Anthony Sayre

Spotify Data Exploration Project

1. Dataset Selection

For this project, I chose two public music-related datasets from Kaggle that complement each other really well. One focuses on artist-level information and the other digs into track-level data with detailed audio features. I picked them because they both had enough variety and size to support some deeper analysis, and I was especially curious to explore how popularity and musical attributes relate to each other across artists and genres.

Datasets used from Kaggle:
- Top 10K Artists and Their Popular Songs
- Spotify Dataset for SQL Practice

Artist dataset structure:
Roughly 10,000 records
Columns: Name, ID, Gender, Age, Country, Genres, Popularity, Followers, URI

Track dataset structure:
Around 1,000 tracks
Columns: Genre, Artist Name, Track Name, Track ID, Popularity, plus 12 audio features like danceability, energy, tempo, etc.

Some of the questions I wanted to explore:
- Do more popular artists tend to produce tracks with higher energy?
- Are there genres that consistently score high in valence or danceability?
- Can we spot patterns in tempo or loudness across top-performing songs?

2. Database Design and Implementation

I created a MySQL database called spotify_db and designed the schema to follow Third Normal Form. The original datasets had some overlap, but once I took a closer look, it made the most sense to separate the data into two logical tables: one for artists and one for tracks. Every track references its artist using a foreign key, and each table is built so that all of its columns depend only on its primary key.

To figure out the structure, I looked for columns that were being repeated or didn't belong in the same table. Artist details like name, genre, and popularity stay the same across multiple songs, so I grouped them together and gave each artist a unique ID. For tracks, I used track_id as the primary key and kept all the audio-related attributes in that table since they describe the songs themselves. Connecting the two tables through artist_id helped eliminate redundancy and made it easier to write queries that pull from both sides without duplicating information.

UML Diagram

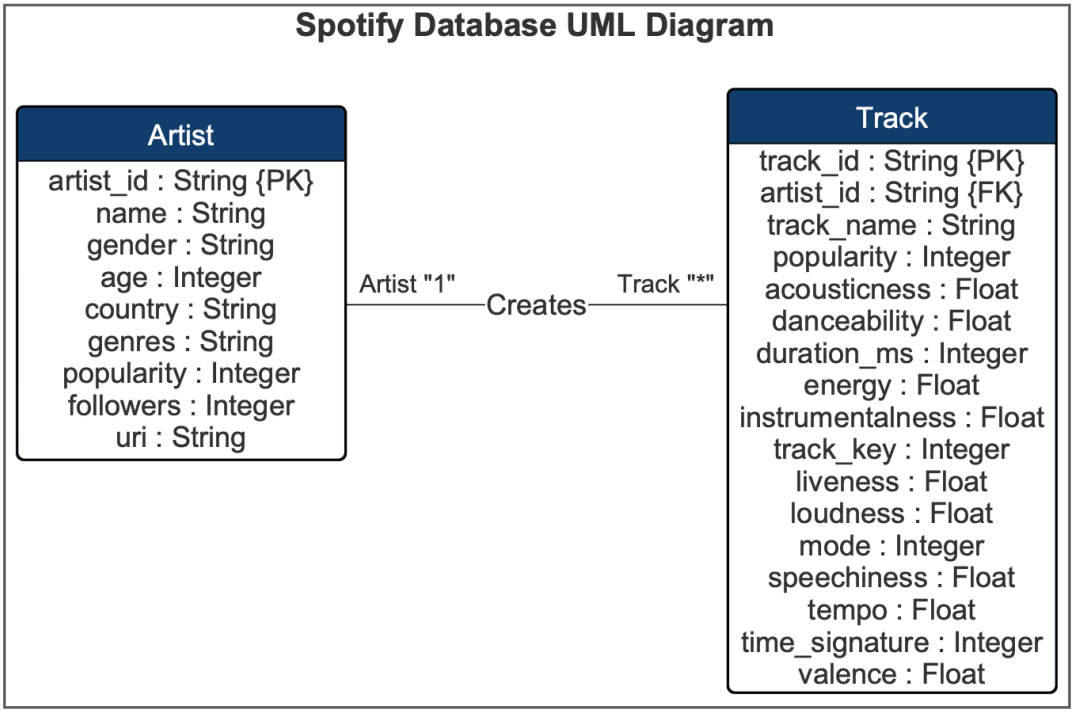

Figure 1: UML Class Diagram for the Spotify Music Database

The artists table includes attributes like name, genre, and popularity. The tracks table stores song metadata and audio features. Instead of trying to cram artist details into the tracks table, I kept them separate to maintain 3NF and reduce redundancy. I also renamed the column “key” to “track_key” to avoid keyword conflicts.

