SQL A2 Operadores de Consulta e Funções de Agregação e Manipulação de Data

Índice

Operadores `not`, `in`, `between`, `is null` e `is not null`

Para exemplificar cada um dos operadores, iremos utilizar a tabela de funcionários abaixo:

ID	Nome	Idade	Profissão	Salário (R$)	Telefone	Email	Departamento
1	Carminha	45	Analista de Sistemas	8500.00	(21) 9876-5432	carminha@ti.com	Desenvolvimento
2	Tufão	35	Desenvolvedor Front-end	7200.00	(21) 8765-4321	tufao@dev.com	Desenvolvimento
3	Max	28	Engenheiro de Software	9500.00	(21) 7654-3210	max@engenharia.com	Desenvolvimento
4	Jorginho	30	Administrador de Redes	6800.00	(21) 6543-2109	jorginho@redes.com	Infraestrutura
5	Rita	25	UX/UI Designer	6000.00	(21) 5432-1098	rita@design.com	Design
6	Leleco	40	Analista de Dados	7800.00	(21) 4321-0987	leleco@dados.com	Dados
7	Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com	Desenvolvimento
8	Ivana	50	Gerente de Projetos	12000.00	(21) 2109-8765	ivana@gerencia.com	Gestão
9	Monalisa	27	Desenvolvedora Back-end	7000.00	(21) 1098-7654	monalisa@backend.com	Desenvolvimento
10	Cadu	33	Especialista em Cloud	9000.00	(21) 0987-6543	cadu@cloud.com	Infraestrutura
11	Dona Nenê	60	Suporte Técnico	4500.00	(21) 9876-5432	nenê@suporte.com	Suporte
12	Tuco	29	Analista de Segurança	8000.00	(21) 8765-4321	tuco@seguranca.com	Segurança
13	Bebel	26	Product Manager	10000.00	(21) 7654-3210	bebel@product.com	Gestão
14	Agostinho	55	Arquiteto de Soluções	11000.00	(21) 6543-2109	agostinho@arquiteto.com	Arquitetura
15	Lineu	58	Consultor em TI	8500.00	(21) 5432-1098	lineu@consultoria.com	Consultoria

- `not`: é utilizado para negar uma condição, invertendo o resultado de uma expressão booleana.

```sql
-- Retorna todas as pessoas que recebem um salário maior que 5000
SELECT * FROM funcionario WHERE salario > 5000;

-- Retorna todas as pessoas que NÃO recebem um salário maior que 5000 (ou seja, um salário menor que 5000)
SELECT * FROM funcionario WHERE NOT salario > 5000;
```

Nome	Idade	Profissão	Salário (R$)	Telefone	Email
Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com
Dona Nenê	60	Suporte Técnico	4500.00	(21) 9876-5432	nenê@suporte.com

- `in`: funciona como um atalho para múltiplas condições `OR`. Ele verifica se um valor corresponde a qualquer um dos valores especificados em uma lista. Ele simplifica a escrita de consultas quando você precisa comparar um valor com múltiplas opções, evitando a necessidade de escrever várias condições com o operador `OR`.

```sql
-- utilizando OR
SELECT * FROM funcionario WHERE departamento = 'Desenvolvimento' or departamento = 'Design' or departamento = 'Gestão';

-- utilizando IN
SELECT * FROM funcionario WEHRE departamento IN ('Desenvolvimento', 'Design', 'Gestão');
```

Nome	Idade	Profissão	Salário (R$)	Telefone	Email	Departamento
Carminha	45	Analista de Sistemas	8500.00	(21) 9876-5432	carminha@ti.com	Desenvolvimento
Tufão	35	Desenvolvedor Front-end	7200.00	(21) 8765-4321	tufao@dev.com	Desenvolvimento
Max	28	Engenheiro de Software	9500.00	(21) 7654-3210	max@engenharia.com	Desenvolvimento
Rita	25	UX/UI Designer	6000.00	(21) 5432-1098	rita@design.com	Design
Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com	Desenvolvimento
Ivana	50	Gerente de Projetos	12000.00	(21) 2109-8765	ivana@gerencia.com	Gestão
Monalisa	27	Desenvolvedora Back-end	7000.00	(21) 1098-7654	monalisa@backend.com	Desenvolvimento
Bebel	26	Product Manager	10000.00	(21) 7654-3210	bebel@product.com	Gestão

