Spotifinder

CIS 550 Final Project
Group 41, Spring 2023

Blair Barineau
Paavnee Chauhan
Aaron Finkelstein
Rohan Kamat

Table of Contents

1. Introduction

The purpose behind Spotifinder was to create a way for users to easily discover and obtain music they may like. Our web application addresses three main issues:

Music discovery can be challenging

Without effort, and a bit of luck, it can be difficult to discover music that you like. Spotifinder allows users to enter a variety of inputs such as a song, artist, album or playlist they like; or numerical parameters about the type of music they like such as danceability, energy or tempo. From there, the web application returns songs, artists, albums, or playlists that the user is likely to enjoy based on their input.

Creating playlists can be time consuming

Going through one's music library, or thinking about what songs to add, in order to create a playlist, can be a time consuming process. Based on the aforementioned inputs, Spotifinder uses data on Spotify playlists to display pre-made playlists that the user is likely to be interested in.

Platforms recommendations can be heavily influenced by advertising

As streaming platforms become one of the primary ways for artists and record labels to promote their music, recommendations on these platforms become increasingly influenced by paid partnerships, in turn lowering the quality of suggestions for users. Spotifinder simply uses numerical data values such as tempo, energy, valence and danceability to suggest music that is solely based on user input and nothing else.

---

2. Architecture

2.1 Database

On Amazon Web Services, we utilized a Relational Database Service (RDS) that used a MySQL Community engine and a db.t3.micro instance type.

2.2 Data Processing

Data was gathered as a combination of JSON files and CSV files (more in Section 4.1). Data was uploaded into Google Drive, to which we connected a Google Colab notebook. Within the notebook, the Pandas library was used to convert the JSON files into a CSV file with our desired format.

2.3 Data Input, DDL, DML

Data was inputted to the aforementioned DB using MySQL on DataGrip. MySQL DDL was used to create the tables and schemas (more in Section 4.2), after which the CSV files were uploaded into DataGrip. All DML and query testing occurred on DataGrip using MySQL DML.

2.4 Frontend

The frontend primarily utilized node.js. The React and Material UI (MUI) javascript libraries were also used to display content and other UI elements.

2.5 APIs

The frontend also utilized the Youtube and Spotify APIs (more in Section 10). The former was used to display the music video of a song on the web application, if selected by the user. The latter was used to display the Spotify images of an album cover or artist profile on their corresponding pages on the web application.

---

3. Data

We utilized two datasets for our project, the links to which can be found in Appendix A.1.

3.1 Spotify Million Playlist Dataset (Playlists)

The online dataset from AIcrowd contains the information of 1,000,000 different public Spotify playlists, composed to promote research on the relationship between music and humans.

The data itself is published in JSON format in separate files of 1,000 playlists each, with nested information on each song. Attributes are located in Appendix A.2.1.

The dataset is primarily used to give playlists recommendations to users based on their selected song, artist, album, or another playlist.

The compiled dataset contains over 2,000,000 unique tracks by approximately 300,000 different artists. The data in its entirety became difficult to evaluate due to its JSON format and sheer size. Thus, a subset of 126,999 playlists were selected for this project, to be discussed further in Section 4. Of these playlists, there are 316,271 unique songs with an average of approximately 71 songs per playlist.

3.2 Spotify 1.2M+ Songs Dataset (Songs)

The online dataset from Kaggle contains the Spotify song attributes for approximately 1,200,000 unique songs on the application. It was compiled using the Spotify API by public users and published directly for the use of others.

The data is published in CSV format in a single file, with each entity representing a single song.

The features included in this dataset are presented in Appendix A.2.2.

This dataset was utilized for the purposes of our project to allow for the recommendation of songs, albums, artists and playlists on the basis of their measured, quantitative qualities. Given the fact that the Spotify Million Playlist Dataset solely includes qualitative attributes, it was necessary to combine its information with that of another dataset to return suitable recommendations. Given the fact that they both directly come from Spotify's API, songs in both datasets are able to be correlated based on their mutual track URIs.

All 1,204,025 tuples included in the original file were used for the purposes of this project, each containing the entirety of the song attributes found in each tuple. However, as not every song included in this dataset was included in a playlist, the final amount of songs left amounted to 316,271.

4. Database

Extensive efforts involving data cleaning and entity resolution were performed due to the dataset's unconventional size and format.

4.1 Data Cleaning

To begin the data cleaning process, each file was individually uploaded into **Google Drive**. A **Google Colab** notebook was created to access and manipulate each dataset using different **Python** techniques. Using the **Drive library**, the files were mounted into the notebook to commence the data cleaning process.

For the Playlists dataset, the **JSON library** and **Pandas library** were used in conjunction to separately append each transformed file into a single dataframe. This was largely performed with the JSON `explode` and `normalize` functions to transform the dataset's granularity to be on the song-level. These efforts were performed in a loop to account for the fact that only 1,000 playlists were contained in a given file, appending each to the master dataframe at the end of each loop iteration.

