Estudos SQL

Dentro deste documento estou listando todo meu aprendizado sobre a linguagem SQL e seus bancos de dados (MySQL, PostgreSQL). Em alguns tópicos que forem diferentes de um banco para o outro eu irei explicar.

Depois de instalar e todas as informações, faça login com sua senha definida e clica em DATABASE > create > set name ”defina um nome pra ele” , e depois use o comando :

```
use nome_do_database
```

Exemplos de como criar uma nova tabela e acessar elementos.

Lembrando que o ; é necessário em algumas atribuições.

Principais classificações de comando

DDL = Data Definition Language. É com ele que os elementos estruturais são definidos

Comandos: CREATE, DROP, ALTER e TRUNCATE

DML = Data Manipulation Language. Usada pra manipular dados/registros em BD

Comandos: INSERT, DELETE e UPDATE

DTL = Data Transaction Language. É a capacidade de agrupar comandos de transações de forma que, se algo der errado, possa voltar e recuperar.

Comandos: BEGIN TRANSACTION, ROLLBACK, COMMIT

DCL = Data Control Language. Controle de segurança de banco de dados

Comandos: GRANT, REVOKE e DENY

DATA QUERY LANGUAGE. Utilizado pra consulta de dados

comando: SELECT

Exemplo pra criação de tabela:

```
CREATE TABLE usuarios(

id SERIAL PRIMARY KEY,

nome VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

idade INT,

data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);
```

Assim você cria sua primeira tabela. Elementos como SERIAL significam que ele irá criar automaticamente começando pelo 1, depois 2....

Lembrando que o SERIAL só funciona no PostgreSQL.

Pra MySQL ele utiliza o AUTO_INCREMENT.

NOT NULL - Significa que não pode ser nulo

UNIQUE - Significa que não pode ser igual a um já existente

Utilizando método CASCADE

Quando criamos uma tabela com uma chave estrangeira, podemos definir ações como :

```
ON DELETE CASCADE
ou
 ON UPDATE CASCADE.
```

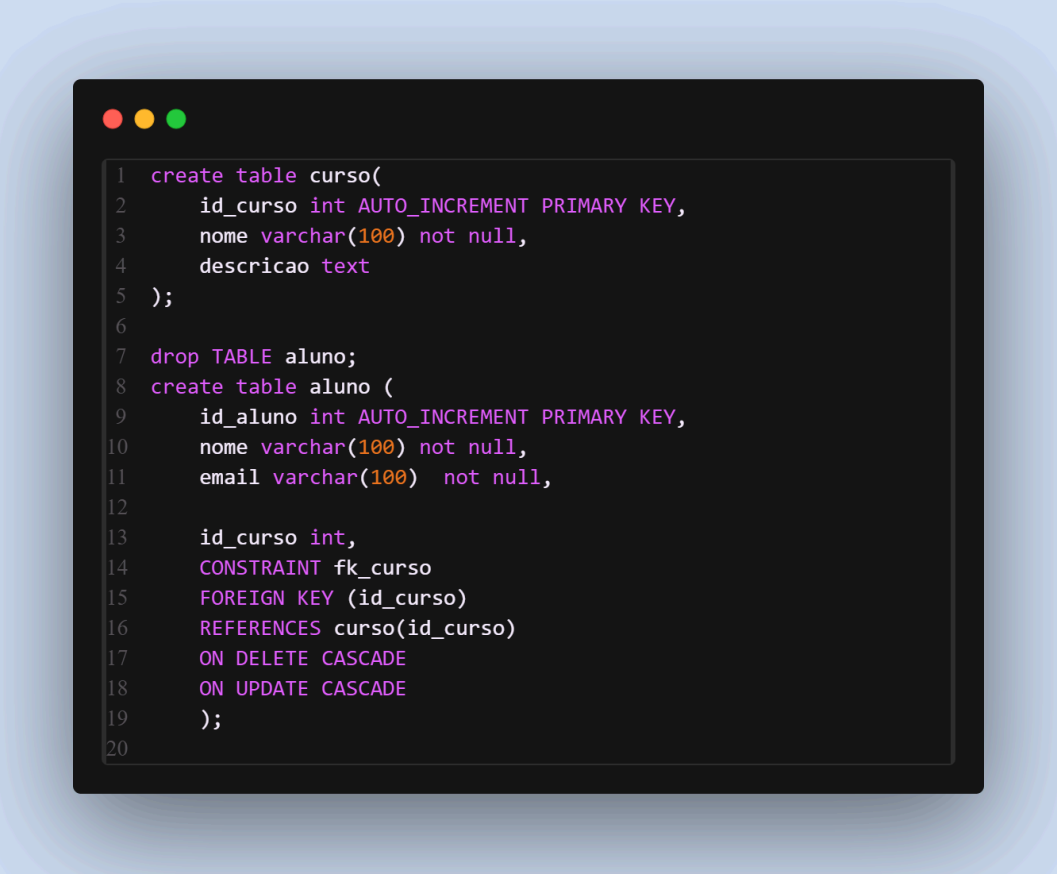
O CASCADE garante a integridade referencial automaticamente.

- **ON DELETE CASCADE:** se um registro da tabela pai for removido, todos os registros relacionados na tabela filha também serão removidos.

• **ON UPDATE CASCADE:** se o valor da chave primária da tabela pai for atualizado, o valor correspondente na tabela filha também será atualizado.

- ON DELETE SET NULL altera o valor para NULL
- ON DELETE NO ACTION = (comportamento padrão do PostgreSQL)

Veja neste exemplo:



```sql
create table curso(
    id_curso int AUTO_INCREMENT PRIMARY KEY,
    nome varchar(100) not null,
    descricao text
);

drop TABLE aluno;
create table aluno (
    id_aluno int AUTO_INCREMENT PRIMARY KEY,
    nome varchar(100) not null,
    email varchar(100)  not null,

    id_curso int,
    CONSTRAINT fk_curso
    FOREIGN KEY (id_curso)
    REFERENCES curso(id_curso)
    ON DELETE CASCADE
    ON UPDATE CASCADE
    );

```

Utilizamos o CONSTRAINT pra criar uma restrição/método de atualização de informação. Caso eu mude de uma tabela, ele irá atualizar ou deletar as informações em ambas, para que não tenha diferença nos dois.

Restringir deletes ou Updates

ON DELETE RESTRICT

Impede que um registro da tabela pai seja deletado caso existam registros relacionados na tabela filha.Ou seja,se houver dependências, o DELETE é bloqueado

```
CREATE TABLE clientes (
    id SERIAL PRIMARY KEY,
    nome VARCHAR(50)
);

CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INT REFERENCES clientes(id) ON DELETE RESTRICT,
    valor DECIMAL(10,2)
);
```

caso tente excluir a informação da tabela pai ele sera impedido caso ainda haja informação na tabela filha associado a ele.

ON UPDATE RESTRICT

Impede atualizar a chave primária da tabela pai quando houver registros dependentes na tabela filha.

```
CREATE TABLE clientes (
    id INT PRIMARY KEY,
    nome VARCHAR(50)
);

CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    cliente_id INT REFERENCES clientes(id) ON UPDATE RESTRICT,
    valor DECIMAL(10,2)
);
```

Caso tente atualizar a chave primaria ja definida a ação é impedida.

Colocando Entidades na tabela.

Primeiro você identifica o que é necessário atribuir na tabela e depois você dá o comando:

```
INSERT INTO usuarios(nome, email, idade)

VALUES("coloque  as informações na ordem que você referenciou entre as pas , definindo tudo que você tem");
```

Pra você visualizar os elementos você escreveu:

```
SELECT * FROM usuarios;
```

Se você quiser colocar vários atributos novos, você utiliza o mesmo comando, só que adicionando , e ao final ;

```
INSERT INTO usuarios(nome, email, idade)

VALUES
('Joao', 'joaogmail.com', 20),
('Heitor', 'hel@gmail.com', 19),
('Leo', 'leo@gmail.com', 20),
('Giovani', 'gio@gmail.com', 17);
```

Para pegar informações precisas de um elemento.

Utilizamos:

```
SELECT nome, email FROM usuarios WHERE id = 1;
```

Para atualizar alguma informação na tabela

Utilizamos os seguintes comandos:

```
UPDATE usuarios

SET idade = (número desejado)

WHERE nome = (nome da pessoa)
```

Se você quiser atualizar a coluna

```
ALTER TABLE (nome da tabela)
ALTER COLUMN (nome da coluna) type (pra qual tipo você queira)
```

Para deletar elementos

Você utiliza o seguinte comando:

```
DELETE FROM usuarios WHERE nome = '' ”
```

Caso queira deletar uma coluna

Utiliza:

```
ALTER TABLE (NOME DA TABELA)

DROP (NOME DA COLUNA);

Para adicionar coluna:
ALTER TABLE (NOME)
ADD COLUMN (NOME DA COLUNA) (TIPO);
```

Criar relacionamentos e tabelas estrangeiras

Utilizamos:

```
CREATE TABLE pedidos(
id INT AUTO_INCREMENT PRIMARY KEY,
usuario_id INT,FOREIGN KEY (usuario_id) REFERENCES usuarios(id),
valor DECIMAL(10, 2),
data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
--No PostgreSQL SERIA

CREATE TABLE pedidos (
    id SERIAL PRIMARY KEY,
    usuario_id INT,
    valor DECIMAL(10, 2),
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_usuario
        FOREIGN KEY (usuario_id)
        REFERENCES usuarios(id)
);
```

Inserindo dados na tabela

Para inserir:

```
INSERT INTO pedidos(usuario_id, valor)
VALUES
(1, 10.00);
```

Você pode utilizar desse jeito também:

```
UPDATE investimentos
SET tipo_investimento = 'LCI'
WHERE banco = 'Itaú';
```

Diferenças entre INSERT e UPDATE

Quando utilizamos “INSERT“ queremos adicionar uma informação nova, criar uma nova linha pra aquela coluna. Ex:

```
INSERT INTO investimentos (banco, valor, tipo_investimento)
VALUES ('Itaú', 1000.00, 'LCI');
```

A partir desse comando você insere novos elementos naquela determinada linha.

