BLG317E Database Systems
Project Documentation

Berke Kurt 150210329
Efe Can Kırbıyık 150220768

January 5, 2025

Contents

1 Project Idea

Our project aims to develop a database system for music streaming platforms to manage and organize large amounts of data including music content, artist and group information, personal playlists, and user interactions. The system will handle artist profiles, music tracks, albums, playlists, and user engagement metrics. The database will have several interconnected tables that establish proper relationships to ensure data integrity and efficient query capabilities. The structure will support essential CRUD operations such as adding new artists and songs, tracking user engagement metrics, and updating play counts.

2 ER Diagram

The ER diagram and ER model are shown in the figures below.

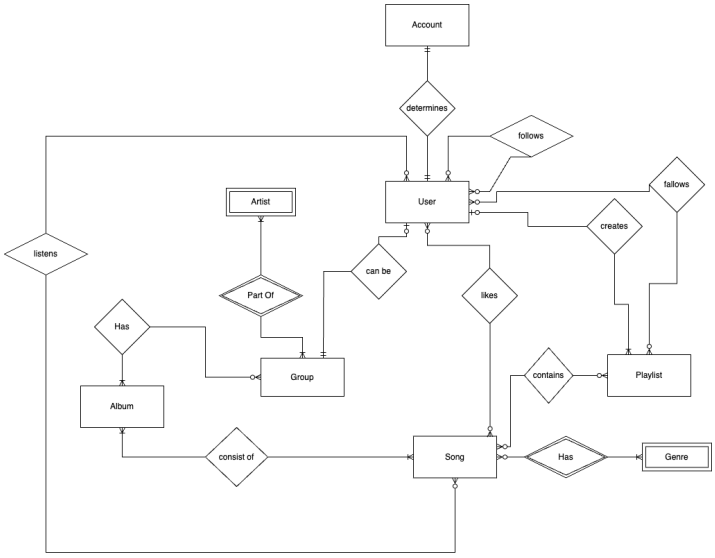

Figure 1: ER Diagram

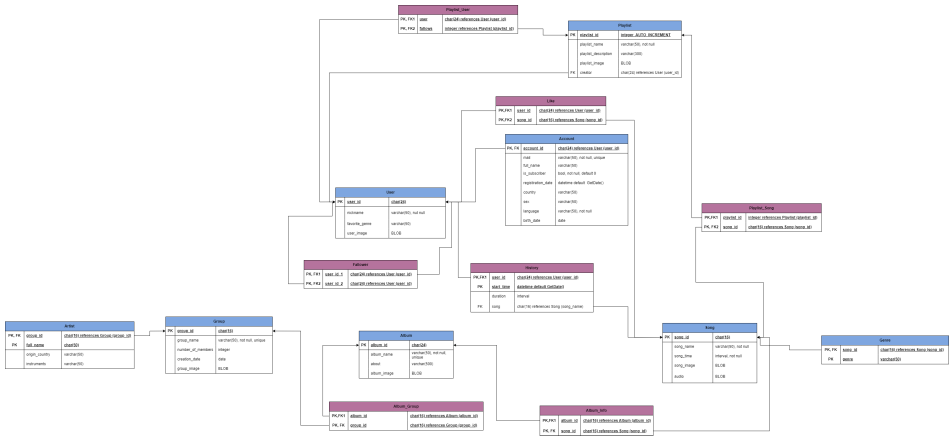

Figure 2: Data Model

2.1 Relationships

The names, cardinalities, and modalities of the relations are shown below.

1. One-to-One Relationships

- **Entities:** Account ↔ User
  - **Name:** is a
  - **Cardinality:** 1:1
  - **Modality:** Mandatory on both sides (each account must have one user, and each user must belong to one account).
- **Entities:** Group ↔ User
  - **Name:** is a
  - **Cardinality:** 1:1
  - **Modality:** Mandatory on group side, (each group must be a user), optional on the user side (a user may not be a group).

