Material de Banco de dados Relacional

O objetivo deste material é dar uma pequena explicação sobre o que é banco de dados, scripts e informações que vã fazer seu conhecimento aumentar sobre bancos relacionais aumentar.

Banco de dados:

Existem os bancos de dados NoSQL (não relacionais) e os relacionais.
Aqui focaremos nos bancos relacionais.

Os bancos de dados relacionais são um conjunto de arquivos relacionados entre si, com registros sobre pessoas, lugares ou coisas de maneira organizada que se relacionam para criar algum sentido (Informação). São de vital importância para empresas e há décadas se tornaram a principal peça dos sistemas de informação.

Um banco de dados não é nada mais nada menos do que um arquivo ou conjunto de arquivo que guardam algum tipo de informação para ser usada, lida ou alterada posteriormente.

Eles permitem o armazenamento e manipulação de dados organizados em forma de tabelas. Suas tabelas são uma forma de organização de dados formada por linhas e colunas, onde podemos dizer que sua estruturação é bem parecida com as planilhas Microsoft Excel.

O que é um SGBD?

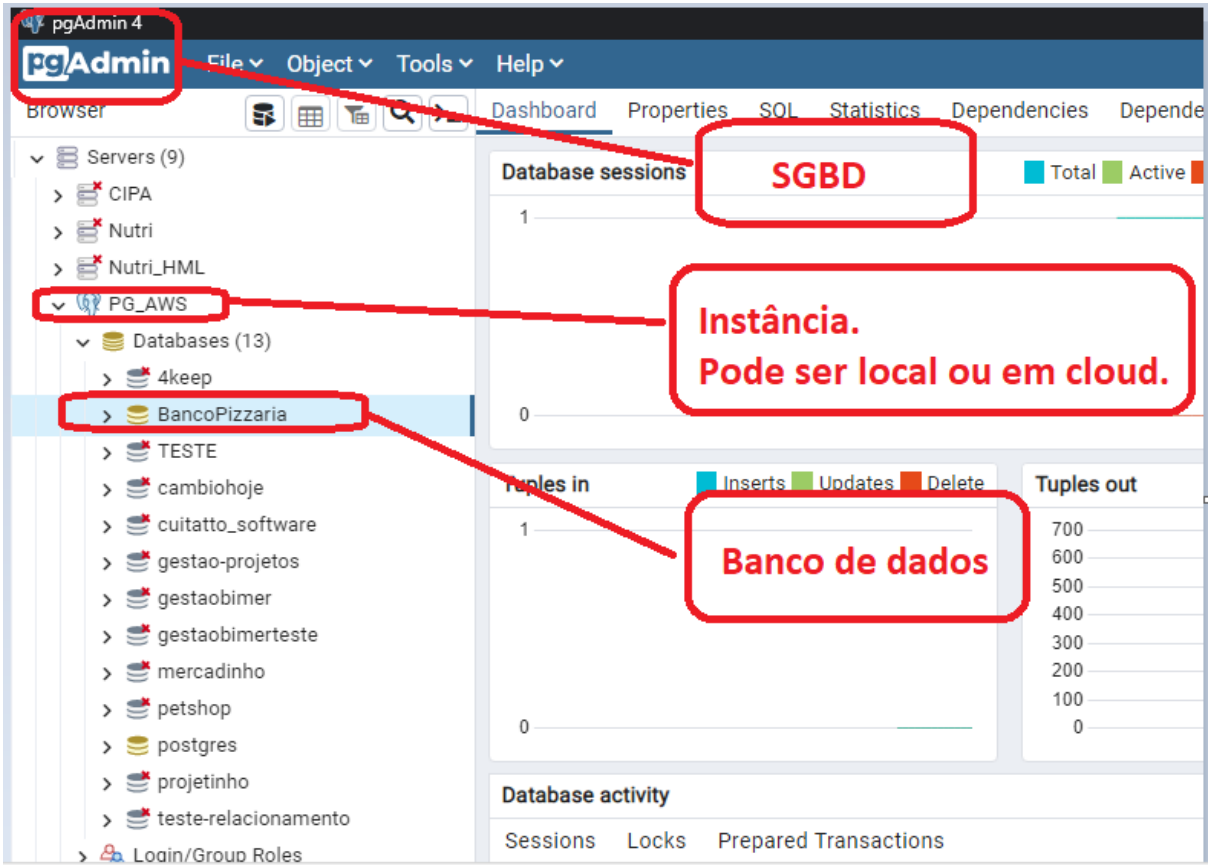
O SGBD é um Sistema Gerenciador de Banco de Dados, ele não é um banco de dados, mas sim um complemento, um grupo de programas para interação com os dados.
No caso do PostgreSQL, utilizamos o PgAdmin como oficial, mas existem outros que conseguem fazer essa função como DBeaver e Azure DataStudio.
Com o SGBD, podemos criar, manipular vários bancos de dados adicionados em várias instâncias.

