Solução do Desafio | Bolsista de Desenvolvimento Suporte

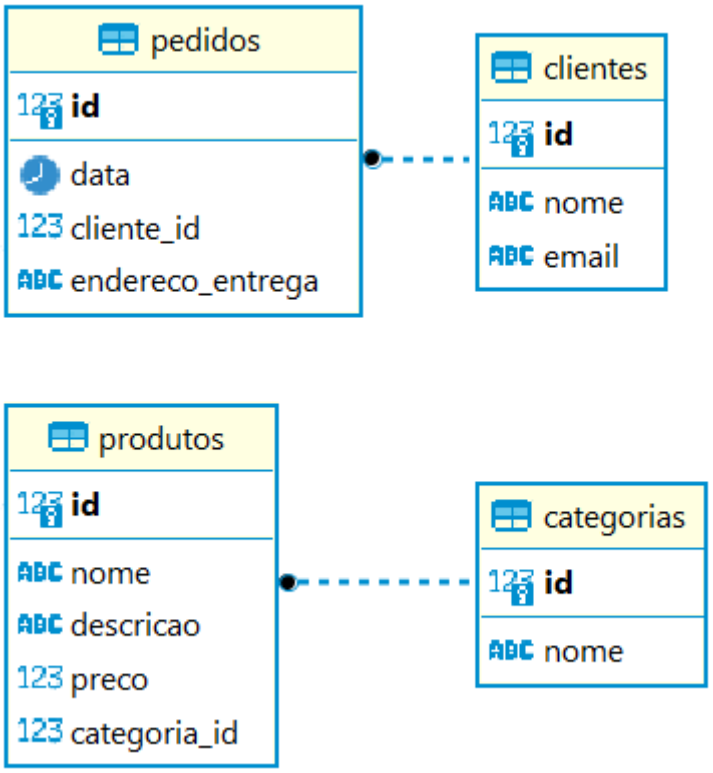
Autor: Felipe Macacari Pierotti

Introdução

Esse é um breve relatório que descreve o processo de solução do desafio proposto durante o processo seletivo. O desafio em questão, envolve a modelagem de um banco de dados relacional para o armazenamento de informações de uma loja online. Portanto, a proposta é a modelagem e criação do esquema, assim como a criação de consultas para verificar a integridade dos dados.

Modelagem do esquema

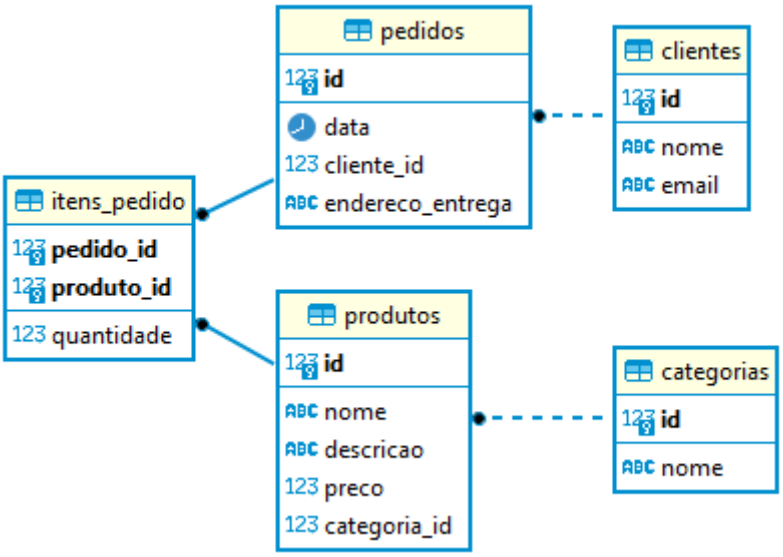
Tendo em mente que vamos abordar uma loja online e seguindo as consultas propostas no desafio, podemos encontrar algumas entidades principais como: Produto, Categoria, Cliente e Pedido. A seguir eu apresento um diagrama com uma tabela para cada entidade citada e os relacionamentos entre elas.



Podemos observar que “produtos” tem um relacionamento *many to one* (vários para um) com “categorias”, isso porque vários produtos serão pertencentes à uma mesma categoria da loja, o mesmo acontece entre “pedidos” e “clientes”. Seguindo essa lógica, precisamos de uma relação de muitos produtos para um mesmo pedido, e esse pedido pode ter vários produtos. Portanto, a solução é a criação de uma tabela pivô “itens_pedido”

que faça o relacionamento entre vários “produtos” e vários “pedidos” (*many to many*), e também, armazena a quantidade de cada produto em um pedido.

