SQL - Resumo

Este é um **resumo completo das minhas anotações** feitas ao longo do curso gratuito de **SQL**, promovido pelo Téo Me Why.

O objetivo deste arquivo é organizar de forma clara e didática os conceitos, práticas e exemplos apresentados durante o curso, servindo como um **guia de estudo pessoal** e referência rápida. Para facilitar a compreensão, dividi o conteúdo em tópicos, incluí explicações passo a passo e destaquei boas práticas e observações importantes.

Todos os exemplos presentes neste resumo são baseados na **análise do database "Téo Me Why - TeoMeWhy Loyalty System"**, disponível no **Kaggle**, que foi utilizado durante todo o curso.

Todo o conteúdo apresentado neste resumo é baseado no **material original do curso** e nos exemplos fornecidos pelo instrutor. Os créditos pelo conteúdo completo vão para o **Téo Me Wjy**, que disponibilizou o curso gratuitamente e de forma acessível.

Este resumo é **uma compilação pessoal**, reorganizando e sintetizando o que aprendi, mas **não altera o conteúdo original do curso**.

1. Introdução: O que é um banco de dados?

Um banco de dados é formado por tabelas. Para entender melhor, pense em uma planilha do Excel: ela possui linhas e colunas.

- Cada **linha** corresponde a um **registro** (ou entidade).
- Cada **coluna** representa um **campo** (ou característica da entidade).

Por exemplo, em uma lista de compras:

- Itens como *tomate, batata e refrigerante* são registros (linhas).
- A quantidade de cada produto é um campo (coluna).

De forma semelhante, uma tabela pode armazenar informações de clientes (ID, nome, e-mail, data de cadastro) ou de vendas (código, cliente, vendedor, data, valor, nota fiscal).

As tabelas, portanto, representam **entidades do mundo real**. Cada coluna descreve uma característica dessa entidade, como altura, peso, gênero ou cor dos olhos no caso de uma pessoa.

Um banco de dados não representa apenas uma única coisa, mas sim um **conjunto de tabelas relacionadas**, que juntas formam a estrutura de um sistema. Por isso, podemos defini-lo como um **repositório organizado de informações**.

O nome técnico desse conjunto é **Sistema de Gerenciamento de Banco de Dados (SGBD)**. Existem diversos SGBDs, também chamados de *motores* ou *engines*, como: SQLite, MySQL, PostgreSQL, MariaDB, SQL Server, Oracle e Spark. A linguagem SQL é utilizada para interagir com todos eles.

2. O que é SQL?

SQL (do inglês **Structured Query Language**, ou **Linguagem de Consulta Estruturada**) é a linguagem usada para **pesquisar e manipular informações** dentro de sistemas de gerenciamento de banco de dados, geralmente **relacionais**.

Apesar de escrevermos códigos em SQL, **não se trata de uma linguagem de programação**, como Python, Java ou Go. Em vez de programar, **consultamos e manipulamos dados**.

2.1. Elementos principais do SQL

Dentro do SQL, existem alguns elementos essenciais:

- **Comandos:** indicam a ação que queremos realizar. Ex.: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `GRANT`.
- **Cláusulas:** complementam os comandos. Ex.: `FROM`, `WHERE`.

- **Expressões:** transformam ou calculam dados. Ex.: somas, divisões, comparações.
- **Predicados:** realizam comparações lógicas, como verificar se algo é maior, igual ou semelhante a outro valor.

Quando combinamos **comando + cláusula + expressão + predicado**, formamos uma **query** (consulta).

Exemplo básico:

```
SELECT coluna FROM tabela;
```

2.2. Tipos de dados

Para organizar informações corretamente, cada coluna precisa ter um **tipo de dado**:

- **Texto (string):** ex.: nome do produto.
- **Número:** ex.: quantidade do produto.
- **Data:** ex.: data de compra.
- **NULL:** representa dado ausente ou desconhecido. *Importante: NULL não é zero.*
- **Booleano:** verdadeiro ou falso.

2.3. Chaves e relacionamentos

- **Chave primária (Primary Key):** identifica unicamente cada registro, garantindo **integridade e consistência**.

  Ex.: o CPF de uma pessoa.

- **Chave estrangeira (Foreign Key):** cria **relacionamentos entre tabelas**, evitando repetição de dados.

  Ex.: em uma tabela de vendas, basta registrar o `ID` do cliente em vez de repetir todas as informações do cliente.

