

# Banco de Dados 2

**05 – Stored Procedure e Triggers**

# Stored Procedures

- No PostgreSQL, um **procedimento armazenado** (*Stored Procedure*) é um conjunto pré-compilado de instruções SQL e estruturas de controle armazenadas no servidor de banco de dados.
- Ao contrário das **funções**, que são projetadas para retornar um valor, os **procedimentos armazenados** são usados principalmente para executar operações e gerenciar transações sem necessariamente retornar um resultado.
- Tanto **Stored Procedures** quanto **funções (functions)** em PL/pgSQL no **PostgreSQL** permitem encapsular lógica de negócio diretamente no banco de dados.
- Porém, há diferenças importantes entre eles, e cada uma é mais vantajosa em contextos específicos.

# Stored Procedures

- Características de um **Stored Procedure**:

- **Encapsulamento de Lógica:**

- Permitem a consolidação de lógica de negócios complexa diretamente no banco de dados, promovendo a reutilização e a manutenibilidade.

- **Gerenciamento de transações:**

- Os procedimentos podem incluir instruções de controle de transações (por exemplo, COMMIT, ROLLBACK), permitindo operações atômicas envolvendo múltiplas instruções SQL.

- **Otimização de desempenho:**

- Ao pré-compilar o código, os procedimentos armazenados podem oferecer benefícios de desempenho, especialmente para operações complexas e executadas com frequência, pois as etapas de análise e otimização são executadas uma vez.

- **Segurança:**

- Eles podem aumentar a segurança permitindo que os usuários executem tarefas específicas sem conceder acesso direto às tabelas subjacentes.

# Stored Procedures

- **Idiomas suportados:**
  - PostgreSQL oferece suporte a várias linguagens procedurais para escrever procedimentos armazenados, incluindo PL/pgSQL (o padrão), SQL, C e outras linguagens por meio de extensões como Perl, Python e TCL.
- **Criando um Procedimento Armazenado:**
  - Procedimentos armazenados são criados usando a instrução **CREATE PROCEDURE**.

```
CREATE PROCEDURE procedure_name(parameter1 datatype, parameter2 datatype, ...)  
LANGUAGE plpgsql -- or other supported language  
AS $$  
BEGIN  
    -- SQL statements and control logic  
END;  
$$;
```

# Stored Procedures

Critério	Função (Function)	Procedure (Stored Procedure)
Retorna valores	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não (a não ser por OUT params)
Pode ser usada em SELECT	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não
Controle interno de transações	<input type="checkbox"/> Não	<input checked="" type="checkbox"/> Sim (COMMIT, ROLLBACK, etc.)
Pode ser chamada com <code>CALL</code>	<input type="checkbox"/> Não	<input checked="" type="checkbox"/> Sim
Indicada para lógica de negócio	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim, especialmente com transações
Integração com aplicações	<input checked="" type="checkbox"/> Boa	<input type="checkbox"/> Limitada (nem todos drivers suportam <code>CALL</code> )

# Stored Procedures

- Quando uma função PL/pgSQL é mais apropriada:
  - 1. Retorno de valores ou conjuntos de dados:
    - Funções são ideais quando você precisa:
    - Retornar **valores escalares, registros ou tabelas**.
    - Usar em consultas SQL diretamente (SELECT minhaFuncao( ... )).
  - 2. Funções imutáveis ou determinísticas:
    - Funções podem ser marcadas como **IMMUTABLE**, **STABLE** ou **VOLATILE**, o que permite **otimizações pelo planejador de consultas**.
    - Isso não é possível com procedures.
  - 3. Facilidade de integração com ORMs (**Mapeamento Objeto-Relacional**) e linguagens externas:
    - Funções são **mais amigáveis** a aplicações externas que usam bibliotecas de acesso ao banco.

# Um Parênteses sobre Funções

- **IMMUTABLE (imutável):**

- A função **sempre retorna o mesmo resultado** para os mesmos parâmetros, **independente de qualquer dado do banco ou do ambiente.**
- Exemplo:

```
CREATE FUNCTION soma(a int, b int) RETURNS int  
AS $$ BEGIN RETURN a + b; END $$ LANGUAGE plpgsql IMMUTABLE;
```

# Um Parênteses sobre Funções

- **STABLE (Estável):**

- A função retorna o mesmo resultado dentro de uma mesma execução da consulta, mas pode variar entre execuções diferentes se acessar dados do banco.
- **Exemplo:**

```
CREATE FUNCTION quantidade_clientes() RETURNS int  
AS $$ SELECT COUNT(*) FROM cliente; $$ LANGUAGE sql STABLE;
```

# Um Parênteses sobre Funções

- **VOLATILE (Volátil):**

- A função pode retornar valores diferentes mesmo com os mesmos parâmetros. Pode depender do tempo, de variáveis externas, ou modificar dados do banco.
- **Exemplo:**

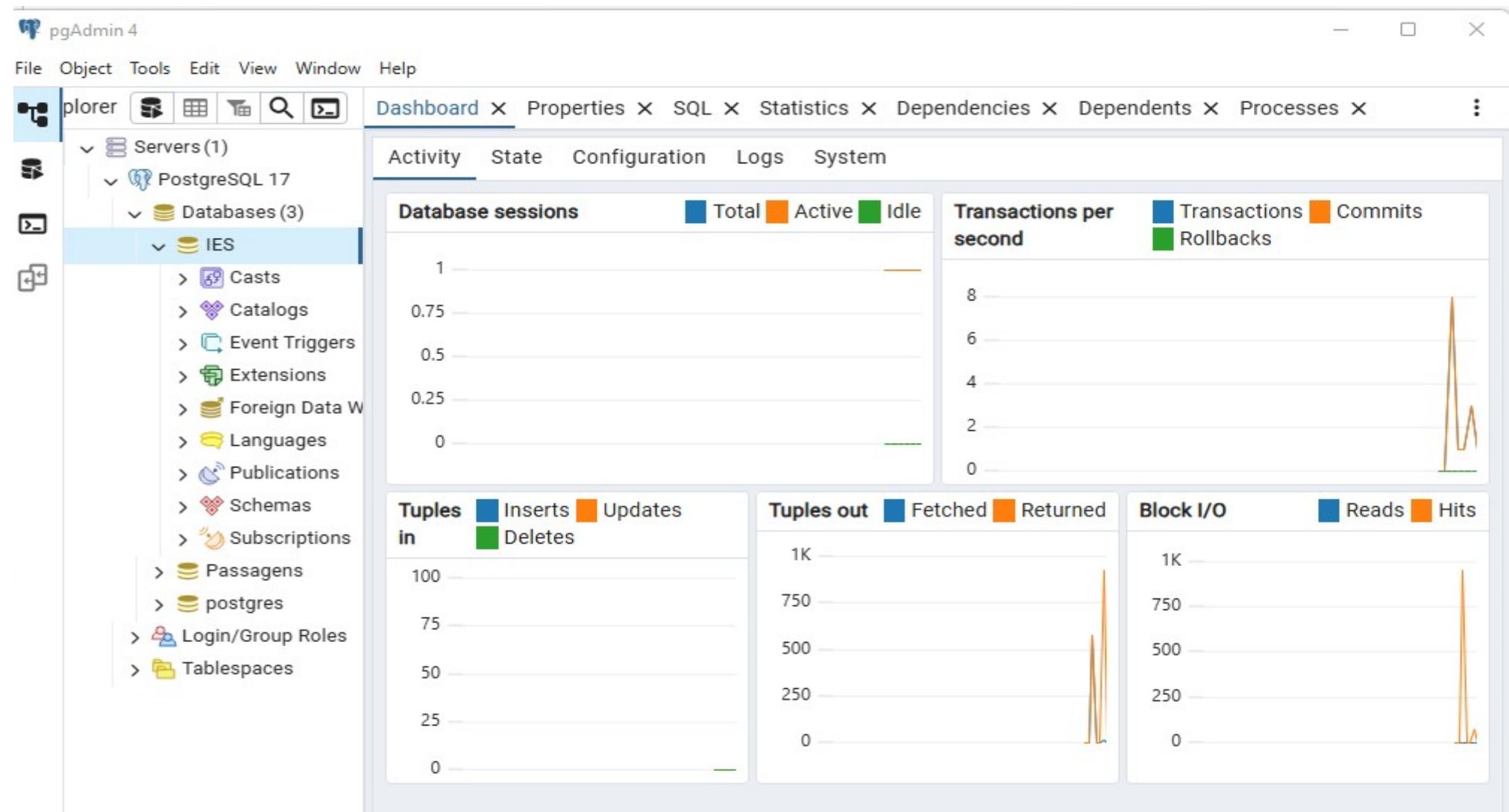
```
CREATE FUNCTION agora() RETURNS timestamp  
AS $$ SELECT now(); $$ LANGUAGE sql VOLATILE;
```

```
CREATE FUNCTION registrar_log(msg text) RETURNS void  
AS $$  
BEGIN  
    INSERT INTO log_sistema (mensagem, data_hora) VALUES (msg, now());  
END;  
$$ LANGUAGE plpgsql VOLATILE;
```

