

alura



Escola  
DATA SCIENCE



01000101 0110011011000011101111 010100 01000001010101000  
01100100 01100101 01000000 01000000011100010101000  
0101110111001100001010

# GLOSSÁRIO SQL

REALIZANDO CONSULTAS COM SQL:  
JOINS, VIEWS E TRANSAÇÕES

# BOAS-VINDAS AO GLOSSÁRIO DE SQL!

Este é um guia prático que simplifica os comandos da linguagem de consulta estruturada. Este glossário foi criado para proporcionar um auxílio aos comandos mais avançados do SQL, como **UNION**, **Subconsultas**, **Joins**, **Views** entre outros. Explore e aprofunde-se no mundo dos bancos de dados, utilizando este guia para fortalecer sua compreensão e aplicação do SQL. Aproveite o material e, em caso de dúvidas, sintá-se à vontade para enviar perguntas através do fórum do curso.

**Um abraço e bons estudos!**

# SUMÁRIO

01000001 01100011 01000001 01000001 01000001 01000001 01000001 01000001  
00100000 01000001 01000001 01000001 01000001 01000001 01000001 01000001  
01000001 01000001 01000001 01000001 01000001 01000001 01000001 01000001

❁ CRIANDO TABELAS.....	04	❁ JOINS .....	28
❁ CRIANDO TABELAS COM DEFAULT .....	07	❁ INNER JOIN.....	30
❁ DELETE CASCADE.....	10	❁ RIGHT JOIN.....	33
❁ UTILIZANDO UNION .....	13	❁ LEFT JOIN .....	36
❁ UTILIZANDO UNION ALL .....	16	❁ FULL JOIN .....	39
❁ SUBCONSULTAS.....	19	❁ VIEW .....	41
❁ SUBCONSULTAS NO IN.....	22	❁ TRIGGER .....	45
❁ SUBCONSULTAS COM HAVING .....	25	❁ TRANSAÇÕES.....	49

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100011  
00001010

// Glossário SQL\_

# CRIANDO TABELAS



A faint, light gray wireframe of a human face is visible in the background, centered on the right side of the slide.

Para criarmos tabelas, utilizamos o comando **CREATE TABLE**.

Além disso, devemos especificar as informações dessa tabela como o nome da tabela, nome das colunas, tipo de dado de cada coluna, as colunas que serão chave primária ou estrangeira, etc.



Por exemplo, a criação da tabela de **produtos**, contendo cinco colunas, ficaria assim:

```
CREATE TABLE Produtos (  
  ID TEXT,  
  Nome VARCHAR(255) NOT NULL,  
  Descricao TEXT,  
  Preço DECIMAL(10, 2) NOT NULL,  
  Categoria VARCHAR(50),  
  PRIMARY KEY (ID) );
```



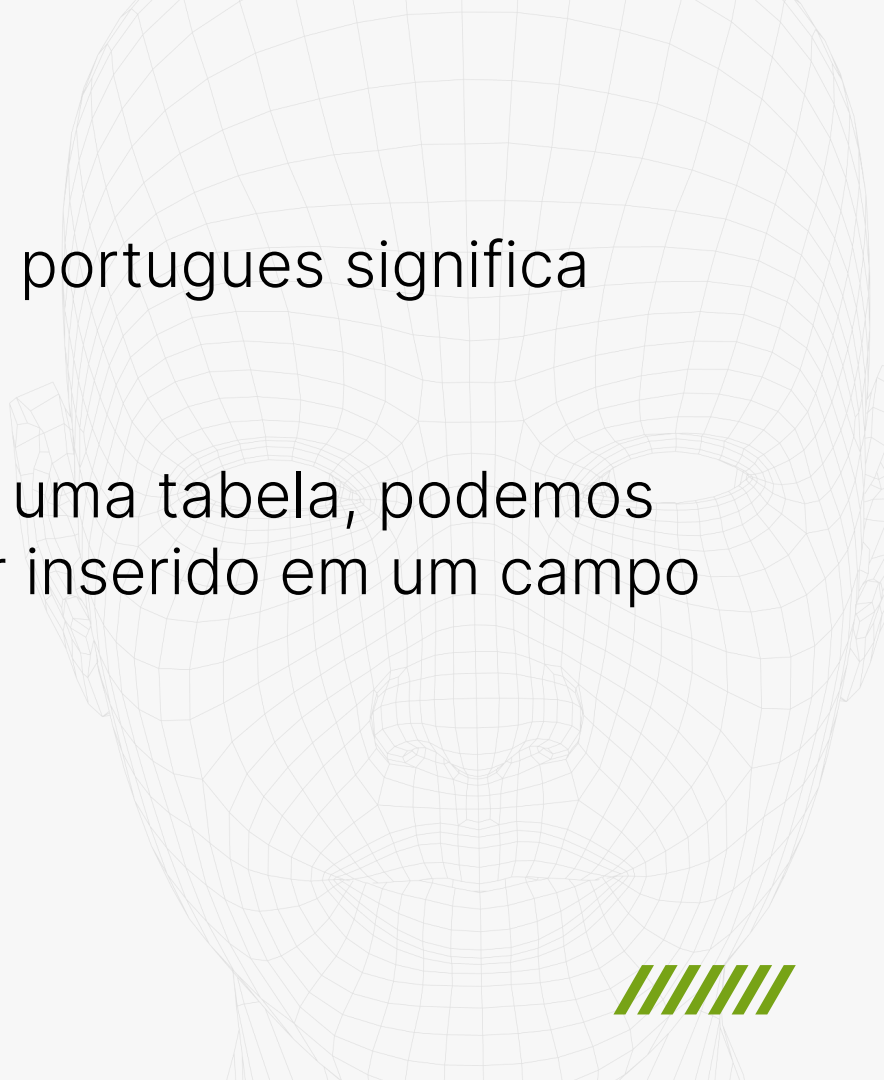
01000101 0110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 0110011  
00001010

