

INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
01 Fundamentos de Banco de Dados	06 - Window Functions Avançadas e Views	Matheus Laureto

CONTEÚDO

1. Revisão da Aula Anterior

- **Funções Numéricas:**
 - **ROUND()** - arredondamento de valores;
 - **CEIL()** - arredondamento para cima;
 - **FLOOR()** - arredondamento para baixo;
 - **ABS()** - retorna o valor absoluto;
 - **RANDOM()** - gera números aleatórios, entre 0 e 1.
- **Funções de Data e Hora:**
 - **NOW()** - retorna data e hora atual;
 - **AGE()** - calcula a diferença entre duas datas;
 - **EXTRACT()** - extrai partes específicas de uma data (ano, mês, dia...)
 - **DATE_TRUNC()** - trunca uma data para o início de um período específico.
- **Window Function:**
 - Calculam, valor sobre um conjunto de linhas sem colapsar tudo em uma única linha;
 - Diferem do **GROUP BY**, pois mantem todas as linhas e adicionam colunas com cálculos extras.
 - Ex: Total gasto por cliente sem perder detalhes: soma por cliente com **SUM() OVER (PARTITION BY CLIENTE_ID)**

```
SELECT v.id,
       v.cliente_id,
       v.valor,
       SUM(v.valor) OVER (PARTITION BY v.cliente_id) AS total_cliente
FROM vendas v;
```

Output Messages Notifications

id [PK] integer	cliente_id integer	valor numeric (10,2)	total_cliente numeric
5	1	399.00	4899.00
1	1	4500.00	4899.00
2	2	150.00	150.00
3	3	3200.00	3200.00
4	4	200.00	200.00
6	5	120.00	120.00
7	6	250.00	250.00

2. Window Function Avançadas

- **ROW_NUMBER()**
 - Numera sequencialmente as linhas dentro de uma determinada partição.
 - Explicação ponto a ponto:
 - **ROW_NUMBER()** – é a Window Function que cria uma numeração sequencial para as linhas selecionadas;
 - **OVER()** – define a janela onde a numeração será aplicada;
 - **PARTITION BY CLIENTE_ID** – significa que a numeração vai recomeçar do 1 para cada cliente (ou seja, a cada troca de cliente, a numeração recomeça).
 - **ORDER BY DATA_VENDA** – define a ordem dentro do cliente, ou seja, qual venda vem antes ou depois.
 - Ex:

```
SELECT cliente_id, valor,  
       ROW_NUMBER() OVER (PARTITION BY cliente_id ORDER BY data_venda) AS num_venda  
FROM vendas;
```

Output Messages Notifications

Showing rows: 1 to 7 Page No: 1 of 1

cliente_id integer	valor numeric (10,2)	num_venda bigint
1	4500.00	1
1	399.00	2
2	150.00	1
3	3200.00	1
4	200.00	1
5	120.00	1
6	250.00	1








- **LAG() e LEAD()**
 - São comandos que permitem “olhar” para as linhas anteriores ou posteriores dentro de uma partição/ordenação.
- Ex:

```

SELECT
  id,
  cliente_id,
  data_venda,
  valor,
  LAG(valor) OVER (ORDER BY data_venda) AS venda_anterior,
  LEAD(valor) OVER (ORDER BY data_venda) AS proxima_venda
FROM vendas
ORDER BY data_venda;

```

Output Messages Notifications

Showing rows: 1 to 36  Page No: 1						
id [PK] integer 	cliente_id integer 	data_venda date 	valor numeric (10,2) 	venda_anterior numeric 	proxima_venda numeric 	
8	1	2023-01-09	3000.00	[null]	1899.00	
9	1	2023-01-10	1899.00	3000.00	2000.00	
10	2	2023-01-11	2000.00	1899.00	1200.00	
11	2	2023-01-12	1200.00	2000.00	1500.00	
12	3	2023-01-13	1500.00	1200.00	1700.00	
13	3	2023-01-14	1700.00	1500.00	250.00	
14	4	2023-01-15	250.00	1700.00	200.00	
15	5	2023-01-16	200.00	250.00	3000.00	
16	1	2023-01-17	3000.00	200.00	1899.00	
17	1	2023-01-18	1899.00	3000.00	2000.00	