# Um Parênteses sobre Funções

Categoria	Retorna mesmo valor sempre?	Pode ler dados?	Pode modificar dados?
IMMUTABLE	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não	<input type="checkbox"/> Não
STABLE	<input type="checkbox"/> Sim, durante a query	<input checked="" type="checkbox"/> Sim	<input type="checkbox"/> Não
VOLATILE	<input type="checkbox"/> Não	<input checked="" type="checkbox"/> Sim	<input checked="" type="checkbox"/> Sim

# Um Parênteses sobre Funções





## Object Explorer



## Servers (1)

## PostgreSQL 17

## Databases (3)

## IES

- > Casts
- > Catalogs
- > Event Triggers
- > Extensions
- > Foreign Data Wrappers
- > Languages
- > Publications
- > Schemas (1)

## public

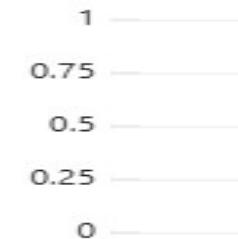
- > Aggregates
- > Collations
- > Domains
- > FTS Configurations
- > FTS Dictionaries
- > FTS Parsers
- > FTS Templates
- > Foreign Tables
- > Functions
- > Materialized Views
- > Operators
- > Procedures
- > Sequences
- > Tables
- > Trigger Functions
- > Types
- > Views



## Dashboard X

## Activity

## Database set



## Tuples

## Ins in De



▼ Schemas (1)

  ▼ public

    > Aggregates

    > Collations

    > Domains

    > FTS Configurations

    > FTS Dictionaries

    > FTS Parsers

    > FTS Templates

    > Foreign Tables

    ▼ Functions (11)

       apagadisciplina(id integer)

       buscar\_disciplinas\_ofertadas(p\_ano integer, p\_semestre integer)

       dividir(val1 numeric, val2 numeric, OUT quociente numeric, OUT resto numeric)

       exibegrade(curso\_id integer, grade integer)

       get\_film\_count(len\_from integer, len\_to integer)

       incrementar(integer)

       insereprofessor(integer, character varying)

       inserir\_disciplina\_ofertada(p\_id\_disciplina\_ofertada integer, p\_ano integer, p\_semestre integer, p\_vagas integer, p\_local character varying)

       listar\_alunos\_por\_curso(p\_curso\_id integer)

       obter\_alunos\_cursor()

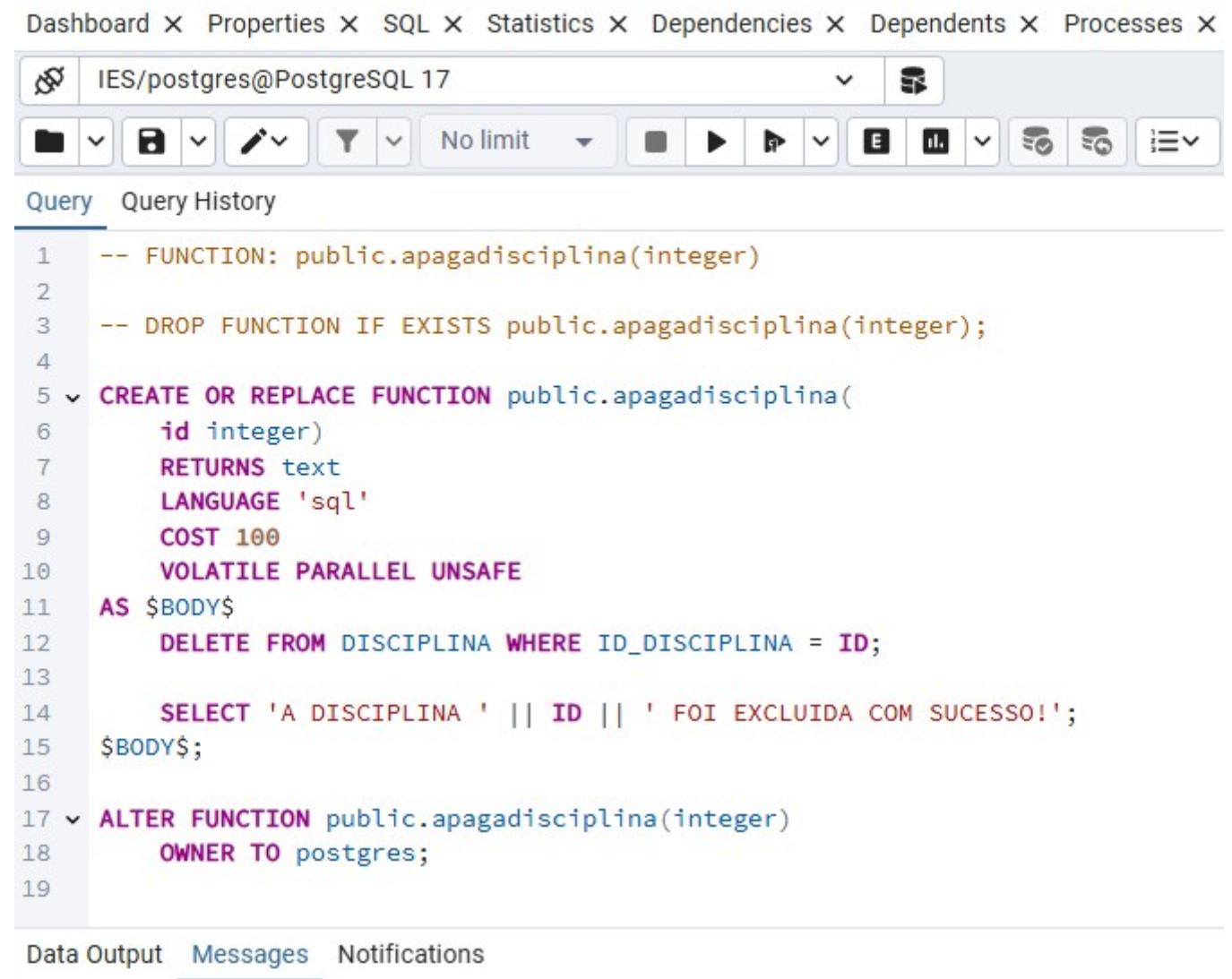
       totaldisciplinas(integer)