For the Songs dataset, Pandas was similarly used to upload the file into its own dataframe. The dataframe containing song-level information was duplicated, dropping all playlist information and duplicates. This dataframe was merged with the newly-created dataframe from Songs, joined on the `track_uri` attribute.

A new dataframe was created from the newly-constructed Million Playlists dataframe containing the song and playlist IDs. This was performed to ensure that the playlist and song relations would solely contain playlist and song information as their lowest levels of granularity, respectively. Thus, all the song information was dropped from the playlist dataframe following the creation of this new dataframe.

After initially seeing extremely lengthy runtimes for the data ingestion in **DataGrip**, we decided to swap the string primary keys for integers. Thus, each song and playlist were given their own unique integer IDs, correlated to the other tables using a dictionary from original track URI to the new integer ID.

Each file was uploaded into Google Drive as a CSV to form the basis of three relations.

4.2 Data Ingestion

To begin the data ingestion process, three relations were created without any data: `Playlists`, `Songs` and `Contains`. The relational schema of each is provided in Section 4.4.

Initially, each CSV was uploaded into its respective relation directly using the **Import Matrix** on DataGrip. However, extensive runtimes due to each relations' string primary keys rendered this initial attempt useless.

After the creation of integer IDs through data cleaning, each relation was dropped and recreated with integer primary keys. The data was uploaded directly using `LOAD DATA INFILE` statements. The entire ingestion process took approximately 3 hours to complete.

4.3 Entity Resolution

A few major problems were encountered in relation to entity resolution.

Foremost, given the fact that the Songs and Playlists datasets were composed by different individuals, it was unlikely that each of the songs in the Playlists dataset would have attributes contained in the Songs dataset. Upon merging both, it was found that only approximately 300,000 songs in playlists were found to have all of the attributes.

To resolve this, natural assumptions were composed about the songs in each playlist and implemented in **MySQL**. Humans often group songs together in playlists that have a similar mood, feel and structure. Thus, for each playlist, the average of songs' included attributes were sequentially set equal to the unincluded attributes of the other songs in the playlist. This allowed for the result size to greatly increase for each query with a logical solution.

For each playlist, the average of all songs' attributes were taken and included as columns in the finalized relation. This was performed to allow for easier access of attributes that would otherwise need to be calculated in multiple queries.

Additionally, the desired functionality included songs and playlists given as recommendations on the basis of a selected album or artist. Thus, `Albums` and `Artists` tables were created to store average attributes of all songs relating to a given album or artist. Similarly, this was performed to allow for easier access of attributes that would otherwise need to be calculated in multiple queries. `Tables` were chosen instead of `Views` in case the data needed to be updated and to minimize performance degradation.

To conclude, a table to store user log-in information to the platform was created as well.

4.4 Relational Schema

Playlists (PID, name, num_albums, collaborative, modified_at, num_tracks, num_followers, num_edits, num_artists, duration_ms, avg_danceability, avg_energy, avg_loudness, avg_speechiness, avg_acousticness, avg_instrumentalness, avg_liveness, avg_valence, avg_tempo, explicit)

Songs (track_uri, track_name, album_name, atrist_name, danceability, energy, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo, explicit, year)

Contains (Playlist_ID, Song_ID, position)
Foreign key (Playlist_ID) references Playlists (pid)
Foreign key (Song_ID) references Songs (track_uri)

Albums (album_name, artist_name, avg_danceability, avg_energy, avg_loudness, avg_speechiness, avg_acousticness, avg_instrumentalness, avg_liveness, avg_valence, avg_tempo, num_songs)
Foreign key (album_name) references Songs (album_name)
Foreign key (artist_name) references Songs (artist_name)

Artists (song_ID, artist_name, avg_danceability, avg_energy, avg_loudness, avg_speechiness, avg_acousticness, avg_instrumentalness, avg_liveness, avg_valence, avg_tempo)
Foreign key (song_id) references Songs (track_uri)
Foreign key (artist_name) references Songs (artist_name)

UserInfo(username, password)

4.5 Relation Information

Relation	Instances
`Playlists`	27,908
`Songs`	316,271
`Contains`	1,985,804
`Albums`	145,440
`Artists`	316,271
`UserInfo`	***Updates for new Users added***

4.6 Entity-Relationship Diagram and Information

The ER diagram for our schema can be found in Appendix B.

Our relations are indicatively in 3NF as each table contains uniquely identifiable rows, there are no partial dependencies, and there are no transitive dependencies.

Upon the creation of the `Albums` and `Artists` table, it was imperative to ensure that there were no partial dependencies, given the creation of new tables with a primary key composed of multiple attributes. Due to the possibility of multiple artists with the same name, each primary key in `Artists` is composed of a song ID and an artist name. As no artist can share the same song, this ensures that each attribute is dependent on both the unique song ID and artist. Although the attribute information is shared between entities with the same artist with different songs, partial dependencies are removed and entities are in 3NF. The same logic was applied for `Albums`, which similarly contains an album name and artist name as its composite primary key.

