

DMQL Project Phase - II

SPOTIFY MUSIC RECOMMENDATION

Name: **Debosmit Neogi, Shivan Mathur**

UBID: **debosmit, shivanma**

UB Person Number: **50548272, 50545084**

I. PROBLEM STATEMENT

In the ever growing domain of music recommendation systems, there is a huge necessity to build low latency, fast running and robust and efficient systems, that can handle big data and its concurrent addition, updation as well as deletion. This calls for a move beyond Excel-based data management towards database solutions. This project seeks to make use of the database management systems to build a recommendation system for Spotify songs. We are using the Spotify Tracks and Spotify Artists datasets. For this project we aim to build a recommendation system that captures the underlying patterns of users' preference of songs and artists and recommend new songs to a user.

II. BACKGROUND:

Music recommendation systems play a vital role in enhancing user engagement and satisfaction on platforms like Spotify. These systems rely on vast amounts of data, including user listening history, song attributes, and artist information, to generate relevant recommendations. Large product based companies like Spotify thrive on specifically making good use of users' data that are present in bulk and raw format. These are called "Big Data" and require proper domain knowledge to handle them and make good sense of them. These large MNCs specifically hire data engineers and data scientists to use the large amount of data available to develop their business model to cater well to users' choice and demands. For this the first step is to transform the data into a structured format that allows ready insertion, updating and deletion of new data concurrently. Hence, proper DBMS is crucial base on top which complex business focused models are based like the recommendation system, that we are discussing here.

III. DATASET USED

For this project, we have used Spotify Dataset from 1921-2020. This dataset consists of over 600,000 tracks. The dataset consists of mainly 2 parts in the form of CSV files: artists.csv and tracks.csv. artist.csv consists of details about artists like artist name, his/her genre, popularity and followers. Tracks.csv consists of data regarding the tracks like the artist of the track,

its genre, when it was released, acoustic and other features related to a track etc.

Apart from these data we have synthetically generated a new table: "User" that consists of user_id, username and user_email.

IV. WHY USE DBMS INSTEAD OF EXCEL

Microsoft Excel's limitations in handling relational data, managing real-time updates, and performing efficient queries hinder the development and deployment of robust recommendation algorithms. Databases, on the other hand, offer a structured and scalable solution for managing music data. By leveraging relational database management systems (RDBMS) or NoSQL databases, recommendation systems can streamline data storage, optimize query processing, and implement advanced algorithms for collaborative filtering, content-based filtering, and hybrid recommendation techniques.

Along that, thousands of new songs are uploaded daily in hundreds of different languages daily. So, we need a system that can handle adding and updating new data concurrently and also ensuring low latency. Considering the nature of the data and frequent ADD, UPDATE operations that needed to be performed, a proper Data Base Management System is needed, instead of excel based system.

V. POTENTIAL CONTRIBUTION:

This project holds significant potential for advancing the field of music recommendation systems. By showcasing the benefits of utilizing databases over Excel for managing Spotify music data, it can improve the performance, accuracy, and scalability of recommendation algorithms. Through efficient data organization, faster query processing, and seamless integration with recommendation models, databases enable more personalized and engaging music recommendations for users. The contribution is crucial as personalized music recommendations drive user retention, increase platform usage, and foster user satisfaction. By embracing databases, music streaming platforms can enhance the discovery experience, promote diverse artists and genres, and ultimately strengthen their competitive edge in the market. Additionally, insights gained from database-driven recommendation systems can inform strategies for content Creation, playlist generation,

and targeted marketing campaigns, thereby fostering a more vibrant and dynamic music ecosystem.

VI. TARGET USERS

Users: The database will be directly used by Spotify company's engineers, data scientists and data engineers. They will use the dataset for updating the database with new features or attributes, Add new data rows of customers and delete any redundant information. Spotify users' will also interact with the database indirectly when they search for any songs in Spotify or when a new song/track is recommended to them. Also, business executives may use it for pitching business objectives.

Database Administrator: Administrating a database is a really complex, challenging and sensitive task. Because a database is the most crucial component of a product. Any error in the database will render the product ineffective, thus resulting in huge monetary loss for the company. Hence, a dedicated team with back-end development experience and capable of handling operations in big data using database has to be hired for the task of database administration.

VII. SCOPE AND METHODOLOGY OF OUR WORK

The Scope of this project includes designing a good recommendation system that can handle large data and can act concurrently on numerous queries. By designing the basis of a recommendation system using a DBMS, organizations can effectively manage user data, item information, and interaction records to generate personalized and relevant recommendations.

1. Data Processing:

This step involves collecting publicly available data from Spotify, and evaluate and assess them properly. The datasets mainly consists of: Track metadata, artist information, user listening history, Track tempo, pitch (and other meta-data) and user preferences. Assess the 4 Vs in Data Intensive Computing: Volume, Veracity, Variety, and Velocity of the dataset to determine its suitability for our proposed problem definition: Spotify Music Recommendation System.

2. Analyzing best method to store and structure data

Analyze the limitations of managing Spotify music data using Excel spreadsheets, focusing on scalability, performance, and data organization. Perform SWOT analysis to Investigate the advantages of utilizing database systems, such as relational database management systems (RDBMS) or NoSQL databases, in terms of data storage, query optimization, and scalability; as well as their disadvantages with respect to cost of maintenance and how they fair against low maintenance, simple systems like Excel. A detailed analysis of DBMS vs Excel has been provided in the earlier sections.

Database Schema Design and Implementation:

Design a database schema tailored to store Spotify music data efficiently, considering factors such as normalization, indexing, and scalability. Implement the database schema using appropriate database management systems (e.g., MySQL, PostgreSQL, MongoDB) to facilitate seamless data storage and retrieval.

The Steps for Database design involves:

a. Determine the Needs: Recognize the objectives and purpose of the database. Stakeholder requirements should be gathered to ascertain what data must be stored and for what purpose.

b. Idea Generation: Develop a conceptual model by defining entities, properties, and relationships using methods such as Entity-Relationship Diagrams (ERD). This is a crucial step that will help us to further design the database schema for our recommendation system.

c. Data Normalization: Examine the conceptual model for inconsistencies and redundant data. To reduce duplication and enhance data integrity, use normalizing procedures (e.g., First Normal Form, Second Normal Form, Third Normal Form). Here, our aim is to normalize the database into BCNF form. In BCNF, is a more stricter version of 3NF.

d. Convert to Logical Architecture: Convert the conceptual model into a logical model, which is commonly shown as a relational database's tables. This additionally includes determining each table's properties and data types in accordance with the specifications listed in the conceptual design.

Designing the product using DBMS Designing the basis of a recommendation system using a database management system (DBMS) involves structuring the database schema to efficiently store and retrieve relevant data for generating recommendations. For designing the recommendation system we need to appropriately define relationship amongst different relations. Firstly we need to define appropriate relations by dividing them using appropriate attributes. There should be appropriate schemes for data modelling, data updating, data query etc. and all these should be done in real time.

Integration with Recommendation Algorithms: Explore various recommendation techniques, including collaborative filtering, content-based filtering, and hybrid methods, suitable for music recommendation. Integrate the database with recommendation algorithms to leverage structured data for generating personalized music recommendations.

