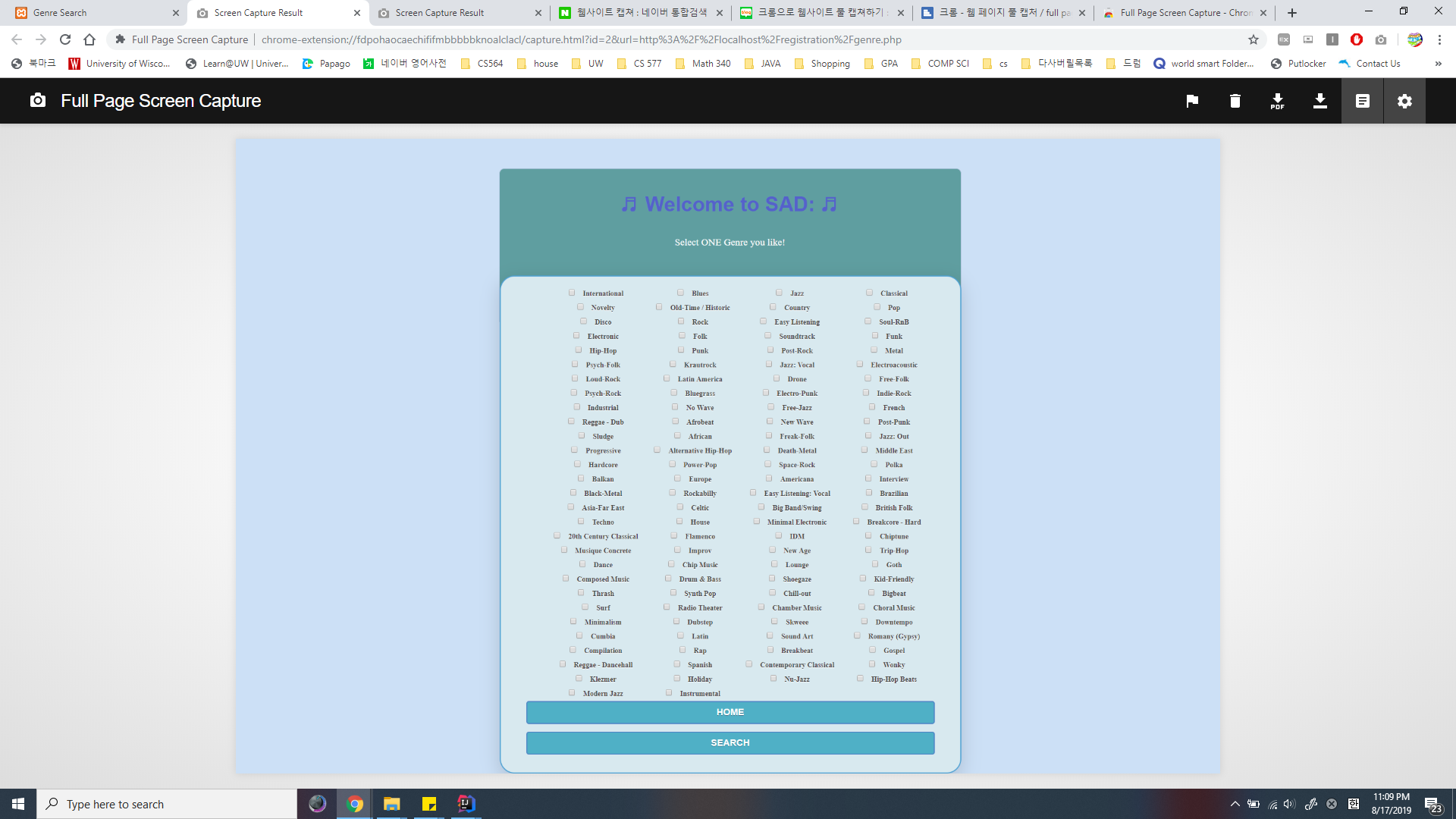


If the user selects Slider Search, they will be taken to a new page that displays 8 different sliders, each with 2 endpoints that can be adjusted, and a submit and home button below. Once the user finalizes adjusting the sliders to their liking, on the right side of the slider bar would give a list of songs that will be displayed in ascending order of popularity containing only the audio feature values between the slider nodes. The user can click the checkboxes to “Like” the song, then press submit to insert their likes to review table on the database. Also, submit button will direct user to User List page which shows the list of songs that user liked.

