Aula 2

Aula de Aprofundamento de SQL

1. Revisão de SQL Básico

Nesta parte, vamos revisar os principais comandos do SQL que formam a base de praticamente todas as consultas: `SELECT`, `FROM`, `WHERE` e `LIMIT`. Esses elementos são fundamentais e vão aparecer em praticamente todos os tópicos seguintes.

—

1.1 SELECT e FROM

O comando `SELECT` é utilizado para escolher quais colunas (ou atributos) queremos visualizar em uma consulta. Já o `FROM` especifica de qual tabela os dados serão buscados.

Sintaxe básica:

```
SELECT coluna1, coluna2, ...
FROM nome_da_tabela;
```

Exemplo: Suponha que temos a tabela `Alunos`:

ID	Nome	Nota
1	Ana Costa	85
2	Bruno Lima	92
3	Carla Souza	78

Se quisermos visualizar apenas os nomes e notas:

```
SELECT Nome, Nota
FROM Alunos;
```

—

1.2 WHERE

A cláusula `WHERE` é utilizada para aplicar filtros, ou seja, retornar apenas as linhas que satisfazem uma condição.

Sintaxe:

```
SELECT coluna1, coluna2
FROM tabela
WHERE condição;
```

Exemplo: Selecionar apenas os alunos com nota maior que 80.

```
SELECT Nome, Nota
FROM Alunos
WHERE Nota > 80;
```

Resultado esperado:

Nome	Nota
Ana Costa	85
Bruno Lima	92

—

1.3 LIMIT

Em muitos casos, não precisamos visualizar todos os registros de uma vez. O comando `LIMIT` é usado para restringir a quantidade de linhas retornadas.

Exemplo: Retornar apenas os dois primeiros alunos.

```
SELECT Nome, Nota
FROM Alunos
LIMIT 2;
```

Resultado esperado:

Nome	Nota
Ana Costa	85
Bruno Lima	92

—

1.4 Exercícios de Fixação

1. Selecione apenas os nomes de todos os alunos.
2. Liste os alunos que tiraram nota menor que 80.
3. Retorne apenas um aluno qualquer (use `LIMIT`).

4. Mostre apenas os IDs e as notas de todos os alunos.

Dica: Sempre pense em `SELECT` como a pergunta "O que quero ver?"e em `FROM` como "De onde vou buscar?". Depois, utilize `WHERE` para "filtrar o que não quero"e `LIMIT` para "reduzir a quantidade de linhas".

2. GROUP BY

O comando `GROUP BY` é utilizado em SQL para agrupar registros que possuem valores iguais em uma ou mais colunas. Esse agrupamento é particularmente útil quando combinado com funções de agregação, pois nos permite resumir informações de grandes volumes de dados em resultados mais concisos.

—

2.1 Conceito de Agrupamento

Quando aplicamos um `GROUP BY`, o banco de dados divide os registros em grupos de acordo com os valores de uma ou mais colunas. Em seguida, podemos aplicar funções de agregação a esses grupos.

—

2.2 Funções de Agregação

Algumas das funções mais utilizadas em conjunto com `GROUP BY` são:

- **`COUNT`**: conta o número de registros em cada grupo.
- **`SUM`**: soma os valores numéricos de cada grupo.
- **`AVG`**: calcula a média dos valores de cada grupo.
- **`MIN`**: retorna o menor valor em cada grupo.
- **`MAX`**: retorna o maior valor em cada grupo.

—

2.3 Exemplo Prático

Considere a seguinte tabela de `Pedidos`:

ID_Pedido	Cliente	Categoria	Valor
101	Alice	Eletrônicos	3500.00
102	Bob	Papelaria	150.00
103	Alice	Eletrônicos	1200.00
104	Carla	Papelaria	75.00
105	Bob	Eletrônicos	2000.00
106	Alice	Papelaria	300.00

—

Exemplo 1: Total de pedidos por cliente.

```
SELECT Cliente, COUNT(*) AS Total_Pedidos
FROM Pedidos
GROUP BY Cliente;
```

Resultado esperado:

Cliente	Total$_{Pedidos}$
Alice	3
Bob	2
Carla	1

—

Exemplo 2: Valor total gasto em cada categoria.

```
SELECT Categoria, SUM(Valor) AS Total_Gasto
FROM Pedidos
GROUP BY Categoria;
```

Resultado esperado:

Categoria	Total$_{Gasto}$
Eletrônicos	6700.00
Papelaria	525.00

—

Exemplo 3: Maior e menor valor de pedido por cliente.

```
SELECT Cliente, MIN(Valor) AS Menor_Pedido, MAX(Valor) AS Maior_Pedido
FROM Pedidos
GROUP BY Cliente;
```

Resultado esperado:

Cliente	Menor$_{Pedido}$	Maior$_{Pedido}$
Alice	300.00	3500.00
Bob	150.00	2000.00
Carla	75.00	75.00

—

2.4 Exercícios de Fixação

1. Conte quantos pedidos cada cliente realizou.
2. Calcule o valor médio dos pedidos em cada categoria.
3. Mostre o cliente que possui o maior valor total de pedidos somados.
4. Liste, para cada categoria, o menor e o maior pedido registrado.

Dica: Sempre que utilizar funções de agregação (`COUNT`, `SUM`, `AVG`, etc.), lembre-se de combinar com `GROUP BY` para obter resultados organizados por grupo.

3. HAVING

Quando trabalhamos com consultas em SQL que envolvem funções de agregação (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`, etc.), muitas vezes precisamos filtrar os resultados após o agrupamento dos dados. Nesses casos, não podemos usar a cláusula `WHERE`, mas sim a cláusula `HAVING`.

—

3.1 Diferença entre WHERE e HAVING

- **`WHERE`**: filtra registros antes do agrupamento. Ele atua sobre as linhas individuais da tabela.
- **`HAVING`**: filtra resultados depois do agrupamento. Ele atua sobre os grupos criados pelo `GROUP BY`.

Exemplo ilustrativo: - Se quisermos selecionar apenas os alunos que têm nota maior que 80, usamos `WHERE`. - Se quisermos selecionar apenas as turmas cuja média das notas é maior que 80, precisamos de `HAVING`.

—

3.2 Exemplo prático com GROUP BY e HAVING

Considere a seguinte tabela de `Notas`:

ID_Aluno	Disciplina	Nota
1	Matemática	85
1	História	90
2	Matemática	70
2	História	75
3	Matemática	95
3	História	88

—

Exemplo 1: Média de notas por disciplina.

```
SELECT Disciplina, AVG(Nota) AS Media
FROM Notas
GROUP BY Disciplina;
```

Resultado esperado:

Disciplina	Média
Matemática	83.3
História	84.3

—

Exemplo 2: Selecionar apenas disciplinas cuja média de notas seja maior que 84.

```
SELECT Disciplina, AVG(Nota) AS Media
FROM Notas
GROUP BY Disciplina
HAVING AVG(Nota) > 84;
```

Resultado esperado:

Disciplina	Média
História	84.3

—

Exemplo 3: Contar quantos alunos existem em cada disciplina e mostrar apenas disciplinas com mais de 2 registros.

```
SELECT Disciplina, COUNT(*) AS Total_Alunos
FROM Notas
GROUP BY Disciplina
HAVING COUNT(*) > 2;
```

Resultado esperado:

Disciplina	Total$_Alunos$
Matemática	3
História	3

—

3.3 Exercícios de Fixação

1. Liste a nota média de cada disciplina.
2. Mostre apenas as disciplinas cuja média seja maior que 85.
3. Conte quantos registros cada disciplina possui e filtre apenas disciplinas com pelo menos 3 alunos.

4. Qual disciplina tem a menor nota máxima registrada?

Dica: Lembre-se:

- Use `WHERE` para filtrar **linhas individuais**.
- Use `HAVING` para filtrar **grupos agregados**.

4. ORDER BY

O comando `ORDER BY` é utilizado para ordenar os resultados de uma consulta em SQL. Ele pode organizar os dados em ordem crescente (`ASC`) ou decrescente (`DESC`). Por padrão, a ordenação é feita de forma crescente.