O método “UPDATE” é utilizado quando queremos atualizar algo na linha que já existe, quando queremos trocar por outro. Ex:

```
UPDATE INVESTIMENTOS

SET (NOME DA COLUNA) = ("INFORMAÇÃO QUE VOCÊ DESEJA COLOCA
R")
```

```
WHERE (PK) = (COLOQUE A IDENTIFICAÇÃO A QUAL ESTÁ DEFINIDO A P
K)
```

Utilizamos o WHERE e o nome da PRIMARY KEY (PK) para definir onde queremos mudar a informação descrita.

Vinculando informações do usuário com pedidos

```
SELECT u.nome, p.valor
FROM usuarios u
JOIN pedidos p on u.id = p.usuario_id;
```

Utilizar JSON na tabela

Podemos começar criando uma tabela:

```
CREATE TABLE produtos(
id SERIAL PRIMARY KEY,
dados JSONB
);
INSERT INTO produtos(dados)
VALUES
('{"nome":"Celular", "preco": 3000}')
```

Forma de consultar produtos:

```
SELECT dados->> 'nome' as nome_produtos FROM produtos;
```

Trabalhando com Arrays

Começando com:

```
CREATE TABLE categorias(
id SERIAL PRIMARY KEY,
nome VARCHAR(100),
tags TEXT[]
);
INSERT INTO categorias(nome, tags)
VALUES
('Eletrônicos', ARRAY['Tecnologia', 'gadgets'])
```

Para fazer backup:

Clica no DATABASE, botão direito backup, seleciona um nome e um formato.

Pra restaurar, clica botão direito Restore, procura nome do backup e clica em Role Name Postgre.

Dicas de desempenho e indexação

Utilizamos:

CREATE INDEX idx_nome on usuarios (nome);

EXPLAIN ANALYZE

Forma de identificar gargalos de performance ou entender como o PostgreSQL executa uma consulta.

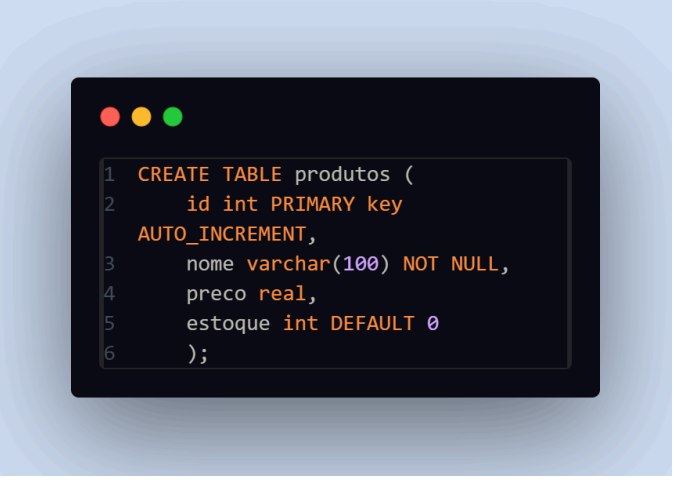
```
EXPLAIN ANALYZE SELECT * FROM usuarios WHERE nome = 'Arthur';
```

- **EXPLAIN**: Mostra o plano de execução sem executar a consulta.
- **EXPLAIN ANALYZE**: Executa a consulta e mostra o plano de execução juntamente com o tempo gasto.

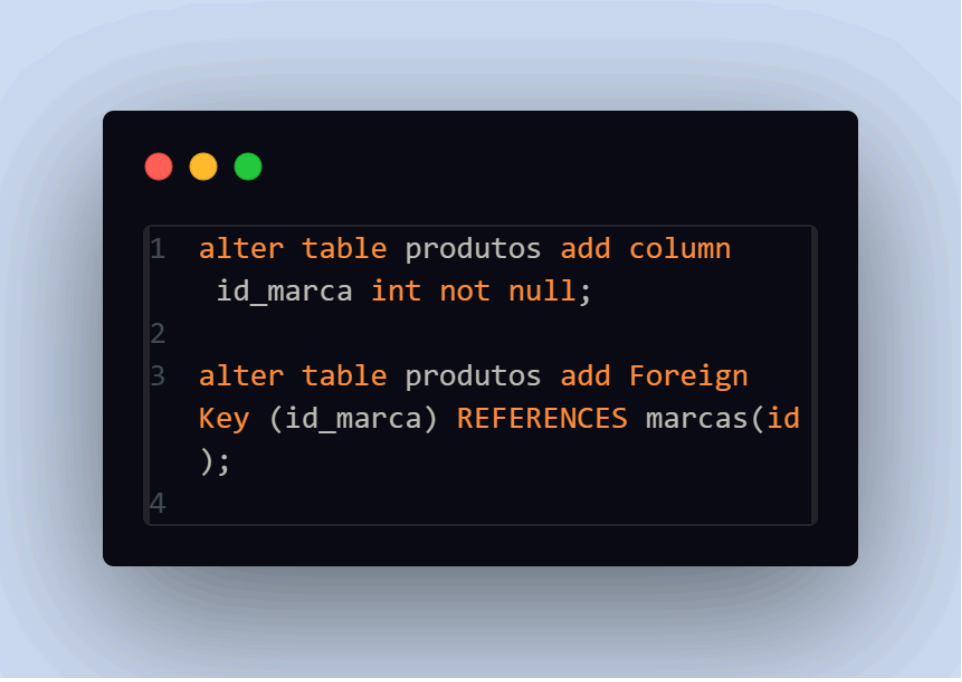
Utilizando o comando INSERT pra pegar informações de outras tabelas

Lembrando que esse comando envia temporariamente e, se você fizer alguma atualização na tabela principal, ele não vai atualizar nesta nova tabela:

Imaginando que você possui e criou uma tabela produtos:

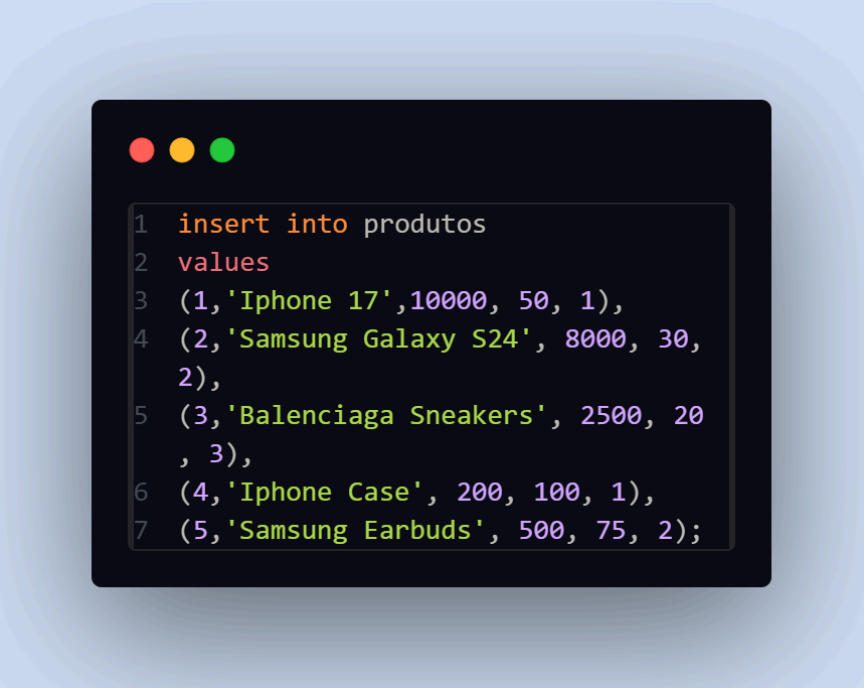


Atribuindo id_marca como FK em produtos:



```sql
alter table produtos add column
 id_marca int not null;

alter table produtos add Foreign
Key (id_marca) REFERENCES marcas(id
);

```

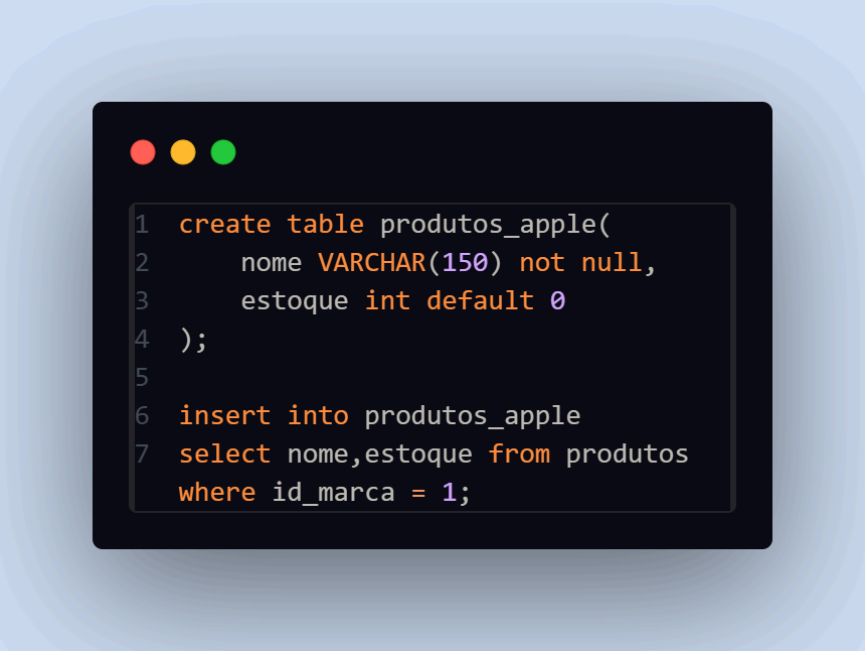
Inserindo as seguintes informações:



```sql
insert into produtos
values
(1,'Iphone 17',10000, 50, 1),
(2,'Samsung Galaxy S24', 8000, 30,
2),
(3,'Balenciaga Sneakers', 2500, 20
, 3),
(4,'Iphone Case', 200, 100, 1),
(5,'Samsung Earbuds', 500, 75, 2);
```

Note que o primeiro valor com números (1, ...) indicando o id. No caso, o 1 significa pra produtos Apple.

Agora vamos supor que você queira criar uma nova tabela temporária com só produtos Apple:



```sql
create table produtos_apple(
    nome VARCHAR(150) not null,
    estoque int default 0
);

insert into produtos_apple
select nome,estoque from produtos
where id_marca = 1;
```

Primeiro criamos a tabela e os respectivos valores e depois utilizamos o INSERT pra colocar elementos na tabela nova e logo após utilizamos o SELECT pra pegar nome, estoque na tabela (FROM) produtos WHERE (onde) id_marca = 1.