- `between`: utilizado para filtrar os resultados dentro de um intervalo de tempo, seja para números ou datas.

```sql
-- selecionar funcionarios que possuem idade entre 20 e 30
SELECT * FROM funcionario WHERE idade BETWEEN 20 and 30;

-- é a mesma coisa de:
SELECT * FROM funcionario WHERE idade ≥ 20 and idade ≤ 30;
```

Nome	Idade	Profissão	Salário (R$)	Telefone	Email	Departamento
Max	28	Engenheiro de Software	9500.00	(21) 7654-3210	max@engenharia.com	Desenvolvimento
Rita	25	UX/UI Designer	6000.00	(21) 5432-1098	rita@design.com	Design
Jorginho	30	Administrador de Redes	6800.00	(21) 6543-2109	jorginho@redes.com	Infraestrutura
Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com	Desenvolvimento
Monalisa	27	Desenvolvedora Back-end	7000.00	(21) 1098-7654	monalisa@backend.com	Desenvolvimento
Tuco	29	Analista de Segurança	8000.00	(21) 8765-4321	tuco@seguranca.com	Segurança
Bebel	26	Product Manager	10000.00	(21) 7654-3210	bebel@product.com	Gestão

- `is not null`: é usado para verificar se uma coluna **não contém valores nulos**. Ou seja, é utilizado para filtrar registros que possuem dados válidos em uma coluna específica, ignorando aqueles onde o valor é **NULL**. Por exemplo: Selecionar apenas os cadastros dos funcionários que possuem o email cadastrado.

```sql
SELECT * FROM funcionario WHERE email IS NOT NULL;
```

- `is null`: funciona de forma contrária ao `is not null`. Enquanto o `is not null` verifica se uma coluna **não contém valores nulos**, o `is null` é utilizado para verificar se uma coluna **contém valores nulos**. Por exemplo: selecionar apenas os cadastros que possuem o campo salário nulo.

```sql
SELECT * FROM funcionario WHERE salario IS NULL;
```

Funções de Agregação - `count`, `sum`, `max`, `min` e `avg`

As **funções de agregação** são utilizadas em SQL para realizar cálculos em conjuntos de dados e retornar um único valor como resultado.

- `sum` : utilizado para retornar a soma de todos os valores de uma coluna especifica (ignora valores nulos).

```sql
SELECT SUM(salario) FROM funcionario WHERE departamento = 'Gestão';
```

SUM(salario)
22000.00

- `avg` : calcula a média dos valores de uma coluna especifica, sendo útil para encontrar a média de um conjunto de números, como a idade média dos clientes ou a média de vendas dentro de um período, por exemplo

Nome	Idade	Profissão	Salário (R$)	Telefone	Email	Departamento
Carminha	45	Analista de Sistemas	8500.00	(21) 9876-5432	carminha@ti.com	Desenvolvimento
Tufão	35	Desenvolvedor Front-end	7200.00	(21) 8765-4321	tufao@dev.com	Desenvolvimento
Max	28	Engenheiro de Software	9500.00	(21) 7654-3210	max@engenharia.com	Desenvolvimento
Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com	Desenvolvimento
Monalisa	27	Desenvolvedora Back-end	7000.00	(21) 1098-7654	monalisa@backend.com	Desenvolvimento

```sql
SELECT AVG(idade) FROM funcionario WHERE departamento = 'Desenvolvimento';
```

AVG(idade)
31.4

- `count` : é usada para contar o número de registros (linhas) que atendem a uma condição específica.

```sql
-- Contar quantos registros no total possui na tabela
SELECT COUNT(*) FROM funcionario
```

