



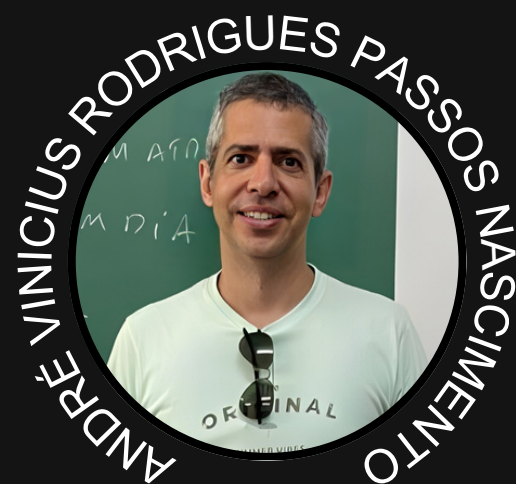
Structured Query Language

Aprenda **SQL** na prática

X SEMAC

Equipe

ORIENTADOR



COORDENADOR



MINISTRANTES



Conteúdo

TB_MINISTRANTE

ID	NOME
1	GUILHERME
2	IGOR
3	JONATHA
4	MARCOS

1

N

TB_ASSUNTO

ID	ID_MINISTRANTE	NOME
1	1	BANCO DE DADOS
2	1	A LINGUAGEM SQL
3	1	SQL SERVER
4	4	AGENDAPLUS
5	3	CREATE DATABASE E USE
6	3	CREATE TABLE
7	3	INSERT INTO
8	3	UPDATE
9	3	DELETE
10	1	SELECT
11	2	RESTRIÇÕES DE INTEGRIDADE
12	4	JUNÇÕES
13	4	FUNÇÕES DE AGREGAÇÃO
14	2	INTRODUÇÃO AO T-SQL



Repositório:

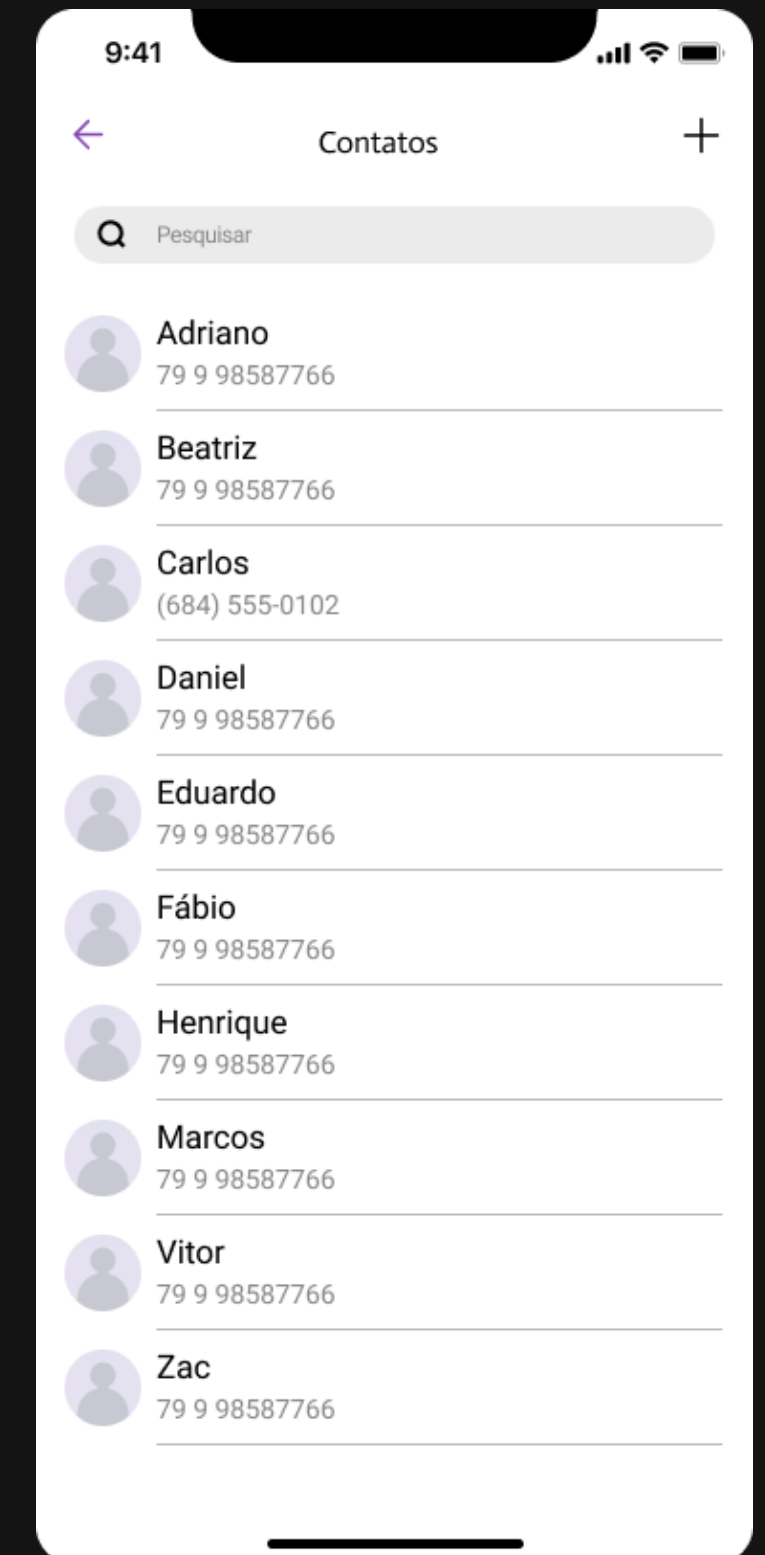
github.com/M4RCOSVS0/Curso-Introducao-SQL-XSEMAC



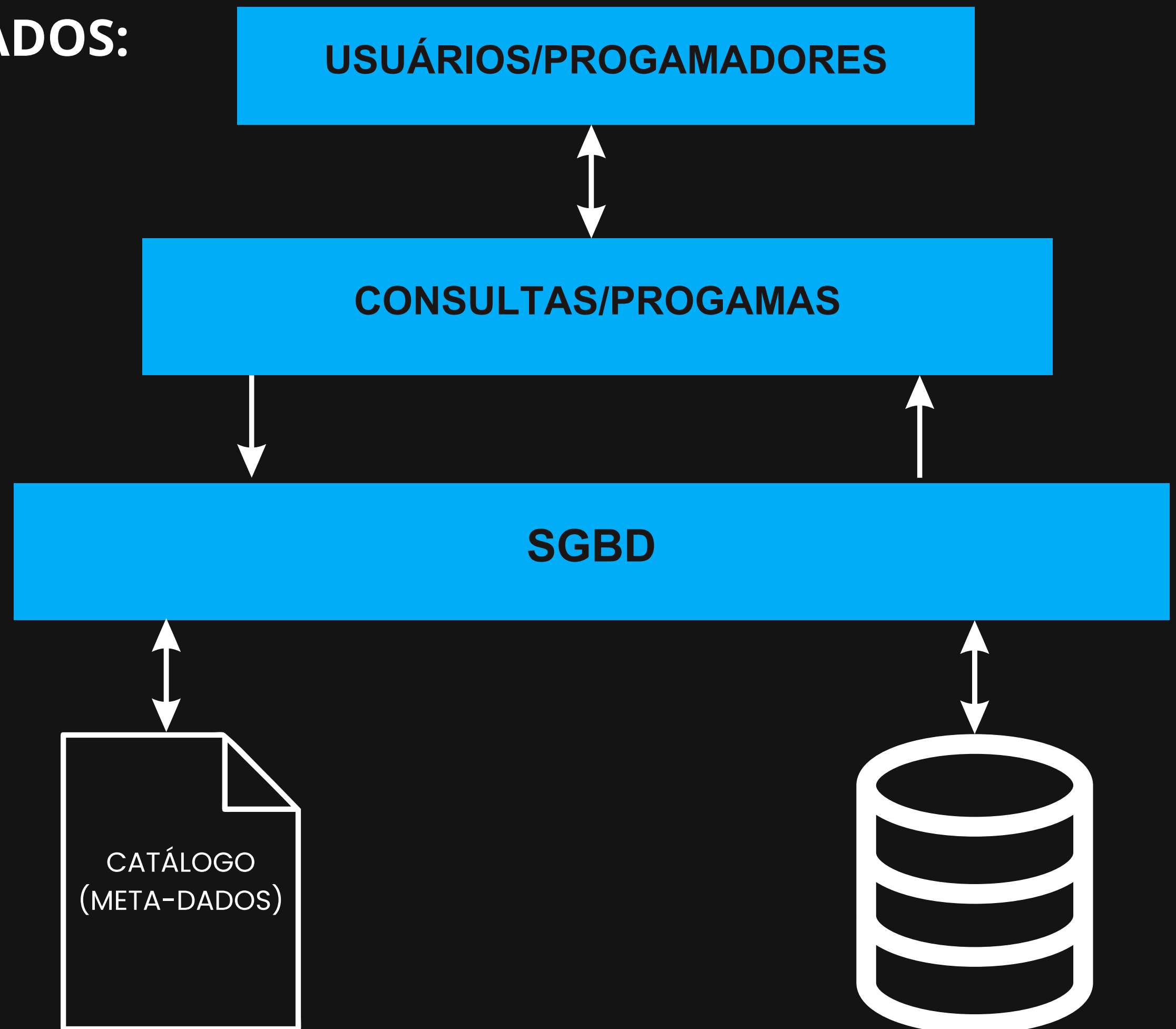
O QUE É UM BANCO DE DADOS?