    > Materialized Views

    > Operators

    > Procedures

- Observe que criamos a função **apagadisciplina( )** em um momento anterior e não especificamos que a mesma deveria ser **VOLATILE**.
- O **PostgreSQL** determinou automaticamente que essa função deveria ser **classificada** como **VOLATILE**.



The screenshot shows the pgAdmin 4 interface with the SQL tab selected. The query window displays the SQL code for creating and altering the `apagadisciplina` function. The code is as follows:

```

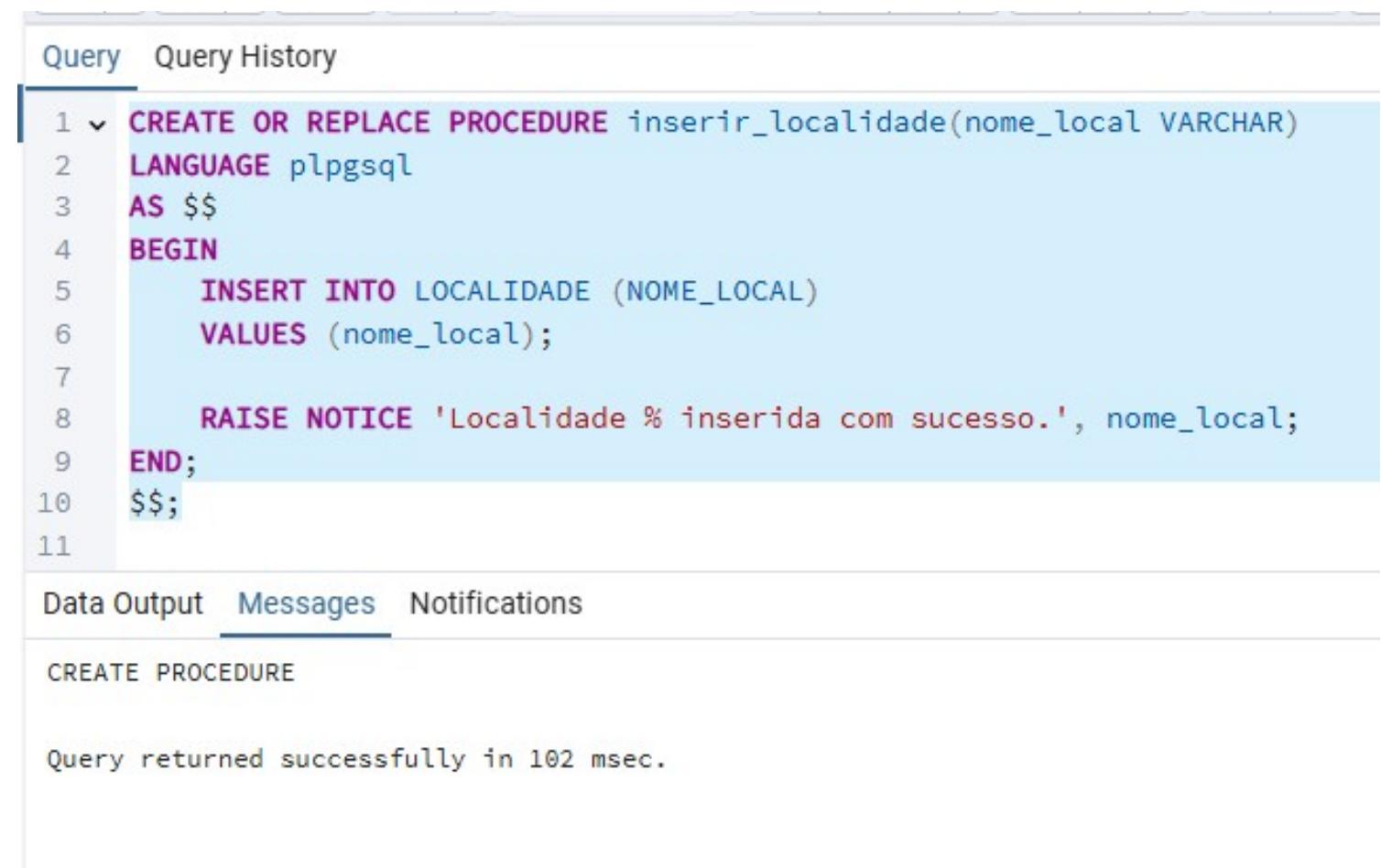
1 -- FUNCTION: public.apagadisciplina(integer)
2
3 -- DROP FUNCTION IF EXISTS public.apagadisciplina(integer);
4
5 CREATE OR REPLACE FUNCTION public.apagadisciplina(
6     id integer)
7 RETURNS text
8 LANGUAGE 'sql'
9 COST 100
10 VOLATILE PARALLEL UNSAFE
11 AS $BODY$
12     DELETE FROM DISCIPLINA WHERE ID_DISCIPLINA = ID;
13
14     SELECT 'A DISCIPLINA ' || ID || ' FOI EXCLUIDA COM SUCESSO!';
15 $BODY$;
16
17 ALTER FUNCTION public.apagadisciplina(integer)
18     OWNER TO postgres;
19

```

The interface includes a toolbar with various icons for database management, a dropdown menu for the connection, and tabs for Data Output, Messages, and Notifications.

# Regressando ao **Stored Procedure**

- Um exemplo simplório de um **Stored Procedure (SP)** para inserir registros na tabela LOCALIDADE.
- Poderíamos ter usado de uma função SQL para o mesmo objetivo, sem qualquer prejuízo.