COUNT(')
15

```sql
-- conta a quantidade de linhas que não possuem valores nulos em uma coluna especifica
SELECT COUNT(salario) FROM funcionario;
```

Nome	Idade	Profissão	Salário (R$)	Telefone	Email
Nina	22	Estagiária em TI	1500.00	(21) 3210-9876	nina@estagio.com
Dona Nenê	60	Suporte Técnico	4500.00	(21) 9876-5432	nenê@suporte.com
Beiçola	66	Tio do Pastel	null	null	null

Por exemplo, se o total de linhas na tabela fossem 3, a query retornaria 2, porque apenas 2 registros possuem valores não nulos na coluna **salário**.

COUNT(salario)
2

```sql
-- conta o número de válores únicos em uma coluna, ignorando valores duplicados
SELECT COUNT(DISTINCT departamento) FROM funcionario;
```

Ou seja, na coluna departamentos temos as categorias: Desenvolvimento, Infraestrutura, Design, Dados, Gestão, Suporte, Arquitetura, Consultoria e Segurança, um total de 9 departamentos, logo esse será o resultado da query.

- Se houver **valores duplicados** na coluna **departamento**, eles serão contados apenas uma vez.
- Se houver **valores nulos** na coluna **departamento**, eles serão **ignorados**.
- A principal diferença entre `COUNT(DISTINCT departamento)` e `COUNT(departamento)` está na forma como os valores são contados:
  - `COUNT(departamento)` : Conta **todas** as ocorrências da coluna, incluindo valores duplicados. Ou seja, se um departamento aparecer várias vezes, cada ocorrência será contada separadamente.
  - `COUNT(DISTINCT departamento)` : Conta **apenas os valores únicos**, eliminando duplicatas. Isso significa que, mesmo que um departamento apareça várias vezes na tabela, ele será contado apenas uma vez.

COUNT(DISTINCT departamento)
9

- `min` : retorna o **menor valor** de uma coluna.

```sql
-- Menor salário da empresa
SELECT min(salario) AS menor_salario FROM funcionario;
```

menor_salario
1500.00

- `max` : retorna o **maior valor** de uma coluna.

```sql
-- Maior idade registrada na tabela
SELECT max(Idade) AS maior_idade FROM funcionarios;
```

maior_idade
60

Operador `AS` (alias)

Observe que, por padrão, os nomes das colunas no resultado das funções de agregação (`COUNT(DISTINCT departamento)` , `SUM(salario)` , `AVG(idade)`) são exatamente os nomes das funções utilizadas. Isso pode tornar o resultado menos intuitivo, especialmente quando há várias funções na mesma consulta, dificultando a compreensão do que cada valor representa.

Para tornar os resultados mais claros e legíveis, podemos utilizar o operador `AS` , que permite atribuir um **apelido** (alias) às colunas no resultado da consulta. Isso melhora a interpretação dos dados sem modificar os nomes das colunas na tabela original.

```sql
SELECT COUNT(DISTINCT departamento) AS quantidade_de_departamentos FROM funcionario;
```

quantidade_de_departamentos
9

Operador `LIKE`

O operador `LIKE` é usado para encontrar palavras ou frases que sigam um determinado padrão. Ele é útil quando você não sabe exatamente o texto completo que procura, mas tem uma ideia de parte dele.

Operadores	Descrição
`%`	Representa qualquer sequência de caracteres, incluindo uma sequência vazia. Ele pode ser usado no início, no meio ou no final de um padrão para encontrar textos que comecem, terminem ou contenham determinado padrão. Por exemplo, `'a%'` encontra qualquer texto que comece com "a", `'%a'` encontra textos que terminam com "a", e `'%a%'` encontra textos que contenham "a" em qualquer posição.
`_`	Representa um único caractere. Ele é usado para definir a posição exata de um caractere dentro de um padrão. Por exemplo, `'a_'` encontra textos onde "a" é seguido por exatamente um caractere, como "ab" ou "ax", mas não "abc".