2.4. Restrições (Constraints)

As restrições definem regras que os dados devem seguir:

- `UNIQUE`: evita duplicidade (ex.: CPF).
- `CHECK`: garante condições específicas (ex.: número positivo).
- `NOT NULL`: obriga preenchimento de um campo (ex.: nome, e-mail).

Essas regras podem ser definidas na **criação da tabela** ou adicionadas posteriormente.

- **ENUMS:** restringem os valores possíveis de um campo, garantindo consistência.

  Ex.: campo "estado" só pode receber `SP`, `RJ`, `MG`, etc.

Em resumo, **SQL é uma linguagem de consulta** que combina comandos, cláusulas, expressões e predicados para **consultar, manipular e organizar dados** dentro de um banco de dados.

3. SELECT: primeiros passos

O comando **SELECT** é usado para **selecionar colunas de uma tabela** em um banco de dados.

3.1. Selecionando todas as colunas

Quando usamos:

```
SELECT * FROM tabela;
```

o `*` significa **"todas as colunas"**. Ou seja, o banco de dados retorna **todos os campos** da tabela, sem precisar listá-los individualmente.

3.2. Selecionando colunas específicas

Se quisermos retornar **apenas algumas colunas**, basta especificá-las:

```
SELECT IdCliente, DtCriacao, DtAtualizacao
FROM clientes;
```

Dessa forma, a consulta retorna apenas essas **três colunas**, tornando o resultado mais objetivo e fácil de analisar.

4. FROM: selecionando tabelas

O comando **FROM** é usado para **indicar de qual tabela os dados serão consultados** em uma query SQL.

4.1. Verificando as tabelas do banco

Para saber quantas tabelas existem no banco de dados que estamos usando:

- **SQLite:**

```
.tables
```

- MySQL/PostgreSQL:

```
SHOW TABLES;
```

Com isso, podemos identificar todas as tabelas existentes, por exemplo: `clientes`, `produtos`, `transacao_produto`, `transacoes`.

4.2. Visualizando uma tabela inteira

Para ver **todas as colunas e linhas** de uma tabela, usamos:

```
SELECT * FROM produtos;
```

Isso retorna **todas as informações da tabela**, permitindo analisar seu conteúdo completo.

4.3. Selecionando colunas específicas

Se quisermos apenas algumas colunas, podemos especificá-las:

```
SELECT IdProduto, DescProduto
FROM produtos;
```

Dessa forma, a consulta retorna apenas **as colunas desejadas**, tornando o resultado mais objetivo.

4.4. Selecionando linhas com LIMIT

Para limitar o número de linhas retornadas, usamos o comando **LIMIT**, que **sempre deve vir ao final da query**:

```
SELECT IdProduto, DescProduto
FROM produtos
LIMIT 2;
```

Isso permite visualizar apenas **uma parte da tabela**, sem precisar carregar todos os registros.

5. WHERE: filtrando dados

Para **filtrar dados no SQL**, usamos a cláusula **WHERE**.

5.1. Selecionando a tabela

Primeiro, precisamos indicar **qual tabela vamos consultar**:

```
SELECT * FROM produtos;
```

Isso retorna todas as colunas da tabela `produtos`.

5.2. Aplicando o filtro com WHERE

Depois do **FROM**, adicionamos a cláusula **WHERE** para **selecionar apenas os registros que atendem a uma condição específica**.

Exemplo:

```
SELECT * FROM produtos
WHERE CategoriaProduto = 'rpg';
```

Essa query retorna **todos os produtos cuja categoria é "rpg"**.

5.3. Observações importantes

- As **aspas simples** em `'rpg'` são usadas para indicar **valores de texto**.
- Em alguns bancos de dados, como MySQL ou PostgreSQL, também pode ser aceito o uso de **aspas duplas**.
- A comparação de texto é **case sensitive**, ou seja, letras maiúsculas e minúsculas são tratadas como valores diferentes.
    - Ex.: `'RPG'` e `'rpg'` seriam considerados diferentes.

6. Criando novas colunas

Até agora, utilizamos **SELECT** apenas para **consultar dados**, sem alterar ou criar colunas. Agora, vamos aprender como **adicionar colunas novas** a partir de valores existentes.

6.1. Selecionando a tabela

Primeiro, indicamos a tabela que queremos consultar, por exemplo:

```
SELECT *
FROM clientes;
```

Isso retorna todas as colunas da tabela `clientes`.