The screenshot shows a PostgreSQL query editor interface. The top navigation bar has tabs for "Query" and "Query History". The main area contains a code editor with numbered lines (1 to 11) displaying the SQL code for creating a stored procedure. The code uses the plpgsql language and includes logic to insert data into the LOCALIDADE table and raise a notice message. Below the code editor, there are tabs for "Data Output", "Messages", and "Notifications", with "Messages" being the active tab. A status message at the bottom indicates the query was executed successfully.

```
1 CREATE OR REPLACE PROCEDURE inserir_localidade(nome_local VARCHAR)
2 LANGUAGE plpgsql
3 AS $$*
4 BEGIN
5     INSERT INTO LOCALIDADE (NOME_LOCAL)
6     VALUES (nome_local);
7
8     RAISE NOTICE 'Localidade % inserida com sucesso.', nome_local;
9 END;
10 $$;
11
```

Data Output Messages Notifications

CREATE PROCEDURE

Query returned successfully in 102 msec.

# Regressando ao Stored Procedure

- Descobrimos o nome da variável que mantém o último valor da coluna ID\_LOCAL (SERIAL):
- **LOCALIDADE\_ID\_LOCAL\_SEQ.**

The screenshot shows a PostgreSQL pgAdmin interface. At the top, there is a code editor window with the following SQL query:

```
11
12 SELECT pg_get_serial_sequence('localidade', 'id_local');
13
```

Below the code editor are three tabs: "Data Output", "Messages", and "Notifications". The "Data Output" tab is selected. A toolbar with various icons is located below the tabs. The results pane displays a single row of data:

	pg_get_serial_sequence	
1	text	public.localidade_id_local_seq

# Regressando ao Stored Procedure

- Atualizamos o valor desta variável usando a função `setval()`.
- Essa função, `setval()`, não é uma função definida pelo usuário (programador).
- Como outras, ela já vem com a instalação do PostgreSQL.

The screenshot shows a PostgreSQL client interface with a code editor and a results pane. The code editor contains the following PL/pgSQL block:

```
11
12 SELECT pg_get_serial_sequence('localidade', 'id_local');
13
14 SELECT setval('localidade_id_local_seq', (SELECT MAX(id_local) FROM localidade));
15
```

The results pane shows the output of the last query, which is a table with one row:

	setval	bigint
1	37	

The interface includes tabs for Data Output, Messages, and Notifications, and a toolbar with various icons.

# Regressando ao Stored Procedure

- Executando o nosso SP.

```
11
12  SELECT pg_get_serial_sequence('localidade', 'id_local');
13
14  SELECT setval('localidade_id_local_seq', (SELECT MAX(id_local) FROM localidade));
15
16  CALL inserir_localidade('CONCEICAO DA BARRA');
17
```

Data Output Messages Notifications

NOTA: Localidade CONCEICAO DA BARRA inserida com sucesso.

CALL

Query returned successfully in 84 msec.

# Regressando ao Stored Procedure

Query    Query History

```
15
16  CALL inserir_localidade('CONCEICAO DA BARRA');
17
18  SELECT * FROM LOCALIDADE;
19
```

Data Output    Messages    Notifications

	id_local [PK] integer	nome_local character varying (40)
14	14	JOAO NEIVA
15	15	ARACRUZ
16	16	COLATINA
17	17	BAUNILHA
18	18	PENDANGA
19	19	MARILANDIA
20	20	BAIXO GUANDU
21	21	LINHARES
22	22	SANTA TERESA
23	23	SAO MATEUS
24	24	JAGUARE
25	25	RIO BANANAL
26	26	SOORETAMA
27	27	ACIOLI
28	28	CAVALINHO
29	29	BOAPABA
30	30	ITAPINA
31	31	NOVA ALMEIDA
32	32	PRAIA GRANDE
33	33	TIMBUI
34	34	BARRA DO TRIUNFO
35	35	SAO ROQUE DO CANAA
36	36	CARAPINA
37	37	MUNIZ FREIRE
38	38	CONCEICAO DA BARRA

# Stored Procedure

```
19
20 ✓ SELECT ID_LINHA, ID_TRECHO, T.ID_LOCAL, NOME_LOCAL, KM
21   FROM TRECHO T, LOCALIDADE L
22 WHERE T.ID_LOCAL = L.ID_LOCAL;
23
```

## Data Output    Messages    Notifications

The screenshot shows the Microsoft Power BI ribbon at the top of a software window. The 'Home' tab is highlighted in blue, indicating it is the active tab. Other tabs visible include 'Get Data', 'Transform Data', 'Home', 'Visualizations', 'Format', 'Report', and 'Help'. Each tab has a corresponding icon to its left.

	<b>id_linha</b> integer	<b>id_trecho</b> integer	<b>id_local</b> integer	<b>nome_local</b> character varying (40)	<b>km</b> integer
1	100	1	36	CARAPINA	18
2	100	2	11	SERRA	28
3	100	3	12	FUNDAO	63
4	100	4	18	PENDANGA	71
5	100	5	13	IBIRACU	75
6	100	6	14	JOAO NEIVA	81
7	100	7	28	CAVALINHO	85
8	100	8	27	ACIOLI	104
9	100	9	17	BAUNILHA	121
10	100	10	16	COLATINA	135