Exemplos

- Buscar nomes que começam com 'Jo' - irá retornar João, José, Jonas, etc.

```sql
SELECT * FROM clientes WHERE nome LIKE 'Jo%';
```

- Buscar nomes que terminam com 'a' - irá retornar Maria, Ana, Julia, etc.

```sql
SELECT * FROM clientes WHERE nome LIKE '%a';
```

- Buscar nomes que contém 'silv'em qualquer parte do texto - irá retornar Silva, Maria da Silva, Silvério, etc.

```sql
SELECT * FROM clientes WHERE nome LIKE '%silv%';
```

- Buscar nomes com exatamente 3 caracteres - Irá retornar Léo, Ana, Bia, etc.

```sql
SELECT * FROM usuarios WHERE nome LIKE '___';
```

- Buscar nomes com 5 caracteres e que começa com 'A' - Irá retornar Alice, André, Aline, etc.

```sql
SELECT * FROM usuarios WHERE nome LIKE 'A____';
```

- Buscar nomes que possuem A na penúltima posição, independente da quantidade de caracteres - Irá retornar Lucas, por exemplo.

```sql
SELECT * FROM usuarios WHERE nome LIKE '%a_';
```

Observações

O operador `LIKE` pode ser **case-sensitive** (sensível a maiúsculas e minúsculas) em alguns sistemas de banco de dados, o que significa que ele diferencia letras maiúsculas de minúsculas durante a busca. Por exemplo, se você buscar por `'joão'` (tudo em minúsculo), a consulta pode não retornar variações como `'João'`, `'jOãO'`, etc., pois essas formas não correspondem exatamente ao padrão especificado na query.

Para contornar essa limitação, podemos utilizar as funções `LOWER` (para converter textos em minúsculas) ou `UPPER` (para converter textos em maiúsculas). Essas funções transformam os dados antes da comparação, permitindo que a busca ignore diferenças entre letras maiúsculas e minúsculas.

```sql
-- garantem que retorne todos os registros que terminem com 'o', independente de estar maiúscula ou minúscula
SELECT * FROM clientes WHERE LOWER(nome) LIKE '%o';
SELECT * FROM clientes WHERE UPPER(nome) LIKE '%O';

-- retorna todos os nomes em letras minusculas
SELECT LOWER(nome) FROM clientes;

-- retorna todos os nomes em letras maiúsculas
SELECT UPPER(nome) FROM clientes
```

`GROUP BY`, `HAVING` e `ORDER BY`

Para esse tópico iremos utilizar a tabela abaixo que contém informações sobre as transações de três bancos:

ID Cliente	Banco	Tipo de Pagamento	Valor (R$)	Data de Pagamento
101	Banco Howl	Pix	250.00	2025-03-01
102	Banco Calcifer	TED	1,500.00	2025-03-02
103	Banco Sophie	Boleto	320.50	2025-03-02
101	Banco Howl	Cartão de Crédito	75.90	2025-03-03
104	Banco Calcifer	Cartão de Débito	680.00	2025-03-04
102	Banco Sophie	Pix	50.00	2025-03-04
105	Banco Howl	TED	2,300.75	2025-03-05
106	Banco Calcifer	Boleto	400.00	2025-03-06
103	Banco Sophie	Cartão de Crédito	199.99	2025-03-07

ID Cliente	Banco	Tipo de Pagamento	Valor (R$)	Data de Pagamento
104	Banco Howl	Cartão de Débito	850.00	2025-03-08
107	Banco Calcifer	Pix	300.00	2025-03-09
108	Banco Sophie	TED	1,200.00	2025-03-10
101	Banco Howl	Boleto	210.30	2025-03-11
102	Banco Calcifer	Cartão de Crédito	500.00	2025-03-12
105	Banco Sophie	Cartão de Débito	620.45	2025-03-13
103	Banco Howl	Pix	150.00	2025-03-14
106	Banco Calcifer	TED	3,000.00	2025-03-15
104	Banco Sophie	Boleto	275.80	2025-03-16
107	Banco Howl	Cartão de Crédito	999.99	2025-03-17
108	Banco Calcifer	Cartão de Débito	410.00	2025-03-18

Como funciona o `GROUP BY`?