// Glossário SQL\_

# CRIANDO TABELAS COM DEFAULT





A faint, light gray wireframe of a human face is visible in the background on the right side of the slide.

**DEFAULT**, traduzindo para o português significa “padrão”.

Ao utilizar **DEFAULT** ao criar uma tabela, podemos definir um valor padrão a ser inserido em um campo específico.





Por exemplo, se quisermos criar a tabela de **clientes** definindo um valor padrão para o campo de email, ficaria dessa forma:

```
CREATE TABLE clientes (  
  id TEXT PRIMARY KEY,  
  nome VARCHAR(255) NOT NULL,  
  telefone VARCHAR(20) NOT NULL,  
  email VARCHAR(100) DEFAULT "Sem email",  
  endereco TEXT NOT NULL );
```



01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# DELETE CASCADE



Quando utilizamos o **DELETE CASCADE** ao criar uma tabela, estamos indicando que sempre que um registro na tabela **pai** é excluído, todos os registros relacionados na tabela **filho (referenciada)** também são automaticamente excluídos

01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010



Por exemplo, se quisermos criar a tabela de **pedidos** utilizando o **DELETE CASCADE**, ficaria dessa forma:

```
CREATE TABLE pedidos (  
    id TEXT PRIMARY KEY,  
    idCliente INTEGER,  
    dataHoraPedido DATETIME,  
    Status VARCHAR(50),  
    FOREIGN KEY (idCliente) REFERENCES Clientes(id) ON  
    DELETE CASCADE  
);
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# UTILIZANDO UNION



O operador **UNION** seleciona apenas valores distintos entre as tabelas. Para isso, ele combina os resultados das queries e, em seguida, executa um **SELECT DISTINCT** para eliminar os valores duplicados.

```
01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 0110011 00001010
```



```
SELECT Rua, Bairro, Cidade, Estado, CEP  
FROM Fornecedores f  
UNION  
SELECT Rua, Bairro, Cidade, Estado, CEP  
FROM Colaboradores c;
```

Por exemplo, se quisermos retornar de forma distinta o **endereço completo** de todos os colaboradores e fornecedores em uma única consulta.





01000101 0110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 0110011  
00001010

// Glossário SQL\_

# UTILIZANDO UNION ALL



O operador **UNION ALL** tem a mesma função do **UNION**, ou seja, ele combina os resultados de duas ou mais queries, a diferença é que ele mantém os valores duplicados de cada **SELECT**.

```
01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 0110011 00001010
```



```
SELECT Rua, Bairro, Cidade, Estado, CEP  
FROM Fornecedores f  
UNION ALL  
SELECT Rua, Bairro, Cidade, Estado, CEP  
FROM Colaboradores c;
```

Por exemplo, se quisermos retornar o **nome** e **endereço completo** de todos os colaboradores e fornecedores em uma única consulta, mesmo as informações que são repetidas.



01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# SUBCONSULTAS



**Subconsultas** são **consultas aninhadas** dentro de outras consultas, que podemos utilizar para retornar informações de uma ou mais **tabelas**.

01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010



Por exemplo, vamos retornar o **nome** de um cliente que fez um pedido em uma **data específica**.

