POSTGRESQL

É um **sistema de gerenciamento de banco de dados** do tipo **relacional.** O Postgresql é um descendente de código fonte aberto desde código original de **Berkeley**, que suporta grande parte do padrão SQL e oferece muitas funcionalidades modernas, como:

- comandos complexos
- chaves estrangeiras
- gatilhos
- visões
- integridade transacional
- controle de simultaneidade multiversão

Além disso, o PostgreSQL pode ser ampliado pelo usuário de muitas maneiras como, por exemplo, adicionando novos:

- tipos de dado
- funções
- operadores
- funções de agregação
- métodos de índice
- linguagens procedurais

BANCO DE DADOS RELACIONAL:

Os bancos de dados relacionais armazenam os dados em linha e colunas, em tabelas, e são baseados na álgebra relacional, um campo da teoria dos conjuntos algébricos.

As linhas se relacionam com as colunas, por isso a denominação **relacional.**

A estrutura dos bancos relacionais permite associar informações de diferentes tabelas, utilizando **chaves estrangeiras**, também chamadas de índices.

BANCO DE DADOS NÃO RELACIONAL:

Já os bancos de dados não relacionais não é estruturado por tabelas, logo, não são necessários esquemas para adicionar um dado relacionando uma tabela com uma linha.

PGADMIN4:

Parte gráfica do db. Pode ser executada pelo software ou por web.

127.0.0.1/pgadmin4 (configurado no ubuntu)

PSQL:

É um arquivo executável, um cliente em modo texto que pode ser encontrado no diretório de instalação do Postgres.

sudo -u postgres psql (configurado para abrir o psql no Ubuntu Shell)

ou

sudo -i -u postgres (digitar a senha configurada)
psql

ou

psql -U postgres (entra direto no usuário postgres psql)

ALGUNS COMANDOS BÁSICOS:

\h (mostra todos os comandos do psql)
\h "nomeComando" (Explica sobre como ela pode ser utilizada)
man "comando" (mostra o manual do comando)
\? (mostra todos os comandos de SU)
\l (lista todos os banco de dados)
q (volta para a tela anterior)
\q (sair)
\du (lista todos os usuários)
\ c "nomedb" (seleciona o banco de dados)
\conninfo (mostra qual usuário está conectado)
\dt **(**lista todas as tabelas)
\password (inserir uma senha para o usuário)
ctrl +l (limpa a tela)

sudo systemctl restart postgresql (reinicia o postgresql)

COMO MUDAR O COMPORTAMENTO PADRÃO DE SENHA PSQL:

Por padrão, ao alterar a senha do psql, o Ubuntu continua não pedindo senha para logar, ele considera que o mesmo usuário do Ubuntu é quem está utilizando o servidor. Para mudar essa configuração padrão e fazer com que, ao tentar no psql, ele peça senha, faça:

show hba_file; (mostra onde está o arquivo pg_hba.conf)

Copie o endereço abaixo e o edite com **nano**:

sudo nano /etc/postgresql/15/main/pg_hba.conf
Ctrl+O (salva)
Enter
Ctrl+X (sair)

METHOD

peer: Por padrão ele vem como peer, seria um padrão “eu consigo em você”, o sistema confia que o mesmo usuário que está logado no Ubuntu é o mesmo que está mexendo no servidor.

md5: Pede senha ao tentar entrar como usuário no db

CRIAR UM NOVO USUÁRIO NO BANCO DE DADOS:

createuser: Cria um novo usuários

sudo -i -u postgres (primeiro, logar com o usuário que tem permissão)

createuser -dPs elbert (cria um novo usuário com permissão para criar um db, pede para inserir uma senha de usuário e permissão para super usuário)

ou

createuser –interactive elbert (cria novo usuário de modo interativo)

Para logar com o novo usuário, é necessário configurá-lo no **hba_file.**
Após configurar o hba_file, reiniciar o psql para que o novo usuário seja reconhecido.

sudo systemctl restart postgresql

Para logar a primeira vez com o usuário, necessita colocá-lo em um db que existe

psql -U elbert postgres (apenas para o primeiro login)

Para os próximos logins, basta:

psql -U elbert

COMO EXCLUIR UM USUÁRIO:

DROP: Serve para excluir

dropuser: utilizado para excluir um usuário

sudo -i -u postgres (loga com o usuário que tem permissão)

dropuser elbert (exclui o usuário)

COMO CRIAR UM BANCO DE DADOS:

create database "name": Cria um novo banco de dados

create database funcionarios; (colocar o ; para finalizar o comando)

ou então (as configurações abaixo são padrões do código acima)

CREATE DATABASE funcionarios
WITH
OWNER = proprietário do db (nome do usuário proprietário do db)
ENCONDING = "UTF8" (codificação do idioma)
LC_COLLATE = "pt_BR.UTF-8" (colação do banco, para os caracteres)
LC_CTYPE = "pt_BR.UTF-8" (tipos de caracteres)
TABLESPACE = "pg_default"
CONNECTION LIMITE -1 (-1 = conexão de user ilimitadas)
(finaliza o comando ;)

GRANT: Dá permissão para o usuário no banco de dados

GRANT ALL ON DATABASE bancoDeDados TO usuario;

REVOKE: Retira a permissão do usuário para o banco de dados

REVOKE ALL ON DATABASE bancoDeDados FROM usuario;

https://www.devmedia.com.br/gerenciando-usuarios-e-permissoes-no-postgresql/14301

COMO RENOMEAR UM USUÁRIO:

ALTER USER nome **RENAME TO** novo_nome;

TIPOS DE DADOS EM POSTGRESQL

https://www.postgresql.org/docs/15/datatype.html

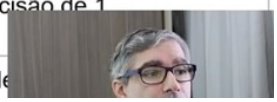
O PostgreSQL possui uma rica variedade de tipos de dados que podem ser empregados na definição de colunas e tabelas.