O `GROUP BY` é utilizado para agrupar linhas que possuem o mesmo valor em uma ou mais colunas especificadas, sendo frequentemente utilizado com funções de agregação. Abaixo temos alguns exemplos de perguntas que podemos responder utilizando o `group by` explorando a tabela acima:

- **Quantos pagamentos estão registrados para cada banco?** - Aqui o `group by` é utilizado para agrupas a quantidade de pagamentos que existem pra cada banco, então utilizamos a coluna `banco` para que o SQL saiba qual coluna ele deve utilizar para identificar valores parecidos em cada registro.

```sql
select banco, COUNT(*) as total_transacoes from pagamentos group by banco;
```

banco	total_transacoes
Banco Calcifer	7
Banco Sophie	6
Banco Howl	7

- **Qual o valor total de pagamentos processados por cada banco?** - O `SUM(valor)` soma todos os valores de pagamentos para cada banco, agrupados pelo `GROUP BY`.

```sql
select banco, SUM(valor) as "Valor total de pagamentos" from pagamentos group by banco;
```

banco	Valor total de pagamentos
Banco Calcifer	6790.00
Banco Sophie	2666.74
Banco Howl	4836.94

- **Quantas transações foram feitas por cada tipo de pagamento?** - O `GROUP BY` agrupa as transações por tipo de pagamento e o `COUNT(*)` conta quantas transações foram feitas para cada tipo.

```sql
select tipo_pagamento, count(*) from pagamentos group by tipo_pagamento;
```

tipo_pagamento	count
Boleto	4
Pix	4
Cartão de Crédito	4
TED	4
Cartão de Débito	4

- **Quantos pagamentos foram feitos em cada dia registrado?** - O `GROUP BY` agrupa os pagamentos por data e o `COUNT(*)` conta quantos pagamentos foram feitos em cada dia.

```
select data_pagamento, count(*) from pagamentos group by data_pagamento;
```

data_pagamento	count
2025-03-11	1
2025-03-02	2
2025-03-13	1
2025-03-05	1
2025-03-14	1
2025-03-04	2
2025-03-01	1
2025-03-07	1
2025-03-10	1
2025-03-12	1
2025-03-17	1
2025-03-16	1
2025-03-06	1
2025-03-15	1
2025-03-18	1
2025-03-08	1
2025-03-09	1
2025-03-03	1

- **Quantas transações cada cliente realizou?** - O `GROUP BY` agrupa as transações por cliente e o `COUNT(*)` conta quantas transações cada cliente realizou.

```
select id_cliente, count(*) from pagamentos group by id_cliente;
```

id_cliente	count
101	3
108	2
103	3
104	3
105	2
107	2
102	3
106	2

- **Quantas transações cada cliente fez por banco?** - O `GROUP BY` agrupa as transações por cliente e banco, e o `COUNT(*)` conta quantas transações cada cliente fez em cada banco. Observe que dessa vez utilizamos duas colunas para que o SQL entenda que ele deve identificar e agrupas registros parecidos em relação aos valores contidos nessas duas colunas.

```
select id_cliente, banco, count(*) from pagamentos group by id_cliente, banco;
```

id_cliente	banco	count
103	Banco Sophie	2
108	Banco Sophie	1

id_cliente	banco	count
101	Banco Howl	3
108	Banco Calcifer	1
105	Banco Howl	1
104	Banco Calcifer	1
104	Banco Howl	1
105	Banco Sophie	1
102	Banco Calcifer	2
107	Banco Howl	1
102	Banco Sophie	1
106	Banco Calcifer	2
103	Banco Howl	1
107	Banco Calcifer	1
104	Banco Sophie	1

