Spotify Data Dive: A Relational Database Design Project

Luca Nardone and Hongjie Zheng
IST-Database - DBM1

This report outlines the design and implementation of a database for a music streaming platform inspired by real-world applications like Spotify. The project demonstrates a practical journey through the database lifecycle, starting from domain understanding, moving to normalized schema design, and culminating in a robust implementation in PostgreSQL.

The main objective was to build a scalable and efficient database that manages key entities such as songs, albums, artists, users, playlists, and subscription plans. By simulating realistic user interactions, such as song listening, rating, and playlist creation, the database offers insights into user preferences and behaviors. Normalization techniques, integrity constraints, and performance optimizations ensure that the database meets functional and scalability requirements.

This report details the key phases of the project: dataset preprocessing, schema design, implementation, query optimization, and addressing the challenges encountered.

The project began with the selection of the **Top 10000 Songs on Spotify 1950-Now** dataset from Kaggle 🔗, which contains artists, albums and their respective songs identified by their spotify URL.

Dataset preprocessing

However, the raw dataset contained some redundancy and irregularities that needed to be addressed before it could be used effectively in our database.

To ensure the data was structured properly for efficient querying and analysis, we applied the **normalization** process to minimize redundancy and properly establish dependency.
We **splitted** the dataset into three main entities:

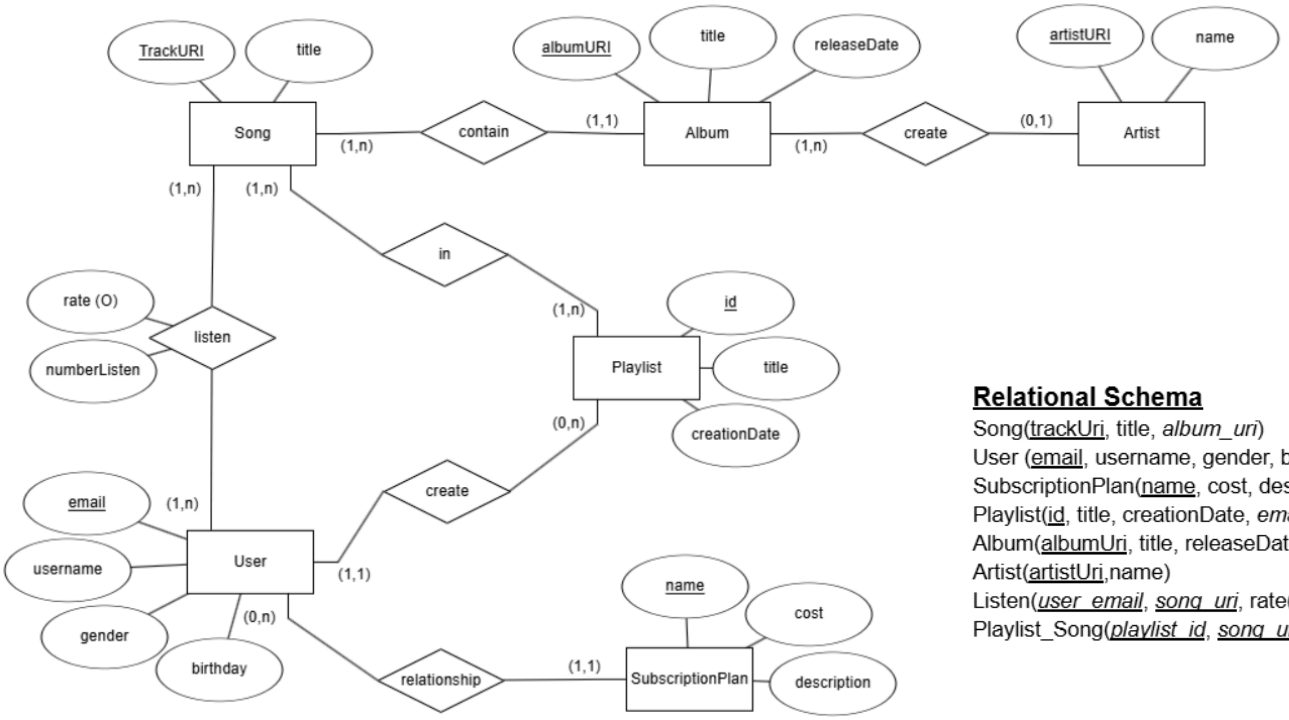
- Tracks: This entity represents the individual songs, storing each song's name and its unique identifier (Track URI), along with a reference to the album it belongs to (Album URI).
- Artists: The artists were separated into their own entity, identified by a unique Artist URI, with their corresponding name.
- Albums: The albums were also treated as a distinct entity, containing each album's name, release date, and its associated artist URI.