6.2. Criando uma nova coluna a partir de dados existentes

Suponha que queremos **adicionar 10 pontos a cada cliente**, criando uma nova coluna chamada `QtdePontos`.

Podemos fazer isso **adicionando a operação diretamente no SELECT**, sem alterar a tabela original:

```
SELECT *, QtdePontos + 10
FROM clientes;
```

- O retorna todas as colunas existentes.
- `QtdePontos + 10` cria **uma nova coluna calculada temporária**, mostrando o valor original da coluna `QtdePontos` somado a 10.

> Importante: essa operação não altera os dados da tabela original, apenas exibe o resultado da coluna calculada na consulta.

7. ORDER BY: ordenando os dados

Até agora, selecionamos linhas e aplicamos filtros, mas **não fizemos nenhuma ordenação**. Para ordenar registros, usamos a cláusula **ORDER BY**.

7.1. Ordenação básica

Para ordenar os clientes pela **quantidade de pontos**, usamos:

```
SELECT *
FROM clientes
ORDER BY QtdePontos;
```

- Por padrão, a ordenação é **crescente (ASC)**, do menor para o maior.

Para ordenar de forma **decrescente**, usamos `DESC`:

```
SELECT *
FROM clientes
ORDER BY QtdePontos DESC;
```

7.2. Limitando o número de resultados

Se quisermos, por exemplo, apenas os **top 10 clientes com mais pontos**, combinamos **ORDER BY** com **LIMIT**:

```
SELECT *
FROM clientes
ORDER BY QtdePontos DESC
LIMIT 10;
```

7.3. Ordenando por data ou múltiplos critérios

Para pegar os clientes mais antigos cadastrados:

```
SELECT *
FROM clientes
ORDER BY DtCriacao;
```

- A ordenação é **ascendente**, do registro mais antigo para o mais recente.

Também é possível ordenar por **mais de um critério**:

```
SELECT *
FROM clientes
ORDER BY DtCriacao ASC, QtdePontos DESC;
```

- Primeiro, os clientes são ordenados pela data de criação (ascendente).
- Dentro da mesma data, são ordenados pela quantidade de pontos (decrescente).

7.4. Combinando WHERE e ORDER BY

Podemos combinar filtros com ordenação. Por exemplo, selecionar apenas clientes que **têm conta na Twitch** e ordenar pelos critérios anteriores:

```
SELECT *
FROM clientes
WHERE fl_twitch = 1
ORDER BY DtCriacao ASC, QtdePontos DESC;
```

> Interpretação: selecione todos os clientes com Twitch ativa, ordenando do mais antigo para o mais recente e, dentro da mesma data, do maior para o menor número de pontos.

Observações importantes

- Dentro de um **SELECT**, só é possível ter **uma cláusula de cada tipo**:
    - Apenas **um WHERE**

    - Apenas **um ORDER BY**
- É possível, no entanto, adicionar **vários critérios dentro dessas cláusulas**.

8. CASE WHEN: criando condições

O **CASE WHEN** no SQL funciona de forma similar ao **if** das linguagens de programação. Ou seja, **dada uma condição, atribuímos um valor a ela**.

8.1. Selecionando os dados

Como exemplo, vamos usar a tabela `clientes`, selecionando apenas a **quantidade de pontos** e o **ID do cliente**:

```
SELECT IdCliente, QtdePontos
FROM clientes
ORDER BY QtdePontos;
```

8.2. Definindo intervalos

Queremos criar **categorias de clientes** com base na quantidade de pontos:

- 0 a 500 → Poney
- 501 a 1000 → Poney Premium
- 1001 a 5000 → Mago Aprendiz
- 5001 a 10000 → Mago Mestre
- Mais de 10000 → Mago Supremo

8.3. Criando a coluna condicional com CASE

```
SELECT
   IdCliente,
   QtdePontos,
   CASE
      WHEN QtdePontos <= 500 THEN 'Poney'
      WHEN QtdePontos <= 1000 THEN 'Poney Premium'
      WHEN QtdePontos <= 5000 THEN 'Mago Aprendiz'
      WHEN QtdePontos <= 10000 THEN 'Mago Mestre'
      ELSE 'Mago Supremo'
   END AS nome_grupo
FROM clientes;
```

Como o CASE funciona:

- **CASE** abre a condição e **END** fecha.
- Cada **WHEN** define uma condição, seguida do **THEN**, que atribui o valor quando a condição é satisfeita.
- A cláusula **ELSE** é opcional, mas útil para valores que não se enquadram em nenhum WHEN.
- A **ordem dos WHEN** é importante: o SQL testa as condições sequencialmente e **para no primeiro WHEN verdadeiro**.
    - Ex.: se `QtdePontos = 468`, o resultado será `'Poney'` e as condições seguintes não serão testadas.
- Cada CASE gera **uma nova coluna**, que pode ser nomeada com **AS**.

