Universidade de Rio Verde
Faculdade de Engenharia de Software
Curso: Ciência de Dados e Machine Learning
Disciplina: Banco de Dados Relacional e Não Relacional

Trabalho Prático e Final da Disciplina

Professor: *Dr. Willian Garcias de Assunção - williang@unirv.edu.br*
Acadêmicos: *Mateus Abreu e Rhogger Freitas*

Sumário

1 Introdução e Objetivo

Este trabalho tem como objetivo a aplicação integrada dos conceitos de bancos de dados relacionais e não relacionais estudados na disciplina. O foco é a implementação de uma solução baseada no padrão de **Persistência Poliglota**, onde diferentes tecnologias de armazenamento de dados são selecionadas para atender a requisitos específicos de um sistema. A atividade consiste em projetar e prototipar a camada de persistência de dados para um sistema de negócio definido.

2 Objeto de Estudo do Projeto

O projeto se baseia na arquitetura de persistência de dados para uma plataforma de e-commerce, denominada **"DataDriven Store"**. O sistema deve ser capaz de gerenciar dados com diferentes naturezas e demandas, incluindo: transações financeiras e de pedidos que

exigem alta consistência; um catálogo de produtos com atributos variados; dados de sessão de usuário e carrinhos de compra com exigência de baixa latência; um grande volume de logs de eventos para análise de comportamento; e uma rede de relacionamentos entre entidades para sistemas de recomendação.

3 Ferramentas Utilizadas

- **Bancos:** PostgreSQL, MongoDB, Redis, Cassandra, Neo4j.
- **Orquestração/Ambiente:** Docker e Docker Compose.
- **API:** Fastify.
- **Controle de Versão:** Git.

4 Escopo e Requisitos do Projeto

A arquitetura do sistema será implementada por meio de uma API central, que realiza as operações de persistência de dados utilizando o padrão de Persistência Poliglota. A seguir, são detalhados os bancos de dados utilizados, suas responsabilidades, os modelos de dados, as justificativas para suas escolhas e os diagramas conceituais da arquitetura.

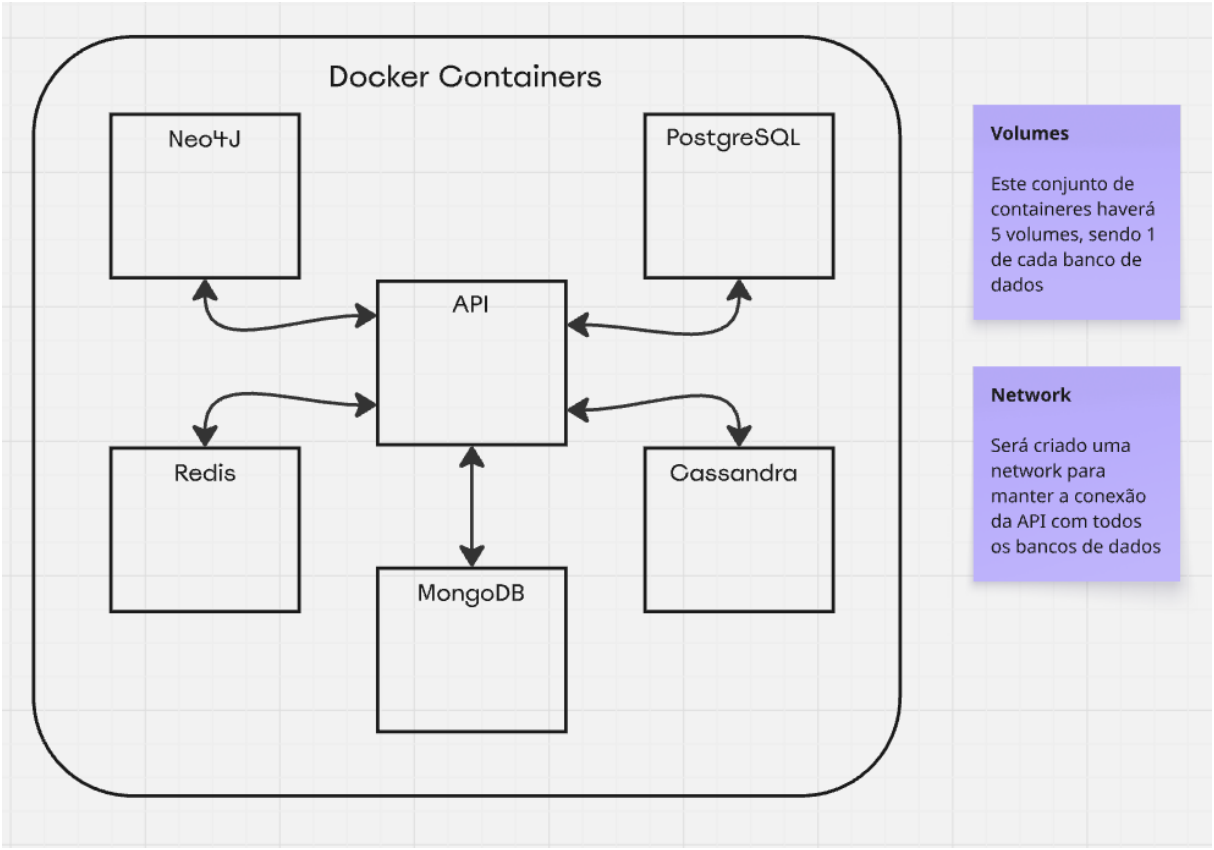

Figura 1: Diagrama da API se comunicando com os bancos

Banco de Dados Relacional (PostgreSQL)

Responsabilidade: Manter a consistência transacional (ACID) para os dados críticos e estruturados do negócio.

Dados a serem modelados: A modelagem relacional, conforme o diagrama abaixo, abrange as entidades fundamentais do e-commerce: `Clientes` (dados cadastrais essenciais), `Pedidos`, `Itens_Pedido` e `Transacoes_Financeiras`. O modelo também inclui uma infraestrutura de dados normalizada para `Enderecos`, `Cidades`, `Estados` e `Metodos_Pagamento`. Para garantir a integridade dos dados, são utilizados tipos enumerados (`ENUMS`) para controlar os status de pedidos, transações e tipos de endereços.

Justificativa: A estrutura rígida e as garantias transacionais do modelo relacional são adequadas para dados onde a integridade é um requisito fundamental. Além disso, a normalização implementada (com entidades como Endereços, Cidades, Estados e Métodos de Pagamento) garante a consistência, minimiza a redundância e otimiza a recuperação de informações críticas para o negócio.