5. Web App Description

Web App Pages Description

5.1 Playlist Reccomender

This page displays pre-made Spotify playlists a user will enjoy based on their input. For instance, a user can enter their favorite artist, after which a table will be displayed containing links to playlists, sorted by number of followers (i.e. popularity), that the user will likely enjoy based on the style of their artist and the style of the songs within that playlist. A similar table is displayed should the user enter their favorite album.

Furthermore, the user can use sliders representing danceability, energy and tempo to obtain playlists matching these values most closely. Therefore, a user does not need to consider similar artists or albums to the style of music they desire, and can use these inputs to obtain playlists matching their style extremely closely.

5.2 Song Reccomender

This page allows users to enter their favorite artist or favorite song. Aggregate values regarding this artist or values of the inputted song are used to return songs that most closely match these values. Songs are displayed in tables, with each value in the table being a link to the corresponding song, album or artist page.

5.3 Search Playlist

This page allows users to search pre-made Spotify playlists by name e.g. “Workout,” allowing users to obtain playlists matching their use case or genre. Playlists are displayed in order of number of followers, as well as aggregate information on number of artists and songs. An individual playlist can be clicked, taking the user to the playlist information page. This page displays the track art of the first four tracks in the playlist, as well as links to the song information page, artist information page, and album information page of each song in the playlist.

5.4 Search Artist

This page allows users to enter the name of an artist, and displays all matching artist names. The user can click on an artist name to take them to the artist information page. Here, we display the Spotify artist image, as well as a list of their albums with corresponding album art and links to album info pages. Furthermore, we also display the links to other artist pages of artists similar to the current artist, to allow the user to discover new artists. We also display the links to playlists containing the most number of songs by this artist.

5.5 Search Album

This page allows users to enter the name of an album, and displays a table containing links to all corresponding albums. On the album page, we display the Spotify album art, as well as links to all songs within the album, and a link to the artist. Song links lead to a window containing the youtube video of the corresponding song. Furthermore, we display the danceability, energy and tempo of each song in the album, allowing the user to determine which songs in the album match their preferences.

5.6 Other Pages

5.6.1 Login Page

This page allows the user to make an account, or login to the page using previously created login credentials. This allows us to track users of the application, potentially for future marketing purposes.

5.6.2 Home Page

This page allows the user to see the artist image and album art of a variety of popular artists and albums featured on the web application, which can entice the user to delve deeper into the application.

5.6.3 About Us

This page displays an image of the creators of the web application, as well as some trivia about them such as name, major, year, and a link to their favorite artist.

6. API Specification

Please refer to Appendix C for the API routes.

7. Queries

Please refer to Appendix D for examples of queries within the web application and explanations of how they are used.

8. Performance Evaluation

Optimization Efforts Examples

In the table below, we showcase the effects of our optimization efforts on a selection of queries. Optimization methods and explanations behind the effects are below.

Example	Query	Pre-Optimization Time (ms)	Post-Optimization Time (ms)
1	Example Query 1	3750	740
2	Example Query 2	3840	2460
3	Example Query 3	739	453

Example 1

To speed up this query, we limited the result of the CTE to 1 tuple, thereby allowing this query to return as soon as it came across a tuple with attribute `track_name = '${songName}'`. Furthermore, through analysis using DataGrip's Explain Plan feature, we saw that scanning the entire `Songs` table to return the results was very costly. As such, we created an index `song_val_index_2` On `Songs (track_name, danceability, energy, valence)`. This prevents the query from having to perform a `Full Scan (Table Scan)` and instead allows the query to perform a `Full Index Scan` over the newly created index, which is a much faster process. As shown in the first row of the table, this greatly improved runtime efficiency.

Example 2

To speed up this query, as in example above, we limited the result of the first CTE to 1 tuple, thereby allowing this query to return as soon as it came across a tuple with attribute `artist_name = '${artistName}'`. Furthermore, we removed the DISTINCT keyword from the second CTE to ensure that we were not unnecessarily slowing down the CTE creation for no reason. Furthermore, through analysis using DataGrip's Explain Plan feature, we saw that scanning the entire `Songs` table to order the results was very costly. As such, we created an index `song_val_index_1` ON `Songs (track_name,artist_name, danceability, energy, valence, album_name, track_uri)` which allowed selected attributes to be returned much faster, using a Full Index Scan over `song_val_index_1` rather than a Full Table Scan over the Songs table. As shown in the second row of this table, this decreased the runtime of this query by over a second.

Example 3

This query optimization is largely due to the creation of the `Albums` relation. Instead of having to repeatedly calculate the average of each album's attributes for every query they are accessed, the intermediate results are stored in a new table. This removes the need for redundant calculations, greatly improving the runtime for this query and others used to access album-level information.

9. Technical Challenges

There were multiple technical challenges encountered throughout this project.

For the data-cleaning portion, the first obstacle to overcome was utilizing the JSON library, an unfamiliar tool that was learned through trial-and-error.

For data ingestion, a key problem was minimizing the runtime to upload the CSV files into each table. The TAs were extremely helpful in this step, advising to create new primary keys based on integer values to allow for increased uploading speed.

A key problem for entity resolution was how to consolidate the average attribute values for `Albums` and `Artists`, which was resolved through the creation of two new tables.

