

# **Database Project 24/25**

## **First Submission**

### **Conceptual Modeling**

#### **L.EIC012 Databases**

**Group 1407:**

Cláudia Mesquita (up202206173)

Pedro Camargo (up202102365)

Pedro Meireles (up202206180)

**20th October, 2024**

# Context Description

The domain of this project is a music streaming application designed to provide users with an expansive library of songs, albums, and playlists, while also allowing personalized user experiences such as playlist creation and membership service. This type of platform mimics popular music services like Spotify or Apple Music.

The goal of the project is to design and implement a service capable of supporting the core functionalities of a music streaming platform. It will manage the vast relationships between users, songs, artists, albums, and playlists while ensuring that the data is efficiently stored.

A user is central to the application and includes information such as username, email, password, birth date, join date, subscription type (free or premium), and country.

Free users have a limited number of skips. Premium users have no such limits and can choose from different subscription types, such as student or family plans. Premium memberships come with a price and a renewal date, and users must pay via either card or PayPal.

If a user pays by card, they need to provide the cardholder's name, expiry date, card number, and security code. If they choose PayPal, a key—typically their email—is required.

Users can add friends, import friends from Facebook, create playlists, follow artists, and add songs to their playlists.

A song includes a name and duration. Each song is part of an album and a genre and may have attached lyrics. Lyrics come with a provider and are specified in a particular language.

A song can feature multiple artists and can be included in user-created playlists. The application also maintains a listening history, which tracks which users listened to which songs and when.

An artist has a name, a debut year, a bio, and a verification status. Artists contribute to songs, either individually or collaboratively, and create albums. They can also perform concerts, which have a date, location (theater, city and country), and ticket price. Additionally, artists can sell their own merchandise through the service, offering products like CDs, vinyl records, and shirts.

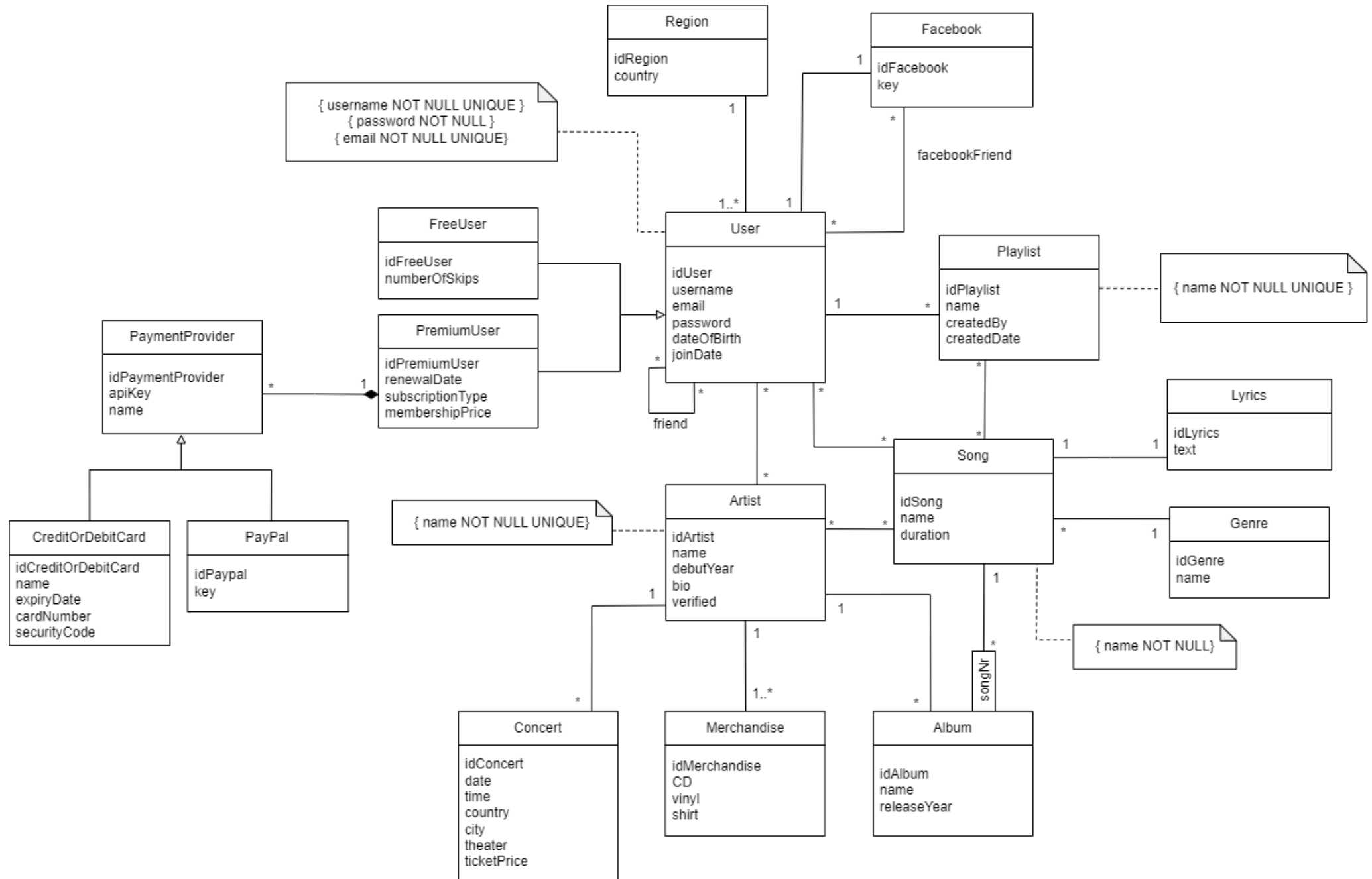
An album includes a name and the year of its release. Each album is created by an artist and contains multiple songs, which are organized with track numbers.