O que é Instância do banco de dados?

Uma instância é um serviço instalado na máquina apontando para alguma porta, essa porta por padrão no postgre é a 5432, mas pode ser alterada caso essa porta já esteja sendo utilizada.

Cada instância pode gerenciar vários bancos de dados simultaneamente. Cada computador pode executar várias instâncias simultaneamente, exemplo: poderíamos ter uma utilizando a porta 5432 e outra na porta 5433.

Os aplicativos conectam à instância para executar trabalhos em um banco de dados gerenciado por ela.



Introdução ao SQL (Structured Query Language)

A linguagem SQL, é uma linguagem estruturada de consulta formada pelo conjunto das linguagens:

DDL (Data Definition Language) Linguagem de Definição de Dados:

- *CREATE: Cria uma estrutura*
- *ALTER: Altera uma estrutura*
- *DROP: Exclui uma estrutura*

DML (Data Manipulation Language) Linguagem de Manipulação de Dados:

- *INSERT : Insere dados*
- *UPDATE: Altera dados*
- *DELETE: Exclui dados*

DQL (Data Query Language) Linguagem de Consulta de Dados:

- *SELECT - Consulta de dados*
- *ORDER BY -Ordenação de dados*
- *GROUP BY - Agrupamento de dados*
- *COUNT, MIN, MAX e outros -* Fun*ções GERAIS*
- *WHERE Filtros de seleção*

DCL (Data Control Language) Linguagem de Controle de Dados

- GRANT : Habilita acesso a dados e operações
- REVOKE: Revoga acesso a dados e operações

Obs: *É geralmente usado pelos DBAs, não se preocupem com esses comandos.*

DTL (Data Transaction Language) Linguagem de Transação de Dados

- *START TRANSACTION: Inicia a transação*
- *COMMIT : Concretiza a transação*
- *ROLLBACK: Anula a transação*

Obs: *É muito utilizado no SQL Server, mas não veremos no momento.*

O que são relacionamentos e chaves?

Os relacionamentos são ligações entre as tabelas de forma planejada para fazer algum sentido baseado na regra de negócio a qual o banco foi planejado. Com isso podemos ter:

- **Chave Primária (Primary Key, PK)**: Coluna com valores únicos.
    - Geralmente é a coluna Id da tabela
    - Geralmente é auto incremente, no postgre é do tipo serial

    .
- **Chave Composta:** Composição de duas ou mais colunas para gerar uma combinação única.

- **Chave Estrangeira (Foreign Key, FK):** Coluna que armazena a chave primária de outra tabela.
    - Geralmente essa coluna leva o nome do id e a tabela de relacionamento.

Também é comum existir bancos onde quem projetou não queira utilizar a integridade relacional entre as tabelas, com isso, o banco não contem chave estrangeira.
Nesses casos o relacionamento é feito de outra forma, usando relacionamentos de IdEntidadeOrigem e NmEntidadeOrigem. Não falaremos disso agora.

Principais tipos no PostgreSQL

O postgre contem muitos tipos de dados, por isso recomento que leia a documentação para conhecê-los bem e saber quando e como utilizá-los.
Documentação: https://pgdocptbr.sourceforge.io/pg80/datatype.html

Segue abaixo os principais tipos, aqueles que são mais utilizados nas aplicações no dia a dia.