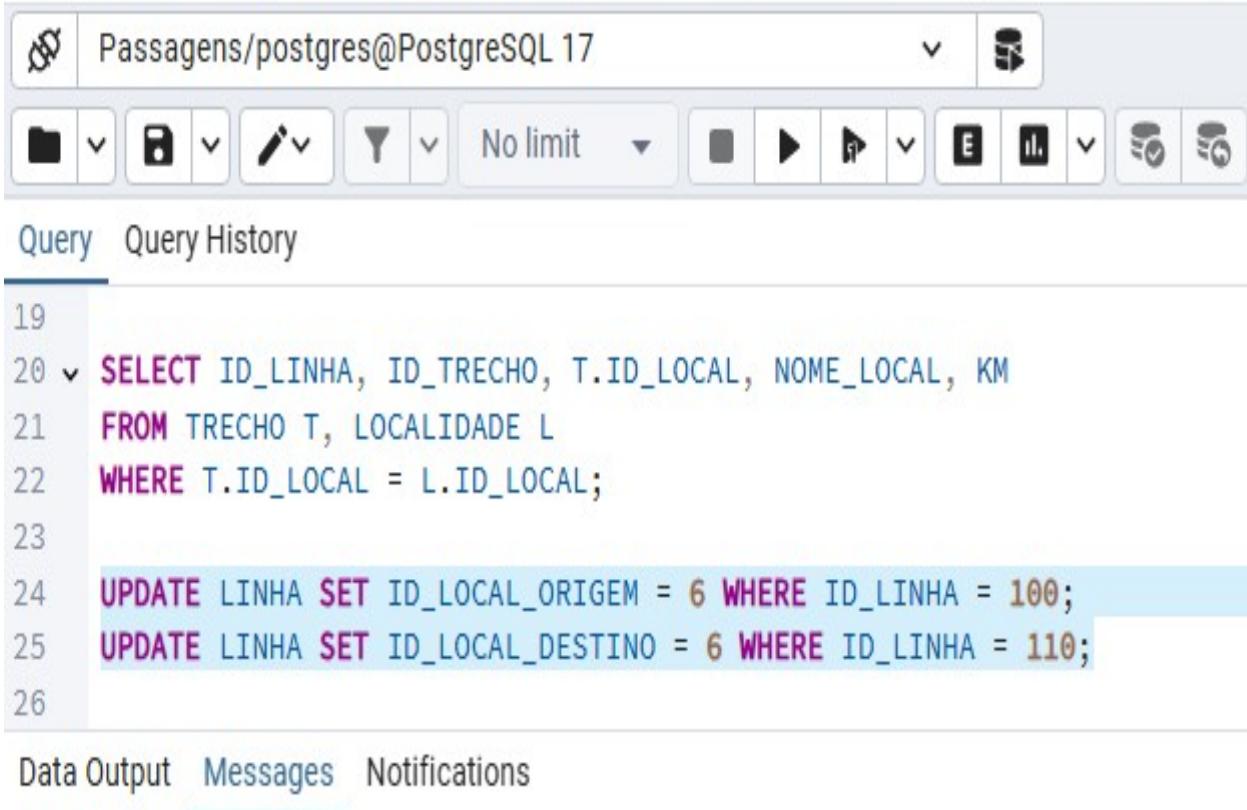
# Stored Procedure

Correção de 2 registros da tabela LINHA.

As LINHAS 100 e 110 estavam como “São Roque de Canaã X Colatina” e “Colatina X São Roque de Canaã”, respectivamente.

Contudo, os TRECHOS descritos para ambas seriam os das LINHAS “Vitória X Colatina” e “Colatina X Vitória”.

Então substituímos São Roque por Vitória.



The screenshot shows a PostgreSQL database client interface. At the top, there's a connection bar with the text "Passagens/postgres@PostgreSQL 17". Below it is a toolbar with various icons for file operations, search, and navigation. The main area is divided into two tabs: "Query" (which is selected) and "Query History". The "Query" tab contains the following SQL code:

```
19
20 v SELECT ID_LINHA, ID_TRECHO, T.ID_LOCAL, NOME_LOCAL, KM
21   FROM TRECHO T, LOCALIDADE L
22 WHERE T.ID_LOCAL = L.ID_LOCAL;
23
24 UPDATE LINHA SET ID_LOCAL_ORIGEM = 6 WHERE ID_LINHA = 100;
25 UPDATE LINHA SET ID_LOCAL_DESTINO = 6 WHERE ID_LINHA = 110;
26
```

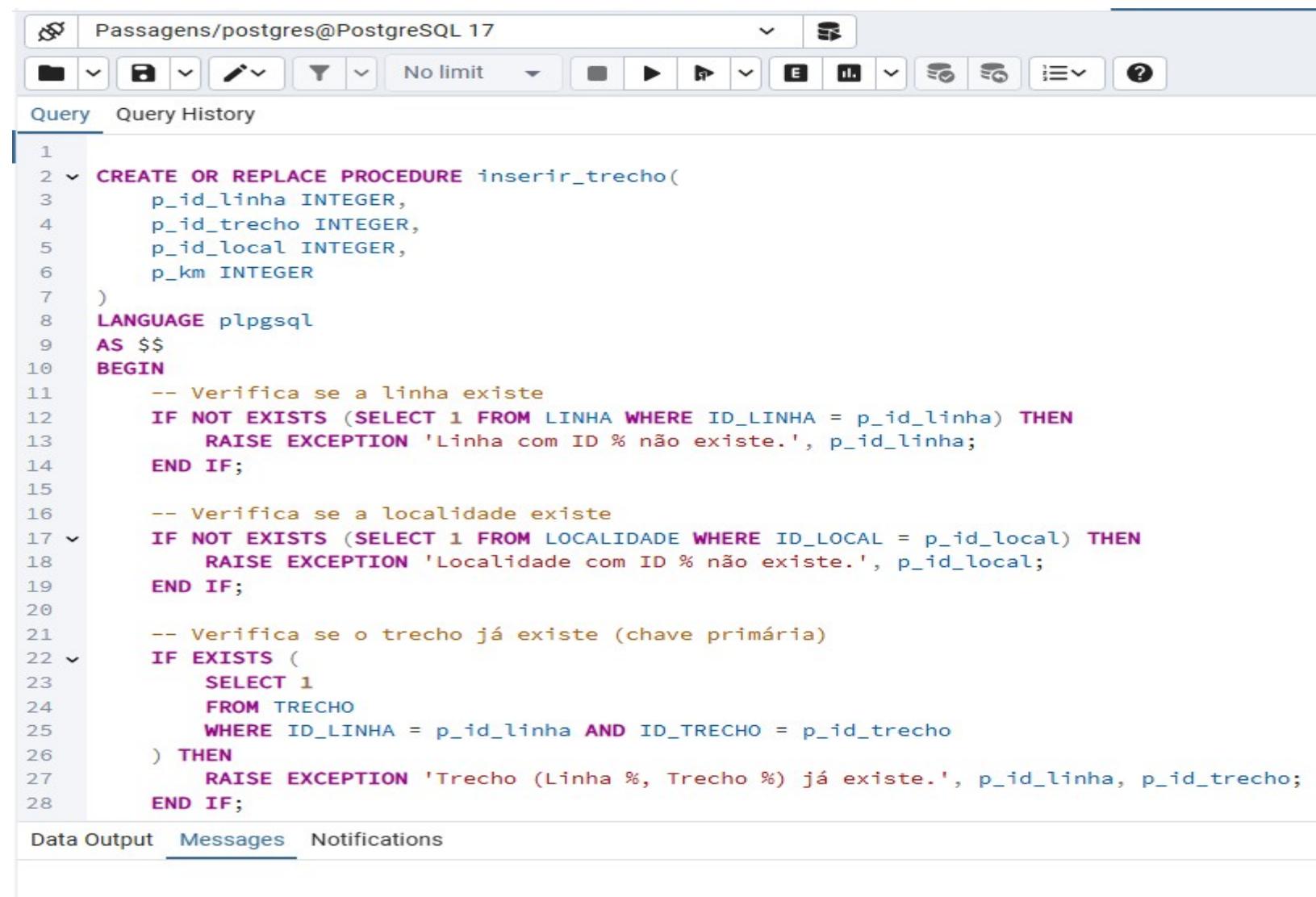
Below the code, there are tabs for "Data Output", "Messages" (which is selected), and "Notifications". The "Messages" tab displays the output of the last update command:

UPDATE 1

At the bottom, a message states: "Query returned successfully in 99 msec."

# Stored Procedure

- Agora criaremos um **SP** para inserir um novo registro de **TRECHO**.