A playlist has a name, the username of the creator, and the date it was created. Users can create custom playlists that consist of multiple songs. Each playlist belongs to a single user and may include songs from various genres, artists, and albums.

A playlist name must be unique, meaning a user cannot have two playlists with the same name.

A genre has a name. Each song belongs to a single genre, but a genre can contain many songs.

# Initial Concept Model



# Generative AI Integration

## ChatGPT

> We provided the AI with a screenshot of our initial UML and asked for feedback and for ways to improve it.

### - **Suggestions for Improvement:**

#### 1. Listening History:

- **Missing Entity:** There is no Listening History entity. Since this is critical for tracking which users listened to which songs (and when), you should add a Listening History entity to track these interactions.
- The Listening History entity could contain attributes like `listen_date`, `listen_time`, and foreign keys `user_id`, `song_id` to create a many-to-many relationship between User and Song.

#### 2. Playlist-Song Relationship:

- Currently, Playlist and Song have a one-to-many relationship, but a playlist usually contains multiple songs, and a song can appear in multiple playlists. This should be a many-to-many relationship, requiring a join table such as `PlaylistSong`.

#### 3. Concert and Merchandise:

- These are interesting additions, but the current relationships between Artist and Concert, Artist and Merchandise are one-to-many. This makes sense for Concert but may need adjustment for Merchandise, especially if multiple artists could collaborate on a single piece of merchandise (such as a concert tour). Consider if there are any many-to-many relationships here.

#### 4. Cardinality Clarification:

- Some relationships are marked with \*, but you should clarify whether these are 1-to-many or many-to-many. For example, the

Song to Album relationship seems to be 1-to-many, but you should ensure this is intentional.

5. Composite Keys and Constraints:

- You can consider adding constraints or composite keys for important entities. For instance, in PaymentProvider, ensure that each CreditOrDebitCard and PayPal account are uniquely associated with a payment provider.

6. Lyrics Entity:

- The Lyrics entity only has a single attribute (text). You could add more detail such as language or author to make it more useful, or you might want to embed this as an attribute within the Song entity instead of having it as a separate table.

- **Other Considerations:**

- Normalization: The model looks fairly normalized already. Ensure that redundancy is avoided (e.g., don't repeat the same data across multiple tables).
- Genre Entity: The Genre entity is fine, but ensure that the relationship with Song is well defined (one song should belong to one genre, but a genre should have multiple songs).
- Facebook Entity: The relationship between User and Facebook should be carefully examined. If it's intended to capture external social media links, it might be more efficient to make this optional or an attribute of the User entity.