Nome	Aliases (Apelido)	Descrição
serial	serial4	inteiro de quatro bytes com auto-incremento
character varying [(n)]	varchar [(n)]	cadeia de caracteres de comprimento variável
text		cadeia de caracteres de comprimento variável
timestamp [(p)] [without time zone]		data e hora
numeric [(p, s)]	decimal [(p, s)]	numérico exato com precisão selecionável
integer	int, int4	inteiro de quatro bytes com sinal
money		quantia monetária
bigint	int8	inteiro de oito bytes com sinal
date		data de calendário (ano, mês,dia)
double precision	float8	número de ponto flutuante de precisão dupla
bigserial	serial8	inteiro de oito bytes com auto-incremento
character [(n)]	char [(n)]	cadeia de caracteres de comprimento fixo

O que são scripts (SQL)?

Os scripts, também chamados de querys, comandos, módulos ou batches são uma sequência de procedimentos executados no banco de dados para tomar alguma ação.

Mão na massa

Vamos agora aprender alguns scripts de exemplo para avançar em nosso conhecimento.

Vamos criar uma tabela onde possamos armazenar as informações de usuários.
Todo usuário deve conter:

- id (PK e Incremental)
- emai:l unico (Obrigatório)
- senha: até 10 caracteres (Obrigatório)
- cpf: unico (Obrigatório)
- data_cadastro (Opcional)
- observacao (Opcional)

Diante disso podemos usar o comando a baixo para criar a tabela:

```
CREATE TABLE public.usuario
(
    id bigserial NOT NULL,
    email character varying(100) NOT NULL,
    senha character varying(50) NOT NULL,
    cpf character varying(11) NOT NULL,
    data_cadastro time without time zone,
    observacao text,
    PRIMARY KEY (id),
    UNIQUE (email),
    UNIQUE (cpf)
);

ALTER TABLE IF EXISTS public.usuario
    OWNER to postgres;
```

Execute o comando em seu SGBD favorito e veja o resultado.

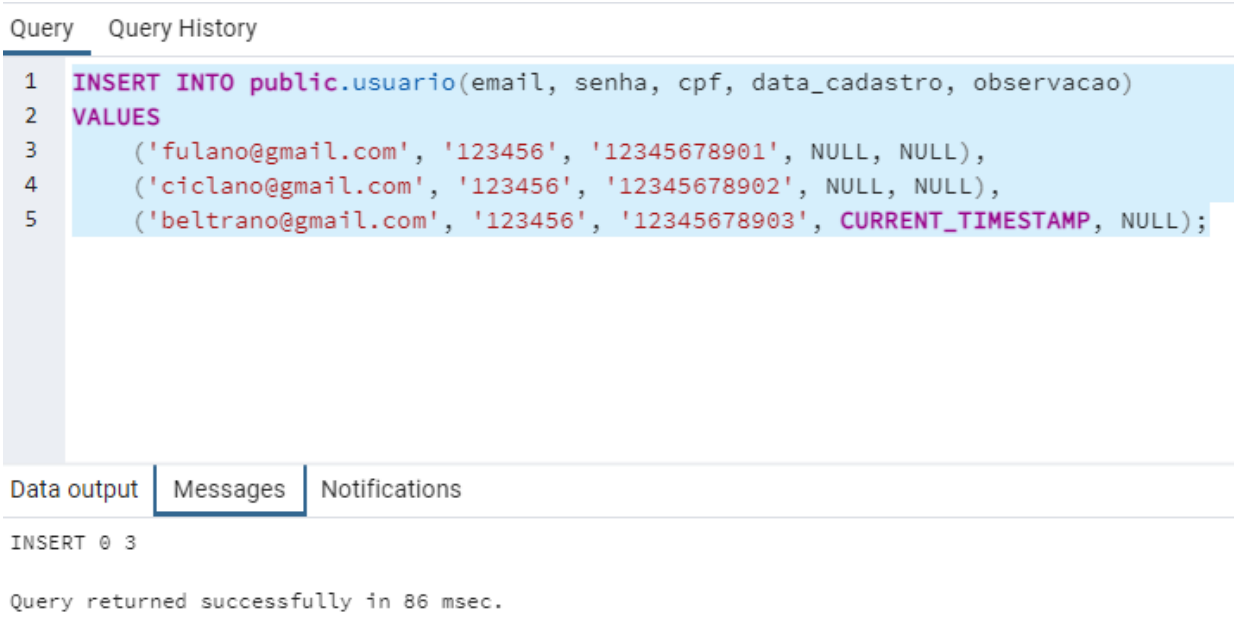
Após criar a tabela, vamos adicionar alguns funcionários, para isso, vamos usar o seguinte comando abaixo onde insere 3 registros de uma única vez:

```
INSERT INTO public.usuario(email, senha, cpf, data_cadastro, observacao)
VALUES
        ('fulano@gmail.com', '123456', '12345678901', NULL, NULL),
        ('ciclano@gmail.com', '123456', '12345678902', NULL, NULL),
        ('beltrano@gmail.com', '123456', '12345678903', CURRENT_TIMESTAMP, NULL);
```