Um **banco de dados** é uma coleção de dados com um significado implícito. Além disso, possui um propósito específico e deve estar organizado de tal maneira que facilite operações como inclusão, leitura, atualização e exclusão.



SISTEMA DE BANCO DE DADOS:



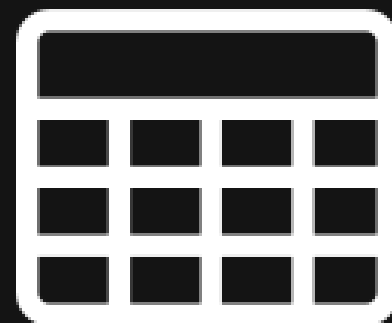
Algumas definições...

- **SISTEMA GERENCIADOR DE BANCO DE DADOS (SBGD)**: é uma coleção de programas que permite o usuário criar e manter um banco de dados;
- **CATÁLOGO OU METADADOS**: possibilita a manipulação de diversos bancos de dados através de um único programa;
- **MODELO DE DADOS**: um modelo de dados é uma abstração para descrever como os dados são estruturados, organizados, relacionados e manipulados em um sistema de banco de dados. Há três tipos:

Conceitual



Lógico



Físico



Fases de um projeto de banco de dados:



1 - Conceitual - Modelo Entidade Relacionamento (ER) - Peter Chen, 1976

TB_EMPREGADO

#cpf	nome	cargo	salario
------	------	-------	---------

TB_TELEFONE_EMPREGADO

#@cpf	#telefone
-------	-----------

TB_PROJETO

#codigo	nome	dt_ini	dt_fim	descricao
---------	------	--------	--------	-----------

TB_TRABALHA

#@cpf	#@codigo
-------	----------

2 - Lógico - Projeto lógico de banco de dados

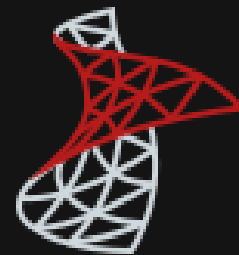
Agenda.sql

```
CREATE TABLE TB_PESSOA (  
    ID INT IDENTITY(1, 1) PRIMARY KEY,  
    NOME VARCHAR(50) NOT NULL,  
    EMAIL VARCHAR(100) NOT NULL UNIQUE,  
    TELEFONE VARCHAR(15)  
)  
  
CREATE TABLE TB_TAREFA (  
    ID INT IDENTITY(1, 1) PRIMARY KEY,  
    DESCRICAO VARCHAR(250) NOT NULL,  
    STATUS_TAREFA VARCHAR(20) NOT NULL,  
    ID_RESPONSAVEL INT NOT NULL,  
    FOREIGN KEY (ID_RESPONSAVEL) REFERENCES TB_PESSOA(ID)  
)  
  
CREATE TABLE TB_EVENTO (  
    ID INT IDENTITY(1, 1) PRIMARY KEY,  
    NOME_EVENTO VARCHAR(100) NOT NULL,  
    DATA_EVENTO DATE NOT NULL,  
    LOCAL_EVENTO VARCHAR(100) NOT NULL,  
    ID_ORGANIZADOR INT NOT NULL,  
    FOREIGN KEY (ID_ORGANIZADOR) REFERENCES TB_PESSOA(ID)  
)
```

3 - Físico - Definições no SGBD

Alguns bancos de dados...

- Relacional:



ORACLE

- NoSql:



- Orientado a objetos:



- Nuvem:



Outros: Objeto-Relacionais, rede...