# Final Concept Model

## Improvement 1: Listening History

Suggestion accepted, we added the Listening History entity as an association class with the following attributes: *listen\_date* and *listen\_time*.

## Improvement 2: Playlist-Song Relationship

In our initial model, Playlist and Song already have a many-to-many relationship, we believe this is a mistake committed by the AI.

## Improvement 3: Concert and Merchandise

Suggestion declined, since in our relation Artist-Merchandise, a merchandise item belongs to only one artist, that is, it cannot be collaborative.

## Improvement 4: Cardinality Clarification

Suggestion accepted, the relation Song to Album was a one-to-many, but now we updated our model to a many-to-one relation, since one album has many songs.

## Improvement 5: Composite Keys and Constraints

Suggestion declined, we decided not to add constraints or composite keys for PaymentProvider, because the generalization already associates the payment methods to the payment provider.

## Improvement 6: Lyrics Entity

Suggestion accepted, we added two more attributes to the Lyrics entity: *provider* and *language*.

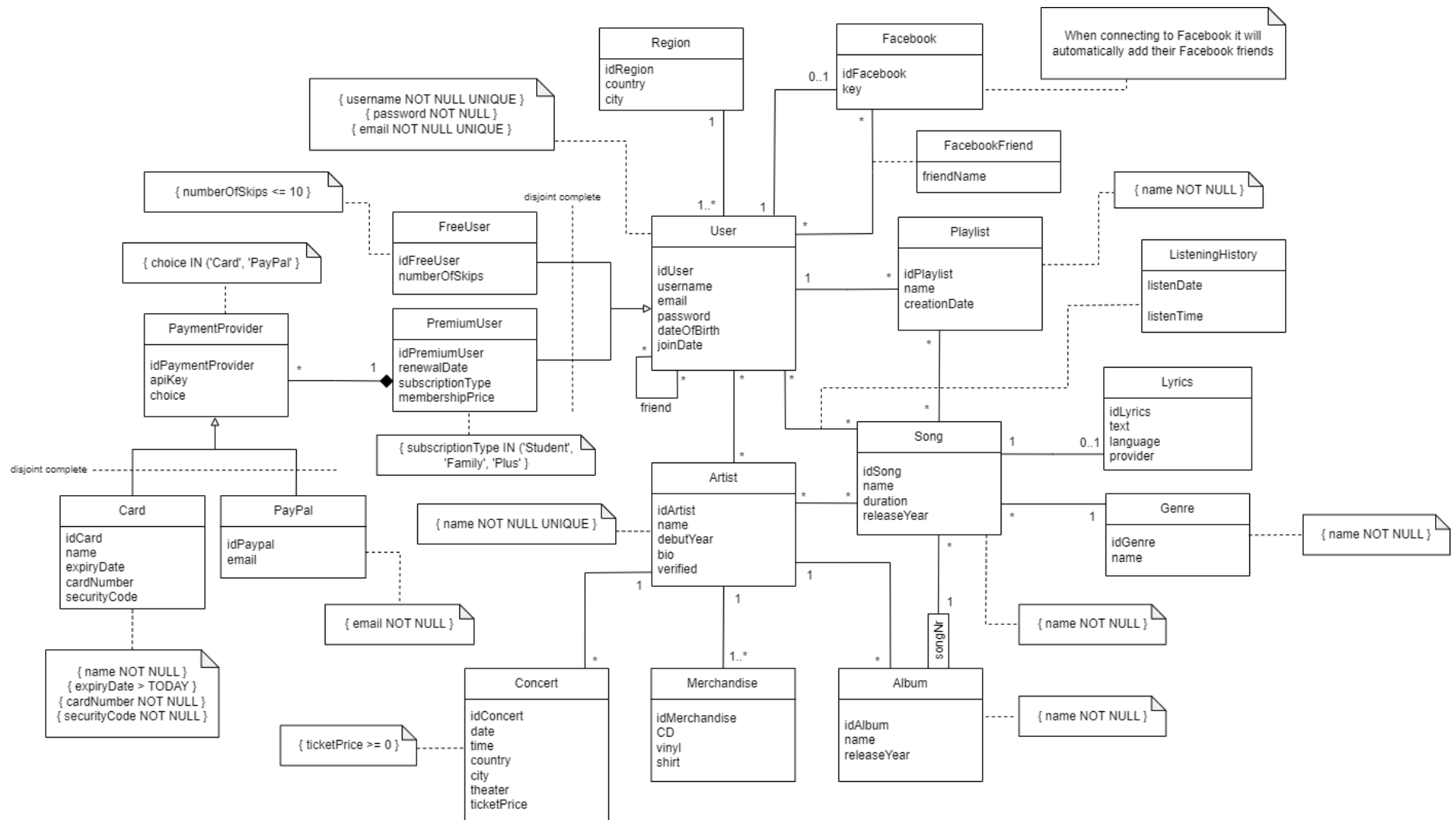




# **Second Submission**

Relational Modelling,  
Database Creation  
and  
Data Population

## Refined Concept Model



# Changes made

Addition of attribute *friendName* in table *FacebookFriend*;  
Addition of the type of generalization in the two present generalizations;  
Changed the name of the *CreditOrDebitCard* table to *Card*;  
Changed the *name* attribute of the *PaymentProvider* table to *choice*;  
Removed the *key* attribute from the *PayPal* table and added the *email* attribute;  
Removed attribute *createdBy* from table *Playlist*;  
Removed attribute *language* from table *Lyrics*;  
Added 9 constraints;  
Added attribute *city* to *Region*.

## Relational Schema

### Initial proposal

album (idAlbum, name, releaseYear, idArtist -> artist)  
artist (idArtist, name, debutYear, bio, verified)  
artistSong (idArtist -> artist, idSong -> song)  
artistUser (idArtist -> artist, idUser -> user)  
concert (idConcert, date, time, country, city, theater, ticketPrice, idArtist -> artist)  
card (idCard -> paymentProvider, name, expiryDate, cardNumber, securityCode)  
facebook (idFacebook, key, idUser -> user)  