Sintaxe:

```
SELECT coluna1, coluna2, ...
FROM tabela
ORDER BY coluna1 [ASC|DESC], coluna2 [ASC|DESC];
```

Considerando a tabela a seguir:

ID	Nome	Nota
1	Ana Costa	85
2	Bruno Lima	92
3	Carla Souza	78
4	Diego Alves	92
5	Fernanda Silva	70

Seguem alguns exemplos de utilização:

Exemplo 1: Ordenar os alunos pelo nome em ordem alfabética.

```
SELECT id, nome, nota
FROM alunos
ORDER BY nome ASC;
```

ID	Nome	Nota
1	Ana Costa	85
2	Bruno Lima	92
3	Carla Souza	78
4	Diego Alves	92
5	Fernanda Silva	70

Exemplo 2: Ordenar os alunos da maior para a menor nota.

```
SELECT id, nome, nota
FROM alunos
ORDER BY nota DESC;
```

ID	Nome	Nota
2	Bruno Lima	92
4	Diego Alves	92
1	Ana Costa	85
3	Carla Souza	78
5	Fernanda Silva	70

Exemplo 3: Ordenar usando múltiplas colunas.
Primeiro pela nota (decrescente) e, em caso de empate, pelo nome (crescente).

```
SELECT id, nome, nota
FROM alunos
ORDER BY nota DESC, nome ASC;
```

ID	Nome	Nota
2	Bruno Lima	92
4	Diego Alves	92
1	Ana Costa	85
3	Carla Souza	78
5	Fernanda Silva	70

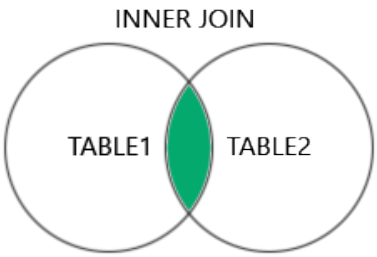
5. JOIN

À medida que os bancos de dados crescem, é fundamental organizar as informações de forma inteligente. Em vez de criar uma única tabela gigante com todos os dados (o que geraria muita repetição e ineficiência), podemos dividir os dados em tabelas menores e temáticas. Por exemplo, uma tabela para Clientes, outra para Pedidos, e uma terceira para Produtos.

Apesar desse tipo de organização ser ideal, ele gera o seguinte problema: e se quisermos obter informações que só são completas quando unimos diferentes dados de diferentes tabelas? E se quisermos, por exemplo, observar os produtos que um determinado cliente comprou? Para isso, podemos utilizar o comando JOIN.

O comando JOIN basicamente permite a união de registros de diferentes relações com base numa condição. Essa condição normalmente é uma igualdade entre valores de atributos de diferentes relações. Por exemplo, podemos realizar um JOIN entre as relações Clientes e Pedidos com base no id do cliente, assim obtendo registros que contém um cliente e seu respectivo pedido. Para tanto, temos diferentes tipos de JOINs, como veremos a seguir:

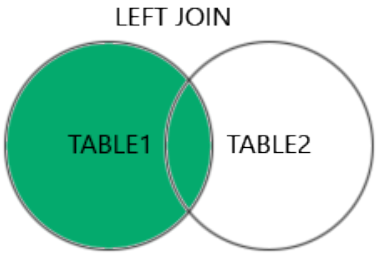
- **Inner Join (Junção Interna):** Retorna apenas os registros que possuem uma correspondência em ambas as tabelas. Para facilitar, imagine que cada relação é um conjunto. O INNER JOIN corresponde aos registros da intersecção dos conjuntos, como na figura abaixo.



Em SQL, podemos utilizá-lo da seguinte forma:

```
SELECT <Lista de Atributos>
FROM Tabela1
INNER JOIN Tabela2 ON Tabela1.Atributo = Tabela2.Atributo;
```

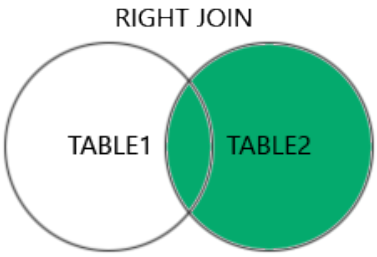
- **Left Join:** Retorna todos os registros da tabela à esquerda (tabela 1) e os registros correspondentes da tabela à direita. Se não houver correspondência na tabela da direita para um registro da esquerda, os campos da direita virão como NULL.



Em SQL, podemos utilizá-lo da seguinte forma:

```
SELECT <Lista de Atributos>
FROM Tabela1
LEFT JOIN Tabela2 ON Tabela1.Atributo = Tabela2.Atributo;
```

- **Right Join:** É o inverso do Left Join. Retorna os registros da tabela 2 e os registros correspondentes da tabela à esquerda. Em caso de não correspondência, registros da tabela à esquerda virão como NULL.



Em SQL, podemos utilizá-lo da seguinte forma:

```
SELECT <Lista de Atributos>
FROM Tabela1
RIGHT JOIN Tabela2 ON Tabela1.Atributo = Tabela2.Atributo;
```

Para ilustrar como o JOIN funciona na prática, suponha que temos as relações Clientes e Pedidos, conforme os registros abaixo.

