



Disciplina: **BANCO DE DADOS RELACIONAIS E NÃO
RELACIONAIS**

Nome do curso: **ARQUITETURA DE DADOS**

Professor: Anderson Theobaldo

Atividade 03 - Projeto de Banco de Dados

Aluno: Nícolas Arruda Carballo

Objetivo	2
Estratégia de Particionamento de Dados	2
Implementação Simulada	3
Ferramentas escolhidas	3
Passos da simulação	3
Testes de desempenho	4
Cenários de testes	4
Resultados Esperados	4
Configuração ambiente	4
Arquivo SQL para Definição do Schema das Tabelas	4
Explicação	6
Integração de PostgreSQL e MongoDB para Replicação de Dados	7
Estrutura do Banco de Dados no MongoDB	7
Pipeline de Replicação	8
Configuração de exemplo	9
Configuração do Debezium	9
Configuração do Kafka Connect para MongoDB	9
Configuração ambiente local com Docker	10
Passos para configuração	12
Popular banco de dados Postgres com dados fictícios	14
Instalação de bibliotecas necessárias execução do script	14
Script em python para persistir dados	14
Explicação do script	18
Dados replicados no MongoDB	19

Objetivo

Desenvolver um sistema de gerenciamento de estoque distribuído e escalável para uma cadeia de supermercados com filiais em diferentes cidades, capaz de lidar com milhões de registros de produtos e realizar consultas e atualizações de forma rápida e eficiente.

Estratégia de Particionamento de Dados

1. Particionamento Horizontal

Escalabilidade: O particionamento horizontal distribui as linhas de uma tabela em diferentes nós. Cada nó armazena um subconjunto de registros, permitindo a adição de novas filiais sem comprometer o desempenho.

Performance: Consultas e atualizações de inventário podem ser distribuídas entre os nós, reduzindo a carga em um único servidor.

Manutenção: Facilita a manutenção e backup, pois cada nó pode ser gerenciado separadamente.

2. Particionamento Vertical

- **Performance:** Em situações onde apenas certos atributos dos produtos são frequentemente acessados (por exemplo, preço e quantidade em estoque), o particionamento vertical pode otimizar essas consultas.
- **Manutenção:** Simplifica a gestão de dados sensíveis ou menos acessados, permitindo armazenamento em servidores com diferentes níveis de segurança e performance.

3. Particionamento por Fragmentação

- **Performance:** Produtos podem ser agrupados por categorias ou tipos de produtos (por exemplo, perecíveis e não perecíveis), permitindo uma otimização das consultas específicas a cada grupo.
- **Eficiência:** Reduz a quantidade de dados a ser varrida durante consultas específicas a um fragmento.

Implementação Simulada

Ferramentas escolhidas

- **Banco de dados:** MongoDB (para particionamento horizontal e fragmentação) e PostgreSQL (para particionamento vertical).
- **Simulação de Dados:** Gerar dados utilizando Python com bibliotecas como Faker para simular produtos e filiais.

Passos da simulação

1. **Geração de dados:** Criar um conjunto de dados representando produtos e filiais, com milhões de registros.
2. **Configuração do Banco de Dados:** Implementar particionamento horizontal no MongoDB, vertical no PostgreSQL e fragmentação onde necessário.

3. **Inserção de Dados:** População do banco de dados com os dados gerados.
4. **Configuração de Índices:** Implementar índices para otimizar consultas frequentes.

Testes de desempenho

Cenários de testes

1. **Consulta de estoque:** Realizar consultas frequentes para verificar a quantidade de determinado produto em várias filiais.
2. **Atualizações de Inventário:** Testar a atualização dos níveis de estoque após vendas ou reposições.
3. **Adição de Novas Filiais:** Avaliar a performance do sistema ao adicionar novas filiais e redistribuir dados.

Resultados Esperados

- **Consultas rápidas:** Tempo de resposta rápido para consultas de estoque.
- **Atualizações Eficientes:** Atualizações rápidas e sem conflitos.
- **Escalabilidade:** Sistema mantendo desempenho com a adição de novas filiais.

Configuração ambiente

Arquivo SQL para Definição do Schema das Tabelas

Vamos criar um script SQL que define o schema das tabelas para o sistema de gerenciamento de estoque de um supermercado. Considerando a abordagem de fragmentação por categorias de produtos (perecíveis e não perecíveis) e particionamento vertical para otimização de consultas frequentes.

```
-- Definindo o schema para o gerenciamento de estoque de um  
supermercado
```

```
-- Schema para Produtos Perecíveis  
CREATE TABLE Produtos_Pereciveis (
```

```
    produto_id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL,
    quantidade INT NOT NULL,
    data_validade DATE NOT NULL,
    categoria VARCHAR(50) NOT NULL,
    filial_id INT NOT NULL
);

-- Schema para Produtos Não Perecíveis
CREATE TABLE Produtos_Nao_Pereciveis (
    produto_id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL,
    quantidade INT NOT NULL,
    categoria VARCHAR(50) NOT NULL,
    filial_id INT NOT NULL
);

-- Schema para Filiais
CREATE TABLE Filiais (
    filial_id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    endereco VARCHAR(255) NOT NULL,
    cidade VARCHAR(100) NOT NULL,
    estado VARCHAR(50) NOT NULL,
    cep VARCHAR(20) NOT NULL
);

-- Índices para otimização de consultas
CREATE INDEX idx_nome_produtos_pereciveis ON
Produtos_Pereciveis (nome);
CREATE INDEX idx_nome_produtos_ao_pereciveis ON
Produtos_Nao_Pereciveis (nome);
CREATE INDEX idx_categoria_produtos_pereciveis ON
Produtos_Pereciveis (categoria);
CREATE INDEX idx_categoria_produtos_ao_pereciveis ON
Produtos_Nao_Pereciveis (categoria);
CREATE INDEX idx_filial_produtos_pereciveis ON
Produtos_Pereciveis (filial_id);
CREATE INDEX idx_filial_produtos_ao_pereciveis ON
Produtos_Nao_Pereciveis (filial_id);
```