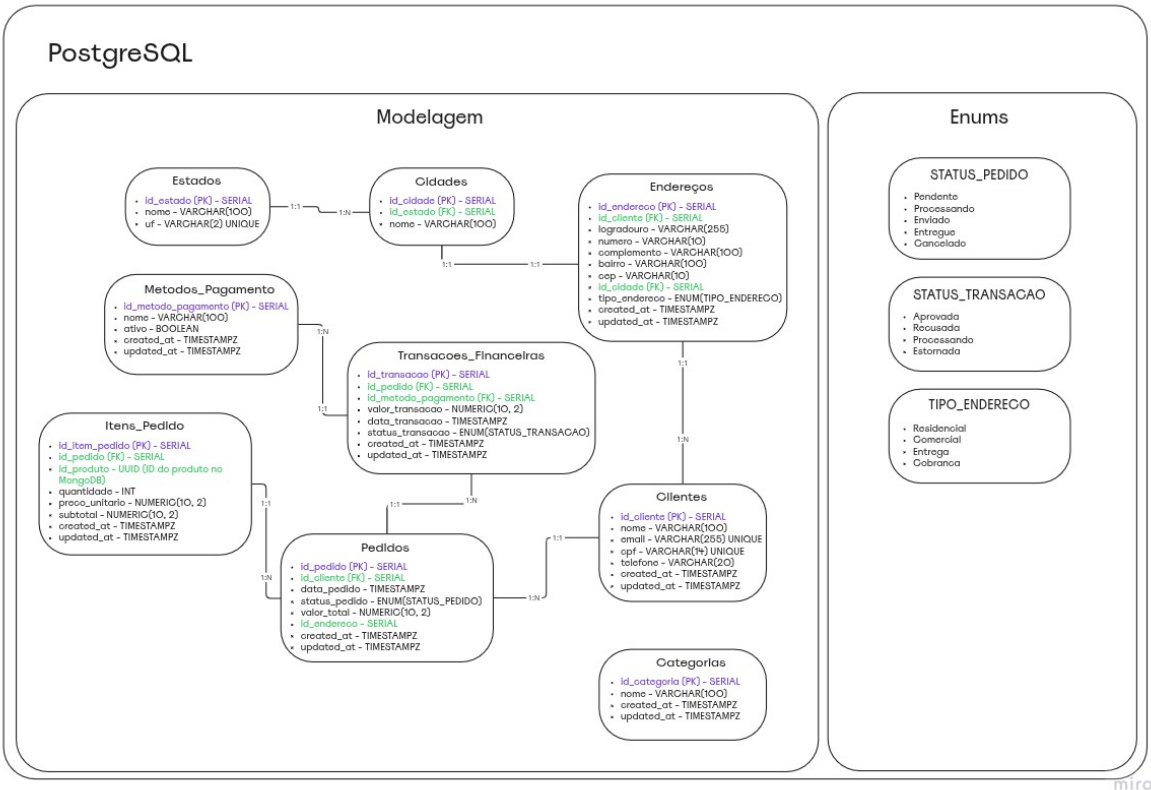

Figura 2: Diagrama Entidade-Relacionamento para o banco de dados PostgreSQL.

Banco de Dados de Documento (MongoDB)

Responsabilidade: Armazenar dados com estrutura flexível ou semiestruturada.

Dados a serem modelados: Catálogo de Produtos, onde cada item pode possuir um conjunto distinto de atributos; Perfis estendidos de usuários (preferências, dados demográficos complementares).

- **Modelagem para o MongoDB:**
  - **Produtos:** Esta coleção é projetada para armazenar o catálogo de produtos. Cada documento nesta coleção representa um produto e pode incluir uma ampla variedade de atributos, alguns dos quais podem ser específicos para diferentes tipos de produtos, bem como informações sobre avaliações.
  - **Perfis_Usuario:** Esta coleção armazena informações complementares e flexíveis sobre os usuários. Cada documento de perfil de usuário contém dados como preferências, informações demográficas adicionais e histórico de navegação recente, vinculados ao ID do cliente principal.

Justificativa: A flexibilidade do esquema (schema-on-read) do MongoDB facilita a evolução do catálogo e dos perfis de usuário sem a necessidade de migrações complexas na estrutura de dados. É ideal para dados que não se encaixam em um esquema relacional rígido e que se beneficiam de aninhamento de documentos para acesso eficiente.