Neste script acima usamos a função **CURRENT_TIMESTAMP** que retorna data e hora atual do servidor.

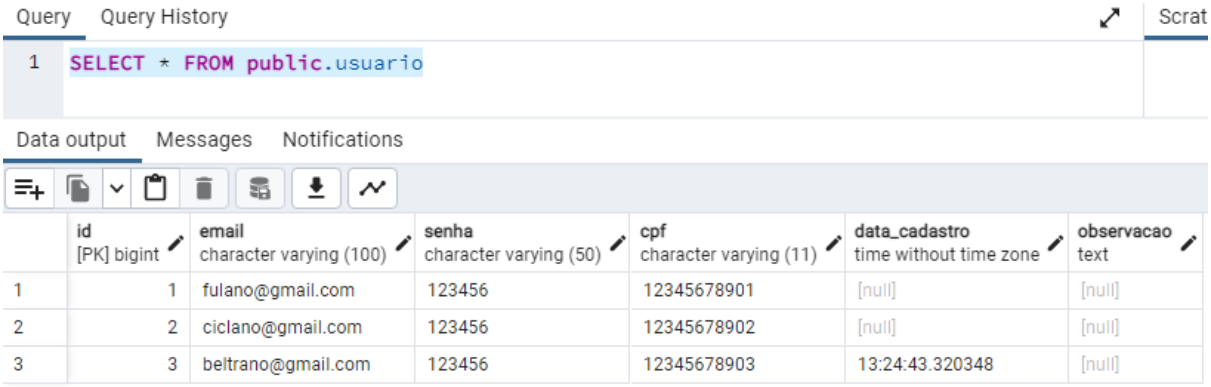
Neste script não informamos o Id, ele será gerado automaticamente pelo próprio banco.

Ao executar o script no banco, você terá uma mensagem informando que foram inseridos 3 registros, essa mensagem pode ter diferenças de SGBD para SGBD. No caso do PgAdmin 4 mostrará desta forma:



Após inserir os dados nesta tabela vamos fazer uma consulta básica:
Execute o seguinte comando: SELECT * FROM public.usuario

Seu resultado deve ser mais ou menos como este:



	id [PK] bigint	email character varying (100)	senha character varying (50)	cpf character varying (11)	data_cadastro time without time zone	observacao text
1	1	fulano@gmail.com	123456	12345678901	[null]	[null]
2	2	ciclano@gmail.com	123456	12345678902	[null]	[null]
3	3	beltrano@gmail.com	123456	12345678903	13:24:43.320348	[null]

Aqui veremos algumas cláusulas que podem ser utilizadas em consultas:

SELECT	-- Serve para buscar/consultar alguma informação.
FROM	-- De onde a informação vai ser consultada.
WHERE	-- Serve como filtro (significa Onde).
ORDER BY	-- Serve para fazer uma ordenação de uma ou mais colunas.
GROUP BY	-- Serve para agrupar dados de uma ou mais colunas.
IS NULL	-- Informo que o campo tem que ser nulo.
IS NOT NULL	-- Informo que o campo não pode ser nulo.
AS	-- Alias são apelidos que podemos dar para colunas e tabelas.
*	-- Caractere curinga, pega todas as colunas da tabela
LIKE ou ILIKE	-- Serve para encontrar por partes da informação
BETWEEN	-- Serve para encontrar registros entre uma condição
IN	-- Serve para encontrar registros dentro de uma condição
NOT IN	-- Serve para encontrar registros que não estão dentro de uma condição
EXISTS	-- Serve para encontrar registros caso uma condição retorne verdade.
AND	-- Significa que ambas as comparações devem ser verdadeiras.
OR	-- Significa que pelo menos uma das comparações tem que ser verdadeira.