-- Exemplo de tabelas verticalmente particionadas para dados frequentemente acessados

-- Tabela para detalhes de preço e quantidade de produtos perecíveis

```
CREATE TABLE Detalhes_Preco_Quantidade_Pereciveis (  
    produto_id SERIAL PRIMARY KEY,  
    preco DECIMAL(10, 2) NOT NULL,  
    quantidade INT NOT NULL,  
    FOREIGN KEY (produto_id) REFERENCES  
Produtos_Pereciveis(produto_id)  
);
```

-- Tabela para detalhes de preço e quantidade de produtos não perecíveis

```
CREATE TABLE Detalhes_Preco_Quantidade_Nao_Pereciveis (  
    produto_id SERIAL PRIMARY KEY,  
    preco DECIMAL(10, 2) NOT NULL,  
    quantidade INT NOT NULL,  
    FOREIGN KEY (produto_id) REFERENCES  
Produtos_Nao_Pereciveis(produto_id)  
);
```

-- Tabela para detalhes de descrição de produtos perecíveis

```
CREATE TABLE Detalhes_Descricao_Pereciveis (  
    produto_id SERIAL PRIMARY KEY,  
    descricao TEXT,  
    data_validade DATE NOT NULL,  
    FOREIGN KEY (produto_id) REFERENCES  
Produtos_Pereciveis(produto_id)  
);
```

-- Tabela para detalhes de descrição de produtos não perecíveis

```
CREATE TABLE Detalhes_Descricao_Nao_Pereciveis (  
    produto_id SERIAL PRIMARY KEY,  
    descricao TEXT,  
    FOREIGN KEY (produto_id) REFERENCES  
Produtos_Nao_Pereciveis(produto_id)  
);
```

Explicação

1. Tabelas para produtos:

- **Produtos_Pereciveis e Produtos_Nao_Pereciveis:** Armazena informações básicas dos produtos, separando-os em perecíveis e não perecíveis.

2. Tabelas para filiais:

- **Filiais:** Armazena informações sobre as filiais do supermercado.

3. Índices:

- Criação de índices para otimizar consultas frequentes baseadas no nome, categoria e filial dos produtos.

4. Particionamento vertical:

- **Detalhes_Preco_Quantidade_Pereciveis** e **Detalhes_Preco_Quantidade_Nao_Pereciveis:** Armazena detalhes de preço e quantidade, frequentemente acessados.
- **Detalhes_Descricao_Pereciveis** e **Detalhes_Descricao_Nao_Pereciveis:** Armazena descrições e outros detalhes menos frequentemente acessados.

Integração de PostgreSQL e MongoDB para Replicação de Dados

Para utilizar o MongoDB como uma réplica dos dados frequentemente acessados que residem no PostgreSQL, podemos definir um pipeline de replicação que garante que as operações CRUD realizadas no PostgreSQL sejam refletidas no MongoDB. Isso pode ser feito utilizando ferramentas como Debezium para captura de mudanças no PostgreSQL e Kafka para transportar essas mudanças para o MongoDB.

Estrutura do Banco de Dados no MongoDB

Primeiro, vamos definir como os dados serão estruturados no MongoDB. A estrutura será similar àquela no PostgreSQL, mas com otimização para leitura e consultas frequentes.

```
{
  "filial_id": 1,
  "produtos_pereciveis": [
    {
      "produto_id": 1,
```

```

    "nome": "Leite",
    "preco": 5.99,
    "quantidade": 100,
    "categoria": "Laticínios",
    "data_validade": "2024-07-01"
  },
  ...
],
"produtos_nao_pereciveis": [
  {
    "produto_id": 2,
    "nome": "Arroz",
    "preco": 2.99,
    "quantidade": 200,
    "categoria": "Grãos"
  },
  ...
]
}

```

Pipeline de Replicação

1. Captura de Mudanças no PostgreSQL (Debezium):

- Debezium é uma plataforma de captura de dados que permite capturar mudanças nos bancos de dados PostgreSQL em tempo real.
- Configurar Debezium para monitorar as tabelas **Detalhes_Preco_Quantidade_Pereciveis** e **Detalhes_Preco_Quantidade_Nao_Pereciveis**.

2. Transporte de Mensagens (Kafka):

- Utilizar Kafka como um intermediário para transportar as mudanças capturadas pelo Debezium para o MongoDB.
- Configurar tópicos do Kafka para os dados de produtos perecíveis e não perecíveis.