Assim você atribui os elementos que você quer naquela determinada tabela.

Utilizando LIKE, ORDER BY, DISTINCT

DISTINCT serve pra evitar duplicatas quando for usar o SELECT.

Utilizamos LIKE da seguinte maneira quando queremos procurar algum item por determinado nome sem ter que colocar ele por completo:

-

Botando entre "%%".

Utilizando ORDER BY pra organizar pelo ASC ou DESC

ASC demonstra números em ordem crescente:

-

Já o DESC mostra em ordem decrescente:

-

Se você quiser também limitar a quantidade que será exibida, utiliza-se LIMIT:

-

O comando assim mostra o top 3 produtos mais caros do sistema da tabela produtos.

Junção de tabelas (JOIN)

Temos as seguintes junções:

INNER JOIN = Retorna apenas linhas que a conexão entre as 2, ou seja, valores que possuem nas 2 tabelas.

FULL JOIN = Recebe todas as informações independente de conexão em uma ou na outra. Caso não tenha, os elementos ficam como NULL.

RIGHT JOIN = Retorna todas as linhas da tabela da direita e as linhas correspondentes na tabela da esquerda. Se não tiver correspondência, fica como NULL.

LEFT JOIN = Retorna todas as linhas da tabela da esquerda e as linhas correspondentes na tabela da direita. Se não tiver correspondência, fica como NULL.

Utilizando o INNER JOIN

Supondo que temos 2 tabelas (pedidos e clientes) e queremos juntar as 2 pra trazer uma informação de qual valor total um determinado cliente pegou e mostrar um histórico:

-

Neste comando ele está dando um SELECT na tabela clientes e pegando nome e na tabela pedidos pegando valor total e juntando os itens que possuem em comum dos 2.

Subquery

Vamos dizer que você quer fazer um filtro onde pega um determinado nome e preço dele, e você queira exibi-lo:

-

Assim este comando devolve todos os itens que possuem o nome "Iphone" ou "Samsung" e seus respectivos preços.

LEFT e RIGHT JOIN

Utilizamos o seguinte comando pra LEFT JOIN, que retorna os itens da tabela da esquerda na direita:

-

E para RIGHT JOIN retornamos elementos da tabela da direita e das linhas correspondentes da esquerda.

-

FULL JOIN

No MySQL não é possível utilizar o FULL JOIN, mas em outras aplicações é possível. Ex de como seria:

-

Uma outra alternativa seria utilizar o LEFT e RIGHT juntos:

-

Agregação e agrupamentos

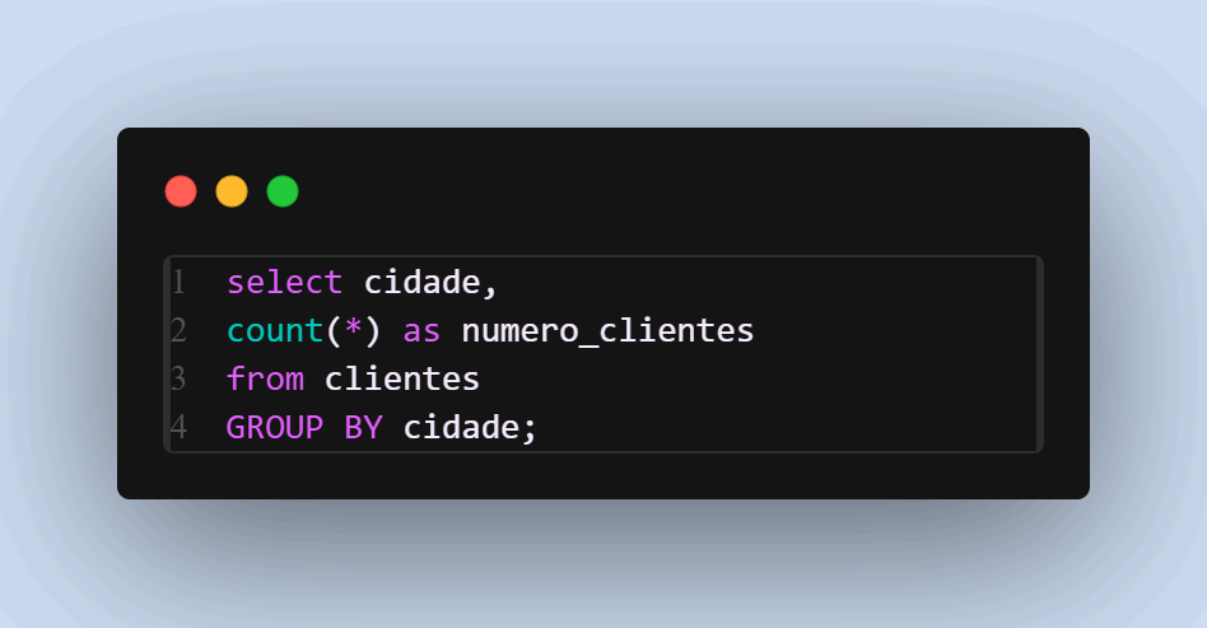
Supondo que você queira filtrar clientes por cidades

Podemos usar também o GROUP BY pra agrupar:

-

Desta forma, se tiver mais de um cliente na mesma cidade, ele vai se repetir, quando utilizamos.

Se quisermos mostrar a quantidade de registro por cidade:



```sql
select cidade,
count(*) as numero_clientes
from clientes
GROUP BY cidade;
```

Podemos fazer também uma média de vendas mensal utilizando função AVG:

Fazendo somatório de valor total:

MIN e MAX

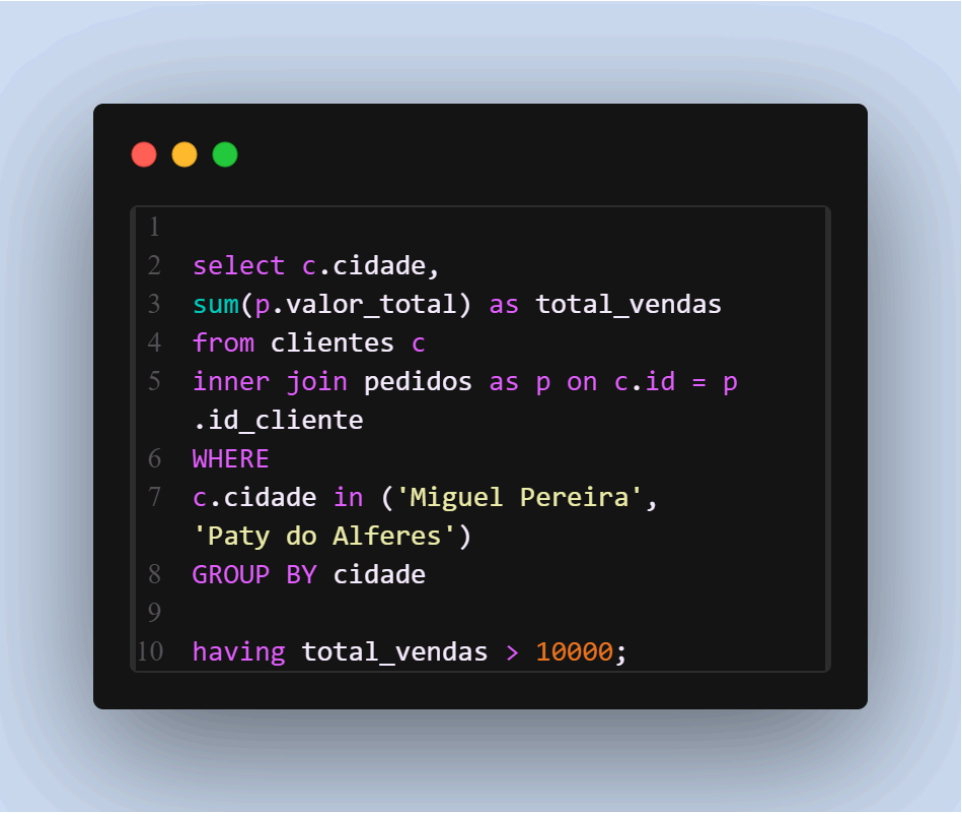
Podemos também pegar o mínimo e o máximo:

AVG

Podemos fazer uma média de produtos abaixo no estoque:

Utilizando INNER JOIN e SUM

Vamos pegar um total de venda de acordo com determinadas cidades:



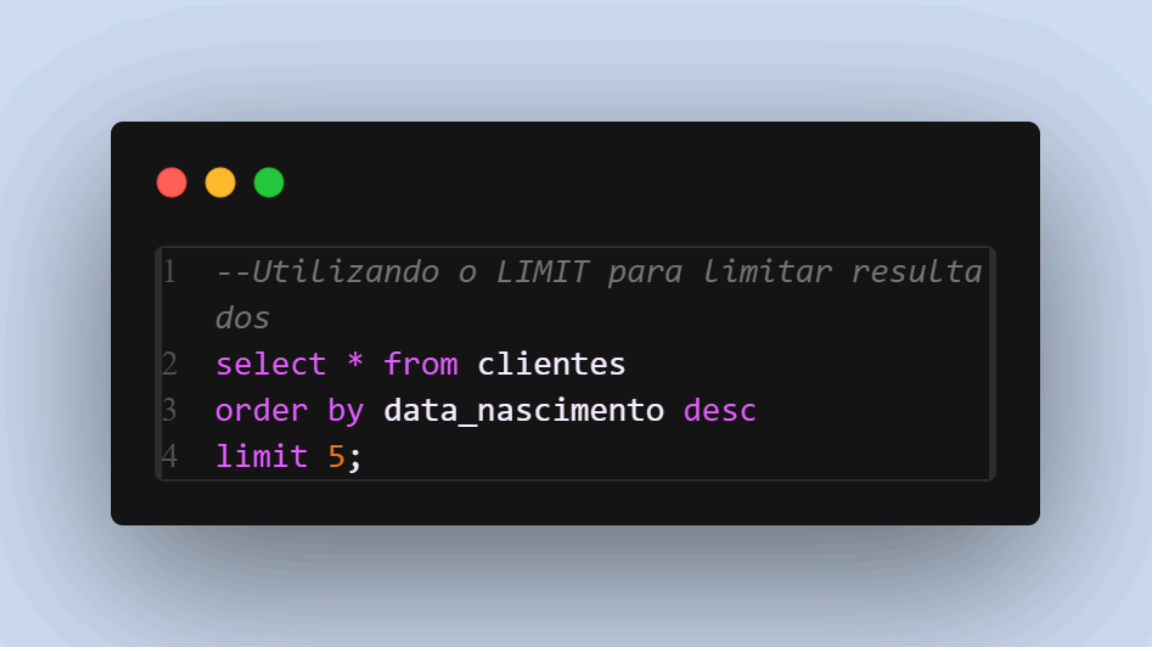
```sql

select c.cidade,
sum(p.valor_total) as total_vendas
from clientes c
inner join pedidos as p on c.id = p
.id_cliente
WHERE
c.cidade in ('Miguel Pereira',
'Paty do Alferes')
GROUP BY cidade

having total_vendas > 10000;
```

A partir desse comando selecionamos o somatório de vendas das pessoas pelo id e relacionando com sua respectiva cidade. Se você quiser especificar um

valor pra ser exibido, utiliza o HAVING, que é quando temos alguma função e você determina o valor específico pra mostrar.

