



# Serviço de Streaming de Música

Base de dados

Licenciatura em Engenharia Informática e Computação

Ana Belém Meireles Bilhoto - up202305035

Maria Luiza Henriques Vieira - up202304306

João Pedro Pinto Lunet - up202207150

# Índice:

1. Contexto.....	2
2. Modelo conceitual (UML).....	3
2.1. Diagrama inicial.....	3
2.2. Integração de inteligência artificial.....	4
2.3. Diagrama final.....	8
3. Avaliação do trabalho individual dos participantes (primeira submissão).....	9
4. Relational Schema.....	10
4.1. Esquema relacional inicial.....	10
4.2. Integração de inteligência artificial.....	11
4.3. Esquema relacional final.....	20
5. Dependências Funcionais.....	21
5.1. Formas Normais.....	23
5.1.1. Terceira Forma Normal.....	23
5.1.2. Forma Normal de Boyce-Codd.....	24
5.2. Integração de Inteligência Artificial.....	25
6. Criação da Base de Dados em SQLite.....	32
6.1. Integração de Inteligência Artificial.....	32
7. Carregamento de Dados.....	40
7.1. Integração de Inteligência Artificial.....	40
8. Avaliação Final do Trabalho Individual dos Participantes.....	42
9. Avaliação Final do Trabalho Individual dos Participantes.....	43

# 1. Contexto

A plataforma que desenhamos é um serviço de streaming de música com funcionalidades semelhantes ao Spotify, que oferece aos utilizadores acesso a uma vasta biblioteca de conteúdos áudio e a oportunidade de interagir com outros utilizadores de diversas formas, tanto uns com os outros quanto com os criadores e artistas. Aquando a criação da conta, o utilizador insere o modo de contacto, ou seja, e-mail ou número de telefone, e escolhe o plano de assinatura.

Entre as principais funcionalidades, os utilizadores podem seguir perfis de outros utilizadores e artistas, criar e partilhar playlists personalizadas, adicionar músicas a playlists e guardar playlists criadas por outros. Além disso, a plataforma permite que os utilizadores guardem nos seus favoritos não apenas músicas e álbuns, mas também episódios de podcasts.

Outra funcionalidade importante é o histórico de audição, que regista o conteúdo que o utilizador ouviu até ao momento, para além de guardar o progresso em episódios, permitindo assim um fácil acesso a músicas, álbuns e podcasts previamente ouvidos.

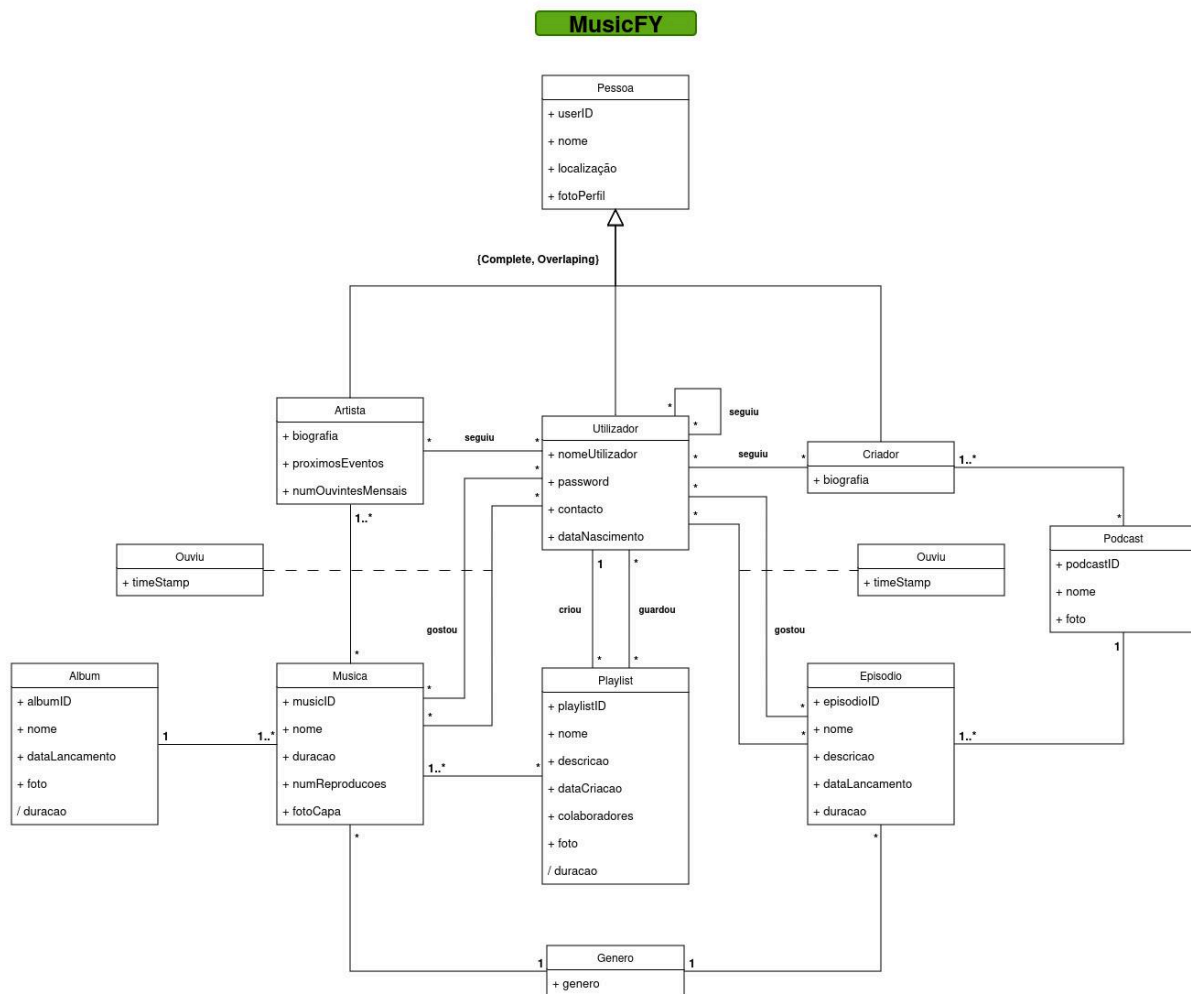
A partir de todos esses dados, como preferências musicais, interações com playlists e histórico de audição, a plataforma aplica algoritmos avançados para recomendar conteúdos que sejam do interesse do utilizador. Desta forma, é possível oferecer uma experiência personalizada, sugerindo músicas, álbuns, artistas e podcasts de acordo com o gosto individual de cada utilizador.

Além de melhorar a experiência de recomendação, essas informações também são utilizadas internamente para gerar estatísticas que permitem à plataforma refinar constantemente os seus serviços, ajustando funcionalidades e melhorando a interação entre utilizadores e artistas.

O objetivo é desenvolver uma base de dados que torne a implementação desta plataforma possível de forma eficaz.

## 2. Modelo conceptual (UML)

### 2.1. Diagrama inicial



## 2.2. Integração de inteligência artificial

Para esta tarefa optamos por usar o ChatGPT, pois este utiliza o modelo IA mais avançado (GPT-4o mini). Foi enviado o seguinte prompt com uma imagem do nosso diagrama UML inicial anexada. Fomos pedindo ao modelo para refinar cada vez mais o UML, repetindo o mesmo processo várias vezes, e compilamos as respostas.

### Prompt:

Olá! Em anexo está o UML para uma plataforma de streaming de música, parecida com o Spotify. Gostaríamos de saber que melhorias podemos fazer ao UML.

### Resposta:

Olá! Eu vi o diagrama UML que enviaste para a plataforma de streaming de música "MusicFY", que está bastante bem estruturado. No entanto, há algumas sugestões de melhoria que podem ajudar a refinar o modelo e torná-lo mais robusto:

#### 1. Relacionamento entre "Música" e "Artista"

- O relacionamento entre a classe "Música" e "Artista" está como 1..\* (um artista pode ter várias músicas, e uma música pode ter vários artistas). Isso faz sentido para colaborações entre artistas. Contudo, pode ser útil especificar se o artista é principal ou colaborador numa música. Podes adicionar um atributo extra ou uma nova associação para diferenciar.

**Avaliação Crítica:** Não acrescentamos essa sugestão, visto que na plataforma não há a diferenciação entre o artista principal e o artista colaborador. A música pertence igualmente a discografia de ambos.