3. Aplicação de Mudanças no MongoDB:

- Utilizar Kafka Connect com o conector MongoDB para aplicar as mudanças capturadas no MongoDB.
- Configurar o MongoDB para particionamento horizontal (sharding) baseado no campo `filial_id`.

Configuração de exemplo

Configuração do Debezium

```
{
  "name": "inventory-connector",
  "config": {
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "localhost",
    "database.port": "5432",
    "database.user": "postgres",
    "database.password": "password",
    "database.dbname": "supermercado",
    "database.server.name": "dbserver1",
    "table.include.list":
"public.detalhes_preco_quantidade_pereciveis,
public.detalhes_preco_quantidade_ao_pereciveis",
    "plugin.name": "pgoutput"
  }
}
```

Configuração do Kafka Connect para MongoDB

```
{
  "name": "mongodb-sink-connector",
  "config": {
    "connector.class":
"com.mongodb.kafka.connect.MongoSinkConnector",
    "tasks.max": "1",
    "topics":
"dbserver1.public.detalhes_preco_quantidade_pereciveis,
dbserver1.public.detalhes_preco_quantidade_ao_pereciveis",
    "connection.uri": "mongodb://localhost:27017",
    "database": "supermercado",
    "collection": "produtos",
    "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
    "value.converter":
"org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    "document.id.strategy":
"com.mongodb.kafka.connect.sink.processor.id.strategy.PartialV
alueStrategy",
    "document.id.strategy.partial.value.projection.type":

```

```
"AllowList",
  "document.id.strategy.partial.value.projection.list":
"produto_id"
}
}
```

Com essa abordagem, os dados frequentemente acessados serão replicados e particionados horizontalmente no MongoDB, enquanto o PostgreSQL continuará sendo a principal fonte de verdade para os dados. Esse setup garante alta disponibilidade, escalabilidade e eficiência nas consultas, ao mesmo tempo em que mantém a integridade dos dados.

Configuração ambiente local com Docker

Vamos usar um arquivo docker-compose.yml para criar os containers parte de nosso projeto de banco de dados.

```
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

  postgres:
```

```
image: postgres:latest
ports:
  - "5432:5432"
environment:
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: password
  POSTGRES_DB: supermercado
volumes:
  - postgres_data:/var/lib/postgresql/data

mongodb:
  image: mongo:latest
  ports:
    - "27017:27017"
  volumes:
    - mongo_data:/data/db

connect:
  image: debezium/connect:latest
  depends_on:
    - kafka
    - postgres
    - mongodb
  ports:
    - "8083:8083"
  environment:
    CONFIG_STORAGE_TOPIC: debezium_config
    OFFSET_STORAGE_TOPIC: debezium_offsets
    STATUS_STORAGE_TOPIC: debezium_status
    BOOTSTRAP_SERVERS: kafka:9092
    GROUP_ID: 1
    KEY_CONVERTER_SCHEMAS_ENABLE: "false"
    VALUE_CONVERTER_SCHEMAS_ENABLE: "false"
    CONNECT_KEY_CONVERTER:
org.apache.kafka.connect.json.JsonConverter
    CONNECT_VALUE_CONVERTER:
org.apache.kafka.connect.json.JsonConverter
    CONNECTOR_CLASS:
io.debezium.connector.postgresql.PostgresConnector
    DATABASE_HOSTNAME: postgres
    DATABASE_PORT: 5432
    DATABASE_USER: postgres
    DATABASE_PASSWORD: password
    DATABASE_DBNAME: supermercado
```

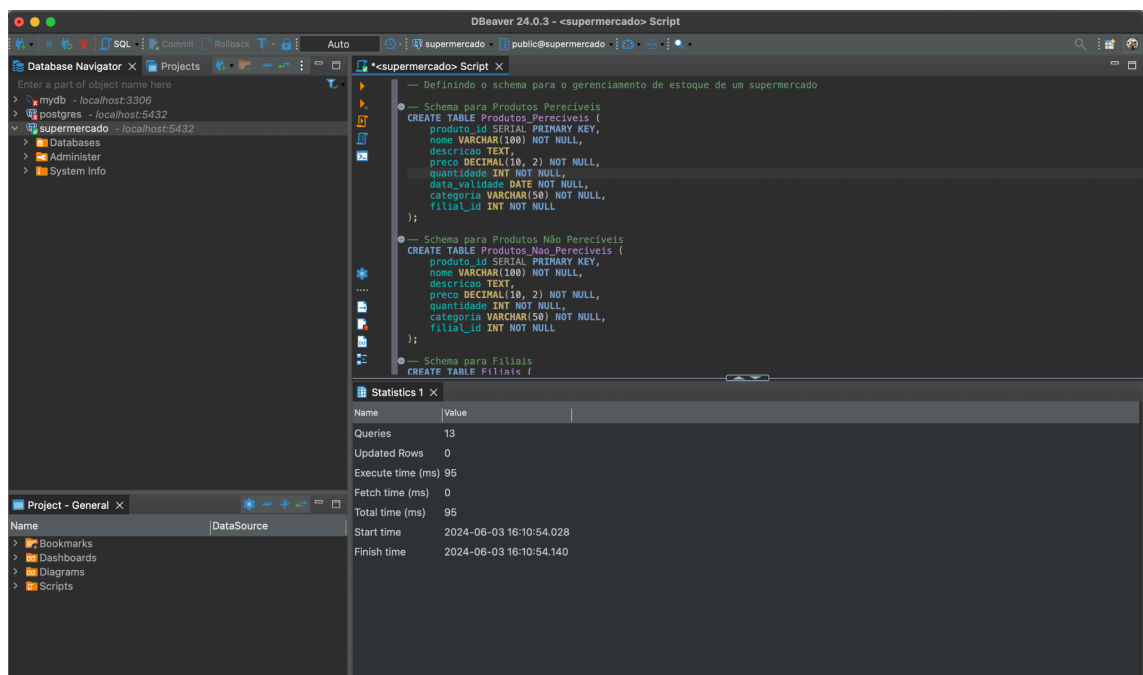
```
DATABASE_SERVER_NAME: dbserver1
PLUGIN_NAME: pgoutput
volumes:
  - connect_data:/kafka/connect
```

```
volumes:
  postgres_data:
  mongo_data:
  connect_data:
```

