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 monetory 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,

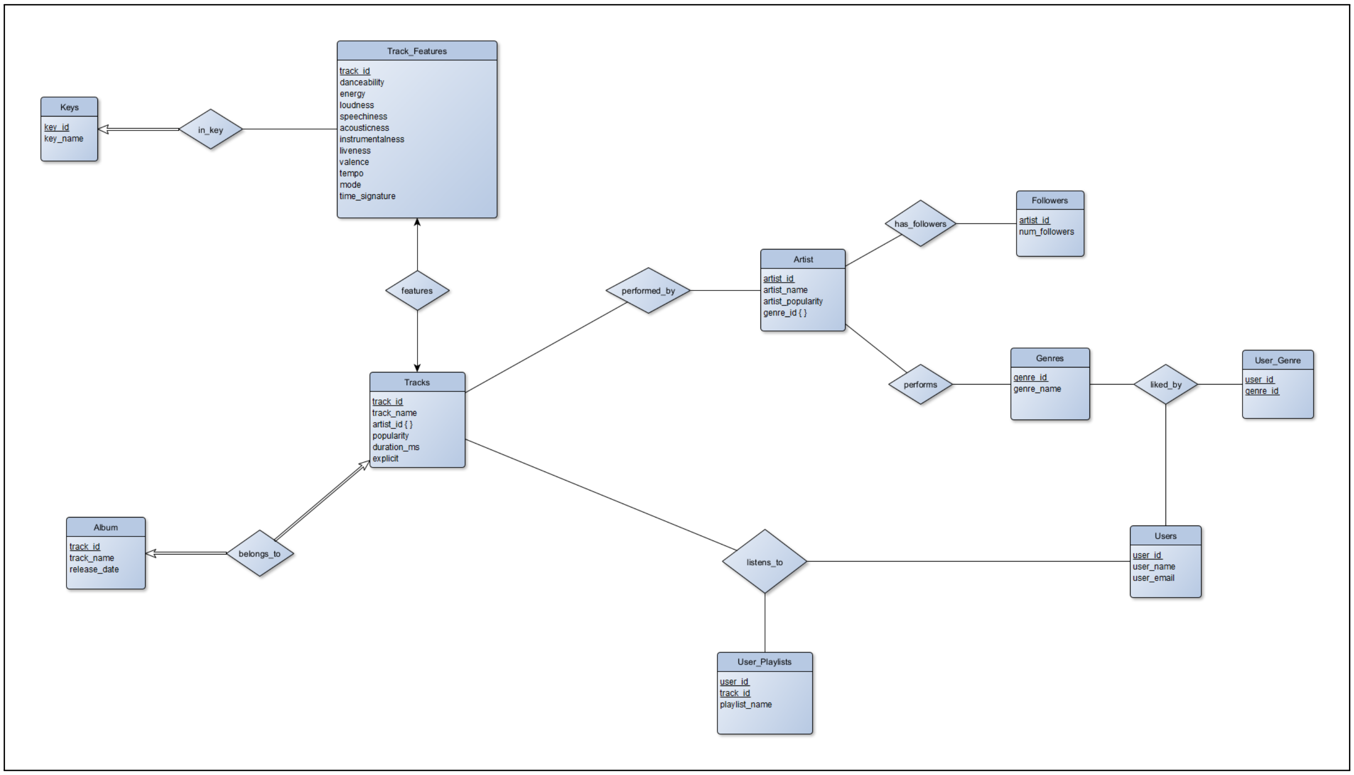


Fig. 1. ER Diagram Before Normalization

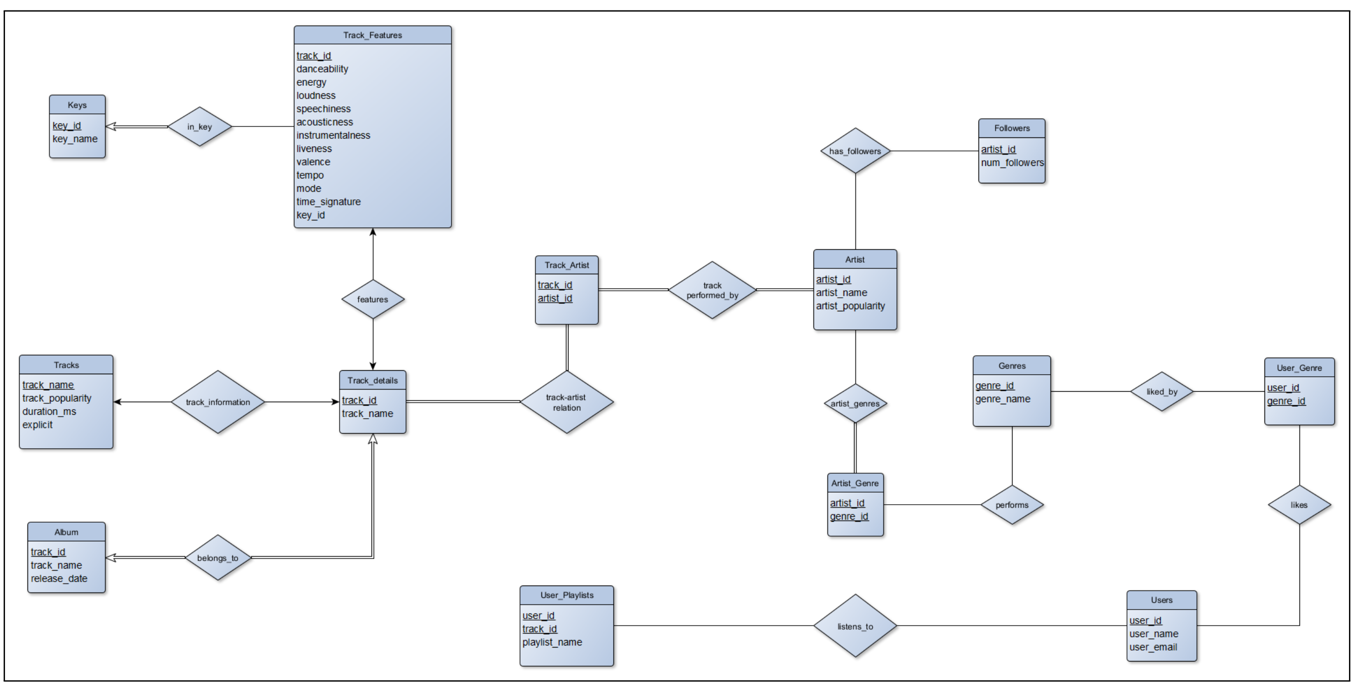


Fig. 2. ER Diagram After Normalization

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:

$$\text{track\_id} \rightarrow \text{track\_name}$$
$$\text{track\_id} \rightarrow\rightarrow \text{artist\_id}$$
$$\text{track\_name} \rightarrow \{\text{track\_popularity}, \text{duration\_ms}, \text{explicit}\}$$

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:

$$\text{track\_id} \rightarrow \{\text{track\_name}\}$$

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:

$$\text{track\_id} \rightarrow \{\text{artist\_id}\}$$

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:

$$\text{track\_name} \rightarrow \{\text{track\_popularity}, \text{duration\_ms}, \text{explicit}\}$$

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:

$$\text{user\_id} \rightarrow \{\text{username, user\_email}\}$$

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:

$$\begin{aligned}&\text{track\_id} \rightarrow \{\text{danceability, energy, loudness, speechiness,}\\&\text{acousticness, instrumentalness, liveness, valence, tempo,}\\&\text{mode, key\_id, time\_signature}\}\end{aligned}$$

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 enjoybility 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.

```sql
-- CREATE TABLE: track_details
CREATE TABLE track_details (
    track_id VARCHAR(255),
    track_name VARCHAR(10000),
    PRIMARY KEY (track_id)
);
```

Fig. 3. Table Creation: TRACK_DETAILS

```sql
-- CREATE TABLE: tracks
CREATE TABLE tracks (
    track_name VARCHAR(10000),
    track_popularity INTEGER,
    duration INTEGER,
    explicit INTEGER,
    PRIMARY KEY (track_name)
);
```

Fig. 4. Table Creation: TRACKS

```sql