VIII. NORMALIZATION OF RELATIONS TO BCNF

In this section, we discuss the different relations (tables) in our database. We have listed all the dependencies for each relations and we have checked if they are in Boyce-Codd Normal Form. If any relation is not in BCNF form,

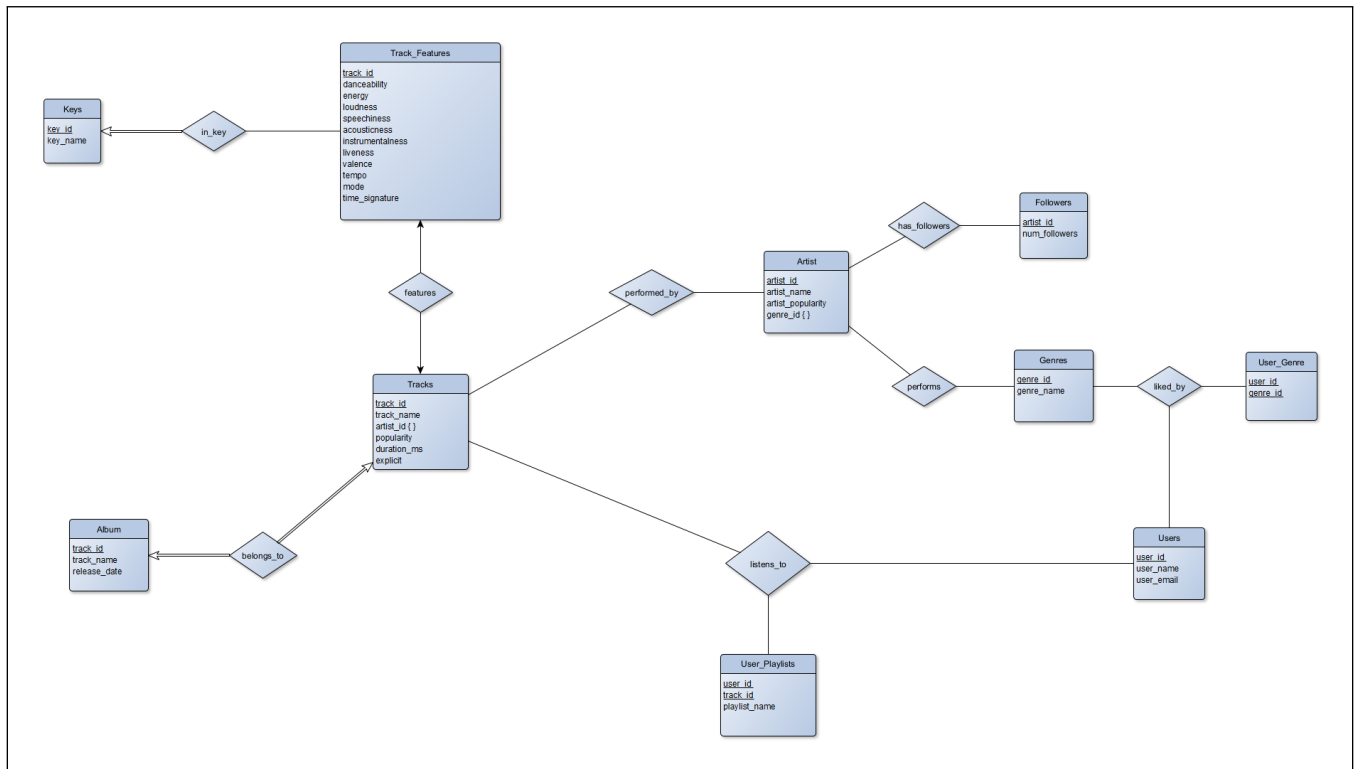


Fig. 1. ER Diagram Before Normalization

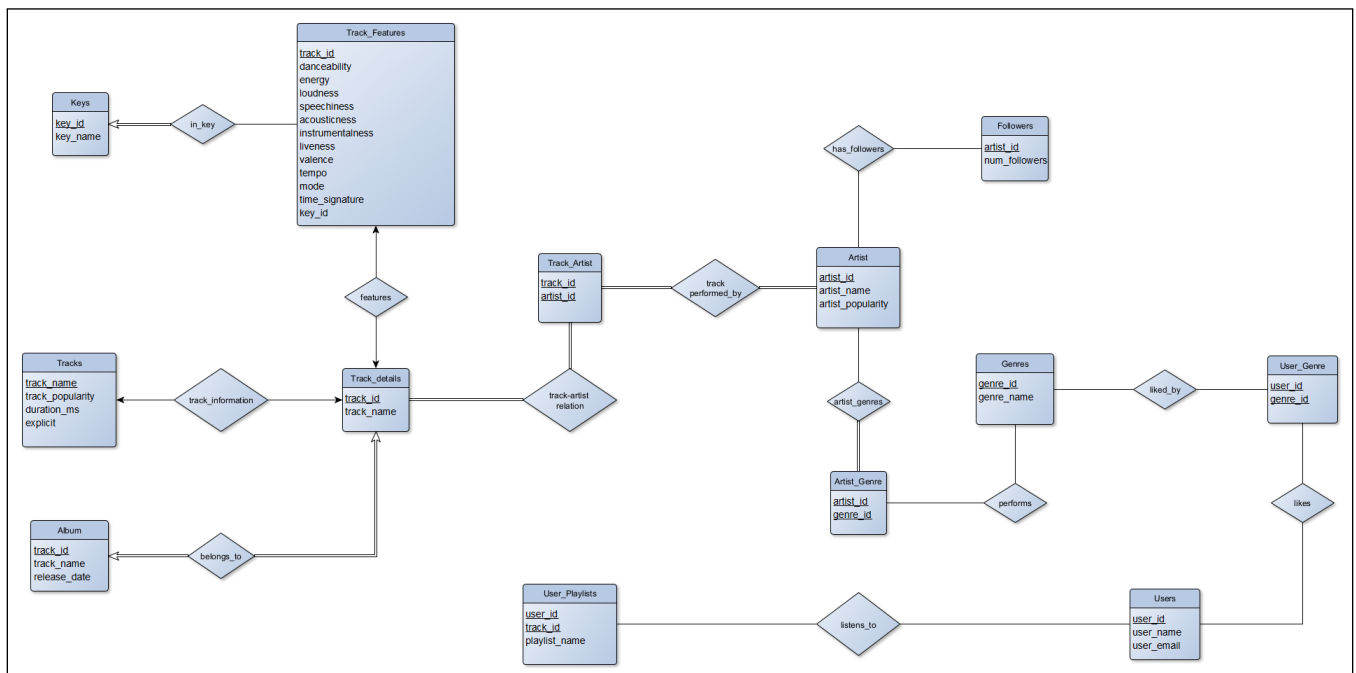


Fig. 2. ER Diagram After Normalization

we have normalized them and decomposed them into multiple relations. The condition to check if relation is in BCNF is: For all the functional dependencies, $\alpha \rightarrow \beta$, α should be the candidate key or super key.

Description of Each Attributes: track_id is the unique id assigned to each track (datatype = String). track_name is the name given to each track (datatype = String). track_popularity denotes how popular a track is in terms of ranking (datatype = Real). artist_id is the unique id given to artist (datatype = String). artist_name is the name of the artist (datatype = String). artist_popularity denotes how popular an artist is in terms of ranking (datatype = Real). duration_ms is the duration of a track in milliseconds (datatype = Real). explicit is a binary label provided to a track (datatype = integer). genre_id is the unique id given to a genre of a track (datatype = String). genre is the theme of a track (datatype = String). Key_name denote the musical key or note in which the track is sung (datatype = integer). Key_id is unique identity for key (datatype = integer). user_id is the unique id given to a user (datatype = integer). username is the name of the user (datatype = String). user_email is the email of the user (datatype = String). danceability, energy, loudness, speechiness, acousticness, instrumentalness, liveness, valence, tempo, (Datatype = Real), time_signature, mode (Datatype = Integer) together signifies the overall characteristics of a song in terms of how users enjoy it and its audio qualities in terms of pitch, acoustics etc. . followers is the number of followers an artist has (datatype = integer). release_date is the date on which a track is released (datatype = Datetime).

There is no initialization or default value for any attribute. And none of the attribute value can be set to NULL.

Relation 1: track_initial

Relation **track_initial** consists of following attributes: "track_id", "track_name", "track_popularity", "artist_id", "duration_ms" and "explicit".

All the functional dependencies for this relation are:
Functional dependencies:

$track_id \rightarrow track_name$

$track_id \rightarrow\rightarrow artist_id$

$track_name \rightarrow \{track_popularity, duration_ms, explicit\}$

track_id is the Candidate Key and also the only Prime Attribute.