2. One-to-Many Relationships

- **Entities:** User ↔ Playlist
  - **Name:** creates
  - **Cardinality:** 1:N
  - **Modality:** Mandatory on the playlist side (each playlist must be created by a user), optional on the user" side (a user may not be created any playlists).

3. Many-to-Many Relationships

- **Entities:** User ↔ Song
  - **Name:** listens
  - **Cardinality:** N:N
  - **Modality:** Optional on both sides (each user may listen many songs and each song may be listened by many users).
- **Entities:** Album ↔ Group
  - **Name:** has
  - **Cardinality:** N:N
  - **Modality:** Mandatory on the album side (each album must belong to a group), optional on the group side (a group may or may not has albums).
- **Entities:** Album ↔ Song
  - **Name:** consist of
  - **Cardinality:** N:N
  - **Modality:** Mandatory on both sides (each song must belong to an album and each album must consist of one or more songs).
- **Entities:** Song ↔ Playlist
  - **Name:** contains
  - **Cardinality:** N:N

  - **Modality:** Optional on both sides (each playlist may contain one or more songs and each song may be in several playlists).

- **Entities:** User ↔ Playlist
  - **Name:** follows
  - **Cardinality:** N:N
  - **Modality:** Optional on both sides (each user may follow one or more playlists and each playlist may be followed by one or more users).

- **Entities:** Song ↔ Genre (weak entity)
  - **Name:** has (weak relation)
  - **Cardinality:** N:N
  - **Modality:** Mandatory on genre side (each genre must belong to one or more songs), optional on song side(a song may or may has one or more genres).

- **Entities:** Group ↔ Artist (weak entity)
  - **Name:** part of (weak relation)
  - **Cardinality:** N:N
  - **Modality:** Mandatory on both sides (each artist must belong to one or more groups and each group should contain at least one artist).

2.2 Entities And Their Attributes

- **Account:** account_id, mail, full_name, is_subscriber, registration_date, country, sex, language, birth_date
- **User:** user_id, nickname, favorite_genre, user_image
- **Playlist:** playlist_id, playlist_name, playlist_description, playlist_image, creator
- **Group:** group_id, group_name, number_of_members, creation_date, group_image
- **Album:** album_id, album_name, about, album_image
- **Song:** song_id, song_name, song_time, song_image, audio
- **Genre (weak entity):** genre
- **Artist (weak entity):** full_name, origin_country, instrument

3 Data Model

Data model is shown in figure 2. The descriptions and purposes of the tables are explained in the following.

- **Account:** Stores detailed account information such as email, subscription status, and personal details.
- **User:** Contains basic user information, including nickname and favorite genre.
- **Playlist:** Holds data on playlists created by users, including descriptions and images.
- **Group:** Stores details about music groups or bands, including their creation date and image.
- **Album:** Contains information on music albums, including album name and image.
- **Song:** Includes details about songs such as name, duration, and audio file.
- **Genre:** Associates songs with their genres.
- **Artist:** Contains information on individual artists, including their instruments.
- **Follower:** Manages user follow relationships.
- **History:** Logs user listening history.
- **Like:** Records which songs are liked by which users.
- **Playlist_Follower:** Tracks which users follow which playlists.
- **Playlist_Song:** Links songs to playlists, indicating which playlist contains which songs.
- **Album_Info:** Links songs to their respective albums, indicating which album contains which songs.
- **Album_Group:** Links albums to groups, indicating which group released which album.