Esses tipos englobam os domínios numéricos, texto, data e hora, geométrico, booleano, endereçamento de redes, enumeração, e até mesmo tipos definidos pelo usuário, com o uso da declaração **CREATE TYPE.**

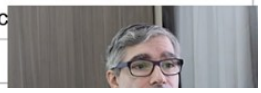
Tipo	Aplicação
smallint	Inteiros de 2 bytes
int	Inteiros de 4 bytes
bigint	Inteiros de 8 bytes
numeric(Precisão,Escala)	números de ponto flutuante, onde: precisão = número escala = nº de casas decimais
real	32 bits, até 6 dígitos decimais de precisão
double precision	64 bits - variável, até 15 dígitos de precisão
serial	32 bits de tamanho, com sinal, números sequenciais
big serial	64 bits de tamanho, com sinal, números sequenciais
money	64 bits, com sinal (2^{63} valores). Para valores monetár

Tipo	Aplicação
text	varchar de tamanho ilimitado. Tipo preferido para strings.
char(n), character(n)	caracteres de tamanho fixo, com padding (preenchimento) e n caracteres.
varchar(n)	varchar de tamanho limitado a até n caracteres.

Tipo	Aplicação
date	4 bytes, apenas datas, precisão de 1 dia
time [without time zone]	8 bytes, hora sem fuso horário, com precisão de 1 microssegundo
time with time zone	12 bytes, armazena data e hora com fuso horário, precisão de 1 microssegundo
timestamp with time zone	8 bytes, armazena data e hora com fuso horário, precisão de 1 microssegundo
timestamp [without time zone]	8 bytes, armazena data e hora sem fuso horário, precisão de 1 microssegundo
interval	16 bytes, armazena faixas de tempo, com precisão de



Tipo	Aplicação
boolean	Tipo lógico; 8 bits (1 byte) - Valores **True** (1 / yes / on) ou **False** (0 / no / off)
cidr	7 ou 19 bytes - endereços de rede IPv4 ou IPv6, como 192.168.14.0/24
inet	7 ou 19 bytes - endereços de hosts IPv4 ou IPv6, como 192.168.14.22/32
macaddr	6 bytes (48 bits), como 00:22:33:44:55:b2 ou 0022.3344.55b2
Geometric Types	Armazenar informações relacionadas com figuras geométricas, como linhas, círculos, polígonos, pontos, caminhos, etc.
Tipos de Enumeração	Criados pelo usuário, para conjuntos de valores estátic
tsvector / tsquery	Tipos para busca completa de texto em documentos.



CRIAR TABELAS

Create Table: Criar tabelas no db

Exemplos:

```
CREATE TABLE clientes (
cod_cliente INT PRIMARY KEY,
nome_cliente VARCHAR(20) NOT NULL,        //NOT NULL = OBRIGATÓRIO
sobrenome_cliente VARCHAR(40) NOT NULL
);
```

Ou:

```
CREATE TABLE produtos (
cod_produto INT PRIMARY KEY,
nome_produto VARCHAR (30) NOT NULL,
descricao TEXT NULL,
```

preco NUMERIC CHECK **(preco >0)** NOT NULL,
qtde_estoque SMALLINT DEFAULT 0
);

Ou:

CREATE TABLE pedidos (
cod_pedidos SERIAL PRIMARY KEY,
cod_clientes INT NOT NULL REFERENCES clientes (cod_cliente),
cod_produtos INT NOT NULL REFERENCES produtos (cod_produtos),
qtde SMALLINT NOT NULL
);

INSERIR REGISTROS NA TABELA

INSERT INTO : Inserir registros na tabela

INSERT INTO nome_tabela (coluna1, coluna2, coluna3 …)
VALUES (dado1, dado2, dado3 …);

Exemplo:

INSERT INTO cliente (cod_cliente, nome_cliente, sobrenome_cliente)
VALUES (1, 'Fábio', 'dos Reis');

SELECT * FROM clientes; (Mostra os clientes cadastrados)

CADASTRANDO VÁRIOS ITENS DE UMA VEZ:

INSERT INTO produtos(
cod_produto, nome_produto, descricao, preco, qtde_estoque)
VALUES
(1, 'Álcool Gel', 'Garrafa de álcool em gel de 1 litro', 12.90, 20),
(2, 'Luvas de Látex', 'Caixa de luvas de látex com 100 unidades' ,32.50 ,25),
(3, 'Pasta de Dentes', 'Tubo de pasta de dentes de 90 gramas', 3.60, 12);

CONSULTAS SIMPLES:

SELECT: Consultas simples em tabelas de um banco de dados.

Sintaxe:

SELECT coluna(s) **FROM** tabela
CONDIÇÕES;

Exemplo:

SELECT * FROM usuario; (***** = todas as colunas da tabela)

SELECT nome_produto, preco **FROM** produtos;

CONSULTAS COM FILTRAGENS WHERE:

https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-where/

https://blog.betrybe.com/sql/sql-delete/

WHERE: Utilizado para realizar filtragens.