LIMIT

Utilizamos o LIMIT quando temos uma tabela muito extensa ou queremos limitar a informação que estamos verificando utilizando o LIMIT.



OFFSET

Este é uma forma de pular linhas pra pegar alguma informação.

No exemplo acima ele utiliza o LIMIT 5 pra mostrar só 5 dados e OFFSET pra pular 2 linhas na hora de exibição.

VIEWS

Podemos utilizar uma função do SQL chamada VIEWS, que cria um método de salvar consultas, principalmente JOINS, de forma simples e só consultar:

-

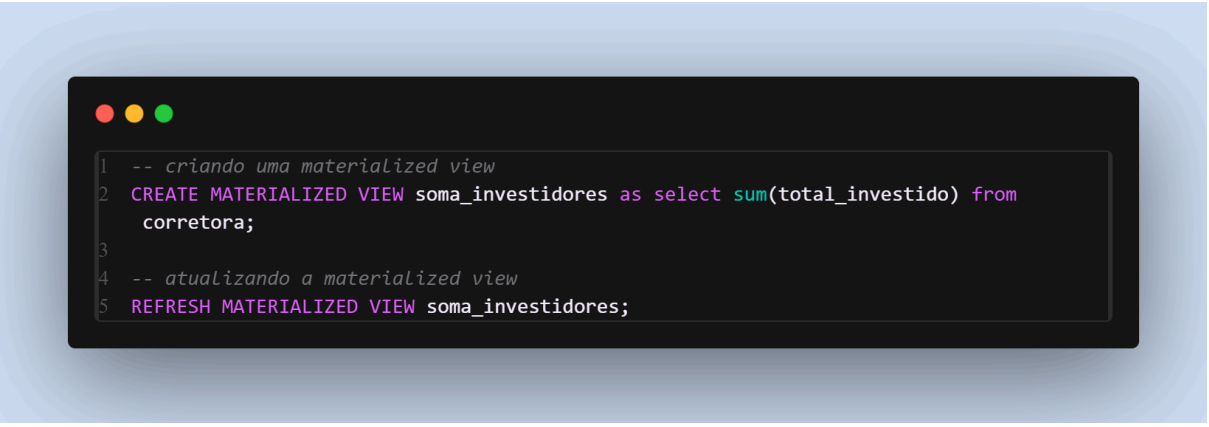
Desta forma criamos uma VIEW que, se utilizarmos o SELECT pra visualizar a tabela, não teremos que dar JOINS múltiplas vezes. Ou digamos que você queira dar informações pra um fornecedor sem ter que dar acesso a todo banco de dados.

Materialized View

A **materialized view** é uma forma de **armazenar o resultado de uma consulta no banco de dados**, permitindo buscar os dados de forma **muito mais rápida**, já que eles ficam **salvos fisicamente**.

Diferente da `VIEW` normal (que sempre executa a consulta original toda vez que é chamada), a **materialized view não atualiza automaticamente**.

Ela mostra os dados **do momento em que foi criada ou atualizada**.

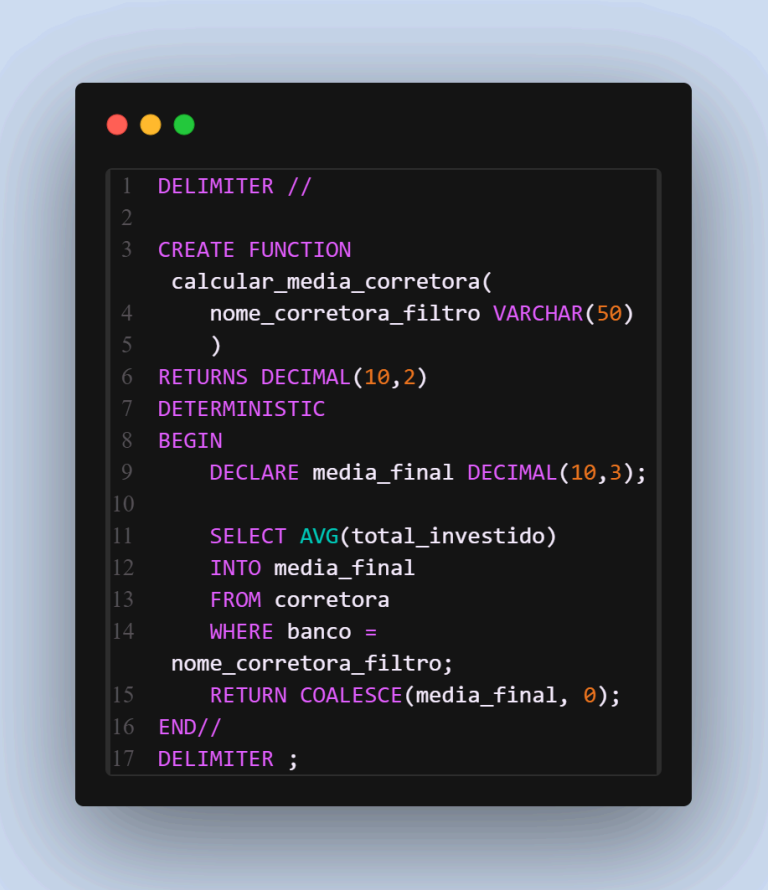


```sql
-- criando uma materialized view
CREATE MATERIALIZED VIEW soma_investidores as select sum(total_investido) from
 corretora;

-- atualizando a materialized view
REFRESH MATERIALIZED VIEW soma_investidores;
```

Desta forma você cria e atualiza os dados caso você tenha mudado alguma informação da tabela que você está puxando a VIEW (ela é boa para, por exemplo, relatórios diários que você só precisa daquela informação 1 vez, e não precisa gastar muito do banco).

Functions

Podemos facilitar queries mais complexas criando uma função pra poder buscar valores. Vamos imaginar que queremos criar uma função que determine a média de investimentos de uma determinada corretora. Sem precisar fazer sempre o mesmo query, podemos só chamar a função após sua criação.



```
DELIMITER //

CREATE FUNCTION
 calcular_media_corretora(
    nome_corretora_filtro VARCHAR(50)
    )
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE media_final DECIMAL(10,3);

    SELECT AVG(total_investido)
    INTO media_final
    FROM corretora
    WHERE banco =
 nome_corretora_filtro;
    RETURN COALESCE(media_final, 0);
END//
DELIMITER ;
```

Com esse comando você cria uma função chamada “calcular_media_corretora “, e depois de criada é só chamar a função utilizando o:

```
SELECT calcular_media_corretora('Itaú');
```

No MySQL, para criar uma FUNCTION, utilizamos o comando

```
DELIMITER //
```

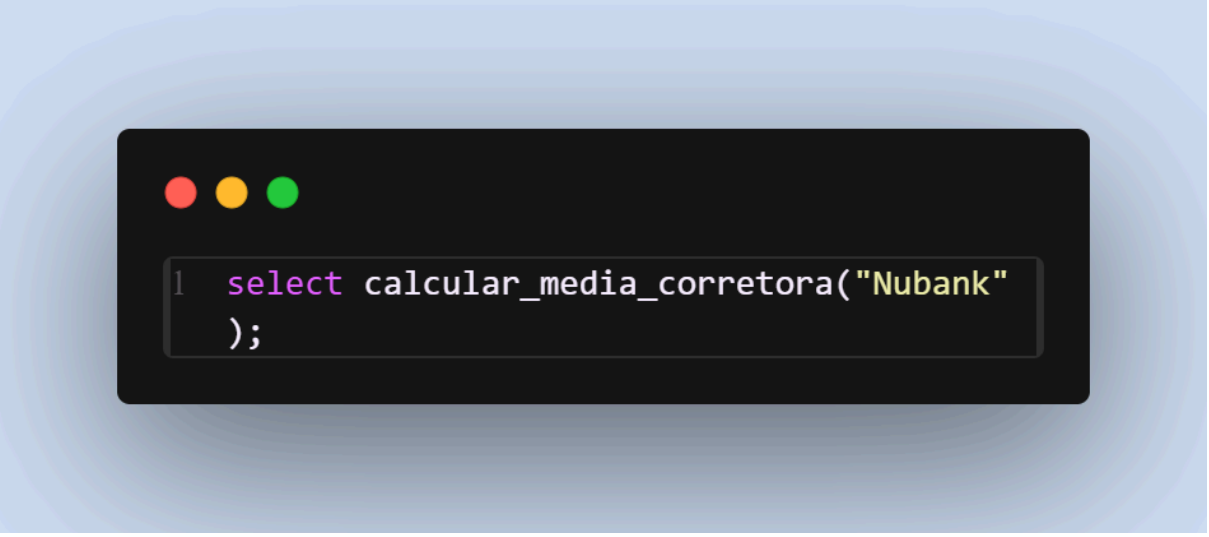
.

Já no PostgreSQL, utilizamos:

```
CREATE OR REPLACE FUNCTION calcular_media_corretora(
   nome_corretora_filtro VARCHAR(50)
)
RETURNS NUMERIC(10,2)
LANGUAGE plpgsql
```

```
AS $$
DECLARE
   media_final NUMERIC(10,3);
BEGIN

   SELECT AVG(total_investido)
   INTO media_final
   FROM corretora
   WHERE banco = nome_corretora_filtro;

   RETURN COALESCE(media_final, 0);

END;
$$;
```

após o corpo da função.



Desta forma ele já vai te resultar na média escolhida.