-- CREATE TABLE: artists
CREATE TABLE artists (
    artist_id VARCHAR(255),
    artist_name VARCHAR(500),
    artist_popularity INTEGER,
    PRIMARY KEY (artist_id)
);
```

Fig. 5. Table Creation: ARTISTS

```sql
-- CREATE TABLE: album
CREATE TABLE album (
    track_id VARCHAR(255),
    track_name VARCHAR(10000),
    release_date DATE,
    PRIMARY KEY (track_id),
    FOREIGN KEY (track_id) REFERENCES track_details(track_id)
    ON DELETE CASCADE
);
```

Fig. 6. Table Creation: ALBUM

```sql
-- CREATE TABLE: genre
CREATE TABLE genre (
    genre_id VARCHAR(255),
    genre_name VARCHAR(500),
    PRIMARY KEY (genre_id)
);
```

Fig. 7. Table Creation: GENRE

```sql
-- CREATE TABLE: track_features
CREATE TABLE track_features (
    track_id VARCHAR(255),
    danceability REAL,
    energy REAL,
    loudness REAL,
    mode INTEGER,
    speechiness REAL,
    acousticness REAL,
    instrumentalness REAL,
    liveness REAL,
    valence REAL,
    tempo REAL,
    time_signature INTEGER,
    key_id INTEGER,
    PRIMARY KEY (track_id),
    FOREIGN KEY (track_id) REFERENCES track_details(track_id)
    ON DELETE CASCADE
);
```

Fig. 8. Table Creation: TRACK_FEATURES

```sql
-- CREATE TABLE: keys
CREATE TABLE keys (
    key_id VARCHAR(255),
    key_name VARCHAR(500),
    PRIMARY KEY (key_id)
);
```

Fig. 9. Table Creation: KEYS

```sql
-- CREATE TABLE: artist_genre
CREATE TABLE artist_genre (
    artist_id VARCHAR(255),
    genre_id VARCHAR(255),
    PRIMARY KEY (artist_id, genre_id),
    FOREIGN KEY (artist_id) REFERENCES artists(artist_id),
    FOREIGN KEY (genre_id) REFERENCES genre(genre_id)
    ON DELETE CASCADE
);
```

Fig. 10. Table Creation: ARTIST_GENRE

```sql
-- CREATE TABLE: followers
CREATE TABLE followers (
    artist_id VARCHAR(255),
    followers REAL,
    PRIMARY KEY (artist_id),
    FOREIGN KEY (artist_id) REFERENCES artists(artist_id)
    ON DELETE CASCADE
);
```