For the queries, it became necessary to craft a “distance” function to define similarity between different songs, albums, artists and playlists. In order to maximize the quality of recommendations while minimizing runtime, danceability, energy and valence were chosen as the metrics of interest in an absolute difference calculation. These attributes were primarily chosen as they represent a given entity’s overall mood over specific details such as instrumentation or liveness. The other attributes are still included in the database to allow for future analyses on song recommendations based on different features.

For optimizations, unfortunately some queries did not optimize as much as expected and required different types of optimization testing, which were completed through research, application and testing using DataGrip’s Explain Plan feature.

An immense number of problems occurred with the front-end development, which was a new tool for all members of the group. Through patience, resilience and the help of the TA’s, the development became possible.

For the Spotify API integration, there were challenges accessing a developer key, which was solved by using a “Bearer” token.

Overall, this project displayed the practical importance of optimization, as well as the creativity necessary to solve for it. Database design is equally an art as it is a science, a concept that could only be seen through the practical creation of one.

10. Extra Credit Features

API Integration

Our data set did not include images for the artist, albums, or playlist. So, when the initial front end was built the user experience was not optimal. As such, we integrated our pages with Spotify for Developer’s Web API to gather images for the respective data. The data from the queries were fetched and referenced as inputs to the Spotify API, and updated with the appropriate media data for each page. We used the *client id* and *secret* *client id* token to fetch the correct API token to generate the images. From there, the data resulting from the query was cross referenced to get the actual image and placed into the data array.

In addition, we leveraged YouTube's Data API. Originally, we wanted to play songs linked using Spotify's API, however they removed the API feature which enables audio to be played due to copyright infringement. Therefore, as an alternative, we obtained the corresponding song's YouTube video (either the official music video or the official track) to be played for the user. The song card component was called within a page by the song name, artist name, and album name and then inputted into YouTube's API query to retrieve the video link, which was displayed as a window over the application.

Appendix

Appendix A: Data

A.1: Data Sources

Spotify Million Playlist Dataset:

https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge

Spotify 1.2M+ Songs Dataset:

https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs

A.2: Data Attributes

A.2.1: Spotify Million Playlist Dataset

The original playlist attributes are:

- **PID:** The unique integer identifier of each playlist
- **Name:** The publicly-listed string name of the playlist
- **Collaborative:** A boolean variable representing if the datasets are shared between Spotify users or not
- **Modified_At:** The timestamp at which the playlist was last modified
- **Num_Tracks:** The integer number of tracks included in the playlist
- **Num_Albums:** The integer number of albums included in the playlist
- **Num_Followers:** The integer number of followers included in the playlist

The nested song attributes are:

- **Pos:** Integer position of a song in a given playlist
- **Artist_Name:** The string name of the song's artist
- **Track_Name:** The string name of the song
- **Track_URI:** The unique string identifier of a song in Spotify
- **Artist_URI:** The unique string identifier of an artist in Spotify
- **Album_URI:** The unique string identifier of an album in Spotify

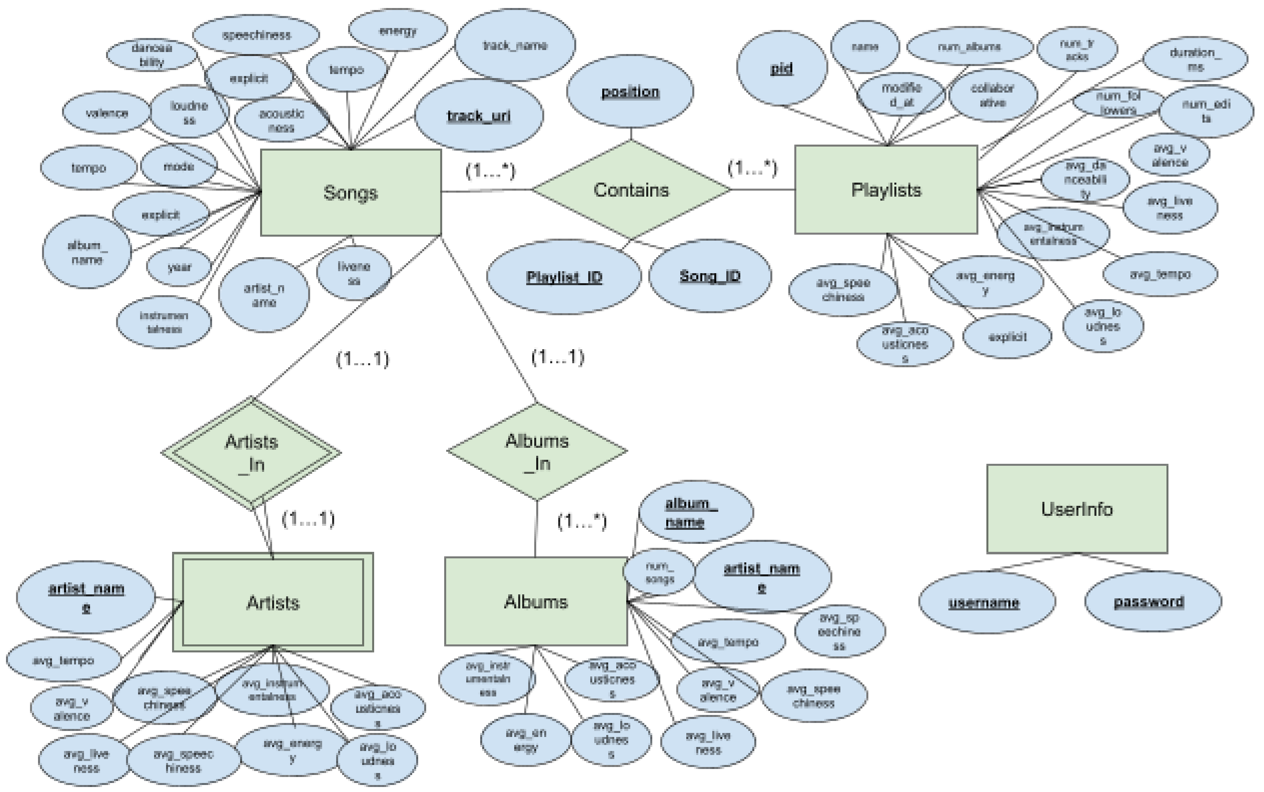
- **Album_Name**: The string name of the album of the song
- **Duration_MS**: The length of the song in milliseconds