Procedures

Mais um método importante do SQL que utilizamos em buscas complexas pra ajudar a manter limpo e mais coerente dentro do banco de dados, utilizando CALL pra realizar a busca. O PROCEDURE ou procedimento é utilizado em exemplo como:

Encapsular lógica de negócio complexa: Imagine que para cadastrar um novo usuário, você precise:

- Verificar se o e-mail já existe.
- Inserir o novo usuário na tabela `usuarios`.
- Criar um perfil padrão para ele na tabela `perfis`.

Gerenciar transações (extremamente importante): Este é um dos superpoderes das procedures. Imagine uma transferência bancária. Você precisa:

- `UPDATE`: Subtrair R$ 100 da conta A.
- `UPDATE`: Adicionar R$ 100 na conta B.
  E se o segundo passo falhar por algum motivo? A conta A perdeu dinheiro e a B não recebeu! Uma procedure pode agrupar isso em uma transação. Se qualquer passo falhar, ela executa um `ROLLBACK` e desfaz tudo, garantindo a integridade dos dados.

DELIMITER é um comando especial para o cliente MySQL entender que o ';' dentro da procedure não termina o comando CREATE PROCEDURE.
DELIMITER

Forma de criar um Procedure:

```sql
DELIMITER//

-- criando uma procedure que insere na tabela investidor e verifica se ele ja exite o
u nao.
create PROCEDURE sp_cadastrar_cliente(
    in p_nome VARCHAR(100),
    in p_email varchar(100),
    in p_cpf varchar(11),
    IN p_estado VARCHAR(50)
)
BEGIN
    declare cpf_existente int;

    select count(*)
    into cpf_existente
    from investidor
    where cpf = p_cpf;

    if cpf_existente = 0 then
        insert into investidor(nome,cpf,email,estado,data_cadastro)
        VALUES(p_nome,p_cpf,p_email,p_estado,NOW());

        select 'Cliente Adicionado com sucesso!' AS resultado;

        else
            select 'Cliente ja cadastrado ' AS resultado;
        end if;
end//

DELIMITER ;
```

```sql
call sp_cadastrar_cliente('Ramon Monsores',
'ramon@gmail.com','23134131312','Rio de Janeiro');
```

Desta forma você consegue verificar se um cliente já está cadastrado ou não. Automaticamente, se ele não estiver cadastrado, ele insere no sistema.

Para o PostgreSQL você utiliza o SELECT ao invés do CALL:

SELECT sp_cadastrar_cliente('Joao Victor', 'joao@gmail.com', '12228976543', 'Rio de Janeiro');

Funções SQL

Funções string

CONCAT()

A função `CONCAT()` concatena strings. Ou seja, aplicando `CONCAT()` sobre duas strings, é possível uni-las em uma única sequência de texto.

```
SELECT CONCAT("SQL ", "Tutorial ", "is ", "fun!") AS ConcatenatedString;
```

LTRIM()

A função `LTRIM()` tira espaços à esquerda (left) de um texto.

```
SELECT LTRIM("     SQL Tutorial") AS LeftTrimmedString;
```

RTRIM()

A função `RTRIM()`, por outro lado, vai remover os espaços à direita (right) da string:

```
SELECT RTRIM("SQL Tutorial     ") AS RightTrimmedString;
```

TRIM()

Já se o nosso objetivo for aparar o texto dos dois lados, podemos usar a função `TRIM()`, que retira espaços tanto da direita quanto da esquerda. Os

espaços no meio da string não são removidos:

```
SELECT TRIM('    SQL Tutorial    ') AS TrimmedString;
```

Outro exemplo interessante é a função `LCASE()`, que transforma um texto que está em caixa alta (ou seja, em maiúsculas) deixando-o com letras minúsculas (em inglês, lowercase):

```
SELECT LCASE("SQL Tutorial is FUN!");
```

Uma alternativa é o `LOWER()`, que faz o mesmo processo:

```
SELECT LOWER("SQL Tutorial is FUN!");
```

UCASE()

Já a função `UCASE()` fará o contrário, deixando toda a string em maiúsculas (em inglês, uppercase):

```
SELECT UCASE("SQL Tutorial is FUN!");
```

Um equivalente a essa função é o `UPPER()`:

```
SELECT UPPER("SQL Tutorial is FUN!");
```

SUBSTRING()

A função `SUBSTRING()` também é bastante usada. Com ela, conseguimos retirar um pedaço de texto de dentro de uma string maior. Para usá-la, devemos informar três parâmetros: a string original, a posição onde começaremos a retirada do texto e o número de caracteres que vamos extrair a partir da posição que especificamos:

```
SELECT SUBSTRING("SQL Tutorial", 5, 3) AS ExtractString;
```

Nesse caso, o texto original é "SQL Tutorial", vamos começar a extrair a partir do quinto elemento (a letra T, maiúscula) e pegaremos três caracteres. Então, a

substring resultante será "Tut".

LENGTH()

A função `LENGTH()` retorna o tamanho de uma string:

```
SELECT LENGTH("SQL Tutorial") AS LengthOfString;
```

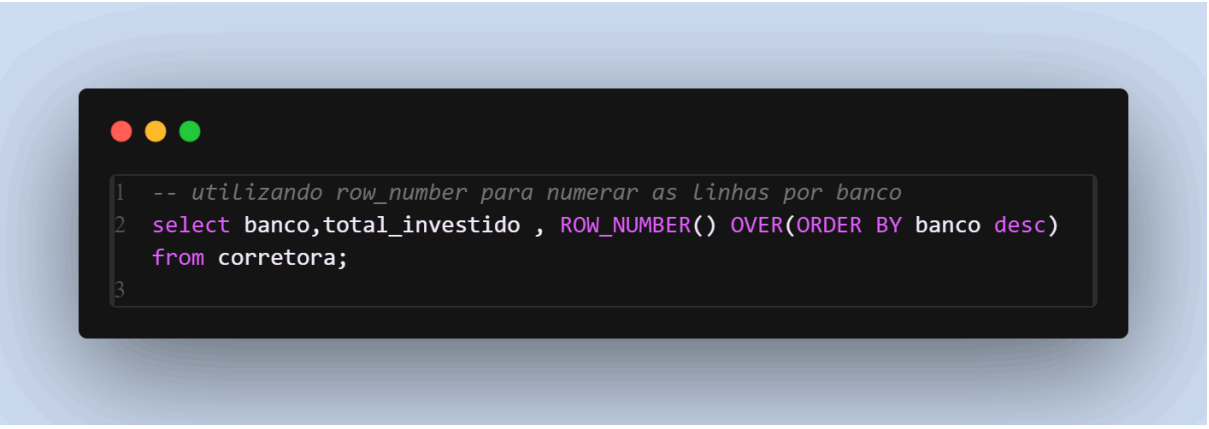
Nesse caso, o retorno será 12, pois o espaço também é contado.

Window Functions

Uma forma prática de descrever as funções de janela (Window Functions) são a forma que elas conseguem separar dados e ajudar na organização tendo como base 3 funções.
Todas as Window Functions utilizam o comando `OVER()`, que define como a função será aplicada, podendo incluir `ORDER BY` e/ou `PARTITION BY`.

ROW_NUMBER()

Utiliza este comando pra contagem de número de linhas, podendo fazer uma listagem, e mesmo que tenha números iguais ele segue contagem normalmente, melhor pra contagem somente.



```sql
-- utilizando row_number para numerar as linhas por banco
select banco,total_investido , ROW_NUMBER() OVER(ORDER BY banco desc)
from corretora;

```

	banco character varying (30)	total_investido numeric (40,3)	row_number bigint
1	XP Investimentos	30000.000	1
2	Santander	20000.000	2
3	Nubank	10000.000	3
4	Nubank	20100.000	4
5	Itaú	29000.000	5
6	Itaú	25000.000	6
7	Itaú	800000.000	7
8	Inter	14569.000	8
9	Bradesco	2400.000	9
10	Bradesco	16000.000	10
11	Banco do Brasil	20000.000	11

RANK()

Mesma função do ROW_NUMBER, mas a função do RANK vai pular. Exemplo: se tiver 2 números iguais, eles terão a mesma posição e a próxima linha vai pular +1 na numeração.

```sql
-- utilizando window function rank() pra demonstrar  rank  de maior valor,usando order by pra organizar pelo Total Investido
select i.nome,c.total_investido, rank() over (order by total_investido desc) as ranking
from corretora c
inner join investidor i on i.cpf = c.cpf_investidor;

```

6	Eduardo Lima	20000.000	6
7	Matheus Felipe	20000.000	6
8	Leonardo Curitiba	16000.000	8

DENSE_RANK()

Mesmo do RANK, só que ele não irá pular linhas em empates, só receberão o mesmo número e seguirá normalmente.

```sql
-- utilizando window function dense_rank() pra demonstrar    rank  de maior valor,usando order by pra organizar pelo Total Investido
select i.nome,c.total_investido,i.estado , DENSE_RANK() OVER(order by total_investido desc)  from corretora c
inner join investidor i on i.cpf = c.cpf_investidor;

```

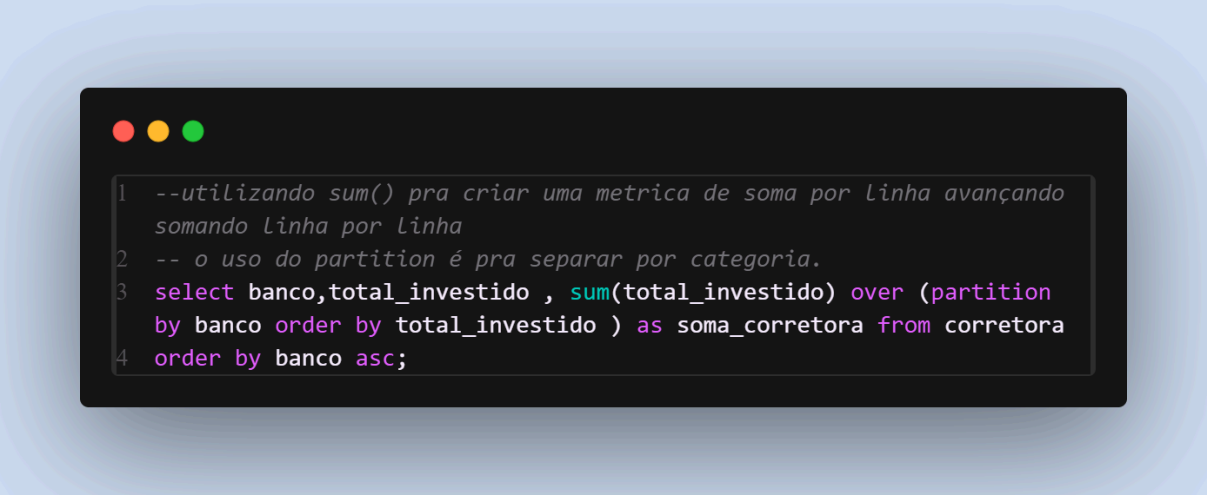
Eduardo Lima	20000.000	Paraná	6
Matheus Felipe	20000.000	São Paulo	6
Leonardo Curitiba	16000.000	São Paulo	7

SUM()

`SUM() OVER (PARTITION BY ... ORDER BY ...)` permite criar uma **métrica de soma acumulada** dentro de cada categoria.