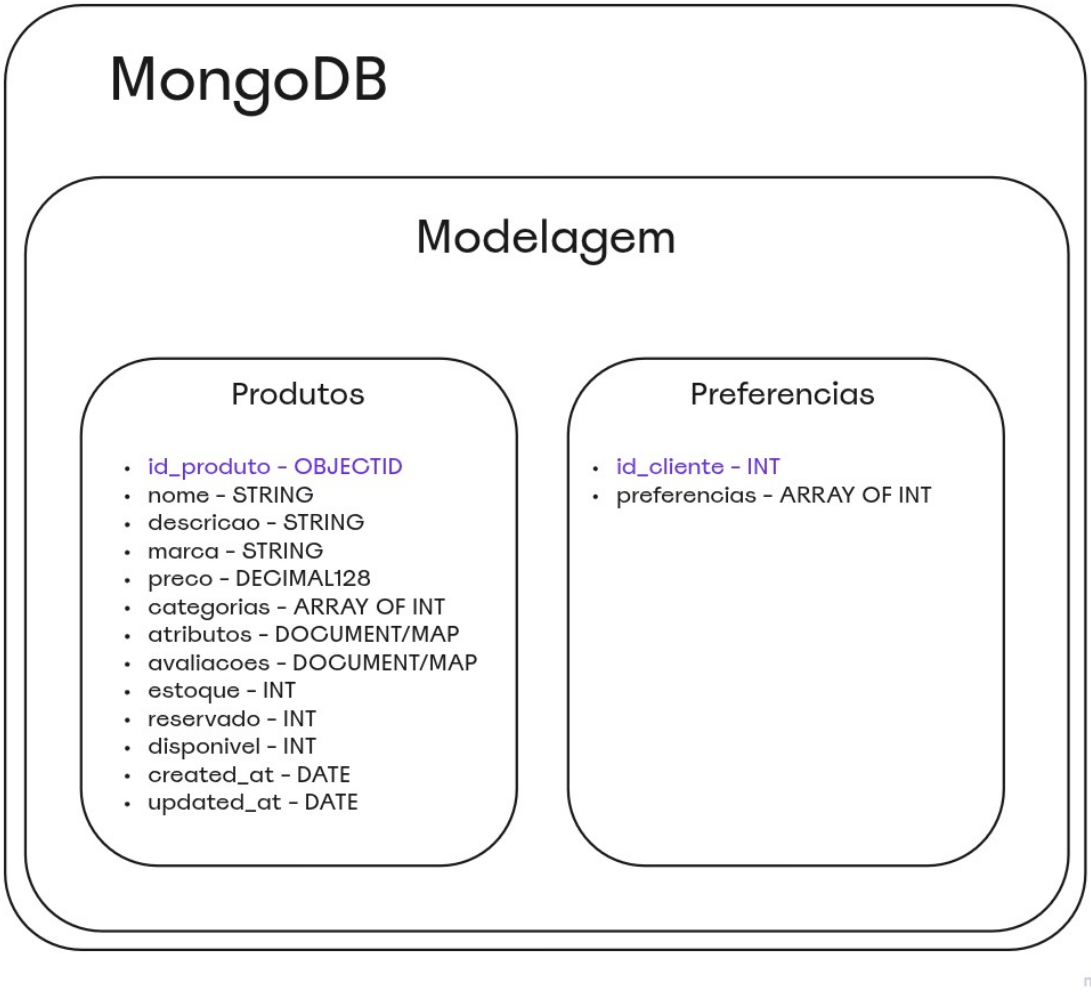

Figura 3: Diagrama de Coleções para o banco de dados MongoDB.

Banco de Dados Chave-Valor (Redis)

Responsabilidade: Gerenciar dados voláteis que requerem acesso com latência mínima.

Dados a serem modelados: Sessões de usuário, carrinhos de compra e cache de dados frequentemente acessados.

- **Modelagem para o Redis:**
  - **Sessão:** A chave representa a sessão ativa de um cliente específico, identificada pelo id_cliente. O valor associado a essa chave é um objeto que contém o token de autenticação e o refresh token do usuário. Esta estrutura é utilizada para manter o estado de login e autenticação, com um tempo de vida (TTL) configurável.
  - **Carrinho:** A chave representa o carrinho de compras de um cliente, identificado também pelo id_cliente. O valor associado a essa chave é uma estrutura de dados que armazena os identificadores dos produtos adicionados ao carrinho.
  - **Produtos:** A chave representa um item específico do catálogo de produtos, utilizado

para caching. O valor associado a essa chave é um objeto que contém os detalhes do produto, como seu nome, descrição, preço e atributos flexíveis. Esta modelagem visa armazenar informações frequentemente acessadas para reduzir a carga sobre o banco de dados principal (MongoDB).

Justificativa: O acesso a dados em memória proporciona o desempenho necessário para operações em tempo real que impactam diretamente a experiência do usuário. O Redis é ideal para cenários de alta taxa de leitura/escrita e dados que podem ter um tempo de vida limitado (TTL), como sessões e caches.

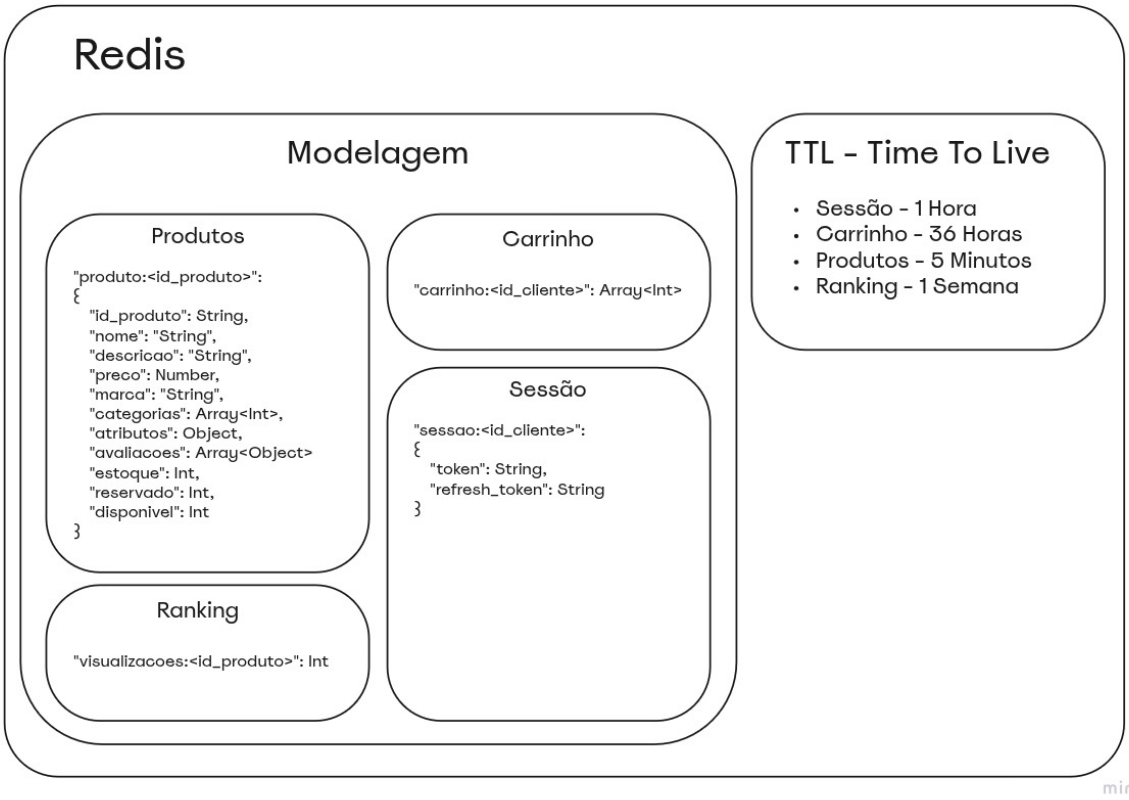

Figura 4: Estrutura de dados chave-valor para o Redis.

Banco de Dados Colunar (Apache Cassandra)

Responsabilidade: Ingestão e processamento de grandes volumes de dados para consultas analíticas (OLAP).

Dados a serem modelados: Logs de eventos da aplicação, como visualizações de produtos, cliques, termos de busca e outras interações do usuário.

- **Tabelas Modeladas para o Cassandra:**
  - **eventos_por_data:** Projetada para a ingestão de todos os eventos brutos gerados pela aplicação, otimizada para consultas que precisam acessar eventos em um determinado período de tempo.

- **eventos_por_usuario:** Modelada para capturar o histórico de atividades de um usuário individual, permitindo consultas eficientes para entender o comportamento completo de um usuário ao longo do tempo.
- **visualizacoes_produto_agregadas_por_dia:** Utilizada para armazenar contagens diárias agregadas do número de visualizações para cada produto, otimizada para responder a perguntas sobre a popularidade de produtos em dias específicos.
- **termos_busca_agregados_por_dia:** Armazena contagens diárias agregadas dos termos de busca mais utilizados, identificando tendências de busca e os termos mais populares.
- **funil_conversao_por_usuario_produto:** Projetada para rastrear o progresso do usuário através de um funil de conversão específico (visualizou -> adicionou ao carrinho -> comprou) para um determinado produto.
- **compras_por_utm_source:** Modelada para listar e analisar usuários que realizaram uma compra e que foram originados de uma 'utm_source' específica, útil para o desempenho de campanhas de marketing.

