A criação de um algoritmo de recomendação de músicas baseado em grafos, como o que você descreveu, é um excelente caso de uso para bancos de dados de grafos como o **Neo4j** e sua linguagem de consulta, o **Cypher**.

O modelo proposto (nós para usuários, músicas, artistas, gêneros e arestas para interações) se encaixa perfeitamente na estrutura de grafos.

🎶 Modelagem do Grafo (Graph Schema)

O primeiro passo é definir os tipos de **Nós (Nodes)** e **Relacionamentos (Relationships)**, incluindo suas propriedades:

Entidade (Nó - Label)	**Propriedades Chave**
User (Usuário)	userId, nome
Song (Música)	songId, titulo, lancamento
Artist (Artista)	artistId, nome
Genre (Gênero)	nome

Interação (Aresta - Type)	**Conexão (De → Para)**	**Propriedades Chave**
PERFORMED_BY (Interpretada por)	**Song** $\to$ **Artist**	
HAS_GENRE (Tem Gênero)	**Song** $\to$ **Genre**	
LIKES (Curte)	**User** $\to$ **Song**	timestamp (data/hora da curtida)

LISTENED (Escutou)	**User** $\to$ **Song**	count (número de vezes), last_listened
FOLLOWS (Segue)	**User** $\to$ **Artist**	timestamp
FOLLOWS_GENRE (Segue Gênero)	**User** $\to$ **Genre**	timestamp

📝 Exemplos de Consultas Cypher para Recomendação

Com o modelo do grafo definido, o **Cypher** permite criar consultas complexas para gerar recomendações personalizadas com base em diferentes lógicas de conexão:

1. Recomendação Baseada em Usuários Semelhantes (Filtragem Colaborativa)

Encontrar músicas curtidas por outros usuários que curtiram músicas em comum com o usuário atual (User_A), mas que User_A ainda não curtiu/escutou.

Consulta Cypher:

```
Cypher
MATCH (User_A:User {userId: 'A123'})-[:LIKES]->(Song_X:Song)<-[:LIKES]-(User_B:User)
WHERE User_A <> User_B
WITH User_B, count(Song_X) AS common_likes
ORDER BY common_likes DESC
LIMIT 10
MATCH (User_B)-[:LIKES]->(Recommended_Song:Song)
WHERE NOT (User_A)-[:LIKES]->(Recommended_Song)
  AND NOT (User_A)-[:LISTENED]->(Recommended_Song)
RETURN Recommended_Song.titulo, common_likes AS similaridade_usuario
ORDER BY common_likes DESC
LIMIT 5
```

Esta consulta encontra "vizinhos" do User_A (User_B) com base no número de músicas curtidas em comum e, em seguida, sugere as músicas curtidas por esses vizinhos.

2. Recomendação Baseada em Gênero/Artista (Filtragem Baseada em Conteúdo)

Encontrar músicas populares dentro dos **Gêneros** que o usuário escutou mais, ou músicas de **Artistas** que o usuário segue.

Consulta Cypher (Gênero):

```
Cypher
MATCH (u:User {userId: 'A123'})-[:LISTENED]->(s:Song)-[:HAS_GENRE]->(g:Genre)
WITH g, sum(r.count) AS total_plays
ORDER BY total_plays DESC
LIMIT 3
MATCH (g)<-[:HAS_GENRE]-(rec:Song)
WHERE NOT (u)-[:LISTENED]->(rec)
RETURN rec.titulo, g.nome
LIMIT 5
```

3. Recomendação por Caminhos no Grafo (Recomendação Mista)

Encontrar músicas que compartilham o mesmo gênero e são interpretadas por um artista que User_A segue.