A.2.2: Spotify 1.2M+ Songs Dataset

- **ID:** The unique string identifier of a song in Spotify
- **Name:** The string name of the song
- **Album:** The string name of the album of a song
- **Album_ID:** The unique string identifier of an album in Spotify
- **Artists:** A list of string names of the song's artists
- **Artist_IDs:** A list of unique string identifier of the song's artist in Spotify
- **Track_Number:** The integer representing the song's location in its album
- **Disc_Number:** The integer representing the disc the song can be played on in its album
- **Explicit**: A boolean variable representing whether the song contains offensive lyrics
- **Danceability:** A float value ranging from 0.0 to 1.0 that combines multiple other attributes to describe how easy the song is for a human to dance along to
- **Energy:** A float value ranging from 0.0 to 1.0 that perceptually measures intensity
- **Key:** An integer value that represents the song's key
- **Loudness:** A float value of the average loudness of a given track in decibels
- **Mode:** An integer value representing the modality of a given track
- **Speechiness:** A float value ranging from 0.0 to 1.0 representing the presence of spoken words in the given track
- **Acousticness:** A float value ranging from 0.0 to 1.0 representing a confidence measure of whether a track contains electrical amplification or not
- **Instrumentalness:** A float value ranging from 0.0 to 1.0 representing whether or not a track contains no vocal content, the inverse of speechiness
- **Liveness:** A float value ranging from 0.0 to 1.0 representing the probability that the song was performed live to an audience
- **Valence:** A float value ranging from 0.0 to 1.0 representing the musical positiveness conveyed by a given song
- **Tempo:** A float value for the estimated tempo in beats per minute (BPM)
- **Duration_MS:** The length of the song in milliseconds

- **Time_Signature:** Float value for how many beats are contained in a given measure on average
- **Year:** The year of the song's release
- **Release Date:** The data of the song's release

Appendix B: ER Diagram



Appendix C: API Specification

Parameters, unless specified, are required. Optional parameters are marked with an asterisk (*).

Route 1

Route: `/search_artist`
Description: returns the names of artists like a given artist name
Route Parameter(s): name*(string)
Query Parameter(s): *None*
Route Handler: search_artists(req, res)
Return Type: JSON array
Return Parameters: [{ artist_name (string) }, ...]
Expected (Output) Behavior: returns all distinct artist names in alphabetical order matching the artist name given

Route 2

Route: `/artist_albums`
Description: returns the albums by a given artist
Route Parameter(s): artist_name (string)
Query Parameter(s): page (int)*, page_size (int)* (default: 5)
Route Handler: artist_albums(req, res)
Return Type: JSON array
Return Parameters: [{ album_name (string) }, ...]
Expected (Output) Behavior: returns all distinct album names by a given artist, limiting/offsetting results by page/page size

Route 3

Route: `/artist_playlists`
Description: returns information about the playlists with the most number of songs by a given artist
Route Parameter(s): artist_name (string)
Query Parameter(s): *None*
Route Handler: playlists_with_artists(req, res)
Return Type: JSON array
Return Parameters: [{ pid (int), name (string), numSongs (int), num_track (int), num_followers (int) }, ...]
Expected (Output) Behavior: returns the top 5 playlists containing the most songs by a given artist in descending order

Route 4

Route: `/album`
Description: returns information about songs in a given album by a given artist
Route Parameter(s): album_name (string), artist_name (string)
Query Parameter(s): *None*
Route Handler: albums_songs(req, res)
Return Type: JSON array
Return Parameters: [{ track_name (string), danceability (float), energy (float), tempo (float) }, ...]
Expected (Output) Behavior: returns information about all songs in a given album by a given artist