Sintaxe:

```
SELECT coluna(s)
FROM tabela(s)
WHERE  condição-filtragem
```

Exemplos:

```
SELECT *
FROM produtos
WHERE cod_produtos = 6;          (busca o produto de código 6 da
                                      tabela produtos)
```

Ou:

```
SELECT nome_produto, preco
FROM produtos
WHERE preco < 7.99               (filtra pelos produtos com valor
                                      abaixo de 7.99)
```

Ou:

```
SELECT *
FROM clientes
WHERE nome_cliente LIKE 'F%';    (filtra todos os clientes que
                                  começam com a letra F)
```

CLAUSURA ORDER BY:

ORDER BY: Permite ordenar os resultados de uma consulta sql, de modo a exibir esses resultados em ordem crescente ou decrescente, numérica ou ordem alfabética ou inversa.

Sintaxe:

```
SELECT coluna(s)
FROM tabela(s)
WHERE  condição-filtragem
ORDER BY coluna, segunda_coluna ASC | DESC;

ASC = Menor para o maior         (ASC é padrão)
```

DESC = Maior para o menor

Exemplos:

```
SELECT nome_produto, preco
FROM produtos
WHERE  preco < 7.00
ORDER BY preco;          OU    ORDER BY nome_produto,preco;
```

Ou:

```
SELECT nome_produto, descricao
FROM produtos
ORDER BY descricao NULLS FIRST | LAST;
```

LIMIT E OFFSET:

https://www.devmedia.com.br/sql-limit/41216

Permite obter uma **parte especificada** das linhas retornadas por uma consulta SQL.

Sintaxe:

```
SELECT coluna(s)
FROM tabela
[ORDER BY coluna]
[LIMIT {contagem | ALL}]
[OFFSET deslocamento]   →   (Quantas linhas devem ser puladas
                            antes de iniciar a contagem do LIMIT)
```

Exemplos:

```
SELECT * FROM Produtos
ORDER BY preco
LIMIT 4;      (obtem apenas os primeiros 4 resultados)
```

Ou:

```
SELECT * FROM Produtos
ORDER BY preco
LIMIT 4       (obtem apenas os primeiros 4 resultados)
OFFSET 2;     (ignora os dois primeiros resultados;)
```

OPERADORES DE COMPARAÇÃO:

Comparar dois valores e retornar um valor booleano, dependendo do resultado da comparação.

<	menor que
>	maior que
<=	menor ou igual a
>=	maior ou igual a
=	igual a
<> ou !=	diferente de (não igual a)

Exempo:

```
SELECT nome_produto, qtde_estoque
FROM produtos
WHERE qtde_estoque >= 10 AND qtde_estoque <=20;
```

OPERADOR DE COMPARAÇÃO BETWEEN:

Operador de comparação, ele permite visualizar no resultado das consultas filtros de intervalos de dados.

Sintaxe:

```
SELECT…
FROM…
WHERE coluna BETWEEN valor1 AND valor2;
```

Exemplos:

```
SELECT nome_produto, preco
FROM produtos
WHERE preco BETWEEN 10.00 AND 20.00;
```

Ou:

```
SELECT nome_produto, preco
FROM produtos
WHERE preco NOT BETWEEN 5.00 AND 12.00;
```

UPDATE:

Atualizar registros da tabela

Sintaxe:

```
UPDATE tabela
SET coluna = novo_valor
WHERE coluna = valor indice;
```

Exemplos:

```
UPDATE produtos
SET descricao = 'desodorante aerosol axe black 90ml'
WHERE cod_produto = 11;
```

Ou:

```
UPDATE produtos
SET qtde_estoque = qtde_estoque -4
WHERE preco > 15.00;
```

Ou:

```
UPDATE produtos
SET preco = preco * 1.10;
```

DELETE:

https://blog.betrybe.com/sql/sql-delete/

DELETE FROM e TRUNCATE TABLE: Apaga linhas de uma tabela.

DELETE FROM:Excluir linhas específicas de uma tabela.

Sintaxe:

```
DELETE FROM nome_tabela
WHERE condições/exclusão;
```

TRUNCATE TABLE: Limpar os registros de uma tabela, porém não exclui a tabela

Sintaxe:

```
TRUNCATE TABLE nome_tabela;
```

Exemplos:

```
DELETE FROM produtos
WHERE cod_produto = 12;
```

Ou:

```
TRUNCATE TABLE pedidos;
```

FUNÇÕES DE AGREGAÇÕES:

Computar um **valor único** a partir de um conjunto de valores de entrada.

Funções básicas:

COUNT	(contagem)
MAX	(máximo)
MIN	(mínimo)
AVG	(média)
SUM	(soma)

Exemplos:

SELECT COUNT (*) **FROM** clientes;

Ou:

SELECT COUNT (**DISTINCT** nome_produtos) **FROM** produtos;
(**DISTINCT =** Conta apenas nomes diferentes, nomes iguais não contabilizam)

Ou:

SELECT COUNT (nome_produto) **FROM** produtos
WHERE preco >= 10.00;

Exemplos:

SELECT MAX (preco) **FROM** produtos;

Ou:

SELECT MIN (preco) **FROM** produtos;

Ou:

SELECT SUM (preco) **FROM** produtos;

Ou:

SELECT AVG (preco) **FROM** produtos;

Ou:

SELECT ROUND (**AVG** (preco),2) **FROM** Produtos;

Ou:

UPDATE produtos
SET preco = **ROUND** ((preco * 1.10),2);

CRIAR ALIAS com AS:

Podemos dar um nome diferente a uma coluna ou tabela ao realizar uma consulta usando um Alias. A sintaxe **AS** é opcional (como mostra no último exemplo).