Consulta Cypher:

```
Cypher
MATCH (u:User {userId:
'A123'})-[:FOLLOWS]->(a:Artist)<-[:PERFORMED_BY]-(s:Song)-[:HAS_GENRE]->(g:Genre)
MATCH (g)<-[:HAS_GENRE]-(rec:Song)-[:PERFORMED_BY]->(artist_rec:Artist)
WHERE NOT (u)-[:LISTENED]->(rec) AND NOT (u)-[:LIKES]->(rec)
RETURN rec.titulo, artist_rec.nome AS artista, g.nome AS genero
LIMIT 5
```

Esta abordagem utiliza múltiplos tipos de relacionamentos para refinar a sugestão.

✨ Próximos Passos na Implementação

1. **Escolha do Banco de Dados de Grafos:** O **Neo4j** é a escolha mais popular e robusta para isso.
2. **Preparação e Importação de Dados:** Coletar dados de usuários, músicas, artistas, gêneros e interações (curtidas, plays, etc.). Usar comandos LOAD CSV ou *drivers* de linguagem (como o *Neo4j Driver* em Python) para carregar os dados no formato de grafo.
3. **Criação de Índices:** Criar índices (ex: CREATE INDEX ON :User(userId)) para otimizar as consultas MATCH e garantir alta performance, essencial para um sistema de recomendação em tempo real.

Com certeza! Para visualizar os nós do seu algoritmo de recomendação, vou descrevê-los e representá-los em um esquema visual que define a estrutura do seu grafo.

🔗 Nós (Nodes) do Sistema de Recomendação

No seu modelo, existem quatro tipos principais de nós (ou *Labels*) que representam as entidades básicas do seu domínio de músicas e usuários.

Label (Rótulo do Nó)	Representa	Propriedades Chave
(:User)	O indivíduo que interage com o sistema.	`userId` (ID Único), `nome`
(:Song)	Uma faixa musical específica.	`songId` (ID Único), `titulo`, `lancamento`
(:Artist)	O músico ou grupo que performa a música.	`artistId` (ID Único), `nome`
(:Genre)	A classificação categórica da música.	`nome`

🎨 Representação Gráfica dos Nós e Conexões

Este é um diagrama simplificado que mostra como esses nós se conectam por meio das arestas (relacionamentos), definindo o **Esquema de Grafo (Graph Schema)** do seu sistema.

No diagrama acima, os nós são os círculos (entidades), e as setas (relações) são as interações e associações que geram as recomendações:

1. **Usuário $\to$ Música:** O usuário **curte** ou **escuta** músicas.
    - Exemplo de Arestas: `[:LIKES]`, `[:LISTENED]`
2. **Música $\to$ Artista:** A música é **performada por** um artista.
    - Exemplo de Aresta: `[:PERFORMED_BY]`
3. **Música $\to$ Gênero:** A música **pertence ao gênero**.
    - Exemplo de Aresta: `[:HAS_GENRE]`
4. **Usuário $\to$ Artista/Gênero:** O usuário pode **seguir** um artista ou gênero.
    - Exemplo de Arestas: `[:FOLLOWS]`, `[:FOLLOWS_GENRE]`

Essa estrutura permite que consultas Cypher encontrem caminhos como "Usuários que curtem as mesmas Músicas, que são do mesmo Gênero".

Para detalhar completamente seu modelo de grafo, precisamos definir os **tipos de Relacionamentos (Arestas)** e as **Propriedades** que eles carregarão. As propriedades nas

arestas são cruciais, pois elas adicionam contexto (como a frequência ou o momento da interação) que o Cypher usará para refinar as recomendações.

🔗 Detalhe das Arestas (Relacionamentos)

Os relacionamentos ligam os nós e representam as interações ou associações entre as entidades.

---

1. Interações do Usuário

Essas arestas capturam o comportamento explícito e implícito do usuário no sistema.

Tipo de Aresta	Conexão (De → Para)	Propriedades Essenciais	Finalidade para Recomendação
[:LISTENED]	**(:User) $\to$ (:Song)**	count (Número de vezes escutadas), last_listened (Data/hora da última reprodução)	**Recomendação Implícita:** Indica forte preferência por volume e recência.
[:LIKES]	**(:User) $\to$ (:Song)**	timestamp (Data/hora da curtida)	**Recomendação Explícita:** Indica preferência positiva e clara.
[:FOLLOWS]	**(:User) $\to$ (:Artist)**	timestamp	**Recomendação Baseada em Conteúdo:** Sugere outras músicas/conteúdos do artista seguido.