A LINGUAGEM SQL

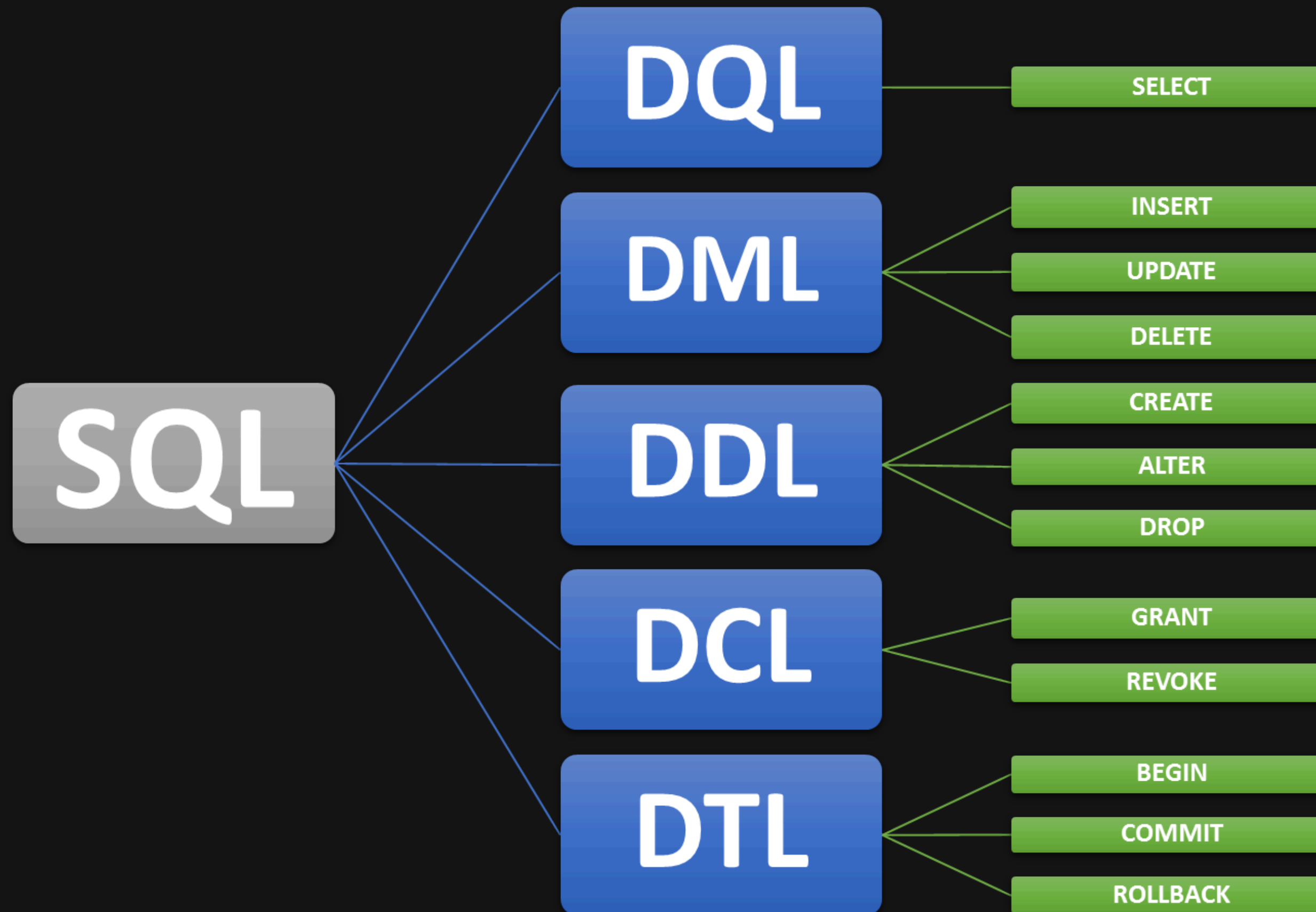


- O que é?

Structured Query Language é uma linguagem de consulta declarativa, baseada em operações sobre conjuntos, usada para manipular dados em Sistemas de Gerenciamento de Banco de Dados (SGBDs) relacionais.

- Quando surgiu?

Originalmente chamado SEQUEL, foi desenvolvido pela IBM em **1974** como parte do projeto SYSTEM R. Foi criada por Donald Chamberlin e Ray Boyce, com o objetivo de fornecer uma maneira estruturada de consulta em bancos de dados relacionais.



Fonte: <https://www.devmedia.com.br/guia/guia-completo-de-sql/38314#ddl>



MICROSOFT SQL SERVER



- **O que é?**

Sistema gerenciador de banco de dados relacional (RDBMS)

- **Responsável por:**

- **Garantir integridade dos dados;**
- **Manter a precisão e a coerência dos dados armazenados;**
- **Restaurar o banco de dados para um estado correto após qualquer problema no sistema.**



AGENDA+



Criando um banco de dados



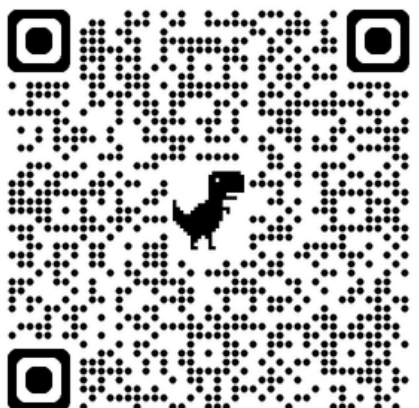
Criação

O processo de criação de um banco de dados é muito simples, para fazê-lo basta executar a instrução abaixo.

sintaxe

CREATE DATABASE nome_do_banco;

Referência



Uso

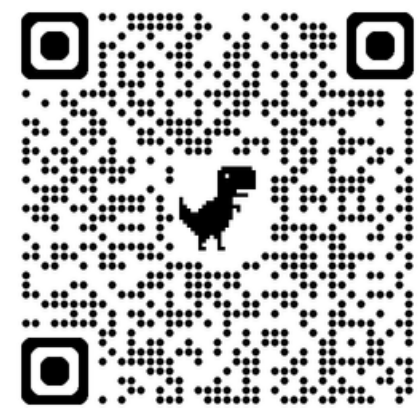
É possível criar vários bancos de dados e por isso é necessário definir o banco no qual queremos trabalhar.

Podemos definir o banco utilizando a instrução abaixo.

sintaxe

USE nome_do_banco;

Referência





Criando tabelas



Conceitos

Tabela é a estrutura que organiza e armazena os dados de um banco em um formato tabular, ou seja, os dados são dispostos seguindo um formato de colunas e linhas.

Para criar uma tabela é preciso definir o nome das colunas e seus respectivos tipos como segue:

sintaxe

```
CREATE TABLE nome_da_tabela (  
    nome_da_coluna    tipo_da_coluna  
);
```

Referência



Tipos de dados

- **INT**: para armazenar valores inteiros.
- **REAL**: para valores flutuantes armazenando de forma aproximada.
- **DECIMAL**: armazena valores com precisão e escala fixas.
- **CHAR**: guarda cadeia de caracteres com um tamanho fixo.
- **VARCHAR**: também guarda cadeia de caracteres, mas com tamanho variável.
- **DATE**: armazena datas no formato AAAA-MM-DD.
- **TIME**: armazena horários no formato hh:mm:ss.
- **DATETIME**: guarda uma data e hora nos formatos acima.

Referência



NULL e NOT NULL

Por padrão (definido implicitamente) as colunas de uma tabela podem conter o valor **NULL**, esse “valor” indica a ausência de um valor.

Esse comportamento é útil quando inicialmente uma coluna não precisa ter um valor ou quando ela é opcional.

Quando esse comportamento não é o desejado é possível especificar a coluna como **NOT NULL**, fazendo isso ela não pode mais ficar sem um valor.

Referência



Para criar uma coluna que explicitamente aceita o valor **NULL** faz-se:

sintaxe

```
CREATE TABLE nome_da_tabela (  
    nome_da_coluna    tipo_da_coluna  NULL  
);
```

Porém isso é opcional. Já para não permitir o valor **NULL** faz-se:

sintaxe

```
CREATE TABLE nome_da_tabela (  
    nome_da_coluna    tipo_da_coluna  NOT NULL  
);
```

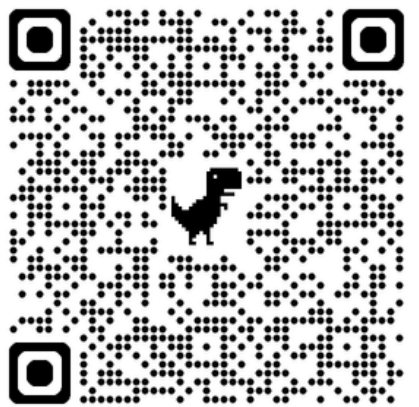
Valor Default

Quando necessário, é possível definir um **valor padrão** para uma coluna, isso é, no momento da adição de dados a uma tabela quando o valor da coluna não for especificado ela assumira o valor padrão.

sintaxe

```
CREATE TABLE nome_da_tabela (  
    nome_da_coluna    tipo_da_coluna DEFAULT valor  
);
```

Referência



Auto-incremento

Outra funcionalidade muito importante é a propriedade **IDENTITY** (auto-incremento em outros **SGBDs**).

Com essa propriedade podemos definir uma coluna que tem seu valor **incrementado automaticamente** em toda inserção de novos dados.

sintaxe

```
CREATE TABLE nome_da_tabela (  
    nome_da_coluna    tipo_da_coluna IDENTITY(s, p)  
);
```

s: valor inicial (opcional)

p: valor de incremento (opcional)

Referência



Remover $\xrightarrow{1 \rightarrow N}$ tabela $\xleftarrow{1 \leftarrow N}$ Alterar

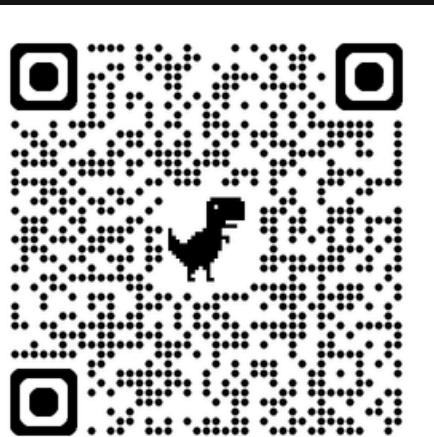
Quando uma tabela não é mais necessária é possível apagá-la e com ela todos os dados armazenados nela ou associados a ela com a instrução abaixo

sintaxe

DROP TABLE nome_da_tabela;

Essa instrução aceita mais de um nome de tabela de uma vez.