8.4. Resumo

- Começa com `CASE` e termina com `END`.
- `AS nome_coluna` permite dar um nome à coluna resultante.
- A ordem dos `WHEN` define a prioridade das condições.
- `ELSE` é opcional, mas cobre casos não previstos.
- Cada CASE cria **uma coluna adicional na query**.

9. COUNT: função de agregação

No SQL, **agregação** é o processo de **resumir ou sintetizar dados**, assim como fazemos em estatística. A ideia é **calcular estatísticas a partir de um conjunto de registros**, obtendo valores como contagem, média, soma, mínimo ou máximo.

Até agora, aplicamos **filtros** e **ordenamos dados**, mas não **resumimos as informações**. Agora, aprenderemos a **calcular estatísticas sobre os dados** de uma tabela.

9.1. Contando registros com COUNT

A função **COUNT** serve para **contar registros** em uma tabela.

Exemplo: contar todos os registros da tabela `clientes`:

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

- `COUNT(*)` retorna o **número total de linhas** da tabela, considerando todas as colunas.
- Ele ignora linhas completamente nulas.

9.2. Contando uma coluna específica

Também é possível contar apenas uma coluna específica, como `IdCliente`:

```
SELECT COUNT(IdCliente)
FROM clientes;
```

Nesse caso, o SQL conta apenas os registros **onde a coluna `IdCliente` não é nula.**

10. DISTINCT: selecionando valores únicos

A cláusula **DISTINCT** no SQL é usada para **retornar apenas valores únicos** em uma coluna ou combinação de colunas.

10.1. Contando clientes distintos

Suponha que queremos saber **quantos clientes diferentes existem** na tabela `cliente`:

```
SELECT COUNT(DISTINCT IdCliente)
FROM cliente;
```

- Aqui, estamos contando apenas valores **únicos** da coluna `IdCliente`.
- Neste caso específico, não há diferença, porque `IdCliente` é uma **chave primária**, ou seja, **não se repete na tabela**.

10.2. Contando clientes distintos em uma condição

Agora, vamos usar a tabela `transacoes` para descobrir **quantos clientes realizaram transações em julho de 2025**:

```
SELECT COUNT(DISTINCT IdCliente)
FROM transacoes
WHERE DtCriacao >= '2025-07-01'
  AND DtCriacao < '2025-08-01';
```

- `DISTINCT` garante que cada cliente seja contado **uma única vez**, mesmo que tenha feito várias transações.
- O **WHERE** filtra os registros apenas para o período desejado.

11. Funções de agregação: SUM, AVG, MIN, MAX

Além do **COUNT**, o SQL oferece outras funções de agregação para **resumir e analisar dados**:

- **SUM** → soma dos valores de uma coluna.
- **AVG** → média dos valores de uma coluna.
- **MIN** → menor valor da coluna.
- **MAX** → maior valor da coluna.

11.1. Somando valores com SUM

Para calcular **quantos pontos foram ganhos no mês de julho**, usamos:

```
SELECT SUM(QtdePontos)
FROM transacoes
WHERE DtCriacao >= '2025-07-01'
  AND DtCriacao < '2025-08-01'
  AND QtdePontos > 0;
```

- `SUM(QtdePontos)` retorna a **soma total dos pontos**.
- O **WHERE** filtra apenas as transações do período desejado e com pontos positivos.

11.2. Calculando a média com AVG

Para obter a **média de pontos ganhos**:

```
SELECT AVG(QtdePontos * 1.0) AS media_pontos
FROM transacoes;
```

- Multiplicar por `1.0` garante que a média seja calculada com **números decimais**, evitando arredondamentos inesperados.