Passos para configuração

1. **Suba os containers:** Execute o comando abaixo no diretório onde o arquivo docker-compose.yml está localizado.

```
docker-compose up -d
```
2. **Criação do Schema no PostgreSQL:** Conecte-se ao container do PostgreSQL e crie o esquema do banco de dados.



3. **Configuração do Debezium:** Depois que o PostgreSQL e MongoDB estiverem configurados e os dados forem populados, configure o Debezium para iniciar a replicação.

Crie um arquivo **register-postgres.json** com o seguinte conteúdo:

```
{
  "name": "inventory-connector",
  "config": {
    "connector.class":
    "io.debezium.connector.postgresql.PostgresConnector",
```

```

"tasks.max": "1",
"database.hostname": "postgres",
"database.port": "5432",
"database.user": "postgres",
"database.password": "password",
"database.dbname": "supermercado",
"database.server.name": "dbserver1",
"table.include.list":
"public.detalhes_preco_quantidade_pereciveis,public.detalhes_p
reco_quantidade_ao_pereciveis",
"plugin.name": "pgoutput",
"database.history.kafka.bootstrap.servers":
"kafka:9092",
"database.history.kafka.topic":
"schema-changes.inventory",
"topic.prefix": "dbserver1"
}
}

```

Registre o conector:

```

curl -i -X POST -H "Accept:application/json" -H
"Content-Type:application/json"
http://localhost:8083/connectors/ -d @register-postgres.json

```



```

HTTP/1.1 200 OK
Date: Mon, 10 Sep 2023 14:12:00 GMT
Location: http://localhost:8083/connectors/inventory-connector
Content-Type: application/json
Content-Length: 645
Server: Jetty(9.4.52.v20230823)

{"name":"inventory-connector","config":{"connector.class":"io.debezium.connector.postgresql.PostgresConnector","tasks.max":"1","database.hostname":"postgres","database.port":"5432","database.us
er":"postgres","database.password":"password","database.dbname":"supermercado","database.server.name":"dbserver1","table.include.list":"public.detalhes_preco_quantidade_pereciveis,public.detalh
es_preco_quantidade_ao_pereciveis","plugin.name":"pgoutput","database.history.kafka.bootstrap.servers":"kafka:9092","database.history.kafka.topic":"schema-changes.inventory","topic.prefix":"db
server1","name":"inventory-connector"},"tasks":[],"type":"source"}

```

Crie um arquivo **register-mongodb-sink.json**:

```

{
  "name": "mongodb-sink-connector",
  "config": {
    "connector.class":
"com.mongodb.kafka.connect.MongoSinkConnector",
    "tasks.max": "1",
    "topics":
"dbserver1.public.detalhes_preco_quantidade_pereciveis,dbserve
r1.public.detalhes_preco_quantidade_ao_pereciveis",
    "connection.uri": "mongodb://mongodb:27017",
    "database": "supermercado",
    "collection": "produtos",

```

```

    "key.converter":
    "org.apache.kafka.connect.storage.StringConverter",
    "value.converter":
    "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false"
  }
}


```

Registre o sink-connector:

```

curl -i -X POST -H "Accept:application/json" -H
"Content-Type:application/json" \
  http://localhost:8083/connectors/ -d
@register-mongodb-sink.json

```



```

~/Pos-grad-b/trabalho-03 curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json" \
http://localhost:8083/connectors/ -d @register-mongodb-sink.json

HTTP/1.1 201 Created
Date: Mon, 03 Jun 2024 20:19:09 GMT
Location: http://localhost:8083/connectors/mongodb-sink-connector
Content-Type: application/json
Content-Length: 800
Server: Jetty(9.4.52.v20230823)

{"name":"mongodb-sink-connector","config":{"connector.class":"com.mongodb.kafka.connect.MongoSinkConnector","tasks.max":"1","topics":"dbserver1.public.detalhes_preco_quantidade_pereciveis,dbserver1.public.detalhes_preco_quantidade_nao_pereciveis","connection.uri":"mongodb://mongodb:27017","database":"supermercado","collection":"produtos","key.converter":"org.apache.kafka.connect.storage.StringConverter","value.converter":"org.apache.kafka.connect.json.JsonConverter","value.converter.schemas.enable":"false","document.id.strategy":"com.mongodb.kafka.connect.sink.processor.id.strategy.PartialValueStrategy","document.id.strategy.partial.value.projection.type":"AllowList","document.id.strategy.partial.value.projection.list":["produto_id","name":"mongodb-sink-connector"],"tasks":[],"type":"sink"}}

