

INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
01 Fundamentos de Banco de Dados	07 - Functions, Procedures, Triggers e Indices	Matheus Laureto

CONTEÚDO

1. Revisão da Aula Anterior

- **View:**
 - São consultas salvas como tabelas virtuais;
 - Não armazenam os dados executados (exceto views materializadas)
 - Sempre que consultadas, executam a query original (melhora a performance).

- Ex:

- **View “Normal”:**

```
CREATE OR REPLACE VIEW vw_total_vendas_cliente AS
SELECT cliente_id, SUM(valor) AS total
FROM vendas
GROUP BY cliente_id;

SELECT * FROM vw_total_vendas_cliente;
```

Output Messages Notifications

cliente_id integer	total numeric
8	200.00
9	14800.00
7	3200.00
1	19596.00

- Primeiro comando para criação da **VIEW**, utilização do **CREATE OR REPLACE** para criar ou substituir a view já criada.
- **SELECT**, selecionando as colunas e **FROM VW_TOTAL_VENDAS_CLIENTE**, fazendo uma consulta na view como se fosse em uma tabela normal.

- **View Materializada:**

- **Diferenças:**

- Nesse momento de execução, o resultado é salvo em disco;
- Isso deixa as consultas mais rápidas, mas os dados não atualizam automaticamente.
- Ex:

Query History

```

1 CREATE MATERIALIZED VIEW mv_total_vendas_cliente AS
2 SELECT cliente_id, SUM(valor) AS total
3 FROM vendas
4 GROUP BY cliente_id;
5
6
7 SELECT * FROM mv_total_vendas_cliente;

```

Data Output Messages Notifications

cliente_id	total
integer	numeric
8	200.00
9	14800.00
7	3200.00
1	19596.00

Total rows: 9 Query complete 00:00:00.358

- Para forçar a atualização dos dados da view materializada, devemos utilizar o comando **REFRESH**:

```

10 REFRESH MATERIALIZED VIEW mv_total_vendas_cliente;
11

```

Data Output Messages Notifications

REFRESH MATERIALIZED VIEW

Query returned successfully in 319 msec.

- **Em resumo:**

- View normal – sempre atualizada, mas executa a query todas as vezes.
- View materializada – mais rápida em consultas, mas precisa de **REFRESH**.

2. Funções:

- Funções (**FUNCTIONS**) são blocos de código armazenados no banco que recebem parâmetros, processam uma lógica previamente compilada e retornam um resultado;
- Funções servem para executar cálculos e devolver valores.
- Ex:

```
6
7 CREATE OR REPLACE FUNCTION somar_numeros(a INT, b INT)
8 RETURNS INT
9 LANGUAGE plpgsql
10 AS $$
11 BEGIN
12     RETURN a + b;
13 END;
14 $$;
15
```

Data Output Messages Notifications

CREATE FUNCTION

Query returned successfully in 228 msec.

- Chamadas:

```
19 SELECT somar_numeros(10, 5);
20 SELECT somar_numeros(7, 3);
```

Data Output Messages Notifications

	somar_numeros	
	integer	
1	10	

- As funções são executadas via **SELECT**, fazendo a chamada da **FUNCTION()** e, entre parênteses, o valor dos parâmetros (e de acordo com a quantidade de parâmetros que foram colocados na chamada da **FUNCTION**)

3. Procedures:

- São procedimentos armazenados (blocos de código SQL/PLpgSQL) salvos dentro do banco de dados que permitem executar ações complexas de forma padronizada e reutilizável.
- Servem para:
 - Automatizar processos;
 - Garantir consistência nas regras de negócio;
 - Executar múltiplas operações de uma vez;
 - Controlar transações;
 - Melhorar segurança e manutenção.