Sintaxe:

SELECT coluna1 **AS** alias_coluna1,
coluna 2 **AS** alias_coluna2
FROM tabela **AS** alias_tabela;

Exemplos:

SELECT nome_cliente **AS** Cliente,
sobrenome_cliente **AS** Sobrenome
FROM clientes **AS** tabelaClientes;

Ou:

SELECT cod_pedido AS “Código do Pedido”
FROM pedidos;

Ou:

```
SELECT cod_pedido AS "Código do Pedido",
qtde AS "Quantidade"
FROM pedidos AS P
ORDER BY "Quantidade" DESC;
```

Ou:

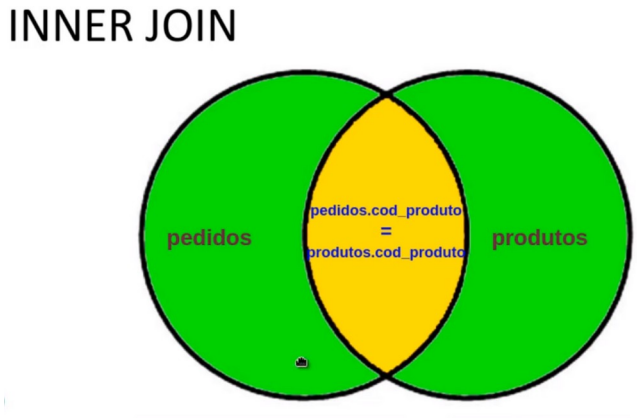
```
SELECT cod_pedido "Código do Pedido",
qtde "Quantidade"
FROM pedidos P
ORDER BY "Quantidade" DESC;
```

JOINS:

Claúsulas **JOIN** (junções) são usadas para **combinar dados provenientes de duas ou mais tabelas em um único conjunto de resultados**, baseado em condições de **join** especificadas.

INNER JOIN: Retorna linhas quando houver pelo menos uma correspondência em ambas as tabelas.

Outer JOIN: Retorna linhas mesmo quando não houver pelo menos uma correspondência em uma das tabelas (ou ambas). O **OUTER JOIN** divide-se em **LEFT JOIN**, **RIGHT JOIN** e cross **JOIN**.



→ A clausula **ON** determina a condição do join, que indica como as tabelas devem ser comparadas

→ No geral, a comparação ocorre por meio de um relacionamento entre c**have primária na primeira tabela** e **chave estrangeira na segunda tabela**.

→ Uma condição de join nomeia uma coluna em cada tabela envolvida no join e indica como as colunas devem ser comparadas.

→ No geral, usamos o operador **=** para obter linhas com colunas correspondentes. É comum usar o relacionamento de **PK** de uma tabela com **FK** de outra tabela.

Nomes de Colunas Qualificados:

Nome de coluna qualificado: Nome da coluna precedido pelo nome da tabela à qual pertence, separados por um ponto.

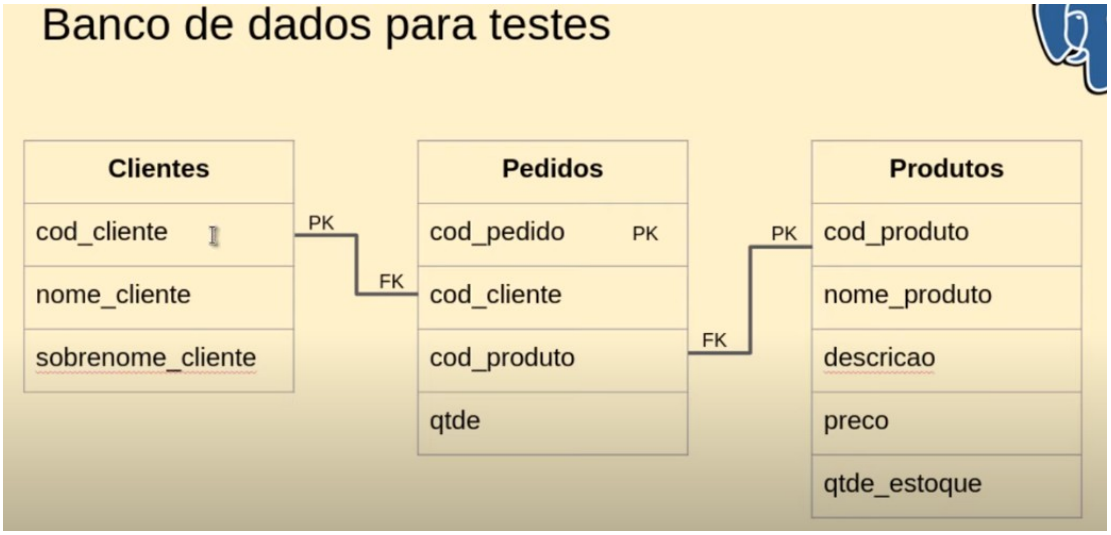
Usamos nomes de colunas qualificados para identificar a qual tabela cada campo envolvido pertence.

Isso evita erro de ambiguidade caso uma coluna tenha o mesmo nome em duas tabelas diferentes.