```
SELECT Nome
FROM clientes
WHERE ID = (
    SELECT ID_Cliente
    FROM pedidos
    WHERE DataHoraPedido='2023-01-02 08:15:00'
);
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# SUBCONSULTAS NO IN





A cláusula **IN** é usada em **SQL** para verificar se um valor corresponde a qualquer valor em uma lista específica de valores.

01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010



Por exemplo, vamos retornar os **nomes** dos clientes que fizeram pedidos no mês de **janeiro**

```
SELECT Nome
FROM clientes
WHERE ID IN (
    SELECT ID_Cliente
    FROM pedidos
    WHERE strftime('%m', DataHoraPedido) = '01'
);
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# SUBCONSULTAS COM HAVING



A cláusula **HAVING** é usado para filtrar dados **depois** que eles foram agrupados com a cláusula **GROUP BY**.

01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 0110011 00001010



Por exemplo, vamos retornar o **nome** e o **preço** dos produtos cujo **preço** é **maior que** o preço médio de todos os **produtos**

```
SELECT Nome, Preço
FROM produtos
GROUP BY Nome, Preço
HAVING Preço > (
    SELECT AVG(Preço)
    FROM produtos
);
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100011  
00001010

// Glossário SQL\_

# JOINS



A cláusula **JOIN** correspondente a uma operação de junção em álgebra relacional - combina colunas de uma ou mais tabelas em um banco de dados relacional.

01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 0110011 00001010





01000101 0110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 0110011  
00001010

// Glossário SQL\_

# INNER JOIN



O **INNER JOIN** combina linhas de **duas tabelas** quando há uma correspondência entre as colunas especificadas.

```
01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010
```



Por exemplo, se quisermos retornar informações sobre os **pedidos** e os **clientes** associados a esses pedidos

```
SELECT p.ID, c.Nome  
FROM pedidos p  
INNER JOIN clientes c  
ON p.IDCliente = c.ID;
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# RIGHT JOIN



O **RIGHT JOIN** retorna todas as linhas da tabela da **direita** e as correspondentes da **esquerda**.

```
01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010
```



Por exemplo, se quisermos retornar todos os registros da tabela de **produtos** que estão em algum registro da tabela de **itensPedidos**.

```
SELECT p.Nome  
FROM ItensPedido ip  
RIGHT JOIN Produtos p  
ON p.ID = ip.IDProduto;
```

01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# LEFT JOIN





O **LEFT JOIN** retorna todas as linhas da tabela da **esquerda** e as linhas correspondentes da tabela da **direita**.

```
01000101 01110011 01100011 01101111 01101100 01100001  
00100000 01100100 01100101 00100000 01000100 01100001  
01100100 01101111 01110011 00001010
```



Por exemplo, se quisermos retornar todos os registros da tabela de **clientes** que estão em algum registro da tabela de **pedidos**.

```
SELECT C.Nome  
FROM Clientes c  
LEFT JOIN Pedidos p  
ON c.ID = p.IDCliente
```



01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# FULL JOIN



O **FULL JOIN** combina as linhas de ambas as tabelas presentes na consulta. Desse jeito, se quisermos retornar **todos** os **clientes** e todos os **pedidos** existentes.