- `PARTITION BY banco` → separa os cálculos por banco.
- `ORDER BY total_investido` → soma os valores linha por linha na ordem correta.

  O resultado é uma **soma progressiva por banco**, mostrando a evolução dos valores investidos.



```sql
--utilizando sum() pra criar uma metrica de soma por linha avançando somando linha por linha
-- o uso do partition é pra separar por categoria.
select banco,total_investido , sum(total_investido) over (partition by banco order by total_investido ) as soma_corretora from corretora
order by banco asc;
```

Dicas de desempenho e indexação

Utilizamos:

CREATE INDEX idx_nome on usuarios (nome);

EXPLAIN ANALYZE

Forma de identificar gargalos ou informações sobre determinada entidade.

EXPLAIN ANALYZE SELECT * FROM usuarios WHERE nome = 'Arthur';

- **EXPLAIN**: Mostra o plano de execução sem executar a consulta.
- **EXPLAIN ANALYZE**: Executa a consulta e mostra o plano de execução juntamente com o tempo gasto.

Ampliando conhecimento sobre índices.

Índices

Um índice em um banco de dados é muito parecido com o índice de um livro. Quando você procura um tópico específico em um livro, em vez de folhear página por página, vai até o índice, encontra o tópico desejado e, a partir dele, localiza rapidamente a página que contém a informação.

Quando temos um sistema muito complexo com dezenas de informações o tempo para encontrar um dado específico pode ocasionar gargalos devido ao grande número de informações dentro do banco, pois quando buscamos por um dado específico sem utilizar um índice o sistema realiza um processo chamado full table scan, ou seja, varredura completa da tabela, podendo ser muito demorado principalmente em milhões de registros. Os índices resolvem esses problemas proporcionando uma otimização pra encontrar os dados sem ler todos os registros.

Tipos de índices

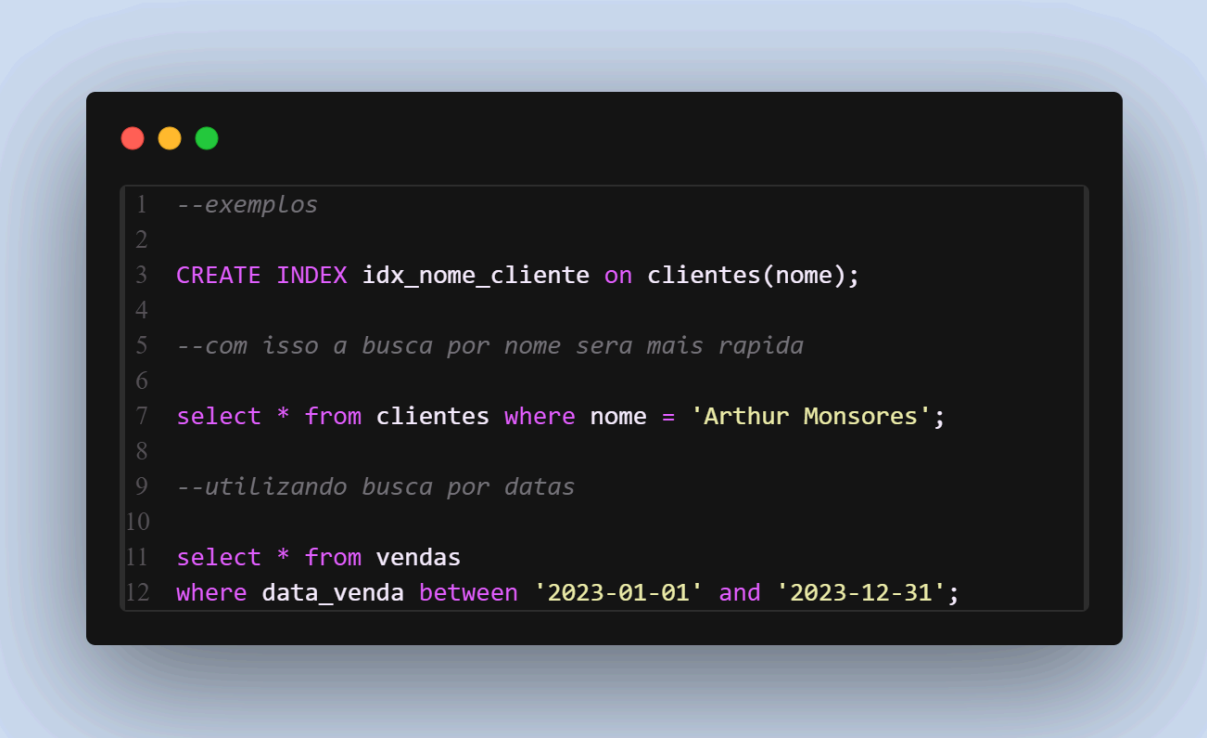
Os principais índices disponíveis no PostgreSQL são: (B-Tree, GIN, GIST, BRIN). Onde cada um tem sua característica e funcionamento pra cada situação ser o mais adequado.

B-Tree

Índice mais comum e utilizado no PostgreSQL. Devido à sua capacidade de pegar valores e mantê-los em ordem, facilitando buscas em que precisam de igualdade ou algum intervalo. Ex: em valores entre X e Y, entre datas etc.

- **Consultas de igualdade**: Quando precisamos buscar registros que tenham um valor específico em uma coluna, como buscar por um cliente específico usando o CPF.
- **Consultas por intervalo**: Quando precisamos de todos os registros em um intervalo específico, como buscar todas as vendas feitas entre duas datas

Exemplo de criação:



```sql
--exemplos

CREATE INDEX idx_nome_cliente on clientes(nome);

--com isso a busca por nome sera mais rapida

select * from clientes where nome = 'Arthur Monsores';

--utilizando busca por datas

select * from vendas
where data_venda between '2023-01-01' and '2023-12-31';
```

Hash

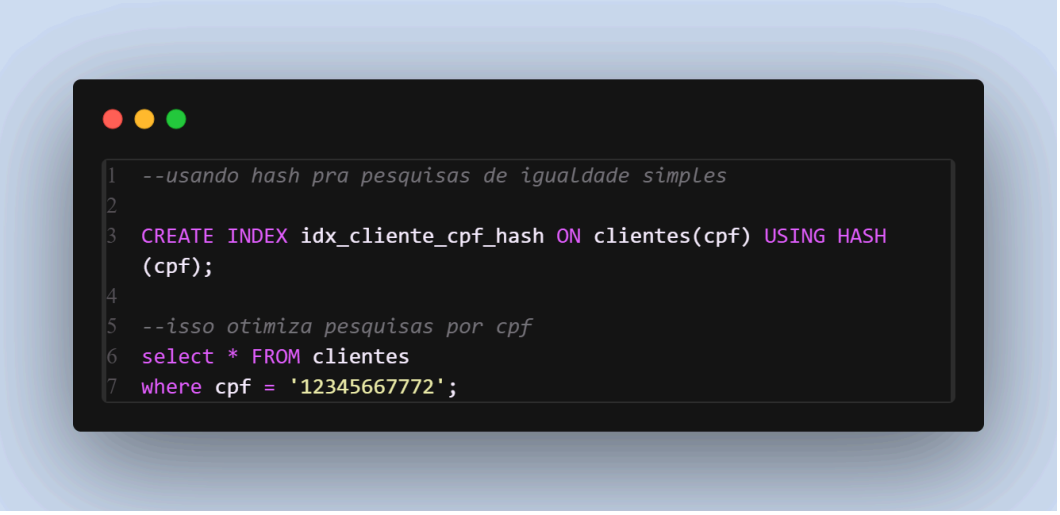
Índice mais utilizado especificamente pra buscas de igualdade, mas ele não é tão eficaz para buscas por intervalo, por isso tendo um uso mais limitado em comparação ao B-Tree, tendo também algumas limitações em termos de manutenção e não oferece um suporte nativo.

Uso:

- **Busca de igualdade simples**: Ideal para situações onde sempre se faz uma

consulta por igualdade, como verificar se um número de identificação específico está presente.

Exemplos de uso:



```sql
--usando hash pra pesquisas de igualdade simples

CREATE INDEX idx_cliente_cpf_hash ON clientes(cpf) USING HASH
(cpf);

--isso otimiza pesquisas por cpf
select * FROM clientes
where cpf = '12345667772';
```

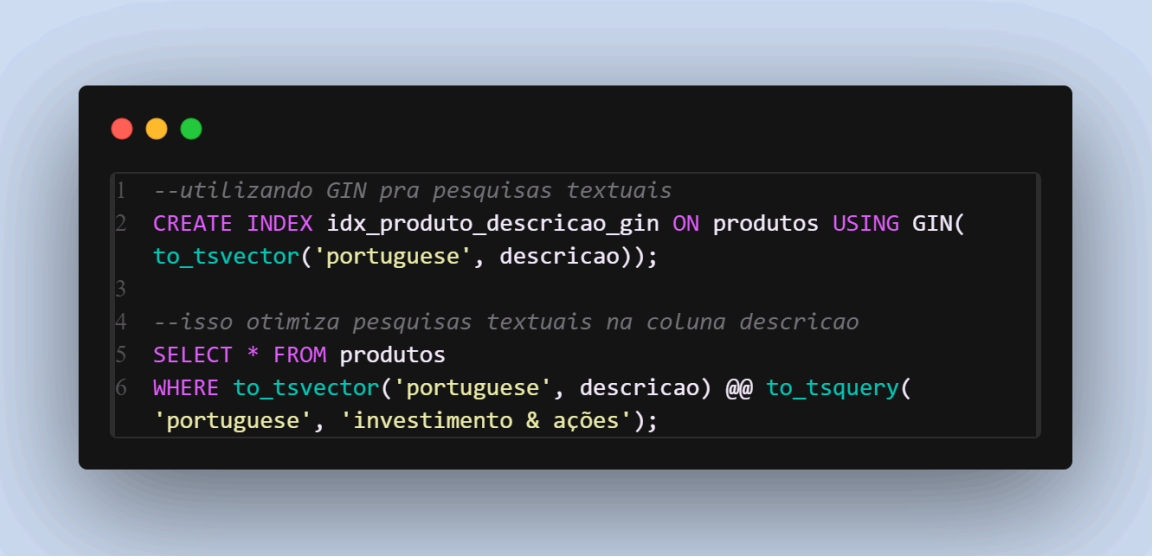
GIN (Generalized Inverted Index)

Um índice extremamente útil pra pesquisas em textos ou arrays. Utilizando o GIN você pode buscar em textos longos como artigos ou descrições palavras-chave ou algo específico, tornando mais rápido, ou realizando pesquisas textuais completas (full-text search).

