Exemplo Prático de CRUD, Consultas Avançadas, Transações e PL/pgSQL no PostgreSQL (para pgAdmin)

Vamos criar um exemplo prático que envolva uma tabela de Produtos e uma tabela de Pedidos para ilustrar todos os conceitos que você mencionou. Você poderá executar esses comandos diretamente no pgAdmin.

1. Criando as Tabelas:

Abra uma janela de Query no pgAdmin e execute os seguintes comandos para criar as tabelas:

SQL

```sql
-- Criar a tabela de Produtos
CREATE TABLE Produtos (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL UNIQUE,
    preco DECIMAL(10, 2) NOT NULL,
    estoque INTEGER NOT NULL DEFAULT 0,
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Criar a tabela de Pedidos
CREATE TABLE Pedidos (
    id SERIAL PRIMARY KEY,
    cliente_nome VARCHAR(100) NOT NULL,
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total DECIMAL(10, 2) DEFAULT 0
);

-- Criar a tabela de itens do pedido (relacionamento muitos-para-muitos)
CREATE TABLE ItensPedido (
    pedido_id INTEGER REFERENCES Pedidos(id),
    produto_id INTEGER REFERENCES Produtos(id),
    quantidade INTEGER NOT NULL DEFAULT 1,
    preco_unitario DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (pedido_id, produto_id)
```

);

2. INSERT (Criar Dados):

Execute os seguintes comandos para inserir alguns dados nas tabelas:

SQL

```
-- Inserir produtos
INSERT INTO Produtos (nome, preco, estoque) VALUES ('Notebook', 1200.50, 10);
INSERT INTO Produtos (nome, preco, estoque) VALUES ('Mouse', 25.99, 50);
INSERT INTO Produtos (nome, preco, estoque) VALUES ('Teclado', 79.90, 30);
INSERT INTO Produtos (nome, preco, estoque) VALUES ('Monitor', 350.00, 15);

-- Inserir pedidos
INSERT INTO Pedidos (cliente_nome) VALUES ('João Silva');
INSERT INTO Pedidos (cliente_nome) VALUES ('Maria Oliveira');

-- Inserir itens nos pedidos
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (1, 1, 1, 1200.50);
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (1, 2, 2, 25.99);
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (2, 3, 1, 79.90);
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (2, 4, 1, 350.00);

-- Atualizar o total dos pedidos (exemplo simples)
UPDATE Pedidos SET total = (SELECT SUM(quantidade * preco_unitario) FROM ItensPedido WHERE pedido_id = 1) WHERE id = 1;
UPDATE Pedidos SET total = (SELECT SUM(quantidade * preco_unitario) FROM ItensPedido WHERE pedido_id = 2) WHERE id = 2;
```

3. READ (Ler Dados):

Execute as seguintes consultas para ler dados das tabelas:

SQL

```sql
-- Selecionar todos os produtos
SELECT * FROM Produtos;

-- Selecionar pedidos com o nome do cliente e o total
SELECT id, cliente_nome, total FROM Pedidos;

-- Usando JOIN para listar os itens de um pedido específico (ID = 1) com o nome do produto
SELECT p.id AS pedido_id, pr.nome AS produto_nome, ip.quantidade, ip.preco_unitario
FROM Pedidos p
JOIN ItensPedido ip ON p.id = ip.pedido_id
JOIN Produtos pr ON ip.produto_id = pr.id
WHERE p.id = 1;

-- Usando subconsulta para encontrar produtos com preço acima da média
SELECT nome, preco
FROM Produtos
WHERE preco > (SELECT AVG(preco) FROM Produtos);

-- Usando função agregada para contar o número de produtos em estoque
SELECT SUM(estoque) AS total_estoque FROM Produtos;
```

4. UPDATE (Atualizar Dados):

Execute os seguintes comandos para atualizar dados nas tabelas:

SQL

```sql
-- Atualizar o estoque de um produto (ID = 2)
UPDATE Produtos SET estoque = 45 WHERE id = 2;

-- Atualizar o preço de um produto (Nome = 'Teclado')
UPDATE Produtos SET preco = 89.99 WHERE nome = 'Teclado';
```

5. DELETE (Excluir Dados):

Execute os seguintes comandos para excluir dados das tabelas:

SQL

```
-- Excluir um item de um pedido (Pedido ID = 1, Produto ID = 2)
DELETE FROM ItensPedido WHERE pedido_id = 1 AND produto_id = 2;

-- Excluir um pedido (ID = 2) - Observe que isso pode gerar erro se houver restrições ON DELETE CASCADE não configuradas
DELETE FROM Pedidos WHERE id = 2;

-- Excluir um produto (Nome = 'Monitor')
DELETE FROM Produtos WHERE nome = 'Monitor';
```

6. Transações e Controle de Concorrência:

Execute o seguinte bloco de comandos para simular uma transação:

SQL

```
-- Iniciar uma transação
BEGIN;

-- Atualizar o estoque e inserir um item de pedido (tudo dentro da mesma transação)
UPDATE Produtos SET estoque = estoque - 1 WHERE id = 1;
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (1, 1, 1, (SELECT preco FROM Produtos WHERE id = 1));
UPDATE Pedidos SET total = total + (SELECT preco FROM Produtos WHERE id = 1) WHERE id = 1;

-- Simular um erro (comente a linha abaixo para commitar)
-- RAISE EXCEPTION 'Erro simulado!';

-- Confirmar a transação se não houver erro
```

```
COMMIT;

-- Ou, para desfazer as alterações em caso de erro
-- ROLLBACK;

-- Verifique os resultados
SELECT * FROM Produtos WHERE id = 1;
SELECT * FROM Pedidos WHERE id = 1;
SELECT * FROM ItensPedido WHERE pedido_id = 1 AND produto_id = 1;
```

Observações sobre Concorrência:

- O PostgreSQL usa um sistema de controle de concorrência MVCC (Multi-Version Concurrency Control) que permite que múltiplas transações leiam e escrevam no banco de dados simultaneamente sem bloquear umas às outras na maioria dos casos.
- Para cenários de alta concorrência com muitas atualizações no mesmo conjunto de dados, você pode precisar considerar níveis de isolamento de transação (padrão é READ COMMITTED) e usar mecanismos de bloqueio explícito (LOCK TABLE, SELECT ... FOR UPDATE) se necessário para garantir a integridade dos dados.