Justificativa: A arquitetura colunar é otimizada para agregações e varreduras em larga escala, sendo ideal para a geração de relatórios e dashboards analíticos. O Apache Cassandra se destaca por sua alta disponibilidade, escalabilidade linear e capacidade de lidar com grandes volumes de escritas rápidas em ambientes distribuídos, o que o torna uma excelente escolha para dados de logs.

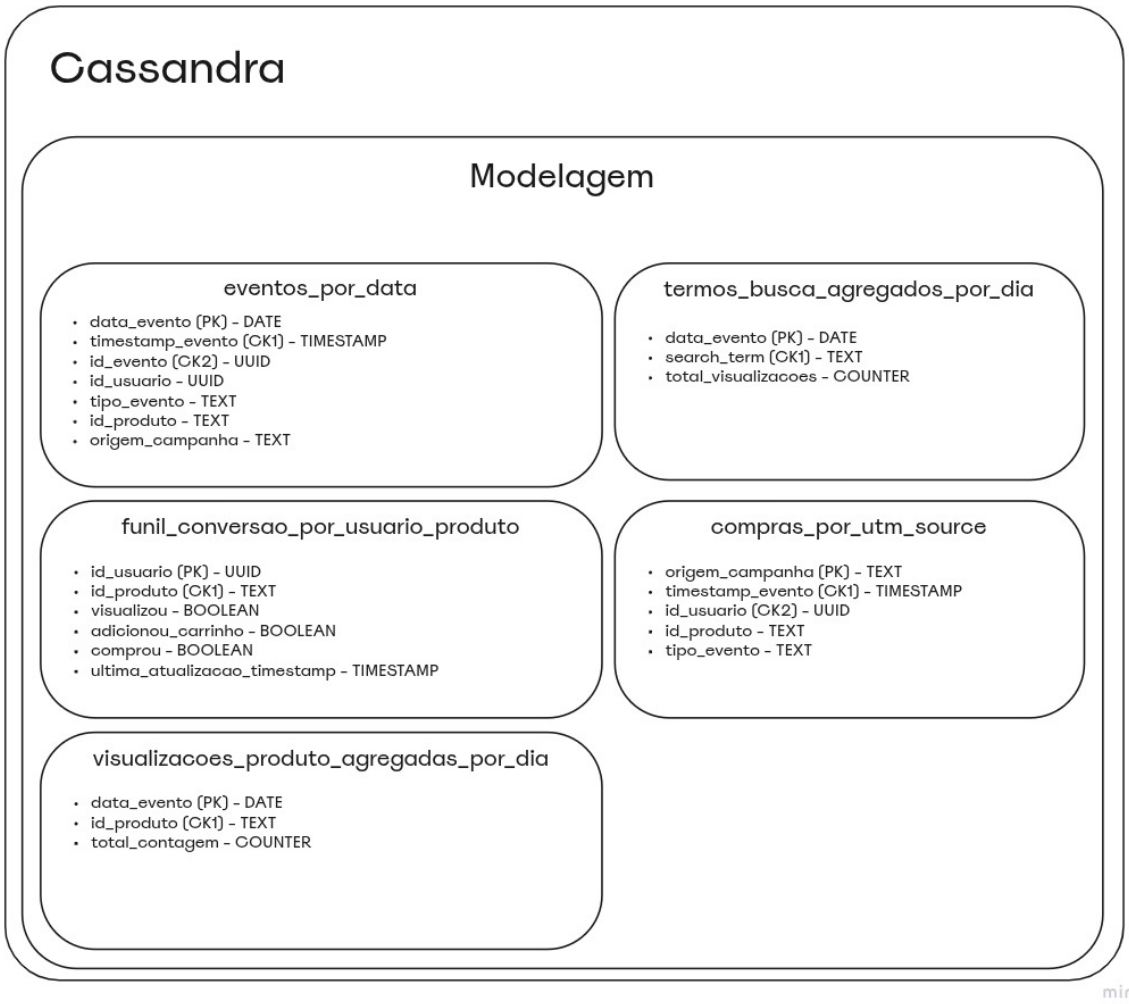

Figura 5: Modelo de dados para o Apache Cassandra.

Banco de Dados de Grafo (Neo4j)

Responsabilidade: Mapear, armazenar e consultar relacionamentos complexos entre as entidades do negócio, servindo como motor para o sistema de recomendação.

Dados a serem modelados: O modelo de grafo foi projetado para representar as principais entidades e suas interações, conforme ilustrado no diagrama. A estrutura é composta por:

- **Nós (Nodes):**
  - **:Cliente:** Representa os usuários, contendo as propriedades `{id_cliente, nome}`.
  - **:Produto:** Representa os itens do catálogo, com as propriedades `{id_produto, nome, preco}`.
  - **:Categoria:** Agrupa os produtos, identificado pela propriedade `{nome}`.
  - **:Marca:** Identifica o fabricante do produto, com a propriedade `{nome}`.
- **Arestas (Relationships):**

- **[:COMPROU]:** Conecta um `(:Cliente)` a um `(:Produto)`, com as propriedades `{data, id_pedido}` para registrar a transação.
- **[:AVALIOU]:** Liga um `(:Cliente)` a um `(:Produto)`, armazenando dados da avaliação nas propriedades `{rating, comentario, data}`.
- **[:VISUALIZOU]:** Registra a interação de um `(:Cliente)` com um `(:Produto)`, contendo a propriedade `{data}`.
- **[:PERTENCE_A]:** Relacionamento estrutural que conecta um `(:Produto)` a sua respectiva `(:Categoria)`.
- **[:PRODUZIDO_POR]:** Relacionamento estrutural que conecta um `(:Produto)` a sua `(:Marca)`.

Justificativa: A estrutura nativa de grafo é a mais eficiente para executar consultas baseadas em relacionamentos, que são a base de sistemas de recomendação. Questões como "clientes que compraram o produto X também compraram quais outros produtos?"(filtragem colaborativa) são resolvidas com travessias de grafo simples e performáticas, em contraste com múltiplos e complexos JOINs que seriam necessários em um banco relacional.