ON pedidos.cod_produto = produto.cod_produto

Sintaxe Simples:

SELECT colunas
FROM tabela1 // (priorizar a tabela que tem chave estrangeira)
[**INNER**] **JOIN** tabela2
ON tabela1.coluna = tabela2.coluna
[**INNER**] **JOIN** tabelaN
ON tabela1.coluna=tabelaN.coluna
[**WHERE** condições_filtragens]



Exemplos:

SELECT pedidos.cod_pedido, produtos.nome_produto, pedidos.qtde
FROM pedidos
INNER JOIN produtos
ON pedidos.cod_produto = produtos.cod_produto;

Ou:

SELECT PE.cod_pedido, **PR**.nome_produto, **PE**.qtde
FROM pedidos **PE**
INNER JOIN produtos **PR**
ON PE.cod_produto = **PR**.cod_produto;

Ou:

```
SELECT PE.cod_pedido, PR.nome_produto, PE.qtde
FROM pedidos PE
INNER JOIN produtos PR
    ON PE.cod_produto = PR.cod_produto
WHERE PE.cod_produto = 9;
```

Ou:

```
SELECT PE.cod_pedido, CL.nome_cliente, PR.nome_produto,
PE.qtde
FROM pedidos PE
INNER JOIN produtos PR
    ON PE.cod_produto=PR.cod_produto
INNER JOIN clientes CL
    ON PE.cod_cliente=CL.cod_cliente;
```

Ou:

```
SELECT PE.cod_pedido, CL.nome_cliente, PR.nome_produto
FROM pedidos PE
INNER JOIN produtos PR
    ON PE.cod_produto=PR.cod_produto
INNER JOIN clientes CL
    ON PE.cod_cliente=CL.cod_cliente
WHERE CL.cod_cliente = 1;
```

VIEWS

https://www.devmedia.com.br/alteracao-e-exclusao-da-view-views-no-sql-server-parte-2/22391

View(exibição/visão) é uma tabela virtual (estrutura de dados), baseada no conjunto de resultados de uma consulta SQL, criada a partir de um conjunto de tabelas (ou outras views) presentes no banco, que serve como tabelas-base.

Mostra sempre resultados de dados atualizados, pois o motor do banco de dados recria os dados toda vez que um usuário consulta a visão.

Aplicações das Views:

→ Simplificar o acesso a dados que estão armazenados em múltiplas tabelas relacionadas

→ Implementar segurança nos dados de uma tabela, por exemplo criando uma visão que limite os dados que podem ser acessados, por meio de uma cláusula WHERE.

→ Prove isolamento de uma aplicação da estrutura específica de tabelas do banco acessado.

Sintaxe:

```
CREATE OR REPLACE VIEW nomeView AS
```

Exemplo:

```
CREATE OR REPLACE VIEW vendas AS
SELECT CL.nome_cliente "Cliente", PR.nome_produto "Produto",
PE.qtde "Quantidade", PE.cod_pedido "Pedido",
PR.preco * PE.qtde "Fatura"
FROM  pedidos PE
INNER JOIN clientes CL
        ON PE.cod_cliente = CL.cod_cliente
INNER JOIN produtos PR
        ON PE.cod_produto = PR.cod_produto;
```

Ou:

```
SELECT "Pedido", "Produto", "Fatura" FROM vendas;
```

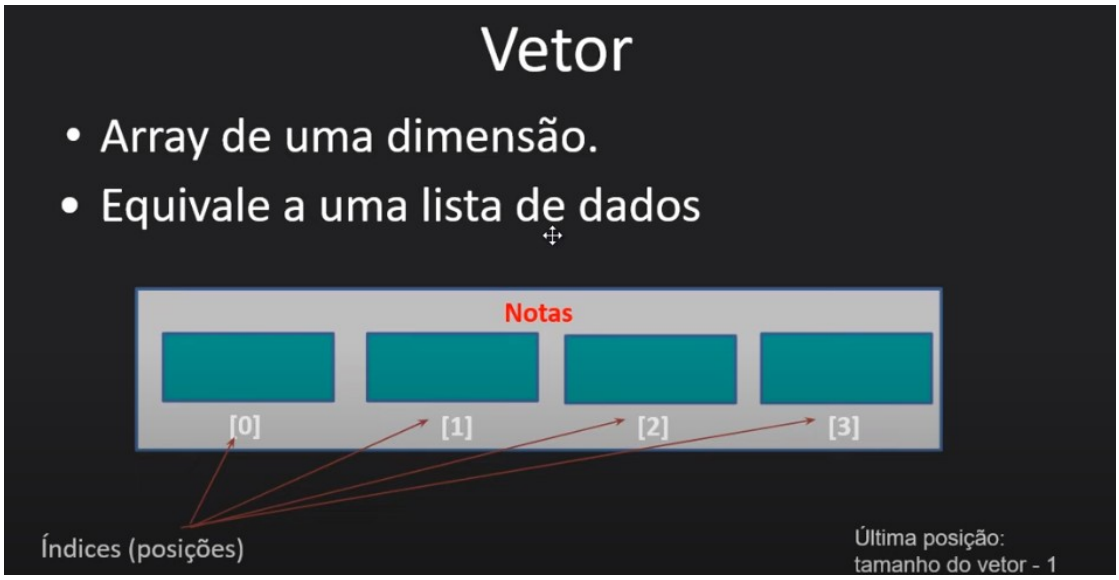
Ou:

```
SELECT  “Cliente”, SUM (“Fatura”)
FROM vendas
GROUP BY “Cliente”;
```

ALTERAR NOME DA VIEW:

```
ALTER VIEW vendas RENAME TO faturas;
```

COMO USAR ARRAYS EM COLUNAS DE TABELAS:



Sintaxe PostgreSQL:

nome_coluna tipo []

Sintaxe ANSI SQL:

nome_coluna tipo ARRAY []

A contagem feita no ARRAY no postgres começa em 1 e não em 0 como no padrão de array.