[:FOLLOWS_GENRE]	**(:User) $\to$ (:Genre)**	timestamp	**Recomendação Baseada em Conteúdo:** Sugere músicas populares dentro do gênero seguido.

---

2. Associações Estruturais

Essas arestas ligam as entidades de conteúdo musical.

Tipo de Aresta	Conexão (De → Para)	Propriedades Essenciais	Finalidade para Recomendação
[:PERFORMED_BY]	**(:Song) $\to$ (:Artist)**	(Nenhuma, a relação é puramente associativa)	**Navegação:** Permite encontrar o artista de uma música ou todas as músicas de um artista.
[:HAS_GENRE]	**(:Song) $\to$ (:Genre)**	(Nenhuma, se a música tiver um gênero principal)	**Navegação:** Permite encontrar músicas pelo gênero ou gêneros de uma música.

💡 Como Usar as Propriedades no Cypher

As propriedades das arestas são vitais para calcular a **força** da conexão. Por exemplo:

- Ponderação (Weighting): Em vez de apenas verificar se um usuário escutou uma música, o Cypher pode usar a propriedade count do relacionamento [:LISTENED] para pesar a importância dessa música. Músicas escutadas 50 vezes terão mais peso na similaridade de usuário do que as escutadas apenas uma vez. $$Similarity = \sum_{S} (Weight_{A, S} \times Weight_{B, S})$$
- **Recência (Recency):** A propriedade last_listened ou timestamp pode ser usada para aplicar um **decay (decaimento)**, priorizando as interações mais recentes,

garantindo que as recomendações não sejam baseadas apenas em gostos de anos atrás.

Com este esquema de nós e arestas detalhado, é possível escrever consultas Cypher extremamente sofisticadas para encontrar os caminhos mais relevantes no seu grafo.

Excelente escolha! Vamos criar uma consulta Cypher que utiliza o número de vezes que os usuários escutaram as músicas (`count` na aresta `[:LISTENED]`) para calcular a **similaridade ponderada** entre eles.

Esta é a base da **Filtragem Colaborativa (Item-Based)**, mas focada na intensidade da interação.

🧮 Consulta Cypher para Similaridade Ponderada

O objetivo é encontrar usuários semelhantes ao `User_A` com base na soma das contagens de escutas que eles têm em comum, e então usar essa similaridade para recomendar novas músicas.

1. Definindo o Usuário Alvo e a Métrica de Similaridade

Vamos assumir que o ID do usuário alvo é `'User_A123'`.

```
Cypher

MATCH (u1:User {userId: 'User_A123'})-[l1:LISTENED]->(s:Song)<-[l2:LISTENED]-(u2:User)
WHERE u1 <> u2
// Calcula o "peso" da similaridade somando as contagens de escuta (count) em comum
// Multiplicamos as contagens (l1.count * l2.count) para dar mais peso às músicas
// que ambos escutaram muitas vezes.
WITH u2, sum(l1.count * l2.count) AS weighted_similarity
ORDER BY weighted_similarity DESC
LIMIT 10
```

- **Explicação:** Esta parte encontra todos os pares de usuários (`u1`, `u2`) que escutaram a mesma música (`s`). A `weighted_similarity` é a soma do produto das contagens de escuta para cada música em comum, dando um alto valor para conexões onde ambos os usuários interagiram intensamente.