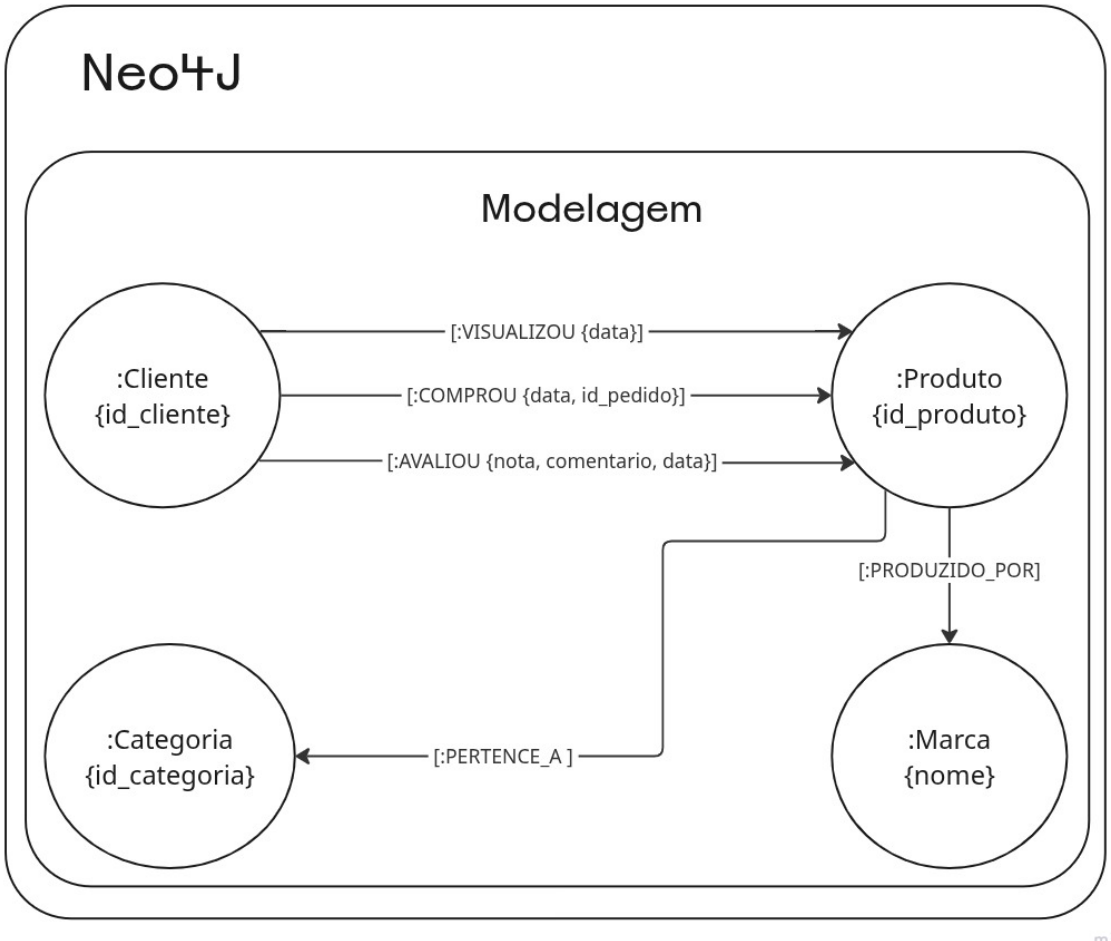

Figura 6: Diagrama do Modelo de Grafo para o Neo4j.

5 Detalhes da Aplicação

Este capítulo descreve a arquitetura e as ferramentas utilizadas para o desenvolvimento da API do "DataDriven Store". A aplicação foi construída como um sistema centralizado que orquestra as operações de persistência de dados, comunicando-se com os diferentes bancos de dados de acordo com a natureza de cada operação, seguindo o padrão de persistência poliglota.

Tecnologias e Frameworks Principais

- **Node.js com TypeScript:** A API foi desenvolvida sobre o ambiente de execução Node.js, escolhido por sua natureza assíncrona e orientada a eventos, o que o torna ideal para operações de I/O intensivas como as de uma API que lida com múltiplos bancos de dados. O uso do TypeScript adiciona uma camada de segurança de tipos ao JavaScript, o que melhora a manutenibilidade e a robustez do código.
- **Fastify:** Para a construção da API, foi utilizado o framework Fastify. Ele é conhecido por seu baixo overhead e alta performance, sendo uma escolha eficiente para criar serviços web rápidos. Sua arquitetura baseada em plugins facilitou a integração com os diversos bancos de dados e outras ferramentas, como o Swagger para documentação.

Orquestração de Ambiente e Dependências

- **Docker e Docker Compose:** Todo o ambiente da aplicação, incluindo a API e os cinco bancos de dados (PostgreSQL, MongoDB, Redis, Cassandra e Neo4j), é orquestrado utilizando Docker e Docker Compose. Isso garante um ambiente de desenvolvimento e execução consistente, isolado e facilmente reprodutível, eliminando problemas de configuração entre diferentes máquinas. Os arquivos `docker-compose.yml` e os scripts de inicialização (`start.sh`, `reset.sh`) automatizam todo o processo de subida e configuração dos serviços.
- **PNPM:** O gerenciamento de pacotes e dependências do projeto foi realizado com o PNPM. Ele foi escolhido por sua eficiência no uso de espaço em disco e performance na instalação de dependências.

Ferramentas de Desenvolvimento e Qualidade de Código

- **TS-Node-Dev:** Durante o desenvolvimento, a ferramenta `ts-node-dev` foi utilizada para habilitar o hot-reloading, permitindo que as alterações no código-fonte fossem refletidas automaticamente no container da API sem a necessidade de reiniciá-lo manualmente.
- **ESLint e Prettier:** Para garantir a qualidade e a consistência do código, foram configurados o ESLint para análise estática e identificação de padrões problemáticos, e o Prettier para a formatação automática do código, mantendo um estilo uniforme em todo o projeto.

Documentação da API

- **Swagger (OpenAPI):** A documentação da API foi gerada automaticamente a partir do código utilizando os plugins `@fastify/swagger` e `@fastify/swagger-ui`. Isso

proporciona uma documentação interativa e sempre atualizada, detalhando todos os endpoints, parâmetros, corpos de requisição e respostas esperadas.

6 Desafios

Durante o desenvolvimento do projeto "DataDriven Store", foram encontrados diversos desafios técnicos que foram cruciais para o aprendizado e aprofundamento nos conceitos de persistência poliglota. Esta seção detalha as principais dificuldades encontradas.

Configuração do Ambiente e Orquestração com Docker

Um dos primeiros grandes desafios foi a configuração e orquestração do ambiente de desenvolvimento. A necessidade de gerenciar cinco tecnologias de banco de dados distintas simultaneamente exigiu uma configuração complexa do arquivo `docker-compose.yml`. Para cada serviço (PostgreSQL, MongoDB, Redis, Cassandra e Neo4j), foi preciso definir não apenas as imagens e volumes, mas também as variáveis de ambiente, as redes e, fundamentalmente, as checagens de saúde (`healthchecks`).