Using the above mentioned condition for BCNF, the first functional dependency track_id is the candidate key. Hence, it is in BCNF.

In the second functional dependency, there is a Multi-Valued Dependency (MVD) between track_id and artist_id.

Also in the third Functional dependency, track_name is not candidate key/ super key.

Hence, the relation is not in BCNF.

To normalize the relation R1: track_initial, we have to decompose it into:

R11: (track_id, track_name),

R12: (track_id, artist_id) and

R13: (track_name, track_popularity, duration_ms, explicit).

After this decomposition, R11 was already normalized, but R13 now has track_name as Candidate Key (because it can uniquely determine other attributes in the relation) and hence it is in BCNF.

For R12, we can split the MVD into different relation to solve it.

Relation 2: track_details (R11)

Relation **track_details** consists of following attributes: "track_id", "track_name".

All the functional dependencies for this relation are:

Functional dependencies:

$track_id \rightarrow \{track_name\}$

Here the Candidate Key is track_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 3: track_artist (R12)

Relation **track_artist** consists of following attributes: "track_id", "artist_id".

All the functional dependencies for this relation are:

Functional dependencies:

$track_id \rightarrow \{artist_id\}$

Here the Candidate Key is the composite key consisting of: track_id and artist_id (because we need them both to uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 4: track (R13)

Relation **track** consists of following attributes: "track_id, track_name", "track_popularity", "duration_ms" and "explicit".

All the functional dependencies for this relation are:

Functional dependencies:

$track_name \rightarrow \{track_popularity, duration_ms, explicit\}$

Here the Candidate Key is track_name (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 5: artist_initial

Relation **artist_initial** consists of following attributes: "artist_id", "artist_name", "artist_popularity", "genre_id".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{artist_id} \rightarrow \{\text{artist_name}, \text{artist_popularity}\}$

$\text{artist_id} \rightarrow \rightarrow \{\text{genre_id}\}$

Here the Candidate Key is artist_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this functional dependency is satisfied.

However, second functional dependency has multi valued dependencies. So, we need to decompose it into R51: artist_id, artist_name, artist_popularity

R52: artist_id, genre_id.

Relation 6: artist (R51)

Relation **artist** consists of following attributes: "artist_id", "artist_name", "artist_popularity", "genre_id".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{artist_id} \rightarrow \{\text{artist_name}, \text{artist_popularity}\}$

Here the Candidate Key is artist_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, the relation is in BCNF.

Relation 7: artist_genre

Relation **artist_genre** consists of following attributes: "artist_id, genre_id".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{artist_id} \rightarrow \{\text{genre_id}\}$

Here the Candidate Key is actually a composite key consisting of artist_id , genre_id (because we need them both to uniquely determine other attributes in the relation) and the prime attributes are:artist_id , genre_id. By the condition for BCNF, this relation is in BCNF.

Relation 8: album

Relation **album** consists of following attributes: "track_id", "track_name", "release_date".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{track_id} \rightarrow \{\text{track_name}, \text{release_date}\}$

Here the Candidate Key is track_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 9: genre

Relation **genre** consists of following attributes: "genre_id", "genre_name".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{genre_id} \rightarrow \{\text{genre_name}\}$

Here the Candidate Key is genre_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 10: keys

Relation **keys** consists of following attributes: "key_id", "key_name".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{key_id} \rightarrow \{\text{key_name}\}$

Here the Candidate Key is key_id (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 11: user_genre

Relation **user_genre** consists of following attributes: "user_id, genre_id".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{user_id} \rightarrow \{\text{genre_id}\}$

Here the Candidate Key is actually a composite key consisting of user_id , genre_id (because we need both to uniquely determine other attributes in the relation) and the prime attributes are:user_id , genre_id. By the condition for BCNF, this relation is in BCNF.

Relation 12: user_playlist

Relation **album** consists of following attributes: "user_id", "track_id", "playlist_name".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{user_id} \rightarrow \{\text{track_id}\}$

$\text{track_id} \rightarrow \{\text{playlist_name}\}$

Here the Candidate Key is composite key consisting of: `user_id` , `track_id` (because we need both to uniquely determine other attributes in the relation) and by the condition for BCNF, both the functional dependencies satisfy the condition and hence, this relation is in BCNF.

Relation 13: followers

Relation **followers** consists of following attributes: "artist_id", "followers".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{artist_id} \rightarrow \{\text{followers}\}$

Here the Candidate Key is `artist_id` (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 14: users

Relation **users** consists of following attributes: "users_id", "username", "user_email".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{user_id} \rightarrow \{\text{username}, \text{user_email}\}$

Here the Candidate Key is `user_id` (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

Relation 15: features_track

Relation **features_track** consists of following attributes: "track_id", "danceability", "energy", "loudness", "speechiness", "acousticness", "instrumentalness", "liveness", "valence", "tempo", "mode", "key_id", "time_signature".

All the functional dependencies for this relation are:

Functional dependencies:

$\text{track_id} \rightarrow \{\text{danceability}, \text{energy}, \text{loudness}, \text{speechiness}, \text{acousticness}, \text{instrumentalness}, \text{liveness}, \text{valence}, \text{tempo}, \text{mode}, \text{key_id}, \text{time_signature}\}$

Here the Candidate Key is `track_id` (because it can uniquely determine other attributes in the relation) and by the condition for BCNF, this relation is in BCNF.

IX. RELATION BETWEEN DIFFERENT TABLES

`users` and `user_genre` are related by: Genres liked by users.
`users` and `user_playlist` are related by: Playlists followed by users.

`genre` and `user_genre` are related by: Genre followed by a user.
`genre` and `artist_genre` are related by: To relate artist to a genre.
`artist` and `artist_genre` are related by: Artist sings song of a

particular genre.

`artist` and `follower` are related by: Follower count of artists.

`artist` and `track_artist` are related by: Track sung by an artist.

`track_details` and `track_artist` are related by: To connect relationship between artist and track.

`track_details` and `track` are related by: Detailed information about tracks.

`track_details` and `album` are related by: Details of a track under an album.

`track_details` and `track_features` are related by: Connects acoustic and enjoyability features of a track to track details.

`track_features` and `keys` are related by: Maps keys used in a track to other features of track.

X. FOREIGN KEY

A. List of Foreign Keys

`album`: FOREIGN KEY (`track_id`) REFERENCES `track_details(track_id)`

`track_features`: FOREIGN KEY (`track_id`) REFERENCES `track_details(track_id)`

`followers`: FOREIGN KEY (`artist_id`) REFERENCES `artists(artist_id)`

`user_playlist`: FOREIGN KEY (`user_id`) REFERENCES `users(user_id)` AND

FOREIGN KEY (`track_id`) REFERENCES `track_details(track_id)`

B. Action taken on Foreign Key when Primary Key is deleted

If we try to delete the some from the database where a key is foreign key referencing primary key of another table, it should not allow the deletion of the data from the table due to the referential integrity constraints being enforced and it will throw an error. For instance, `album` table references `track_id` from `track_details` table. If there is any data for the `track_id` attribute in the `track_details` table and we try to delete that `track_id` from this table, it will not allow to delete. However, we can use `ON CASCADE DELETE` while deleting data from the `track_details` table, in order to maintain data integrity.

