1. **Procedimentos armazenados** (ou stored procedures) são blocos de código SQL que podem ser salvos no banco de dados e executados sob demanda. No PostgreSQL, eles são usados para encapsular operações que envolvem várias instruções SQL, simplificando a repetição de processos complexos e melhorando a organização do código.

   Características dos procedimentos armazenados:

- **Armazenamento no servidor**: O código fica armazenado no SGBD e pode ser reutilizado várias vezes.
- **Parâmetros**: Podem receber e retornar parâmetros *(IN, OUT, INOUT)*.
- **Controle de fluxo**: Suportam estruturas de controle como *`IF`, `WHILE`, `LOOP`.*
- **Performance**: Podem melhorar a performance ao reduzir o tráfego entre o cliente e o servidor.

Diferença entre funções e procedimentos no PostgreSQL

- **Funções**: Retornam valores e podem ser usadas em consultas SQL.
- **Procedimentos**: Não precisam retornar valores e são invocadas usando `CALL`, não sendo usadas diretamente em consultas.

Como criar um procedimento armazenado no PostgreSQL

A partir do PostgreSQL 11, foi introduzido o suporte a procedimentos com a palavra-chave `CALL`. Veja como criar um:

1. **Sintaxe básica:**

```
CREATE PROCEDURE nome_procedimento (param1 tipo1, param2 tipo2, ...)

LANGUAGE plpgsql

AS $$

BEGIN

    -- corpo do procedimento

END;

$$;
```

2. **Exemplo de procedimento que insere um novo registro em uma tabela:**

   Suponha que temos uma tabela chamada `usuarios` com as colunas `nome` e `email`:

```
CREATE PROCEDURE inserir_usuario(nome_usuario TEXT, email_usuario TEXT)

LANGUAGE plpgsql

AS $$

BEGIN

    INSERT INTO usuarios (nome, email) VALUES (nome_usuario, email_usuario);

END;

$$;
```

   Esse procedimento insere um novo usuário na tabela `usuarios` quando chamado.

3. **Chamar um procedimento:**

   Usa-se a instrução `CALL` para executar o procedimento:

```
CALL inserir_usuario('João Silva', 'joao.silva@email.com');
```

Procedimentos com controle de fluxo

Também podemos adicionar lógica condicional, como no exemplo abaixo:

```
CREATE PROCEDURE atualizar_email_usuario(id_usuario INT, novo_email TEXT)

LANGUAGE plpgsql

AS $$

BEGIN
```

```
    IF EXISTS (SELECT 1 FROM usuarios WHERE id = id_usuario)
THEN

        UPDATE usuarios SET email = novo_email WHERE id =
id_usuario;

    ELSE

        RAISE NOTICE 'Usuário não encontrado';

    END IF;

END;

$$;
```

Este procedimento atualiza o e-mail de um usuário, mas só se ele existir.

Vantagens dos Procedimentos Armazenados:

- **Redução do tráfego**: Minimiza o número de interações cliente-servidor.
- **Reuso de código**: Encapsula lógica complexa em um único lugar.
- **Segurança**: Permite controlar permissões específicas para certos procedimentos.
- **Modularidade**: Organiza melhor a lógica do banco de dados.

Desvantagens:

- **Portabilidade**: Procedimentos são dependentes do SGBD, o que pode dificultar migrações.
- **Debugging**: Debug de procedimentos armazenados pode ser mais complexo do que em código de aplicação.

Com esses conceitos, você pode criar procedimentos que facilitam a gestão e automatizam processos no seu banco de dados PostgreSQL.

Atividades

1. Criar a tabela de usuários com id, email e nome. Criar um procedimento armazenado para inserir novos usuários.
2. Criar um procedimento para inserir itens de venda. O procedimento deve verificar se existe estoque e após a inserção, atualizar o estoque do item

2. Triggers o que são?

Triggers são mecanismos em sistemas de banco de dados que permitem executar uma ação automaticamente em resposta a eventos específicos que ocorrem em uma tabela. No PostgreSQL, os gatilhos (triggers) podem ser acionados por eventos como **INSERT**, **UPDATE**, **DELETE**, ou até mesmo **TRUNCATE**. Eles são úteis para manter a integridade dos dados, automatizar processos, gerar logs, e implementar regras de negócio diretamente no banco de dados.

Como funcionam os Triggers no PostgreSQL?

1. **Eventos**: O evento que aciona o trigger, como uma inserção, atualização ou exclusão.
2. **Momento**: O trigger pode ser executado antes ou depois do evento ocorrer (**BEFORE** ou **AFTER**).
3. **Ação**: A ação que será executada, geralmente uma função que contém o código a ser executado.

Estrutura básica de um Trigger no PostgreSQL

1. **Função de Trigger**: É uma função que define o que será feito quando o trigger for acionado.
2. **Definição do Trigger**: Vincula a função a um evento e a uma tabela específica.

Exemplo de Implementação

1. Criando a função do trigger

```
CREATE OR REPLACE FUNCTION verificar_saldo_minimo()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.saldo < 0 THEN
        RAISE EXCEPTION 'O saldo não pode ser negativo.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Essa função verifica se o saldo de uma conta (neste exemplo) é negativo. Se for, lança uma exceção.

2. Criando o trigger

```
CREATE TRIGGER trigger_verificar_saldo
BEFORE INSERT OR UPDATE ON contas
FOR EACH ROW
EXECUTE FUNCTION verificar_saldo_minimo();
```

Aqui estamos dizendo que, antes de cada inserção ou atualização na tabela `contas`, o PostgreSQL deve executar a função **`verificar_saldo_minimo()`**.

Tipos de Eventos que Podem Acionar Triggers

- **INSERT**: Acionado quando uma nova linha é inserida.
- **UPDATE**: Acionado quando uma linha existente é atualizada.
- **DELETE**: Acionado quando uma linha é excluída.
- **TRUNCATE**: Acionado quando a tabela é truncada, removendo todas as linhas.

Momentos de Execução

- **BEFORE**: O trigger é executado antes do evento ocorrer. Pode ser útil para validar dados ou modificar valores antes que eles sejam salvos no banco.
- **AFTER**: O trigger é executado depois que o evento ocorre, geralmente usado para auditoria ou outros processos pós-salvamento.

Exemplo de um Trigger AFTER INSERT

Esse exemplo grava um log sempre que uma nova conta é criada:

1. **Função do Trigger**:

```
CREATE OR REPLACE FUNCTION log_insercao_conta()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO log_contas (id_conta, data_insercao)
    VALUES (NEW.id, NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

2. **Trigger**:

```
CREATE TRIGGER trigger_log_insercao
AFTER INSERT ON contas
```

```
FOR EACH ROW
EXECUTE FUNCTION log_insercao_conta();
```

Nesse exemplo, toda vez que uma nova linha for inserida na tabela `contas`, o sistema grava uma entrada na tabela `log_contas`, contendo o ID da conta e a data de inserção.

Considerações

- Triggers podem impactar a performance do banco de dados, especialmente quando envolvem operações complexas em grandes volumes de dados.
- É importante testar o comportamento dos triggers para garantir que não causem efeitos indesejados ou erros lógicos no sistema.