Analisando uma consulta simples ao banco de dados:

```
SELECT - - Consulte ou Selecione.
        email - - O email do usuario.
FROM - - De onde.
        public.usuario - - Origem dos dados (Nome do Schema e Tabela)
```

Antes de evoluir mais, vamos criar mais algumas tabelas:

Vamos criar a tabela produto, a mesma deve conter:

- id: (PK e incremental)
- nome: (Obrigatório)
- quantidade: (Obrigatório - Só números inteiros)
- valor_custo: (Obrigatório)
- valor_venda: (Obrigatório)
- data_cadastro: (Opcional)
- observacao: (Opcional)

Com base nos campos nosso script poderia ser assim:

```
CREATE TABLE public.produto
(
    id bigserial NOT NULL,
    nome character varying NOT NULL,
    quantidade integer NOT NULL,
    valor_custo numeric NOT NULL,
    valor_venda numeric NOT NULL,
    data_cadastro timestamp without time zone,
    observacao text,
    PRIMARY KEY (id)
);

ALTER TABLE IF EXISTS public.produto
    OWNER to postgres;
```

Execute os comandos no seu sgbd favorito.

Agora vamos inserir alguns produtos:

```
INSERT INTO public.produto(nome, quantidade, valor_custo, valor_venda, data_cadastro,
observacao)
VALUES
        ('Camisa da Barcelona', 20, 70.0, 140.0, CURRENT_TIMESTAMP, NULL),
        ('Camisa da Real Madrid', 25, 70.0, 140.0, CURRENT_TIMESTAMP, NULL),
        ('Camisa do Flamengo', 0, 60.0, 120.0, CURRENT_TIMESTAMP, NULL),
        ('Camisa do Vasco', 0, 60.0, 120.0, CURRENT_TIMESTAMP, NULL),
        ('Camisa do Fluminense', 10, 60.0, 120.0, CURRENT_TIMESTAMP, NULL),
        ('Camisa do Botafogo', 2, 60.0, 120.0, CURRENT_TIMESTAMP, NULL);
```

Ao executar esta query, serão inseridas 6 camisas na tabela produtos.

Para ver as camisas já inseridas, execute o seguinte comando:

```
SELECT * FROM public.produto
```

Vamos agora criar nossa primeira tabela com relacionamento. Vamos criar a tabela de orçamento.

Esta tabela deve contar os seguintes informações:

- id (PK e incremental)
- id_usuario (FK) – id do usuário que fez o orçamento.
- data_cadastro - (Obrigatório)
- valor_desconto - (Obrigatório)
- valor_acrescimo - (Obrigatório)
- valor_total - (Obrigatório)

Com base nos campos nosso script poderia ser assim:

```
CREATE TABLE public.orcamento
(
    id bigserial NOT NULL,
    id_usuario bigserial NOT NULL,
    valor_desconto numeric NOT NULL,
    valor_acrescimo numeric NOT NULL,
    valor_total numeric NOT NULL,
    data_cadastro time without time zone NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT fk_orcamento_usuario FOREIGN KEY (id_usuario)
        REFERENCES public.usuario (id) MATCH FULL
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
);

ALTER TABLE IF EXISTS public.orcamento
    OWNER to postgres;
```

Vamos agora inserir alguns registros fictícios, esses registros serão utilizados em algumas das nossas consultas.

```
INSERT INTO public.orcamento(id_usuario, valor_desconto, valor_acrescimo, valor_total,
data_cadastro)
VALUES
        (1, 0.0, 0.0, 0.0, CURRENT_TIMESTAMP),
        (2, 20.0, 1.0, 0.0, CURRENT_TIMESTAMP),
        (2, 0.0, 10.0, 0.0, CURRENT_TIMESTAMP),
        (1, 10.0, 0.0, 0.0, CURRENT_TIMESTAMP),
        (2, 0.0, 50.0, 0.0, CURRENT_TIMESTAMP),
        (1, 0.0, 0.0, 0.0, CURRENT_TIMESTAMP);
```

