Katie Kim (kjk5drb), Andrew Shin (ajs5qgm), Gabriella Woo (gdw5bv), Winston Zhang (wyz5rge)
CS 4750: Database Systems
29. September, 2023

CS 4750 Semester Project Checkpoint 1

Introduction

Checkpoint 1 is one of four deliverables for the semester project. The purpose of checkpoint 1 is to develop an Entity-Relationship model based off of the functional requirements set forth in the project proposal. This is first done by drawing up an E-R diagram, which is then used to extrapolate a set of tables, expressed as schema statements, which describe the set of entity sets and relation sets in the form of text. Functional dependencies are then identified, and tables are decomposed and normalized.

Project Amendments

Since the submission of the project proposal, several changes have been made to expand the scope of the project such that the implemented application will make use of enough tables to adhere to assignment requirements.

The changes are enumerated and explained:

1. **UI Customization**

   The original project proposal suggested that the app UI be text-based to mimic a UNIX shell, with text being rendered in green against a black background. It was decided that only offering green-on-black UI was uninspired and too basic. Moving forward, the application will give the user the option to choose different UI themes if they'd rather die than look at a green-on-black terminal. These are added as attributes to the user table.

2. **Song Recommendations**

   We have decided to implement a way for the application to recommend songs to the user. Recommendations will be generated for users via Spotify API call. This is a weak entity, as recommendations are personalized. Currently, we are undecided on how exactly recommended songs are fetched using the API, as it takes a wide range of parameters, such as songs, artists, and albums. We are also undecided on how these recommendations are shown to the user, but suggested methods are to display them automatically as a dialogue on login, as well as via a terminal command manually invoked by the user. An example is shown below:

```
Welcome to EchoShell!
Logged in as: [username]

For a list of navigation commands, please use the 'help' command
For a list of EchoShell commands, please use 'echoshell --help'

Today's song recommendations:
# song recommendations listed line-by-line here

[user@echoshell] $
```

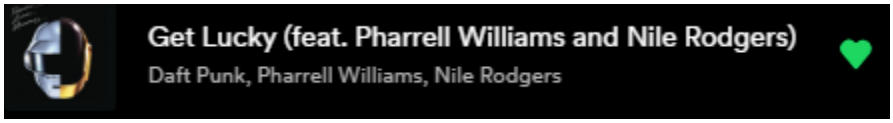
3. **Artists**

Users are able to follow artists on Spotify, which allows for easier access to those artists' content. To emulate this, we add an artist entity, as a relation set that tracks which users are following which artists. This is separate from Spotify followers, and following an artist in our application will not make an API call to Spotify to cause a user to actually follow them on Spotify. This is dictated by our decision to not integrate actual Spotify accounts for data security concerns.

Artists function like playlists and will be displayed in a user's library, showing that artist's most popular songs. Users cannot edit an artist list.

4. **Liked Songs**

Spotify offers several options for users to save tracks. In addition to adding a song to a playlist, Spotify has a heart-shaped icon next to songs that users can click to "like", as shown:



"Liking" a song in this context adds that song to a user's "Liked Songs", which functionally behaves just like a playlist.

In Spotify, the Liked Songs list has several special behaviors, such as the fact that it is displayed at the top level of the user's library and cannot be moved into a folder, as well as the ordering of songs, which is last-in-first-out ordered, just like a stack data structure. These behaviors are less related to the structure of the database, and more the business logic of the application, so it is difficult to note these differences in schema statements, as the same attributes and datatypes are used.