#### 2. Nova Entidade "Ouvintes"

- **NumOuvintesMensais em Artista:** Se o campo "numOuvintesMensais" em "Artista" for atualizado frequentemente, pode ser mais eficiente mover isso para uma entidade separada que armazena essas estatísticas.

**Avaliação Crítica:** Optamos por seguir esta sugestão e criar uma classe *Ouvintes* com o atributo "numOuvintesMensais", uma vez que este atributo seria atualizado sempre que um novo utilizador ouvisse uma música do artista em questão, tornando assim a base de dados mais eficiente.

### 3. Detalhamento de "Ouvinte" e "Ouvinte de Podcasts"

- Atualmente, tanto a "Música" como o "Podcast" têm uma associação com "Ouviu" para registrar o histórico de reproduções. No entanto, o diagrama parece tratar episódios e músicas da mesma forma. Talvez fosse interessante separar melhor esses dois tipos de conteúdo, adicionando, por exemplo, um tipo de métrica diferente para cada um (como contagem de minutos ouvidos para episódios).

**Avaliação crítica:** Seguimos esta sugestão e acrescentamos o atributo "progressoEpisodio" na associação "Ouviu" entre o "Utilizador" e o "Podcast", para que assim o utilizador possa voltar a ouvir a partir do momento em que parou pela última vez.

### 4. Playlists e Colaboradores

- A classe "Playlist" tem um campo de "colaboradores". Uma playlist pode ter vários colaboradores, mas parece que esse relacionamento poderia ser melhor estruturado. Um modelo mais claro poderia ser a criação de uma tabela associativa entre "Utilizador" e "Playlist" para representar essa colaboração de forma mais explícita.

**Avaliação crítica:** Optamos por apenas alterar a multiplicidade da associação "criou" para que uma playlist pudesse ser criada por mais do que 1 utilizador.

### 5. Tipo de Assinatura (Premium/Gratuita)

- A classe "Utilizador" poderia ter um atributo para determinar o tipo de assinatura (gratuita ou premium), que pode alterar as funcionalidades disponíveis para o utilizador. Este campo pode influenciar a quantidade de saltos de música, a qualidade do áudio ou a exibição de anúncios.

**Avaliação crítica:** Adicionamos o atributo "planoAssinatura" à classe "Utilizador" permitindo agora diferenciar utilizadores com plano pago, ou gratuito.

## 6. Histórico de Gostos e Seguimentos

- Existe um relacionamento entre o "Utilizador" e a "Música" (gostou), mas se um utilizador pudesse gostar ou seguir mais conteúdos, como álbuns, **playlists**, podcasts ou criadores, seria útil padronizar um histórico de interações para todos esses tipos de conteúdo. Isto facilitaria a expansão futura do modelo.

**Avaliação crítica:** Optamos por não fazer essa alteração uma vez que os gostos em cada conteúdo são guardados separadamente e de maneiras diferentes, sendo assim desnecessário um histórico de interações.

## 7. Atributos adicionais para "Playlist" e "Episódio"

- As playlists poderiam ter um campo para armazenar a duração total baseada na soma das durações das músicas. Também seria interessante adicionar um campo de visualizações para entender quantas vezes a playlist foi acedida.
- Para "Episódio", seria bom incluir uma relação com tags ou temas, o que ajudaria os utilizadores a encontrar episódios por tópicos de interesse.

**Avaliação crítica:** Procedemos à alteração do nome do atributo "duracao" para "duracaoTotal" de modo a tornar a análise do modelo conceptual mais simples e direta. Substituímos também a associação das classes "*Genero*" e "*Episodio*" pela criação da classe *Temas* que se associa por sua vez à classe "*Episodio*". A associação entre as classes "*Genero*" e "*Podcast*" também foi adicionada, permitindo assim uma maior diferenciação e organização entre Podcasts e os respectivos episódios.

## 8. Melhoria dos Nomes de Atributos e Entidades

- Nomes de atributos mais claros: Alguns nomes de atributos poderiam ser mais descritivos. Por exemplo, em vez de "duracao", poderias usar algo como "duracaoSegundos" para especificar a unidade da duração.

**Avaliação crítica:** Acabamos por seguir esta sugestão, uma vez que torna a interpretação do modelo conceptual mais fácil.

## 9. Relacionamentos

- Múltiplos Géneros: Parece que tanto "Música" quanto "Álbum" têm uma relação de 1 para 1 com "Género". Na prática, uma música ou álbum pode ter múltiplos géneros. Sugiro mudar o relacionamento para muitos para muitos.

**Avaliação crítica:** Neste caso decidimos adotar esta medida, visto que se assemelha mais à realidade do conceito de género musical, em comparação com a multiplicidade implementada anteriormente.

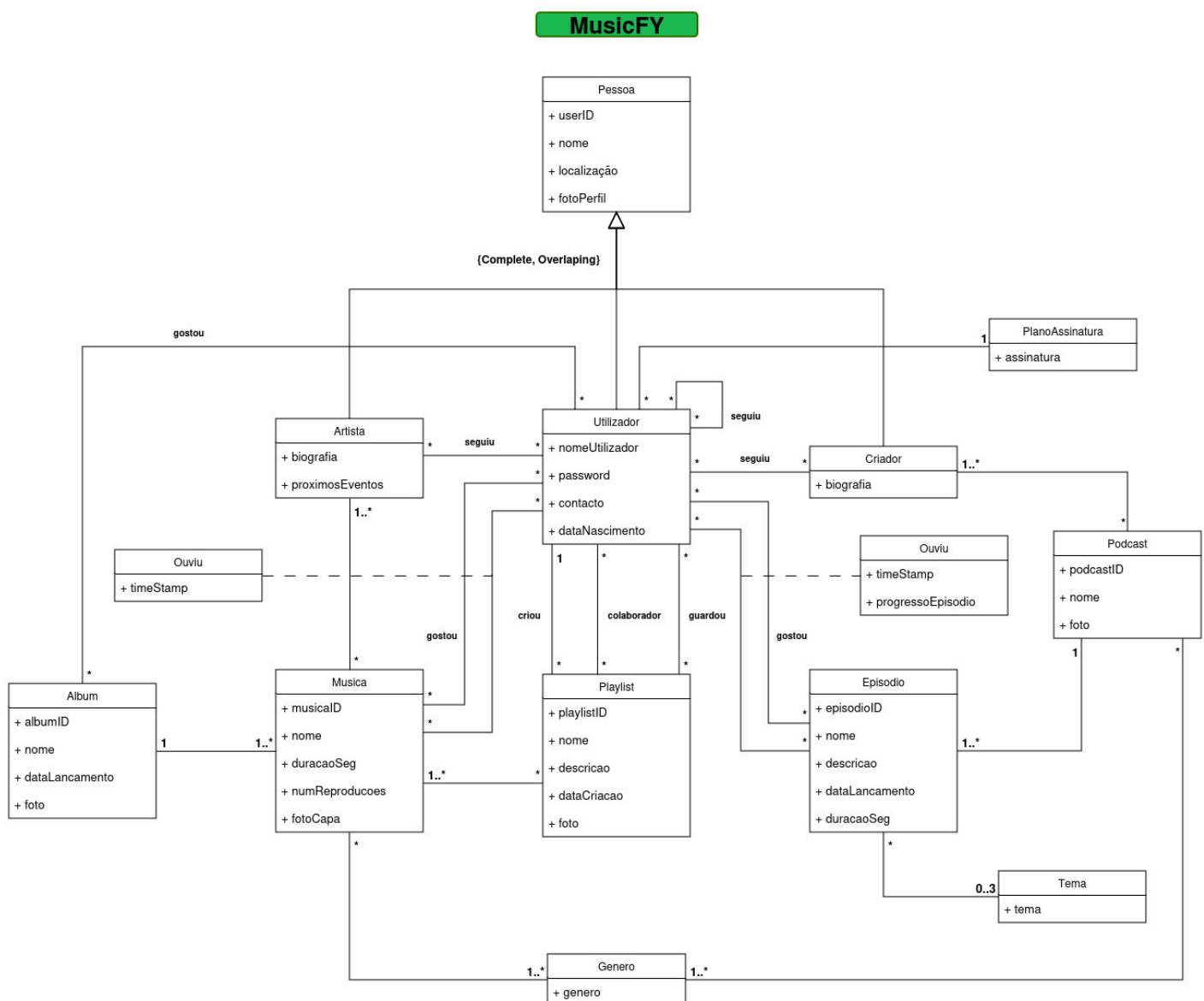
Estas são algumas melhorias que podem refinar o modelo UML, facilitando futuras expansões da plataforma. Se precisares de mais alguma ajuda ou se houver outra área do diagrama que gostarias de discutir mais a fundo, estou à disposição!