Referência



Em algumas ocasiões é preciso alterar uma tabela após a criação dela, isso pode ser feito da seguinte forma base:

sintaxe

ALTER TABLE nome_da_tabela
INSTRUÇÃO_DE_ALTERAÇÃO;

Referência



Continuação: alterar

A instrução de alteração de uma tabela pode ser a adição de uma coluna:

sintaxe

ALTER TABLE *nome_da_tabela*

ADD *nome_da_coluna tipo_da_coluna;*

Bem como a remoção de uma coluna:

sintaxe

ALTER TABLE *nome_da_tabela*

DROP COLUMN *nome_da_coluna;*

Em ambos os casos é possível especificar mais de uma coluna.

Também é possível alterar uma coluna específica da tabela através de:

sintaxe

ALTER TABLE *nome_da_tabela*

ALTER COLUMN *nome_da_coluna tipo_da_coluna;*

Referência





Manipulando dados



Inserção

A adição de dados é feita através da instrução **INSERT** podendo ser inseridos mais de uma linha de dados por vez.

sintaxe

INSERT INTO nome_da_tabela (lista_de_colunas)
VALUES (valores_das_colunas);

Listar as colunas para inserção é opcional, quando não incluído todas as colunas são afetadas pela inserção.

Referência



Modificação

Quando necessário modificar os dados usa-se a instrução **UPDATE** com a sintaxe abaixo:

sintaxe

UPDATE nome_da_tabela
SET nome_da_coluna = valor_da_coluna
WHERE condição;

Um detalhe importante é que nunca se executa um **UPDATE** sem **WHERE**!

Referência



Remoção

Para podermos remover linhas das tabelas usamos o comando **DELETE**.

sintaxe

DELETE FROM *nome_da_tabela*
WHERE *condição*;

Quando o **WHERE** e a condição não são definidos na instrução todas as linhas da tabela serão removidas.

Referência



Truncate

Uma maneira mais eficiente de remover todas as linhas da tabela é usando a instrução **TRUNCATE TABLE**.

sintaxe

TRUNCATE TABLE *nome_da_tabela*;

Referência



DELETE vs TRUNCATE vs DROP

Quando falamos de remover todos os dados de uma tabela pode surgir a dúvida entre qual instrução utilizar já que existem três possibilidades: **DELETE**, **TRUNCATE TABLE** E **DROP TABLE**.

DELETE é o primeiro método e o mais lento (principalmente em tabelas grandes) já que ele faz uma checagem em cada linha da tabela e permite a reversão de uma exclusão.

TRUNCATE TABLE por outro lado não faz a mesma checagem que o **DELETE** também não permite a reversão da execução do comando.

A ultima opção é o **DROP TABLE** (o mais rápido) que assim como o **TRUNCATE TABLE** não faz checagens linha a linha para fazer a remoção, mas além de remover os dados da tabela, essa instrução **APAGA** a estrutura da tabela e qualquer informação associada a ela.

Para cada cenário é preciso avaliar qual método tem o comportamento que mais se aproxima do desejado.

Referência





Comando Select



- Para que serve?

O comando **SELECT** serve para consultar e obter dados armazenados nas tabelas do banco de dados.

```
SELECT COLUNA1, COLUNA2, COLUNA3... FROM NOME_DA_TABELA
```

```
SELECT * FROM NOME_DA_TABELA
```

O * substitui todas as colunas da tabela

- **DISTINCT**

A cláusula **DISTINCT** é usada para eliminar duplicatas nas linhas retornadas por uma consulta.


```
SELECT DISTINCT MARCA FROM PRODUTO
```


- **ALIASES**

É possível atribuir apelidos (aliases) às colunas de uma tabela.

```
SELECT DT_FIM AS 'DATA FIM', (CUSTO_TOTAL - CUSTO_PREVISTO) LUCRO  
FROM PRODUTO
```

O **AS** pode ser omitido para nomes simples.



- **WHERE**

Com a cláusula **WHERE** é possível filtrar os dados de uma consulta, retornando apenas as linhas que atendem a uma condição específica.

```
SELECT NOME FROM ALUNO  
WHERE IDADE >= 18
```

• OPERADORES NA FILTRAGEM

AND

```
SELECT NOME FROM ALUNO  
WHERE IDADE >= 18 AND  
CURSO = 'BIOLOGIA'
```

OR

```
SELECT NOME FROM ALUNO  
WHERE CURSO = 'BIOLOGIA'  
OR CURSO = 'FÍSICA'
```

IN

```
SELECT NOME FROM ALUNO  
WHERE CURSO IN ('BIOLOGIA',  
'FÍSICA', 'LETRAS')
```

NOT IN

```
SELECT NOME FROM ALUNO  
WHERE CURSO NOT IN ('BIOLOGIA',  
'FÍSICA', 'LETRAS')
```

LIKE

```
SELECT NOME FROM PROFESSOR  
WHERE NOME LIKE '%BENICASA%'
```

```
SELECT NOME FROM PROFESSOR  
WHERE NOME LIKE '_LCIDE%'
```

BETWEEN

```
SELECT NOME FROM FUNCIONARIO  
WHERE SALARIO BETWEEN 1000  
AND 2000
```

```
SELECT NOME FROM FUNCIONARIO  
WHERE DT_ADMISSAO BETWEEN  
'2024-11-01' AND '2024-11-31'
```

- **Definição dos operadores:**

AND

Filtra registros que atendem a todas as condições especificadas.

OR

Filtra registros que atendem a pelo menos uma das condições especificadas.

IN

Filtra registros cujos valores de uma coluna estão em uma lista de valores especificada.

NOT IN

Filtra registros cujos valores de uma coluna não estão em uma lista de valores especificada.

LIKE

Filtra registros cujos valores de uma coluna correspondem a um padrão especificado, com suporte a curingas como % e _.

BETWEEN

Filtra registros dentro de um intervalo de valores, inclusivo para os limites especificados.

- **ORDER BY**

Com a cláusula **ORDER BY** é possível ordenar os resultados de uma consulta em ordem crescente ou decrescente com base em uma ou mais colunas.

```
SELECT NOME, CARGO FROM FUNCIONARIO  
ORDER BY SALARIO ASC
```

```
SELECT NOME, CARGO FROM FUNCIONARIO  
ORDER BY SALARIO DESC
```

```
SELECT NOME, CARGO FROM FUNCIONARIO  
ORDER BY SALARIO DESC, IDADE ASC
```

- **TOP**

Com a cláusula **TOP** é possível limitar o número de linhas retornados por uma consulta

```
SELECT TOP 5 NOME, CARGO FROM FUNCIONARIO
```

```
SELECT TOP 50 PERCENT NOME, CARGO FROM FUNCIONARIO
```


- **INSERT + SELECT**

É possível usar o resultado de uma consulta como valores para inserir dados em uma tabela com o comando INSERT.

```
INSERT INTO TB_LOG_VENDAS (CD_FUNC, CD_PRODUTO, QUANTIDADE, DATA)  
SELECT CD_FUNC, CD_PRODUTO, QUANTIDADE, DATA FROM TB_VENDAS
```

```
INSERT INTO TB_CLIENTE (CD_CLIENTE, NOME, EMAIL, TELEFONE)  
SELECT CD_CLIENTE, NOME, EMAIL, TELEFONE FROM TB_CLIENTE_ANTIGO  
WHERE STATUS = 'ATIVO'
```



RESTRIÇÕES DE INTEGRIDADE



Conceito

As **restrições de integridade** são regras aplicadas a um banco de dados relacional para garantir a consistência e a validade dos dados.