Fig. 11. Table Creation: FOLLOWERS

```sql
-- CREATE TABLE: users
CREATE TABLE users (
    user_id VARCHAR(255),
    user_name VARCHAR(255),
    user_email VARCHAR(255),
    PRIMARY KEY (user_id)
);
```

Fig. 12. Table Creation: USERS

```sql
-- CREATE TABLE: user_playlist
CREATE TABLE user_playlist (
    user_id VARCHAR(255),
    track_id VARCHAR(255),
    playlist_name VARCHAR(255),
    PRIMARY KEY (user_id, track_id),
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (track_id) REFERENCES track_details(track_id)
    ON DELETE CASCADE
);
```

Fig. 13. Table Creation: USER_PLAYLIST

```sql
-- CREATE TABLE: track_artist
CREATE TABLE track_artist (
    track_id VARCHAR(255),
    artist_id VARCHAR(255),
    PRIMARY KEY (track_id, artist_id),
    FOREIGN KEY (track_id) REFERENCES track_details(track_id),
    FOREIGN KEY (artist_id) REFERENCES artists(artist_id)
    ON DELETE CASCADE
);
```

Fig. 14. Table Creation: TRACK_ARTIST

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

```sql
------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------IMPORTING DATA-----------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------

-- insert data from track_details csv, which is created from the original dataset
COPY track_details (track_id, track_name)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/track_details.csv' DELIMITER ',' CSV HEADER;


-- insert data from track csv, which is created from the original dataset
COPY tracks (track_name, track_popularity, duration, explicit)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/track.csv' DELIMITER ',' CSV HEADER;


-- insert data from artist csv, which is created from the original dataset
COPY artists (artist_id, artist_name, artist_popularity)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/artist.csv' DELIMITER ',' CSV HEADER;


-- insert data from album csv, which is created from the original dataset
COPY album (track_id, track_name, release_date)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/album.csv' DELIMITER ',' CSV HEADER


-- insert data from genre csv, which is created from the original dataset
COPY genre (genre_id, genre_name)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/genre.csv' DELIMITER ',' CSV HEADER
```

Fig. 15. Data Import into tables 1

```sql
-- insert data from features csv, which is created from the original dataset
COPY track_features (track_id, danceability, energy, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo,
                     time_signature, key_id)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/features.csv' DELIMITER ',' CSV HEADER


-- insert data from key csv, which is created from the original dataset
COPY keys (key_id, key_name)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/key.csv' DELIMITER ',' CSV HEADER


-- insert data from follower csv, which is created from the original dataset
COPY followers (artist_id, followers)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/follower.csv' DELIMITER ',' CSV HEADER


-- insert data from user csv, which is created from the original dataset
COPY users (user_id, user_name, user_email)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/user.csv' DELIMITER ',' CSV HEADER


-- insert data from user_playlist csv, which is created from the original dataset
COPY user_playlist (user_id, playlist_name, track_id)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/user_playlist.csv' DELIMITER ',' CSV HEADER


-- insert data from track_artist csv, which is created from the original dataset
COPY track_artist (track_id, artist_id)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/track_artist.csv' DELIMITER ',' CSV HEADER


-- insert data from artist_genre csv, which is created from the original dataset
COPY artist_genre (artist_id, genre_id)
FROM 'D:/UB/Spring 2024/CSE 560 - Data Models and Query Language/Project/artist_genre.csv' DELIMITER ',' CSV HEADER
```

Fig. 16. Data Import into tables 2

XII. Issues working with large database

The major problem with working on a large database it 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.

```sql
SELECT * FROM album WHERE track_name = 'Love Yourself';
```

```
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 to 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.

```sql
SELECT * FROM album WHERE track_name = 'Love Yourself';

CREATE INDEX idx_album ON album(track_id, track_name);
```

```
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.

```sql
-----------------------------------------------------------------------------------------------------
----------------------------------------INSERT DATA INTO TABLES---------------------------------------
-----------------------------------------------------------------------------------------------------

INSERT INTO users (user_id, user_name, user_email) VALUES ('10001', 'ShivanMathur', 'shivanma@buffalo.edu');
INSERT INTO users (user_id, user_name, user_email) VALUES ('10002', 'DebosmitNeogi', 'debosmit@buffalo.edu');
```

```
Data Output   Messages   Notifications

INSERT 0 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.

```sql
-- 2. Inserting new artist record into all the relevant tables
INSERT INTO artists (artist_id, artist_name, artist_popularity) VALUES ('01HWY4F4SMKCEEDF1GDQB0BQCY', 'Utkarsh Mathur', '10');
INSERT INTO artist_genre (artist_id, genre_id) VALUES ('01HWY4F4SMKCEEDF1GDQB0BQCY', '1400');
INSERT INTO artist_genre (artist_id, genre_id) VALUES ('01HWY4F4SMKCEEDF1GDQB0BQCY', '2331');
INSERT INTO followers (artist_id, followers) VALUES ('01HWY4F4SMKCEEDF1GDQB0BQCY', 50);
INSERT INTO track_artist (track_id, artist_id) VALUES ('1h0qIqWO2n9vV4QTyszRhm', '01HWY4F4SMKCEEDF1GDQB0BQCY');
```