### 2.3. Diagrama final

A integração de IA, neste caso com o uso de ChatGPT, foi muito importante para o refinamento do modelo UML, propondo melhorias estruturais e a aplicação de boas práticas de modelagem. A sua integração permitiu acelerar o processo, no entanto, a avaliação crítica humana revelou-se indispensável para definir o que faria ou não sentido no contexto da base de dados.

Após analisar e avaliar as alterações propostas pelo modelo de inteligência artificial escolhido, o nosso modelo conceptual sofreu algumas alterações, sendo esta a nossa proposta final:



### 3. Avaliação do trabalho individual dos participantes (primeira submissão)

Trabalhamos em conjunto durante a maior parte do trabalho e dividimos algumas tarefas a serem feitas individualmente, que após concluídas discutimos entre nós e finalizamos o relatório em grupo.

## 4. Relational Schema

### 4.1. Esquema relacional inicial

Pessoa (userID, nome, localizacao, fotoPerfil)

Artista (artistaID -> Pessoa.userID, biografia, proximosEventos)

Utilizador (userID -> Pessoa.userID, nomeUtilizador, password, contacto, dataNascimento, assinaturaID -> PlanoAssinatura)

PlanoAssinatura (assinaturaID, assinatura)

Criador (criadorID -> Pessoa.userID, biografia)

Playlist (playlistID, nome, descricao, dataCriacao, foto, userID -> Utilizador)

Musica (musicalID, nome, duracaoSeg, numReproducoes, fotoCapa, albumID -> Album)

Album (albumID, nome, dataLancamento, fotoCapa)

Podcast (podcastID, nome, fotoCapa)

Episodio (episodioID, nome, descricao, dataLancamento, duracaoSeg)

Genero (generoID, genero)

Tema (temaID, tema)

-----  
OuviuEpisodio (episodioID -> Episodio, userID -> Utilizador, dataehora, progressoEpisodio )

OuviuMusica (musicalID -> Musica, userID -> Utilizador, dataehora)

-----  
SeguiuUtilizador (SeguidorID -> Utilizador.userID, SeguidoID -> Utilizador.userID)

SeguiuArtista (userID -> Utilizador, artistaID -> Artista)

SeguiuCriador (userID -> Utilizador, criadorID -> Criador)

GostouAlbum (userID -> Utilizador, albumID -> Album)

GostouEpisodio (userID -> Utilizador, episodioID -> Episodio)

GostouMusica (userID -> Utilizador, musicalID -> Musica)

Colaborador (userID -> Utilizador, playlistID -> Playlist)

Guardou (userID -> Utilizador, playlistID -> Playlist)

ArtistaMusica (userID -> Artista, musicalID -> Musica)

MusicaPlaylist (musicalID -> Musica, playlistID -> Playlist)

MusicaGenero (musicalID -> Musica, generoID -> Genero)

PodcastGenero (musicalID -> Musica, podcastID -> Podcast)

PodcastEpisodio ( podcastID -> Podcast, episodioID -> Episodio)

CriadorPodcast (userID -> Criador, podcastID -> Podcast)

EpisodioTema (episodioID -> Episodio, temaID -> Tema)

## 4.2. Integração de inteligência artificial

Para esta tarefa optamos por usar o ChatGPT, pois este utiliza o modelo IA mais avançado (GPT-4o mini). Foi enviado o seguinte prompt com uma imagem do nosso relational schema inicial anexada.

### Prompt:

Estamos a fazer uma base de dados para uma plataforma de streaming de música, parecida ao Spotify. Em anexo encontra-se o relational schema para tal projeto, gostaríamos de saber as suas sugestões de como podemos melhorá-lo.

### Resposta:

O *relational schema* apresentado está bem estruturado e cobre diversos aspectos importantes para uma plataforma de streaming de música. Aqui estão algumas sugestões e considerações para possíveis melhorias e otimizações:

#### 1. Normalização e Redundância

- **Campo `numReproducoes` na tabela `Música`:** Considere se este atributo deve ser mantido na tabela ou calculado dinamicamente com base nos registos em `OuviuMusica`. Guardar este valor na tabela pode introduzir redundância e possíveis inconsistências.
- **Progresso do episódio:** O atributo `progressoEpisodio` pode ser dinâmico e recalculado com base nas interações do utilizador, dependendo de como será utilizado.

#### 2. Chaves Primárias e Estrangeiras

- Certifique-se de que todas as tabelas têm chaves primárias definidas. Embora isso esteja implícito, uma anotação explícita seria útil, especialmente para entidades como `Tema` ou `Genero`.
- Considere o uso de chaves compostas ou tabelas associativas para representar relacionamentos muitos-para-muitos de maneira mais eficiente, como em `MusicaGenero`.

#### 3. Segurança e Privacidade

- **Senha:** O campo `password` deve ser armazenado de forma segura (encriptado/hasheado) e isso pode ser destacado no esquema como uma *constraint* ou uma nota.
- **Histórico de reprodução:** Considere anonimizar os dados relacionados a `OuviuEpisodio` e `OuviuMusica` para proteção da privacidade.

## 4. Otimização para Consultas Frequentes

- **Índices:** Adicione índices em campos que são frequentemente utilizados em pesquisas ou filtros, como `generoID` em *MusicaGenero*, `artistaID` em *ArtistaMusica*, e `userID` em *SeguiuUtilizador*.
- **Caches e Pré-calculados:** Para informações como "álbuns mais populares" ou "artistas mais ouvidos", considere armazenar estes dados em tabelas ou *views* que podem ser atualizadas periodicamente.

## 5. Funcionalidades Adicionais

- **Tags e Playlists Colaborativas:** Adicione uma tabela para tags que possam ser associadas a músicas, álbuns ou playlists, permitindo categorização mais dinâmica.
- **Favoritos:** Adicione uma tabela específica para músicas ou playlists marcadas como favoritas pelo utilizador, caso o *GostouMusica* e *Guardou* não sejam suficientes para representar isso.
- **Dados Analíticos:** Considere tabelas específicas para guardar métricas agregadas (e.g., total de reproduções por utilizador por mês), úteis para relatórios e algoritmos de recomendação.

## 6. Recomendações e Algoritmos

- **Histórico de Pesquisa:** Pode ser útil guardar as pesquisas feitas pelos utilizadores.
- **Relacionamento entre Entidades:** Adicionar relações mais ricas, como "Músicas Recomendadas", que possam ser preenchidas por algoritmos.

Estas sugestões dependerão também do contexto funcional e técnico do projeto, bem como do desempenho esperado. Caso deseje, podemos aprofundar algum destes pontos.

### Prompt:

Por favor explique melhor o ponto 1.

### Resposta:

Claro! Vamos aprofundar o **Ponto 1 - Normalização e Redundância**, que aborda como os dados são armazenados para evitar inconsistências e manter um bom desempenho.

#### 1.1 Campo numReproducoes na tabela Musica

No *relational schema*, o campo `numReproducoes` representa o número de vezes que uma música foi reproduzida. Este campo pode parecer prático, mas ele **introduz redundância** porque já existe a tabela *OuviuMusica*, que registra cada reprodução.

#### Problemas potenciais de redundância:

- **Desatualização dos dados:** Sempre que uma música for reproduzida, o valor de `numReproducoes` precisará ser atualizado na tabela *Musica*. Se houver falhas na atualização (exemplo: uma reprodução registrada em *OuviuMusica* mas não refletida em *Musica*), pode haver inconsistências.
- **Mais complexidade no código:** O sistema terá que garantir que os dois locais (a tabela *Musica* e *OuviuMusica*) sejam mantidos sincronizados.
- **Desempenho reduzido em grande escala:** Em sistemas com muitas reproduções simultâneas, o acesso e atualização constante da tabela *Musica* podem ser mais lentos.

**Solução recomendada:** Em vez de armazenar `numReproducoes` diretamente na tabela *Musica*, você pode calculá-lo dinamicamente sempre que necessário, consultando a tabela *OuviuMusica*. Por exemplo:

```
Unset
SELECT musicaID, COUNT(*) AS numReproducoes
FROM OuviuMusica
GROUP BY musicaID;
```

#### Vantagens desta abordagem:

1. **Sem redundância:** Os dados de reproduções são mantidos somente em *OuviuMusica*, garantindo consistência.
2. **Atualização em tempo real:** Sempre que a tabela *OuviuMusica* é atualizada, o número de reproduções reflete automaticamente a realidade.
3. **Economia de espaço:** Não é necessário armazenar um campo adicional em *Musica*.