11.3. Encontrando o mínimo e máximo

Para descobrir o **menor e maior número de pontos ganhos**:

```
SELECT
    MIN(QtdePontos) AS min_carteira,
    MAX(QtdePontos) AS max_carteira
FROM transacoes;
```

- `MIN(QtdePontos)` retorna o **menor valor** da coluna.
- `MAX(QtdePontos)` retorna o **maior valor** da coluna.

12. GROUP BY: agregando dados

Ao trabalhar com bancos de dados, muitas vezes não queremos apenas analisar **registros individuais**, mas sim **informações resumidas**.

Por exemplo, em vez de ver todas as transações de um mês, pode ser mais útil descobrir **quantos pontos cada cliente acumulou**. É nesse contexto que usamos **funções de agregação** junto com **GROUP BY**.

12.1. Diferença entre WHERE e GROUP BY

- **WHERE** → filtra **registros individuais** antes da agregação.

- **GROUP BY** → agrupa registros por uma ou mais colunas para **resumir os dados em categorias**.

12.2. Exemplo de contagem para um produto específico

Se quisermos contar quantas vezes um produto específico foi vendido:

```
SELECT COUNT(*)
FROM transacao_produto
WHERE IdProduto = 11;
```

- Retorna o número de transações do **produto com ID 11**.

12.3. Agrupando todos os produtos

Para analisar as vendas de **todos os produtos ao mesmo tempo**, usamos **GROUP BY**:

```
SELECT IdProduto, COUNT(*)
FROM transacao_produto
GROUP BY IdProduto;
```

- Agora temos uma **visão consolidada** das vendas de cada produto.
- Não precisamos rodar uma query para cada produto individualmente.

12.4. Combinando GROUP BY com funções de agregação

É possível usar **COUNT, SUM, AVG, MAX e MIN** junto com **GROUP BY** para obter **informações resumidas por categoria**.

Exemplo: somar os pontos acumulados por cliente:

```
SELECT IdCliente, SUM(QtdePontos) AS total_pontos
FROM transacoes
```

```
GROUP BY IdCliente;
```

Observação final

Sempre que perceber que está rodando várias queries mudando apenas um filtro, provavelmente **uma única query com GROUP BY** pode gerar o mesmo resultado de forma mais eficiente.

13. HAVING: como e onde usar

Uma distinção importante no SQL é **quando usar WHERE e quando usar HAVING**:

- **WHERE** → filtra os registros **antes do agrupamento**, ou seja, atua nos dados individuais.
- **HAVING** → filtra os resultados **depois do agrupamento**, atuando sobre os dados agregados.

De forma lúdica, podemos pensar:

> O HAVING é o WHERE do GROUP BY — ele aplica o filtro após a agregação, permitindo selecionar apenas grupos que atendam a determinada condição.

14. JOINS: como funcionam?

No SQL, o JOIN é utilizado para combinar informações de duas ou mais tabelas com base em uma coluna comum. Para entender melhor, imagine que temos duas tabelas: vendas e clientes. A tabela **vendas** contém informações sobre as vendas

realizadas, com colunas *id_venda, cliente e valor*. Já a tabela **clientes** contém os dados dos clientes, com colunas *id_cliente, nome e sobrenome*.

- **Tabela VENDAS:**

Id_venda	Cliente	Valor
1	1	R$ 34,00
2	1	R$ 23,00
3	2	R$ 54,00
4	4	R$ 12,00
5	5	R$ 65,00

- **Tabela CLIENTES:**

Id_cliente	Nome	Sobrenome
1	Téo	Calvo
2	Mari	Silva
3	Nah	Ataide
5	Estela	Estrea
6	Pedro	Rocha

14.1. INNER JOIN:

Um INNER JOIN retorna apenas os registros que têm correspondência em ambas as tabelas. Ou seja, ele traz apenas as vendas cujo cliente existe na tabela de clientes. Por exemplo, se um cliente fez uma venda mas não existe na tabela clientes, ou se houver um cliente sem venda, ele não aparecerá no resultado.

```
SELECT * FROM vendas INNER JOIN clientes ON idCliente = idCliente
```

ESQUERDA			DIREITA		
idVenda	idCliente	Valor	idCliente	Nome	Sobrenome
1	1	R$ 34,00	1	Téo	Calvo
2	1	R$ 23,00	1	Téo	Calvo
3	2	R$ 54,00	2	Mari	Silva
5	5	R$ 65,00	5	Estela	Estrela