- Procedure para inserção de clientes:

```
-- Procedure simples para inserir cliente
DROP PROCEDURE IF EXISTS inserir_cliente;
✓ CREATE OR REPLACE PROCEDURE inserir_cliente(p_nome VARCHAR, p_idade INT)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Validação: não aceita idade negativa
    IF p_idade < 0 THEN
        RAISE EXCEPTION 'Idade inválida: %', p_idade;
    END IF;

    -- Inserção
    ✓ INSERT INTO clientes (nome, idade)
    VALUES (p_nome, p_idade);

    -- Confirma a transação
    COMMIT;

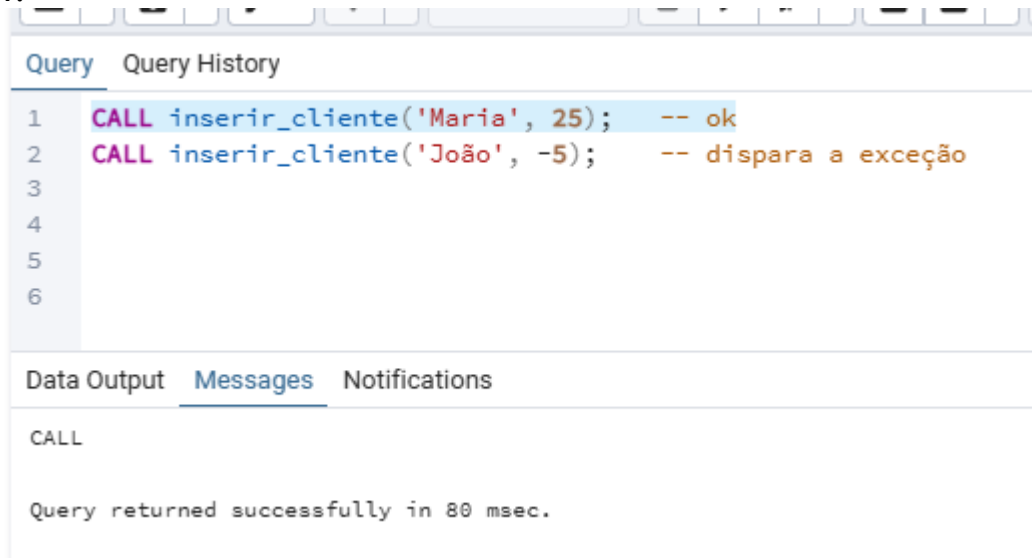
    ✓ EXCEPTION
    WHEN OTHERS THEN
        -- Em caso de erro, desfaz o que foi feito
        ROLLBACK;
        RAISE NOTICE 'Erro ao inserir cliente: %', SQLERRM;
END;
$$;
```

- Para a execução da procedure, fazemos com o comando **CALL()**:

```
CALL inserir_cliente('Maria', 25); -- ok
CALL inserir_cliente('João', -5);  -- dispara a exceção
```

- Aqui vemos as execuções da procedure:

- OK:



The screenshot shows a database query tool interface. At the top, there are tabs for 'Query' and 'Query History'. Below the tabs, a list of queries is shown with line numbers 1 through 6. The first query is 'CALL inserir_cliente('Maria', 25); -- ok' and the second is 'CALL inserir_cliente('João', -5); -- dispara a exceção'. Below the queries, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is selected, showing the text 'CALL' and 'Query returned successfully in 80 msec.'

```
Query Query History
1 CALL inserir_cliente('Maria', 25); -- ok
2 CALL inserir_cliente('João', -5); -- dispara a exceção
3
4
5
6

Data Output Messages Notifications
CALL
Query returned successfully in 80 msec.
```

- Exception:



The screenshot shows the same database query tool interface as before. The 'Messages' tab is selected, and it now displays an error message: 'NOTA: Erro ao inserir cliente: Idade inválida: -5'. Below this, the text 'CALL' and 'Query returned successfully in 138 msec.' are visible.

```
Query Query History
1 CALL inserir_cliente('Maria', 25); -- ok
2 CALL inserir_cliente('João', -5); -- dispara a exceção
3
4
5
6

Data Output Messages Notifications
NOTA: Erro ao inserir cliente: Idade inválida: -5
CALL
Query returned successfully in 138 msec.
```

- Resultado da execução da procedure:

```
CALL inserir_cliente('Matheus', 36);  
select * from clientes
```