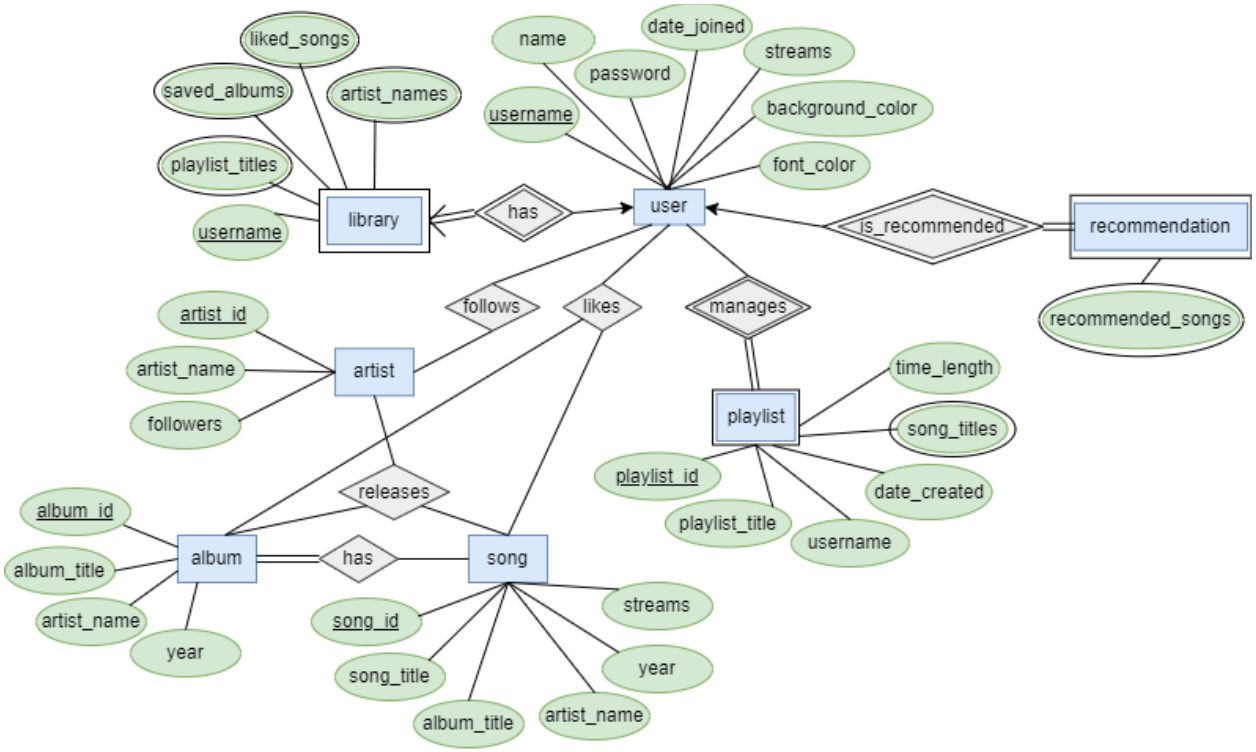
5. **Albums**

Albums exist as another way to organize songs so the user can discover more through the application. Because there exist instances in which songs belong to multiple albums, albums and songs are modeled in a many-to-many relation. This is purely for backend purposes, as songs will appear to only belong to one album on the frontend, which is the same way Spotify handles this.

Similarly to songs, Spotify allows users to like albums, which saves them to the user's library. Our application will allow users to do the same. Albums will be displayed in a user's library just like a playlist, but just like an artist list, cannot be edited or reordered in any way by users.

1. E-R Diagram

An E-R diagram in accordance with the specifications made in the project proposal, in addition to the project amendments mentioned above:



2. Schema Statements

A collection of tables were derived from the E-R diagram that was created in Part 1. These tables are listed as follows, in the form of schema statements:

user(username, name, password, date_joined, streams, background_color, font_color)
follows(username, artist_id)
likes(username, release_id)
manages(username, playlist_id)

recommendations(username, song_id)

song(song_id, song_title, artist, album_title, year, streams, album_id)
album(album_id, album_title, artist, year)
album_has(album_id, song_id)

artist(artist_id, artist_name, spotify_followers)
releases(artist_id, release_id)

playlist(playlist_id, playlist_title, owner_username, date_created, time_length, song_titles)
playlist_songs(playlist_id, song_id)

library(username, playlists, artists, liked_songs, saved_albums)
library_playlists(username, playlist_id)
library_artists(username, artist_id,)
library_liked_songs(username, song_id)
library_saved_albums(username, album_id)

Many of the primary keys for the tables, such as song ID, album ID, and artist ID, are pre-existing in Spotify's web databases and will be fetched via API calls in order to populate our tables.