Elas ajudam a prevenir a inserção de dados inválidos ou inconsistentes e asseguram que as relações entre as tabelas sejam mantidas corretamente.

Tipos de Restrições

- **Primary Key**
- **Unique Key**
- **Foreign Key**
- **Check**
- **Default**

• Definição das Restrições e sua Sintaxe:

Primary key:

A **Primary Key** é uma restrição em uma tabela que identifica de forma única cada registro. Ela garante que não haja valores duplicados nem valores nulos na coluna (ou conjunto de colunas) que a define.

Sintaxe:

```
CREATE TABLE Alunos (  
  ID INT PRIMARY KEY, -- Chave primária na coluna ID  
  Nome VARCHAR(100),  
  DataNascimento DATE )
```

```
CREATE TABLE Matriculas (  
  ID_Aluno INT,  
  ID_Turma INT,  
  DataMatricula DATE,  
  PRIMARY KEY (ID_Aluno, ID_Turma) -- Chave primária composta  
)
```

```
ALTER TABLE Alunos  
  ADD CONSTRAINT PK_Alunos PRIMARY KEY (ID)
```

• Definição das Restrições e sua Sintaxe:

Unique Key (Chave Única)

A **Unique Key** é uma restrição que assegura que os valores em uma ou mais colunas de uma tabela sejam únicos, ou seja, não podem se repetir. Diferentemente da **Primary Key**, uma tabela pode ter várias **Unique Keys**, e as colunas podem conter valores nulos (mas os valores nulos também não podem se repetir).

Sintaxe:

```
CREATE TABLE Funcionarios (  
  ID INT PRIMARY KEY, -- Chave primária  
  CPF VARCHAR(11) UNIQUE, -- CPF deve ser único  
  Email VARCHAR(100) UNIQUE -- Email deve ser único  
)
```

```
CREATE TABLE Reservas (  
  ID INT PRIMARY KEY,  
  DataReserva DATE,  
  HoraReserva TIME,  
  Sala VARCHAR(50),  
  UNIQUE (DataReserva, HoraReserva, Sala) -- Combinação única de data, hora e sala );
```

```
ALTER TABLE Funcionarios  
  ADD CONSTRAINT UQ_Funcionarios_CPF UNIQUE (CPF);
```

• Definição das Restrições e sua Sintaxe:

Foreign Key (Chave Estrangeira)

A **Foreign Key** é uma restrição usada para estabelecer um vínculo entre duas tabelas em um banco de dados relacional. Ela assegura que os valores em uma ou mais colunas de uma tabela correspondam a valores existentes na **Primary Key (ou Unique Key)** de outra tabela, garantindo a integridade referencial

Sintaxe:

```
CREATE TABLE Clientes (  
  ID INT PRIMARY KEY,  
  Nome VARCHAR(100) );
```

```
CREATE TABLE Pedidos ( ID INT PRIMARY KEY,  
  DataPedido DATE,  
  ID_Cliente INT,
```

```
  FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID) -- Chave estrangeira ligando à tabela Clientes );
```

```
ALTER TABLE Pedidos  
  ADD CONSTRAINT FK_Pedidos_Clientes FOREIGN KEY (ID_Cliente) REFERENCES Clientes(ID);
```

- **Removendo Restrições**

PRIMARY KEY

```
ALTER TABLE Alunos  
DROP CONSTRAINT PK_Alunos;
```

UNIQUE KEY

```
ALTER TABLE Funcionarios  
DROP CONSTRAINT UQ_Funcionarios_CPF;
```

FOREIGN KEY

```
ALTER TABLE Pedidos  
DROP CONSTRAINT FK_Pedidos_Clientes;
```

• Definição das Restrições e sua Sintaxe:

CHECK (Restrição de Validação)

A restrição **CHECK** é usada para impor condições que os valores de uma coluna devem atender. É uma forma de validar os dados no momento da inserção ou atualização, garantindo que obedecem a regras específicas definidas pelo desenvolvedor.

Sintaxe:

```
CREATE TABLE Produtos (  
ID INT PRIMARY KEY,  
Nome VARCHAR(100),  
Preco DECIMAL(10, 2), CHECK (Preco > 0), -- O preço deve  
ser maior que zero  
Estoque INT, CHECK (Estoque >= 0) -- O estoque não  
pode ser negativo );
```

```
CREATE TABLE Funcionarios (  
ID INT PRIMARY KEY,  
Nome VARCHAR(100),  
Salario DECIMAL(10, 2),  
Bonus DECIMAL(10, 2), CHECK (Salario + Bonus <= 10000) -- A soma de salário e bônus  
não pode ultrapassar 10.000 );
```

```
ALTER TABLE Funcionarios  
ADD CONSTRAINT CK_Salario CHECK (Salario > 0);
```


• Definição das Restrições e sua Sintaxe:

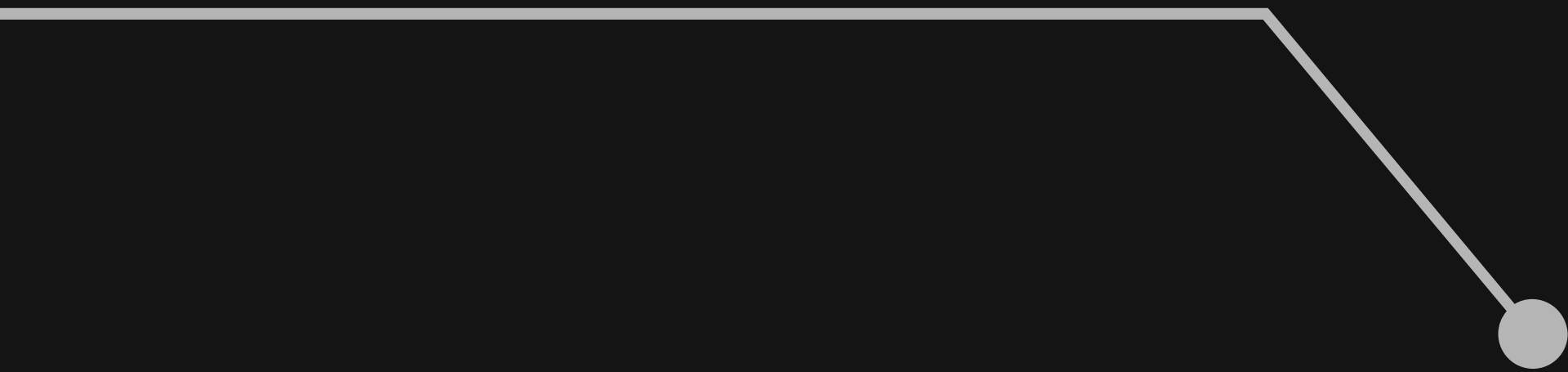
DEFAULT (Valor Padrão)

A restrição **DEFAULT** é usada para definir um valor padrão que será atribuído automaticamente a uma coluna caso nenhum valor seja especificado durante a inserção (INSERT). Ela é útil para garantir consistência nos dados e reduzir a necessidade de fornecer valores manualmente para campos comuns.

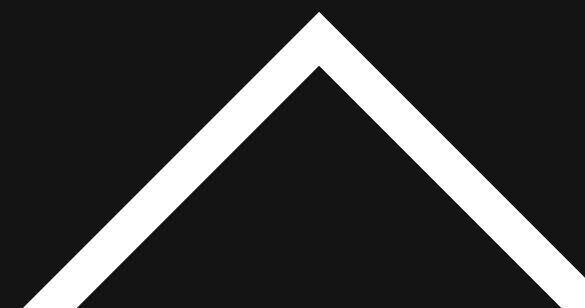
Sintaxe:

```
CREATE TABLE Funcionarios (  
ID INT PRIMARY KEY,  
Nome VARCHAR(100),  
Cargo VARCHAR(50) DEFAULT 'Analista', -- Valor padrão para Cargo  
DataContratacao DATE DEFAULT GETDATE() );
```

```
ALTER TABLE Funcionarios  
ADD CONSTRAINT DF_Cargo DEFAULT 'Analista' FOR Cargo;
```



Comando Join



Conceito

Join em SQL é uma operação que permite combinar dados de duas ou mais tabelas em um banco de dados com base em uma condição de relacionamento entre elas. A junção permite que você consulte dados distribuídos em várias tabelas de maneira eficiente, retornando resultados que conectam informações de cada tabela relacionada.