Garantir que cada banco de dados estivesse não apenas em execução, mas totalmente operacional antes que a API principal tentasse se conectar, foi uma tarefa complexa que demandou múltiplos ajustes nos parâmetros de `interval`, `timeout` e `start_period` de cada `healthcheck`. Além disso, a gestão de um grande número de variáveis de ambiente no arquivo `.env` para as conexões de todos os bancos, tanto para a API quanto para os scripts de inicialização, representou uma camada adicional de complexidade na configuração inicial.

Adoção de Novas Tecnologias de Banco de Dados

Com exceção do PostgreSQL, com o qual a equipe já possuía familiaridade, as outras quatro tecnologias de banco de dados (MongoDB, Redis, Cassandra e Neo4j) representaram uma curva de aprendizado acentuada. Para a maioria dos integrantes, este foi o primeiro contato prático com os paradigmas de bancos de dados de documento, chave-valor, colunar e de grafo.

Este desafio não se limitou a aprender a sintaxe de novas linguagens de consulta, como a CQL (Cassandra Query Language) ou a Cypher (para Neo4j), mas envolveu a compreensão de como modelar os dados de forma otimizada para cada paradigma. Foi necessário estudar os casos de uso ideais e as melhores práticas de cada tecnologia para justificar corretamente sua aplicação na arquitetura do "DataDriven Store".

Complexidade na Carga de Dados (Seeding)

A etapa de povoamento dos bancos de dados (*seeding*) com dados de exemplo revelou-se um dos desafios mais complexos do projeto, devido às dependências de dados entre os diferentes sistemas de persistência. A natureza distribuída da arquitetura significava que a simples execução de scripts de carga de forma isolada era inviável.

O exemplo mais claro dessa dependência foi a criação de pedidos: a tabela `pedidos` e suas relacionadas (`itens_pedido`, `transacoes_financeiras`) no PostgreSQL dependem da existência de produtos, que, por sua vez, são armazenados no MongoDB. Isso impôs uma ordem de execução rigorosa para os scripts de *seeding*, que foi orquestrada no arquivo `seed-all.ts`:

1. Inicializar os esquemas das tabelas no PostgreSQL.
2. Popular o MongoDB com o catálogo de produtos.
3. Utilizar os dados dos produtos recém-criados no MongoDB para, então, popular as tabelas de pedidos no PostgreSQL.

Este padrão de dependência se repetiu em outras partes do sistema. Para popular o Neo4j com relacionamentos de compra (`[:COMPROU]`), era necessário ter tanto os clientes do PostgreSQL quanto os produtos do MongoDB já existentes. Da mesma forma, os eventos analíticos no Cassandra precisavam de identificadores de usuários e produtos reais, que existiam em outros bancos. Essa orquestração exigiu um planejamento cuidadoso e a criação de scripts robustos para garantir a integridade dos dados entre as diferentes tecnologias.

Configuração de Ambiente Multiplataforma

Por fim, garantir que o ambiente de desenvolvimento funcionasse de maneira consistente em diferentes sistemas operacionais, como Linux e Windows, apresentou seus próprios desafios. Problemas como diferenças em caminhos de arquivos, terminações de linha em scripts (`LF` vs. `CRLF`) e a compatibilidade de scripts de shell (`.sh`) precisaram ser gerenciados. A criação de um script como o `fix-permissions.sh` evidencia a necessidade de tratar questões de permissões de arquivos, que se comportam de maneira distinta em ambientes baseados em Unix e no Windows, para que o Docker pudesse montar os volumes corretamente sem erros.

7 Inicialização do Projeto

Esta seção apresenta um guia passo a passo para configurar e executar o ambiente completo do projeto "DataDriven Store"localmente. As instruções foram elaboradas para funcionar em ambientes Linux e Windows (utilizando um terminal com suporte a Bash, como o Git Bash).

Pré-requisitos

Antes de iniciar, certifique-se de que as seguintes ferramentas estão instaladas em sua máquina:

- **Git** – Para clonar o repositório de código.
- **Docker e Docker Compose** – Para a orquestração dos contêineres da aplicação e dos bancos de dados.
- **Terminal Bash** – Para usuários de Windows, é recomendado o uso do Git Bash.
- **Node.js e PNPM** – Necessários para o gerenciamento de dependências. As seções a seguir detalham a instalação.

Passo a Passo para Instalação

Siga as etapas abaixo para configurar o ambiente de desenvolvimento.

1. Clonar o Repositório

Primeiro, clone o repositório do projeto a partir do GitHub e acesse o diretório criado:

```
git clone https://github.com/rhogger/data-driven-store.git
cd data-driven-store
```

2. Configuração do Ambiente de Desenvolvimento

Escolha a opção que corresponde ao seu sistema operacional.

Opção A: Linux/macOS com ASDF (Recomendado) O uso do **asdf** é recomendado para ambientes baseados em Unix, pois gerencia as versões das ferramentas automaticamente e garante a consistência.

- Se você não tiver o **asdf** instalado, siga as instruções em <https://asdf-vm.com/guide/getting-started.html>.
- Adicione os plugins necessários para Node.js e PNPM:

```
asdf plugin-add nodejs
asdf plugin-add pnpm
```

- Instale as versões corretas das ferramentas, que estão definidas no arquivo `.tool-versions`:

```
asdf install
```

Opção B: Windows ou Instalação Manual Se estiver no Windows ou preferir não usar o ASDF, será necessário instalar o Node.js e o PNPM manualmente.