7. Otimização de Consultas Específicas do PostgreSQL:

- **EXPLAIN:** Use o comando EXPLAIN seguido de uma consulta para ver o plano de execução da consulta. Isso ajuda a identificar gargalos e oportunidades de otimização.

```
SQL

EXPLAIN SELECT nome FROM Produtos WHERE preco > 100;
```

- **Índices:** Crie índices nas colunas frequentemente usadas em cláusulas WHERE e JOIN para acelerar as consultas.

```
SQL
```

```
CREATE INDEX idx_produto_nome ON Produtos (nome);
CREATE INDEX idx_itenspedido_pedido_id ON ItensPedido (pedido_id);
CREATE INDEX idx_itenspedido_produto_id ON ItensPedido (produto_id);
```

- **ANALYZE:** Execute o comando ANALYZE para atualizar as estatísticas internas usadas pelo otimizador de consultas.

SQL

```
ANALYZE Produtos;
ANALYZE Pedidos;
ANALYZE ItensPedido;
```

- **Particionamento:** Para tabelas muito grandes, considere o particionamento para melhorar o desempenho de consultas e a gestão dos dados.
- **Tipos de Dados Específicos:** Use os tipos de dados avançados do PostgreSQL (JSONB, arrays, etc.) de forma eficiente para otimizar o armazenamento e a consulta de dados complexos.

8. PL/pgSQL Básico (Stored Procedure, Function e Trigger):

- **Stored Procedure (Procedure):**

SQL

```
CREATE OR REPLACE PROCEDURE atualizar_estoque(produto_nome VARCHAR,
quantidade_alteracao INTEGER)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE Produtos
    SET estoque = estoque + quantidade_alteracao
    WHERE nome = produto_nome;
    RAISE NOTICE 'Estoque do produto % atualizado em % unidades', produto_nome,
```

```
quantidade_alteracao;
END;
$$;

-- Chamar a stored procedure
CALL atualizar_estoque('Mouse', -5);
SELECT * FROM Produtos WHERE nome = 'Mouse';
```

- **Function (Função que retorna um valor):**

```
SQL


CREATE OR REPLACE FUNCTION calcular_total_pedido(pedido_id INTEGER)
RETURNS DECIMAL
LANGUAGE plpgsql
AS $$
DECLARE
    total_pedido DECIMAL;
BEGIN
    SELECT SUM(quantidade * preco_unitario) INTO total_pedido
    FROM ItensPedido
    WHERE pedido_id = calcular_total_pedido.pedido_id;
    RETURN total_pedido;
END;
$$;

-- Chamar a função
SELECT calcular_total_pedido(1);
UPDATE Pedidos SET total = calcular_total_pedido(id) WHERE id = 1;
SELECT * FROM Pedidos WHERE id = 1;
```

- **Trigger (Gatilho que executa uma ação em resposta a um evento):**

```
SQL
```

```
CREATE OR REPLACE FUNCTION atualizar_total_pedido_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
   UPDATE Pedidos
   SET total = (SELECT SUM(quantidade * preco_unitario) FROM ItensPedido WHERE
pedido_id = NEW.pedido_id)
   WHERE id = NEW.pedido_id;
   RETURN NEW;
END;
$$;

CREATE TRIGGER atualizar_total_pedido_apos_item
AFTER INSERT OR UPDATE OR DELETE ON ItensPedido
FOR EACH ROW
EXECUTE PROCEDURE atualizar_total_pedido_trigger();

-- Testar o trigger
INSERT INTO ItensPedido (pedido_id, produto_id, quantidade, preco_unitario) VALUES (1,
3, 2, 79.90);
SELECT * FROM Pedidos WHERE id = 1;

UPDATE ItensPedido SET quantidade = 3 WHERE pedido_id = 1 AND produto_id = 3;
SELECT * FROM Pedidos WHERE id = 1;

DELETE FROM ItensPedido WHERE pedido_id = 1 AND produto_id = 3;
SELECT * FROM Pedidos WHERE id = 1;
```

Como usar no pgAdmin:

1. Abra o pgAdmin.
2. Conecte-se ao seu servidor e banco de dados PostgreSQL.
3. Clique com o botão direito no banco de dados ou em um schema específico (como public).
4. Selecione "Query Tool".
5. Cole e execute os comandos SQL em sequência. Você pode executar blocos de comandos selecionando-os e pressionando F5 ou o botão "Execute".
6. Para criar Stored Procedures, Functions e Triggers, você também pode usar a interface gráfica do pgAdmin navegando na árvore de objetos (Database ->

Schemas -> public -> Functions/Procedures/Triggers) e clicando com o botão direito para criar novos objetos.

Este exemplo abrangente deve fornecer uma base sólida para você começar a explorar os conceitos de CRUD, consultas avançadas, transações, otimização e PL/pgSQL no PostgreSQL usando o pgAdmin. Experimente executar os comandos, modificar os dados e analisar os resultados para um melhor entendimento.