---

2. Gerando as Recomendações

Agora, usamos os usuários mais semelhantes (`u2`) para encontrar as músicas que o `User_A123` ainda não conhece.

```
Cypher

MATCH (u1:User {userId: 'User_A123'})-[l1:LISTENED]->(s:Song)<-[l2:LISTENED]-(u2:User)
WHERE u1 <> u2
```

```
WITH u1, u2, sum(l1.count * l2.count) AS weighted_similarity
ORDER BY weighted_similarity DESC
LIMIT 10 // Pega os 10 vizinhos mais semelhantes

MATCH (u2)-[rec_listen:LISTENED]->(recommended_song:Song)
// Filtra: A música recomendada não deve ter sido escutada ou curtida pelo usuário alvo (u1)
WHERE NOT (u1)-[:LISTENED]->(recommended_song) AND NOT
(u1)-[:LIKES]->(recommended_song)

// Agrupa as músicas e soma a similaridade ponderada de todos os vizinhos que escutaram
essa música
RETURN recommended_song.titulo,
       sum(weighted_similarity * rec_listen.count) AS final_recommendation_score
ORDER BY final_recommendation_score DESC
LIMIT 5
```

- **final_recommendation_score:** Este score combina a similaridade do vizinho (weighted_similarity) com a intensidade da escuta do vizinho (rec_listen.count) para a música recomendada. Isso garante que a recomendação seja de uma música que o vizinho *muito semelhante* escutou *muitas vezes*.

Esta consulta representa um algoritmo de recomendação robusto e personalizado, utilizando as propriedades das arestas do seu grafo.

Com prazer! Vamos ver como usar a linguagem Cypher do Neo4j para criar os nós e, mais importante, o relacionamento [:LISTENED] com a propriedade count.

O comando **MERGE** é o mais recomendado aqui, pois ele age como um "CREATE OR MATCH": se a entidade (nó ou relacionamento) não existe, ele a cria; se já existe, ele a encontra (match), evitando duplicatas.

1. Criando os Nós (Entidades)

Primeiro, vamos garantir que as entidades básicas existam:

1.1 Criar Usuário (User)

```
Cypher

MERGE (u:User {userId: 'User_A123'})
SET u.nome = 'Alice'
RETURN u
```

1.2 Criar Artista (Artist)

```
Cypher

MERGE (a:Artist {artistId: 'Art_001'})
```

```
SET a.nome = 'The Graphers'
RETURN a
```

1.3 Criar Gênero (Genre)

```
Cypher

MERGE (g:Genre {nome: 'Rock Alternativo'})
RETURN g
```

1.4 Criar Música (Song) e suas Associações

Aqui criamos a música e a conectamos ao Artista e ao Gênero em uma única transação, usando o `MERGE` para garantir que os nós de suporte (Artista, Gênero) já existam antes de criar as arestas:

```
Cypher

// 1. Garante que o Artista e o Gênero existam (ou os encontra)
MERGE (a:Artist {artistId: 'Art_001'})
MERGE (g:Genre {nome: 'Rock Alternativo'})

// 2. Cria ou encontra a Música
MERGE (s:Song {songId: 'Song_999'})
SET s.titulo = 'Consulta Complexa',
    s.lancamento = 2024

// 3. Cria as Arestas Estruturais
MERGE (s)-[:PERFORMED_BY]->(a)
MERGE (s)-[:HAS_GENRE]->(g)

RETURN s, a, g
```

2. Criando a Aresta com Propriedade (Interação)

Agora, vamos focar no relacionamento principal para a recomendação: `[:LISTENED]`, garantindo que ele tenha a propriedade `count` (contador de vezes que foi escutada).

Usamos o **`MATCH`** para encontrar as entidades que já criamos e, em seguida, **`MERGE`** para o relacionamento.