**Quando manter `numReproducoes` pode ser útil:** Se o sistema precisar frequentemente deste valor (ex.: para exibir em uma página inicial "as músicas mais tocadas"), pode ser vantajoso manter o campo `numReproducoes` em *Musica* com uma estratégia de cache. Neste caso:

- Atualize este campo periodicamente (por exemplo, de hora em hora ou diariamente), em vez de em tempo real.
- Utilize *triggers* ou scripts agendados para recalcular os valores e evitar inconsistências.

## 1.2 Progresso do Episódio (*progressoEpisodio* na tabela *OuviuEpisodio*)

De forma semelhante, o progresso que um utilizador fez ao ouvir um episódio (*progressoEpisodio*) pode ser um valor calculado ou atualizado de forma dinâmica.

Por exemplo, se você armazena eventos como *pausa* ou *avanço rápido* em uma tabela separada, é possível calcular o progresso de um episódio somando o tempo acumulado de reprodução. No entanto:

- Armazenar *progressoEpisodio* diretamente em *OuviuEpisodio* faz sentido se for um dado acessado com frequência e se não houver mudanças constantes no progresso.

## Resumo

- Remova o campo *numReproducoes* de *Musica* e calcule-o diretamente de *OuviuMusica*, a menos que você precise desse valor com muita frequência, nesse caso use cache.
- O mesmo raciocínio aplica-se ao *progressoEpisodio* em *OuviuEpisodio*, embora neste caso ele seja menos propenso a inconsistências.

Se precisar, posso sugerir *queries* ou estratégias específicas para implementar essa lógica!

## Prompt:

No ponto 4, por que índices seriam úteis e como poderíamos incluí-los?

## Resposta:

### Por que os índices seriam úteis?

Os índices são estruturas de dados que os bancos de dados utilizam para **acelerar a busca e o acesso** a informações em tabelas. Sem um índice, o banco de dados precisa percorrer todas as linhas de uma tabela (*full table scan*) para encontrar os dados solicitados, o que pode ser muito lento em tabelas grandes.

Na sua plataforma de streaming, índices seriam particularmente úteis em colunas que:

1. São frequentemente utilizadas em **consultas** (ex.: buscas, filtros).
2. São **chaves estrangeiras** usadas para relacionar tabelas (ex.: `musicaID` em `OuviruMusica`).
3. Participam em **ordenações ou agrupamentos** (ex.: ordenação por número de reproduções).

---

### Exemplos de casos úteis de índices no seu projeto

#### 1. Pesquisas em tabelas grandes

Se os utilizadores frequentemente buscam músicas por `nome`:

```
Unset
SELECT * FROM Musica WHERE nome = 'Song Name';
```

Um índice na coluna `nome` da tabela `Musica` aceleraria essa busca, pois o banco de dados não precisaria verificar linha por linha.



## 2. Relacionamentos entre tabelas

Em uma consulta para encontrar os utilizadores que ouviram uma determinada música:

```
Unset
SELECT *

FROM OuviuMusica

WHERE musicaID = 12345;
```

Um índice na coluna `musicaID` da tabela `OuviuMusica` permite localizar rapidamente todas as entradas relacionadas a essa música.

## 3. Filtros frequentes em colunas

Se os utilizadores frequentemente filtram músicas por `generoID` (gênero musical) ou álbuns por `dataLancamento`, índices nessas colunas otimizam o desempenho.

## 4. Ordenação e Agrupamento

Se você precisa exibir as músicas mais ouvidas:

```
Unset
SELECT musicaID, COUNT(*) AS total

FROM OuviuMusica

GROUP BY musicaID

ORDER BY total DESC

LIMIT 10;
```

Um índice em `musicaID` acelera a agregação e a ordenação.

---

## Como incluir índices no banco de dados

Os índices podem ser adicionados no momento da criação da tabela ou posteriormente. Aqui estão exemplos para cada caso:

### 1. Adicionar índices ao criar a tabela

Ao definir uma tabela, use a palavra-chave `INDEX` para criar índices.

---

```

Unset
CREATE TABLE Musica (

    musicaID SERIAL PRIMARY KEY,

    nome VARCHAR(255) NOT NULL,

    albumID INT NOT NULL,

    generoID INT NOT NULL,

    numReproducoes INT DEFAULT 0,

    INDEX idx_nome (nome),          -- Índice para pesquisas pelo nome

    INDEX idx_albumID (albumID),    -- Índice para chave estrangeira

    INDEX idx_generoID (generoID)   -- Índice para filtro por gênero

);

```

## 2. Adicionar índices em tabelas existentes

Se a tabela já existe, crie índices com `CREATE INDEX`:

```

Unset
-- Índice simples

CREATE INDEX idx_nome ON Musica(nome);

-- Índice em chave estrangeira

CREATE INDEX idx_albumID ON Musica(albumID);

-- Índice para múltiplas colunas (índice composto)

CREATE INDEX idx_nome_genero ON Musica(nome, generoID);

```

### 3. Índices únicos

Se uma coluna (ou conjunto de colunas) deve ter valores únicos, use `UNIQUE`:

```
Unset
CREATE UNIQUE INDEX idx_user_nome ON Utilizador(nomeUtilizador);
```

Isso também pode ser útil para garantir que não existam nomes de utilizadores duplicados.

### 4. Índices para buscas textuais (Full-Text Index)

Se a plataforma suportar busca por texto (ex.: busca por parte do nome da música ou letras), use índices de texto completo:

- **PostgreSQL:**

```
Unset
CREATE INDEX idx_fulltext_nome ON Musica USING gin(to_tsvector('portuguese',
nome));
```

- **MySQL:**

```
Unset
CREATE FULLTEXT INDEX idx_fulltext_nome ON Musica(nome);
```

---

### Cuidados ao usar índices

Embora os índices melhorem a performance de consultas, eles têm algumas **desvantagens**:

1. **Armazenamento adicional:** Índices ocupam espaço no disco.
2. **Impacto em operações de escrita:** Inserções, atualizações e exclusões em tabelas com muitos índices podem ser mais lentas, pois os índices precisam ser atualizados.

## Boas práticas:

- Crie índices apenas nas colunas mais utilizadas para buscas, filtros ou ordenações.
- Evite criar índices desnecessários ou redundantes.
- Use ferramentas como o **EXPLAIN** (ou **EXPLAIN ANALYZE**) para identificar quais consultas precisam de otimização.

## Resumo dos índices sugeridos para o seu esquema

### 1. Colunas de Pesquisa:

- Índice em `nome` (música, playlist, utilizador, etc.).
- Índice em `generoID` (para filtrar músicas por gênero).
- Índice em `dataLancamento` (álbuns, episódios, etc.).

### 2. Chaves Estrangeiras:

- Índice em `musicaID` (em *OuviuMusica*, *GostouMusica*, etc.).
- Índice em `playlistID` (em *MusicaPlaylist*, *Guardou*).

### 3. Consultas Agregadas:

- Índice em `musicaID` (em *OuviuMusica* para contagens rápidas).
- Índice em `userID` (em *SeguiuUtilizador*, *OuviuMusica*).

### 4. Full-Text Index:

- Se o sistema suportar buscas textuais, adicione índices de texto completo para colunas como `nome` (em *Musica*, *Playlist*, etc.).

Se precisar de exemplos adicionais ou ajuda para implementar, é só perguntar!

## Avaliação Crítica:

Desta vez, a integração de IA não foi tão útil, uma vez que algumas das sugestões obtidas já estão implementadas no nosso esquema relacional, como no ponto 2, onde foram referidas chaves primárias e estrangeiras. Além disso, também houveram sugestões que achamos desnecessárias para o nosso projeto, como a Otimização para Consultas Frequentes, Recomendações e Algoritmos e as Funcionalidades Adicionais. No entanto, implementamos uma das sugestões dadas pela IA, nesse caso a referente ao atributo `numReproducoes`, já que este pode ser facilmente obtido através de queries que contam o número de tuplos associados a uma determinada música.