As previously mentioned, following an artist in our application is separate from following on Spotify. The *followers* attribute in the *artist* schema statement refers to the number of followers that an artist has on Spotify, which will be fetched via API call and displayed in UI. Following an artist in our application does not change this number.

The *releases* relation set for artists, although a multiway relation in the E-R diagram, is treated as a binary relationship in the schema statements. This is because Spotify uses the same naming convention when creating string IDs for songs and albums. This mitigates the headache of modeling multiway relations, and then determining if an API call returns a song versus an album can be handled in the business logic by examining the contents of the JSON object returned by the API call.
Users liking songs and albums will also be treated like a binary relation despite being modeled like a multiway relation for the same reason.

3. Normalization

We decomposed our tables using 3NF.

user(<u>username</u>, name, password, date_joined, streams, background_color, font_color)
FD = {username -> name, password, date_joined, streams, background_color, font_color}

1. Write all LHS, copy FD as is
    - username -> name, password, date_joined, streams, background_color, font_color
2. Remove reflexivity, remove extraneous
    - username -> name, password, date_joined, streams, background_color, font_color
3. Normalized table:
    - R(username, name, password, date_joined, streams, background_color, font_color)

follows(<u>username</u>, <u>artist_id</u>)
FD = {username -> artist_id}

1. Write all LHS, copy FD as is
    - username -> artist_id
2. Remove reflexivity, remove extraneous
    - username -> artist_id
3. Normalized table:
    - R(username, artist_id)

likes(<u>username</u>, <u>release_id</u>)
FD = {username -> release_id}

1. Write all LHS, copy FD as is
    - username -> release_id
2. Remove reflexivity, remove extraneous
    - username -> release_id
3. Normalized table:
    - R(username, release_id)

manages(<u>username</u>, <u>playlist_id</u>)
FD = {username -> playlist_id}

1. Write all LHS, copy FD as is
    - username -> playlist_id
2. Remove reflexivity, remove extraneous
    - username -> playlist_id
3. Normalized table:
    - R(username, playlist_id)

recommendations(<u>username</u>, song_id)
FD = {username -> song_id}

1. Write all LHS, copy FD as is
    - username -> song_id
2. Remove reflexivity, remove extraneous
    - username -> song_id
3. Normalized table:
    - R(username, song_id)

song(<u>song_id</u>, song_title, artist, album_title, year, streams, album_id)
FD = {artist -> song_title, album_title; song_id -> artist, song_title, year, streams; album_id -> artist, album_title}

1. Write all LHS, copy FD as is
    - artist -> song_title, album_title
    - song_id -> artist, song_title, year, streams
    - album_id -> artist, album_title
2. Remove reflexivity, remove extraneous
    - artist -> song_title, album_title
    - song_id -> artist, song_title, year, streams
    - album_id -> artist, album_title
3. Normalized tables: artist, song_title, album_title // song_id, artist, song_title, year, streams // album_id, artist, album_title

album(<u>album_id</u>, album_title, artist, year)
FD = {album_id -> album_title, year, artist}

1. Write all LHS, copy FD as is
    - album_id -> album_title, year, artist
2. Remove reflexivity, remove extraneous
    - album_id -> album_title, year, artist
3. Normalized table:
    - R(album_id, album_title, year, artist)

album_has(<u>album_id</u>, <u>song_id</u>)
FD = {album_id -> song_id}

1. Write all LHS, copy FD as is
    - album_id -> song_id
2. Remove reflexivity, remove extraneous
    - album_id -> song_id
3. Normalized table: R(album_id, song_id)

artist(<u>artist_id</u>, artist_name, spotify_followers)
FD = {artist_id -> artist_name, spotify_followers; artist_name -> spotify_followers)

