Workshop in Data Management
Project's Software Documentation

Orel Zang[1]

[1] Dept. of Computer Science, Bar-Ilan University, Ramat-Gan, Israel

1 The Application

The application is essentially a music playlists generation system. The user will be able to find and discover recommendations for new songs he might like based on songs he already enjoy or based on a few supplied preferences. The application will generate a music playlist based on the given song or based on the supplied preferences. In order to use the application the user must first register an account using the application's registration form. After registration, the user will be able to sign in into his account in the main page. Once signed in, the user can generate new playlsits. The playlist generation algorithm comes in two flavors:

- **Song based generation:** the user can choose his favorite song and the application will generate a playlist of songs which best matches the supplied song audio features (e.g., tempo, loudness, etc).
- **Preferences based generation:** the user can choose if he wants happy or sad songs, the desired tag for the songs (e.g., rock, pop, jazz, etc) and the application will find him the song in the database which best matches his preferences.

For each generated playlist, the user can save it for later revisiting. Additionally, the user can see some interesting statistics details about each of the saved playlists: artists distribution, decades distribution, genres distribution, tempo progression, loudness progression and artists familiarity.

2 The Data

In order to develop a music playlists generation system I leveraged the "Million Song Dataset"[1], which is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. The core of the dataset is the feature analysis and metadata for one million songs, provided by The Echo Nest. Each song in the dataset contains many audio features such as the song's loudness, energy, key, tempo and much more. I developed an algorithm which ranks songs' similarity based on these features and then use this ranking in order to generate a playlist based on a given song or based on preferences. I determined, based on my basic music theory knowledge, which of the many features supplied by the Million Song Dataset will be the most beneficial for my algorithm and narrowed down the dataset to only these features, as described in 3.

The dataset files come in HDF5 format, which cannot be easily migrated into mysql databases. For this reason, I have implemented a python script which reads these files using the supplied wrappers by the "Million Song Dataset" and saves the required data in txt files. Then, using the LOAD TABLE SQL statement I was able to load these txt files into our database.

3 The Database Design

Figure 1 depicts the database schema. I will now elaborate about the different tables and columns (bold column denotes primary key and italic column denotes foreign key):

– **artists_table -** A table which contains the different artists in our database, each tuple corresponds to a single unique artist.
  - **artist_id:** a unique 36 characters string which identifies this artist.
  - artist_name: the name of the artist.
  - artist_familiarity: a value ranged from 0 to 1 which estimates how "familiar" the artist is.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - artist_id_unique: a standard index to enforce uniqueness on the artist_id column.
    - artist_name_index: this index is used to speed up the search of an artist's songs using his name in the application (in the song based playlist generator).

– **songs_table -** A table which contains the different songs in our database, each tuple corresponds to a single unique song (Note that each song can has different tracks associated with it, but in our case using one track for each song is quite enough as there are very slight audio differences between tracks associated with the same song, according to the Million song dataset documentation[1].)
  - **song_id:** a unique 18 characters string which identifies this song.
  - song_name: the name of the song.
  - *song_artist_id:* a FK which is the id of the song's artist.
  - song_album_name: album name from which the song was taken.
  - song_year: year when this song was released.
  - song_tempo: tempo of the song in BPM according to The Echo Nest.
  - song_danceability: danceability measure of this song according to The Echo Nest (between 0 and 1, 0 = not analyzed).
  - song_duration: duration of the song in seconds.
  - song_energy: energy measure according to The Echo Nest (between 0 and 1, 0 = not analyzed).