```

Popular banco de dados Postgres com dados fictícios

Vamos criar um script Python que utiliza a biblioteca faker para gerar dados fictícios e populá-los nas tabelas do PostgreSQL. Para interagir com o PostgreSQL, usaremos a biblioteca psycopg2.

Instalação de bibliotecas necessárias execução do script

```
pip install psycopg2-binary faker
```

Script em python para persistir dados

```

import psycopg2
from faker import Faker
import random
from datetime import datetime, timedelta

```

```
# Configurações de conexão ao PostgreSQL
conn = psycopg2.connect(
    host="localhost",
    database="supermercado",
    user="postgres",
    password="password"
)
cur = conn.cursor()

# Instancia o gerador de dados Faker
fake = Faker()

# Função para criar filiais
def create_filiais(n):
    filiais = []
    for _ in range(n):
        nome = fake.company()
        endereco = fake.street_address()
        cidade = fake.city()
        estado = fake.state()
        cep = fake.zipcode()
        filiais.append((nome, endereco, cidade, estado, cep))
    cur.executemany(
        "INSERT INTO Filiais (nome, endereco, cidade, estado,
cep) VALUES (%s, %s, %s, %s, %s)",
        filiais
    )
    conn.commit()

# Função para criar produtos perecíveis
def create_produtos_pereciveis(n):
    produtos = []
    for _ in range(n):
        nome = fake.word()
        descricao = fake.text()
        preco = round(random.uniform(1.0, 100.0), 2)
        quantidade = random.randint(1, 100)
        data_validade = fake.date_between(start_date="today",
end_date="+1y")
        categoria = fake.word(ext_word_list=['Laticínios',
'Carnes', 'Frutas', 'Verduras', 'Bebidas'])
        filial_id = random.randint(1, num_filiais)
        produtos.append((nome, descricao, preco, quantidade,
data_validade, categoria, filial_id))
```

```
cur.executemany(
    "INSERT INTO Produtos_Pereciveis (nome, descricao,
preco, quantidade, data_validade, categoria, filial_id) VALUES
(%s, %s, %s, %s, %s, %s, %s)",
    produtos
)
conn.commit()

# Função para criar produtos não perecíveis
def create_produtos_nao_pereciveis(n):
    produtos = []
    for _ in range(n):
        nome = fake.word()
        descricao = fake.text()
        preco = round(random.uniform(1.0, 100.0), 2)
        quantidade = random.randint(1, 100)
        categoria = fake.word(ext_word_list=['Grãos',
'Enlatados', 'Bebidas', 'Snacks', 'Condimentos'])
        filial_id = random.randint(1, num_filiais)
        produtos.append((nome, descricao, preco, quantidade,
categoria, filial_id))
    cur.executemany(
        "INSERT INTO Produtos_Nao_Pereciveis (nome, descricao,
preco, quantidade, categoria, filial_id) VALUES (%s, %s, %s,
%s, %s, %s)",
        produtos
    )
    conn.commit()

# Função para criar detalhes de preço e quantidade de produtos
perecíveis
def create_detalhes_preco_quantidade_pereciveis():
    cur.execute("SELECT produto_id, preco, quantidade FROM
Produtos_Pereciveis")
    produtos = cur.fetchall()
    detalhes = [(produto[0], produto[1], produto[2]) for
produto in produtos]
    cur.executemany(
        "INSERT INTO Detalhes_Preco_Quantidade_Pereciveis
(produto_id, preco, quantidade) VALUES (%s, %s, %s)",
        detalhes
    )
    conn.commit()
```



```
# Função para criar detalhes de preço e quantidade de produtos
não perecíveis
def create_detalhes_preco_quantidade_nao_pereciveis():
    cur.execute("SELECT produto_id, preco, quantidade FROM
Produtos_Nao_Pereciveis")
    produtos = cur.fetchall()
    detalhes = [(produto[0], produto[1], produto[2]) for
produto in produtos]
    cur.executemany(
        "INSERT INTO Detalhes_Preco_Quantidade_Nao_Pereciveis
(produto_id, preco, quantidade) VALUES (%s, %s, %s)",
        detalhes
    )
    conn.commit()

# Função para criar detalhes de descrição de produtos
perecíveis
def create_detalhes_descricao_pereciveis():
    cur.execute("SELECT produto_id, descricao, data_validade
FROM Produtos_Pereciveis")
    produtos = cur.fetchall()
    detalhes = [(produto[0], produto[1], produto[2]) for
produto in produtos]
    cur.executemany(
        "INSERT INTO Detalhes_Descricao_Pereciveis
(produto_id, descricao, data_validade) VALUES (%s, %s, %s)",
        detalhes
    )
    conn.commit()

# Função para criar detalhes de descrição de produtos não
perecíveis
def create_detalhes_descricao_nao_pereciveis():
    cur.execute("SELECT produto_id, descricao FROM
Produtos_Nao_Pereciveis")
    produtos = cur.fetchall()
    detalhes = [(produto[0], produto[1]) for produto in
produtos]
    cur.executemany(
        "INSERT INTO Detalhes_Descricao_Nao_Pereciveis
(produto_id, descricao) VALUES (%s, %s)",
        detalhes
    )
    conn.commit()
```

```
# Número de filiais a serem criadas
num_filiais = 100
create_filiais(num_filiais)

# Número de produtos a serem criados em cada categoria
num_produtos_pereciveis = 50000
num_produtos_nao_pereciveis = 50000
create_produtos_pereciveis(num_produtos_pereciveis)
create_produtos_nao_pereciveis(num_produtos_nao_pereciveis)

# Criar detalhes de produtos
create_detalhes_preco_quantidade_pereciveis()
create_detalhes_preco_quantidade_nao_pereciveis()
create_detalhes_descricao_pereciveis()
create_detalhes_descricao_nao_pereciveis()

# Fechar a conexão
cur.close()
conn.close()
```