## 4.3. Esquema relacional final

Pessoa (userID, nome, localizacao, fotoPerfil)

Artista (artistaID -> Pessoa.userID, biografia, proximosEventos)

Utilizador (userID -> Pessoa.userID, nomeUtilizador, password, contacto, dataNascimento, assinaturaID -> PlanoAssinatura)

PlanoAssinatura (assinaturaID, assinatura)

Criador (criadorID -> Pessoa.userID, biografia)

Playlist (playlistID, nome, descricao, dataCriacao, foto, userID -> Utilizador)

Musica (musicalID, nome, duracaoSeg, fotoCapa, albumID -> Album)

Album (albumID, nome, dataLancamento, fotoCapa)

Podcast (podcastID, nome, fotoCapa)

Episodio (episodioID, nome, descricao, dataLancamento, duracaoSeg)

Genero (generoID, genero)

Tema (temaID, tema)

---

OuviuEpisodio (episodioID -> Episodio, userID -> Utilizador, dataehora, progressoEpisodio)

OuviuMusica (musicalID -> Musica, userID -> Utilizador, dataehora)

---

SeguiuUtilizador (SeguidorID -> Utilizador.userID, SeguidoID -> Utilizador.userID)

SeguiuArtista (userID -> Utilizador, artistaID -> Artista)

SeguiuCriador (userID -> Utilizador, criadorID -> Criador)

GostouAlbum (userID -> Utilizador, albumID -> Album)

GostouEpisodio (userID -> Utilizador, episodioID -> Episodio)

GostouMusica (userID -> Utilizador, musicalID -> Musica)

Colaborador (userID -> Utilizador, playlistID -> Playlist)

Guardou (userID -> Utilizador, playlistID -> Playlist)

ArtistaMusica (userID -> Artista, musicalID -> Musica)

MusicaPlaylist (musicalID -> Musica, playlistID -> Playlist)

MusicaGenero (musicalID -> Musica, generoID -> Genero)

PodcastGenero (musicalID -> Musica, podcastID -> Podcast)

PodcastEpisodio (podcastID -> Podcast, episodioID -> Episodio)

CriadorPodcast (userID -> Criador, podcastID -> Podcast)

EpisodioTema (episodioID -> Episodio, temaID -> Tema)

## 5. Dependências Funcionais

- Pessoa (userID, nome, localizacao, fotoPerfil):  
userID → nome, localizacao, fotoPerfil
- Artista (artistaID → Pessoa.userID, biografia, proximosEventos):  
artistaID → biografia, proximosEventos
- Utilizador (userID → Pessoa.userID, nomeUtilizador, password, contacto, dataNascimento, assinaturalID → PlanoAssinatura):  
userID → nomeUtilizador, password, dataNascimento, contacto, assinaturalID  
nomeUtilizador → userID, password, contacto, dataNascimento, assinaturalID  
contacto → userID, nomeUtilizador, password, dataNascimento, assinaturalID
- PlanoAssinatura (assinaturalID, assinatura):  
assinaturalID → assinatura
- Criador (criadorID → Pessoa.userID, biografia):  
criadorID → biografia
- Playlist (playlistID, nome, descricao, dataCriacao, foto, userID → Utilizador):  
playlistID → nome, descricao, dataCriacao, foto, userID
- Musica (musicalID, nome, duracaoSeg, fotoCapa, albumID → Album):  
musicalID → nome, duracaoSeg, fotoCapa, albumID
- Album (albumID, nome, dataLancamento, fotoCapa):  
albumID → nome, dataLançamento, fotoCapa
- Podcast (podcastID, nome, fotoCapa):  
podcastID → nome, fotoCapa
- Episodio (episodioid, nome, descricao, dataLancamento, duracaoSeg):  
episodioid → nome, descricao, dataLançamento, duracaoSeg
- Genero (generoid, genero):  
generoid → genero
- Tema (temaid, tema):  
temaid → tema
- OuviuEpisodio (episodioid → Episodio, userID → Utilizador, dataehora, progressoEpisodio):

episodioID, userID -> dataehora, progressoEpisodio

- OuviuMusica (musicalID -> Musica, userID -> Utilizador, dataehora):  
musicalID, userID -> dataehora

Qualquer relação que seja apenas constituída por uma chave composta não vai apresentar dependências funcionais não triviais como tal já está normalizada, algumas relações deste tipo são:

- SeguiuUtilizador (userID1 -> Utilizador, userID2 -> Utilizador)
- SeguiuArtista (userID -> Utilizador, artistaID -> Artista)
- SeguiuCriador (userID -> Utilizador, criadorID -> Criador)
- GostouAlbum (userID -> Utilizador, albumID -> Album)
- GostouEpisodio (userID -> Utilizador, episodioID -> Episodio)
- GostouMusica (userID -> Utilizador, musicalID -> Musica)
- Colaborador (userID -> Utilizador, playlistID -> Playlist)
- Guardou (userID -> Utilizador, playlistID -> Playlist)
- ArtistaMusica (userID -> Artista, musicalID -> Musica)
- MusicaPlaylist (musicalID -> Musica, playlistID -> Playlist)
- MusicaGenero (musicalID -> Musica, generoID -> Genero)
- PodcastGenero (musicalID -> Musica, podcastID -> Podcast)
- PodcastEpisodio (podcastID -> Podcast, episodioID -> Episodio)
- CriadorPodcast (userID -> Criador, podcastID -> Podcast)
- EpisodioTema (episodioID -> Episodio, temaID -> Tema)

Relações que possuam apenas dois atributos já se encontram na forma normal de Boyce-Codd, visto que devido ao número limitado de elementos a condição de uma superchave determinar os restantes elementos já está satisfeita, entre estas relações encontram-se:

- Genero (generoID, genero)
- Tema (temaID, tema)
- PlanoAssinatura (assinaturaID, assinatura)

## 5.1. Formas Normais

### 5.1.1. Terceira Forma Normal

Para um esquema estar na terceira forma normal precisa primeiro de estar na segunda forma normal e de não existirem atributos não-chave que dependam por transitividade da chave-primária.

Através da análise das dependências funcionais identificadas por nós, verificamos que não existe nenhum esquema que viole a terceira forma normal.

Vamos agora analisar um caso em específico que precisa de uma explicação mais aprofundada, esse é:

Utilizador (userID -> Pessoa.userID, nomeUtilizador, password, contacto, dataNascimento, assinaturaID -> PlanoAssinatura):

- userID -> nomeUtilizador, password, dataNascimento, contacto, assinatura
- nomeUtilizador -> userID, password, contacto, dataNascimento, assinaturaID
- contacto -> userID, nomeUtilizador, password, contacto, dataNascimento, assinaturaID

Como podemos ver contacto e nomeUtilizador são atributos não-chave, no entanto são chaves candidatas. Este facto impede que a relação apresentada viole a terceira forma normal, já que a formalização da definição afirma que todos os atributos não-chave devem ser funcionalmente dependentes das chaves candidatas e não de uma parte da chave ou de um atributo não-chave.



## 5.1.2. Forma Normal de Boyce-Codd

Um esquema está na forma normal de Boyce-Codd se estiver na terceira forma normal e todas as dependências funcionais forem do tipo  $A \rightarrow B$ , sendo A uma superchave.

Anteriormente analisamos a terceira forma normal e verificamos, que não ocorria qualquer infração desta forma. Porém, o mesmo não pode ser dito para a forma normal de Boyce-Codd, visto que há uma relação que quebra com a regra definida.

Utilizador (userID  $\rightarrow$  Pessoa.userID, nomeUtilizador, password, contacto, dataNascimento, assinaturaID  $\rightarrow$  PlanoAssinatura):

- userID  $\rightarrow$  nomeUtilizador, password, dataNascimento, contacto, assinaturaID
- contacto  $\rightarrow$  userID, nomeUtilizador, password, dataNascimento, assinaturaID
- nomeUtilizador  $\rightarrow$  userID, password, dataNascimento, contacto, assinaturaID

Esta relação não cumpre com a forma de Boyce-Codd pois apesar de contacto e nomeUtilizador serem chaves candidatas, estas dependem funcionalmente do userID e vice-versa. Desta forma a relação precisa de ser subdividida em três relações:

- Utilizador (userID  $\rightarrow$  Pessoa.userID, nomeUtilizador, password, contactoID  $\rightarrow$  contacto, dataNascimento, assinaturaID  $\rightarrow$  PlanoAssinatura)
- contacto (contactoID, contacto)
- nomeUtilizador (nomeUtilizadorID, nomeUtilizador)

Assim as dependências funcionais das relações seriam:

- userID  $\rightarrow$  nomeUtilizadorID, password, dataNascimento, contactoID, assinaturaID
- contactoID  $\rightarrow$  contacto
- nomeUtilizadorID  $\rightarrow$  nomeUtilizador

Desta forma, todas as relações estão todas na forma normal de Boyce-Codd, já que os atributos não-chave dependem agora apenas de uma superchave, como dita a regra.