Exemplo:

```
CREATE TABLE escalatrabalho (
codFunc smallint,
escala char(3)[]);
```

Após:

```
INSERT INTO escalatrabalho(codFunc, escala)
VALUES
(1,'{"SEG", "TER", "QUA"}'),
(2,'{"QUI", "SEX", "SAB"}'),
(3,'{"SEG","QUA", "SEX"}');
```

#Escala de trabalho do funcionário 2

```
SELECT escala FROM escalatrabalho
WHERE codFunc =2;
```

#Qual dia da semana cada funcionário começa a trabalhar.

```
SELECT codFunc, escala[1]
FROM escalatrabalho;
```

#Qual funcionário trabalha na Quarta-feira?

```
SELECT codFunc
FROM escalatrabalho
WHERE 'QUA' = ANY (escala);     (ANY= qualquer)
```

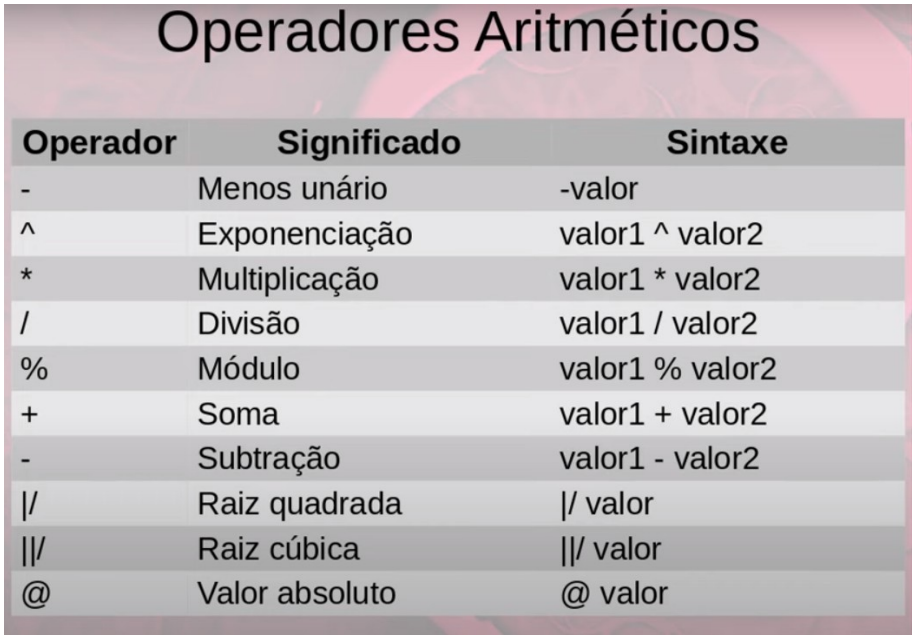
Alterar a escala de trabalho do funcionário 3

```
UPDATE escalatrabalho
SET escala = '{"QUA", "QUI", "SEX"}'
WHERE codFunc = 3;
```

Ou:

```
UPDATE escalatrabalho
SET escala = ARRAY ['TER','QUA','SAB']
WHERE codFunc = 3;
```

OPERADORES ARITMÉTICOS:



Operadores Aritméticos

Operador	Significado	Sintaxe
-	Menos unário	-valor
^	Exponenciação	valor1 ^ valor2
*	Multiplicação	valor1 * valor2
/	Divisão	valor1 / valor2
%	Módulo	valor1 % valor2
+	Soma	valor1 + valor2
-	Subtração	valor1 - valor2
|/	Raiz quadrada	|/ valor
||/	Raiz cúbica	||/ valor
@	Valor absoluto	@ valor

Sintaxe:

SELECT valor1 **Operador** valor2**;**

Exemplo:

SELECT 5+5 **AS** Resultado; (**AS** é opcional)

Ou:

```
SELECT preco * 10 FROM produtos
WHERE cod_produto = 10;
```

Ou:

```
SELECT ROUND((preco / 100),2) "CUSTO DE LUVA UNITÁRIO"
FROM produtos
WHERE cod_produto = 2;
```

Ou:

```
SELECT SUM(preco * qtde_estoque)
FROM produtos;
```

CRIAR TABELAS HERDADAS:

https://www.devmedia.com.br/implementando-heranca-de-tabela-no-sql-server/12758

É um conceito da orientação à objetos onde, uma determinada classe herda características (no caso das classes - propriedades ou atributos e métodos ou funções) de uma outra "super" classe.

Exemplo:

```
CREATE TABLE publicacao (
id serial constraint pk_id_pub primary key,
nome varchar(50),
dataPub date,
idioma varchar(25)
);
```

Após:

```
CREATE TABLE livros (
isbn13 char (13) unique,
tipo_capa varchar(20),
edicao smallint)
inherits(publicacao);
```

Após:

```
CREATE TABLE revistas (
issn char(8),
numero smallint,
unique (issn, numero))
inherits(publicacao);
```

Para verificar:

```
SELECT * FROM publicacao;
SELECT * FROM livros;
SELECT * FROM revistas;
```

Após:

```
INSERT INTO publicacao (nome, dataPub, idioma)
VALUES ('Le Monde','20220411','Francês');
```

Após:

```
INSERT INTO livros
(nome, dataPub, idioma, isbn13, tipo_capa, edicao)
VALUES
('50 ideias de Química','20220202','Português',
'9786555356519','Brochura',2);
```

Após:

```
INSERT INTO revistas
(nome, dataPub, idioma, issn, numero)
VALUES('Saber Eletrônica','19971201','Português','01016717',299);
```

Após:

```
ALTER TABLE publicacao
ADD COLUMN nPaginas smallint;
```

Após:

```
UPDATE publicacao
SET npaginas = 629
WHERE id = 3;
```

COMO USAR TIPO JSON EM COLUNAS:

JSON: JavaScript Object Notation

→ Formato aberto e popular para representação e troca de dados, assim como XML e YAML (porém mais leve e simples.)
→ É fácil para humanos ler e editar esse formato, e simples para os computadores processarem e gerarem dados JSON
→ É independente de linguagem de programação.