Dados replicados no MongoDB

O processo de replicação de dados entre PostgreSQL, Kafka e MongoDB é realizado através de dois conectores principais. O primeiro conector, configurado com o Debezium, consome dados do banco de dados PostgreSQL e os envia ao Kafka. Este conector monitora as mudanças em tabelas específicas do PostgreSQL (no nosso caso, **detalhes_preco_quantidade_pereciveis** e **detalhes_preco_quantidade_nao_pereciveis**), e sempre que ocorrem inserções, atualizações ou deleções, essas alterações são capturadas e publicadas como mensagens nos tópicos do Kafka. O segundo conector, configurado como um conector MongoDB Sink, consome as mensagens dos tópicos do Kafka e as insere no MongoDB. Cada mensagem recebida é transformada em um documento MongoDB e inserida na coleção especificada.

Essa arquitetura de replicação de dados oferece diversas vantagens. Primeiramente, a utilização do Kafka como intermediário permite um desacoplamento entre o banco de dados de origem (PostgreSQL) e o banco de

dados de destino (MongoDB), facilitando a escalabilidade e a resiliência do sistema. Além disso, o Kafka atua como um buffer, possibilitando a ingestão contínua de dados mesmo que o MongoDB esteja temporariamente indisponível. A arquitetura também permite a replicação em tempo real das alterações de dados, garantindo que o MongoDB mantenha uma cópia atualizada e consistente do banco de dados PostgreSQL. Isso é especialmente útil para cenários de análise de dados em tempo real, backup e recuperação de dados, e migrações de dados entre sistemas heterogêneos.

Escolhas dos bancos de dados

PostgreSQL com Particionamento Vertical

O particionamento vertical em PostgreSQL foi escolhido para dividir a tabela de produtos em colunas específicas, armazenando detalhes de preços e quantidades separadamente das descrições e outras informações. Isso melhora a performance das consultas que frequentemente acessam apenas uma parte dos dados, como o preço e a quantidade. Além disso, PostgreSQL é um banco de dados relacional robusto e eficiente para lidar com transações ACID, garantindo integridade e consistência dos dados.

MongoDB com Particionamento Horizontal

MongoDB foi escolhido por sua capacidade de lidar com grandes volumes de dados de maneira distribuída e escalável. O particionamento horizontal permite distribuir os dados de diferentes filiais em vários shards, o que melhora a performance e a escalabilidade. MongoDB é ideal para leituras rápidas e flexíveis, além de facilitar a replicação e a distribuição geográfica dos dados.

Atendendo às Premissas do Exercício

Cada filial possui um grande volume de produtos em seu estoque

Com PostgreSQL, utilizamos o particionamento vertical para otimizar o armazenamento e a consulta de dados. MongoDB, com seu particionamento

horizontal, permite a distribuição dos dados em shards, facilitando a gestão de grandes volumes de produtos por filial.

A consulta de estoque e atualizações de inventário devem ser rápidas e eficientes

PostgreSQL garante transações rápidas e consistentes através de índices e particionamento vertical. MongoDB proporciona leituras rápidas devido à sua capacidade de armazenar dados em formato de documento e replicar esses dados em diferentes shards, permitindo acesso rápido e eficiente.

A escalabilidade do sistema é essencial, pois novas filiais podem ser adicionadas no futuro

A utilização de Kafka como intermediário entre PostgreSQL e MongoDB permite um desacoplamento que facilita a escalabilidade. Novas filiais podem ser adicionadas sem impacto significativo na arquitetura existente. O particionamento horizontal de MongoDB garante que o sistema possa escalar horizontalmente, adicionando novos shards conforme necessário para suportar o aumento de dados e de carga de trabalho.