ID_Cliente	Nome	Cidade
1	Alice Silva	São Paulo
2	Bob Souza	São Carlos
3	Maria Oliveira	Rio de Janeiro
4	Gabriel Santos	Minas Gerais

ID_Pedido	ID_Cliente	Produto	Valor
101	1	Notebook	3500.00
102	3	Mouse	80.00
103	2	Teclado	150.00
104	2	Monitor	1100.00

Nesse caso, queremos criar uma consulta que mostre o nome de cada cliente e seu respectivo produto. Para isso, observe que a relação Produtos possui como chave estrangeira o id do cliente. Logo, podemos utilizar isso para realizar o JOIN, como está abaixo:

```
SELECT
    Clientes.Nome,
    Pedidos.Produto,
    Pedidos.Valor
FROM
    Clientes
INNER JOIN
    Pedidos ON Clientes.ID_Cliente = Pedidos.ID_Cliente;
```

Com isso, teremos um resultado semelhante ao mostrado abaixo.

Nome	Produto	Valor
Alice Silva	Notebook	3500.00
Maria Oliveira	Mouse	80.00
Bob Souza	Teclado	150.00
Bob Souza	Monitor	1100.00

Nesse exemplo, vale a pena ressaltar o seguinte ponto: em situações em que precisamos escrever inúmeros atributos (em SELECT, WHERE, JOIN, etc.) de uma relação, ao invés de utilizar a estrutura Nome_Relação.Atributo, podemos utilizar o denominado aliases (apelido). A ideia é que podemos chamar a relação por um nome mais simples e utilizá-lo no restante da consulta. No caso anterior, poderíamos fazer:

```
SELECT
    C.Nome,
    Pedidos.Produto,
    Pedidos.Valor
FROM
    Clientes AS C
INNER JOIN
    Pedidos ON C.ID_Cliente = C.ID_Cliente;
```

6. SUBQUERY

O conceito de **subquery** (ou subconsulta) é, essencialmente, uma consulta SQL aninhada dentro de outra consulta principal. Imagine que você precisa resolver um problema, mas para isso, precisa de uma informação que só pode ser obtida por meio de outra consulta. É aí que as subqueries entram em cena, agindo como um "sub-passo"para o seu objetivo final.

A grande vantagem das subqueries é que elas permitem criar consultas complexas de forma modular e legível. Você pode usá-las para:

- **Filtragem de dados (cláusula `WHERE`):** Para filtrar registros da consulta principal com base no resultado da subquery.
- **Cálculos intermediários (cláusula `SELECT`):** Para retornar um único valor que será usado como uma nova coluna na sua consulta principal.
- **Tabelas temporárias (cláusula `FROM`):** Para criar uma tabela virtual "em tempo real"que será usada na consulta principal.

—

Subquery na cláusula `WHERE`

Este é um dos usos mais comuns. Você utiliza uma subquery para retornar um conjunto de valores que serão usados na condição da sua cláusula `WHERE`.

Exemplo prático:

Vamos supor que você tem uma tabela de `Funcionarios` e uma tabela de `Departamentos`. Você quer encontrar o nome de todos os funcionários que trabalham no departamento chamado 'Vendas'.

Uma maneira de fazer isso sem uma subquery seria primeiro encontrar o `ID` do departamento 'Vendas' e depois usar esse `ID` na consulta de funcionários. Com uma subquery, você faz tudo em uma única linha.

```
SELECT
    Nome
FROM
    Funcionarios
WHERE
    ID_Departamento = (
        SELECT
            ID_Departamento
        FROM
            Departamentos
        WHERE
            Nome_Departamento = 'Vendas'
    );
```

Como funciona:

1. A subquery interna é executada primeiro: `SELECT ID_Departamento FROM Departamentos WHERE Nome_Departamento = 'Vendas'`. Ela retorna o `ID_Departamento` correspondente a 'Vendas'.

2. O valor retornado (por exemplo, `5`) é então substituído na consulta principal.

3. A consulta principal é executada como se você tivesse escrito `SELECT Nome FROM Funcionarios WHERE ID_Departamento = 5;`.