Uso:

- **Full-Text Search**: Utilizado para acelerar buscas em textos longos, como artigos ou descrições.
- **Arrays**: Muito útil quando uma coluna armazena um array de valores e precisamos realizar consultas que envolvam um dos elementos do array.

Exemplos de onde utilizar:



```sql
--utilizando GIN pra pesquisas textuais
CREATE INDEX idx_produto_descricao_gin ON produtos USING GIN(
to_tsvector('portuguese', descricao));

--isso otimiza pesquisas textuais na coluna descricao
SELECT * FROM produtos
WHERE to_tsvector('portuguese', descricao) @@ to_tsquery(
'portuguese', 'investimento & ações');
```

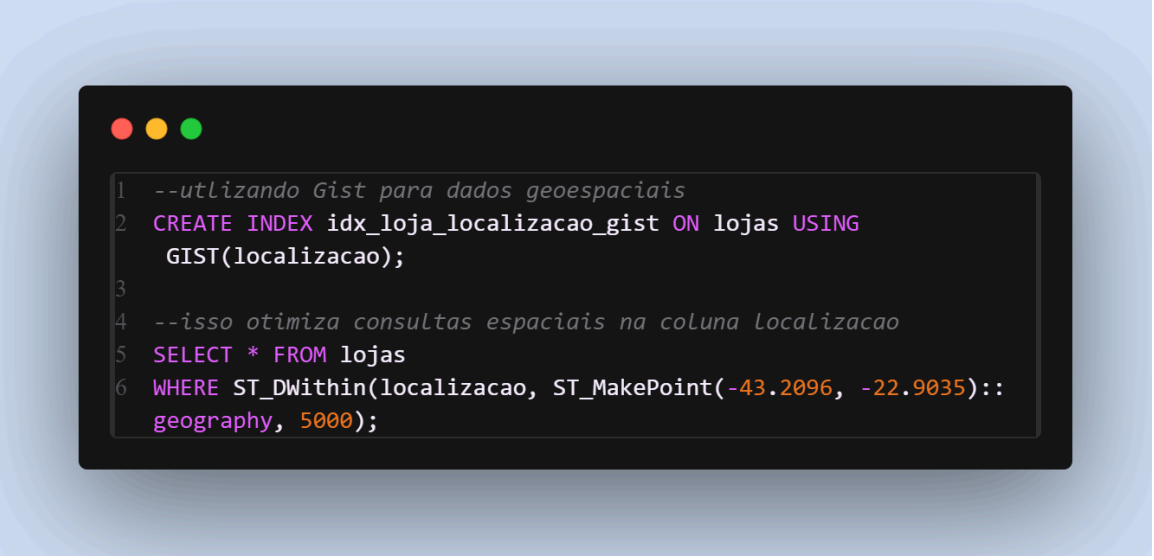
GiST (Generalized Search Tree)

Tipo de índice flexível permitindo personalização, ideal pra dados não triviais, como dados espaciais, e suporta operações complexas, como por proximidade, utilizada mais com dados geográficos permitindo a consulta por uma área determinada ou proximidade.

Uso:

- **Dados espaciais**: Quando lidamos com coordenadas geográficas e precisamos buscar dados dentro de uma determinada área.
- **Dados multidimensionais**: Utilizado em tabelas com dados que não são puramente textuais ou numéricos, mas que representam formas ou outras dimensões.

Exemplo de uso:



```sql
--utlizando Gist para dados geoespaciais
CREATE INDEX idx_loja_localizacao_gist ON lojas USING
 GIST(localizacao);

--isso otimiza consultas espaciais na coluna localizacao
SELECT * FROM lojas
WHERE ST_DWithin(localizacao, ST_MakePoint(-43.2096, -22.9035)::
geography, 5000);
```

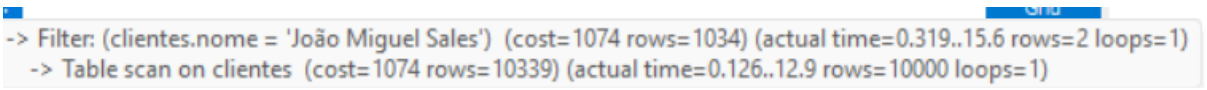
Instruções sobre cada etapa do EXPLAIN ANALYZE

Campo	Significado
cost = (start..end)	Custo estimado, usado pelo planejador. start = custo inicial; end = custo total.
rows (estimado)	Quantidade de linhas que o planner acha que virão. Diferenças grandes indicam estatísticas ruins.
width	Tamanho médio da linha (bytes).
actual time (start..end)	Tempo real gasto na operação.
actual rows	Número real de linhas processadas.
loops	Quantas vezes esse nó executou.
Filter	Filtro aplicado depois de ler os dados (não usa índice).
Rows Removed by Filter	Linhas descartadas pelo filtro. Indica falta de índice.
Index Cond	Condição aplicada diretamente no índice (melhor caso).
Recheck Cond	Verificação usada no Bitmap Heap Scan.
buffers	Indica se os dados vieram de memória ou disco.
Sort Method	Método de ordenação usado.
Planning Time	Tempo para gerar o plano de execução.
Execution Time	Tempo total real da consulta.

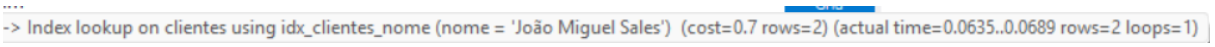
Utilizando testes pra ver a diferença entre não usar índices e utilizando.

Utilizando B-Tree (feito em um BD simples com poucas aplicações). Teste realizado em MySQL Workbench.

Sem index.



Com index.



Usando testes de index em uma tabela com 100k de registros

Primeiramente testei sem index(feito no Postgresql):

text
Sort (cost=2615.69..2621.49 rows=2320 width=23) (actual time=15.649..16.050 rows=2387.00 loops=1)
Sort Key: data_venda
Sort Method: quicksort Memory: 208kB
Buffers: shared hit=736
-> Seq Scan on vendas (cost=0.00..2486.00 rows=2320 width=23) (actual time=0.035..15.050 rows=2387....
Filter: ((data_venda >= '2025-02-01'::date) AND (data_venda <= '2025-04-01'::date) AND (produto = 'Café'...
Rows Removed by Filter: 97613
Buffers: shared hit=736
Planning Time: 0.190 ms
Execution Time: 16.294 ms

Custo:

seq scan on vendas(

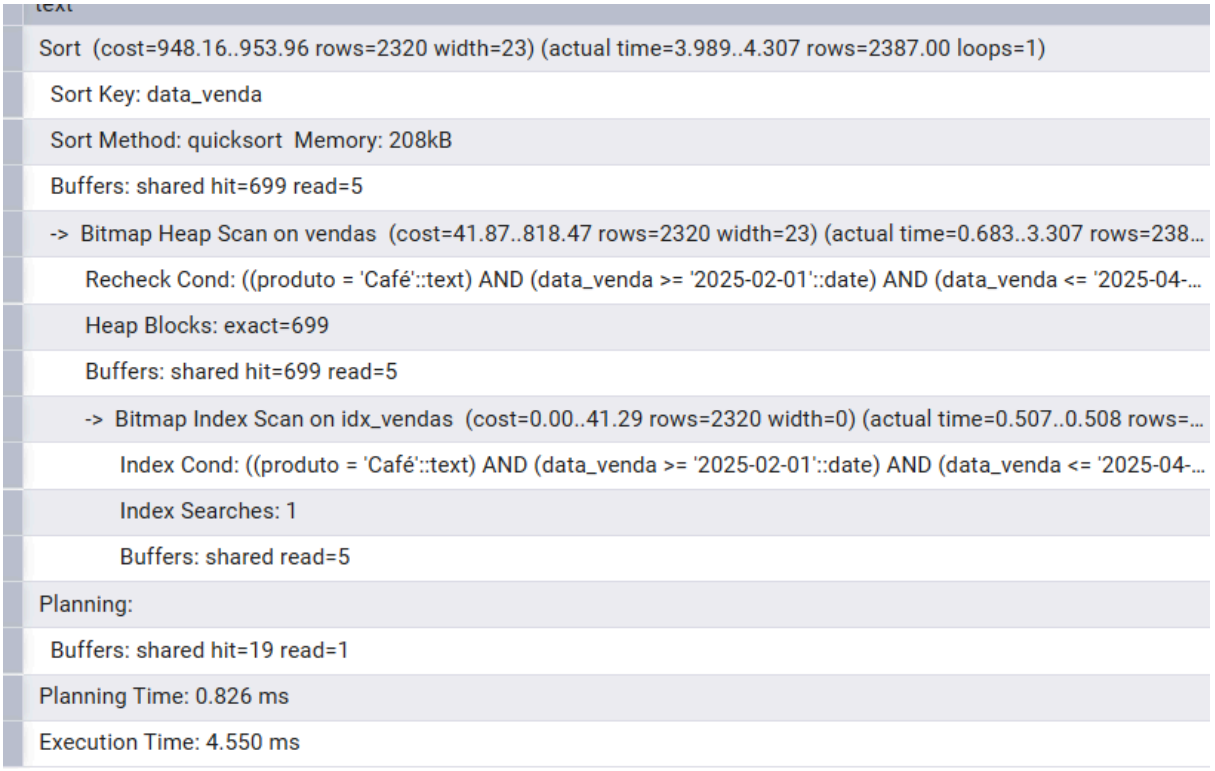
cost = 0.00..2486.00

rows = 2320 (actual time = 0.035..15.050 rows = 2387 loops = 1)
)

Execution time = 16.294 ms

Planning Time = 0.190 ms

Segundo teste com index:



Valores pós index =

cost = 948.16

rows = 2320

actual time = 3.989 ms

execution time = 4.550 ms

Normalização de dados

Uma forma de distribuir os dados sem que haja redundância ou repetições desnecessárias e evitar anomalias (problemas em inserções, atualizações e exclusões).