## 5.2. Integração de Inteligência Artificial

Mais uma vez, utilizamos o ChatGPT para auxiliar na melhoria ou não do nosso trabalho.

### Prompt:

Estamos a fazer um projeto que consiste numa base de dados para uma plataforma de streaming de música, semelhante ao Spotify. Escrevemos as dependências funcionais a partir do esquema relacional e analisamos o cumprimento da Terceira Forma Normal e da Forma Normal de Boyce-Codd, como envio em seguida. Gostaríamos de saber as possíveis melhorias que podemos fazer além de uma análise das formas normais.

[texto das Dependências Funcionais e Formas Normais]

### Resposta:

O projeto de modelação de dados para a plataforma de streaming de música está muito bem estruturado. O esforço em analisar as formas normais e identificar pontos de melhoria, como a divisão da relação `Utilizador`, demonstra uma atenção cuidada aos princípios da normalização.

Aqui ficam algumas considerações e sugestões para possíveis melhorias para além da análise das formas normais:

---

### 1. Redundância e Eficiência das Consultas

Embora a normalização reduza redundâncias e assegure consistência, pode aumentar o custo de algumas consultas devido ao número de junções necessárias. Avalie os seguintes pontos:

- **Desnormalização controlada:** Para consultas frequentes, como a exibição de playlists ou detalhes de músicas, poderá ser vantajoso adicionar colunas derivadas diretamente no esquema para evitar muitas junções.
- **Índices:** Identifique os atributos mais consultados e considere a criação de índices para melhorar o desempenho das consultas.

---

### 2. Normalização versus Flexibilidade

A estrutura atual, altamente normalizada, é ideal para evitar inconsistências. Contudo:

- Certifique-se de que as operações de manutenção (inserções, atualizações) sejam práticas e não sobrecarregadas pelo número elevado de relações interligadas.

- Considere **caching** para dados frequentemente acedidos, como listas de reprodução ou artistas populares, para melhorar a experiência do utilizador.

### 3. Controlo da Integridade Referencial

Já especificaram várias chaves estrangeiras, o que é excelente para assegurar a integridade dos dados. Além disso:

- **Regras ON DELETE/ON UPDATE:** Especifique o comportamento para remoção ou atualização de registos em tabelas como **Artista**, **Utilizador** e **Playlist**. Por exemplo, ao remover um artista, as músicas associadas devem ser removidas? Ou mantidas como histórico?
  - Utilize restrições para garantir valores consistentes, como formatos de emails ou contactos no atributo **contacto**.
- 

### 4. Versatilidade na Representação de Dados

Plataformas como o Spotify lidam com dados dinâmicos, como versões de músicas, remixes ou co-autorias. Para maior flexibilidade:

- Considere adicionar uma tabela para associar múltiplos artistas a um álbum ou música (**ColaboracaoArtistaMusica**), além da já existente **ArtistaMusica**.
  - Permita a atribuição de múltiplos géneros a músicas e episódios, alargando as tabelas **MusicaGenero** e **PodcastGenero** para incluir géneros principais e secundários.
- 

### 5. Gestão de Histórico

Plataformas de streaming frequentemente registam históricos detalhados dos utilizadores:

- **Histórico de Alterações:** Para assinaturas, playlists ou preferências, inclua tabelas de histórico para acompanhar mudanças ao longo do tempo.
  - **Interações do Utilizador:** Considere uma tabela **Interacoes** para capturar ações como gostos, partilhas ou reproduções específicas.
-

## 6. Segurança e Privacidade

Dado que se trata de dados sensíveis (como `contacto`, `password`):

- **Hashing de Senhas:** Assegure-se de armazenar as senhas em formato hash seguro (e.g., `bcrypt` ou `Argon2`).
- **Encriptação de Dados Sensíveis:** Como `contacto` e `dataNascimento`, se necessário para conformidade com leis de privacidade.

## 7. Suporte à Internacionalização

Se a plataforma tiver alcance global, inclua suporte para:

- Múltiplos idiomas em atributos como `biografia`, `descricao` e `genero`.
  - Formatos de data/hora ajustados aos fusos horários dos utilizadores.
- 

## Conclusão

O vosso projeto já atende aos principais requisitos de normalização. As sugestões acima complementam o esquema relacional com foco em escalabilidade, desempenho e adaptabilidade. Avaliem as necessidades previstas para a plataforma para encontrar um equilíbrio entre normalização, eficiência e flexibilidade.

### Prompt:

Aqui vai um esquema relacional, consegues identificar as dependências funcionais e verificar se a relação se encontra na forma normal de Boyce-codd.

[Esquema relacional final]

### Resposta:

Para analisar se o seu esquema relacional está na **Forma Normal de Boyce-Codd (BCNF)**, precisamos identificar as dependências funcionais de cada relação. Vou examinar algumas das tabelas mencionadas e suas dependências:

### 1. Artista

- **Atributos:** `artistaID`, `Pessoa.userID`, `biografia`, `proximosEventos`
  - **Dependência funcional:**
    - `artistaID → Pessoa.userID, biografia, proximosEventos`
  - A relação **Artista** está em BCNF, porque `artistaID` é uma superchave que determina todos os outros atributos.
-

## 2. Utilizador

- **Atributos:** `userID`, `Pessoa.userID`, `nomeUtilizador`, `password`, `contacto`, `dataNascimento`, `assinaturaID`
- **Dependência funcional:**
  - `userID` → `Pessoa.userID`, `nomeUtilizador`, `password`, `contacto`, `dataNascimento`, `assinaturaID`
  - `nomeUtilizador` → `userID`, `password`, `contacto`, `dataNascimento`, `assinaturaID`
  - `contacto` → `userID`, `nomeUtilizador`, `password`, `dataNascimento`, `assinaturaID`
- Esta relação **não está em BCNF**. A dependência `nomeUtilizador` → `userID`, `password`, `contacto`, `dataNascimento`, `assinaturaID` indica que `nomeUtilizador` pode determinar os outros atributos, mas não é uma superchave, violando as regras da BCNF.

## 3. PlanoAssinatura

- **Atributos:** `assinaturaID`, `assinatura`
- **Dependência funcional:**
  - `assinaturaID` → `assinatura`
- Esta relação está em **BCNF**, porque `assinaturaID` é uma superchave que determina o único outro atributo.

## 4. Playlist

- **Atributos:** `playlistID`, `nome`, `descricao`, `dataCriacao`, `foto`, `userID`
- **Dependência funcional:**
  - `playlistID` → `nome`, `descricao`, `dataCriacao`, `foto`, `userID`
- Está em **BCNF**, pois `playlistID` determina todos os outros atributos.

## 5. Musica

- **Atributos:** `musicaID`, `nome`, `duracaoSeg`, `fotoCapa`, `albumID`
- **Dependência funcional:**
  - `musicaID` → `nome`, `duracaoSeg`, `fotoCapa`, `albumID`
- Está em **BCNF**, pois `musicaID` é uma superchave que determina os outros atributos.

## 6. SeguiuUtilizador

- **Atributos:** `SeguidorID`, `SeguidoID`
- **Dependência funcional:**
  - `SeguidorID` → `SeguidoID`
- Esta tabela está em **BCNF**, pois `SeguidorID` é uma superchave.

## 7. MusicaGenero

- **Atributos:** `musicaID`, `generoID`
- **Dependência funcional:**
  - `musicaID, generoID →` (dependendo da implementação, poderia ser atributos extras)
- A dependência está em **BCNF** porque a combinação das chaves primárias determina os outros atributos.

## Conclusão:

Algumas tabelas como **Utilizador** não estão em **BCNF** devido a dependências funcionais não triviais (como `nomeUtilizador` determinando `userID`), que violam a condição de que a superchave deve determinar todos os outros atributos. Para normalizar essas tabelas, você poderia separar as dependências em tabelas diferentes, como uma tabela **UtilizadorNome** com `nomeUtilizador` e seus atributos, e uma tabela **Utilizador** para os atributos restantes.

**Prompt:**

Esta foi a nossa análise do esquema, concordas ou acrescentarias algo mais?  
[texto na íntegra das secções]

**Resposta:**

Sua análise do esquema relacional e das dependências funcionais está muito bem estruturada, e você fez um bom trabalho identificando as possíveis violações da forma normal de Boyce-Codd (BCNF). Concordo com a maior parte da análise e dos exemplos fornecidos.