Design highlights:

- Each artist has a unique ID and each track has a unique track_id
- tracks.artist_id is a foreign key referencing artists.ID
- All audio-related attributes live in the tracks table, so I could easily filter and group by them in queries

SQL Schema

```sql
-- Drop Database and tables if they exist
DROP DATABASE IF EXISTS spotify_db;
CREATE DATABASE spotify_db;
USE spotify_db;

-- Create Artist and Tracks tables
CREATE TABLE artists (
    Name VARCHAR(255),
    ID VARCHAR(100) PRIMARY KEY,
    Gender VARCHAR(50),
    Age INT,
    Country VARCHAR(100),
    Genres TEXT,
    Popularity INT,
    Followers INT,
    URI VARCHAR(255)
);
CREATE TABLE tracks (
    genre            VARCHAR(100),
    artist_name      VARCHAR(255),
    track_name       VARCHAR(255),
    track_id         VARCHAR(100) PRIMARY KEY,
    popularity       INT,
    acousticness     FLOAT,
    danceability     FLOAT,
    duration_ms      INT,
    energy           FLOAT,
    instrumentalness FLOAT,
    track_key        VARCHAR(10),
    liveness         FLOAT,
    loudness         FLOAT,
    mode             VARCHAR(10),
    speechiness      FLOAT,
    tempo            FLOAT,
    time_signature   INT,
    valence          FLOAT
);

-- Enable loading local files
SET GLOBAL local_infile = 1;

-- Load data into tables
LOAD DATA LOCAL INFILE 'US Top 10K Artists.csv'
INTO TABLE artists
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(Name, ID, Gender, Age, Country, Genres, Popularity, Followers, URI);

LOAD DATA LOCAL INFILE 'Spotify Tracks DB.csv'
INTO TABLE tracks
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(genre, artist_name, track_name, track_id, popularity, acousticness, danceability, duration_ms, energy,
 instrumentalness, track_key, liveness, loudness, mode, speechiness, tempo, time_signature, valence);
```

Figure 2: SQL Schema and data loading

To support the structure I designed, I wrote a full SQL file that creates the database, defines both tables, and loads in the data directly from the CSVs. I made sure the columns matched the CSV headers exactly, and I renamed the “key” column in the tracks file to “track_key” to avoid conflicts. Each table is created in 3NF, and the data is loaded using LOAD DATA LOCAL INFILE, which made it easy to test and reload as needed.

The file also includes a few setup commands like dropping the database if it already exists and enabling local file loading in MySQL. This made the workflow more flexible while I was building and debugging queries. Once everything was set up, I was able to jump straight into writing and running SQL queries without needing to clean or reshape anything manually.

3. Data Exploration and Insights

I wrote 14 SQL queries to explore trends, compare genres, analyze artist output, and look at relationships between different audio features. Here’s each query and the ones with standout results.

Query 1: Count of artists by country

```
SELECT Country, COUNT(*) AS artist_count
FROM artists
GROUP BY Country
ORDER BY artist  count DESC;
```

This query breaks down how many artists are in the dataset per country. The U.S. had the most (1,207), followed by the U.K. and Germany. What stood out to me was the blank country value at the top with 3,183 entries, which probably means the dataset had a lot of missing or unspecified country values.

Query 2: Count of tracks by genre

```
SELECT genre, COUNT(*) AS track_count
FROM tracks
GROUP BY genre
ORDER BY track  count DESC;
```

Comedy surprisingly had the most tracks, which I didn’t expect. Genres like Electronic, Alternative, and Anime were also way up there. Classical and Reggae were close too, showing how well-represented niche and global genres are in the dataset.

Query 3: Top 10 most popular artists

```
SELECT Name, Popularity
FROM artists
ORDER BY Popularity DESC
LIMIT 10;
```

This query aligned with my expectations. The top 10 included Taylor Swift, Bad Bunny, Drake, and The Weeknd, all scoring in the 90s or 100s for popularity. It showed that the dataset aligns with real-world artist rankings.

Query 4: Top 10 tracks by energy

```
SELECT track_name, energy
FROM tracks
ORDER BY energy DESC
LIMIT 10;
```

All ten songs had an energy score of 0.999, which was surprising. I expected more variation, but it seems that extremely energetic tracks cluster right at the top. I did manually verify other tracks had more variation but a lot were maxed out at the energy level.