4 Examples of Complex Queries

- **Get follower count for each customer:**

  This query calculates the number of followers for each user. It uses a subquery to count the followers for each user ID in the `Fallower` table and then performs a LEFT JOIN with the `User` table to include users with no followers.

```
SELECT User.user_id, f.fallower_count
FROM User
LEFT JOIN (
    SELECT user_id_2, COUNT(*) AS fallower_count
    FROM Fallower
    GROUP BY user_id_2
) AS f
ON User.user_id = f.user_id_2;
```

- **Get total listening time of each album in descending order:**

  This query calculates the total listening time for each album in the database. It first joins the `Album` table with the `Album_Info` table to associate albums with their songs. Then, it computes the total listening time for each song by summing the durations recorded in the `History` table. Finally, the query aggregates the listening times for all songs in each album, grouping them by album ID and name, and orders the results in descending order of total listening time. Albums with no listening records are included with a total listening time of zero due to the use of a `LEFT JOIN`.

```
SELECT albums.album_name AS album_name,
       SUM(stream.total_listen_time) AS total_listen_time
FROM (
    SELECT Album.album_id, Album.album_name, Album.song_id
    FROM Album
    LEFT JOIN Album_Info
    ON Album.album_id = Album_Info.album_id
) AS albums
LEFT JOIN (
    SELECT Song.song_id, Song.song_name,
           SUM(History.duration) AS total_listen_time
    FROM History
    JOIN Song ON History.song = Song.song_id
    GROUP BY Song.song_id
) AS stream
ON albums.song_id = stream.song_id
GROUP BY albums.album_id, albums.album_name
ORDER BY total_listen_time DESC;
```

- **Find most listened genre in the last month:**

  This query finds the most listened genre during the previous month by summing the listening durations from the `History` table for each genre. It filters data for the desired time period, groups results by genre, and orders them in descending order of total listening time, returning the genre with the highest total duration.

```
SELECT g.genre, SUM(h.duration) AS total_listen_time
FROM History h
JOIN Song s ON h.song = s.song_name
JOIN Genre g ON s.song_id = g.song_id
WHERE h.start_time >= GetDate() - INTERVAL "2 month"
  AND h.start_time <= GetDate() - INTERVAL "1 month"
GROUP BY g.genre
ORDER BY total_listen_time DESC
LIMIT 1;
```

5 CRUD operations and API design

The project implements a RESTful API using Flask-RESTX framework to manage the database. The system features a modular architecture with SQLite as the database backend, JWT authentication for security, and well-defined models for data validation. The API provides standard CRUD operations across various entities, ensuring a consistent and secure interface for the music streaming service.

5.1 Authentication Operations

```

POST /auth/login
  - Login and get a token

POST /auth/register
  - Register a new user
```

5.2 Account Management

```
POST /accounts/
  - Create a new account

GET /accounts/
  - Get all accounts

GET /accounts/{account_id}
  - Get an account by ID

PUT /accounts/{account_id}
  - Update an account

DELETE /accounts/{account_id}
  - Delete an account
```

5.3 User Management

```
POST /users/
  - Create a new user

GET /users/
  - Get all users

GET /users/follower-counts
  - Get all users with their follower counts

GET /users/nickname/{nickname}
  - Get a user by nickname

GET /users/{user_id}
  - Get a user by ID

PUT /users/{user_id}
  - Update a user

DELETE /users/{user_id}
  - Delete a user
```

5.4 Follower Management

```
POST /followers/
  - Create a new follower relationship

GET /followers/
  - Get all follower relationships

GET /followers/{user_id_1}/followers/{user_id_2}
  - Get a specific follower relationship

DELETE /followers/{user_id_1}/followers/{user_id_2}
  - Delete a follower relationship
```

5.5 Playlist Management

```
POST /playlists/
  - Create a new playlist

GET /playlists/
  - Get all playlists

GET /playlists/{playlist_id}
  - Get a playlist by ID

PUT /playlists/{playlist_id}
  - Update a playlist

DELETE /playlists/{playlist_id}
  - Delete a playlist
```

5.6 Playlist-User Relationships

```
POST /playlist_users/
  - Create a new playlist-user relationship

GET /playlist_users/
  - Get all playlist-user relationships

GET /playlist_users/{user}/{fallows}
  - Get a specific playlist-user relationship

DELETE /playlist_users/{user}/{fallows}
  - Delete a playlist-user relationship
```