- **Node.js:** Acesse a página oficial de downloads do Node.js em <https://nodejs.org/pt/download> e baixe o instalador correspondente à versão **24.3.0**.
- **PNPM:** Após a instalação do Node.js, o **npm** (Node Package Manager) estará disponível. Utilize-o para instalar a versão exata do PNPM necessária para o projeto com o seguinte comando em seu terminal:

```
npm install -g pnpm@10.12.4
```

3. Instalar as Dependências do Projeto

Com o Node.js e o PNPM disponíveis em seu terminal, instale todas as dependências do projeto:

```
pnpm install
```

4. Configurar Variáveis de Ambiente

Crie um arquivo chamado `.env` na raiz do projeto, copiando o conteúdo abaixo. Este arquivo configura as conexões com os bancos de dados e outras variáveis da aplicação.

```
NODE_ENV=development

APP_PORT=3000

POSTGRES_HOST=postgres
POSTGRES_DB=datadriven_store
POSTGRES_USER=admin
POSTGRES_PASSWORD=admin
POSTGRES_PORT=5432

# MongoDB Configuration
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=admin
MONGO_INITDB_DATABASE=datadriven_store
MONGO_APP_USER=app_user
MONGO_APP_PASSWORD=app_password123
MONGODB_URI=mongodb://app_user:app_password123@mongo:27017/datadriven_store
MONGODB_DATABASE=datadriven_store

# Redis Configuration
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_DB=0

# Neo4j Configuration
NEO4J_URI=bolt://neo4j:7687
NEO4J_USER=neo4j
NEO4J_PASSWORD=admin123!*@

# Cassandra Configuration
CASSANDRA_HOST=cassandra
CASSANDRA_PORT=9042
CASSANDRA_USER=cassandra
CASSANDRA_PASSWORD=cassandra
CASSANDRA_KEYSPACE=datadriven
```

5. Executar o Ambiente com Docker

O projeto inclui scripts para automatizar a inicialização do ambiente. Para a primeira execução ou para resetar completamente o ambiente, utilize o script `reset.sh`. Ele é responsável por:

- Corrigir permissões de arquivos, se necessário.
- Parar e remover todos os contêineres e volumes da execução anterior.
- Reconstruir a imagem da API.

- Iniciar todos os serviços (API e bancos de dados) via Docker Compose.
- Executar os scripts de *seeding* para popular os bancos de dados.

Para executar o script, utilize o seguinte comando no seu terminal (Git Bash no Windows):

```
./reset.sh
```

6. Acessar a Aplicação

Após a conclusão do script `reset.sh`, o ambiente estará totalmente funcional. A API estará em execução e a documentação interativa, gerada pelo Swagger, pode ser acessada no seu navegador através do seguinte endereço:

<http://localhost:3000/docs>

O terminal que executou o script `reset.sh` passará a exibir os logs da API em tempo real. A partir da documentação do Swagger, é possível testar todos os endpoints disponíveis.

8 Sugestões de Consultas

Este capítulo serve como um guia prático para testar as funcionalidades da API "DataDriven Store", demonstrando como cada uma das cinco tecnologias de banco de dados é utilizada para atender a requisitos específicos. Cada consulta sugerida no escopo do projeto corresponde a um ou mais endpoints que podem ser executados através da documentação interativa do Swagger.

Autenticação: Pré-requisito para Testes

Antes de executar as consultas que exigem interação do usuário, é necessário realizar o processo de autenticação.

1. **Listar Categorias para Cadastro:** Para se cadastrar, é preciso informar as preferências de categoria. Visualize as categorias disponíveis executando um `GET` no endpoint:

   `/api/categories`

   Anote os IDs das categorias de sua preferência.

2. **Realizar o Cadastro:** Utilize o endpoint `POST /api/auth/register` e forneça no corpo da requisição seu nome, e-mail, CPF, telefone e um array com os IDs das categorias escolhidas.

3. **Realizar o Login:** Com o usuário criado, execute um `POST` no endpoint `/api/auth/login`, informando o e-mail cadastrado no corpo da requisição.

4. **Autorizar no Swagger:** A resposta do login conterá um `token`. Copie este token. No canto superior direito da página do Swagger, clique no botão **Authorize**, cole o token no campo "Value"e clique em **Authorize** novamente. A partir deste momento, todas as requisições enviadas pelo Swagger estarão autenticadas.

Consultas

PostgreSQL

1. **Consulta:** Transação atômica para criar um pedido, seus itens e atualizar o estoque.
   - **Endpoint:** `POST /api/orders`
   - **Instruções:** Para executar esta consulta, você precisará do ID de um cliente (o seu, após o cadastro) e do ID de um endereço associado a ele. Além disso, precisará de IDs de produtos existentes.
     (a) Obtenha IDs de produtos executando `GET /api/products`.
     (b) No corpo da requisição do endpoint de criação de pedido, forneça o `id_cliente`, `id_endereco` e um array de `itens`, cada um contendo o `id_produto` e a `quantidade`.
2. **Consulta:** Listar os 5 clientes com maior faturamento nos últimos 6 meses.
   - **Endpoint:** `GET /api/reports/top-customers`
   - **Instruções:** Basta executar o endpoint. Ele não requer parâmetros e retornará a lista dos top 5 clientes.
3. **Consulta:** Gerar um relatório de faturamento mensal, agrupado por categoria.
   - **Endpoint:** `GET /api/reports/billing-by-category`
   - **Instruções:** Execute o endpoint diretamente. Ele consolidará os dados de pedidos e produtos para gerar o relatório.
4. **Consulta:** Identificar produtos com estoque abaixo de um limiar.
   - **Endpoint:** `GET /api/products/low-stock`
   - **Instruções:** Este endpoint aceita um parâmetro de query opcional chamado `limiar`. Se não for fornecido, o padrão é 10. Você pode testar com diferentes valores, como `/api/products/low-stock?limiar=20`.
5. **Consulta:** Listar todos os pedidos de um cliente, incluindo o valor total.
   - **Endpoint:** `GET /api/orders/by-client/{id_cliente}`
   - **Instruções:** Substitua `{id_cliente}` pelo ID do cliente que você deseja consultar (por exemplo, o ID do seu usuário cadastrado).

MongoDB

1. **Consulta:** Usar o Aggregation Framework para calcular a média de preço por marca.
   - **Endpoint:** `GET /api/products/reports/average-price-by-brand`
   - **Instruções:** Execute o endpoint diretamente. A API utilizará o Aggregation Framework do MongoDB para processar os dados e retornar o resultado.
2. **Consulta:** Buscar produtos com atributos específicos (e.g., ‘processador: "i7"‘) e com preço em uma faixa definida.

- **Endpoint:** `POST /api/products/search`
- **Instruções:** No corpo da requisição, construa um objeto JSON para a busca. Por exemplo:

```
{
  "atributos": {
    "processador": "i7"
  },
  "preco_min": 500,
  "preco_max": 4000
}
```

3. **Consulta:** Adicionar um novo campo a todos os produtos de uma categoria.
   - **Endpoint:** `PUT /api/products/category/{categoryId}/add-field`
   - **Instruções:** Primeiro, obtenha um `categoryId` válido do endpoint `GET /api/categories`. Em seguida, no corpo da requisição, forneça o nome do campo e o valor a ser adicionado. Exemplo:

```
{
  "field_name": "em_promocao",
  "field_value": true
}
```

4. **Consulta:** Listar as avaliações de um produto, ordenadas por data.
   - **Endpoint:** `GET /api/products/{id}/reviews`
   - **Instruções:** Obtenha o `id` de um produto (via `GET /api/products`) e utilize-o no path do endpoint. É possível usar os parâmetros de query `page` e `pageSize` para paginação.
5. **Consulta:** Encontrar usuários que tenham uma preferência específica em seu perfil.
   - **Endpoint:** `GET /api/users/by-preference/{categoryId}`
   - **Instruções:** Obtenha um `categoryId` (via `GET /api/categories`) e utilize-o no path para encontrar todos os usuários que registraram preferência por ela.