At the same moment, we also performed basic **data cleaning** to remove inconsistencies

1. **Data Formatting**: We standardized the format of the data, stripping leading or trailing spaces in URIs to prevent discrepancies.
2. **Eliminating duplicates**: We ensured that each song, artist, and album appeared only once in the dataset.
3. **Handling missing values**: Records with missing critical fields, such as a track's name or unique URI, were removed entirely, as they would have compromised the integrity of the database.

Database Desing

After obtaining the clean data, we first decided to understand how we wanted to structure our Database.
To do this we created an **ER diagram** with *ErdPlus* and the respective **relation schema**:



Relational Schema

Song(trackUri, title, *album_uri*)
User (email, username, gender, birthday, *subscription_plan*)
SubscriptionPlan(name, cost, description)
Playlist(id, title, creationDate, *email_user*)
Album(albumUri, title, releaseDate, *artist_uri*)
Artist(artistUri,name)
Listen(*user_email*, *song_uri*, rate(o), numberListen)
Playlist_Song(*playlist_id*, *song_uri*)

DataBase implementation

After cleaning and normalizing the dataset, the next step was to implement the database in PostgreSQL.
The structure of it is straightforward, with clear relationships linking songs to albums and albums to artist with cascading updates and delete to maintain consistency. We have also decided to leave this last report as optional, to allow the inclusion of albums by unknown authors.

Challenges Encountered

- **Foreign Key Dependencies**: During the insertion of albums, we encountered records with artists that did not exist in the database. To address this:
  - If the artist existed, the album was linked correctly.
  - If the artist was missing, the album's uri was set to `NULL` to maintain database integrity while allowing incomplete data to be stored for potential future updates.
- **Invalid Release Dates**: Some albums had incomplete or incorrect release dates which did not allow insertion according to the `DATE` format
  - If the date contained only a year or only the year and month, was added a default “01” for each incomplete part.
  - Rows with entirely invalid dates were skipped and these errors were logged for further analysis.

Extending the Database for a Music Application

Once the principal structure was set, we proceeded with the creation of the other entities and relationships, following our ER, to make the database not only suitable for containing songs, but a real music application.

The first step was the **generation** of a dataset of users, with essential information like email, username, gender, and birthdate, through the Mockaroo site. But before inserting them we had to create and insert the subscription plans, to be able to randomly assign the available membership tiers for each user as a foreign, during the users insert.

Next, for user-song interactions we create the table listen by inserting a user-song pair and random data to simulate user interactions with songs, as a rating and number of listens.
At the end we also created playlists for users, generating playlist names (with `uuid` python module) and linking them to random users. Then we created the song_playlist table to connect it playlist with random songs with a multi-multi relation.

Challenges Encountered

- **Generating random data presented some issues:** Generating data without proper checks, duplicate association pairs could have been inserted, violating database constraints.
  - We addressed this by tracking generated associations in Python to ensure uniqueness before inserting them into the database.
- **Reserved SQL Keywords:** The term *user* is a reserved keyword in PostgreSQL, preventing us from naming a table as "user".
  - We renamed the table to "users" to comply with SQL naming conventions

Query & relational Algebra

We designed several SQL queries to explore the database's capabilities and ensure the relationships between entities were implemented correctly:

1. Albums released between 2016 and 2020

```
SELECT albumURI, title, releaseDate
FROM album
WHERE releaseDate BETWEEN '2016-01-01' AND '2020-12-31';
```

π albumURI,title,releaseDate (σ2016−01−01≤releaseDate≤2020−12−31 (album))

OR

```
SELECT albumURI, title, releaseDate
FROM album
WHERE releaseDate >= '2016-01-01' AND releaseDate <= '2020-12-31';
```

π albumURI,title,releaseDate (σreleaseDate≥2016−01−01 ∧ releaseDate≤2020−12−31 (album))

2. Find all users of the "Female" genre who have created at least one playlist

```
SELECT email, username, gender
FROM users
WHERE gender = 'Female' AND email IN (SELECT email FROM playlist);
```

σ gender='Female'(Users)∩πemail(Playlist)

Advanced query

We also wanted to test our Database with more advanced Queries that allow for data analysis, which is very useful for music applications to check trends and progress:

3. The most popular subscription plan

```
SELECT subscription_plan, COUNT(*) AS users_count
FROM "users"
GROUP BY subscription_plan
ORDER BY users_count DESC
LIMIT 1;
```

4. The number of album created by all the artists

```
SELECT a.name AS artist_name, COUNT(al.albumuri) AS num_albums
FROM artist a
JOIN album al ON a.artisturi = al.artist_uri
GROUP BY a.name
ORDER BY num_albums DESC;
```

5. Get the most 3 popular album based on the total number of listens for its songs, including the album ID, title, and artist name

```
SELECT a.albumuri AS album_id, a.title AS album_title, ar.name AS artist_name,
SUM(l.number_listen) AS total_listens
FROM Album a
JOIN Song s ON s.album_uri = a.albumuri
JOIN listens l ON s.TrackURI = l.song_uri
JOIN Artist ar ON ar.artisturi = a.artist_uri
GROUP BY a.albumuri, a.title, ar.name
ORDER BY total_listens DESC
LIMIT 3;
```

In conclusion, thanks to queries, we have seen how the database can manage even more complex requests by integrating data from multiple entities.
Furthermore, the last query shows exactly the downside of normalization. Indeed retrieving data requires joining multiple tables leading to much more complex queries.