Execute o comando a seguir para ver os registros inseridos:

```
SELECT * FROM public.orcamento
```

Seu resultado deve ser parecido com este aqui:

Query | Query History

```
1  SELECT * FROM public.orcamento
```

Data output | Messages | Notifications

	id [PK] bigint	id_usuario bigint	valor_desconto money	valor_acrescimo money	valor_total money	data_cadastro time without time zone
1	1	1	R$ 0,00	R$ 0,00	R$ 0,00	14:54:53.481393
2	2	2	R$ 20,00	R$ 1,00	R$ 0,00	14:54:53.481393
3	3	2	R$ 0,00	R$ 10,00	R$ 0,00	14:54:53.481393
4	4	1	R$ 10,00	R$ 0,00	R$ 0,00	14:54:53.481393
5	5	2	R$ 0,00	R$ 50,00	R$ 0,00	14:54:53.481393
6	6	1	R$ 0,00	R$ 0,00	R$ 0,00	14:54:53.481393

Agora que temos a tabela de orçamento, precisamos criar uma tabela para criar os itens desse orçamento, pois um orçamento poderá ter vários itens. Diante disso, vamos criar a tabela orcamento_item que deve ter as seguintes informações:

- id (PK e incremental)
- id_orcamento (FK) – id do orçamento
- id_produto (FK) – id do produto
- valor_venda - (Obrigatório)
- valor_desconto - (Obrigatório)
- valor_acrescimo - (Obrigatório)
- valor_total - (Obrigatório)

Para estes campos usaremos uma query que faça o relacionamento com outras duas tabelas. Neste caso, a tabela orcamento_item vai conter 2 FK, um apontando para a tabela de orçamento e outro para a tabela de produto.

É muito com ter tabelas com múltiplos relacionamentos.

Vejamos a seguir como ficaria a query para criação da tabela orcamento_item:

```
CREATE TABLE public.orcamento_item
(
    id bigserial NOT NULL,
    id_orcamento bigint NOT NULL,
    id_produto bigint NOT NULL,
    valor_venda numeric NOT NULL DEFAULT 0.0,
    valor_desconto numeric DEFAULT 0.0,
    valor_acrescimo numeric DEFAULT 0.0,
    valor_total numeric DEFAULT 0.0,
    PRIMARY KEY (id),
    CONSTRAINT fk_orcamento_item_orcamento FOREIGN KEY (id_orcamento)
        REFERENCES public.orcamento (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT fk_orcamento_item_produto FOREIGN KEY (id_produto)
        REFERENCES public.produto (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
);

ALTER TABLE IF EXISTS public.orcamento_item
    OWNER to postgres;
```

Agora vamos inserir alguns dados ficticios:

```
INSERT INTO public.orcamento_item(
id_orcamento, id_produto, valor_venda, valor_desconto, valor_acrescimo, valor_total)
VALUES
        (1, 1, 140, 10.0, 0.0, 130.0),
        (1, 2, 140, 20.0, 0.0, 120.0),
        (2, 1, 140, 0.0, 0.0, 140.0),
        (3, 3, 120, 10.0, 0.0, 110.0);
```

Consulte os registros inseridos com o comando
`SELECT * FROM public.orcamento_item`

Agora que temos várias tabelas e informações registradas, vams conhecer alguns comandos de consulta.

Cláusula WHERE:

Podemos fazer consultas filtrando alguns dados usando a cláusula WHERE, a mesma significa ONDE um determinado valor ou condição atende o solicitado.
Vejamos alguns exemplos:

```
-- Aqui buscamos o produto onde tenha seu id igual a 2.
SELECT * FROM public.produto WHERE id = 2

-- Aqui buscamos os produtos com quantidade maior que 10.
SELECT * FROM public.produto WHERE quantidade > 10

-- Aqui buscamos os produtos com quantidade maior ou igual a 20 E valor de venda igual a 120.0
SELECT * FROM public.produto
WHERE quantidade <= 20 AND valor_venda = 120

-- Aqui buscamos os produtos com quantidade maior ou igual a 20 OU valor de venda igual a 120.0
SELECT * FROM public.produto
WHERE quantidade = 20 OR valor_venda = 120

-- Aqui buscamos os produtos com quantidade entre 10 e 25
SELECT * FROM public.produto
WHERE quantidade BETWEEN 10 AND 25

-- Aqui buscamos os produtos QUE NÃO tenham quantidade entre 10 e 25
SELECT * FROM public.produto
WHERE quantidade NOT BETWEEN 10 AND 25

-- Aqui buscamos os produtos que começam com a letra F
O ILIKE é diferente do LIKE, pois ele não é case sensitive
SELECT * FROM public.produto
WHERE nome ILIKE 'f%'
```