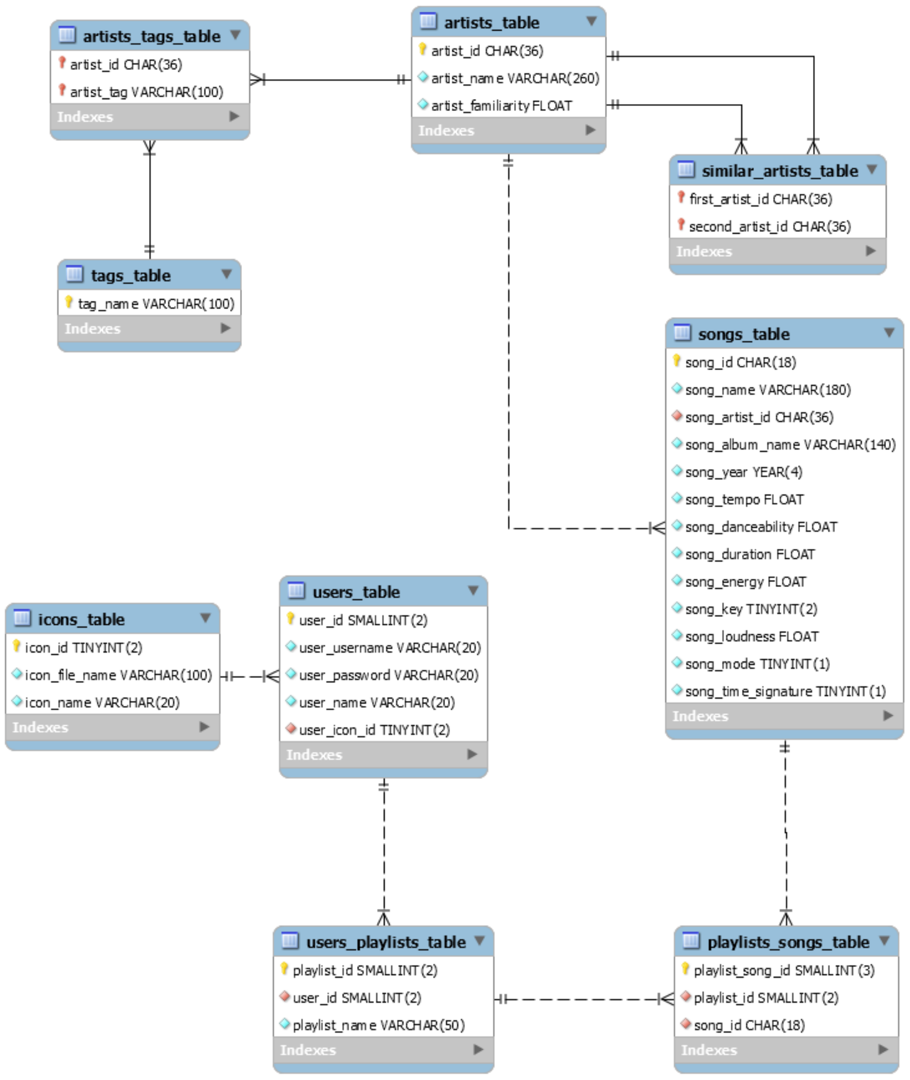

Fig. 1: The database schema

- song_key: estimation of the key the song is in by The Echo Nest.
- song_loudness: general loudness of the track.
- song_mode: a single bit which indicates the estimation of the mode the song is in by The Echo Nest (0 = minor, 1= major).
- song_time_signature: time signature of the song according to The Echo Nest, i.e. usual number of beats per bar.

  Indexes:
  * PRIMARY: a standard index on the primary key.
  * song_id_unique: a standard index to enforce uniqueness on the song_id column.
  * song_artist_id_idx: this index is used to speed up the search of songs which are performed by a specific artist in the application (in the song based playlist generator).
  * song_name_idx: this index is used to speed up the search of songs which are performed by a specific artist in the application (in the song based playlist generator).
  * song_mode_idx: this index is used to speed up the search of songs which have a specific mode in the application (used in the generation algorithms).
  * song_time_signature_idx: this index is used to speed up the search of songs which have a specific time signature in the application (used in the generation algorithms).

– **tags_table** - A table which contains the different tags which an artist may be associated with them, each tuple corresponds to a single unique tag.
- **tag_name:** the name of the tag (e.g, pop, rock, jazz, etc).

  Indexes:
  * PRIMARY: a standard index on the primary key.
  * tag_name_unique: a standard index to enforce uniqueness on the tag_name column.