Tipos de Join

- **INNER JOIN:** Retorna apenas as linhas que têm correspondência em ambas as tabelas.
- **LEFT JOIN:** Retorna todas as linhas da tabela à esquerda e as correspondentes da tabela à direita. Se não houver correspondência, os resultados da tabela à direita serão nulos.
- **RIGHT JOIN:** Retorna todas as linhas da tabela à direita e as correspondentes da tabela à esquerda. Se não houver correspondência, os resultados da tabela à esquerda serão nulos.
- **FULL JOIN:** Retorna todas as linhas quando há uma correspondência em uma das tabelas.

Exemplo



O mesmo exemplo, mas agora em sql

sintaxe

```
CREATE TABLE TB_JOGADORES (  
    ID_JOGADOR INT IDENTITY(1,1) PRIMARY KEY,  
    NOME VARCHAR(100) NOT NULL,  
    POSICAO VARCHAR(50) NOT NULL,  
    APELIDO VARCHAR(50) NULL  
);  
  
CREATE TABLE TB_EQUIPES (  
    ID_EQUIPE INT PRIMARY KEY,  
    NOME_EQUIPE VARCHAR(100) NOT NULL  
);  
  
CREATE TABLE TB_JOGADORES_EQUIPE (  
    ID_JOGADOR INT,  
    ID_EQUIPE INT,  
    PRIMARY KEY (ID_JOGADOR, ID_EQUIPE),  
    FOREIGN KEY (ID_JOGADOR) REFERENCES TB_JOGADORES(ID_JOGADOR),  
    FOREIGN KEY (ID_EQUIPE) REFERENCES TB_EQUIPES(ID_EQUIPE)  
);
```

O insert do mesmo exemplo

sintaxe

```
INSERT INTO JOGADORES (NOME,APELIDO,POSICAO)  
VALUES
```

```
('IGOR ', 'INDIANO','ATACANTE'),  
( 'GUILHERME', NULL,'MEIO-CAMPO'),  
( 'JONATHA', 'J0ng4b','ZAGUEIRO'),  
( 'MARCOS','CEBOLA','GOLEIRO'),  
( 'ANDRÉ', 'MISTER' ,'TÉCNICO'),  
( 'AÉLIO',NULL, 'AUXILIAR'),  
( 'RAPHAEL',NULL, 'PRESIDENTE');
```

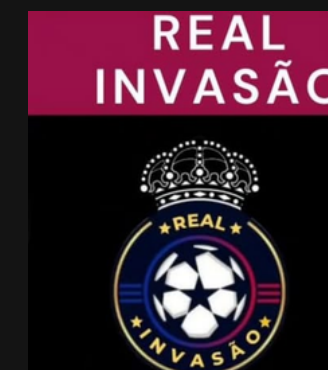
```
INSERT INTO EQUIPES (ID_EQUIPE, NOME_EQUIPE)  
VALUES
```

```
(1, 'FUMAÇA FC'),  
(2, 'REAL INVASÃO FC'),  
(3, 'DENDENZEIRO FC'),  
(4,'ATLÉTICO DA RUA DA LAMA'),  
(5,'GRÊMIO DA TERRA DURA');
```

```
INSERT INTO JOGADORES_EQUIPE (ID_JOGADOR,  
ID_EQUIPE)
```

```
VALUES
```

```
(1, 4),  
(2, 2),  
(3, 3),  
(4, 4),  
(5, 2),  
(7, 1);
```



Tá mas e como usa o join no exemplo agora?

Comece fazendo **Select** de alguma das três tabelas, vou escolher **TB_JOGADORES**, após isso use o comando **JOIN** e una com a tabela que possui relação com ela, nesse caso **TB_JOGADORES_EQUIPES**, após isso junte elas onde a chave primaria for igual a chave estrangeira e repita o processo para juntar com **TB_EQUIPES**.

SELECT

FROM

TB_JOGADORES tj

JOIN

TB_JOGADORES_EQUIPE tje ON tj.ID_JOGADOR = tje.ID_JOGADOR

JOIN

TB_EQUIPES te ON tje.ID_EQUIPE = te.ID_EQUIPE;

E como usa Left join no mesmo exemplo?

O **LEFT JOIN** segue o mesmo princípio do **join**, mas ele tem a novidade de exibir todas as linhas da tabela à **esquerda** e as correspondentes da tabela à direita. Se não houver correspondência, os resultados da tabela à direita serão **nulos**. Nesse **SELECT** irei exibir **TODOS** os Jogadores, ou seja jogadores sem clube também serão vistos.

SELECT

tj.NOME,
te.NOME_EQUIPE

FROM

TB_JOGADORES tj

LEFT JOIN

TB_JOGADORES_EQUIPE tje **ON** tj.ID_JOGADOR = tje.ID_JOGADOR

LEFT JOIN

TB_EQUIPES te **ON** tje.ID_EQUIPE = te.ID_EQUIPE;

E como usa Right join no exemplo?

O **Right join** segue o mesmo princípio do **left join**, mas ele tem a novidade de exibir **todas** as linhas da tabela à **direita** e as correspondentes da tabela à **esquerda**. Se não houver correspondência, os resultados da tabela à esquerda serão **nulos**. Nesse **SELECT** irei exibir **TODAS** as **EQUIPES**, ou seja clubes sem jogadores também serão vistos.

SELECT

tj.NOME,
te.NOME_EQUIPE

FROM

TB_JOGADORES tj

RIGHT JOIN

TB_JOGADORES_EQUIPE tje **ON** tj.ID_JOGADOR = tje.ID_JOGADOR

RIGHT JOIN

TB_EQUIPES te **ON** tje.ID_EQUIPE = te.ID_EQUIPE;

E como usa Full join no exemplo???

O **full join** segue o mesmo princípio do **left** e **right JOIN**, mas ele tem a novidade de exibir **todas** as linhas quando há uma correspondência em **uma** das tabelas. Nesse **SELECT** irei exibir **TODAS** as **EQUIPES** e **TODOS** os **Jogadores**, ou seja clubes sem jogadores também serão vistos e jogadores sem clube também serão vistos.

SELECT

tj.NOME,
te.NOME_EQUIPE

FROM

TB_JOGADORES tj

FULL JOIN

TB_JOGADORES_EQUIPE tje **ON** tj.ID_JOGADOR = tje.ID_JOGADOR

FULL JOIN

TB_EQUIPES te **ON** tje.ID_EQUIPE = te.ID_EQUIPE;

Agora é com vocês!

Faça o exercício sobre JOIN



Repositório:

github.com/M4RCOSVS0/Curso-Introducao-SQL-XSEMAC



Funções de Agregação



Conceito

As funções de agregação são usadas para realizar cálculos em um conjunto de dados e retornar um único valor. As funções mais comuns são:

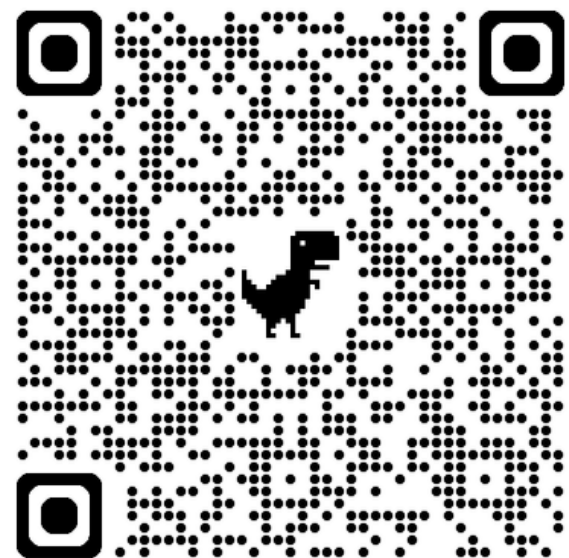
COUNT(): Conta o número de linhas.