Exemplo: Queremos criar uma tabela Pedidos onde posso armazenar inúmeros dados como endereço de um cliente, só que quando o cliente mudar de endereço teria que atualizar todas as linhas. Com a normalização, o seu objetivo é facilitar a manutenção e consistência dos dados.

Tipos de dependências

Irei listar os 3 tipos de dependências principais

Dependência total

A dependência total é quando um atributo depende totalmente da chave primária composta.

Exemplo:

Uma tabela Curso_Aluno, onde seus atributos são (**id_curso, id_aluno, nota**) o atributo **nota** depende da combinação de id_curso e id_aluno para existir.

Dependência parcial

Uma dependência parcial é quando um atributo depende de uma parte da chave primária composta e não da chave completa.

Exemplo:

Uma tabela curso_aluno (onde possui **id_curso, id_aluno, nome_curso**) o atributo **nome_curso** depende somente do **id_curso**. Violando a 2FN, onde seria necessária a criação de uma outra tabela

Dependência transitiva

Uma dependência transitiva é quando um atributo não-chave depende de outro atributo não-chave ao invés da chave primária.

Exemplo:

Uma tabela de vendas onde seus atributos são (id_cod (PK), id_vendedor, nome_vendedor, qntd) note que o atributo vendedor depende do id_vendedor e não propriamente da chave primária da tabela (**id_cod**). Isso viola a 3FN, onde o certo seria criar uma nova tabela

Primeira forma normal (1FN)

Forma de particionar os dados sem exibir atributos multivalorados, de forma que só tenha um único valor em cada linha, como endereço, que pode ser dividido em rua, número, cidade etc. Com esta primeira forma criamos uma tabela associada à tabela pai, de forma que os dados sejam mostrados mais simples.

id_pedido	valor	id_cliente	endereço

Desta forma teria atributos multivalorados, o correto para se fazer nessa situação é a criação de uma tabela para o cliente armazenando o valor do endereço melhor.

cliente_id	nome	endereço

Segunda forma normal (2FN)

A segunda forma normal consiste em que a tabela esteja na 1FN e não possua dependências parciais,
ou seja, **nenhum atributo não-chave pode depender de apenas uma parte da chave primária composta** — ele deve depender da chave inteira.

Como o exemplo abaixo:

id_curso	id_aluno	nome_curso	nota
1	1	SQL	10
2	2	Algoritmos	9
3	3	Matemática	8

Note que a tabela está na 1FN, mas não está seguindo a forma em que a 2FN deve seguir, pois a coluna nome_curso existe indiferente de possuir o id_aluno ou não, criando uma dependência parcial. Neste caso deve-se criar uma tabela somente para id_aluno com outras informações

Tabela: Curso_Aluno

id_curso	id_aluno	nota

Tabela Curso

id_curso	nome_curso	

Agora com essa nova tabela está de forma mais organizada onde a tabela Curso_Aluno possui a nota com dependência total entre id_curso e id_aluno. Na tabela Curso, somente o que é necessário.

Terceira forma normal (3FN)

A terceira forma normal é feita pra tabelas onde já estão seguindo as duas outras formas, e nela não pode conter dependência transitiva, quando um atributo não-chave depende de outro atributo não-chave.

Uma forma de resolver isso é pegar os dois elementos que têm essa dependência e o valor não-chave vira uma PK de uma nova tabela trazendo

aquele valor para a outra tabela B. O valor não-chave que virou PK na tabela B vira uma FK na tabela principal.

Exemplo:

cod_nota_fiscal	cod_vendedor	nome_vendedor	qntd_vendida	cod_prod
1	01	Arthur	3	22
2	02	Hans	1	11

Note que eu posso identificar o nome do vendedor pelo seu código, o que acaba criando uma dependência transitiva onde não seria necessário a tabela estar estruturada desta forma.

Tabela 1:

cod_nota_fiscal (PK)	cod_vendedor (FK)	qntd_vendida	cod_prod

Tabela 2:

cod_vendedor (PK)	Nome_vendedor

Tipos de SCAN

Seq Scan

O scan mais básico lê todas as linhas da tabela, mas acaba sendo mais lento por percorrer todas as linhas da tabela. Geralmente utilizado quando não existe nenhum índice na coluna filtrada, quando é uma tabela pequena.

Exemplo de uso:

“Digamos que possuímos uma tabela chamada ‘supermarket_sales’ onde queremos fazer uma pesquisa pelo gênero”

```
SELECT * FROM supermarket_sales where gender = 'Female'
```

Index Scan

Um scan completo utilizado para localizar as linhas e depois acessar a tabela para busca completa. É fortemente utilizado quando o filtro tem uma coluna indexada e a consulta retorna um percentual pequeno da tabela.

Vamos criar um índice para uma tabela reservas na coluna de preço total. Assim, toda pesquisa que utiliza esta coluna será um pouco mais otimizada.

```
CREATE INDEX idx_reservas_preco ON reservas(preco_total);
```

```
EXPLAIN ANALYZE
SELECT * FROM reservas WHERE preco_total > 100;
```

Index Only Scan

É o tipo de scan **mais rápido**, pois o PostgreSQL não precisa acessar a tabela — todas as informações necessárias estão no índice. Quando essa leitura ocorre:

- A consulta usa apenas colunas presentes no índice.

- O *visibility map* está atualizado.

```
CREATE INDEX idx_reservas_preco ON reservas(preco_total);

EXPLAIN ANALYZE
SELECT preco_total FROM reservas WHERE preco_total > 100;
```

Bitmap Heap Scan e Bitmap Index Scan

São dois scans trabalhando juntos. O Bitmap Heap Scan pega vários registros do índice, enquanto o Heap Scan lê os blocos da tabela de forma otimizada. Acontece quando uma consulta retorna muitos registros, mas não o suficiente pra o Seq Scan, filtros com mais de um índice ao mesmo tempo.

```
EXPLAIN ANALYZE
SELECT * FROM reservas
WHERE preco_total > 100
  AND hospedagem_id = 2;
```

Joins e Estratégias de Execução (Nested Loop, Hash Join, Merge Join)

Vamos entender como os JOINS são executados pelo Postgresql internamente, pois entender como ele funciona ele leva uma melhor performance, impacta no

índice escolhido o tempo da consulta e tipo de scan. Isso tudo de informações ocorre dentro do seu EXPLAIN ANALYZE.

Nested Loop Join

Tipo de JOIN mais simples:

- Para cada linha da tabela A o Postgresql vai procurar por correspondências na tabela B.

Quando é utilizado:

- Quando uma das tabelas é muito pequena.
- Quando o indice da tabela secundaria é bem aproveitado

Exemplo:

```
EXPLAIN ANALYZE
SELECT * FROM clientes c
JOIN pedidos p on p.cliente_idsim= c.id

-- Se tiver um indice para o pedidos.cliente_id no seu EXPLAIN ANALYZ
E terá:

Nested Loop
→ Seq Scan ou Index Scan em clientes
→Index Scan em pedidos
```

Hash Join

O método mais rápido para grandes tabelas que não possuem índices. Seu funcionamento é dividido em etapas:

- Primeiro, o PostgreSQL escolhe uma das tabelas — geralmente a menor.

- Cria uma **tabela hash em memória** contendo os valores da coluna utilizada no JOIN. Essa estrutura hash permite comparações rápidas de igualdade(=). Não possui relação com os ‘índices hash’; o próprio PostgreSQL cria sua própria hash interna.
- Percorre a outra tabela e busca valores hash correspondentes.

O PostgreSQL escolhe utilizar o Hash Join quando:

- Não existem índices para o JOIN executado.
- Há muitos registros na tabela.
- A condição do JOIN retorna muitas linhas.

Breve Explicação sobre Hash.

Hash é uma função que transforma um valor qualquer (como texto ou número) em um código numérico fixo. Esse código é rápido de comparar e permite buscas extremamente eficientes. No Hash Join, o PostgreSQL cria uma estrutura hash na memória com os valores da tabela menor, permitindo localizar correspondências de igualdade de forma muito rápida, mesmo sem índices.

Merge Join

Método de JOIN que funciona comparando duas tabelas ordenadas pela coluna utilizada no JOIN. É extremamente eficiente para tabelas já ordenadas ou que possuem índices B-tree nas colunas.

Exemplo:

```
Tabela A (ordenada)
1,2,3,5,7

Tabela B
1,2,3,4,7
```

A partir daí, ele compara o primeiro valor de cada tabela. Quando são iguais, o JOIN é registrado. Quando há diferença, ele avança na tabela com menor valor.

Isso o torna rápido pelos seguintes motivos:

- Comparar duas linhas ordenadas é computacionalmente mais barato.
- Não precisa criar uma tabela hash.
- Não precisa fazer busca linha por linha.
- Percorre cada tabela apenas uma vez.

Ele é utilizado quando:

- Quando **ambas as tabelas já estão ordenadas** pela coluna do JOIN
- Existem **índices B-tree** nas colunas utilizadas
- Quando existe um **ORDER BY** que combina com a coluna do JOIN
- O custo de criar uma hash (Hash Join) é maior
- Quando o JOIN retorna muitas linhas
- Quando tabelas são grandes e ordenadas

Considerações finais

Esta documentação reuniu os principais conceitos de SQL e PostgreSQL, com foco em exemplos práticos e conteúdos baseados em estudos, testes e documentações oficiais.

Referencias

- Documentação Oficial PostgreSQL
- PostgreSQL EXPLAIN Planner
- PostgreSQL Indexes & Index Types
- PostgreSQL Window Functions Guide
- SQL ANSI Standard
- Materiais de estudo próprios
- Aulas e exercícios da faculdade

- Testes e consultas realizadas localmente