The screenshot shows the pgAdmin 4 interface with a query editor window. The title bar reads "Passagens/postgres@PostgreSQL 17". The toolbar has various icons for database management. The main area is a "Query" tab with the following PostgreSQL code:

```
1
2  CREATE OR REPLACE PROCEDURE inserir_trecho(
3      p_id_linha INTEGER,
4      p_id_trecho INTEGER,
5      p_id_local INTEGER,
6      p_km INTEGER
7  )
8  LANGUAGE plpgsql
9  AS $$
10 BEGIN
11     -- Verifica se a linha existe
12     IF NOT EXISTS (SELECT 1 FROM LINHA WHERE ID_LINHA = p_id_linha) THEN
13         RAISE EXCEPTION 'Linha com ID % não existe.', p_id_linha;
14     END IF;
15
16     -- Verifica se a localidade existe
17     IF NOT EXISTS (SELECT 1 FROM LOCALIDADE WHERE ID_LOCAL = p_id_local) THEN
18         RAISE EXCEPTION 'Localidade com ID % não existe.', p_id_local;
19     END IF;
20
21     -- Verifica se o trecho já existe (chave primária)
22     IF EXISTS (
23         SELECT 1
24         FROM TRECHO
25         WHERE ID_LINHA = p_id_linha AND ID_TRECHO = p_id_trecho
26     ) THEN
27         RAISE EXCEPTION 'Trecho (Linha %, Trecho %) já existe.', p_id_linha, p_id_trecho;
28     END IF;
```

Below the code, there are tabs for "Data Output", "Messages", and "Notifications".

# Stored Procedure

```
-- 
29
30      -- Verifica se o KM é válido
31  IF p_km <= 0 THEN
32      RAISE EXCEPTION 'Valor de KM inválido: %. Deve ser maior que zero.', p_km;
33  END IF;
34
35      -- Se tudo OK, insere o trecho
36  INSERT INTO TRECHO (ID_LINHA, ID_TRECHO, ID_LOCAL, KM)
37  VALUES (p_id_linha, p_id_trecho, p_id_local, p_km);
38
39      RAISE NOTICE 'Trecho (Linha %, Trecho %) inserido com sucesso.', p_id_linha, p_id_trecho;
40
41 END;
42 $$;
43
```

Data Output Messages Notifications

# Stored Procedure

- Chamada do SP.
- Linha “Colatina X Vitória”  
(ID\_LINHA = 110).
- Trecho de COLATINA até BAUNILHA (20 km).

```
38
39      RAISE NOTICE 'Trecho (Linha %, Trecho %) inserido com sucesso.', p_id_linha, p_id_trecho;
40
41  END;
42  $$;
43
44  CALL inserir_trecho(110, 1, 17, 20);
45
```

Data Output Messages Notifications

NOTA: Trecho (Linha 110, Trecho 1) inserido com sucesso.

CALL

Query returned successfully in 104 msec.

# Stored Procedure

```
38
39      RAISE NOTICE 'Trecho (Linha %, Trecho %) inserido com sucesso.', p_id_linha, p_id_trecho;
40
41 END;
42 $$;
43
44 CALL inserir_trecho(110, 1, 17, 20);
45
46 SELECT * FROM TRECHO WHERE ID_LINHA = 110;
```

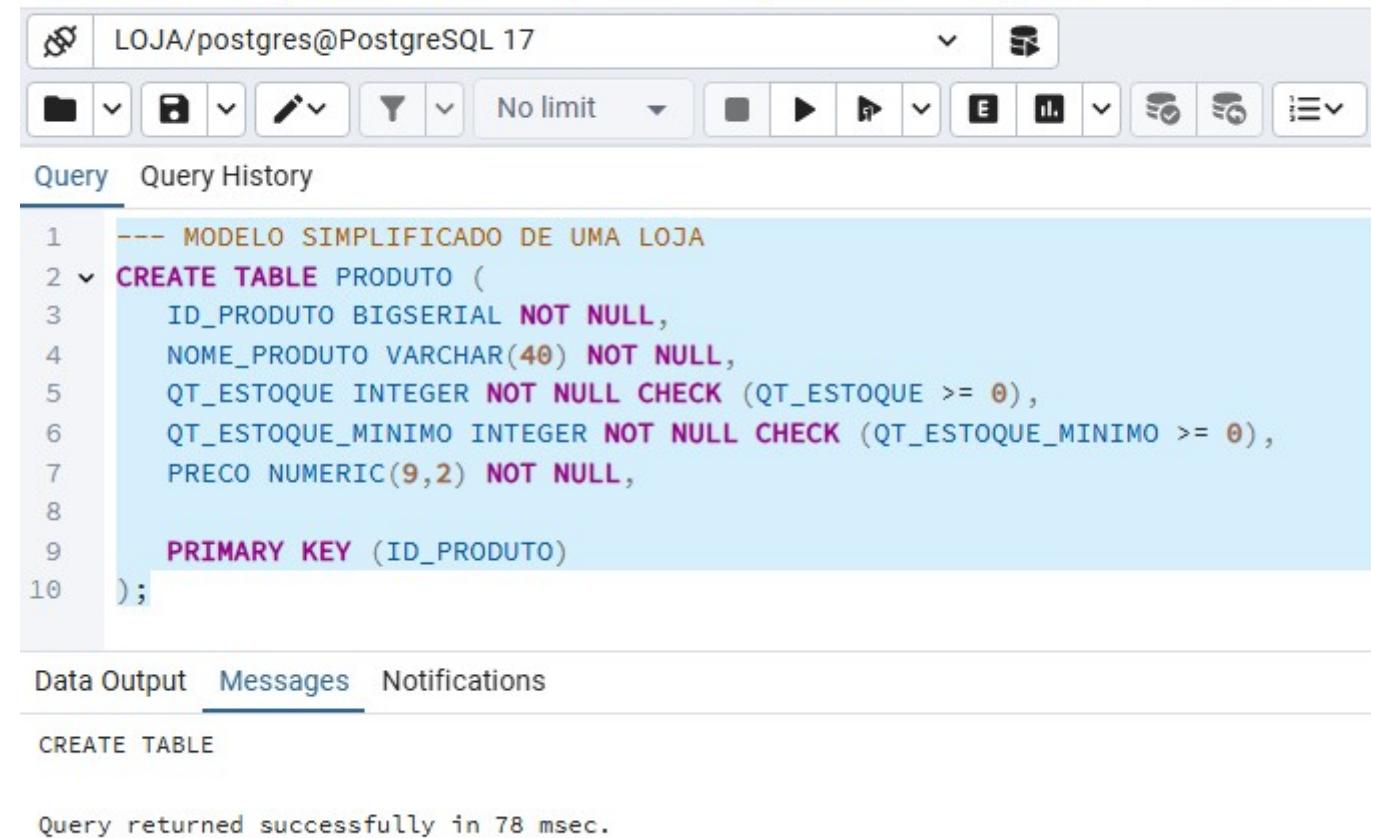
Data Output Messages Notifications

The screenshot shows a pgAdmin interface with a query editor and a results pane. The query editor contains a stored procedure definition and a call to it, followed by a select statement. The results pane displays the output of the select statement, showing a single row with columns: id\_linha, id\_trecho, id\_local, and km.

	id_linha [PK] integer	id_trecho [PK] integer	id_local integer	km integer
1	110	1	17	20

# NOVO BANCO DE DADOS

- Para recriar um exemplo clássico de utilização de **Stored Procedures** e **Triggers** definiremos um novo **banco de dados**: **LOJA**.