Route 5

Route: `/album_info`
Description: returns the name of an album and its artist like a given album name and artist name
Route Parameter(s): album_name (string), artist_name (string)
Query Parameter(s): *None*
Route Handler: album_info(req, res)
Return Type: JSON object
Return Parameters: { album_name (string), artist_name (string) }
Expected (Output) Behavior: returns the album name and its artist name matching the given album name and artist name

Route 6

Route: `/search_album`
Description: returns information about albums with album name like a given album name
Route Parameter(s): albumName* (string)
Query Parameter(s): *None*
Route Handler: find_albums(req, res)
Return Type: JSON array
Return Parameters: [{ album_name (string), artist_name (string), num_songs (int), avg_danceability (float), avg_energy (float), avg_tempo (float) }, ...]
Expected (Output) Behavior: returns information about albums with album name like a given album name in alphabetical album name order

Route 7

Route: `/login`
Description: checks if a user and corresponding password exists
Route Parameter(s): username (string), password (string)
Query Parameter(s): *None*
Route Handler: login(res, req)
Return Type: response
Return Parameters: response (string)
Expected (Output) Behavior:

- If username and password in DB:
    - 'success' returned
- If username and password not in DB:
    - 'failure' returned
- If either route parameter was not entered:
    - status(400) sent, with text 'Please enter username and password'

Route 8

Route: `/create_user`
Description: inserts a new tuple into the userInfo table
Route Parameter(s): username (string), password (string)
Query Parameter(s): *None*
Route Handler: create_user(req, res)
Return Type: response
Return Parameters: response (string)
Expected (Output) Behavior:

- If tuple added successfully:
    - 'success' returned
- Else:
    - 'False', 'Error creating user' returned

Route 9

Route: `/search_playlist`
Description: returns information about playlists with playlist name matching a given playlist name
Route Parameter(s): name* (string)
Query Parameter(s): *None*
Route Handler: search_playlist(req, res)
Return Type: JSON array

Return Parameters: [{ pid (int), name (string), num_followers (int), num_artists (int), num_tracks(int) }, …]
Expected (Output) Behavior: returns distinct information about playlists with playlist name like a given playlist name, sorted by num_followers (descending), then by duration_ms (descending), then by name (alphabetical)

Route 10

Route: `/playlist_song`
Description: returns the informations about songs within a given playlist
Route Parameter(s): pid (int)
Query Parameter(s): *None*
Route Handler: playlist_song(req, res)
Return Type: JSON array
Return Parameters: [{ track_name (string), album_name (string), track_uri (int), artist_name (string) }, …]
Expected (Output) Behavior: returns information about all songs in a playlist with a given pid

Route 11

Route: `/playlist`
Description: returns information about a given playlist
Route Parameter(s): pid (int)
Query Parameter(s): *None*
Route Handler: playlist(req, res)
Return Type: JSON object
Return Parameters: { name (string), num_followers (int), collaborative (string) }
Expected (Output) Behavior: returns information about a playlist with a given pid

Route 12

Route: `/song`
Description: returns information about a given song
Route Parameter(s): track_uri (int)
Query Parameter(s): *None*
Route Handler: song(req, res)
Return Type: JSON array
Return Parameters: [{ track_name (string), album_name (string), artist_name (string) }, ..]

Expected (Output) Behavior: returns information about a song with a given track_uri

Route 13

Route: `/recommendSongs`
Description: returns songs a user may like based on a given song
Route Parameter(s): track_name (string)
Query Parameter(s): *None*
Route Handler: song_song_rec(req, res)
Return Type: JSON array
Return Parameters: [{ track_name (string), artist_name (string), track_uri (int), album_name (string) }, ..]
Expected (Output) Behavior: returns information on 5 songs a user may like based on a given song in descending order of similarity to the given song

Route 14

Route: `/recommendSong`
Description: returns songs a user may like based on a given artist
Route Parameter(s): artistName* (string)
Query Parameter(s): *None*
Route Handler: song_artist_rec(req, res)
Return Type: JSON array
Return Parameters: [{ track_name (string), album_name (string), artist_name (string), track_uri (int) }, ..]
Expected (Output) Behavior: returns information on 5 songs a user may like based on a given artist in descending order of similarity to songs by the given artist, not including songs by the given artist

Route 15
Route: `/aboutUs`
Description: returns all information about the application developers
Route Parameter(s): *None*
Query Parameter(s): *None*
Route Handler: people(res)
Return Type: JSON array
Return Parameters: [{ name (string), year (int), major (string), url (string), artist_name (string) }, ...]
Expected (Output) Behavior: returns information about all the application developers

Route 16

Route: `/recommendPlaylist`
Description: returns information about playlists a user may like based on a given artist
Route Parameter(s): artist_name (string)
Query Parameter(s): *None*
Route Handler: playlist_from_artist_vibe(req, res)
Return Type: JSON array
Return Parameters: [{ pid (int), name (string), num_followers (int), num_artists (int), num_tracks(int) }, ...]
Expected (Output) Behavior: returns information about the top 5 playlists that contain songs that on average most closely match the songs by a given artist, ordered by descending order of average similarity