Aqui estão algumas observações e sugestões para complementar sua análise:

**1.Chaves candidatas em Utilizador:**

Você mencionou que `contacto` e `nomeUtilizador` são chaves candidatas, mas isso implica que, ao serem usadas como determinantes, violam a BCNF, pois não são superchaves. A reestruturação que você propôs para dividir a relação `Utilizador` em três tabelas é uma boa solução para resolver essa violação.

**2.PlanoAssinatura:**

A relação `PlanoAssinatura` está na BCNF corretamente, já que `assinaturaID` é uma chave primária que determina a única outra coluna `assinatura`.

**3.Relações com chaves compostas:**

As relações que possuem apenas uma chave composta, como `SeguiuUtilizador`, `GostouAlbum`, etc., estão corretamente identificadas como normalizadas, pois não há dependências funcionais não triviais que possam violar as formas normais.

**4.Normalização em BCNF:**

Sua análise mostra que a maior parte das tabelas está bem estruturada, mas a relação `Utilizador` é o principal exemplo de violação da BCNF, o que você corretamente corrigiu ao dividir em três tabelas.

Em resumo, sua análise está detalhada e a solução para normalizar o esquema está bem fundamentada. A subdivisão da tabela `Utilizador` resolve eficazmente o problema da BCNF.

**Avaliação Crítica**

Inicialmente, a AI não foi muito útil já que não abordou o tema que realmente queríamos. Deste modo, decidimos alterar a estruturação dos prompts para obter uma resposta, a qual elucidar-nos-ia se algo nos houvesse escapado ou pudesse ser melhorado. Contudo, como é

possível constatar as respostas acabaram por ir de encontro com o nosso trabalho e nenhuma sugestão foi feita.



## 6. Criação da Base de Dados em SQLite

### 6.1.Integração de Inteligência Artificial

Novamente, utilizamos o ChatGPT uma vez que acreditamos ser a IA mais avançada.

#### **Prompt:**

Envio em anexo o SQL para a base de dados de uma plataforma de streaming de música. Que melhorias podemos fazer?

[create.sql]

#### **Reposta:**

Vou analisar o ficheiro SQL que enviaste e já te dou um feedback sobre possíveis melhorias na base de dados. Um momento, por favor.

Já observei a estrutura inicial das tabelas. Vou analisar a totalidade do esquema e já te indico sugestões de melhorias. Um momento, por favor.

#### **Sugestões de melhorias para a base de dados:**

##### 1. Nomes de Colunas e Consistência:

- Algumas tabelas, como `Utilizador`, têm uma coluna `nomeUtilizador` definida como `INT`, o que pode estar incorreto. Deve ser `VARCHAR` em vez de `INT`.
- Considere padronizar o uso de `ID` ou `id` nas colunas para consistência (`artistaID`, `playlistID`, etc.).

##### 2. Normalização e Redundância:

- Em tabelas como `OuviuMusica`, `OuviuEpisodio`, etc., há um uso excessivo de `UNIQUE` em campos de relação. Se cada registro representar uma ocorrência, o `UNIQUE` pode restringir a inserção de múltiplos eventos.
- Reduza duplicação de informações, como a repetição de comandos `ON DELETE CASCADE` em cada linha de colunas.

##### 3. Tipos de Dados:

- Colunas como `contacto` devem ser ajustadas para `VARCHAR` em vez de `INT`, especialmente se representarem números de telefone, que podem começar com zero.
- Considere o uso de tipos específicos, como `BOOLEAN`, onde aplicável, para representar estados (por exemplo, numa coluna de status).

##### 4. Relações e Chaves Estrangeiras:

- Em tabelas com múltiplos `FOREIGN KEY`, certifique-se de que todas as referências são necessárias e bem implementadas. A tabela

`PlanoAssinatura` é referenciada em `Utilizador`, mas o tratamento `ON DELETE SET NULL` deve estar alinhado com a lógica do negócio.

#### 5. Índices e Desempenho:

- Para tabelas com muitas relações (por exemplo, `MusicaPlaylist`, `ArtistaMusica`), considere criar índices para otimizar as consultas.
- Utilize índices compostos onde fizer sentido, especialmente em tabelas de associação.

#### 6. Validação de Dados:

- A restrição `CHECK` na coluna `palavrapasse` é útil, mas pode ser expandida para incluir requisitos de segurança adicionais, como a presença de caracteres especiais.

#### 7. Nomes das Tabelas e Relações:

- O nome `CridorPodcast` parece conter um erro de digitação e deve ser corrigido para `CriadorPodcast`.
- Verifique nomes para evitar ambiguidades, como a tabela `Contacto`, que pode gerar confusão se estiver relacionada a outras entidades.

Queres mais ajuda em alguma área específica?

### Prompt:

Sobre as ações ON CASCADE e ON DELETE, acha que estão bem implementadas ou deveriam ser outras?

### Resposta:

As ações `ON DELETE` e `ON UPDATE` desempenham um papel crucial na manutenção da integridade referencial entre as tabelas. Analisando o teu esquema, vou destacar alguns pontos que podem necessitar de ajustes ou melhorias:

---

#### 1. Uso excessivo de ON DELETE CASCADE:

- Em tabelas como `OuviuMusica`, `OuviuEpisodio`, e `SeguiuArtista`, a ação `ON DELETE CASCADE` está aplicada a muitas colunas. Isto significa que, ao eliminar um utilizador ou entidade relacionada, todos os registos associados serão apagados automaticamente.
  - **Risco:** Esta abordagem pode levar à perda massiva de dados, especialmente se uma entidade central, como um utilizador, for eliminada. Pode ser mais prudente manter esses dados e apenas desassociar a referência.
  - **Sugestão:** Em vez de `ON DELETE CASCADE`, considera utilizar `ON DELETE SET NULL` ou `ON DELETE RESTRICT` em algumas relações para evitar a eliminação em cascata de dados críticos.

#### 2. Relações com ON DELETE SET NULL:

- Na tabela `Utilizador`, a coluna `assinaturaID` utiliza `ON DELETE SET NULL`, o que significa que, ao eliminar um plano de assinatura, o campo será definido como `NULL`.
    - **Adequado para:** Se quiseses manter os dados do utilizador mesmo após a eliminação de um plano, isto faz sentido.
    - **Sugestão:** Certifica-te de que a lógica da aplicação pode lidar com utilizadores sem plano (`assinaturaID NULL`). Pode ser necessário criar uma política para tratar estes utilizadores (por exemplo, atribuir um plano padrão ou restringir o acesso).
-

### 3. ON UPDATE CASCADE em múltiplas colunas:

- Esta ação está frequentemente aplicada a tabelas de junção (`MusicaPlaylist`, `ArtistaMusica`, etc.). Isto significa que, se o identificador de uma entidade for alterado, todas as referências serão atualizadas automaticamente.
    - **Adequado para:** Garantir consistência em caso de alterações de chaves primárias.
    - **Sugestão:** Avalia se as chaves primárias realmente precisam de ser atualizadas com frequência. Se não, podes evitar `ON UPDATE CASCADE` para reduzir complexidade e riscos.
- 

### 4. Tabelas de junção (Ex: `GostouMusica`, `GuardouPlaylist`):

- Atualmente, todas têm `ON DELETE CASCADE` em ambas as chaves estrangeiras.
    - **Recomendação:** Em vez de eliminar registros automaticamente ao apagar utilizadores ou músicas, podes considerar:
      - `ON DELETE RESTRICT`: Evita a eliminação de músicas ou utilizadores que têm dependências.
      - `ON DELETE SET NULL` (menos comum em tabelas de junção): Se quiseres manter um histórico.
- 

### 5. Importância da lógica de negócio:

- A decisão depende também da lógica específica da plataforma. Pergunta-te:
  - Se um artista for eliminado, queres realmente apagar todas as músicas associadas?
  - Se um utilizador apagar a conta, deverás eliminar playlists ou músicas favoritas?
  - Se um podcast for apagado, os episódios devem ser mantidos?

#### Resumo das recomendações:

1. Reduzir o uso de `ON DELETE CASCADE` nas tabelas de entidades principais para evitar perda massiva de dados.
2. Utilizar `ON DELETE RESTRICT` em tabelas de associação, quando a eliminação de dados deve ser controlada.
3. Rever a necessidade de `ON UPDATE CASCADE` em chaves primárias que raramente mudam.

Se me deres mais detalhes sobre a lógica de negócio, posso oferecer sugestões mais personalizadas!

### Prompt:

Existe alguma outra mudança que sugere que façamos?