Query 5: Average danceability by artist

```
SELECT artist_name, ROUND(AVG(danceability), 3) AS avg_danceability
FROM tracks
GROUP BY artist_name
ORDER BY avg_danceability DESC
LIMIT 10;
```

This was my favorite query to run as the highest ranked artist by danceability was Vanilla Ice. Something I did not expect to see in the year 2025. With his popularity peaking in the late 90s.

Query 6: Tracks by U.S. artists (join)

```
SELECT t.track_name, a.Name AS artist, a.Country
FROM tracks AS t
JOIN artists AS a
  ON t.artist_name = a.Name
WHERE a.Country = 'US'
LIMIT 10;
```

This was a join to check which songs were made by American artists. Most of the results were what I expected a mix of pop and hip-hop artists.

Query 7: Number of tracks per artist (join)

```
SELECT a.Name AS artist, COUNT(t.track_id) AS total_tracks
FROM artists a
JOIN tracks t ON a.Name = t.artist_name
GROUP BY a.Name
HAVING total_tracks > 5
ORDER BY total_tracks DESC
LIMIT 10;
```

This query aims to identify the artist with the most tracks. Mozart holds the record with over 800 tracks. In comparison, even the top current artists have significantly fewer tracks. It's fascinating to consider that this was achieved in an era before the advent of electronics and the extensive manual labor involved.

Query 8: Average valence per genre

```
SELECT genre, ROUND(AVG(valence), 3) AS avg_valence
FROM tracks
GROUP BY genre
ORDER BY avg_valence DESC;
```

This query calculates the average valence (or musical "positivity") for each genre in the dataset. It gives a general sense of which genres tend to sound happier or more upbeat.

Query 9: Tracks with above-average popularity (subquery)

```
SELECT track_name, popularity
FROM tracks
WHERE popularity > (
  SELECT AVG(popularity)
  FROM tracks
);
```

This query finds all tracks whose popularity score is greater than the overall average popularity across the dataset. It uses a subquery to calculate that average and then filters the results accordingly.

Query 10: Create a view for high-popularity tracks

```
CREATE VIEW high_popularity_tracks AS
SELECT track_name, artist_name, popularity
FROM tracks
WHERE popularity >= 80;
```

This query creates a view called high_popularity_tracks that stores all songs with a popularity score of 80 or higher. It's useful for reusing this filtered data in later queries without having to rewrite the conditions.

Query 11: Select from high-popularity view

```
SELECT *
FROM high_popularity_tracks;
```

This pulls everything from the view created in Query 10. It's a way to get a list of the most popular songs in the dataset without repeating the filtering logic.

Query 12: Average track duration per genre

```
SELECT genre, ROUND(AVG(duration_ms)/1000, 1) AS avg_duration_sec
FROM tracks
GROUP BY genre
ORDER BY avg_duration_sec DESC;
```

This query calculates the average track duration in seconds for each genre. It helps show which genres tend to have longer or shorter songs on average.

Query 13: Most common track keys by genre

```
SELECT t.genre, t.track_key, COUNT(*) AS count
FROM tracks t
JOIN artists a ON t.artist_name = a.Name
GROUP BY t.genre, t.track_key
ORDER BY count DESC
LIMIT 10;
```

This finds the most frequently used key signatures for each genre. It uses a join with the artist table and groups by both genre and track key, then returns the most common combinations.

Query 14: Average energy by track key

```
SELECT t.track_key, ROUND(AVG(t.energy), 3) AS avg_energy
FROM tracks t
JOIN artists a ON t.artist_name = a.Name
GROUP BY t.track_key
ORDER BY avg_energy DESC;
```

This query looks at whether certain key signatures tend to have higher or lower energy. It groups songs by key and calculates the average energy for each one. B had the highest average energy, while D# had the lowest. Interesting enough I find D# or Eb to be one of the more emotional key signatures relating to why it would be on the opposite end of the ‘energy’ spectrum.

4. Final Summary and Reflection

Working on this project helped me see how important normalization is when it comes to organizing a database and writing flexible queries. It also gave me the chance to look at the kind of music data I already enjoy in a more structured and technical way. The datasets seemed pretty straightforward at first, but once I got deeper into writing queries and connecting the tables, I realized how much insight you can actually get from a clean relational design.

A lot of the results matched what I expected, but it still felt really rewarding to back those ideas up with actual numbers. Getting to use different query types like groupings, joins, and

subqueries also gave me more confidence with SQL overall. One thing I might have done differently is splitting out all the track-level audio features into a separate table instead of keeping them all in tracks. That could've made the structure even more normalized and easier to expand if I wanted to include more features later on. But overall, I'm happy with how the design turned out and how easy it was to work with once everything was loaded.