Segue o diagrama atualizado com a nova tabela:



Populando as tabelas

Seguindo as orientações do desafio, precisamos de no mínimo 50 registros em cada tabela e que essa seja uma quantidade alterável em nosso script, que deve receber essa quantidade como parâmetro.

Como sugerido, os dados seguirão um padrão como “Produto1”, “Produto2”, “Produto…” e assim por diante de acordo com cada tabela. Para isso, faremos o uso da função “generate_series” que retorna uma sequência de valores em um intervalo definido, e vamos concatenar esse valor retornado com uma string “Produto” por exemplo.

Categorias

Portanto, essa é a rotina para popular os dados na tabela de “categorias”:

```
INSERT INTO Categorias (nome)
SELECT 'Categoria' || i
FROM generate_series(1, 50) AS i;
```

O que será responsável por gerar 50 colunas na nossa tabela em sequência de “Categoria1” até “Categoria50”.

Produtos

Para a tabela de “produtos”, temos a seguinte rotina:

```
INSERT INTO Produtos (nome, descricao, preco, categoria_id)
SELECT 'Produto' || i,
       'Descriçao produto' || i,
       (i + 0.99)::decimal(10, 2),
       (i % 50) + 1
FROM generate_series(1, 50) AS i;
```

Entre as novidades aqui, está a coluna de “preço”, que irá formar um decimal também em sequência de acordo com o número gerado pela “generate_series”, e também a coluna de “categoria_id” que recebe um inteiro também até 50.

Clientes

A rotina para “clientes” apenas gera os textos em sequência:

```
INSERT INTO Clientes (nome, email)
SELECT 'Cliente' || i, 'cliente' || i || '@email.com'
FROM generate_series(1, 50) AS i;
```

Pedidos

Tabela de “pedidos”:

```
INSERT INTO Pedidos (data, endereco_entrega, cliente_id)
SELECT now() - (i * interval '1 hour'), 'Endereco' || i, (i % 50) + 1
FROM generate_series(1, 50) AS i;
```

Aqui, vale ressaltar o caso da coluna “data”, em que foi utilizado uma combinação da hora atual, menos, um intervalo de *i* horas de acordo com o número atual da sequência [1].

Itens_pedido

Como é esperado que cada pedido possa ter mais de um produto, é importante popular a tabela com mais de um produto por pedido e garantir que irá funcionar como o esperado. Para isso, a solução foi a seguinte:

```
INSERT INTO Itens_pedido (pedido_id, produto_id, quantidade)
SELECT (i % 50) + 1, (i % 47) + p, (i % 5) + 1
FROM generate_series(1, 50) AS i,
     generate_series(1, 3) AS p;
```

Esta solução foi montada para que cada pedido tenha três produtos com quantidades quaisquer. Isso foi feito através de uma sequência que percorre todos os pedidos e uma nova sequência de 1 a 3 que acontece para cada item da sequência anterior. Além disso, vale pontuar que teremos os mesmos produtos em diferentes pedidos e em quantidades diferentes.

Como o desafio impos que a quantidade de registros gerados deve ser controlada a partir de parâmetros, vamos prosseguir com a criação de uma *procedure* que representa um conjunto de blocos de execução SQL. Com a nossa *procedure* podemos definir variáveis que recebem os parâmetros esperados e facilita a chamada no momento de popular as tabelas. Foi especificado também que esse parâmetro deve ser variável entre as tabelas, dessa forma, foi criado uma *procedure* para cada tabela.

```
CREATE OR REPLACE PROCEDURE PopularCategorias (
    Qnt INTEGER
)
LANGUAGE SQL
AS $$
    -- Inserindo dados de Categorias
    INSERT INTO Categorias (nome)
    SELECT 'Categoria' || i
    FROM generate_series(1, Qnt) AS i;
$$;

CALL POPULARCATEGORIAS(50)
```

Esse é um exemplo da criação e chamada da *procedure* feita para a tabela “categorias”, para todas as outras tabelas foi seguido o mesmo padrão de nomenclatura “Popular(nome da tabela)” e todas recebem pelo menos um parâmetro que representa quantidade de registros que serão criados.