Avaliação de desempenho da estratégia de dados escolhida

Para realizar testes de desempenho que avaliem a eficácia da estratégia de particionamento, vamos criar um script Python que realiza as seguintes operações:

1. **Inserção de Dados:** Inserir um grande volume de dados nas tabelas de produtos perecíveis e não perecíveis no PostgreSQL.
2. **Consultas de Estoque:** Realizar consultas frequentes para verificar os dados de estoque.
3. **Atualizações de Inventário:** Atualizar a quantidade de produtos no estoque.
4. **Medição de Tempo:** Medir o tempo de execução dessas operações para avaliar a performance.

```
import psycopg2
from faker import Faker
import random
import time

# Configurações de conexão ao PostgreSQL
conn = psycopg2.connect(
    host="localhost",
    database="supermercado",
    user="postgres",
    password="password"
)
cur = conn.cursor()

# Instancia o gerador de dados Faker
fake = Faker()

# Função para inserir dados em massa
def insert_massive_data(n):
    produtos_pereciveis = []
    produtos_nao_pereciveis = []
    for _ in range(n):
        # Dados para produtos perecíveis
        nome_p = fake.word()
        descricao_p = fake.text()
        preco_p = round(random.uniform(1.0, 100.0), 2)
        quantidade_p = random.randint(1, 100)
        data_validade_p =
fake.date_between(start_date="today", end_date="+1y")
        categoria_p = fake.word(ext_word_list=['Laticínios',
'Carnes', 'Frutas', 'Verduras', 'Bebidas'])
        filial_id_p = random.randint(1, 10)
        produtos_pereciveis.append((nome_p, descricao_p,
preco_p, quantidade_p, data_validade_p, categoria_p,
filial_id_p))

        # Dados para produtos não perecíveis
        nome_np = fake.word()
        descricao_np = fake.text()
        preco_np = round(random.uniform(1.0, 100.0), 2)
        quantidade_np = random.randint(1, 100)
        categoria_np = fake.word(ext_word_list=['Grãos',
'Enlatados', 'Bebidas', 'Snacks', 'Condimentos'])
        filial_id_np = random.randint(1, 10)
```

```
produtos_ao_pereciveis.append((nome_np, descricao_np,
preco_np, quantidade_np, categoria_np, filial_id_np))
```

```
# Inserindo dados nas tabelas
start_time = time.time()
cur.executemany(
    "INSERT INTO Produtos_Pereciveis (nome, descricao,
preco, quantidade, data_validade, categoria, filial_id) VALUES
(%s, %s, %s, %s, %s, %s, %s)",
    produtos_pereciveis
)
conn.commit()
pereciveis_insert_time = time.time() - start_time

start_time = time.time()
cur.executemany(
    "INSERT INTO Produtos_Nao_Pereciveis (nome, descricao,
preco, quantidade, categoria, filial_id) VALUES (%s, %s, %s,
%s, %s, %s)",
    produtos_ao_pereciveis
)
conn.commit()
ao_pereciveis_insert_time = time.time() - start_time

return pereciveis_insert_time, ao_pereciveis_insert_time
```

Função para realizar consultas de estoque

```
def query_stock(n):
    start_time = time.time()
    for _ in range(n):
        filial_id = random.randint(1, 10)
        cur.execute("SELECT * FROM Produtos_Pereciveis WHERE
filial_id = %s", (filial_id,))
        cur.fetchall()
        pereciveis_query_time = time.time() - start_time

    start_time = time.time()
    for _ in range(n):
        filial_id = random.randint(1, 10)
        cur.execute("SELECT * FROM Produtos_Nao_Pereciveis
WHERE filial_id = %s", (filial_id,))
        cur.fetchall()
        ao_pereciveis_query_time = time.time() - start_time
```

```
return pereciveis_query_time, nao_pereciveis_query_time

# Função para realizar atualizações de inventário
def update_inventory(n):
    start_time = time.time()
    for _ in range(n):
        produto_id = random.randint(1, 10000)
        nova_quantidade = random.randint(1, 200)
        cur.execute("UPDATE Produtos_Pereciveis SET quantidade
= %s WHERE produto_id = %s", (nova_quantidade, produto_id))
        conn.commit()
        pereciveis_update_time = time.time() - start_time

    start_time = time.time()
    for _ in range(n):
        produto_id = random.randint(1, 10000)
        nova_quantidade = random.randint(1, 200)
        cur.execute("UPDATE Produtos_Nao_Pereciveis SET
quantidade = %s WHERE produto_id = %s", (nova_quantidade,
produto_id))
        conn.commit()
        nao_pereciveis_update_time = time.time() - start_time

    return pereciveis_update_time, nao_pereciveis_update_time

# Função principal para executar os testes
def main():
    num_records = 10000
    num_queries = 1000
    num_updates = 1000

    # Inserir dados em massa
    pereciveis_insert_time, nao_pereciveis_insert_time =
insert_massive_data(num_records)
    print(f"Inserção de {num_records} produtos perecíveis:
{pereciveis_insert_time:.2f} segundos")
    print(f"Inserção de {num_records} produtos não perecíveis:
{nao_pereciveis_insert_time:.2f} segundos")

    # Consultar estoque
    pereciveis_query_time, nao_pereciveis_query_time =
query_stock(num_queries)
    print(f"Consulta de estoque {num_queries} vezes para
produtos perecíveis: {pereciveis_query_time:.2f} segundos")
```

```
print(f"Consulta de estoque {num_queries} vezes para  
produtos não perecíveis: {nao_pereciveis_query_time:.2f}  
segundos")  
  
# Atualizar inventário  
pereciveis_update_time, nao_pereciveis_update_time =  
update_inventory(num_updates)  
print(f"Atualização de inventário {num_updates} vezes para  
produtos perecíveis: {pereciveis_update_time:.2f} segundos")  
print(f"Atualização de inventário {num_updates} vezes para  
produtos não perecíveis: {nao_pereciveis_update_time:.2f}  
segundos")  
  
# Executar a função principal  
if __name__ == "__main__":  
    main()  
  
# Fechar a conexão  
cur.close()  
conn.close()
```