14.2. LEFT JOIN:

O LEFT JOIN mantém todos os registros da tabela da esquerda (no nosso caso, vendas) e preenche com NULL as informações da tabela da direita (clientes) quando não houver correspondência. Assim, conseguimos ver todas as vendas, mesmo que alguns clientes não estejam cadastrados na tabela de clientes.

```
SELECT * FROM vendas LEFT JOIN clientes ON idCliente = idCliente
```

ESQUERDA			DIREITA		
idVenda	idCliente	Valor	idCliente	Nome	Sobrenome
1	1	R$ 34,00	1	Téo	Calvo
2	1	R$ 23,00	1	Téo	Calvo
3	2	R$ 54,00	2	Mari	Silva
4	4	R$ 12,00	NULL	NULL	NULL
5	5	R$ 65,00	5	Estela	Estrela

14.3. RIGHT JOIN:

O RIGHT JOIN funciona de forma oposta: mantém todos os registros da tabela da direita (clientes) e preenche com NULL as informações da tabela da esquerda (vendas) quando não houver correspondência. Esse tipo de join é útil quando queremos ter certeza de que todos os clientes serão considerados, mesmo que não tenham realizado nenhuma compra.

```
SELECT * FROM vendas RIGHT JOIN clientes
```

ESQUERDA			DIREITA		
idVenda	idCliente	Valor	idCliente	Nome	Sobrenome
1	1	R$ 34,00	1	Téo	Calvo
2	1	R$ 23,00	1	Téo	Calvo
3	2	R$ 54,00	2	Mari	Silva
5	5	R$ 65,00	5	Estela	Estrela
NULL	NULL	NULL	3	Nah	Ataide
NULL	NULL	NULL	6	Pedro	Rocha

14.4. FULL JOIN:

Por fim, o FULL JOIN combina as funcionalidades dos dois anteriores: ele retorna todos os registros de ambas as tabelas, preenchendo com NULL sempre que não houver correspondência. Ou seja, ele garante que todas as vendas e todos os clientes apareçam no resultado, mesmo que algumas informações estejam ausentes.

15. Subquery: o que é?

Uma **subquery** é uma **query dentro de outra query**. Ou seja, você faz uma consulta que gera um resultado e depois usa esse resultado em outra consulta.

15.1. Exemplo prático

Imagine que queremos listar todas as transações que contêm o produto chamado **“Resgatar Ponei”**:

```
SELECT *
FROM transacao_produto AS t1
WHERE t1.IdProduto IN (
    SELECT IdProduto
    FROM produtos
    WHERE DescProduto = 'Resgatar Ponei'
);
```

- A **subquery** (parte dentro dos parênteses) busca o `IdProduto` correspondente ao produto com descrição **“Resgatar Ponei”** na tabela `produtos`.
- A **query principal** (parte de fora) busca todas as linhas da tabela `transacao_produto` cujo `IdProduto` esteja na lista retornada pela subquery.
- O operador **IN** compara cada `IdProduto` da tabela `transacao_produto` com o resultado da subquery. Se houver correspondência, a linha é retornada.

Por que usar subquery?

- Evita a necessidade de **saber manualmente o IdProduto** de um produto.
- Torna o código **mais dinâmico e seguro**.
- Permite criar consultas **mais complexas e eficientes**, combinando resultados de diferentes tabelas.

16. CTE (Common Table Expression): tabelas temporárias

Uma **CTE** é uma forma de criar **tabelas temporárias dentro de uma query**. Ela permite **dividir uma consulta complexa em etapas menores**, tornando o código mais **organizado, modular e fácil de entender**.

Cada CTE funciona como uma **“mini tabela”**, que pode ser usada **logo depois na mesma query**, sem precisar criar tabelas permanentes no banco.

16.1. Exemplo: cálculo de retenção de clientes

Queremos calcular a **retenção de clientes** entre o **primeiro** e o **último dia de um curso**:

```
WITH tb_cliente_primeiro_dia AS (
    SELECT DISTINCT IdCliente
    FROM transacoes
    WHERE substr(DtCriacao, 1, 10) = '2025-08-25'
),

tb_cliente_ultimo_dia AS (
    SELECT DISTINCT IdCliente
    FROM transacoes
    WHERE substr(DtCriacao, 1, 10) = '2025-08-29'
),

tb_join AS (
    SELECT t1.IdCliente AS PrimCliente,
           t2.IdCliente AS UltCliente
    FROM tb_cliente_primeiro_dia AS t1
    LEFT JOIN tb_cliente_ultimo_dia AS t2
    ON t1.IdCliente = t2.IdCliente
)

SELECT count(PrimCliente),
       count(UltCliente),
       1. * count(UltCliente) / count(PrimCliente) AS ProporcaoClientes
FROM tb_join;
```