```
Data Output   Messages   Notifications

INSERT 0 1

Query returned successfully in 110 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.

```sql
-----------------------------------------------------------------------------------------------------
--------------------------------------------- UPDATE DATA -------------------------------------------
-----------------------------------------------------------------------------------------------------

-- 1. Update specific artist's genre in the table:
UPDATE artist_genre SET genre_id = '76'
WHERE artist_id = '01HWY4F4SMKCEEDF1GDQB0BQCY'
AND genre_id = '1400';
```

```
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.

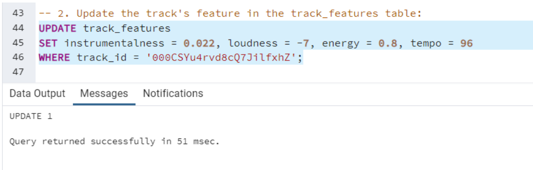


Fig. 22. Data Update 2

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.

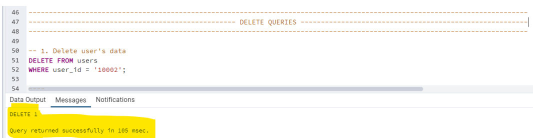


Fig. 23. Data Deletion 1

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

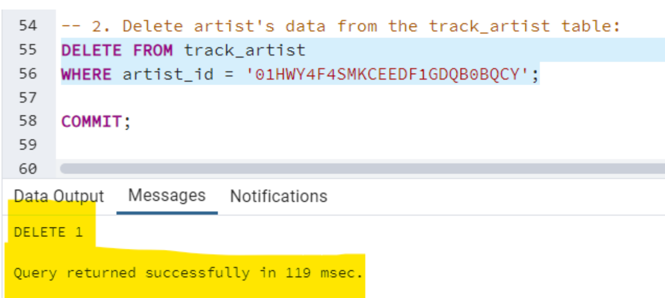


Fig. 24. Data Deletion 2

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:

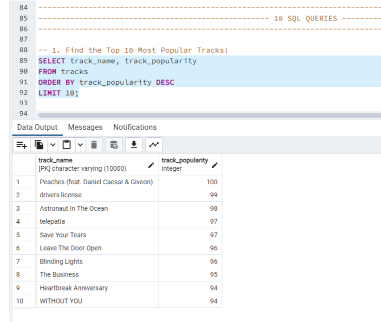


Fig. 25. SQL SELECT Query 1

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

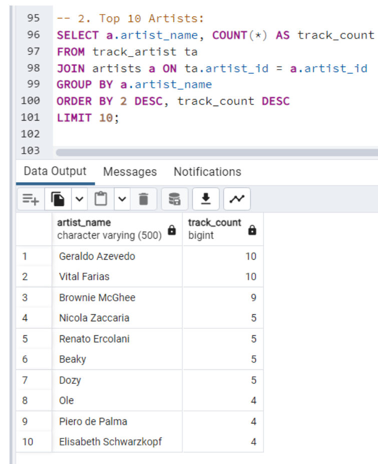


Fig. 26. SQL SELECT Query 2

3. Query to find Most Popular Genre:

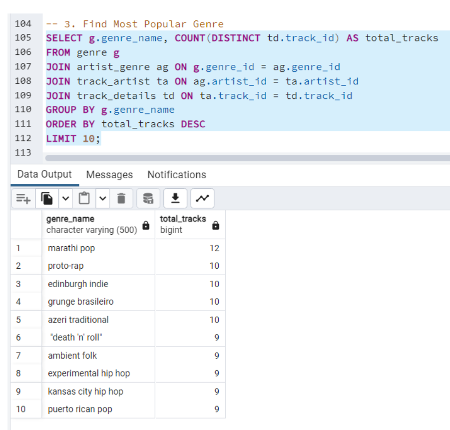
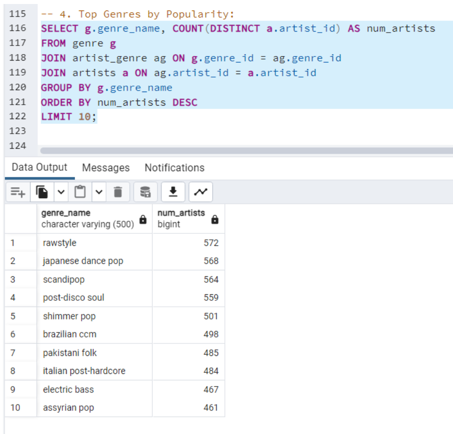


Fig. 27. SQL SELECT Query 3

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



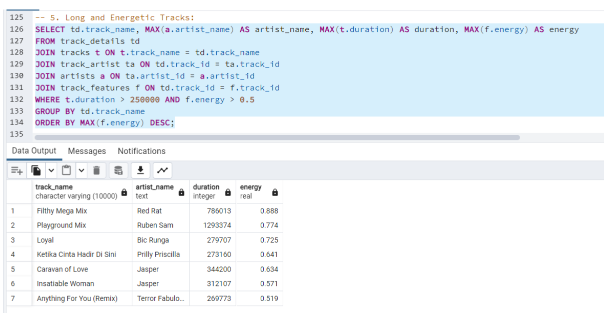
```sql
-- 4. Top Genres by Popularity:
SELECT g.genre_name, COUNT(DISTINCT a.artist_id) AS num_artists