Route 17

Route: `/recommendPlaylistAlbum`
Description: returns information on playlists containing the most number of songs from a given album
Route Parameter(s): album_name (string)
Query Parameter(s): *None*
Route Handler: playlist_album(req, res)
Return Type: JSON array
Return Parameters: [{ pid (int), name (string), num_followers (int), num_artists (int), num_tracks(int), numSongs (int) }, ...]
Expected (Output) Behavior:

Route 18

Route: `/recommendPlaylistInput`
Description: returns information on playlists most close in value to inputted values on danceability, energy, tempo
Route Parameter(s): avg_danceability* (float) (default: 0.61), avg_energy* (float) (default: 120.54), avg_tempo* (float) (default: 0.5)
Query Parameter(s): None
Route Handler: playlists_by_danceability
Return Type: JSON array
Return Parameters: [{ pid (int), name (string), num_followers (int), num_artists (int), num_tracks(int) }, ...]
Expected (Output) Behavior: returns information on the top 10 playlists most close in value to inputted values on danceability, energy, tempo ordered by similarity

Route 19

Route: `/album_slider`
Description: returns information about selected albums to display
Route Parameter(s): *None*
Query Parameter(s): *None*
Route Handler: album_slider
Return Type: JSON array
Return Parameters: [{ artist_name (string), album_name (string) }, ...]
Expected (Output) Behavior: returns information about selected albums to display

Route 20

Route: `/artist_slider`
Description: returns information about selected artists to display
Route Parameter(s): *None*
Query Parameter(s): *None*
Route Handler: artist_slider
Return Type: JSON array
Return Parameters: [{ artist_name (string) }, ...]
Expected (Output) Behavior: returns information about selected artists to display

Appendix D: Example Queries

Example Query 1

```
with cte_song AS (
        SELECT danceability, energy, valence
        FROM Songs
        WHERE track_name = '${songName}'
        LIMIT 1)
SELECT distinct track_name, artist_name, track_uri, album_name
FROM Songs song, cte_song
WHERE track_name <> '${songName}'
ORDER BY ABS ((song.danceability - cte_song.danceability)
         + (song.energy - cte_song.energy)
         + (song.valence - cte_song.valence)) DESC
LIMIT 5;
```

This query is used in Route 13 described above. This route and thus query is used within the Song Reccomender page.

Using user input consisting of a song name, the query created a common table expression (CTE) to obtain a tuple consisting of danceability, energy, and valence values from the Songs table where the track_name was the user's inputted song name. Furthermore, we limited the query to the first result to optimize the query as only 1 tuple was necessary.

From there, we created a cartesian product between the single tuple returned from the CTE and the Song table which did not increase the number of tuples beyond the number of tuples in the Songs table as the CTE table consisted of just 1 tuple. To prevent the query from returning the same song, we added the above where clause. Results were then ordered by the distance/similarity function we have been using throughout the application to determine suggestions. Furthermore, the results were limited to 5 to return a suitable number of suggestions.

Example Query 2

```
WITH cte_artists AS (
        SELECT artist_name, avg_danceability, avg_energy, avg_valence
        FROM Artists
        WHERE artist_name = '${artistName}'
        LIMIT 1),
cte_songs AS (
        SELECT track_name, artist_name, danceability, energy,
               valence, album_name, track_uri
        FROM Songs s
        WHERE artist_name <> '${artistName}')
SELECT DISTINCT s.track_name, s.album_name, s.artist_name,s.track_uri
FROM cte_songs s, cte_artists a
ORDER BY ABS((danceability - avg_danceability)
         + (energy - avg_energy) + (valence - avg_valence))
LIMIT 5;
```

This query is used in Route 14 described above. This route and thus query is used within the Song Reccomender page. Using user input consisting of an artist name, the query created 2 CTEs. The first, cte_artists, obtained a single tuple consisting of artist_name, avg_danceability, avg_energy and avg_valence values from the Artists table where the artist_name as the user's inputted artist name.

Furthermore, we limited the query to the first result to optimize the query as only 1 tuple was necessary.

The second CTE, cte_songs, obtained all relevant information from the Songs table of all tuples where the artist_name was not the user's inputted artist name. This allowed us to not return song tuples with the same artist name to give the best suggestions possible and allow the user to discover new songs not by artists they already like.

From there, we created a cartesian product of the two CTEs, which did not increase the number of tuples beyond the number of tuples in the cte_songs table as the cte_artists table consisted of just 1 tuple. Results were then ordered by the distance/similarity function we have been using throughout the application to determine suggestions. Furthermore, the results were limited to 5 to return a suitable number of suggestions.