Você também pode usar operadores como `IN` para lidar com múltiplos resultados da subquery. Por exemplo, para encontrar os funcionários que trabalham em 'Vendas' ou 'Marketing', a subquery retornaria mais de um `ID_Departamento`.

```
SELECT
    Nome
FROM
    Funcionarios
```

```
WHERE
    ID_Departamento IN (
        SELECT
            ID_Departamento
        FROM
            Departamentos
        WHERE
            Nome_Departamento IN ('Vendas', 'Marketing')
    );
```

—

Subquery na cláusula `SELECT`

Neste caso, a subquery atua como uma coluna, retornando um único valor para cada linha da consulta principal. Esse valor pode ser um cálculo ou uma contagem, por exemplo.

Exemplo prático:

Vamos supor que você tem uma tabela de `Clientes` e uma tabela de `Pedidos`. Você quer listar o nome de cada cliente e, ao lado, a quantidade total de pedidos que cada um fez.

```
SELECT
    Nome_Cliente,
    (
        SELECT
            COUNT(ID_Pedido)
        FROM
            Pedidos
        WHERE
            Pedidos.ID_Cliente = Clientes.ID_Cliente
    ) AS Total_Pedidos
FROM
    Clientes;
```

Como funciona:

1. A consulta principal itera sobre cada cliente na tabela `Clientes`.
2. Para cada linha de cliente, a subquery é executada. Ela conta quantos pedidos na tabela `Pedidos` correspondem ao `ID_Cliente` da linha atual da consulta principal.
3. O resultado da contagem é retornado como uma nova coluna chamada `Total_Pedidos` para aquela linha.
4. É importante notar que a subquery aqui é uma **subquery correlacionada**, pois ela se refere a uma coluna (`Clientes.ID_Cliente`) da consulta principal.

—

Subquery na cláusula `FROM`

Neste cenário, a subquery retorna um conjunto de resultados que é tratado como uma nova tabela virtual temporária na cláusula `FROM`. Essa "tabela"pode então ser usada na consulta principal.

Exemplo prático:

Imagine que você quer encontrar o nome dos clientes que fizeram mais de dois pedidos. Para isso, primeiro você precisa agrupar os clientes por ID e contar seus pedidos. A subquery faria isso, e a consulta principal usaria o resultado.

```
SELECT
    C.Nome
FROM
    Clientes AS C
INNER JOIN
    (
        SELECT
            ID_Cliente,
            COUNT(*) AS Total_Pedidos
        FROM
            Pedidos
        GROUP BY
            ID_Cliente
        HAVING
            COUNT(*) > 2
    ) AS Clientes_Com_Mais_De_2_Pedidos
ON
    C.ID_Cliente = Clientes_Com_Mais_De_2_Pedidos.ID_Cliente;
```

Como funciona:

1. A subquery interna é executada primeiro. Ela agrupa os pedidos por `ID_Cliente` e filtra apenas os grupos que têm mais de dois pedidos (`HAVING COUNT(*) > 2`). O resultado é uma tabela temporária (apelidada de `Clientes_Com_Mais_De_2_Pedidos`) com duas colunas: `ID_Cliente` e `Total_Pedidos`.

2. A consulta principal então faz um `INNER JOIN` entre a tabela `Clientes` e essa nova tabela temporária, unindo as linhas com base no `ID_Cliente`.

3. Por fim, ela seleciona o nome dos clientes que satisfazem a condição.

Este uso de subqueries é particularmente útil quando você precisa realizar agrupamentos ou aplicar funções de agregação antes de unir os dados com outras tabelas.

—

Vantagens e Desvantagens das Subqueries

Vantagens	Desvantagens
Legibilidade: facilita a leitura de consultas complexas, dividindo-as em partes lógicas.	**Performance:** podem ser menos eficientes que `JOIN`s em certos casos, dependendo do otimizador do SGBD.
Flexibilidade: permitem operações que seriam difíceis ou impossíveis com outros comandos.	**Complexidade:** subqueries muito aninhadas tornam o código difícil de depurar e manter.
Modularidade: possibilitam pensar no problema em etapas, onde cada subquery resolve uma parte do problema maior.	**Retorno:** subqueries em `SELECT` e em `WHERE` (sem `IN`) só podem retornar um único valor.

Em geral, prefira `JOIN` para unir tabelas quando a operação for simplesmente combinar dados — é a abordagem mais performática na maioria dos casos. Use **subqueries** quando precisar de filtros dinâmicos, cálculos intermediários ou quando a lógica do problema se encaixar melhor em "consulta dentro de consulta".