SUM(): Soma os valores de uma coluna.

AVG(): Calcula a média de uma coluna.

MIN(): Retorna o valor mínimo.

MAX(): Retorna o valor máximo.



Exemplo Prático:

Tabela Vendas:

id_venda	produto	valor
1	mouse	100
2	teclado	200
3	headset	150
4	mouse	80
5	mouse	95
6	teclado	60

Exemplo usando a função SUM():

```
SELECT SUM(valor) AS total_vendas  
FROM Vendas;
```

Resultado:

total_vendas

450

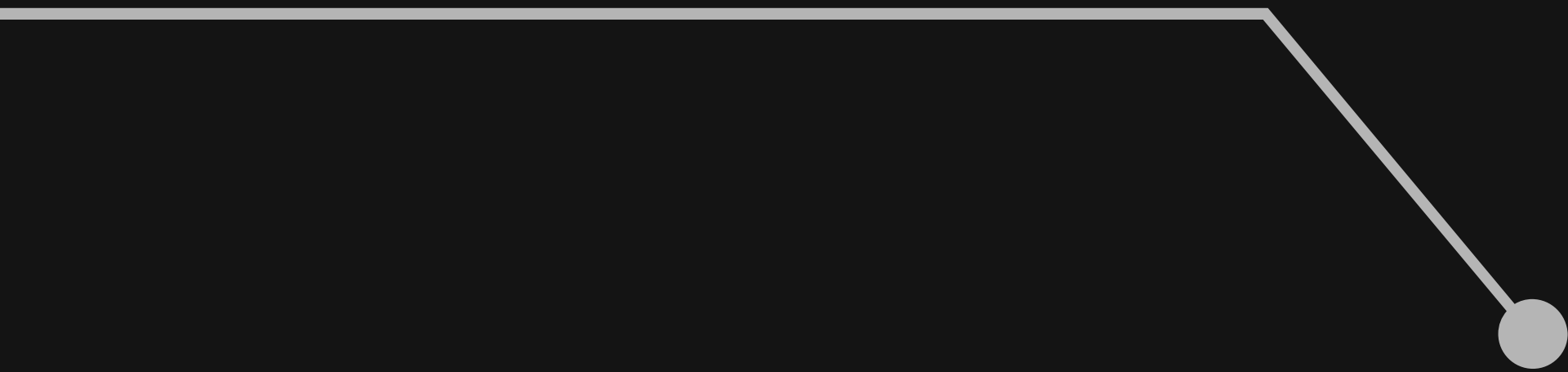
Blz craque, mas temos um problema...

Caso você tente usar alguma função de agregação mas quiser não só exibir uma coluna e querer por exemplo exibir além do valor. Já que agora temos produtos com mais de uma venda, como vou exibir a quantidade de vendas por produto?

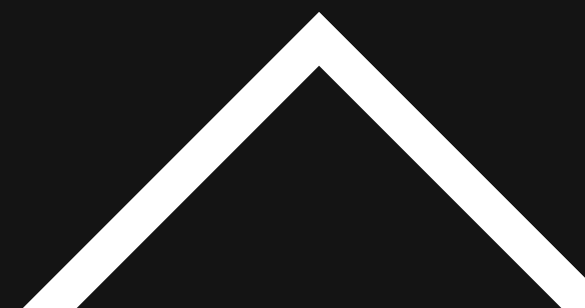
id_venda	produto	valor
1	mouse	100
2	teclado	200
3	headset	150
4	mouse	80
5	mouse	95
6	teclado	60

Se fazer como antes e só adicionar uma linha vai dar erro...

```
SELECT produto, SUM(valor) AS total_vendas  
FROM Vendas;
```



Group By



O que é o GROUP BY?

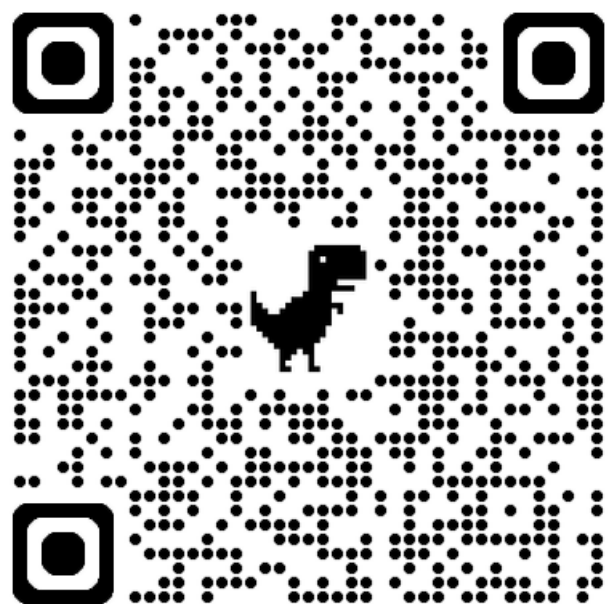
O GROUP BY é utilizado para agrupar linhas que possuem valores comuns em uma ou mais colunas.

Portanto, o comando serve para uma organização e agrupamento de dados de maneira eficiente.

Geralmente combinado com funções agregadas como SUM, COUNT, AVG, MAX, e MIN.

resolvendo nosso problema com Group By

```
SELECT
    produto,
    SUM(valor) AS total_vendas
FROM
    Vendas
Group By
    produto;
```



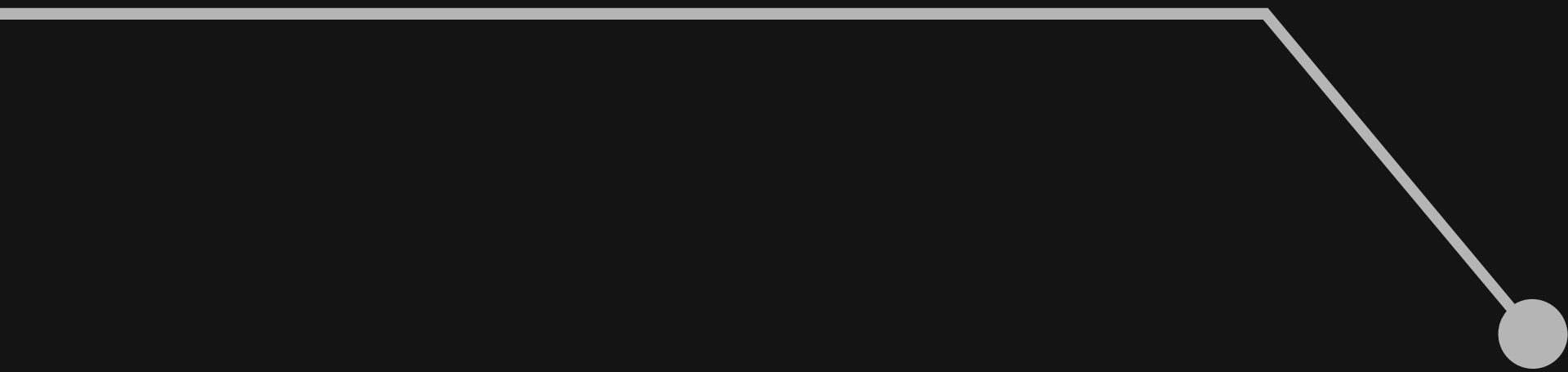
Blz craque, mas temos outro problema...

Imagene agr que você deve exibir no select apenas os produtos que tenham mais de uma venda?