DOCUMENTOS JSON:

→ Um documento (objeto) JSON é um conjunto não-ordenado de dados armazenados em um par **"nome" : valor** (campo), que inica e termina com chaves **{}**.
→ Todos os nomes-chave são englobados em **aspas duplas**, e são separados de seus valores por dois pontos **:**.
→ Os pares (campos) são separados um do outro por vírgulas.

Exemplo:

```
CREATE TABLE peds (
id serial NOT NULL PRIMARY KEY,
pedido json NOT NULL
);
```

Após:

```
INSERT INTO peds (pedido)
VALUES
('{"comprador": "Fábio", "produtos": {"bebida": "Suco de Caju",
"comida": "Pizza de Atum"}}'),
('{"comprador": "Mônica", "produtos": {"bebida": "Água Tônica",
"comida": "Beirute"}}'),
('{"comprador": "Lauro", "produtos": {"bebida": "Campari", "comida":
"Espaguete"}}')
('{"comprador": "Henrique", "produtos": {"bebida": "Coca-cola",
"comida": "feijoada"}}');
```

Consultas:

```
SELECT * FROM peds;
```

OPERADORES JSON:

Exemplo:

```
SELECT pedido → 'comprador' AS Comprador
FROM peds;                    (→ Retorna em forma de JSON)
```

Ou:

```
SELECT pedido -» 'comprador' AS Comprador
FROM peds;                    (-» Retorna em forma de string)
```

Retorna as bebidas vendidas:

```
SELECT pedido -> 'produtos' ->> 'bebida' AS "Bebida Vendida"
FROM peds;
```

Pesquisa quem comprou Coca-cola

```
SELECT pedido ->> 'comprador' AS Comprador
FROM peds
WHERE pedido -> 'produtos' ->> 'bebida' = 'Coca-cola';
```

FUNÇÕES JSON:

Exemplo:

```
SELECT json_each(pedido)
FROM peds;
```

Ou:

```
SELECT json_each_text(pedido)
FROM peds;
```

Ou:

```
SELECT json_object_keys(pedido->'produtos')
FROM peds;
```

COMO CRIAR MATERALIZED VIEWS

→ Tabela virtual (lógica) cujo conteúdo é baseado no retorno de uma consulta pré-definida, realizada em uma ou mais tabelas (ou outras views), mas que não contém os dados em si.

→ Simplificar as consultas e controlar o acesso aos dados de forma simplificada para os usuários com permissões.

→ Permite criar uma visão mais lógica da modelagem do banco para os usuários.

→ O conjunto de dados da view é gerado no momento em que ela é executada.

MATERIALIZED VIEW:

→ É um objeto de db que armazena o resultado de uma consulta de forma persistente. É uma tabela auxiliar que permite maior performance no acesso aos dados.

Exemplo: (**AO INVÉS DÊ**)

```
SELECT clientes.nome_cliente, produtos.nome_produto,
pedidos.qtde
FROM pedidos
JOIN clientes
ON pedidos.cod_cliente = clientes.cod_cliente
JOIN produtos
ON pedidos.cod_produto = produtos.cod_produto
ORDER BY clientes.nome_cliente;
```

Fazer:

```
CREATE MATERIALIZED VIEW view_produto AS
SELECT clientes.nome_cliente, produtos.nome_produto,
pedidos.qtde
FROM pedidos
JOIN clientes
ON pedidos.cod_cliente = clientes.cod_cliente
JOIN produtos
ON pedidos.cod_produto = produtos.cod_produto
ORDER BY clientes.nome_cliente
WITH NO DATA / WITH DATA;    (inicia a tabela sem dados / com
                                                    dados)
```

Após criar a view com **WITH NO DATA**:

```
REFRESH MATERIALIZED VIEW view_compras;
```

Mudar nome de coluna:

```
ALTER MATERIALIZED VIEW view_compras
RENAME COLUMN nome_produto TO Produtos;
```

BACKUP E RESTAURAÇÃO DO DB COM PG_DUMP:

Sintaxe:

```
sudo -u postgres pg_dump nome_banco > nome_backup;
```

Exemplo:

```
sudo -u postgres pg_dump ej_informatica > ej_informatica_backup;
```

EXCLUIR UM BANCO DA DADOS:

Utilizar o **DROP DATABASE:**

```
DROP DATABASE ej_informatica;
```

Caso o Banco de dados não exclua por estar sendo utilizado, fazer:
(Com esse comando, o banco de dados não pode ser reconectado após)

```
REVOKE CONNECT ON DATABASE ej_informatica FROM public;
```

Terminar as conexões existentes no banco (para nenhum outro usuário utilizar no momento da tentativa de exclusão do db)

```
SELECT pg_terminate_backend(pg_stat_activity.pid)
FROM pg_stat_activity
WHERE pg_stat_activity.datname = 'ej_informatica';
```

Por fim:

```
DROP DATABASE ej_informatica;
```

RESTAURAR UM BANCO DE DADOS:

Criar um banco de dados para base da exportação:

```
CREATE DATABASE ej_informatica TEMPLATE template0;
```

Para restaurar (no terminal):

```
sudo -u postgres psql ej_informatica < ej_informatica_backup
```

SUBCONSULTAS (SUBQUERIES):

http://www.bosontreinamentos.com.br/bancos-de-dados/o-que-e-uma-subconsulta-sql-subquery/

Uma subconsulta é uma consulta embutida dentro de outra consulta, de forma aninhada, passando os resultados da consulta mais interna para a consulta mais externa por meio de uma cláusula **WHERE** ou de uma cláusula **HAVING**.