- **Qual o total de pagamentos por banco e tipo de pagamento?** - O `GROUP BY` agrupa os pagamentos por banco e tipo de pagamento, e o `COUNT(*)` conta quantas transações foram feitas para cada combinação de banco e tipo de pagamento.

```
select banco, tipo_pagamento, count(*) from pagamentos group by tipo_pagamento, banco;
```

banco	tipo_pagamento	count
Banco Calcifer	TED	2
Banco Calcifer	Cartão de Débito	2
Banco Calcifer	Boleto	1
Banco Calcifer	Cartão de Crédito	1
Banco Calcifer	Pix	1
Banco Howl	Cartão de Crédito	2
Banco Howl	Cartão de Débito	1
Banco Howl	Pix	2
Banco Howl	Boleto	1
Banco Howl	TED	1
Banco Sophie	Boleto	2
Banco Sophie	Cartão de Crédito	1
Banco Sophie	Cartão de Débito	1
Banco Sophie	Pix	1
Banco Sophie	TED	1

- **Qual é o valor mínimo e máximo de pagamento registrado para cada banco na tabela de pagamentos?**

```
select banco, min(valor) AS Valor_Minimo, max(valor) AS Valor_Maximo FROM pagamentos GROUP BY banco;
```

banco	valor_minimo	valor_maximo
Banco Calcifer	300.00	3000.00
Banco Sophie	50.00	1200.00
Banco Howl	75.90	2300.75

Porque `SELECT * FROM table GROUP BY column` não faz sentido e retorna erro?

Quando utilizamos o comando `GROUP BY`, estamos instruindo o banco de dados a agrupar as linhas da tabela com base em valores iguais em uma ou mais colunas específicas. No entanto, o `SELECT *` (que seleciona todas as colunas da tabela) em conjunto com o `GROUP BY` não faz sentido e geralmente resulta em erro.

Se você executar o comando:

```
SELECT * FROM pagamentos GROUP BY banco;
```

O banco de dados ficará confuso. Ele sabe que precisa agrupar as linhas com base na coluna `banco`, mas **não sabe o que fazer com as outras colunas** (como `id_Cliente`, `tipo_pagamento`, `valor`, etc.).

- **Qual valor de `ID cliente` ele deve retornar para cada grupo?**
- **Qual `tipo de pagamento` deve ser exibido?**
- **Qual `valor` deve ser mostrado?**

Essas perguntas não têm uma resposta clara, pois o `GROUP BY` está agrupando várias linhas em uma única linha por banco, mas as outras colunas contêm valores diferentes que não podem ser simplesmente "agrupados" sem uma função de agregação.

- o banco de dados não sabe o que fazer com as colunas que não fazem parte do `GROUP BY`, ou que não são usadas em uma função de agregação. Isso acontece porque, após o agrupamento, o banco de dados precisa de uma regra para retornar as colunas que não estão envolvidas no `GROUP BY`, já que não faria sentido retornar várias linhas para as colunas que não estão agrupadas.

Ou seja, o banco de dados exige que todas as colunas no `SELECT` sejam **ou** colunas usadas no `GROUP BY`, **ou** colunas que passem por uma função de agregação.

Para evitar esse erro, você deve selecionar **apenas as colunas que fazem sentido no contexto do agrupamento**. Isso significa:

1. **Incluir no `SELECT` apenas as colunas que estão no `GROUP BY`.**
2. **Usar funções de agregação para as colunas que não estão no `GROUP BY`.**

Voltando ao exemplo dado acima quando queríamos saber a quantidade de pagamentos registradas em cada banco escrevemos o seguinte comando:

```
SELECT banco, COUNT(*) AS total_transacoes FROM pagamentos GROUP BY banco;
```

- `banco` está no `GROUP BY`, então o banco de dados agrupa as linhas com base nos valores dessa coluna. Observe que essa mesma coluna é passada para retornar no `select`
- `COUNT(*)` é uma função de agregação que conta o número de linhas (transações) para cada grupo (banco).
- **Não incluímos outras colunas** (como `id_cliente` ou `tipo_pagamento`) porque elas não fazem sentido no contexto dessa pergunta. Se fossem incluídas sem uma função de agregação, o banco de dados não saberia como lidar com elas, resultando em erro.