XI. DESIGNING SPOTIFY DATABASE USING POSTGRESQL

A. Creating Required Tables

We have created tables on PostgreSQL by following the ER diagrams and the way we designed our database schema.

```
26 -- CREATE TABLE: track_details
27 CREATE TABLE track_details (
28     track_id VARCHAR(255),
29     track_name VARCHAR(10000),
30     PRIMARY KEY (track_id)
31 );
32
```

Fig. 3. Table Creation: TRACK_DETAILS

```

33 -- CREATE TABLE: tracks
34 CREATE TABLE tracks (
35     track_name VARCHAR(10000),
36     track_popularity INTEGER,
37     duration INTEGER,
38     explicit INTEGER,
39     PRIMARY KEY (track_name)
40 );

```

Fig. 4. Table Creation: TRACKS

```

42 -- CREATE TABLE: artists
43 CREATE TABLE artists (
44     artist_id VARCHAR(255),
45     artist_name VARCHAR(500),
46     artist_popularity INTEGER,
47     PRIMARY KEY (artist_id)
48 );

```

Fig. 5. Table Creation: ARTISTS

```

50 -- CREATE TABLE: album
51 CREATE TABLE album (
52     track_id VARCHAR(255),
53     track_name VARCHAR(10000),
54     release_date DATE,
55     PRIMARY KEY (track_id),
56     FOREIGN KEY (track_id) REFERENCES track_details(track_id)
57     ON DELETE CASCADE
58 );

```

Fig. 6. Table Creation: ALBUM

```

60 -- CREATE TABLE: genre
61 CREATE TABLE genre (
62     genre_id VARCHAR(255),
63     genre_name VARCHAR(500),
64     PRIMARY KEY (genre_id)
65 );

```

Fig. 7. Table Creation: GENRE

```

67 -- CREATE TABLE: track_features
68 CREATE TABLE track_features (
69     track_id VARCHAR(255),
70     danceability REAL,
71     energy REAL,
72     loudness REAL,
73     mode INTEGER,
74     speechiness REAL,
75     acousticness REAL,
76     instrumentalness REAL,
77     liveness REAL,
78     valence REAL,
79     tempo REAL,
80     time_signature INTEGER,
81     key_id INTEGER,
82     PRIMARY KEY (track_id),
83     FOREIGN KEY (track_id) REFERENCES track_details(track_id)
84     ON DELETE CASCADE
85 );

```

Fig. 8. Table Creation: TRACK_FEATURES

```

87 -- CREATE TABLE: keys
88 CREATE TABLE keys (
89     key_id VARCHAR(255),
90     key_name VARCHAR(500),
91     PRIMARY KEY (key_id)
92 );

```

Fig. 9. Table Creation: KEYS

```

132 -- CREATE TABLE: artist_genre
133 CREATE TABLE artist_genre (
134     artist_id VARCHAR(255),
135     genre_id VARCHAR(255),
136     PRIMARY KEY (artist_id, genre_id),
137     FOREIGN KEY (artist_id) REFERENCES artists(artist_id),
138     FOREIGN KEY (genre_id) REFERENCES genre(genre_id)
139     ON DELETE CASCADE
140 );

```

Fig. 10. Table Creation: ARTIST_GENRE

```

94 -- CREATE TABLE: followers
95 CREATE TABLE followers (
96     artist_id VARCHAR(255),
97     followers REAL,
98     PRIMARY KEY (artist_id),
99     FOREIGN KEY (artist_id) REFERENCES artists(artist_id)
100     ON DELETE CASCADE
101 );

```

Fig. 11. Table Creation: FOLLOWERS

```

103 -- CREATE TABLE: users
104 CREATE TABLE users (
105     user_id VARCHAR(255),
106     user_name VARCHAR(255),
107     user_email VARCHAR(255),
108     PRIMARY KEY (user_id)
109 );

```

Fig. 12. Table Creation: USERS

```

111 -- CREATE TABLE: user_playlist
112 CREATE TABLE user_playlist (
113     user_id VARCHAR(255),
114     track_id VARCHAR(255),
115     playlist_name VARCHAR(255),
116     PRIMARY KEY (user_id, track_id),
117     FOREIGN KEY (user_id) REFERENCES users(user_id),
118     FOREIGN KEY (track_id) REFERENCES track_details(track_id)
119     ON DELETE CASCADE
120 );

```

Fig. 13. Table Creation: USER_PLAYLIST

```

122 -- CREATE TABLE: track_artist
123 CREATE TABLE track_artist (
124     track_id VARCHAR(255),
125     artist_id VARCHAR(255),
126     PRIMARY KEY (track_id, artist_id),
127     FOREIGN KEY (track_id) REFERENCES track_details(track_id),
128     FOREIGN KEY (artist_id) REFERENCES artists(artist_id)
129     ON DELETE CASCADE
130 );

```

Fig. 14. Table Creation: TRACK_ARTIST

we imported those files and loaded them into the tables. We imported the csv files as shown in the below figures.

```

142 -----IMPORTING DATA-----
143 -----IMPORTING DATA-----
144 -----IMPORTING DATA-----
145 -----IMPORTING DATA-----
146 -- Insert data from track_details csv, which is created from the original dataset
147 COPY track_details (track_id, track_name)
148 FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/track_details.csv' DELIMITER ',' CSV HEADER;
149 -----IMPORTING DATA-----
150 -----IMPORTING DATA-----
151 -- Insert data from track csv, which is created from the original dataset
152 COPY tracks (track_name, track_popularity, duration, explicit)
153 FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/track.csv' DELIMITER ',' CSV HEADER;
154 -----IMPORTING DATA-----
155 -----IMPORTING DATA-----
156 -- Insert data from artist csv, which is created from the original dataset
157 COPY artists (artist_id, artist_name, artist_popularity)
158 FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/artist.csv' DELIMITER ',' CSV HEADER;
159 -----IMPORTING DATA-----
160 -----IMPORTING DATA-----
161 -- Insert data from album csv, which is created from the original dataset
162 COPY album (track_id, track_name, release_date)
163 FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/album.csv' DELIMITER ',' CSV HEADER;
164 -----IMPORTING DATA-----
165 -----IMPORTING DATA-----
166 -- Insert data from genre csv, which is created from the original dataset
167 COPY genre (genre_id, genre_name)
168 FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/genre.csv' DELIMITER ',' CSV HEADER;
169 -----IMPORTING DATA-----

```

Fig. 15. Data Import into tables 1

B. Importing Data into Tables

We created a python script to generate a separate csv file for each table based on the original dataset. Using PostgreSQL,

```

171 -- Insert data from features csv, which is created from the original dataset
172 COPY track_features (track_id, danceability, energy, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo,
173 time_signature, key_id)
174 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/features.csv' DELIMITER ',' CSV HEADER
175
176
177 -- Insert data from key csv, which is created from the original dataset
178 COPY keys (key_id, key_name)
179 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/key.csv' DELIMITER ',' CSV HEADER
180
181
182 -- Insert data from follower csv, which is created from the original dataset
183 COPY followers (artist_id, followers)
184 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/follower.csv' DELIMITER ',' CSV HEADER
185
186
187 -- Insert data from user csv, which is created from the original dataset
188 COPY users (user_id, user_name, user_email)
189 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/user.csv' DELIMITER ',' CSV HEADER
190
191
192 -- Insert data from user_playlist csv, which is created from the original dataset
193 COPY user_playlist (user_id, playlist_name, track_id)
194 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/user_playlist.csv' DELIMITER ',' CSV HEADER
195
196
197 -- Insert data from track_artist csv, which is created from the original dataset
198 COPY track_artist (track_id, artist_id)
199 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/track_artist.csv' DELIMITER ',' CSV HEADER
200
201
202 -- Insert data from artist_genre csv, which is created from the original dataset
203 COPY artist_genre (artist_id, genre_id)
204 FROM 'D:/JB/Spring 2024/CSE 568 - Data Models and Query Language/Project/artist_genre.csv' DELIMITER ',' CSV HEADER
205

```

Fig. 16. Data Import into tables 2

XII. ISSUES WORKING WITH LARGE DATABASE

The major problem with working on a large database is query performance. We need to ensure the query executes efficiently by improving and optimizing query execution time and query execution plan. While testing the database, we tried to fetch records from the album table using filter.

```

84
85 SELECT * FROM album WHERE track_name = 'Love Yourself';
86
87

```

Data Output Messages Notifications

Successfully run. Total query runtime: 211 msec.
21 rows affected.