Exemplo de Aresta :LISTENED com 'count'

Suponha que `'User_A123'` escutou a música `'Song_999'` **15 vezes** pela última vez em **2025-11-13**.

```
Cypher
```

```
MATCH (u:User {userId: 'User_A123'})
MATCH (s:Song {songId: 'Song_999'})

// Cria ou encontra a Aresta
MERGE (u)-[l:LISTENED]->(s)

// Define ou Atualiza as Propriedades da Aresta
SET l.count = 15,
    l.last_listened = date('2025-11-13')

RETURN u, l, s
```

💡 Dica para Atualizações:

Se o usuário escutar a música novamente, você pode usar a cláusula **SET** para **incrementar** a propriedade count e atualizar o last_listened:

```
Cypher

MATCH (u:User {userId: 'User_A123'})-[l:LISTENED]->(s:Song {songId: 'Song_999'})

// Incrementa o contador e atualiza a data
SET l.count = l.count + 1,
    l.last_listened = date() // Usa a data atual do sistema

RETURN l
```

Com este conhecimento, você pode criar todo o seu grafo de recomendação, garantindo que as arestas contenham as informações quantitativas (count) necessárias para a sua lógica de similaridade ponderada.

Com certeza! Para um sistema de recomendação real, você usará o comando **LOAD CSV** para importar grandes volumes de dados de forma eficiente. Este método é essencial para popular seu grafo com milhares ou milhões de interações.

💾 O Comando LOAD CSV

O LOAD CSV lê dados de um arquivo CSV (local ou remoto) e os utiliza dentro de comandos Cypher como MERGE ou CREATE. Usaremos o **WITH HEADERS** para tratar a primeira linha do CSV como cabeçalhos, tornando o código mais legível.

1. Pré-requisitos e Estrutura do CSV

O arquivo CSV deve ser salvo no diretório de importação (<neo4j-home>/import) do seu banco de dados Neo4j, ou em um local acessível.

Estrutura do Arquivo interacoes_musicas.csv

userId	songId	listen_count	last_listened
User_A123	Song_999	15	2025-11-13
User_B456	Song_999	5	2025-11-12
User_A123	Song_888	30	2025-11-10
User_C789	Song_888	10	2025-11-14
...	...	...	...

---

2. Cypher para Importar Interações

A consulta a seguir fará o seguinte para cada linha do CSV:

1. **Encontrar/Criar Nós:** Garante que os nós `User` e `Song` existam, usando `MERGE` com base nos IDs.
2. **Criar/Atualizar Aresta:** Cria ou encontra a aresta `[:LISTENED]` entre eles.
3. **Definir Propriedades:** Define as propriedades `count` e `last_listened` na aresta, convertendo os valores de *string* (vindos do CSV) para tipos numéricos e de data.

```
Cypher

// 1. Especifica o caminho do arquivo e usa os cabeçalhos
LOAD CSV WITH HEADERS FROM 'file:///interacoes_musicas.csv' AS row

// 2. Cria ou encontra os nós de User e Song
// É VITAL ter índices em :User(userId) e :Song(songId) para performance!
MATCH (u:User {userId: row.userId})
MATCH (s:Song {songId: row.songId})

// 3. Cria ou encontra o relacionamento :LISTENED
MERGE (u)-[l:LISTENED]->(s)

// 4. Define as propriedades do relacionamento (conversão de tipos)
SET l.count = toInteger(row.listen_count),
    l.last_listened = date(row.last_listened)
```

```
// Retorna o total de linhas processadas, a título de confirmação
RETURN count(row) AS total_interacoes_importadas
```

3. Otimização Essencial: Índices

Antes de rodar o `LOAD CSV` para grandes volumes, você **deve** criar índices ou restrições de unicidade nos IDs que você está usando no `MATCH` e `MERGE`. Isso transforma a busca de dados de lenta (escaneamento de grafo) para extremamente rápida (busca por índice).

```
Cypher

// Cria restrição de unicidade (e índice) para os nós User
CREATE CONSTRAINT user_id_unique IF NOT EXISTS FOR (u:User) REQUIRE u.userId IS UNIQUE;

// Cria restrição de unicidade (e índice) para os nós Song
CREATE CONSTRAINT song_id_unique IF NOT EXISTS FOR (s:Song) REQUIRE s.songId IS UNIQUE;
```

Com essas restrições e o comando `LOAD CSV`, você pode importar milhões de interações de forma rápida e segura para seu motor de recomendação.

https://github.com/MBARUDI