Likewise, if the user selects Genre Search, they will be taken to a new page, this time with a list of all genres next to checkboxes, as well as a search and home button. The user need to select only one genre before pressing search, or it will not search. After pressing “search” the page displays identically to the slider search, but this time only for songs that are labeled as the selected genre. Again, the user may click the checkboxes to “Like” the song, then hit “Submit” to save their preferences to the database and go to User List page.

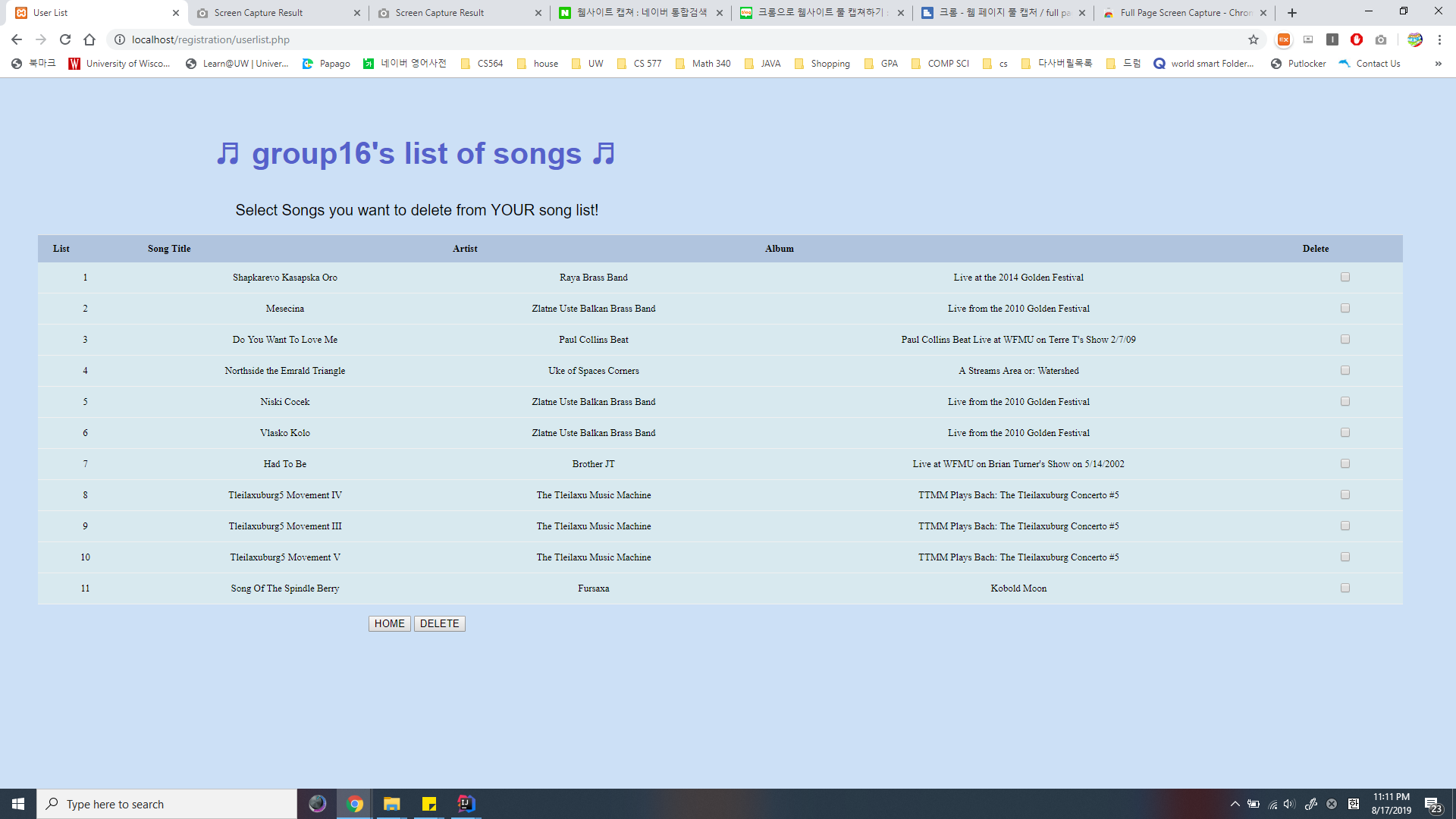








The final feature implemented is a page to display a list of all the “Liked” songs for the corresponding user. This way, the user can easily see what music they have enjoyed the most so far. When the user presses the “User List” button on the homepage, they will be taken to a new page that displays all songs where the user has clicked a check in the checkbox and submitted previously from both search page. The list of songs is displayed columns for song name, artist name, album name, and “Delete”. When user does not like certain songs in User List, they could select checkbox on Delete column and presses “Delete” button to delete selected songs from the User List. At the same time, the songs will be deleted from Review table on database.



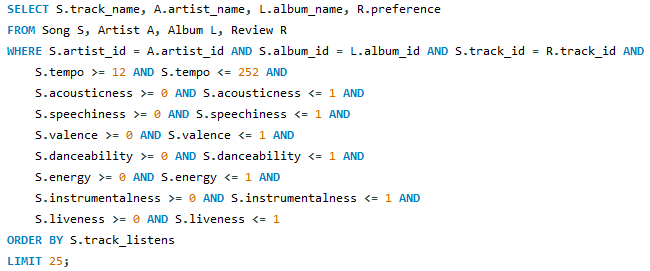


## Evaluation

### Test SQL Queries

Two SQL queries were observed for their performance and accuracy: one that performs the slider search and the other that fills the Review table with an entry for every song for a new user when it gets created.

#### Slider Search Results Evaluation



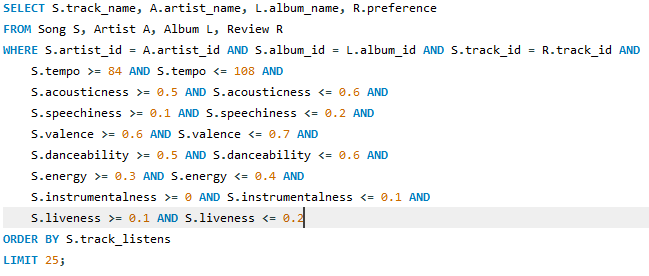