facebookFriend (idFacebook -> facebook, idUser -> user, friendName)  
freeUser (idFreeUser -> user, numberOfSkips)  
friend (idFriend1 -> user, idFriend2 -> user)  
genre (idGenre -> genre, name)  
listeningHistory (idUser -> user, idSong -> song, listenDate, listenTime)  
lyrics (idLyrics, text, provider, idSong -> song)  
merchandise (idMerchandise, CD, vinyl, shirt, idArtist -> artist)  
paymentProvider (idPaymentProvider, apiKey, choice, idPremiumUser -> premiumUser)  
paypal (idPaypal -> paymentProvider, email)  
playlist (idPlaylist, name, createdBy, createdDate, idUser -> user)  
playlistSong (idPlaylist -> playlist, idSong -> song)  
premiumUser (idPremiumUser -> user, renewalDate, subscriptionType, membershipPrice)  
region (idRegion, country, city)  
song (idSong, name, duration, idGenre -> genre, idAlbum -> album)

songNumber (idAlbum -> album, idSong -> song, songNr)

user (idUser, username, email, dateOfBirth, joinDate, idRegion -> region)

## Generative AI assistance with ChatGPT

We provided the AI a screenshot of the Refined Concept Model followed by our initial proposal for the Relational Model, and asked the AI to analyze it and point out any errors. We were answered with:

### 1. `card` Table

- **Definition:** `card (idCard -> paymentProvider, name, expiryDate, cardNumber, securityCode)`
- **Issue:**
  - There should be a constraint ensuring `expiryDate` is **in the future**, as specified in your UML.
  - `cardNumber` must be unique to prevent duplicates.
- **Recommendation:**
  - Add constraints: `expiryDate > CURRENT_DATE` and `UNIQUE(cardNumber)`.

Attribute *expiryDate* already has this constraint, so we will not change. For the attribute *cardNumber* we accept the suggestion, so now *cardNumber* is NOT NULL and UNIQUE.

### 2. `facebookFriend` Table

- **Definition:** `facebookFriend (idFacebook -> facebook, idUser -> user, friendName)`
- **Issue:**
  - The UML specifies that friendships are automatically added when connecting to Facebook. This implies the table should only store references (`idFacebook` and `idUser`), and `friendName` might not be needed as it can be retrieved from the `user` or `facebook` table.
- **Recommendation:**
  - Remove `friendName` if redundant or ensure it matches the user's Facebook friend's `username` for consistency.