5.7 Playlist-Song Relationships

```
POST /playlist_songs/
  - Create a new playlist-song relationship

GET /playlist_songs/
  - Get all playlist-song relationships

GET /playlist_songs/{playlist_id}/songs/{song_id}
  - Get a specific playlist-song relationship

DELETE /playlist_songs/{playlist_id}/songs/{song_id}
  - Delete a playlist-song relationship
```

5.8 Likes Management

```
POST /likes/
  - Create a new like

GET /likes/
  - Get all likes

GET /likes/{user_id}/songs/{song_id}
  - Get a specific like

DELETE /likes/{user_id}/songs/{song_id}
  - Delete a like
```

5.9 Song Management

```
POST /songs/
  - Create a new song

GET /songs/
  - Get all songs

GET /songs/name/{song_name}
  - Get a song by name

GET /songs/{song_id}
  - Get a song by ID

PUT /songs/{song_id}
  - Update a song

DELETE /songs/{song_id}
  - Delete a song
```

5.10 Genre Management

```
POST /genres/
  - Create a new genre

GET /genres/
  - Get all genres

GET /genres/{song_id}
  - Get a genre by song ID

GET /genres/most-listened-last-month
  - Get the most listened genre in the last month

DELETE /genres/{song_id}
  - Delete a genre
```

5.11 Album Management

```
POST /albums/
  - Create a new album

GET /albums/
  - Get all albums

GET /albums/{album_id}
  - Get an album by ID

PUT /albums/{album_id}
  - Update an album

DELETE /albums/{album_id}
  - Delete an album

GET /albums/streaming-stats
  - Get total listening time for each album
```

5.12 Album-Song Relationships

```
POST /album_infos/
  - Create a new album-song relationship

GET /album_infos/
  - Get all album-song relationships

GET /album_infos/{album_id}/songs/{song_id}
  - Get a specific album-song relationship

DELETE /album_infos/{album_id}/songs/{song_id}
  - Delete an album-song relationship
```

5.13 Group Management

```
POST /groups/
  - Create a new group

GET /groups/
  - Get all groups

GET /groups/{group_id}
  - Get a group by ID

PUT /groups/{group_id}
  - Update a group

DELETE /groups/{group_id}
  - Delete a group
```

5.14 Album-Group Relationships

```
POST /album_groups/
  - Create a new album-group relationship

GET /album_groups/
  - Get all album-group relationships

GET /album_groups/{album_id}/groups/{group_id}
  - Get a specific album-group relationship

DELETE /album_groups/{album_id}/groups/{group_id}
  - Delete an album-group relationship
```

5.15 Artist Management

```
POST /artists/
  - Create a new artist

GET /artists/
  - Get all artists

GET /artists/{group_id}/artists/{full_name}
  - Get an artist by group ID and name

DELETE /artists/{group_id}/artists/{full_name}
  - Delete an artist
```

5.16 Listening History Management

```
POST /histories/
  - Create a new history record

GET /histories/
  - Get all history records

GET /histories/{user_id}/history/{start_time}
  - Get a specific history record

DELETE /histories/{user_id}/history/{start_time}
  - Delete a history record
```

6 Challenges and Solutions

During the development of the API, several technical challenges were encountered and successfully addressed. A significant challenge was using SQLite3 because during the database design phase, the schema was initially created with MySQL in mind, utilizing specific data types and functions such as `INTERVAL` and `DATETIME`. However, when transitioning to SQLite3 for implementation, compatibility issues arose, as SQLite3 does not natively support some of these features. This mismatch led to unexpected errors and debugging difficulties, Resolving these issues required significant effort, including researching SQLite3's supported features and adapting queries and data types to ensure compatibility and functionality across the database and API layers. Another challenge involved maintaining referential integrity across the complex network of relationships between entities such as users, playlists, songs, and albums. This was resolved through the careful implementation of foreign key constraints and cascading operations. The authentication system also presented challenges in terms of security and token management, which were addressed by implementing JWT-based authentication with proper token validation and user session handling. These challenges were systematically tackled through robust error handling, proper connection management, and careful consideration of data type conversions between the database and API layers.