Fig. 17. Select query with filter before indexing

We then utilized the important concept in database system, which is indexing to improve the query performance. We created an index on track_id and track_name on album table and then tested the query execution. We observed significant amount of reduction in query execution time after applying indexing.

```

84
85 SELECT * FROM album WHERE track_name = 'Love Yourself';
86
87 CREATE INDEX idx_album ON album(track_id, track_name);
88

```

Data Output Messages Explain × Notifications

Successfully run. Total query runtime: 165 msec.
21 rows affected.

Fig. 18. Select query after indexing

XIII. TEST THE SPOTIFY DATABASE

We tested our database by trying to insert, update and delete the data into some tables and executed some business queries to check the results of our application.

A. Data Insertion

In the figure 17, we tried to add two new users into the database so we executed the INSERT query to add those entries and it ran successfully and we were able to view the new records.

```

19
20
21
22
23
24
25
26
27
28
29
30
31
32

```

-- INSERT DATA INTO TABLES --

```

23 INSERT INTO users (user_id, user_name, user_email) VALUES ('18881', 'ShivanMethur', 'shivanm@buffalo.edu');
24 INSERT INTO users (user_id, user_name, user_email) VALUES ('18882', 'DebosmitNeogi', 'debosmit@buffalo.edu');
25

```

Data Output Messages Notifications

INSERT 6 1

Query returned successfully in 151 msec.

Fig. 19. Data Insertion 1

Next, figure 18, we tried to add a new artist to the database. So, we created INSERT queries for all the relevant tables where the artist data needs to be added.

```

25
26
27
28
29
30
31
32

```

-- 2. Inserting new artist record into all the relevant tables --

```

26 INSERT INTO artists (artist_id, artist_name, artist_popularity) VALUES ('01HWY4F4SMKCEDF1GQB8BQCY', 'Utkarsh Methur', '18');
27 INSERT INTO artist_genre (artist_id, genre_id) VALUES ('01HWY4F4SMKCEDF1GQB8BQCY', '1488');
28 INSERT INTO artist_genre (artist_id, genre_id) VALUES ('01HWY4F4SMKCEDF1GQB8BQCY', '2331');
29 INSERT INTO followers (artist_id, followers) VALUES ('01HWY4F4SMKCEDF1GQB8BQCY', 50);
30 INSERT INTO track_artist (track_id, artist_id) VALUES ('1h8qIqW02nbV4Q7yszRhe', '01HWY4F4SMKCEDF1GQB8BQCY');
31

```

Data Output Messages Notifications

INSERT 6 1

Query returned successfully in 118 msec.

Fig. 20. Data Insertion 2

B. Data Update

While continuing to test our database, we tried to update some records in the database. As we can see in the figure 19, we tried to update the newly added artist's genre in the database. So we used the UPDATE command and wrote query to perform the task.

```

33
34
35
36
37
38
39
40
41
42
43

```

-- UPDATE DATA --

```

37 -- 1. Update specific artist's genre in the table:
38 UPDATE artist_genre SET genre_id = '76'
39 WHERE artist_id = '01HWY4F4SMKCEDF1GQB8BQCY'
40 AND genre_id = '1488';
41

```

Data Output Messages Notifications

UPDATE 1

Query returned successfully in 117 msec.

Fig. 21. Data Update 1

We also tried to update a track's features in the track_features table.


```

43 -- 2. Update the track's feature in the track_features table:
44 UPDATE track_features
45 SET instrumentalness = 0.022, loudness = -7, energy = 0.8, tempo = 96
46 WHERE track_id = '000CSYu4rvd8cQ7j1fxhZ';
47
Data Output Messages Notifications
UPDATE 1
Query returned successfully in 51 msec.

```

Fig. 22. Data Update 2

C. Data Deletion

Next, we try to delete certain data from the database. We tried to delete the new user record, which we added as part of testing using the DELETE command.

```

46 ----- DELETE QUERIES -----
47
48 -- 1. Delete user's data
49 DELETE FROM users
50 WHERE user_id = '100002';
51
Data Output Messages Notifications
DELETE 1
Query returned successfully in 120 msec.

```

Fig. 23. Data Deletion 1

We also tried to delete the newly added artist's record from the track_artist table, to remove artist's association from the track.

```

54 -- 2. Delete artist's data from the track_artist table:
55 DELETE FROM track_artist
56 WHERE artist_id = '01HWY4F4SMKCEEDF1GDQB0BQCY';
57
58 COMMIT;
59
60
Data Output Messages Notifications
DELETE 1
Query returned successfully in 119 msec.

```

Fig. 24. Data Deletion 2

NOTE: Since INSERT, UPDATE, and DELETE are data manipulation language (DML) commands, we need to make sure we COMMIT the changes to permanently save them.

D. SQL Queries

Further, we analyzed our database by creating and running variety of queries, which can potentially be used by end users to fetch some important results.

1. Query to find the Top 10 Most Popular Tracks:

```

84 ----- 10 SQL QUERIES -----
85
86 -- 1. Find the Top 10 Most Popular Tracks:
87 SELECT track_name, track_popularity
88 FROM tracks
89 ORDER BY track_popularity DESC
90 LIMIT 10;
91
Data Output Messages Notifications
track_name track_popularity
1 Peaches (feat. Daniel Caesar & Giveon) 100
2 drivers license 99
3 Astronaut In The Ocean 98
4 telepatía 97
5 Save Your Tears 96
6 Leave The Door Open 96
7 Blinding Lights 96
8 The Business 95
9 Heartbreak Anniversary 94
10 WITHOUT YOU 94

```

Fig. 25. SQL SELECT Query 1

2. Query to find Top 10 Artists in the database based on the number of tracks they have sung:

```

95 -- 2. Top 10 Artists:
96 SELECT a.artist_name, COUNT(*) AS track_count
97 FROM track_artist ta
98 JOIN artists a ON ta.artist_id = a.artist_id
99 GROUP BY a.artist_name
100 ORDER BY 2 DESC, track_count DESC
101 LIMIT 10;
102
103
Data Output Messages Notifications
artist_name track_count
1 Geraldo Azevedo 10
2 Vital Farias 10
3 Brownie McGhee 9
4 Nicola Zaccaria 5
5 Renato Ercolani 5
6 Beaky 5
7 Dozy 5
8 Ole 4
9 Piero de Palma 4
10 Elisabeth Schwarzkopf 4

```

Fig. 26. SQL SELECT Query 2

3. Query to find Most Popular Genre:

```

104 -- 3. Find Most Popular Genre
105 SELECT g.genre_name, COUNT(DISTINCT td.track_id) AS total_tracks
106 FROM genre g
107 JOIN artist_genre ag ON g.genre_id = ag.genre_id
108 JOIN track_artist ta ON ag.artist_id = ta.artist_id
109 JOIN track_details td ON ta.track_id = td.track_id
110 GROUP BY g.genre_name
111 ORDER BY total_tracks DESC
112 LIMIT 10;
113
Data Output Messages Notifications
genre_name total_tracks
1 marathi pop 12
2 protorap 10
3 edinburgh indie 10
4 grunge brasileiro 10
5 azeri traditional 10
6 "death 'n' roll" 9
7 ambient folk 9
8 experimental hip hop 9
9 kansas city hip hop 9
10 puerto rican pop 9

```

Fig. 27. SQL SELECT Query 3

4. Query to find top Genres based on the popularity:

```

115 -- 4. Top Genres by Popularity:
116 SELECT g.genre_name, COUNT(DISTINCT a.artist_id) AS num_artists
117 FROM genre g
118 JOIN artist_genre ag ON g.genre_id = ag.genre_id
119 JOIN artists a ON ag.artist_id = a.artist_id
120 GROUP BY g.genre_name
121 ORDER BY num_artists DESC
122 LIMIT 10;
123
124

```

	genre_name character varying (500)	num_artists bigint
1	rawstyle	572
2	japanese dance pop	568
3	scandipop	564
4	post-disco soul	559
5	shimmer pop	501
6	brazilian ccm	498
7	pakistani folk	485
8	italian post-hardcore	484
9	electric bass	467
10	assyrian pop	461