Sintaxe base:

```
SELECT coluna(s)
FROM tabela(s)
WHERE coluna operador (SELECT coluna
FROM tabela WHERE condições);
```

Exemplo: (Me retorna o maior preço do livro)

```
SELECT nome_produto, preco
FROM produtos
WHERE preco = (
        SELECT MAX (preco)
        FROM produtos
);
```

Atualizar os preços:

```
UPDATE tbl_livros
SET preco_livro = preco_livro * 1.15
WHERE editora = (
        SELECT id_editora
        FROM tbm_editoras
        WHERE nome_editora LIKE '%Microsoft%'
);
```

FUNÇÕES DE MANIPULAÇÕES DE STRINGS:

Concatenação de strings: Operador **||**

Retorna um valor de texto (string). Junta os dois textos

Sintaxe:

string ou não-string || string ou não-string

Exemplo:

```
SELECT nome_cliente || ' ' || sobrenome_cliente
FROM clientes
ORDER BY nome_cliente;
```

Ou:

```
SELECT sobrenome_cliente || ', ' || nome_cliente
FROM clientes
ORDER BY sobrenome_cliente;
```

Ou:

```
SELECT 'O produto' || ' ' || nome_produto || ' ' ||'custa' || ' ' || preco
FROM produtos;
```

Ou:

```
SELECT 'O produto ' || nome_produto || ' custa ' || preco
FROM produtos;
```

CONCAT ()

Outra opção para concatenação de strings é o emprego da função CONCAT(), disponível desde a versão 9.1 do PostgreSQL.

Sintaxe:

```
CONCAT (string1,strin2)
```

Exemplo:

```
SELECT CONCAT (nome_cliente,' ',sobrenome_cliente)
FROM clientes
ORDER BY nome_cliente;
```

Ou:

```
SELECT CONCAT ('O produto', ' ', nome_produto, ' ','custa', ' ', preco)
FROM produtos;
```

Ou:

```
SELECT CONCAT ('O produto ', nome_produto, ' custa ', preco)
FROM produtos;
```

FUNÇÃO bit_lenght

Retorna o número de bits em uma string

Sintaxe:

```
bit_lenght (string);
```

Exemplo:

```
SELECT nome_cliente, bit_length(nome_cliente)
FROM clientes
WHERE cod_cliente = 1;
```

FUNÇÃO char_lenght

Retorna o número de caracteres em uma string

Sintaxe:

```
char_lenght (string);
```

Exemplo:

```
SELECT nome_cliente, char_length(nome_cliente)
FROM clientes
WHERE cod_cliente = 1;
```

FUNÇÃO LENGHT()

Também retorna o número de caracteres em uma string

Sintaxe:

```
lenght (string);
```

Exemplo:

```
SELECT nome_cliente, length(nome_cliente)
FROM clientes
WHERE cod_cliente = 1;
```

FUNÇÃO lower()

Retorna a string convertida em caixa baixa

Sintaxe:

```
lower (string);
```

Exemplo:

```
SELECT nome_cliente, lower(nome_cliente)
FROM clientes
WHERE cod_cliente = 1;
```

FUNÇÃO upper()

Retorna a string convertida em caixa alta

Sintaxe:

```
upper (string);
```

Exemplo:

```
SELECT nome_cliente, upper(nome_cliente)
FROM clientes
WHERE cod_cliente = 1;
```

FUNÇÃO overlay()

Substitui uma **substring** (sequência de caracteres que é parte da string)

Sintaxe:

```
overlay (string placing string from int [for int])        //(for é a
                                                    quantidade de caracteres desejados)
```

Exemplo:

```
SELECT overlay ('Erbelt' placing 'lber' from 2 for 4);
```

FUNÇÃO position()

Fornece a localização de uma **substring** especificada.

Sintaxe:

```
position (substring in string)
```

Exemplo:

```
SELECT position('Treina' in 'Elbert Treinamentos');
```

Ou:

```
SELECT cod_produto, position ('Luv' in nome_produto)
FROM produtos;
```

FUNÇÃO substring()

Extrai uma **substring** (sequência de caracteres)

Sintaxe:

```
substring (string [from int] [for int])
```

Exemplo:

```
SELECT substring('Elbert Jean' from 8);
```

Ou:

```
SELECT substring('Elbert Jean' from 7 for 3);
```

Ou:

```
SELECT substring(nome_produto from 1 for 3)
FROM produtos;
```

Combinação:

```
SELECT upper(substring(nome_produto from 1 for 3))
FROM produtos;
```

FUNÇÃO trim()

Remove a **substring** contendo os caracteres informados no início, final ou em ambos os extremos da string (por padrão remove espaços)

Sintaxe:

```
trim([leading | trailing | both] [caracteres] from string)
```

Exemplo:

```
SELECT trim(both from '      Elbert     ');
```

Ou:

```
INSERT INTO produtos (nome_produto)
VALUES
trim ('     Vassoura sem cabo  ');
```