#### 





The first test run on the slider search query was to simulate a search where the slider arms were not adjusted at all, meaning there are no songs filtered out of the results and the list returned will simply be all the songs in ascending order of listens (only 25 will show). In practice, the numbers will be taken from the slider placement in the interface, but to test the query, we hard-coded the maximum and minimum amounts for each audio feature (all range from 0 to 1 besides tempo).

Running this query initially proved problematic - MySQL default max runtime is 30 seconds, so it would stop and return this error. After adjusting the max runtime to a higher value, the query finished running in about 37 seconds, returning what is shown in Figure 8. These results show the proper columns (track name, artist name, album name, and preference) and the order of songs matches up to the .csv file when sorted ascending by listens in Excel. There also does not seem to be any results potentially missing, so as long as the max runtime is increased, this query will be able to function properly in the base case.







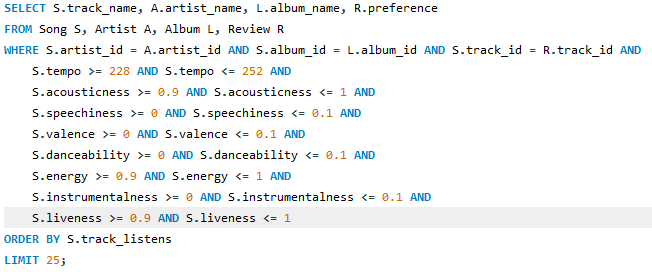






Testing the same query again but with a different scenario, we try to make sure the query will return a specific result if we search for it. The row in Excel of the 20th least popular song is shown in Figure 10. Using the audio features of this song, we make the sliders as narrow as possible while still containing these values (the sliders go from position 0 to position 10, hence most minimum ranges are 0.1). Having the sliders adjusted this way should return at least the song “ReIntro,” possibly others if there are any songs with extremely similar features.