The screenshot shows a PostgreSQL client window titled "LOJA/postgres@PostgreSQL 17". The main area displays the following SQL code:

```
1 --- MODELO SIMPLIFICADO DE UMA LOJA
2 CREATE TABLE PRODUTO (
3     ID_PRODUTO BIGSERIAL NOT NULL,
4     NOME_PRODUTO VARCHAR(40) NOT NULL,
5     QT_ESTOQUE INTEGER NOT NULL CHECK (QT_ESTOQUE >= 0),
6     QT_ESTOQUE_MINIMO INTEGER NOT NULL CHECK (QT_ESTOQUE_MINIMO >= 0),
7     PRECO NUMERIC(9,2) NOT NULL,
8
9     PRIMARY KEY (ID_PRODUTO)
10 );
```

Below the code, the status bar indicates "CREATE TABLE" and "Query returned successfully in 78 msec."

# NOVO BANCO DE DADOS

- A carga dos registros de PRODUTO.

The screenshot shows a PostgreSQL client interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file, copy, paste, search, and various database operations.
- Query Tab:** Active tab, showing the SQL code.
- Code:**

```
1 --- MODELO SIMPLIFICADO DE UMA LOJA
2 CREATE TABLE PRODUTO (
3     ID_PRODUTO BIGSERIAL NOT NULL,
4     NOME_PRODUTO VARCHAR(40) NOT NULL,
5     QT_ESTOQUE INTEGER NOT NULL CHECK (QT_ESTOQUE >= 0),
6     QT_ESTOQUE_MINIMO INTEGER NOT NULL CHECK (QT_ESTOQUE_MINIMO >= 0),
7     PRECO NUMERIC(9,2) NOT NULL,
8
9     PRIMARY KEY (ID_PRODUTO)
10);
11
12 INSERT INTO PRODUTO VALUES (1200,'APARELHO DE TV EXCELSIOR', 100, 20, 2450.00);
13 INSERT INTO PRODUTO VALUES (1210,'SMARTPHONE CELTOP', 104, 100, 1999.99);
14 INSERT INTO PRODUTO VALUES (1220,'NOTEBOOK EPSTOLA', 50, 51, 3500.00);
15
```
- Data Output Tab:** Shows the result of the last query: "INSERT 0 1".
- Messages Tab:** Shows the message: "Query returned successfully in 69 msec."

# NOVO BANCO DE DADOS

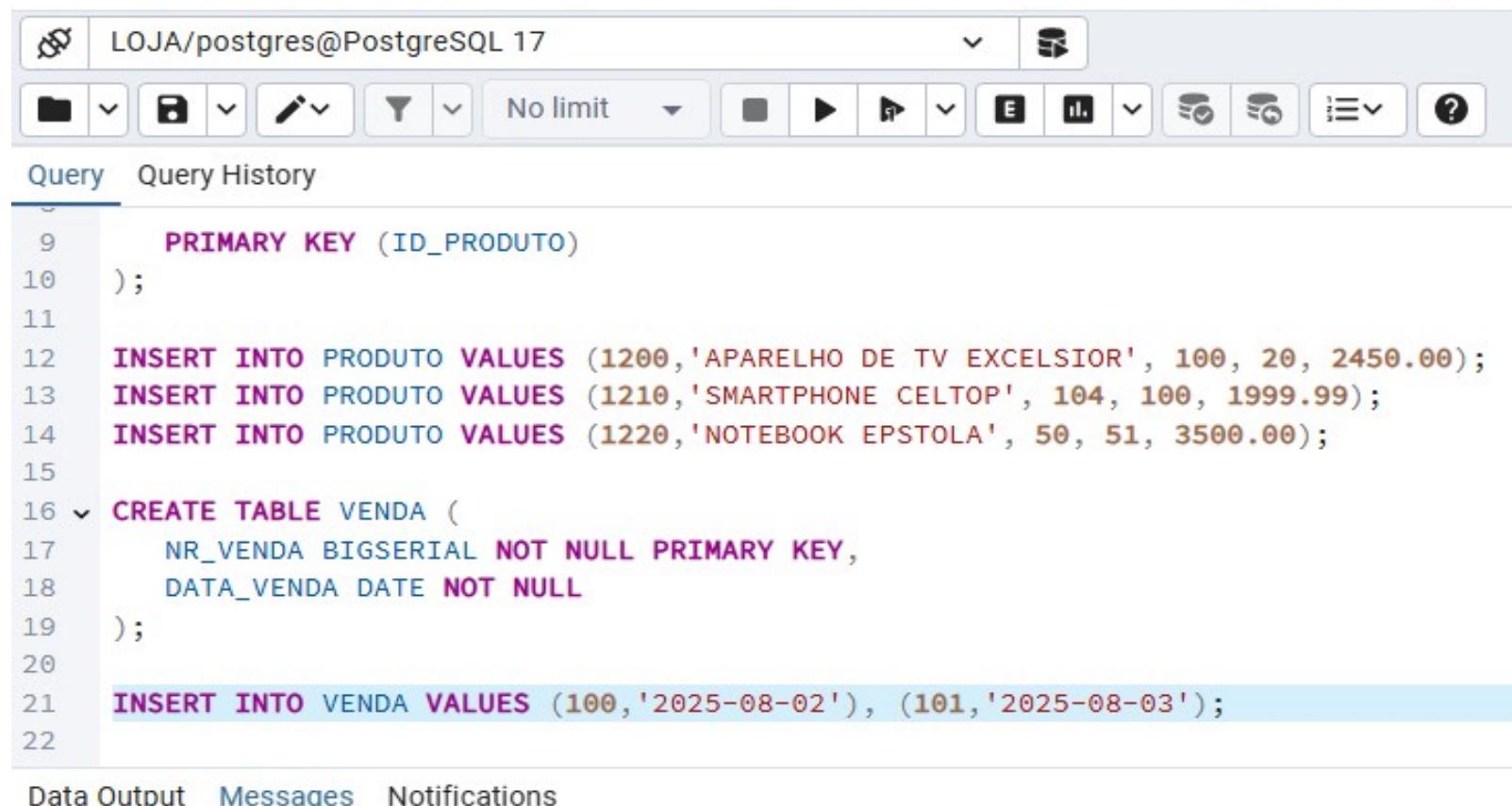
```
15  
16 CREATE TABLE VENDA (  
17     NR_VENDA BIGSERIAL NOT NULL PRIMARY KEY,  
18     DATA_VENDA DATE NOT NULL  
19 );  
20  
21
```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 81 msec.