- **RANK()** e **DENSE_RANK()**

- **RANK()** – pula posições em caso de empate;
- **DENSE_RANK()** – não pula posições;
- Explicação ponto a ponto:
 - **RANK() OVER()** cria uma posição no ranking conforme o que foi definido.
 - Quando há empate, todos os empatados ficam com a mesma posição e o próximo pula posições, ou seja, cliente 2 e cliente 9 empataram em 2º lugar, os dois ficaram com POS_RANK = 2.
 - O próximo cliente não vai ficar em 3º, mas sim em 4º, porque a posição 3 foi “pulada”.
 - **DENSE_RANK() OVER()** também cria um ranking, mas não pula posições.
 - Se dois clientes estão empatados em 2º, o próximo fica em 3º e não em 4º lugar (sem pular).
 - Cliente 2 e cliente 9 estão empatados em 2º, os dois ficam com POS_DENSE = 2, e o próximo fica com POS_DENSE = 3, sem criar buracos.

◦ Ex:

```
-- Consulta com RANK() e DENSE_RANK()
SELECT cliente_id,
       SUM(valor) AS total,
       RANK() OVER (ORDER BY SUM(valor) DESC) AS pos_rank,
       DENSE_RANK() OVER (ORDER BY SUM(valor) DESC) AS pos_dense
FROM vendas
GROUP BY cliente_id
ORDER BY total DESC;
```

Output Messages Notifications

cliente_id integer	total numeric	pos_rank bigint	pos_dense bigint
1	19596.00	1	1
3	12800.00	2	2
9	12800.00	2	2
2	9750.00	4	3
7	3200.00	5	4
4	950.00	6	5
5	720.00	7	6
6	250.00	8	7
8	200.00	9	8

- **Funções agregadas com OVER()**

- **SUM() OVER()** – Soma acumulada
- Mostra a evolução do total gasto por cliente ao longo do tempo:

```
✓ SELECT cliente_id,
        data_venda,
        valor,
        SUM(valor) OVER (PARTITION BY cliente_id ORDER BY data_venda) AS soma_acumulada
FROM vendas
where data_venda is not null
ORDER BY cliente_id, data_venda;
```

Output Messages Notifications

Showing rows: 1 to 7 Page No: 1 of 1

cliente_id integer	data_venda date	valor numeric (10,2)	soma_acumulada numeric
1	2024-10-01	4500.00	4500.00
1	2024-10-07	399.00	4899.00
2	2024-10-02	150.00	150.00
3	2024-10-05	3200.00	3200.00
4	2024-10-06	200.00	200.00
5	2024-10-08	120.00	120.00
6	2024-10-10	250.00	250.00

- **AVG() OVER()** – Média acumulada
- Calcula a média móvel das compras de cada cliente até a data de venda:

```
✓ SELECT cliente_id,
        data_venda,
        valor,
        AVG(valor) OVER (PARTITION BY cliente_id ORDER BY data_venda) AS media_acumulada
FROM vendas
where data_venda is not null
ORDER BY cliente_id, data_venda;
```

Output Messages Notifications

Showing rows: 1 to 7 Page No: 1 of 1

cliente_id integer	data_venda date	valor numeric (10,2)	media_acumulada numeric
1	2024-10-01	4500.00	4500.0000000000000000
1	2024-10-07	399.00	2449.5000000000000000
2	2024-10-02	150.00	150.0000000000000000
3	2024-10-05	3200.00	3200.0000000000000000
4	2024-10-06	200.00	200.0000000000000000
5	2024-10-08	120.00	120.0000000000000000
6	2024-10-10	250.00	250.0000000000000000

- **COUNT() OVER()** - Contagem acumulada
- Indica quantas compras o cliente já fez até aquele ponto:

```
SELECT cliente_id,
       data_venda,
       valor,
       COUNT(*) OVER (PARTITION BY cliente_id ORDER BY data_venda) AS numero_compras
FROM vendas
where data_venda is not null
ORDER BY cliente_id, data_venda;
```