Example Query 3

```sql
CREATE TABLE Albums (
   album_name            varchar(255) not null,
   artist_name           varchar(255) not null,
   avg_danceability      float        null,
   avg_energy            float        null,
   avg_loudness          float        null,
   avg_speechiness       float        null,
   avg_acousticness      float        null,
   avg_instrumentalness float        null,
   avg_liveness          float        null,
   avg_valence           float        null,
   avg_tempo             float        null,
   num_songs             int          null,
   primary key (album_name, artist_name),
   foreign key (album_name) references Songs (album_name),
   foreign key (artist_name) references Songs (artist_name)
);

INSERT INTO Albums (album_name, artist_name)
(SELECT DISTINCT album_name, artist_name FROM Songs);

UPDATE Albums
SET avg_danceability = (SELECT AVG(danceability) FROM Songs WHERE Songs.album_name
= Albums.album_name AND Songs.artist_name = Albums.artist_name),
   avg_energy = (SELECT AVG(energy) FROM Songs WHERE Songs.album_name =
Albums.album_name AND Songs.artist_name = Albums.artist_name),
   avg_loudness = (SELECT AVG(loudness) FROM Songs WEHERE Songs.album_name =
Albums.album_name AND Songs.artist_name = Albums.artist_name),
/* attributes between avg_loudness and avg_valence not shown to save space, follow
same format as above */
   avg_tempo = (SELECT AVG(tempo) FROM Songs WHERE Songs.album_name =
Albums.album_name AND Songs.artist_name = Albums.artist_name),
   num_songs = (SELECT COUNT(*) FROM Songs WHERE Songs.album_name =
Albums.album_name AND Songs.artist_name = Albums.artist_name);

SELECT album_name, artist_name, num_songs, avg_danceability, avg_energy, avg_tempo
FROM Albums
WHERE album_name LIKE '%${title}%'
ORDER BY album_name ASC;
```

This query is used in Route 6 described above. This route and thus query is used within the Search Album page. Initially, the query of returning the songs within an album, along with the average danceability, energy and tempo of the songs within an album was very complex and took substantial time.

Since these values do not change, we determined that it would be best to create a new table, Albums, containing the attributes seen in the above DDL. To set each value, we used aggregate functions over the Songs table.

Once this table was created and filled, we used it to return album information on albums with album_name like a user inputted album name, allowing the web application to display information about albums, despite us nt having them explicitly as a data set.

The Albums table was also subsequently used in other queries.

Example Query 4

```sql
CREATE TABLE Artists (
   artist_name          varchar(255) not null,
   song_id              int          not null,
   avg_danceability     float        null,
   avg_energy           float        null,
   avg_loudness         float        null,
   avg_speechiness      float        null,
   avg_acousticness     float        null,
   avg_instrumentalness float        null,
   avg_liveness         float        null,
   avg_valence          float        null,
   avg_tempo            float        null,
   primary key (song_id, artist_name),
   foreign key (song_id) references Songs (track_uri),
   foreign key (artist_name) references Songs (artist_name)
);

INSERT INTO Artists (song_id, artist_name)
(SELECT distinct track_uri, artist_name FROM Songs);

UPDATE Artists
SET avg_danceability = (SELECT AVG(danceability) FROM Songs WHERE Songs.artist_name
= Artists.artist_name),
   avg_energy = (SELECT AVG(energy) FROM Songs WHERE Songs.artist_name =
Artists.artist_name),
   avg_loudness = (SELECT AVG(loudness) FROM Songs WHERE Songs.artist_name =
Artists.artist_name),
/* attributes between avg_loudness and avg_valence not shown to save space, follow
same format as above */
   avg_tempo = (SELECT AVG(tempo) FROM Songs WHERE Songs.artist_name =
Artists.artist_name);

SELECT DISTINCT al.album_name FROM Albums al
INNER JOIN Artists ar ON ar.artist_name = al.artist_name
WHERE ar.artist_name = '${artist_name}'
LIMIT ${page_size}
OFFSET ${(page-1)*(page_size)};
```

This query is used in Route 2 described above. This route and thus query is used within the Search Artist page. Initially, the query of returning the songs of an artist, along with the average danceability, energy and tempo of the songs by an artist was very complex and took substantial time.

Since these values do not change, we determined that it would be best to create a new table, Artists, containing the attributes seen in the above DDL. To set each value, we used aggregate functions over the Songs table.

Once this table was created and filled, we used it to return artist information. In the above query, we join this table on the Albums table to return the albums of an artist with artist_name like a user inputted artist name, allowing the web application to display information about albums by a certain artist, despite us not having this explicitly as a data set. Tuples are selected based on the page size and page number inputs based on the current user page of the web application table.

The Artists table was also subsequently used in other queries.

Example Query 5

```
SELECT plays.pid, plays.name, COUNT(*) as numSongs, plays.num_tracks, plays.num_followers
FROM (Contains contain JOIN Songs song ON contain.Song_ID = song.track_uri)
     JOIN Playlists plays on contain.playlist_id = plays.pid
WHERE song.artist_name = '${artistName}'
GROUP BY plays.pid
ORDER BY numSongs DESC
LIMIT 5;
```

This query is used in Route 3 described above. This route and thus query is used within the Search Artist page. In this query, we join the Contains, Playlists, and Songs table, group by playlist id, and count the number of songs within a playlist by a user inputted artist. We order the results by this count in descending order to show the user the playlists that they may like best if they like this artist, and return the relevant playlist information and count. We limit the number of results to 5 to return a suitable number of suggestions.