We accepted the recommendation and removed *friendName* attribute. This attribute was added just above on the Refined Context Model, but it is now removed.

### 3. `playlist` Table

- **Definition:** `playlist (idPlaylist, name, createdBy, createdAt, idUser -> user)`
- **Issue:**
  - The UML specifies that playlist names must be **unique per user**. This constraint is missing in your schema.
- **Recommendation:**
  - Add a unique constraint: `UNIQUE(name, idUser)`.

We accepted the recommendation, since our Context Description says: “A playlist name must be unique, meaning a user cannot have two playlists with the same name.”

### 4. `listeningHistory` Table

- **Definition:** `listeningHistory (idUser -> user, idSong -> song, listenDate, listenTime)`
- **Issue:**
  - There is no primary key or unique constraint to ensure each `idUser`, `idSong`, and `listenDate` combination is distinct.
- **Recommendation:**
  - Add a composite primary key: `PRIMARY KEY (idUser, idSong, listenDate, listenTime)`.

We accepted the recommendation.

### 5. `freeUser` and `premiumUser` Tables

- **Definition:**
  - `freeUser (idFreeUser -> user, numberOfSkips)`
  - `premiumUser (idPremiumUser -> user, renewalDate, subscriptionType, membershipPrice)`
- **Issue:**
  - Ensure that `idFreeUser` and `idPremiumUser` match the `idUser` in the `user` table (since these are subtypes).
- **Recommendation:**
  - Use foreign keys referencing `user(idUser)` for `idFreeUser` and `idPremiumUser`.

The foreign keys are already referencing `user(idUser)`.

#### 6. `region` Table

- **Definition:** `region (idRegion -> country)`
- **Issue:**
  - `country` might not need to be referenced in the schema if it's not a separate entity but just an attribute.
- **Recommendation:**
  - If `country` is simply a name, rename it as `name` and make it a direct attribute.

We decided to keep the attribute name as *country*.

#### 7. `songNumber` Table

- **Definition:** `songNumber (idAlbum -> album, idSong -> song, songNr)`
- **Issue:**
  - The `songNr` attribute should likely be unique per album to avoid duplicate track numbers in the same album.
- **Recommendation:**
  - Add a unique constraint: `UNIQUE(idAlbum, songNr)`.

We accepted the recommendation.

#### 8. `paypal` Table

- **Definition:** `paypal (idPaypal -> paymentProvider, email)`
- **Issue:**
  - `email` should likely be unique to avoid duplicate PayPal accounts in the system.
- **Recommendation:**
  - Add `UNIQUE(email)`.

We accepted the recommendation.

## 9. General Primary Keys and Foreign Keys

- Ensure every table has a defined **primary key**.
- Verify that all foreign keys are properly defined, e.g.:
  - `idArtist` -> `artist` in the `album`, `artistSong`, and `merchandise` tables.
  - `idUser` -> `user` in the `playlist`, `facebookFriend`, and `artistUser` tables.
  - `idSong` -> `song` in the `lyrics`, `listeningHistory`, and `playlistSong` tables.

All these foreign keys are already properly defined.

## Final proposal

All the changes that we accepted will not reflect on the final proposal for the relational schema, so our schema will remain the same. The changes can be seen on SQLite Database Creation.