Fig. 28. SQL SELECT Query 4

5. Query to find Long and Energetic Tracks where the track duration is greater than 25000 and energy is greater than 0.5:

```

125 -- 5. Long and Energetic Tracks:
126 SELECT td.track_name, MAX(a.artist_name) AS artist_name, MAX(t.duration) AS duration, MAX(f.energy) AS energy
127 FROM track_details td
128 JOIN tracks t ON t.track_name = td.track_name
129 JOIN track_artist ta ON td.track_id = ta.track_id
130 JOIN artists a ON ta.artist_id = a.artist_id
131 JOIN track_features f ON td.track_id = f.track_id
132 WHERE t.duration > 25000 AND f.energy > 0.5
133 GROUP BY td.track_name
134 ORDER BY MAX(f.energy) DESC;
135

```

	track_name character varying (10000)	artist_name text	duration integer	energy real
1	Fifty Mega Mix	Red Bull	766013	0.880
2	Playground Mix	Ruben Dam	1260374	0.774
3	Loyal	Bic Runga	279707	0.725
4	Ketika Cinta Hadir Di Sini	Prilly Priscilla	273160	0.641
5	Caravan of Love	Jasper	344200	0.634
6	Insatiable Woman	Jasper	312107	0.571
7	Anything For You (Remix)	Terror Fabulo...	269773	0.519

Fig. 29. SQL SELECT Query 5

6. Query to find artists who have frequently collaborated:

```

138 -- 6. Frequent Artist Collaborators:
139 SELECT a1.artist_name AS artist1, a2.artist_name AS artist2, COUNT(*) AS collaborations
140 FROM track_artist ta1
141 JOIN track_artist ta2 ON ta1.track_id = ta2.track_id
142 JOIN artists a1 ON ta1.artist_id = a1.artist_id
143 JOIN artists a2 ON ta2.artist_id = a2.artist_id
144 WHERE a1.artist_id <> a2.artist_id
145 GROUP BY a1.artist_name, a2.artist_name
146 HAVING COUNT(*) > 3
147 ORDER BY collaborations DESC;
148

```

	artist1 character varying (500)	artist2 character varying (500)	collaborations bigint
1	Vital Farias	Geraldo Azevedo	10
2	Geraldo Azevedo	Vital Farias	10
3	Dozy	Beaky	5
4	Beaky	Dozy	5
5	Elisabeth Schwarzkopf	Nicola Zaccaria	4
6	Nicola Zaccaria	Elisabeth Schwarzkopf	4
7	Renato Ercolani	Piero de Palma	4
8	Piero de Palma	Renato Ercolani	4

Fig. 30. SQL SELECT Query 6

7. Query to find the number of tracks in each genre:

```

150 -- 7. Count of Tracks in each Genre
151 SELECT g.genre_name, COUNT(*) AS track_count
152 FROM artist_genre ag
153 JOIN genre g ON ag.genre_id = g.genre_id
154 GROUP BY g.genre_name
155 ORDER BY 2 DESC
156 LIMIT 25;
157

```

	genre_name character varying (500)	track_count bigint
1	rawstyle	572
2	japanese dance pop	568
3	scandipop	564
4	post-disco soul	559
5	shimmer pop	501
6	brazilian ccm	498
7	pakistani folk	485
8	italian post-hardcore	484
9	electric bass	467
10	assyrian pop	461
11	trap cristiano	460
12	pinoy alternative rap	458
13	didgeridoo	453
14	protest folk	446
15	musica caririense	444
16	indie quebecois	441
17	psychobilly	441
18	viola	431
19	alabama indie	426
20	indonesian emo rap	420

Total rows: 25 of 25 Query complete 00:00:00.142

Fig. 31. SQL SELECT Query 7

8. Query to find the average features of tracks:

```

159 -- 8. Average Features of a Track
160 SELECT AVG(tf.loudness) AS average_loudness, AVG(tf.energy) AS average_energy, AVG(tf.speechiness) AS average_speechiness,
161 AVG(tf.acousticness) AS average_acousticness, AVG(tf.instrumentalness) AS average_instrumentalness
162 FROM track_details td
163 JOIN track_features tf ON td.track_id = tf.track_id;
164
165

```

	average_loudness double precision	average_energy double precision	average_speechiness double precision	average_acousticness double precision	average_instrumentalness double precision
1	-10.20606809001115	0.5420359919512312	0.10486354186641378	0.4498627244137007	0.11345078219471409

Fig. 32. SQL SELECT Query 8

9. Query to display Track Names and their respective Artists' names:

	track_name character varying (10000)	artist_name character varying (500)
1	10,000 Years	Sticks
2	A Man is Nothing But a Fool	Brownie McGhee
3	Abertura (Desafio do Auto da Catingueira/Repente/Noven...	Geraldo Azevedo
4	Abertura (Desafio do Auto da Catingueira/Repente/Noven...	Vital Farias
5	Adiexodo	Dionysis Savvopoulos
6	Alibaba a štyridsať lúpežníkov	M.Badinková
7	Alibaba a štyridsať lúpežníkov	Z.Ďurindová
8	Anything For You (Remix)	Beenie Man
9	Anything For You (Remix)	Buju Banton
10	Anything For You (Remix)	Terror Fabulous

10. Query to display the Artist name and Genre Name based on the Popularity of Artist in a Genre:

Fig. 34. SQL SELECT Query 10

A. Problematic Query - 1

users table, the user's data from the user_playlist must be deleted before deleting the users record, otherwise the system will not allow to delete the user's record. It will throw an error, violating foreign key constraints, since user_playlist table reference user_id from users table as its foreign key.

```

100 .....
101 ..... TESTING TRIGGERS .....
102 .....
103 .....
104 SELECT * FROM users WHERE user_id = "787330";
105 .....
106 SELECT * FROM user_playlog WHERE user_id = "787330";
107 .....

```

Data Output Messages Explain Notifications

	user_id	IPv4_address	user_agent
1	787330	194.168.100.100	character-reading (CIS)

TESTING TRIGGERS

```

189
190
191
192
193 SELECT * FROM users WHERE user_id = :user_id;
194
195

```

Data Output Messages Explain Notifications

	PK1 character varying (255)	PK2 character varying (255)	PK3 character varying (255)	PK1_name character varying (255)
1	56730	3Y0K0V94PMQAD00090x48	Playst_1	
2	56730	500C9C94M000000000000	Playst_1	
3	56730	40C0C4J0M0C0000000000	Playst_1	
4	56730	20B0C0000000000000000	Playst_1	
5	56730	50P0C4000000000000000	Playst_1	
6	56730	01C00C000000000000000	Playst_1	
7	56730	60C00C000000000000000	Playst_1	
8	56730	50C000000000000000000	Playst_1	
9	56730	70x4M00000000000000000	Playst_1	
10	56730	3AGu000000000000000000	Playst_1	
11	56730	20P0000000000000000000	Playst_1	
12	56730	6000000000000000000000	Playst_1	
13	56730	40P2000000000000000000	Playst_1	
14	56730	30W1700000000000000000	Playst_1	
Total rows: 500 of 500				
	Query completed: 00:00:00.201			Rowset 1

```

104 SELECT * FROM users WHERE user_id = '107870';
105 SELECT * FROM user_playlist WHERE user_id = '107870';
106
107 DELETE FROM users WHERE user_id = '107870';
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

So, we have implemented this functionality by creating a trigger which will be invoked when we try to delete a user's record. Before deleting the user's record, the trigger will call the function to delete the records in `user_playlist` table associated with the user in action.