Existem muitas formas de se utilizar a cláusula WHERE, mas durante os próximos exemplos vamos ir vendo.

Cláusula Order By:

A cláusula Order By pode ser executada de duas formas, Ascendente e Descendente.
Por padrão a mesma já ordena de forma Ascendente (do menor para o maior), porém podemos informar “Asc”. Para ordenar de forma Descendente (Do maior para o menor) temos que informar a palavra “Desc” ao final.

```
-- Ordenando de forma ascendente
SELECT * FROM public.produto
ORDER BY quantidade
'
-- Ordenando de forma descendente
SELECT * FROM public.produto
ORDER BY quantidade DESC
```

Podemos ordenar por qualquer coluna, ou podemos ordenar por mais de uma coluna.

Junções de tabelas ou Joins:

Joins são utilizados para fazer junções de tabelas e transformar o resultado em uma única tabela.

Tipos de Joins:

- **INNER JOIN** ou **JOIN**
- **LEFT OUTER JOIN** ou **SOMENTE LEFT JOIN:**
- **RIGHT OUTER JOIN** ou **SOMENTE RIGTH JOIN:**
- **CROSS JOIN**
- **CROSS APPLY**

INNER JOIN ou somente JOIN:

O INNER JOIN ou somente JOIN filtra tudo das tabelas informadas comparando o que está sendo especificado na cláusula ON.
A cláusula ON funciona como se fosse um WHERE, serve para comparar campos que sejam comum em ambas as tabelas.

LEFT OUTER JOIN ou somente LEFT JOIN:

A cláusula LEFT JOIN ou LEFT OUTER JOIN permite obter não apenas os dados relacionados de duas tabelas, mas também os dados não relacionados encontrados na tabela à esquerda da cláusula JOIN. Caso não existam dados relacionados entre as tabelas à esquerda e à direita, do JOIN, os valores resultantes de todas as colunas da lista de seleção da tabela à direita serão nulos.

RIGTH OUTER JOIN ou somente RIGTH JOIN:

Ao contrário do LEFT JOIN, a cláusula RIGHT JOIN ou RIGHT OUTER JOIN retorna todos os dados encontrados na tabela à direita de JOIN. Caso não existam dados associados

entre as tabelas à esquerda e à direita de JOIN, serão retornados valores nulos.

CROSS JOIN ou FULL JOIN:

Faz um produto Cartesiano do resultado, ou seja, multiplica tudo da tabela da esquerda com tudo da tabela da Direita.

CROSS APPLY (Não funciona em todos os bancos):

Funciona como o CROSS JOIN, mas tem algumas variações quando está trabalhando com outros tipos de informações como CTEs e TVF. Esse tipo de JOIN é mais utilizado no SQL SERVER.

Exemplo abaixo do uso do INNER JOIN obtendo todas as informações da tabela de orcamento com a tabela de orcamento_item

```
SELECT
	*
FROM
	public.orcamento O
INNER JOIN
	public.orcamento_item ON orcamento_item.id_orcamento = O.id
```

Exemplo de LEFT JOIN obtendo todos os usuários e os orçamentos que podem estar vinculados a eles.

```
SELECT
	*
FROM
	public.usuario U
LEFT JOIN
	public.orcamento ORC ON ORC.id_usuario = U.id
```

Exemplo de RIGHT JOIN onde pegamos todos os usuários e os orçamentos vinculado aos mesmos,

```
SELECT
	*
FROM
	public.orcamento O
RIGHT JOIN
	public.usuario ON usuario.id = O.id_usuario
```