`album` (`idAlbum`, `name`, `releaseYear`, `idArtist` -> `artist`)  
`artist` (`idArtist`, `name`, `debutYear`, `bio`, `verified`)  
`artistSong` (`idArtist` -> `artist`, `idSong` -> `song`)  
`artistUser` (`idArtist` -> `artist`, `idUser` -> `user`)  
`concert` (`idConcert`, `date`, `time`, `country`, `city`, `theater`, `ticketPrice`, `idArtist` -> `artist`)  
`card` (`idCard` -> `paymentProvider`, `name`, `expiryDate`, `cardNumber`, `securityCode`)  
`facebook` (`idFacebook`, `key`, `idUser` -> `user`)  
`facebookFriend` (`idFacebook` -> `facebook`, `idUser` -> `user`, `friendName`)  
`freeUser` (`idFreeUser` -> `user`, `numberOfSkips`)  
`friend` (`idFriend1` -> `user`, `idFriend2` -> `user`)  
`genre` (`idGenre` -> `genre`, `name`)  
`listeningHistory` (`idUser` -> `user`, `idSong` -> `song`, `listenDate`, `listenTime`)  
`lyrics` (`idLyrics`, `text`, `provider`, `idSong` -> `song`)  
`merchandise` (`idMerchandise`, `CD`, `vinyl`, `shirt`, `idArtist` -> `artist`)  
`paymentProvider` (`idPaymentProvider`, `apiKey`, `choice`, `idPremiumUser` -> `premiumUser`)  
`paypal` (`idPaypal` -> `paymentProvider`, `email`)  
`playlist` (`idPlaylist`, `name`, `createdBy`, `createdDate`, `idUser` -> `user`)  
`playlistSong` (`idPlaylist` -> `playlist`, `idSong` -> `song`)  
`premiumUser` (`idPremiumUser` -> `user`, `renewalDate`, `subscriptionType`, `membershipPrice`)  
`region` (`idRegion`, `country`, `city`)  
`song` (`idSong`, `name`, `duration`, `idGenre` -> `genre`, `idAlbum` -> `album`)  
`songNumber` (`idAlbum` -> `album`, `idSong` -> `song`, `songNr`)

user (idUser, username, email, dateOfBirth, joinDate, idRegion -> region)

## Initial Functional Dependencies

idUser-> username, email, dateOfBirth, joinDate

idPlaylist -> name, createdBy, createdDate

idSong -> name, duration

idUser, idSong -> listenDate, listenTime

idGenre -> name

idAlbum -> name, releaseYear

idAlbum, idSong -> songNr

idLyrics -> text, provider, language

idFacebook -> key

idArtist -> name, debutYear, bio, verified

idMerchandise -> CD, vinyl, shirt

idConcert -> date, time, country, city, theater, ticketPrice

idFreeUser -> numberOfSkips

idPremiumUser -> renewalDate, subscriptionType, membershipPrice

idRegion -> country, city

idPaymentProvider -> apiKey, choice

idCard -> name, expiryDate, cardNumber, securityCode

idPaypal -> email

## Initial Normal Forms Analysis

Our relational model complies with both Boyce-Codd Normal Form (BCNF) and Third Normal Form (3NF).

We know that the Relational Schema adheres to Boyce-Codd Normal Form because:

- Each relation has a unique and exclusive Primary Key (PK), and the Functional Dependencies ensure that all attributes not part of the Primary Key are fully dependent on it.
- The use of composite primary keys, when necessary, ensures the uniqueness of each relational record, avoiding duplication of information. This design reduces redundancy and preserves data integrity.

We know that the Relational Schema adheres to Third Normal Form because:

- All relations eliminate transitive dependencies, ensuring that attributes that are not Primary Keys do not depend on other attributes that are also not Primary Keys.



- The decomposition of tables, such as the example of the **FreeUser** and **PremiumUser** entities, demonstrates logical separation to remove transitive dependencies. This ensures that each table has a specific and well-defined responsibility.
- The correct use of Foreign Keys guarantees referential integrity between tables, avoiding issues such as update or deletion anomalies, thereby strengthening data consistency.

This relational design reflects a robust, efficient, and well-normalized model aligned with best practices in database modeling.

## Generative AI assistance with ChatGPT

### Functional Dependencies

We provided the AI a screenshot of the Refined Concept Model followed by our initial proposal for the Functional Dependencies, and asked the AI to analyze it and point out any errors. We were answered with:

#### Errors/Improvements Identified:

1. `idPlaylist -> createdBy` : The diagram does not indicate a `createdBy` attribute for playlists.
2. `idAlbum, idSong -> songNr` : `songNr` is not shown in the diagram.
3. `idLyrics -> language` : `language` is missing from the diagram.
4. `idMerchandise -> CD, vinyl, shirt` : Clarify the representation of these attributes in the `Merchandise` entity.

1. `createdBy` is in fact a relation and not an attribute, so we accepted this suggestion.
2. `songNr` is the qualifier of the association between album and song, and we have come to the conclusion, according to other sources, that this is the correct way to represent it, so we rejected this suggestion.
3. `language` was an attribute that we wanted to add to the diagram but ultimately decided not to do so, suggestion accepted.
4. We did not deem it necessary to clarify the representation of these attributes, so this suggestion was also rejected.