Descrição do código

1. Inserção de Dados em Massa:

- A função **insert_massive_data** insere um grande volume de registros nas tabelas **Produtos_Pereciveis** e **Produtos_Nao_Pereciveis**.
- Mede o tempo necessário para inserir os registros e retorna esses tempos.

2. Consultas de Estoque:

- A função **query_stock** realiza consultas frequentes nas tabelas de produtos, filtrando por **filial_id**.
- Mede o tempo total das consultas e retorna esses tempos.

3. Atualizações de Inventário:

- A função **update_inventory** atualiza a quantidade de produtos em várias linhas.
- Mede o tempo necessário para realizar essas atualizações e retorna esses tempos.

4. Adição de Novas Filiais:

- Adiciona novos registros na tabela **Filiais**.
- Mede o tempo necessário para inserir os registros e retorna esse tempo.

5. Execução dos Testes:

A função main executa os três tipos de operações (inserção, consulta e atualização) e imprime os tempos de execução para cada operação.



```
~/doc/pos-grad-/banco-de-dados-relacionais-e-nao-relacionais/trabalho-03 /opt/homebrew/bin/python3.9 /Users/nicholascarballo/Documents/pos-grad-arquitetura-de-dados/b
anco-de-dados-relacionais-e-nao-relacionais/trabalho-03/performance_test.py
Inserção de 10000 produtos perecíveis: 0.83 segundos
Inserção de 10000 produtos não perecíveis: 0.80 segundos
Consulta de estoque 1000 vezes para produtos perecíveis: 12.46 segundos
Consulta de estoque 1000 vezes para produtos não perecíveis: 12.22 segundos
Atualização de inventário 1000 vezes para produtos perecíveis: 0.64 segundos
Atualização de inventário 1000 vezes para produtos não perecíveis: 0.66 segundos
Inserção de 100 novas filiais: 0.06 segundos
```

Vantagens da Arquitetura:

1. Eficiência na Inserção e Consulta de Dados:

O particionamento vertical no PostgreSQL melhora a eficiência das consultas e atualizações frequentes, permitindo acessar apenas os dados necessários.

2. Escalabilidade: A arquitetura usando Kafka como intermediário permite que novos consumidores sejam adicionados facilmente, e o particionamento horizontal no MongoDB permite distribuir a carga de trabalho e armazenamento de forma eficiente.

3. Desacoplamento e Resiliência: Kafka proporciona um buffer que desacopla os sistemas de origem e destino, garantindo que dados possam ser processados de forma assíncrona, aumentando a resiliência do sistema.

4. Replicação em Tempo Real: A replicação em tempo real através do Debezium e Kafka assegura que o MongoDB esteja sempre atualizado com as mudanças no PostgreSQL, permitindo análises em tempo real e outras operações dependentes de dados atualizados.

Estas operações permitem avaliar a eficácia da estratégia de particionamento vertical e horizontal, garantindo que o sistema possa lidar com grandes volumes de dados e escalar eficientemente conforme novas filiais são adicionadas.

Pontos de atenção

Complexidades de Utilizar uma Arquitetura de Replicação com Kafka e Debezium

Gerenciamento de Volume de Dados

Desafios: Kafka pode enfrentar problemas de desempenho com grandes volumes de dados e alta taxa de transferência.

Soluções: Configuração adequada de tópicos, partições e replicação; uso de compactação de logs e políticas de retenção de dados.

Consistência de Dados

Desafios: Garantir a consistência de dados em sistemas distribuídos pode ser complicado.

Soluções: Implementação de estratégias de entrega garantida, como Exactly-Once Semantics (EOS), e monitoramento rigoroso dos conectores.

Manutenção e Monitoramento

Desafios: Manter e monitorar clusters Kafka e pipelines de Debezium requer ferramentas especializadas e expertise.

Soluções: Utilização de ferramentas de monitoramento como Prometheus, Grafana, e Kafka Manager; automação de tarefas de manutenção com Ansible.

Escalabilidade e Disponibilidade

Desafios: Garantir que o sistema escale eficientemente e mantenha alta disponibilidade.

Soluções: Implementação de clusters de Kafka e MongoDB replicados, uso de balanceamento de carga e failover automatizado.

Conclusão

A combinação de PostgreSQL com particionamento vertical e MongoDB com particionamento horizontal, intermediada pelo Kafka, proporciona um sistema robusto, escalável e eficiente para a gestão de estoque de uma cadeia de supermercados. Essa arquitetura não só atende às necessidades atuais de volume de dados e performance, mas também está preparada para futuras expansões e integrações.