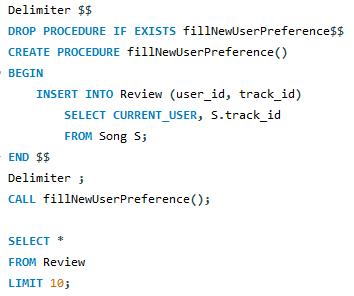
Figure 11 shows that “ReIntro” was the only result from this adjusted query. This means we do have the results we want and there are no other songs within this specific range of features. Another upside is running this query only took .016 seconds as opposed to 37 seconds when the sliders were unchanged. This shows that the average runtime of the query could possibly be at a more reasonable runtime than 37 seconds, since many users will want to adjust the sliders somewhat.



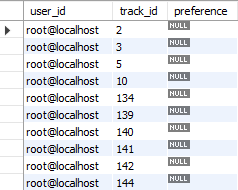


The final scenario we used to test the slider search is setting all the slider ranges to be very small, but at either end of the total range. This is to test if our data contains any entries that correspond to these very extreme values, in case the user adjusts the sliders in a very odd way. Running this query provides no results. However, this is not necessarily a bad thing, it just indicates that our data is not expansive enough to account for every combination of slider ranges. There may be multiple other possible queries that return nothing, so this is simply a fact of the data the user should be aware of.

#### Fill Review Table Evaluation









When a new user gets created, we need a way to store that user’s preference for each song if they choose to “Like” a song. To do this, we will have the query in Figure 13 execute when a new user signs up. This query should insert 14264 rows into the Review table, with each row containing a different track\_id, the same user\_id, and leaving the preference column untouched. Here the query is wrapped in a stored procedure for ease in calling the query. To visualize the results of the query, we will observe the first 10 rows in the Review table after running it. Viewing the results in FIgure 14, the Review table is updated properly after running the query. The track\_id column matches up to the first 10 track\_ids in the .csv file, MySQL reports 14264 rows are inserted, the user\_id is all the same (the ‘root’ user, in this test case), and the preference column still contains only NULL values.

# Conclusion

Throughout the time of working on this project, we noticed a few interesting and helpful ideas when applying things we learned in the class. To start, designing the database become a lot easier and more clear once the ER diagram was completed and normalized. The design of the ER diagram helped clarify what would and would not be needed in the application in terms of data, and indirectly in terms of the backend and interface. Stored procedures became useful tools to control structure and make for easier calls to queries. Putting together an application also called for a variety of software and programming languages - we used PHP, HTML, CSS, JavaScript, MySQL, XAMPP, and Python to complete ours.

Learning about database systems unsurprisingly aided in handling data better outside of the database. The relational schema that was constructed early on in the project actually turned out to be a great way to visualize how the final tables would look when we got to the data cleaning stage. Each relation was a table and each attribute was a column of that table - simple! We learned in class that keys of a table cannot have any null values, so we took extra precaution by removing all null values in our data to make sure this was never an issue and so we always had full data returned when running queries. The SQL queries are also fairly translatable to data summarizing in other languages such as Python and R, aside from a vastly different syntax. Logically, however, the similarities are striking, as the thought process involved in writing a SQL query can be used almost identically (but adapted syntactically) in other data languages.

As the backend and frontend programmer, using multiple languages and program package was challenging. However, we learned a great amount of the lessons as solving each problem, such as connecting database, backend and frontend, learning multiple new languages, and using developing tools to use local server.

A major obstacle we faced was connecting the interface to the backend code. We ended up using a language that no one had used prior to get the HTML and CSS of the interface to work properly with our MySQL queries and the functionalities of the application. Next time, we’ll know to schedule extra time to work on coding when building an application, as without code, the application is just simply an idea!

Given more time, our next step would be giving some artificial intelligence to genre search algorithm. We would do this by applying the data collected from user reviews to genre search algorithm using advanced mathematics. Therefore, the application would give more precise results to each user. Another consideration that would come with future iterations would be to add a function of listening the music, if we have a license to play the music. We believe that our first iterations has proved that program SAD: has a lot of potential, and we can’t wait to see what’s next!

# References

1. Defferrard, M., Benzi, K., Vandergheynst, P., & Bresson, X. (2016). Fma: A dataset for music analysis. arXiv preprint arXiv:1612.01840. <https://arxiv.org/abs/1612.01840>