## Normal Form Analysis

We provided the AI a screenshot of the Refined Concept Model followed by our initial proposal for the Normal Form Analysis, and asked the AI to analyze it and point out any errors. We were answered with:

### Errors or Clarifications to Address

#### 1. Transitive Dependency in PaymentProvider?

- The `PaymentProvider` entity includes `choice` (`Card` or `PayPal`) as an attribute. However, it also has a disjoint relationship with `Card` and `PayPal`, suggesting the attributes of `Card` (e.g., `name`, `expiryDate`) or `PayPal` (`email`) may transitively depend on `choice`.

**Resolution:** Ensure the `PaymentProvider` table itself does not store attributes belonging to its subtypes.

#### 2. Missing `songNr` in Diagram:

- `idAlbum, idSong -> songNr` implies `songNr` exists, but this is missing in the diagram. Without this, the functional dependency cannot be validated, potentially violating BCNF.

#### 3. Clarify Composite Keys:

- While you claim that composite keys ensure uniqueness, it is not explicitly clear in the diagram how this is applied to relationships like `ListeningHistory`. Ensure all composite keys are explicitly defined and enforced.

#### 4. Foreign Key Dependencies:

- Ensure all foreign keys are correctly implemented. For instance:
  - `idFacebook` is optional (`0..1` cardinality), so referential integrity might not be fully enforced.
  - The connection between `idAlbum` and `idSong` in the context of `songNr` should be explicitly validated.

1. We checked and ensured that `PaymentProvider` did not in fact store attributes belonging to its subtypes.
2. As we mentioned before, `songNr` is the qualifier of the association between album and song, and we have come to the conclusion, according to other sources, that this is the correct way to represent it, so we rejected this suggestion.
3. We decided to add examples when talking about the Composite Keys in our analysis, suggestion accepted.
4. We decided to add examples when talking about the Foreign Keys in our analysis, suggestion accepted.

# Final Proposal Functional Dependencies

idUser-> username, email, dateOfBirth, joinDate

idPlaylist -> name, createdDate

idSong -> name, duration

idUser, idSong -> listenDate, listenTime

idGenre -> name

idAlbum -> name, releaseYear

idAlbum, idSong -> songNr

idLyrics -> text, provider

idFacebook -> key

idArtist -> name, debutYear, bio, verified

idMerchandise -> CD, vinyl, shirt

idConcert -> date, time, country, city, theater, ticketPrice

idFreeUser -> numberOfSkips

idPremiumUser -> renewalDate, subscriptionType, membershipPrice

idRegion -> country, city

idPaymentProvider -> apiKey, choice

idCard -> name, expiryDate, cardNumber, securityCode

idPaypal -> email

## Final Proposal Normal Form Analysis

Our relational model complies with both Boyce-Codd Normal Form (BCNF) and Third Normal Form (3NF).

We know that the Relational Schema adheres to Boyce-Codd Normal Form because:

- Each relation has a unique and exclusive Primary Key (PK), and the Functional Dependencies ensure that all attributes not part of the Primary Key are fully dependent on it.
- The use of composite primary keys, , such as (idUser, idSong) in ListeningHistory and (idAlbum, idSong) in AlbumSong, ensures the uniqueness of each relational record, avoiding duplication of information. This design reduces redundancy and preserves data integrity.

We know that the Relational Schema adheres to Third Normal Form because:

- All relations eliminate transitive dependencies, ensuring that attributes that are not Primary Keys do not depend on other attributes that are also not Primary Keys.

- The decomposition of tables, such as the example of the **FreeUser** and **PremiumUser** entities, demonstrates logical separation to remove transitive dependencies. This ensures that each table has a specific and well-defined responsibility.
- The correct use of Foreign Keys, for example, `idUser` in `Playlist` references `User(idUser)`, ensures that a playlist cannot exist without an associated user and guarantees referential integrity between tables, avoiding issues such as update or deletion anomalies, thereby strengthening data consistency.

This relational design reflects a robust, efficient, and well-normalized model aligned with best practices in database modeling.

## SQLite Database Creation

For the creation of the database we didn't use any AI tools. For simplicity we will include the file next to the pdf.

We also added some name constraints to the User Table to show it could be done and every CHECK is also named so it's easier to be traced.