Output Messages Notifications

SQL					
id [PK] integer	nome character varying (100)	idade integer	cidade character varying (100)	estado character varying (2)	
27	Ana	28	[null]	[null]	
28	Ana	28	[null]	[null]	
29	Ana	28	[null]	[null]	
30	Ana	28	[null]	[null]	
41	Maria	25	[null]	[null]	
42	Maria	25	[null]	[null]	
43	Maria	25	[null]	[null]	
44	Maria	25	[null]	[null]	
45	Maria	25	[null]	[null]	
46	Matheus	36	[null]	[null]	
Total rows: 27 Query complete 00:00:00.349					

4. Triggers:

- O que são:

- Um **TRIGGER** (gatilho) é um mecanismo que executa automaticamente uma função quando ocorre algum evento específico em uma tabela ou view.
- **TRIGGERS** trabalham com funções que retornam **TRIGGER** (padrão linguagem PostgreSQL), não tem a mesma característica das funções que vimos anteriormente.
- Ex:

```
36 CREATE OR REPLACE FUNCTION fn_log_insercao_clientes()  
37 RETURNS TRIGGER  
38 LANGUAGE plpgsql  
39 AS $$  
40 BEGIN  
41     INSERT INTO log_clientes (cliente_id, nome)  
42     VALUES (NEW.id, NEW.nome);  
43  
44     RETURN NEW;  
45 END;  
46 $$;  
47
```

Data Output Messages Notifications

CREATE FUNCTION

- Esse tipo de função especial sempre retorna um gatilho, por isso essa função só pode ser ligada a **TRIGGERS**;
- Atenção para a informação “**NEW.**”: esse é um comando especial que está buscando e posicionando em memória o novo registro que está sendo inserido no banco.
- Importante: Sempre colocar **RETURN NEW** no final da função, porque esse comando “devolve” a nova linha para que a operação original (insert into log_clientes...) continue normalmente.
Se colocar **RETURN NULL**, a inserção é cancelada.

- Criando a TRIGGER e vinculando a FUNCTION:

```

50
51 CREATE TRIGGER trg_log_insercao_clientes
52 AFTER INSERT ON clientes
53 FOR EACH ROW
54 EXECUTE FUNCTION fn_log_insercao_clientes();
55

```

Data Output Messages Notifications

CREATE TRIGGER

Query returned successfully in 94 msec.

- Aqui criamos de fato o gatilho, onde definimos como a TRIGGER vai funcionar (em qual momento), onde vai funcionar (a qual tabela estará vinculada) e o que deve ser feito (código da **FUNCTION TRIGGER**):
 - **AFTER INSERT** - o gatilho roda depois que o cliente for inserido na tabela **CLIENTES**;
 - Se fosse **BEFORE**, o gatilho rodaria antes da inserção (permitindo validar ou até impedir o **INSERT**).
 - **ON CLIENTES** - define a qual tabela esse gatilho está vinculado;
 - **FOR EACH ROW** - indica que o gatilho vai rodar para cada linha inserida, ou seja, se o **INSERT** conta com 10 linhas, o gatilho será disparado 10 vezes.
Se for **FOR EACH STATEMENT**, roda somente uma vez por comando **INSERT**.
 - **EXECUTE FUNCTION()** - informa qual função será chamada toda vez que o gatilho disparar.

- Executando o gatilho:

```

57
58 INSERT INTO clientes (nome, idade) VALUES ('Ana', 30);
59

```

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 245 msec.

- Nesse momento, é inserido um registro na tabela **CLIENTES**;
- Como o gatilho está definido AFTER INSERT, ele dispara logo em seguida;
- O **TRIGGER** chama a função **FN_LOG_INSERCAO_CLIENTES()**;
- A função busca **NEW.ID** e **NEW.NOME** do registro original inserido e grava em **LOG_CLIENTES**.
- O comando **RETURN NEW** garante que a operação continue e finalize normalmente.

- Resultado:

`SELECT * FROM log_clientes`

ta Output Messages Notifications

id [PK] integer	cliente_id integer	nome text	data_log timestamp without time zone
1	47	Ana	2025-08-25 18:00:24.653983

5. Índices:

- O que é um índice?