Portanto, para evitar problemas, sempre selecione apenas as colunas que fazem sentido no contexto do agrupamento e use funções de agregação quando necessário. Isso garante que sua consulta seja clara, eficiente e livre de erros.

Como funciona o `HAVING`?

O `HAVING` é utilizado para filtrar dados depois de um agrupamento feito pelo `GROUP BY`. Por exemplo, suponha que queremos verificar **quantos pagamentos foram feitos em cada dia**, mas apenas para os dias **posteriores a 2025-03-06**.

```
select data_pagamento, count(*) from pagamentos group by data_pagamento having data_pagamento > '2025-03-06'
```

data_pagamento	count
2025-03-11	1
2025-03-13	1
2025-03-07	1
2025-03-10	1
2025-03-14	1
2025-03-12	1
2025-03-17	1
2025-03-16	1
2025-03-15	1
2025-03-18	1
2025-03-08	1

data_pagamento	count
2025-03-09	1

Qual a diferença entre `HAVING` e `WHERE`?

O `WHERE` e o `HAVING` são comandos usados para filtrar dados, mas a diferença entre eles está no momento e no tipo de dados que eles filtram:

- `WHERE`: Filtra as linhas de dados antes de qualquer agrupamento ou agregação. Ou seja, ele é usado para restringir as linhas da tabela base, filtrando os dados antes de realizar qualquer agregação.
- `HAVING`: Filtra os grupos de dados após o agrupamento ou agregação (após o `GROUP BY`). Ou seja, ele é usado para filtrar os resultados agregados ou os resultados de uma função de agregação, como `COUNT`, `SUM`, `AVG`, etc.

A principal diferença entre `WHERE` e `HAVING` é que o `WHERE` não pode filtrar valores resultantes de funções de agregação. O `WHERE` é usado para filtrar as linhas de dados antes que qualquer agregação ou agrupamento seja realizado. Se você tentar filtrar por uma função de agregação no `WHERE`, isso resultará em erro, porque o `WHERE` não tem acesso aos resultados agregados ainda.

Por outro lado, o `HAVING` é projetado para filtrar os resultados **após** o agrupamento e a agregação, permitindo que você utilize funções de agregação para realizar filtros.

```
-- ESSE COMANDO RETORNARIA ERRO:
SELECT banco, sum(valor) FROM pagamentos WHERE sum(valor) > 2000 group by banco;

-- ESSE FUNCIONARIA
SELECT banco, sum(valor) FROM pagamentos group by banco having sum(valor) > 2000;
```

Como funciona o `ORDER BY`?

O `ORDER BY` é utilizado para **ordenar os resultados de uma consulta** com base em uma ou mais colunas. Ele permite que você organize os dados de forma crescente (do menor para o maior) ou decrescente (do maior para o menor), podendo ser aplicado a números, datas e texto.

- `ASC` (ascendente): Ordena do menor para o maior (padrão, se não for especificado).
- `DESC` (descendente): Ordena do maior para o menor.

Exemplo 1: Ordenando por uma Coluna (ID do Cliente)

```
select * from pagamentos order by id_cliente;
```

id_cliente	banco	tipo_pagamento	valor	data_pagamento
101	Banco Howl	Boleto	210.30	2025-03-11
101	Banco Howl	Pix	250.00	2025-03-01
101	Banco Howl	Cartão de Crédito	75.90	2025-03-03
102	Banco Calcifer	TED	1500.00	2025-03-02
102	Banco Calcifer	Cartão de Crédito	500.00	2025-03-12
102	Banco Sophie	Pix	50.00	2025-03-04
103	Banco Howl	Pix	150.00	2025-03-14
103	Banco Sophie	Boleto	320.50	2025-03-02
103	Banco Sophie	Cartão de Crédito	199.99	2025-03-07
104	Banco Howl	Cartão de Débito	850.00	2025-03-08
...	...	...	...	...