```

211 -----CREATING FUNCTIONS AND TRIGGERS-----
212
213
214
215 CREATE OR REPLACE FUNCTION delete_user_playlist()
216 RETURNS TRIGGER AS $$
217 BEGIN
218     DELETE FROM user_playlist
219     WHERE user_id = OLD.user_id;
220     RETURN OLD;
221 END;
222 $$ LANGUAGE plpgsql;
223
224
225 CREATE OR REPLACE TRIGGER user_playlist_deletion_trigger
226 BEFORE DELETE ON
227 users
228 FOR EACH ROW
229 EXECUTE FUNCTION delete_user_playlist();
230

```

Data Output Messages Notifications

CREATE TRIGGER

Query returned successfully in 36 msec.

Fig. 38. Function and Trigger Creation

```

193 SELECT * FROM users WHERE user_id = '36730';
194 SELECT * FROM user_playlist WHERE user_id = '36730';
195
196 DELETE FROM users WHERE user_id = '36730';
197
198
199

```

Data Output Messages Explain × Notifications

DELETE 1

Query returned successfully in 43 msec.

Fig. 39. User record deleted successfully

Verifying user record deleted successfully:

```

193 SELECT * FROM users WHERE user_id = '36730';
194 SELECT * FROM user_playlist WHERE user_id = '36730';
195
196 DELETE FROM users WHERE user_id = '36730';
197
198
199

```

Data Output Messages Explain × Notifications

user_id	user_name	user_email
[PK] character varying (255)	character varying (255)	character varying (255)

Fig. 40. User record in users' table before deletion

```

193 SELECT * FROM users WHERE user_id = '36730';
194 SELECT * FROM user_playlist WHERE user_id = '36730';
195
196 DELETE FROM users WHERE user_id = '36730';
197
198
199

```

Data Output Messages Explain × Notifications

user_id	track_id	playlist_name
[PK] character varying (255)	[PK] character varying (255)	character varying (255)

Fig. 41. User record in user_playlist table before deletion

B. Problematic Query - 2

Working on large database, pose challenges in terms of query performance and optimization. The issue we observed working on larger database was slower query performance. When we tried to utilize joins and/or filtering, it took a significant amount of time for the query execution. We executed the query in figure 39 and noted the query execution time and checked the query execution plan using EXPLAIN.

```

214 -----INDEXING-----
215
216
217
218 SELECT g.genre_name, COUNT(*) AS track_count
219 FROM artist_genre ag
220 JOIN genre g ON ag.genre_id = g.genre_id
221 GROUP BY g.genre_name
222 ORDER BY 2 DESC
223 LIMIT 25;
224

```

Data Output Messages Explain × Notifications

genre_name	track_count
character varying (500)	bigint
1 rawstyle	572
2 japanese dance pop	568
3 scandipop	564
4 post-disco soul	559
5 shimmer pop	501
6 brazilian com	498
7 pakistani folk	485

Total rows: 25 of 25 Query complete 00:00:00.125

Fig. 42. Query Execution time before indexing

```

217 EXPLAIN SELECT g.genre_name, COUNT(*) AS track_count
218 FROM artist_genre ag
219 JOIN genre g ON ag.genre_id = g.genre_id
220 GROUP BY g.genre_name
221 ORDER BY 2 DESC
222 LIMIT 25;

```

Data Output Messages Notifications

QUERY PLAN

Sort

1 Limit (cost=9504.36..9504.42 rows=25 width=23)

2 Sort (cost=9504.36..9518.34 rows=5993 width=23)

3 Sort Key (count(*)) DESC

4 HashAggregate (cost=4290.60..4346.53 rows=5993 width=23)

5 Group Key: g.genre_name

6 Gather (cost=8060.14..8234.67 rows=11186 width=23)

7 Workers Planned: 2

8 Parallel HashAggregate (cost=7060.14..7116.07 rows=5993 width=23)

9 Group Key: g.genre_name

10 Hash Join (cost=161.87..4693.72 rows=195284 width=19)

11 Hash Cond: (ag.genre_id = (g.genre_id))

12 Parallel Seq Scan on artist_genre ag (cost=0.00..5409.84 rows=195284 width=19)

13 Hash (cost=91.04..91.04 rows=5594 width=19)

14 Seq Scan on genre g (cost=0.00..91.04 rows=5594 width=19)

Fig. 43. Query execution statistics