- Um índice é como o sumário de um livro: acelera a busca de informações:

Sem índice - o banco precisa varrer a tabela inteira (full table scan).

Com índice - o banco localiza rapidamente os registros na árvore de busca (B-tree, Hash, etc.).

- Prós e Contras:

- Índices deixam nossas consultas muito mais rápidas (desde que feita a consulta de maneira apropriada utilizando o índice);
- Porém, a cada índice criado na tabela, ocupa maior espaço em disco e deixa todo comando de **INSERT/UPDATE/DELETE** cerca de 30% mais lento.

- Tipos de Índice:

- Índice de consulta simples:

```
4
5 CREATE INDEX idx_vendas_cliente ON vendas(cliente_id);
6 SELECT *
7 FROM vendas
8 WHERE cliente_id = 1;
9
```

Data Output Messages Notifications

	id [PK] integer	cliente_id integer	valor numeric (10,2)	data_venda date
	1	1	4500.00	2024-10-01
	5	1	399.00	2024-10-07
	8	1	3000.00	2023-01-09
	16	1	3000.00	2023-01-17
	17	1	1899.00	2023-01-18
	24	1	3000.00	2023-01-25
	25	1	1899.00	2023-01-26
	9	1	1899.00	2023-01-09

- Toda vez que executar essa consulta e filtrar por CLIENTE_ID, o banco usará o índice ao invés de percorrer a tabela inteira.

- Índice Único:

```
71
72 CREATE UNIQUE INDEX idx_clientes_email ON clientes(email);
73
```

Data Output Messages Notifications

CREATE INDEX

Query returned successfully in 94 msec.

- Garante que, os registros sejam distintos, não permitindo registros duplicados na coluna.

- Índices Compostos:

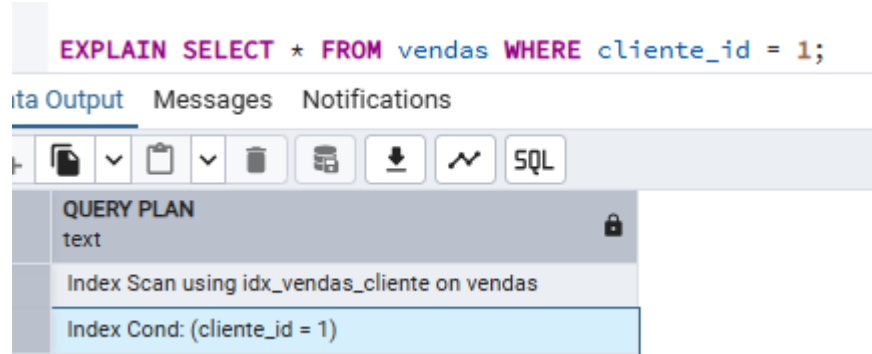
```
5
6
7 CREATE INDEX idx_vendas_cliente_data
8 ON vendas(cliente_id, data_venda);
9
10 SELECT * FROM vendas
11 WHERE cliente_id = 1 AND data_venda >= '2000-01-01';
12
```

Data Output Messages Notifications

	id [PK] integer	cliente_id integer	valor numeric (10,2)	data_venda date
	1	1	4500.00	2024-10-01
	5	1	399.00	2024-10-07
	8	1	3000.00	2023-01-09
	16	1	3000.00	2023-01-17
	17	1	1899.00	2023-01-18
	24	1	3000.00	2023-01-25
	25	1	1899.00	2023-01-26
	9	1	1899.00	2023-01-09

- Dessa forma, o índice criado só vai ser utilizado caso seja informado no **WHERE** todos os campos e na sua ordem no índice.
- Caso informe somente **CLIENTE_ID** ou **DATA_VENDA**, o índice não será utilizado.

- Para verificar se o índice está sendo utilizado:



- O comando **EXPLAIN** mostra o plano de execução do comando posterior, não executando a query, somente mostrando quais índices serão utilizados.
- Obs: nem sempre o índice será utilizado, isso pode acontecer em caso de tabelas pequenas, onde o banco decide que fazer a leitura completa (FULL SCAM) possa ser mais rápida que o índice.