Exemplo 2: Ordenando por Múltiplas Colunas (ID do Cliente e Data de Pagamento)

```
select * from pagamentos order by id_cliente, data_pagamento;
```

id_cliente	banco	tipo_pagamento	valor	data_pagamento
101	Banco Howl	Pix	250.00	2025-03-01
101	Banco Howl	Cartão de Crédito	75.90	2025-03-03

id_cliente	banco	tipo_pagamento	valor	data_pagamento
101	Banco Howl	Boleto	210.30	2025-03-11
102	Banco Calcifer	TED	1500.00	2025-03-02
102	Banco Sophie	Pix	50.00	2025-03-04
102	Banco Calcifer	Cartão de Crédito	500.00	2025-03-12
103	Banco Sophie	Boleto	320.50	2025-03-02
103	Banco Sophie	Cartão de Crédito	199.99	2025-03-07
103	Banco Howl	Pix	150.00	2025-03-14
104	Banco Calcifer	Cartão de Débito	680.00	2025-03-04
...	...	...	...	...

- O banco de dados **ordenou primeiro pelo** `id_cliente`, agrupando todos os pagamentos de cada cliente.
- Dentro de cada grupo de cliente, ele **ordenou os pagamentos pela** `data_pagamento`, do mais antigo para o mais recente.
- A ordem dos `id_cliente` foi mantida, ou seja, os pagamentos do cliente **101** continuam aparecendo antes dos pagamentos do cliente **102**, e assim por diante.

Exemplo 3: Ordenando pela posição das colunas

Podemos fazer a mesma ordenação passando para o comando o número da posição da coluna, ao invés de passar o nome delas.

```
select * from pagamentos order by 1, 5;
```

1	2	3	4	5
id_cliente	banco	tipo_pagamento	valor	data_pagamento
101	Banco Howl	Pix	250.00	2025-03-01
101	Banco Howl	Cartão de Crédito	75.90	2025-03-03
101	Banco Howl	Boleto	210.30	2025-03-11
102	Banco Calcifer	TED	1500.00	2025-03-02
102	Banco Sophie	Pix	50.00	2025-03-04
102	Banco Calcifer	Cartão de Crédito	500.00	2025-03-12
103	Banco Sophie	Boleto	320.50	2025-03-02
103	Banco Sophie	Cartão de Crédito	199.99	2025-03-07
103	Banco Howl	Pix	150.00	2025-03-14
104	Banco Calcifer	Cartão de Débito	680.00	2025-03-04
...	...	...	...	...

Exemplo 4: Ordenando por uma Coluna em Ordem Decrescente (valor)

```
select * from pagamentos order by 4 desc;
```

id_cliente	banco	tipo_pagamento	valor	data_pagamento
106	Banco Calcifer	TED	3000.00	2025-03-15
105	Banco Howl	TED	2300.75	2025-03-05
102	Banco Calcifer	TED	1500.00	2025-03-02
108	Banco Sophie	TED	1200.00	2025-03-10
107	Banco Howl	Cartão de Crédito	999.99	2025-03-17
104	Banco Howl	Cartão de Débito	850.00	2025-03-08
104	Banco Calcifer	Cartão de Débito	680.00	2025-03-04
105	Banco Sophie	Cartão de Débito	620.45	2025-03-13
102	Banco Calcifer	Cartão de Crédito	500.00	2025-03-12

id_cliente	banco	tipo_pagamento	valor	data_pagamento
...	...	...	...	...

Exemplo 5: Ordenando após um agrupamento

```
select id_cliente, banco, count(*) from pagamentos group by id_cliente, banco order by 1;
```

id_cliente	banco	count
101	Banco Howl	3
102	Banco Calcifer	2
102	Banco Sophie	1
103	Banco Sophie	2
103	Banco Howl	1
104	Banco Sophie	1
104	Banco Calcifer	1

Funções de Manipulação de Datas (em construção :)