### Resposta:

Sim, há mais algumas mudanças que podem melhorar a robustez e a eficiência da base de dados. Vou sugerir algumas alterações adicionais:

---

## 7. Melhorar a Estrutura de Tabelas de Associação:

- **Tabelas como `MusicaPlaylist` e `ArtistaMusica`:** Estas são tabelas de associação entre entidades. Certifica-te de que cada combinação (`musicaID`, `playlistID`, etc.) tem uma restrição `UNIQUE` para evitar duplicações.

Unset

```
ALTER TABLE MusicaPlaylist ADD CONSTRAINT unique_musica_playlist UNIQUE (musicaID, playlistID);
```

---

## 8. Chaves Primárias Compostas:

- Para tabelas de relacionamento (`GostouMusica`, `SeguiuArtista`), podes utilizar chaves primárias compostas em vez de uma chave primária individual.

Unset

```
ALTER TABLE GostouMusica DROP PRIMARY KEY;  
ALTER TABLE GostouMusica ADD PRIMARY KEY (utilizadorID, musicaID);
```

## 9. Auditoria e Histórico:

- Adicionar tabelas de auditoria para acompanhar alterações importantes. Por exemplo, uma tabela `HistoricoUtilizador` pode guardar informações sobre mudanças de estado (ex.: criação, desativação de conta).

Unset

```
CREATE TABLE HistoricoUtilizador (  
    historicoID INT AUTO_INCREMENT PRIMARY KEY,  
    utilizadorID INT,  
    acao VARCHAR(50),  
    data TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (utilizadorID) REFERENCES Utilizador(utilizadorID)  
);
```

---

## 10. Controlo de Dados Nulos:

- Evita permitir valores `NULL` em colunas críticas, como `email` ou `nomeUtilizador`, se ainda não estiverem restritas.

Unset

```
ALTER TABLE Utilizador MODIFY email VARCHAR(100) NOT NULL;
```

---

## 11. Uso de Colunas ENUM ou Tabelas de Referência:

- Se há colunas que armazenam estados (ex.: `tipoAssinatura`, `estadoConta`), considera usar `ENUM` ou criar tabelas de referência para melhorar a integridade.

Unset

```
ALTER TABLE Utilizador MODIFY estadoConta ENUM('ativo', 'inativo',  
'suspensao') DEFAULT 'ativo';
```

Ou criar uma tabela separada:

Unset

```
CREATE TABLE EstadoConta (  
    estadoID INT PRIMARY KEY,  
    descricao VARCHAR(20) UNIQUE
```

---

```
);
```

## 12. Automatização com TRIGGERS:

- Utiliza **TRIGGERS** para automatizar tarefas, como atualizar a data de modificação ou manter um histórico de alterações.

```
Unset
CREATE TRIGGER before_update_utilizador
BEFORE UPDATE ON Utilizador
FOR EACH ROW
SET NEW.ultimaAtualizacao = NOW();
```

## 13. Validação de Dados para Evitar Dados Inconsistentes:

- Implementa mais **CHECK** constraints para garantir que valores lógicos são respeitados, por exemplo:

```
Unset
ALTER TABLE Musica ADD CONSTRAINT CHECK (duracao > 0);
```

## Resumo das Sugestões Adicionais:

1. Adicionar restrições **UNIQUE** em tabelas de associação para evitar duplicações.
2. Utilizar chaves primárias compostas em tabelas de relacionamento.
3. Criar tabelas de auditoria para acompanhar mudanças importantes.
4. Controlar valores nulos em colunas críticas.
5. Implementar **ENUM** ou tabelas de referência para dados categóricos.
6. Utilizar **TRIGGERS** para automatizar auditorias e ações repetitivas.
7. Aplicar mais validações com **CHECK constraints**.

Estas mudanças vão fortalecer a integridade, segurança e flexibilidade da base de dados. Se precisares de ajuda para aplicar alguma delas ou ajustar conforme a lógica do projeto, estou aqui para ajudar!

## **Avaliação Crítica**

A integração com a inteligência artificial teve algumas utilidades desta vez, como acentuar o problema relacionado a assinatura e o SET NULL, além de corrigir erros ortográficos. No entanto, houveram algumas mudanças sugeridas que não faziam muito sentido, como passar o contacto para VARCHAR, sendo que ele se refere ao ID, ou seja, um INT. Também não concordamos com as sugestões referentes ao ON DELETE CASCADE que, na nossa opinião, está bem implementado. Por último, algumas ideias apresentadas pelo ChatGPT pensamos que são desnecessárias no âmbito do nosso projeto.



# 7. Carregamento de Dados

## 7.1. Integração de Inteligência Artificial

### Prompt:

Olá, estou a desenvolver uma base de dados para um serviço de streaming de música e podcasts. Podes avaliar a população que criei para esta base de dados, e dar-me ideias de como a melhorar por favor? Deixo em anexo o ficheiro .sql com a população.

### Resposta:

O ficheiro contém comandos SQL para popular tabelas relacionadas com um serviço de streaming, incluindo planos de assinatura, temas, géneros musicais e dados de pessoas que parecem incluir artistas, criadores e utilizadores. Com base na população inicial, aqui estão alguns pontos para analisar e melhorar a base de dados:

### Observações Iniciais

- Estruturação das Tabelas:**
  - As tabelas como `PlanoAssinatura`, `Tema`, e `Genero` estão bem organizadas com inserções simples.
  - A tabela `Pessoa` parece suportar múltiplos tipos de usuários (artistas, criadores, utilizadores), mas é importante garantir que os papéis estão claramente definidos.
- Dados Diversificados:**
  - A base cobre temas e géneros amplos, o que é bom. No entanto, pode ser útil incluir mais opções e detalhes adicionais (ex., subgéneros musicais).
- Chaves Primárias e Estrangeiras:**
  - Algumas tabelas têm referências claras, mas será importante garantir que os relacionamentos entre entidades (ex., artistas e suas músicas ou criadores e seus podcasts) estejam bem definidos.
- Detalhes Ausentes:**
  - Alguns campos, como `fotoPerfil`, têm valores `NULL`. Pode ser útil adicionar dados fictícios (ex.: URLs de imagens placeholder).
  - Nenhuma tabela ou relação explícita entre músicas, podcasts e criadores foi observada até agora.

## Sugestões de Melhoria

1. **Tipos de Usuários:**
  - Adicionar uma tabela separada para papéis (ex., **Artista**, **Criador**, **Usuário**) e usar uma relação para mapear os papéis para os usuários na tabela **Pessoa**. Isso facilita a gestão de novos tipos de utilizadores no futuro.
2. **Relacionamentos Avançados:**
  - Criar tabelas para **Música** e **Podcast** associando-as a criadores/artistas.
  - Associar temas a podcasts e géneros a músicas para categorizações dinâmicas.
3. **Campos Adicionais:**
  - Para **Música** e **Podcast**:
    - **Duração**
    - **Data de Lançamento**
    - **Popularidade** ou **Número de Reproduções**.
  - Para **Pessoa**:
    - **Descrição** (para criadores e artistas).
    - **Data de Registo** (para todos os usuários).
4. **Dados Relacionados a Planos de Assinatura:**
  - Especificar limites e características de cada plano (ex., **Premium** permite downloads offline, **Grátis** inclui anúncios).
5. **Dados Geográficos e Estatísticas:**
  - Adicionar informações como país e idioma para personalizar a experiência do usuário.
6. **Exemplo de População para Relacionamentos:**
  - Relacionar criadores aos seus podcasts e artistas às suas músicas para criar um cenário funcional.
  - Incluir playlists, que podem ser criadas por usuários ou geradas automaticamente.

Queres que implemente algumas destas sugestões diretamente no SQL fornecido?

## Avaliação Crítica:

Não achamos úteis as sugestões dadas pelo IA neste ponto, uma vez que acreditamos que não acrescentaria nada relevante ao nosso trabalho.

## 8. Avaliação da Integração de Inteligência Artificial

De maneira geral, a integração de inteligência artificial foi útil, uma vez que nos ajudou a compreender melhor o trabalho e recebemos sugestões de melhorias válidas que nós implementamos. Porém, também deu algumas sugestões já previamente implementadas e outras que consideramos desnecessárias no âmbito do nosso projeto.

## 9. Avaliação Final do Trabalho Individual dos Participantes

As tarefas da segunda submissão foram igualmente divididas, ficando cada um responsável por uma parte. No final, nos reunimos para concluir o projeto em conjunto de maneira que todos estivéssemos de acordo com o trabalho realizado.