Consultas do desafio

Listar todos os produtos com nome, descrição e preço em ordem alfabética crescente:

É uma consulta simples e que reúne os dados da tabela de “produto” com exceção do campo de categoria e deve listar em ordem alfabética. Portanto a consulta é a seguinte:

```
SELECT
    p.id,
    p.nome,
    p.descricao,
    p.preco
FROM Produtos AS p
ORDER BY p.nome ASC
```

Listar todas as categorias com nome e número de produtos associados, em ordem alfabética crescente:

Precisamos listar todas as categorias e a quantidade de produtos associados, portanto é importante lembrar que podemos ter categorias sem produtos, por isso a escolha de um “LEFT JOIN” que irá trazer as categorias de forma independente. A consulta:

```sql
SELECT
    c.id,
    c.nome,
    COUNT(p.id) AS qnt_produtos
FROM Categorias AS c
    LEFT JOIN Produtos AS p ON (p.categoria_id = c.id)
GROUP BY
    c.id,
    c.nome
ORDER BY c.nome
```

Listar todos os pedidos com data, endereço de entrega e total do pedido (soma dos preços dos itens), em ordem decrescente de data:

Aqui nós vamos utilizar a nossa tabela pivô de “Itens_pedido” e fazer um “INNER JOIN” com as tabelas de “Produtos” e “Pedidos” para completar as informações necessárias. Como o enunciado informa, devemos trazer o valor total de cada pedido, ou seja, precisamos somar o valor total de cada produto, que é representado pelo valor do produto multiplicado pela quantidade pedida. Portanto, utilizando da função “SUM”, essa é a consulta final:

```sql
SELECT
    ped.id,
    ped.data,
    ped.endereco_entrega,
    SUM(pro.preco * ip.quantidade) AS total
FROM Itens_pedido AS ip
    INNER JOIN Pedidos AS ped ON (ip.pedido_id = ped.id)
    INNER JOIN Produtos AS pro ON (ip.produto_id = pro.id)
GROUP BY
    ped.id,
    ped.data,
    ped.endereco_entrega
```

Listar todos os produtos que já foram vendidos em pelo menos um pedido, com nome, descrição, preço e quantidade total vendida, em ordem decrescente de quantidade total vendida:

Aqui podemos fazer o uso da nossa tabela pivô novamente, visto que ela representa cada vez que um produto é vendido, vamos usar novamente a “SUM” para somar a quantidade vendida em cada pedido:

```sql
SELECT
    p.id,
    p.nome,
    p.descricao,
    p.preco,
    SUM(ip.quantidade) AS quantidade_total
FROM Itens_pedido ip
INNER JOIN Produtos AS p ON (p.id = ip.produto_id)
GROUP BY
    p.id
ORDER BY quantidade_total DESC
```

Listar todos os pedidos feitos por um determinado cliente, filtrando-os por um determinado período, em ordem alfabética crescente do nome do cliente e ordem crescente da data do pedido:

Ficou uma dúvida nesse enunciado, se a consulta deveria receber também um filtro por cliente, quando lemos “pedidos feitos por um determinado cliente” ou se ela deve trazer todos os clientes quando lemos “em ordem alfabética crescente do nome do cliente”. Foi escolhido trazer os dados de todos os clientes e a consulta, com o filtro de data, ficou dessa maneira:

```sql
SELECT
    c.id AS id_cliente,
    c.nome AS nome_cliente,
    c.email AS email_cliente,
    p.id AS id_pedido,
    p.data,
    p.endereco_entrega
FROM Clientes c
INNER JOIN Pedidos AS p ON (p.cliente_id = c.id)
WHERE
    '2023-03-25' <= p.data AND p.data <= '2023-03-29'
ORDER BY c.nome ASC,
    p.DATA ASC
```

Listar possíveis produtos com nome replicado e a quantidade de replicações, em ordem decrescente de quantidade de replicações:

Para encontrar os produtos com o nome repetido, podemos realizar um “SELECT” na nossa tabela e agrupar os registros pelo campo “nome”. Isso nos retornaria todos os produtos, mesmo que repetidos, apenas uma vez. Portanto, vamos retornar o resultado da função “COUNT” que irá contar a quantidade de cada repetição e posteriormente usar a cláusula “HAVING” para condicionar os grupos criados. Nesse caso, vamos criar uma condição de que o valor de “COUNT” seja maior do que 1, ou seja, o registro apareceu mais de uma vez. Consulta:

```sql
SELECT
    p.nome,
    (COUNT(*) - 1) AS replicacoes
FROM Produtos p
GROUP BY p.nome
HAVING COUNT(*) > 1
ORDER BY replicacoes DESC
```

A escolha do “COUNT(*) - 1” foi feita para trazer apenas o número de vezes em que o registro está repetido.

[1] - How to create (lots!) of sample time-series data with PostgreSQL generate_series()