```

217 EXPLAIN SELECT g.genre_name, COUNT(*) AS track_count
218 FROM artist_genre ag
219 JOIN genre g ON ag.genre_id = g.genre_id
220 GROUP BY g.genre_name
221 ORDER BY 2 DESC
222 LIMIT 25;

```

Data Output Messages Notifications

QUERY PLAN

Sort

1 Limit (cost=9504.36..9504.42 rows=25 width=23)

2 Sort (cost=9504.36..9518.34 rows=5993 width=23)

3 Sort Key (count(*)) DESC

4 HashAggregate (cost=4290.60..4346.53 rows=5993 width=23)

5 Group Key: g.genre_name

6 Gather (cost=8060.14..8234.67 rows=11186 width=23)

7 Workers Planned: 2

8 Parallel HashAggregate (cost=7060.14..7116.07 rows=5993 width=23)

9 Group Key: g.genre_name

10 Hash Join (cost=161.87..4693.72 rows=195284 width=19)

11 Hash Cond: (ag.genre_id = (g.genre_id))

12 Parallel Seq Scan on artist_genre ag (cost=0.00..5409.84 rows=195284 width=19)

13 Hash (cost=91.04..91.04 rows=5594 width=19)

14 Seq Scan on genre g (cost=0.00..91.04 rows=5594 width=19)

Fig. 44. Query Execution Plan

After analyzing, we implemented Indexing in genre and artist_genre tables on (genre_id) and (artist_id, genre_id) attributes respectively and then executed the query again. We observed slight improvement and reduction in the execution time.

C. Problematic Query - 3

Another problematic query we observed where we created an complex query with common table expression to first fetch maximum popularity of artists within each genre, and

used joins with relevant tables to associated artists with their respective genres.



Fig. 45. Problematic Query 3 Execution

This query was taking longer time, approximately 4 seconds to execute. We utilized the EXPLAIN tool to analyze the query execution plan and identify potential bottlenecks and inefficiencies in the query execution.

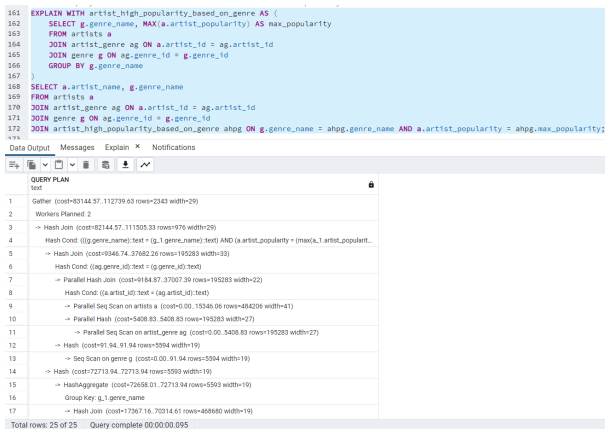


Fig. 46. Problematic Query 3 Analysis using EXPLAIN

The hash joins could be the potential bottleneck in the slower query performance, which is increasing the cost. Also, we could observe high cost while performing sequential scans on the tables which is time-consuming and resource-intensive. Hash Joins are used extensively in the query. To improve the hash join performance, we can use indexing. We also need to filter the records first before performing joins, reducing the intermediate result size.

Below is the result after optimizing the query.



Fig. 47. Problematic Query 3 Optimized Execution

D. Problematic Query - 4

Another problematic query we noted is where we try to find the to genres based on the popularity, where we used aggregate functions, joins, group by and order by clause. The query was approximately taking 5 seconds to execute. We analysed the query using EXPLAIN to identify potential bottlenecks and inefficiencies in the query execution.



Fig. 48. Problematic Query 4 Execution

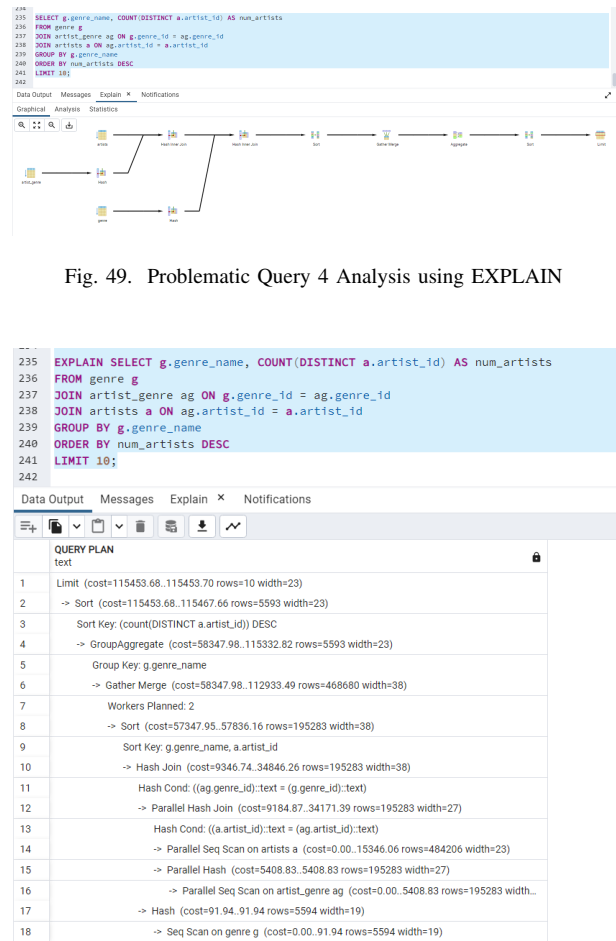


Fig. 49. Problematic Query 4 Analysis using EXPLAIN

Fig. 50. Problematic Query 4 Analysis using EXPLAIN

Sorting and aggregate function could be the potential bottleneck in the slower query performance, as cost for both the operation is high. Also, we could observe high cost while performing hash joins and sequential scans on the tables which is time-consuming and resource-intensive task. To improve the query performance, we need to ensure we use relevant columns to perform indexing. We can also modify

the query by repositioning group by and aggregate operations closer such that it reduces the amount of data to be scanned while performing join operations.

XV. DEPLOYING APPLICATION ON WEBSITE

We have created a website using Streamlit to display the query results on the user interface connecting PostgreSQL database at the back-end, where the user has been given several options to play with.

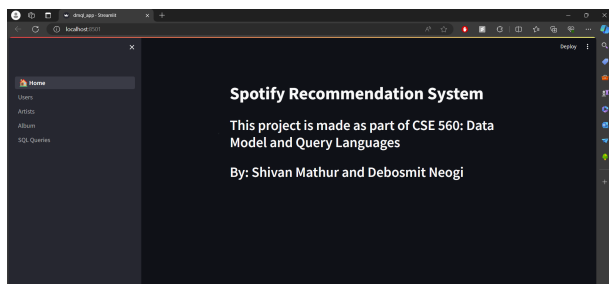


Fig. 51. Home Page

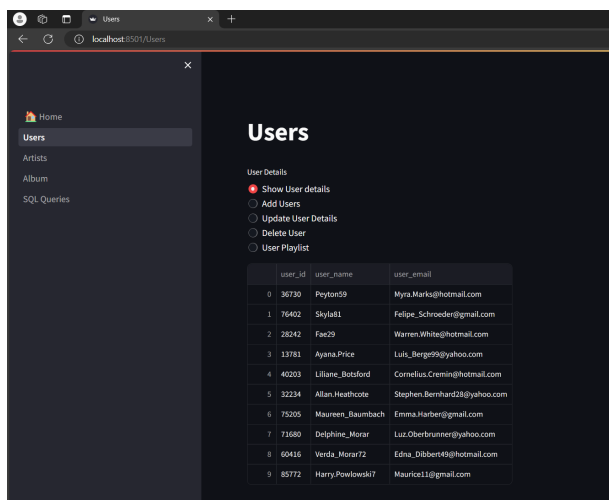


Fig. 52. User Page: Show User Details

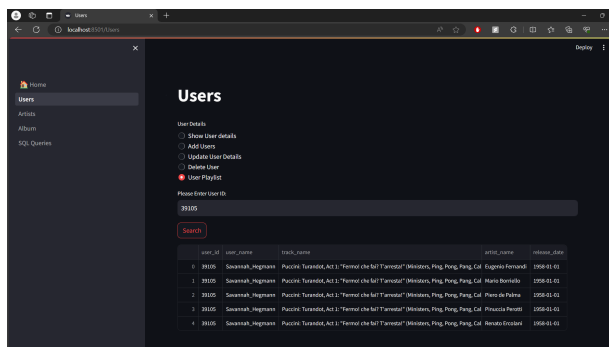


Fig. 53. User Page: Show User Playlist

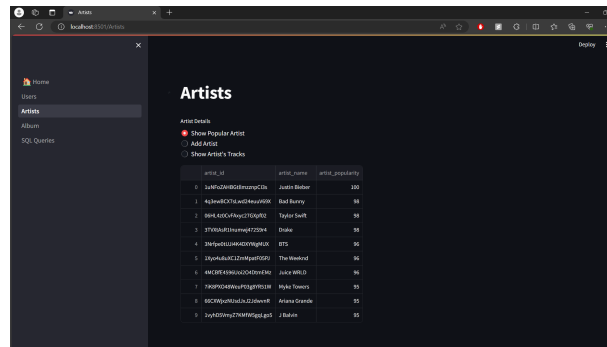


Fig. 54. Artists Page: Show Popular Artist

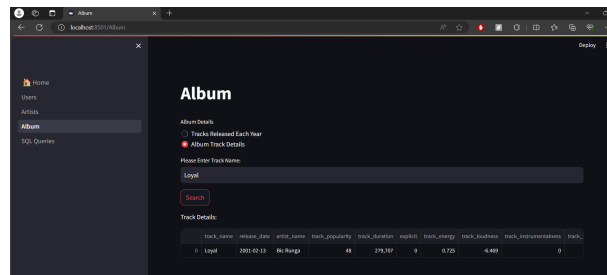


Fig. 55. Album Page: Show Album Track Details

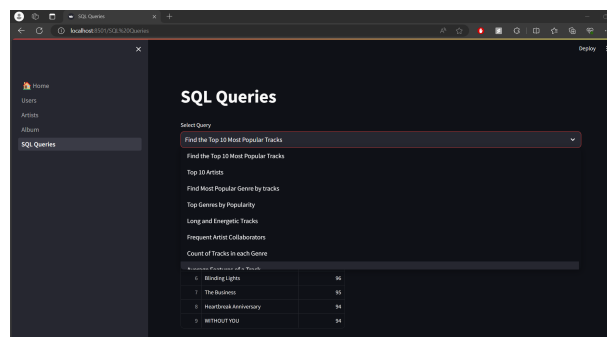


Fig. 56. SQL Queries: Drop-down option to display the required results

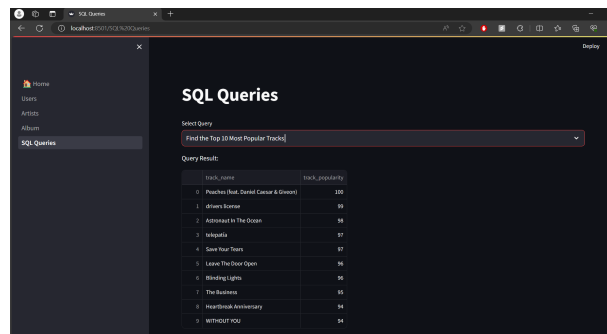


Fig. 57. SQL Queries: Display Top 10 Most Popular Tracks

XVI. CONTRIBUTION OF TEAM MEMBERS

We divided our tasks equally like splitting the sections of the report, creating database creation, SQL queries, and deploying

the database queries by building the user interface and the different page functionalities.

XVII. REFERENCE

[1.] <https://www.kaggle.com/datasets/yamaerenay/spotify-dataset-19212020-600k-tracks>

[2.] <https://www.rndgen.com/data-generator>