```
SELECT DISTINCT c.Nome, p.IDCliente  
FROM Clientes c  
FULL JOIN Pedidos p  
ON c.ID = p.IDCliente
```

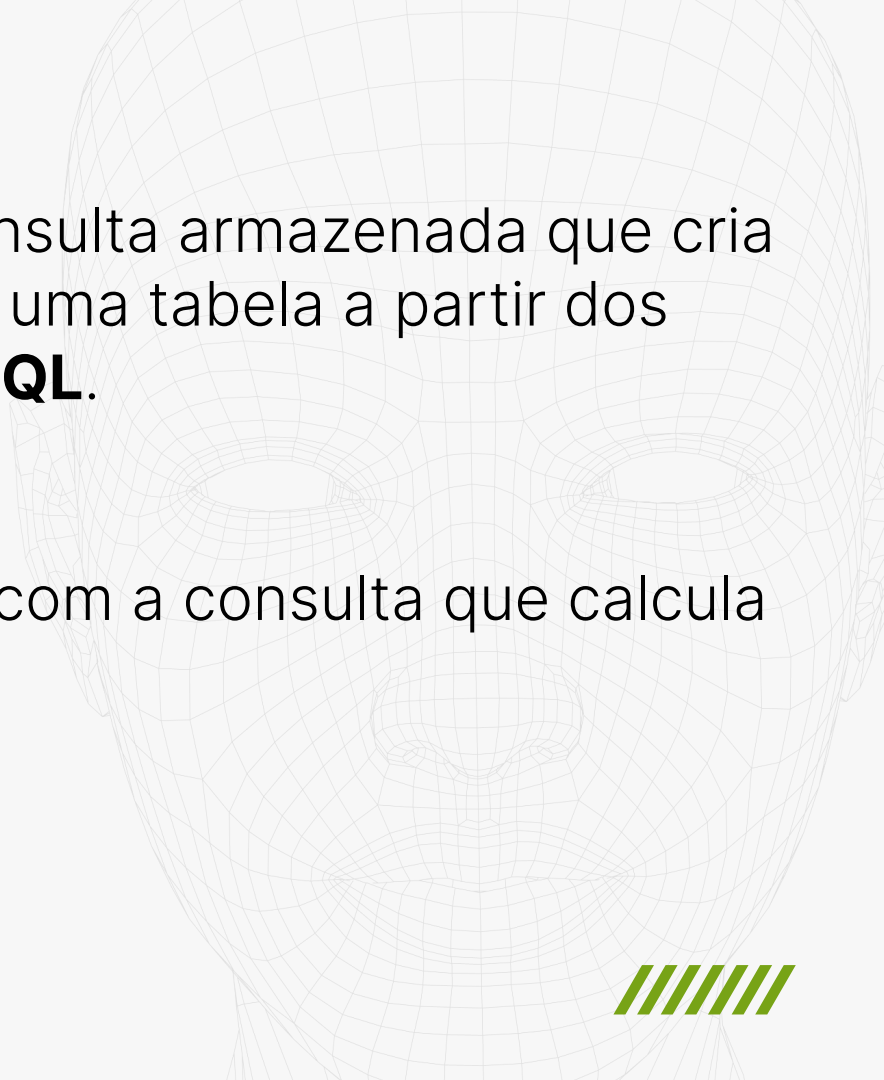


01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100011  
00001010

// Glossário SQL\_

# VIEWS



A faint, light gray wireframe of a human face is visible in the background, centered on the right side of the slide. It consists of a grid of lines forming the facial structure.

Uma **VIEW** em **SQL** é uma consulta armazenada que cria uma representação virtual de uma tabela a partir dos resultados de uma consulta **SQL**.

Por exemplo, criar uma **View** com a consulta que calcula o **total de cada pedido**.



```
CREATE VIEW ViewTotalPorPedido AS  
SELECT  
    P.ID AS ID_Pedido,  
    P.DataHoraPedido,  
    C.Nome AS NomeCliente,  
    SUM(IP.Quantidade * IP.PrecoUnitario) AS  
TotalPorPedido  
FROM Pedidos AS P  
JOIN Clientes AS C ON P.ID_Cliente = C.ID  
JOIN ItensPedido AS IP ON P.ID = IP.ID_Pedido  
GROUP BY P.ID, P.DataHoraPedido, C.Nome;
```

Após criar a **View** podemos utilizá-la normalmente como uma **tabela**.