– **artists_tags_table** - A connector table which indicating the many-to-many relation between artists and tags (an artist may have many tags and a single tag may be associated with many artists), each tuple corresponds to an artist and his tag.
- ***artist_id:*** a FK which is the id of the artist this tag associated with.
- ***artist_tag:*** a FK which is the name of the tag this artist associated with.

  Indexes:
  * PRIMARY: a standard index on the primary key.
  * artist_id_tag_unique: a standard index to enforce uniqueness on the artist_tag column.
  * artist_tag_idx: this index is used to speed up the search of artists associated with a specific tag in the application (in the preferences based playlist generator).

- **similar_artists_table** - A table which contains pairs of artists who are similar, note that this relation is bidirectional (if artist x is similar to artist y then artist y is similar to artist x):
  - ***first_artist_id:*** a FK which is the id of an artist.
  - ***second_artist_id:*** a FK which is the id of an another artist which is similar to the first one.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - artists_ids_unique: a standard index to enforce uniqueness on the first_artist_id and second_artist_id columns.
- **users_table** - A table which contains the different registered users in the application, each tuple corresponds to a single unique user:
  - **user_id:** a unique auto-increment number which identifies this user.
  - user_username: the username of the user in the system.
  - user_password: the password of the user in the system.
  - user_name: the name of the user in the system.
  - *user_icon_id*: a FK which is the id of the user's icon image in the system.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - user_id_unique: a standard index to enforce uniqueness on the user_id column.
    - user_username_unique: a standard index to enforce uniqueness on the user_username column.
    - username_index: this index is used to speed up the search for a user in the database using a username in the login process.
    - password_index: this index is used to speed up the search for a user in the database using a password in the login process.
- **icons_table** - A table which contains the icons in the application which the user can choose from to be his avatar, each tuple corresponds to a single unique icon (the relation between a user and an icon is one-to-one - each user has one icon):
  - **icon_id:** a unique auto-increment number which identifies this icon.
  - icon_file_name: the name of icon's file.
  - icon_name: the name of the icon to as it will be presented in the application.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - icon_id_unique: a standard index to enforce uniqueness on the icon_id column.
    - icon_file_name_unique: a standard index to enforce uniqueness on the icon_file_name column.
    - icon_name_unique: a standard index to enforce uniqueness on the icon_name column.

- **users_playlists_table** - A table which contains the different playlists which the users created in the application, each tuple corresponds to a single unique playlist (each playlist belongs to a single user):
  - **playlist_id:** a unique auto-increment number which identifies this playlist.
  - ***user_id:*** a FK which is the id of the user this playlist belongs to.
  - playlist_name: the name of the playlist which will be displayed in the application.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - playlist_id_unique: a standard index to enforce uniqueness on the playlist_id column.
    - user_id_idx: this index is used to speed up the search for a user's playlists in the applications.
- **playlists_songs_table** - A connector table which indicating the many-to-many relation between songs and playlists (a playlist may have many songs in it and a single song may be in many playlist), each tuple corresponds to a playlist and its song.
  - **playlist_song_id:** a unique auto-increment number which identifies this specific song in this specific playlist.
  - *playlist_id:* a unique auto-increment number which identifies this playlist.
  - *song_id:* a FK which is the id of the song which belongs to this playlist.

    Indexes:
    - PRIMARY: a standard index on the primary key.
    - playlist_song_id_unique: a standard index to enforce uniqueness on the playlist_song_id column.
    - playlist_song_unique: a standard index to enforce uniqueness on the playlist_id column.
    - song_id_idx: this index is used to speed up the search for a user's playlists' songs in the applications.