FROM genre g
JOIN artist_genre ag ON g.genre_id = ag.genre_id
JOIN artists a ON ag.artist_id = a.artist_id
GROUP BY g.genre_name
ORDER BY num_artists DESC
LIMIT 10;
```

Data Output Messages Notifications

	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:



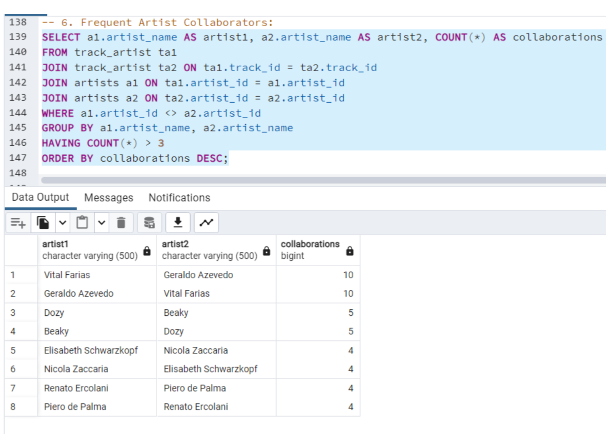
```sql
-- 5. Long and Energetic Tracks:
SELECT td.track_name, MAX(a.artist_name) AS artist_name, MAX(t.duration) AS duration, MAX(f.energy) AS energy
FROM track_details td
JOIN tracks t ON t.track_name = td.track_name
JOIN track_artist ta ON td.track_id = ta.track_id
JOIN artists a ON ta.artist_id = a.artist_id
JOIN track_features f ON td.track_id = f.track_id
WHERE t.duration > 250000 AND f.energy > 0.5
GROUP BY td.track_name
ORDER BY MAX(f.energy) DESC;
```

Data Output Messages Notifications

	track_name character varying (10000)	artist_name text	duration integer	energy real
1	Filthy Mega Mix	Red Rat	786013	0.888
2	Playground Mix	Ruben Sam	1293374	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:



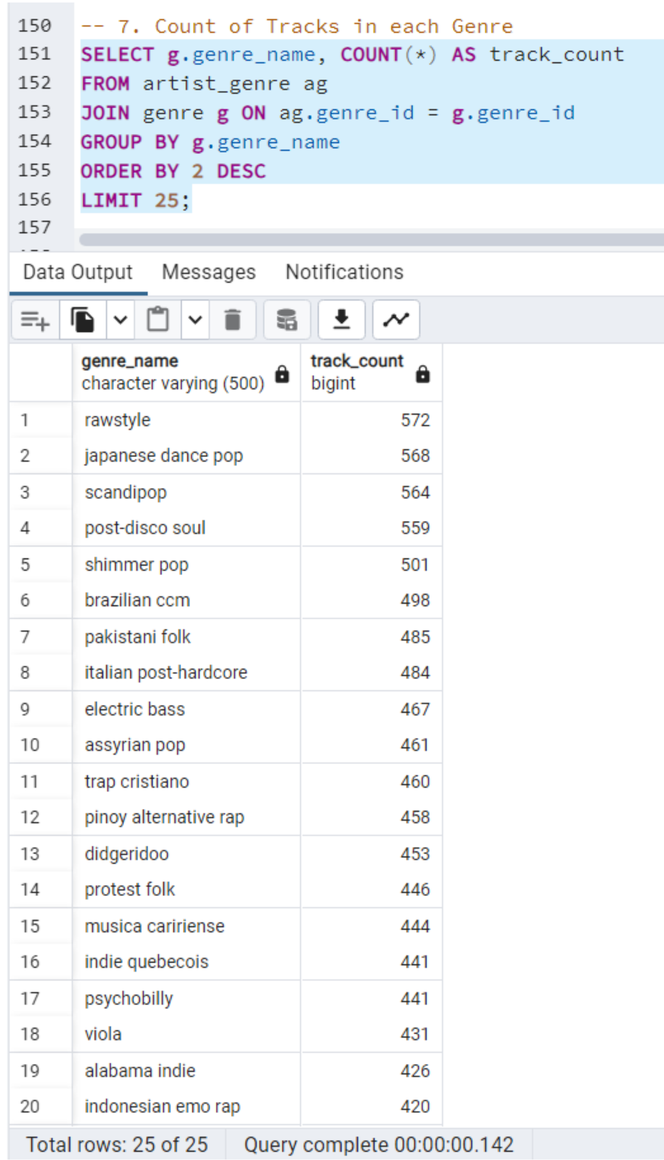
```sql
-- 6. Frequent Artist Collaborators:
SELECT a1.artist_name AS artist1, a2.artist_name AS artist2, COUNT(*) AS collaborations
FROM track_artist ta1
JOIN track_artist ta2 ON ta1.track_id = ta2.track_id
JOIN artists a1 ON ta1.artist_id = a1.artist_id
JOIN artists a2 ON ta2.artist_id = a2.artist_id
WHERE a1.artist_id <> a2.artist_id
GROUP BY a1.artist_name, a2.artist_name
HAVING COUNT(*) > 3
ORDER BY collaborations DESC;
```

Data Output Messages Notifications

	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:



```sql
-- 7. Count of Tracks in each Genre
SELECT g.genre_name, COUNT(*) AS track_count
FROM artist_genre ag
JOIN genre g ON ag.genre_id = g.genre_id
GROUP BY g.genre_name
ORDER BY 2 DESC
LIMIT 25;
```

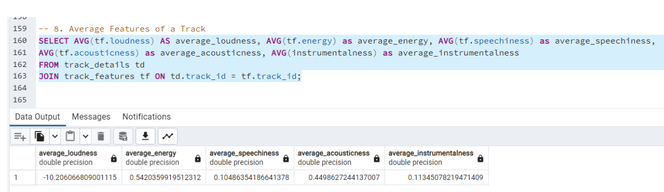
Data Output Messages Notifications

	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:



```sql
-- 8. Average Features of a Track
SELECT AVG(tf.loudness) AS average_loudness, AVG(tf.energy) as average_energy, AVG(tf.speechiness) as average_speechiness,
AVG(tf.acousticness) as average_acousticness, AVG(instrumentalness) as average_instrumentalness
FROM track_details td
JOIN track_features tf ON td.track_id = tf.track_id;
```

Data Output Messages Notifications

	average_loudness double precision	average_energy double precision	average_speechiness double precision	average_acousticness double precision	average_instrumentalness double precision
1	-10.206066809001115	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:

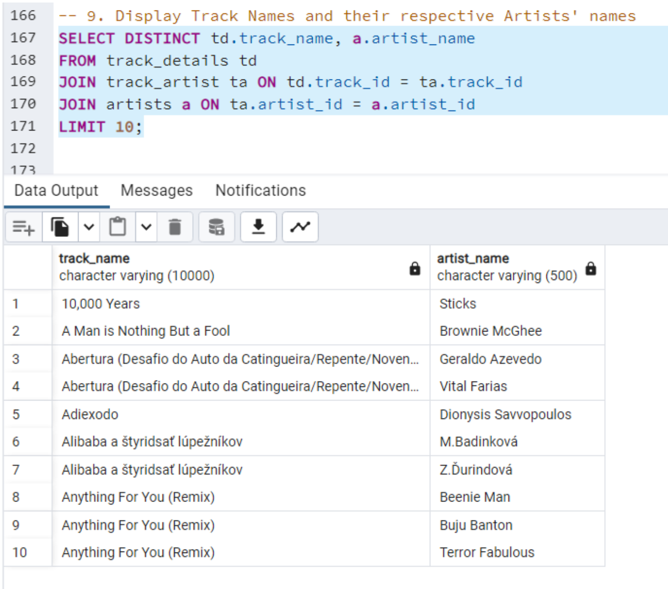


Fig. 33. SQL SELECT Query 9

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

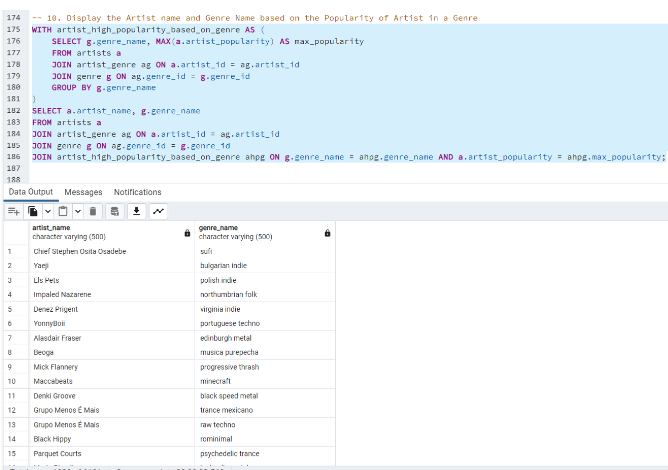


Fig. 34. SQL SELECT Query 10

XIV. Query Execution Analysis

A. Problematic Query - 1