```
SELECT *  
FROM ViewTotalPorPedido;
```



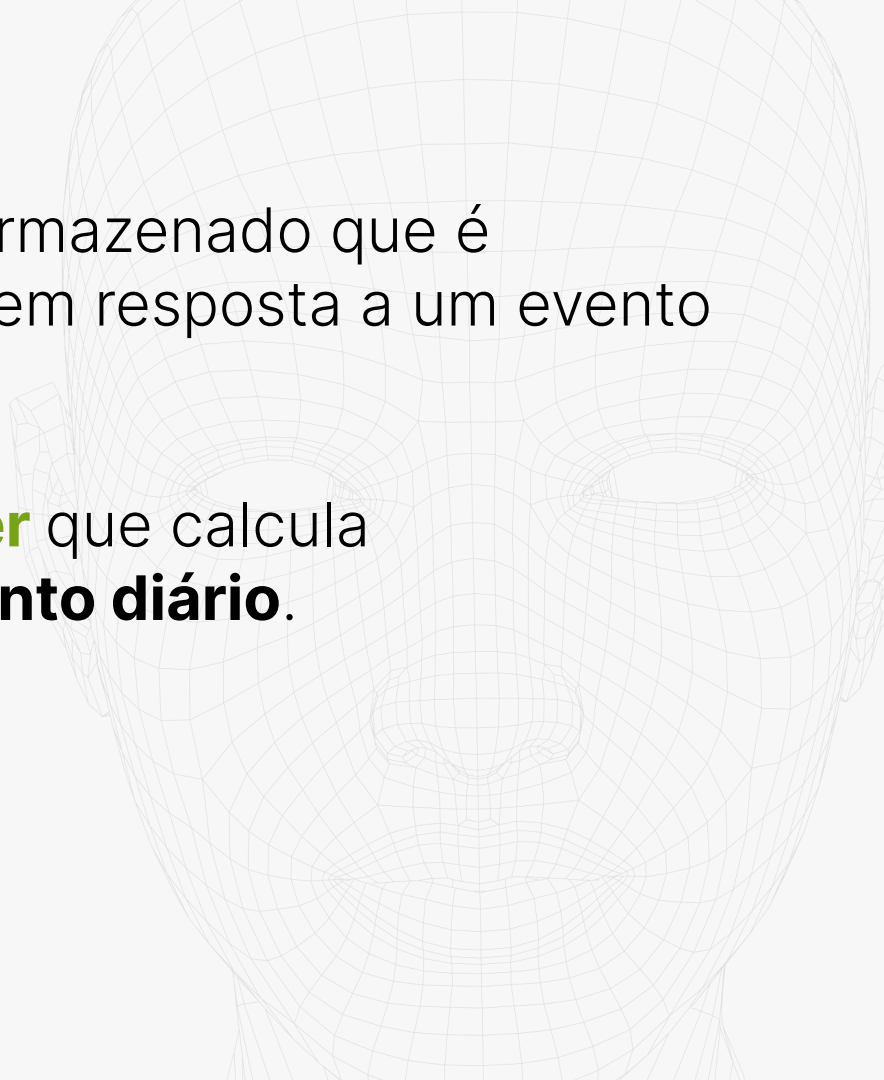


01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100011  
00001010

// Glossário SQL\_

# TRIGGER



A faint, stylized wireframe illustration of a human face, composed of a grid of lines, serves as a background for the right side of the slide.

**Trigger** é um procedimento armazenado que é executado automaticamente em resposta a um evento específico em uma tabela.

Por exemplo, criar uma **trigger** que calcula automaticamente o **faturamento diário**.

```
CREATE TRIGGER CalculoFaturamentoDiario
AFTER INSERT ON ItensPedido
FOR EACH ROW
BEGIN
    DELETE FROM FaturamentoDiario ;
    INSERT INTO FaturamentoDiario (Dia, FaturamentoTotal)
    SELECT
        DATE(P.DataHoraPedido) AS Dia,
        SUM(IP.Quantidade * IP.PrecoUnitario) AS Faturamento
    FROM Pedidos AS P
    JOIN ItensPedido AS IP ON P.ID = IP.ID_Pedido
    GROUP BY Dia
    ORDER BY Dia;
END;
```

Ao inserir novos registros na tabela de **itenspedidos**, a trigger será **acionada**:

```
INSERT INTO Pedidos(ID, IDCliente, DataHoraPedido,  
Status)  
VALUES(451, 27, '2023-10-07 14:30:00', 'Em Andamento' );
```