Output Messages Notifications

Showing rows: 1 to 7 Page No: 1 of 1

cliente_id integer	data_venda date	valor numeric (10,2)	numero_compras bigint
1	2024-10-01	4500.00	1
1	2024-10-07	399.00	2
2	2024-10-02	150.00	1
3	2024-10-05	3200.00	1
4	2024-10-06	200.00	1
5	2024-10-08	120.00	1
6	2024-10-10	250.00	1

- **MIN()/MAX() OVER()** - Valor mínimo/máximo dentro do grupo
- Mostra a menor compra e a maior compra de cada cliente, útil para comparar com a compra atual:

```
SELECT cliente_id,
       data_venda,
       valor,
       MIN(valor) OVER (PARTITION BY cliente_id) AS menor_compra,
       MAX(valor) OVER (PARTITION BY cliente_id) AS maior_compra
FROM vendas
where data_venda is not null
ORDER BY cliente_id, data_venda;
```

Output Messages Notifications

Showing rows: 1 to 7 Page No: 1 of 1

cliente_id integer	data_venda date	valor numeric (10,2)	menor_compra numeric	maior_compra numeric
1	2024-10-01	4500.00	399.00	4500.00
1	2024-10-07	399.00	399.00	4500.00
2	2024-10-02	150.00	150.00	150.00
3	2024-10-05	3200.00	3200.00	3200.00
4	2024-10-06	200.00	200.00	200.00
5	2024-10-08	120.00	120.00	120.00
6	2024-10-10	250.00	250.00	250.00

3. VIEWS

- É uma consulta salva no banco de dados como se fosse uma tabela virtual;
- Não armazena os dados em si (exceto views materializadas), apenas a definição da query;
- Toda vez que você consulta a view, o banco executa a query original de maneira implícita.
- Exemplo de criação de view com total de vendas por cliente:

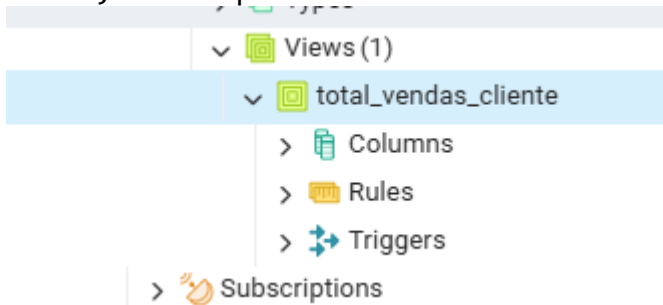
```
224
225 CREATE VIEW total_vendas_cliente AS
226 SELECT cliente_id,
227         SUM(valor) AS total_gasto,
228         COUNT(*) AS num_compras
229 FROM vendas
230 GROUP BY cliente_id;
231
```

Data Output Messages Notifications

CREATE VIEW

Query returned successfully in 160 msec.

- A partir deste momento, já temos o objeto compilado e salvo no banco de dados, sendo possível a consulta no menu lateral do pgAdmin.



- E para fazer a consulta dessa view, podemos executar o seguinte comando:

```
SELECT * FROM total_vendas_cliente;
```

Output Messages Notifications

cliente_id integer	total_gasto numeric	num_compras bigint
9	12800.00	2
3	12800.00	7
5	720.00	4
4	950.00	4
6	250.00	1
2	9750.00	7
7	3200.00	2
1	19596.00	8
8	200.00	1

5. Exercícios

1. Na tabela VENDAS, numere as vendas de cada cliente por data e traga apenas a primeira compra de cada um (ROW_NUMBER).
2. Calcule o total por cliente e mostre o ranking com RANK() e DENSE_RANK(), mostrando por ordem crescente de valor.
3. Para cada venda, mostre a venda anterior do mesmo cliente e a diferença para a atual.
4. Crie uma VIEW que traga o total de vendas por mês.
5. Consulte na VIEW criada apenas os meses com total acima de R\$1000.