4 The Algorithms

Algorithm 1 shows the algorithm of the song based playlist generator. It takes as input a reference song and the number of songs required to be in the generated playlist. First, the algorithm gets a candidate songs list based on the given song. Specifically, using a SQL query statement we get a list of artists which are considered similar to the artist of the reference song, we get these artists' songs and keep only those songs which have the same mode and time signature as the reference song. Then, for each of these candidate songs we calculate the "similarity score". This score is based on the tempo distance and loudness distance between the reference song and the current song. This score is also effected by the familiarity of the current song as we want to push songs with more familiar artists to the top. Then, we sort in decrease order the candidate songs list based

Algorithm 1 Song Based Playlist Generator Algorithm

```
 1: procedure SongBasedPlaylistGenerator(referenceSong, numberOfSongsIn-
    Playlist)
 2:     candidateSongs $\leftarrow$ getCandidateSimilarSongs(referenceSong)
 3:     for each song $\in$ candidateSongs do
 4:         tempoSimilarity $\leftarrow$ 1-(abs(getNormalizedTempo(referenceSong) - getNormalizedTempo(song)))
 5:         loudnessSimilarity $\leftarrow$ 1-(abs(getNormalizedLoudness(referenceSong) - getNormalizedLoudness(song)))
 6:         similarityScore $\leftarrow$ 0.4*tempoSimilarity+0.4*loudnessSimilarity+0.2*song.getArtist().getFamiliarity()
 7:         song.setSimilarityScore(similarityScore)
 8:     candidateSongs.sort(By similarityScore).reversed()
 9:     i $\leftarrow$ 0
10:     while i<numberOfSongsInPlaylist and i<candidateSongs.size() do
11:         add candidateSongs[i] to generatedPlaylist
12:         i++
13:     return generatedPlaylist
```

Algorithm 2 Preferences Based Playlist Generator Algorithm

```
 1: procedure PreferencesBasedPlaylistGenerator(genre, happinessValue,
    numberOfSongsInPlaylist)
 2:     candidateSongs $\leftarrow$ getCandidateSimilarSongs(happinessValue,genre)
 3:     for each song $\in$ candidateSongs do
 4:         songTempo $\leftarrow$ getNormalizedTempo(song)
 5:         songLoudness $\leftarrow$ getNormalizedLoudness(song)
 6:         happinessScore $\leftarrow$ 0.5*tempoSimilarity+0.3*loudnessSimilarity+0.2*song.getArtist().getFamiliarity()
 7:         song.setHappinessScore(happinessScore)
 8:     candidateSongs.sort(By happinessScore)
 9:     if happinessValue <0 then
10:         candidateSongs $\leftarrow$ candidateSongs.reversed()
11:     i $\leftarrow$ happinessValue/100 * (candidateSongs.size()-1)
12:     while i >=0 and i <candidateSongs.size() and generatedPlaylist.size()!=numberOfSongsInPlaylist
    do
13:         add candidateSongs[i] to generatedPlaylist
14:         i-=1
15:     return generatedPlaylist
```

on the similarity score. Finally, we iterate on the sorted candidate songs list and add the first k songs in it (based on the numberOfSongsInPlaylist parameter) to the generated playlist and return this playlist in the end.

Algorithm 2 shows the algorithm of the preferences based playlist generator. It takes as input the required genre, the required happiness value (which ranges from -100 as the saddest to 100 as the happiest) and the number of songs required to be in the generated playlist. Overall this algorithm is similar to the previous one except for these changes: 1) The candidate songs are the ones which has the required genre tag associated with them. 2) The candidate songs list is sorted by the happiness score which is calculated proportionally to the song's tempo and loudness. 3) Instead of taking the first k songs in the candidate songs list we are taking the k songs which are roughly the requiredHappiness% of the candidate songs list.

5 Implementation

I have implemented the application in the Java programming language using the MVC (Model-View-Controller) architectural design pattern. JavaFX was used as the platform for the GUI building.

In my implementation I have used the following packages:

– **JDBC** - JDBC was used as the API for accessing the database in the application.
– **JFoenix**[1] - JFoenix is an open source Java library, that implements Google Material Design using Java components. This was used for our GUI elements.
– **FontAwesomeFX**[2] - FontAwesomeFX is an open source Java library, which gives you scalable vector icons that can be customized. This was used for our GUI design.

References

1. T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[1] http://www.jfoenix.com/
[2] https://bitbucket.org/Jerady/fontawesomefx