```
INSERT INTO ItensPedidos  
(IDPedido, IDProduto, Quantidade, PrecoUnitario)  
VALUES(451, 14, 1, 6.0),  
        (451, 13, 1, 7.0);
```

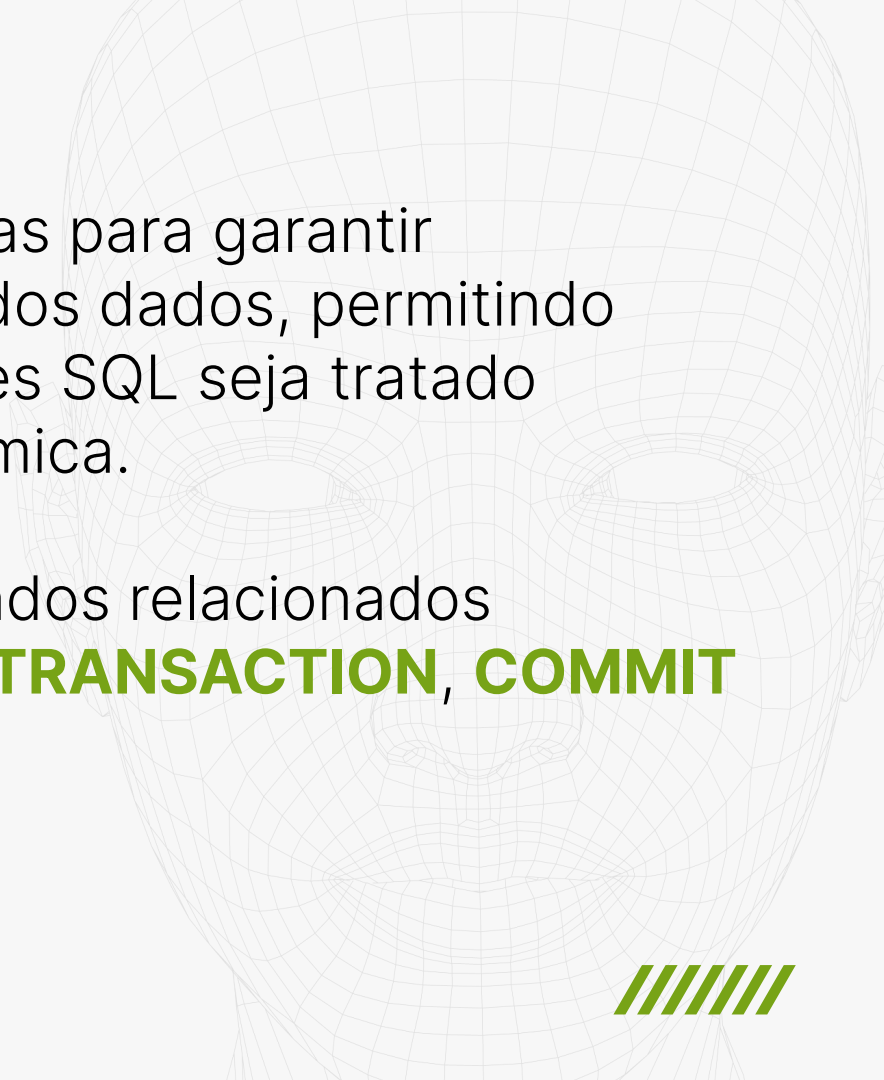


01000101 01110011 01100011 01101111 01101100  
01100001 00100000 01100100 01100101 00100000  
01000100 01100001 01100100 01101111 01100111  
00001010

// Glossário SQL\_

# TRANSAÇÕES



A faint, stylized wireframe of a human face is visible in the background, composed of a grid of lines forming the facial structure.

Transações no **SQL** são usadas para garantir a consistência e integridade dos dados, permitindo que um conjunto de operações SQL seja tratado como uma única unidade atômica.

Existem três principais comandos relacionados a transações no **SQL**: **BEGIN TRANSACTION**, **COMMIT** e **ROLLBACK**.



Por exemplo, se quisermos iniciar uma **transação** utilizamos o **BEGIN** ou o **BEGIN TRANSACTION**.

**BEGIN TRANSACTION;**



Por exemplo, se quisermos reverter as alterações realizadas nos dados, utilizamos o comando **ROLLBACK**.

**ROLLBACK;**





Por exemplo, se quisermos confirmar as alterações realizadas nos dados, utilizamos o comando **COMMIT**.

**COMMIT;**



## UTILIZE E DOMINE O SQL!

Parabéns por explorar o Glossário de SQL! Agora que você adquiriu os comandos mais avançados da linguagem, é hora de aplicar esse conhecimento. Utilize este material como referência em seus projetos e desafios, praticando para aprimorar suas habilidades na manipulação de bancos de dados. Ao se tornar mais confiante na linguagem SQL, você estará preparado para enfrentar novos desafios.

Muito obrigado por chegar até aqui e nos vemos nos próximos cursos da formação em SQL da Alura. Até mais!

# AVALIE O CURSO E DEIXE UM COMENTÁRIO.

Compartilhe um resumo de seus novos conhecimentos em suas redes sociais.

alura



Escola Data Science