1. Write all LHS, copy FD as is
    - artist_id -> artist_name, spotify_followers
    - artist_name -> spotify_followers
2. Remove reflexivity, remove extraneous
    - artist_id -> artist_name, ~~spotify_followers~~
    - artist_name -> spotify_followers
3. Normalized table: artist_id, artist_name // artist_name, spotify_followers

releases(artist_id, release_id)
FD = {artist_id -> release_id}
1. Write all LHS, copy FD as is
    - artist_id -> release_id
2. Remove reflexivity, remove extraneous
    - artist_id -> release_id
3. Normalized table: R(artist_id, release_id)

playlist(playlist_id, playlist_title, owner_username, date_created, time_length, song_titles)
FD = {playlist_id -> playlist_title, owner_username, date_created, time_length, song_titles;
playlist_title -> song_titles}
1. Write all LHS, copy FD as is
    - playlist_id -> playlist_title, owner_username, date_created, time_length, song_titles
    - playlist_title -> song_titles
2. Remove reflexivity, remove extraneous
    - playlist_id -> playlist_title, owner_username, date_created, time_length, song_titles
3. Normalized table:
    - R(playlist_id, playlist_title, owner_username, date_created, time_length, song_titles)

playlist_songs(playlist_id, song_id)
FD = {playlist_id -> song_id)
1. Write all LHS, copy FD as is
    - playlist_id -> song_id
2. Remove reflexivity, remove extraneous
    - playlist_id -> song_id
3. Normalized table: R(playlist_id, song_id)

library(username, playlists, artists, liked_songs, saved_albums)
FD = {username -> playlists, artists, liked_songs, saved_albums}
1. Write all LHS, copy FD as is
    - username -> playlists, artists, liked_songs, saved_albums
2. Remove reflexivity, remove extraneous
    - username -> playlists, artists, liked_songs, saved_albums
3. Normalized table: R(username, playlists, artists, liked_songs, saved_albums)

library_playlists(username, playlist_id)
FD = {username -> playlist_id}
1. Write all LHS, copy FD as is
    - username -> playlist_id
2. Remove reflexivity, remove extraneous
    - username -> playlist_id

3. Normalized table: R(username, playlist_id)

library_artists(<u>username</u>, <u>artist_id</u>)
FD = {username -> artist_id}
1. Write all LHS, copy FD as is
    - username -> artist_id
2. Remove reflexivity, remove extraneous
    - username -> artist_id
3. Normalized table: R(username, artist_id)

library_liked_songs(<u>username</u>, <u>song_id</u>)
FD = {username -> song_id}
1. Write all LHS, copy FD as is
    - username -> song_id
2. Remove reflexivity, remove extraneous
    - username -> song_id
3. Normalized table: R(username, song_id)

library_saved_albums(<u>username</u>, <u>album_id</u>)
FD = {username -> album_id}
1. Write all LHS, copy FD as is
    - username -> album_id
2. Remove reflexivity, remove extraneous
    - username -> album_id
3. Normalized table: R(username,album_id)

Normalized Tables
- user(username, name, password, date_joined, streams, background_color, font_color)
- follows(username, artist_id)
- likes(username, release_id)
- manages(username, playlist_id)
- recommendations(username, song_id)
- owns(artist, song_title, album_title)
- song(song_id, artist, song_title, year, streams)
- album(album_id, album_title, year, artist)
- album_has(album_id, song_id)
- artist(artist_id, artist_name)
- artist(artist_name, spotify_followers)
- releases(artist_id, release_id)
- playlist(playlist_id, playlist_title, owner_username, date_created, time_length, song_titles)
- playlist_songs(playlist_id, song_id)
- library(username, playlists, artists, liked_songs, saved_albums)
- library_playlists(username, playlist_id)
- library_artists(username, artist_id)
- library_liked_songs(username, song_id)
- library_saved_albums(username,album_id)

Honor Pledge

On our honor as students, we have neither given nor received unauthorized aid on this assignment.

Katie Kim (kjk5drb),
Andrew Shin (ajs5qgm),
Gabriella Woo (gdw5bv),
Winston Zhang (wyz5rge)

29. September, 2023