As for `create2.sql`, we asked the AI previously mentioned with this prompt ([“analyze this sqlite3 and I dare you to make it better without adding any unnecessary stuff”](#)) for feedback and this was the response that was given by them:

### “Analysis of Schema

This schema is a comprehensive design for a music application with features like user management, playlists, songs, artists, concerts, and payment systems. Here's a breakdown of issues and optimizations:

#### 1. Redundant Tables

- **FacebookFriend and UserFriend**: Both track friendships. Consider consolidating into one **Friend** table.
- **FreeUser and PremiumUser**: Separate tables for user types can lead to duplication. Use a **userType** column in the **User** table instead.
- **Card and Paypal**: These could be unified under a **PaymentMethod** table with specific details stored in an auxiliary table.

#### 2. Foreign Key Enforcement

- Foreign keys are temporarily disabled during schema setup (`PRAGMA foreign_keys = OFF`). Ensure they are enabled (`PRAGMA foreign_keys = ON`) during production.

### 3. Schema Design Observations

**Normalization:** The schema is mostly normalized but could benefit from:

- Simplifying relationships where redundant entities exist.

**Indexes:** Ensure indexes on frequently joined columns (`FOREIGN KEY` columns).

**Constraints:** Good use of `CHECK` constraints to enforce domain rules (e.g., `userType`, `subscriptionType`, `ticketPrice > 0`).

### 4. Optimizations

**Merge and Simplify:**

- Consolidate `FreeUser` and `PremiumUser` into the `User` table using the `userType` column.

- Create a unified `Friend` table for all friendships.

- Combine `Card` and `Paypal` into a `PaymentMethod` table.

**Performance:**

- Add indexes on columns like `idUser`, `idSong`, `idArtist` for faster querying.

**Relationships:**

- Avoid ambiguous relations (e.g., `PlaylistSong` could benefit from a clearer table name like `PlaylistItems`).

For the populate we gave some examples and asked the AI to create a populate. After that we fixed the entries by making them make sense with each other. For the second population, we asked them to fill with more entries.”

## Breakdown:

1. Redundant tables:
  - a. FacebookFriends and Friends might as well be a single table but in the event there's a unlinking it would result in faster queries if we kept them distinct.
  - b. FreeUser and PremiumUser are good as it is separate since we would break Normal Form if we made it separate, also we have added a check and the userType argument to ensure no duplication. And as a byproduct we would also be able to allow faster swaps between user types.
  - c. This would make it harder to add new types of payments and would also break normal form
2. Foreign Key Enforcement:
  - a. It's already done at the end
3. Schema Design Observations
  - a. Indexes are not covered in this UC.

After reading the feedback from the AI we decided to enforce the security code to always have 3 numbers and those changes are on create2.sql.

```
CONSTRAINT valid_securitycode CHECK (securityCode > 99 AND securityCode < 1000)
```

## Data Loading

The populate1.sql was written with data without real word correlation as an example, even though we tried to make it credible. After populating the database and checking everything we realized it felt a bit short and so we decided to improve on that on the next step.

## Generative AI integration

As mentioned before, to solve the lackster populate, we feed populate1.sql with the following prompt ("[analyze this populate.sql and I dare you to make every table 5x bigger, keep in mind we have checks for userType \('Free','Premium'\) subscriptiontype \('Student','Family','Plus'\) choice\('Card','Paypal'\) and expiry date \(> always make them 2026\)](#)").

The previous AI decided to instead of generating new values, to just write them out 5 times which led to some issues with our unique constraint, and since we currently don't have a subscription to it we resorted to another one called Perplexity using the GPT-4.0 model.

This was their [response](#):

To make every table in the provided `populate.sql` file five times larger while maintaining the specified constraints, you can follow these steps. This

involves duplicating existing entries with slight modifications to ensure uniqueness and compliance with the constraints such as `userType`, `subscriptionType`, `choice`, and expiry dates.

Steps to Expand Each Table:

... (For the sake of simplicity I won't make this longer)

That wasn't at all what we expected but then we just asked 3 tables at a time and added them to our script. As expected there were some syntax errors and some constraints that failed. After fixing them, (and also some random “”) we verified that `populate2.sql` was doing what was expected.