Como a query funciona:

1. **Primeiro dia:** selecionamos todos os clientes presentes no **primeiro dia** do curso e armazenamos na CTE `tb_cliente_primeiro_dia`.
2. **Último dia:** selecionamos os clientes presentes no **último dia** do curso e armazenamos na CTE `tb_cliente_ultimo_dia`.
3. **Combinação:** fazemos um **LEFT JOIN** entre as duas CTEs para identificar quais clientes apareceram em ambos os dias.
4. **Cálculo final:** contamos quantos clientes **começaram** e quantos **chegaram até o final**, obtendo a **proporção de retenção**.

17. WINDOW FUNCTION

As **Window Functions** permitem realizar cálculos sobre um conjunto de linhas relacionadas (uma "janela"), **sem condensar os dados em uma única linha**, como fazem funções de agregação tradicionais (SUM, COUNT, AVG etc.).

Elas são úteis para:

- Comparar valores dentro de um mesmo grupo;
- Numerar linhas;
- Calcular médias móveis;
- Identificar máximos e mínimos.

No nosso exemplo, queremos descobrir **qual foi o dia de maior engajamento de cada cliente** que iniciou o curso em `2025-08-25`.

17.1. Identificar os alunos presentes no dia 01

Primeiro, selecionamos os clientes que estavam presentes no dia 01:

```
WITH alunos_dia01 AS (
    SELECT DISTINCT IdCliente
```

```
    FROM transacoes
    WHERE substr(DtCriacao, 1, 10) = '2025-08-25'
)
```

O que acontece aqui:

- Criamos uma **CTE temporária** chamada `alunos_dia01`;
- Selecionamos apenas os clientes (`IdCliente`) que tiveram transações no dia 25/08/2025;
- `DISTINCT` garante que cada cliente apareça apenas uma vez

17.2. Obter as transações dos clientes em todos os dias do curso

Agora, juntamos a CTE `alunos_dia01` com a tabela `transacoes` para obter todas as interações desses clientes entre 25/08/2025 e 29/08/2025:

```
tb_cliente_dia AS (
    SELECT t1.IdCliente,
           substr(t2.DtCriacao, 1, 10) AS DtDia,
           COUNT(*) AS QtdeInteracoes
    FROM alunos_dia01 AS t1
    LEFT JOIN transacoes AS t2
        ON t1.IdCliente = t2.IdCliente
       AND t2.DtCriacao >= '2025-08-25'
       AND t2.DtCriacao < '2025-08-30'
    GROUP BY t1.IdCliente, DtDia
    ORDER BY t1.IdCliente, DtDia
)
```

Explicação detalhada:

- `LEFT JOIN` garante que todos os clientes apareçam, mesmo que não tenham transações em algum dia;
- `COUNT(*)` conta quantas interações cada cliente teve por dia;

- `GROUP BY` agrupa os resultados por cliente e dia;
- Resultado: uma tabela com **quantidade de interações por cliente em cada dia**.

17.3. Identificar o dia de maior engajamento usando Window Function

Agora, usamos uma **Window Function** para numerar os dias de cada cliente, ordenando pela quantidade de interações:

```
tb_rn AS (
    SELECT *,
           ROW_NUMBER() OVER (
              PARTITION BY IdCliente
              ORDER BY QtdeInteracoes DESC, DtDia
           ) AS rn
    FROM tb_cliente_dia
)
```

Como funciona:

- `PARTITION BY IdCliente`: cria uma "janela" para cada cliente;
- `ORDER BY QtdeInteracoes DESC, DtDia`: ordena os dias do cliente pelo maior número de interações (se houver empate, o dia mais antigo vem primeiro);
- `ROW_NUMBER()`: atribui um número sequencial a cada linha dentro da janela;
- Resultado: a linha com `rn = 1` corresponde ao **dia de maior engajamento de cada cliente**.

17.4. Selecionar o dia de maior engajamento

Por fim, basta selecionar apenas as linhas em que `rn = 1`, obtendo assim o dia de maior engajamento de cada cliente:

```
SELECT *
FROM tb_rn
WHERE rn = 1;
```