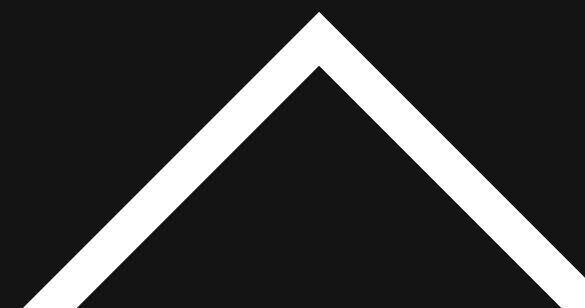
id_venda	produto	valor
1	mouse	100
2	teclado	200
3	headset	150
4	mouse	80
5	mouse	95
6	teclado	60

Se fazer como antes e só adicionar o group by e usar where como limitação não funciona

```
SELECT produto, count(produto) AS quantidade_vendas
FROM Vendas
Where count(produto) > 1
Group By produto;
```



Having



O que é Having?

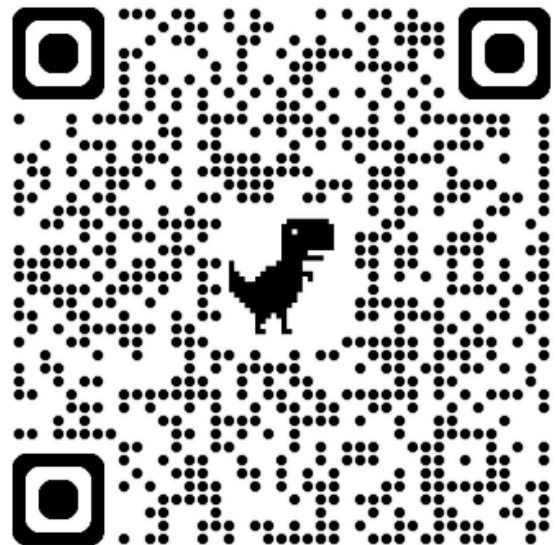
A cláusula **HAVING** é usada para filtrar os grupos gerados pelo comando **GROUP BY**, de forma semelhante ao que a cláusula **WHERE** faz para linhas individuais. A principal diferença entre **WHERE** e **HAVING** é que:

WHERE filtra dados antes de eles serem agrupados.

HAVING filtra os grupos depois de eles serem criados.

resolvendo nosso problema com Having

```
SELECT
    produto,
    count(produto) AS quantidade_vendas
FROM
    vendas
Group By
    produto
HAVING
    COUNT(produto) > 1
```



Agora é com vocês!

Faça o exercício sobre FUNÇÕES DE AGREGAÇÃO



Repositório:

github.com/M4RCOSVS0/Curso-Introducao-SQL-XSEMAC



T-SQL



O que é T-SQL?

O T-SQL é uma extensão do SQL padrão desenvolvida pela Microsoft e usada no Microsoft SQL Server e no Azure SQL Database.

Ele adiciona funcionalidades avançadas ao SQL, como controle de fluxo, tratamento de erros, e manipulação procedural, permitindo maior flexibilidade e poder no gerenciamento de bancos de dados.

Características T-SQL

Controle de Fluxo:

Oferece comandos como IF...ELSE, WHILE, e CASE para lógica condicional e repetição.

Manipulação de Variáveis:

Permite declarar e usar variáveis para armazenar valores temporários.

Procedimentos Armazenados e Funções:

Suporte a criação de rotinas reutilizáveis para encapsular lógica de negócios no banco.

Manipulação de Cursores:

Permite trabalhar com conjuntos de resultados de forma iterativa.

Tratamento de Erros:

Suporte a comandos como TRY...CATCH para gerenciar erros em scripts e procedimentos.

• T-SQL

Declaração de Variáveis no T-SQL

No **T-SQL**, as **variáveis** são usadas para armazenar valores temporários durante a execução de scripts ou procedimentos. Elas são úteis para cálculos, condições e armazenamento de resultados intermediários.

Sintaxe:

```
DECLARE @Nome VARCHAR(50);  
DECLARE @Idade INT;
```

```
SET @Nome = 'João';  
SET @Idade = 30;
```

```
DECLARE @CategoriaID INT = 3;  
  
SELECT *FROM Produtos  
WHERE CategoriaID = @CategoriaID;
```

if...else

O **IF...ELSE** no **T-SQL** é usado para implementar lógica condicional em scripts ou procedimentos armazenados. Ele permite executar diferentes blocos de código dependendo se uma condição é verdadeira ou falsa.

IF (CONDIÇÃO)

BEGIN-- Bloco de código executado se a condição for verdadeira

END

ELSE

BEGIN-- Bloco de código executado se a condição for falsa

END;

Exemplo prático if...else

```
DECLARE @Pontuacao INT = 85;
```

```
IF (@Pontuacao >= 90)
```

```
BEGIN
```

```
    PRINT 'Nota: A';
```

```
END
```

```
ELSE IF (@Pontuacao >= 80)
```

```
BEGIN
```

```
    PRINT 'Nota: B';
```

```
END
```

```
ELSE IF (@Pontuacao >= 70)
```

```
BEGIN
```

```
    PRINT 'Nota: C';
```

```
END
```

```
ELSE
```

```
BEGIN
```

```
    PRINT 'Nota: F';
```

```
END;
```


Laços(ou loops)

No T-SQL, os laços (ou loops) são usados para executar repetidamente um bloco de código enquanto uma condição é verdadeira. Eles são úteis para processar conjuntos de dados ou executar ações iterativas em um banco de dados.

WHILE (CONDIÇÃO)

BEGIN-- Bloco de código a ser executado enquanto a condição for verdadeira

END;

Exemplo prático WHILE com Break e continue

DECLARE @Numero INT = 0;

WHILE (@Numero < 10)

BEGIN

SET @Numero = @Numero + 1;

IF (@Numero = 5)

CONTINUE; -- Pula o resto do código e vai para a próxima iteração

PRINT 'Número: ' + CAST(@Numero AS VARCHAR);

IF (@Numero = 8)

BREAK; -- Sai do laço quando o número for 8

END;

Cursor

Os Cursors no T-SQL são usados para percorrer linha por linha os resultados de uma consulta, permitindo a manipulação de dados de forma sequencial. Eles são úteis em situações onde operações linha a linha são necessárias, mas devem ser usados com cuidado devido ao impacto no desempenho.

```
DECLARE nome_cursor CURSOR FOR  
    SELECT coluna1, coluna2 FROM tabela;
```

```
OPEN nome_cursor;
```

```
FETCH NEXT FROM nome_cursor INTO @variavel1,  
@variavel2;
```

```
WHILE @@FETCH_STATUS = 0  
BEGIN  
    -- Operações a serem realizadas com os dados  
    FETCH NEXT FROM nome_cursor INTO @variavel1,  
@variavel2;  
END;
```

```
CLOSE nome_cursor;  
DEALLOCATE nome_cursor;
```

Exemplo prático do uso do cursor para percorrer linhas

```
DECLARE @ID INT, @Nome VARCHAR(50);
```

```
DECLARE cursor_funcionarios CURSOR FOR  
    SELECT ID, Nome FROM Funcionarios;
```

```
OPEN cursor_funcionarios;
```

```
FETCH NEXT FROM cursor_funcionarios INTO @ID, @Nome;
```

```
WHILE @@FETCH_STATUS = 0  
BEGIN  
    PRINT 'Funcionário: ' + CAST(@ID AS VARCHAR) + ' - ' +  
@Nome;
```

```
    FETCH NEXT FROM cursor_funcionarios INTO @ID,  
@Nome;  
END;
```

```
CLOSE cursor_funcionarios;  
DEALLOCATE cursor_funcionarios;
```

FICOU CLARO, PESSOAL?

ALGUMA DÚVIDA?

OBRIGADO A TODOS

PERFEITO

TÁ OK?