There are cases when after performing some DML operations on a table, we need to take actions on the related tables too. A way to automate this process, SQL has the mechanism to handle this task. We can use functions and triggers, such that whenever a certain task has been performed, the trigger will be invoked to take actions on the other required tables.

In our case, when we want to delete a user record from the 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.

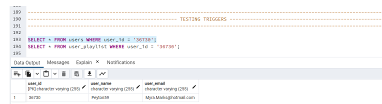


Fig. 35. User record in users' table before deletion

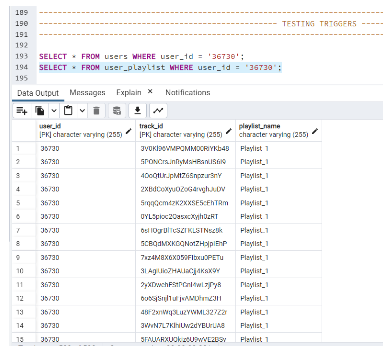


Fig. 36. User record in user_playlist table before deletion

Fig. 37. Deletion error; violating foreign key constraint

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.

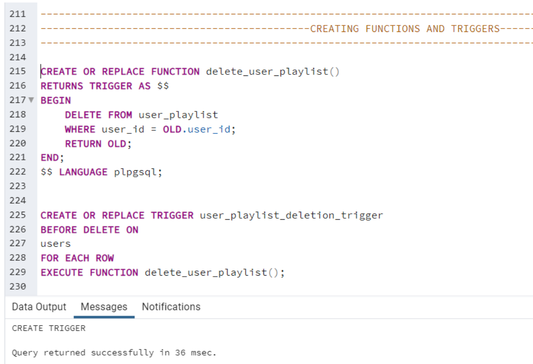


Fig. 38. Function and Trigger Creation

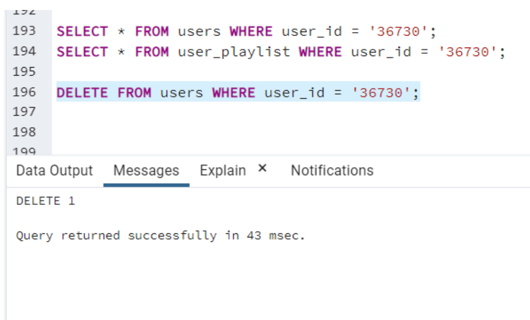


Fig. 39. User record deleted successfully

Verifying user record deleted successfully:

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

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.

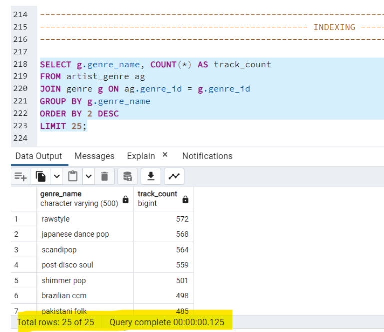


Fig. 42. Query Execution time before indexing

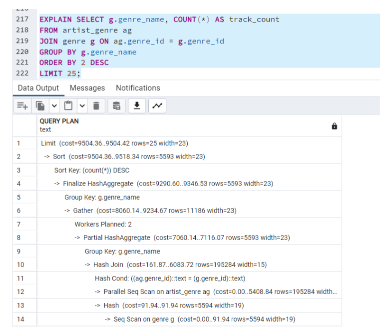