Performance Improvements and Database Optimization

As the dataset grew and the database became more complex, we focused on optimizing the performance of our DataBase.

Index

To improve query performance, we created indexes tailored to specific use cases.
There are two main kinds of index:

- **The B-tree index:** default index that can accelerate a wider variety of queries.
- **The Hash index:** performs well on some simple small tasks, but it does not perform well on range queries.

For experiment the difference performs tra i due, we tried with two query operations:

EXPLAIN ANALYZE
SELECT * FROM album WHERE title = 'Let It Bleed';

B-TREE = CREATE INDEX idx_album_title ON album(title)
HASH = CREATE INDEX idx_album_title ON album USING HASH (title);

EXPLAIN ANALYZE
SELECT * FROM album WHERE releaseDate > '2020-01-01';

B-TREE = CREATE INDEX idx_album_releaseDate ON album(releasedate)
HASH = CREATE INDEX idx_album_releaseDate ON album USING HASH (releasedate

Results:

Type of queries	Index	Planning time(ms)	Execution time(ms)
exact match query	NO index	4.049	22.968
	WITH index: B_tree	1.624	0.173
	WITH index: HASH	0.527	0.099
range query	NO index	0.098	1.309
	WITH index: B_tree	0.091	0.475
	WITH index: HASH	0.219	1.320

Therefore we can see that:

- First query, with both indexing the process becomes faster, but the **hash index** performs very better.
- Second query, the hash index becomes useless (since it is even worse than the process without index) and the B_tree index still performs well.

Integrity Constraints

To ensure data validity and consistency in the database, we applied several integrity constraints.

We ensured that no user could be inserted with a future birthdate by adding a CHECK constraint on the birthday column in the users table:

```
ALTER TABLE "users"
ADD CONSTRAINT check_birthdate
CHECK (birthday < CURRENT_DATE);
```

To avoid duplicate listen records, we applied a UNIQUE constraint on the user_email and song_uri columns in the listens table:

```
ALTER TABLE "listens"
ADD CONSTRAINT unique_user_song_pair
UNIQUE (user_email, song_uri);
```

When we are doing a wrong insertion:

```
INSERT INTO "users" (email, username, gender, birthday, subscription_plan)
VALUES ('futureuser@example.com', 'FutureUser', 'Female', '2050-01-01', 'Free');
```

The error appears and we can keep the database clean and correct:

ERROR: Failing row contains (futureuser@example.com, FutureUser, Female, 2050-01-01, Free).new row for relation "users" violates check constraint "check_birthdate"

Transaction

To finish testing our database, we tried entering new data manually, to do so we utilized **transactions.**
Because a transaction ensures that multiple operations are executed as a single unit and if any operation fails, the database can be rolled back to its previous state, maintaining data consistency during complex insertions.

```
BEGIN TRANSACTION;

INSERT INTO artist (artistUri, name)
VALUES ('spotify:artist:3ALm6zJLaJMWV0r89kuYtu', 'Modà');

INSERT INTO album (albumUri, title, releaseDate, artist_uri)
VALUES ('spotify:album:2euaairB4BCeCBHrpeMqHu', 'Viva i Romantici', '2011-02-16',
'spotify:artist:3ALm6zJLaJMWV0r89kuYtu');

INSERT INTO song (trackUri, title, album_uri)
VALUES
('spotify:track:2KDUheuY5UkgATQvR2K3un', 'Come un pittore',
'spotify:album:2euaairB4BCeCBHrpeMqHu'),
('spotify:track:5sUeQNphyv55ywYlARruNb', 'La notte',
'spotify:album:2euaairB4BCeCBHrpeMqHu'),
('spotify:track:25RA3QmKxNTof4hk3tlnnf', 'Tappeto di fragole',
'spotify:album:2euaairB4BCeCBHrpeMqHu'),
('spotify:track:0HHCJVAwOjEckFKFQ5aqZO', 'Arriverà',
'spotify:album:2euaairB4BCeCBHrpeMqHu'),
('spotify:track:06DSeOsqgLf3ao7mMMB28l', 'Sono già solo',
'spotify:album:2euaairB4BCeCBHrpeMqHu');

COMMIT;
```

This process ensures that all the insertions are successful and consistent. If something goes wrong, the database will not be affected by partial changes.