Redis

1. **Consulta:** Simular login de usuário (comando SET com expiração).
   - **Endpoint:** `POST /api/auth/login`
   - **Instruções:** Conforme detalhado na seção de autenticação, ao executar o login, uma sessão com TTL é criada no Redis. A cada nova requisição autenticada, a validade da sessão é verificada e, se necessário, renovada.
2. **Consulta:** Gerenciar um carrinho de compras.

- **Instruções:** As rotas para manipulação direta do carrinho não foram implementadas, mas a lógica de repositório (`CartRepository`) existe e utiliza comandos HASH do Redis (`HSET`, `HDEL`, `HGETALL`) para gerenciar os itens. Esta funcionalidade seria a base para endpoints como `/api/cart/add`.

3. **Consulta:** Implementar cache de produtos.
   - **Endpoint:** `GET /api/products/{id}`
   - **Instruções:** Ao buscar um produto por ID pela primeira vez, a API busca no MongoDB e armazena o resultado no Redis com um TTL. Em chamadas subsequentes (dentro de 5 minutos), o resultado virá diretamente do cache do Redis, o que pode ser observado pela latência menor na resposta.
4. **Consulta:** Manter um ranking de produtos mais vistos (usando Sorted Set).
   - **Endpoint:** `GET /api/products/ranking`
   - **Instruções:** Este endpoint consulta o Sorted Set no Redis e retorna a lista de produtos mais vistos, ordenados por popularidade.
5. **Consulta:** Contar visualizações de página de um produto (usando INCR).
   - **Endpoint:** `POST /api/products/{id_produto}/view`
   - **Instruções:** Execute este endpoint para um `id_produto` específico. Cada execução incrementará o contador de visualizações daquele produto no Redis e atualizará sua posição no ranking.

Cassandra

1. **Consulta:** Consulta de funil de conversão.
   - **Endpoint:** `GET /api/analytics/conversion-funnel`
   - **Instruções:** Execute o endpoint diretamente. Ele agregará os dados da tabela `funil_conversao_por_usuario_produto` para calcular as taxas de conversão.
2. **Consulta:** Calcular o número de eventos de "visualização"por dia na última semana.
   - **Endpoint:** `GET /api/analytics/weekly-views`
   - **Instruções:** Basta executar o endpoint para obter um relatório das visualizações agregadas dos últimos 7 dias.
3. **Consulta:** Identificar os 10 termos de busca mais utilizados.
   - **Endpoint:** `GET /api/analytics/top-search-terms`
   - **Instruções:** Execute o endpoint para consultar os termos mais buscados, que são agregados na tabela `termos_busca_agregados_por_dia`.
4. **Consulta:** Calcular a taxa de cliques (CTR) de uma campanha.
   - **Endpoint:** `GET /api/analytics/campaign-ctr/{origemCampanha}`
   - **Instruções:** Substitua `{origemCampanha}` por uma fonte de campanha, como `google`, `facebook` ou `email`.

5. **Consulta:** Listar usuários que vieram de uma 'utm_source' específica e realizaram compra.
    - **Endpoint:** `GET /api/analytics/users-by-utm/{utmSource}`
    - **Instruções:** Substitua `{utmSource}` por uma fonte de UTM válida (ex: `google`) para ver a lista de usuários que compraram vindos deste canal.

Neo4j

1. **Consulta:** Filtragem Colaborativa (Item-Item): "Produtos frequentemente comprados juntos".
    - **Endpoint:** `GET /api/recommendations/{produtoId}/frequently-bought-together`
    - **Instruções:** Obtenha um `produtoId` (via `GET /api/products`) e utilize-o no endpoint para ver quais outros produtos foram comprados pelos mesmos clientes.
2. **Consulta:** Filtragem Colaborativa (User-User): "Recomendações baseadas em clientes similares".
    - **Endpoint:** `GET /api/recommendations/customers/{clienteId}/user-based`
    - **Instruções:** Utilize o seu `clienteId` (ou de outro usuário) para receber recomendações de produtos baseadas no histórico de compra de clientes com gostos parecidos.
3. **Consulta:** Encontrar o caminho mais curto entre dois produtos.
    - **Endpoint:** `GET /api/recommendations/shortest-path/{produtoOrigemId}/{produtoDe`
    - **Instruções:** Obtenha os IDs de dois produtos distintos e utilize-os nos parâmetros do endpoint para descobrir a relação mais curta entre eles no grafo (passando por categorias e marcas).
4. **Consulta:** Identificar clientes "influenciadores".
    - **Endpoint:** `GET /api/recommendations/influencers`
    - **Instruções:** Execute o endpoint diretamente para obter uma lista de clientes cujas avaliações positivas se correlacionam com um aumento nas vendas dos produtos que avaliaram.
5. **Consulta:** Recomendar produtos de categorias que um cliente visualizou, mas não comprou.
    - **Endpoint:** `GET /api/recommendations/customers/{clienteId}/category-based`
    - **Instruções:** Para que esta consulta retorne resultados, primeiro visualize alguns produtos sem comprá-los. Para isso, execute `POST /api/products/{id_produto}/view` para alguns produtos de uma categoria. Em seguida, execute este endpoint com o seu `clienteId` para receber recomendações de outros produtos daquela categoria.

9 Conclusão

O desenvolvimento do projeto cumpriu com sucesso o objetivo de aplicar, de forma prática, o padrão de persistência poliglota. Através da orquestração de cinco tecnologias de banco de dados distintas, a arquitetura final demonstrou a importância de utilizar a ferramenta certa para cada requisito específico: PostgreSQL para a integridade transacional, MongoDB para a flexibilidade de dados de produtos, Redis para a performance de caches e sessões, Cassandra para a ingestão de grandes volumes de eventos analíticos, e Neo4j para a complexa rede de relacionamentos do sistema de recomendação.

Os desafios enfrentados, desde a complexa configuração do ambiente com Docker até a orquestração da carga de dados entre os diferentes sistemas, foram fundamentais para solidificar o conhecimento teórico e transformar conceitos em experiência prática.

Ao final, o projeto se consolida como um estudo de caso abrangente e funcional, demonstrando a viabilidade e os benefícios de uma abordagem poliglota para a construção de sistemas de software modernos e orientados a dados.