Fig. 43. Query execution statistics

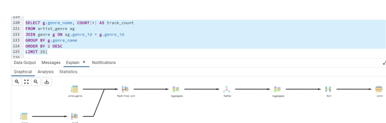


Fig. 44. Query Execution Plan

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.

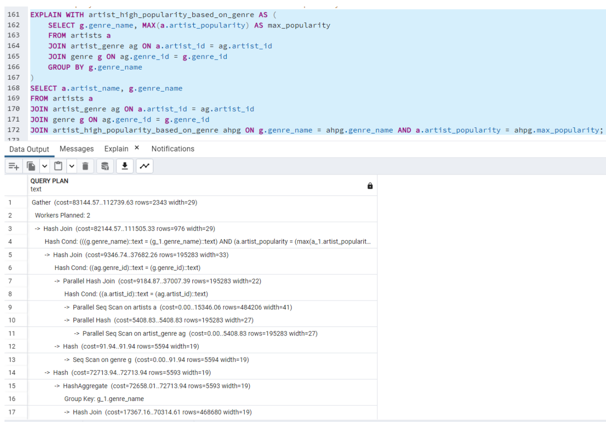


Fig. 46. Problematic Query 3 Analysis using EXPLAIN

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

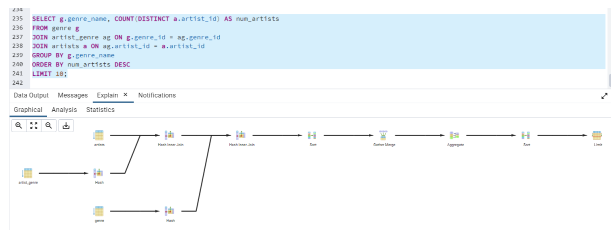


Fig. 49. Problematic Query 4 Analysis using EXPLAIN

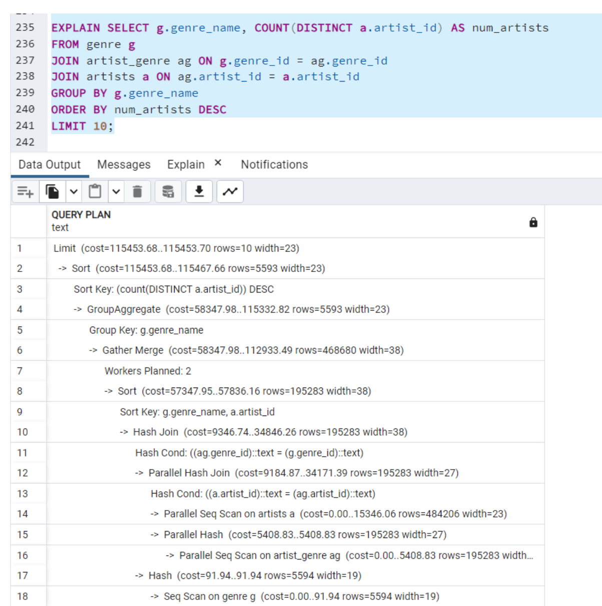


Fig. 50. Problematic Query 4 Analysis using EXPLAIN

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.

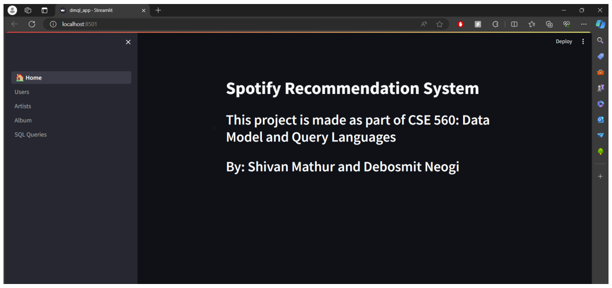


Fig. 51. Home Page

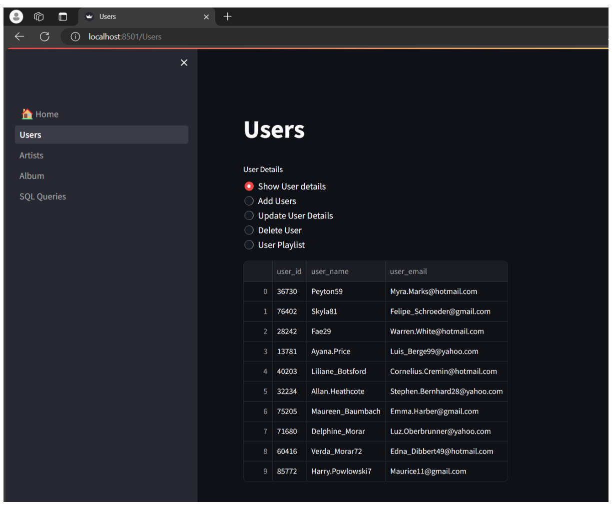


Fig. 52. User Page: Show User Details

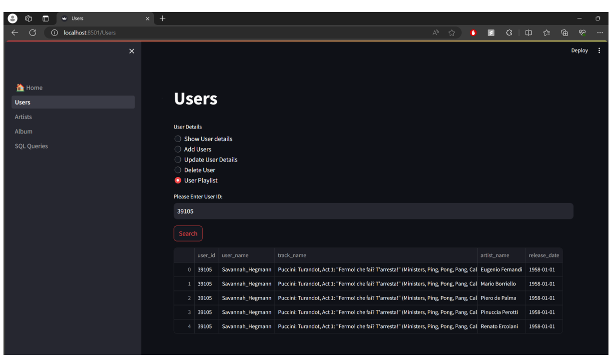


Fig. 53. User Page: Show User Playlist

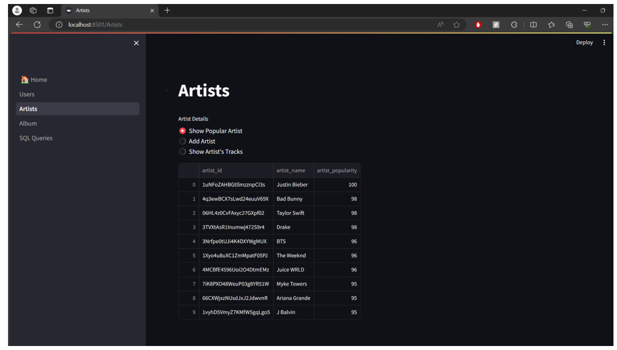


Fig. 54. Artists Page: Show Popular Artist

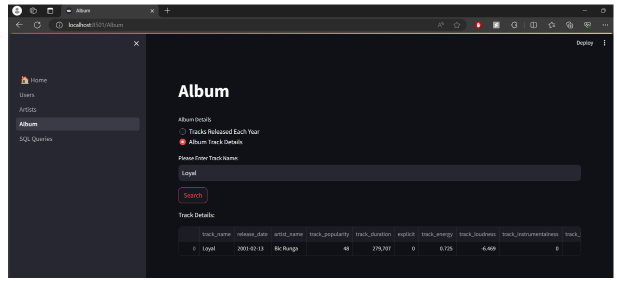


Fig. 55. Album Page: Show Album Track Details

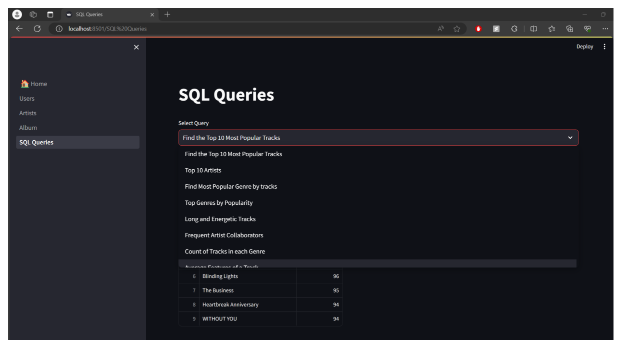


Fig. 56. SQL Queries: Drop-down option to display the required results

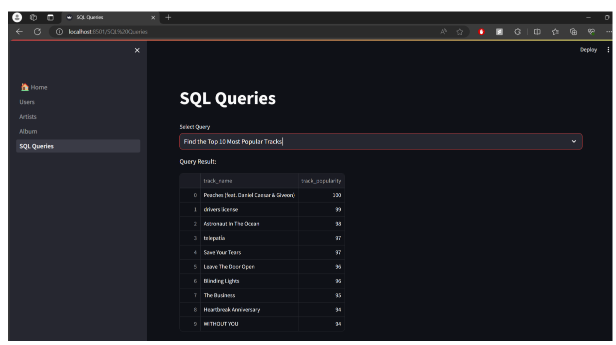


Fig. 57. SQL Queries: Display Top 10 Most Popular Tracks

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