# NOVO BANCO DE DADOS



The screenshot shows a PostgreSQL database client interface. The title bar indicates the connection is to 'LOJA/postgres@PostgreSQL 17'. The toolbar includes various icons for file operations, search, and navigation. Below the toolbar, there are tabs for 'Query' (which is selected) and 'Query History'. The main area displays the following SQL code:

```
9      PRIMARY KEY (ID_PRODUTO)
10 );
11
12 INSERT INTO PRODUTO VALUES (1200,'APARELHO DE TV EXCELSIOR', 100, 20, 2450.00);
13 INSERT INTO PRODUTO VALUES (1210,'SMARTPHONE CELTOP', 104, 100, 1999.99);
14 INSERT INTO PRODUTO VALUES (1220,'NOTEBOOK EPSTOLA', 50, 51, 3500.00);
15
16 CREATE TABLE VENDA (
17     NR_VENDA BIGSERIAL NOT NULL PRIMARY KEY,
18     DATA_VENDA DATE NOT NULL
19 );
20
21 INSERT INTO VENDA VALUES (100,'2025-08-02'), (101,'2025-08-03');
22
```

Data Output    Messages    Notifications

INSERT 0 2

Query returned successfully in 69 msec.

# NOVO BANCO DE DA

The screenshot shows a PostgreSQL database client interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file operations, search, edit, and navigation.
- Query History:** Shows the history of executed queries.
- Current Query:** A multi-line SQL script for creating a table and inserting data.
- Data Output:** Displays the result of the last query, indicating a successful CREATE TABLE operation.
- Messages:** Shows the message "Query returned successfully in 82 msec."

```
21 INSERT INTO VENDA VALUES (100,'2025-08-02'), (101,'2025-08-03');
22
23 CREATE TABLE PRODUTO_VENDIDO (
24     NR_VENDA BIGINT NOT NULL,
25     NR_SEQUENCIAL INTEGER NOT NULL,
26     ID_PRODUTO BIGINT NOT NULL,
27     QUANTIDADE INTEGER NOT NULL,
28     PRECO NUMERIC(9,2) NOT NULL,
29
30     PRIMARY KEY (NR_VENDA, NR_SEQUENCIAL, ID_PRODUTO),
31
32     FOREIGN KEY (NR_VENDA) REFERENCES VENDA (NR_VENDA)
33     ON DELETE RESTRICT,
34
35     FOREIGN KEY (ID_PRODUTO) REFERENCES PRODUTO (ID_PRODUTO)
36     ON DELETE RESTRICT
37
38 );
39
40
```

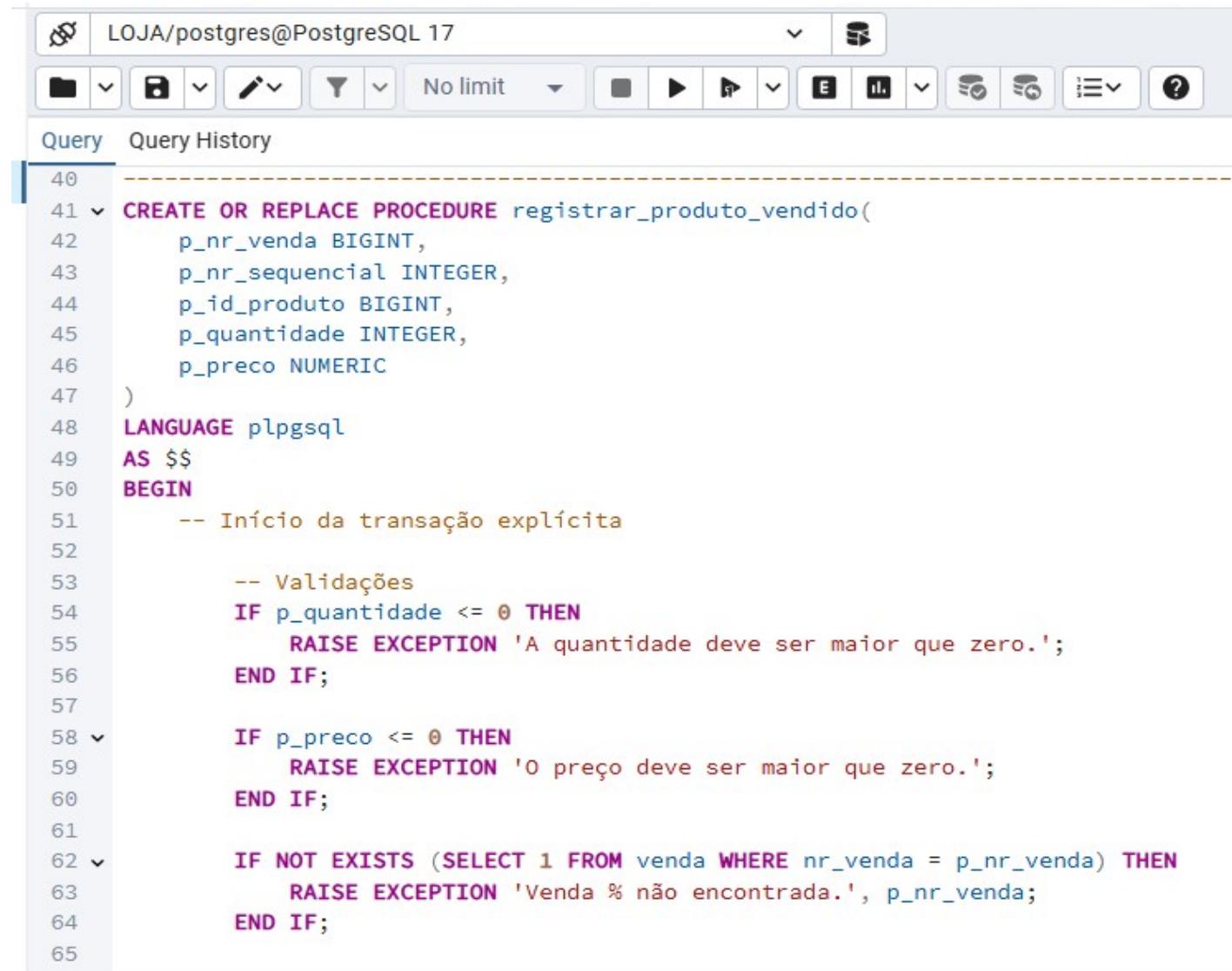
Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 82 msec.

# Stored Procedure com Transação

- Eis um SP para registrar um novo **PRODUTO\_VENDIDO**.
- Durante o processo, uma **transação** englobará o processo de inserção em **PRODUTO\_VENDIDO** e a atualização do estoque em **PRODUTO**.



The screenshot shows the pgAdmin 4 interface with a query editor window. The title bar indicates the connection is to 'LOJA/postgres@PostgreSQL 17'. The toolbar has various icons for database management. The main area is titled 'Query' and shows a code editor with a numbered line view. The code is a PostgreSQL stored procedure named 'registrar\_produto\_vendido'.

```
40
41 CREATE OR REPLACE PROCEDURE registrar_produto_vendido(
42     p_nr_venda BIGINT,
43     p_nr_sequencial INTEGER,
44     p_id_produto BIGINT,
45     p_quantidade INTEGER,
46     p_preco NUMERIC
47 )
48 LANGUAGE plpgsql
49 AS $$ 
50 BEGIN
    -- Início da transação explícita

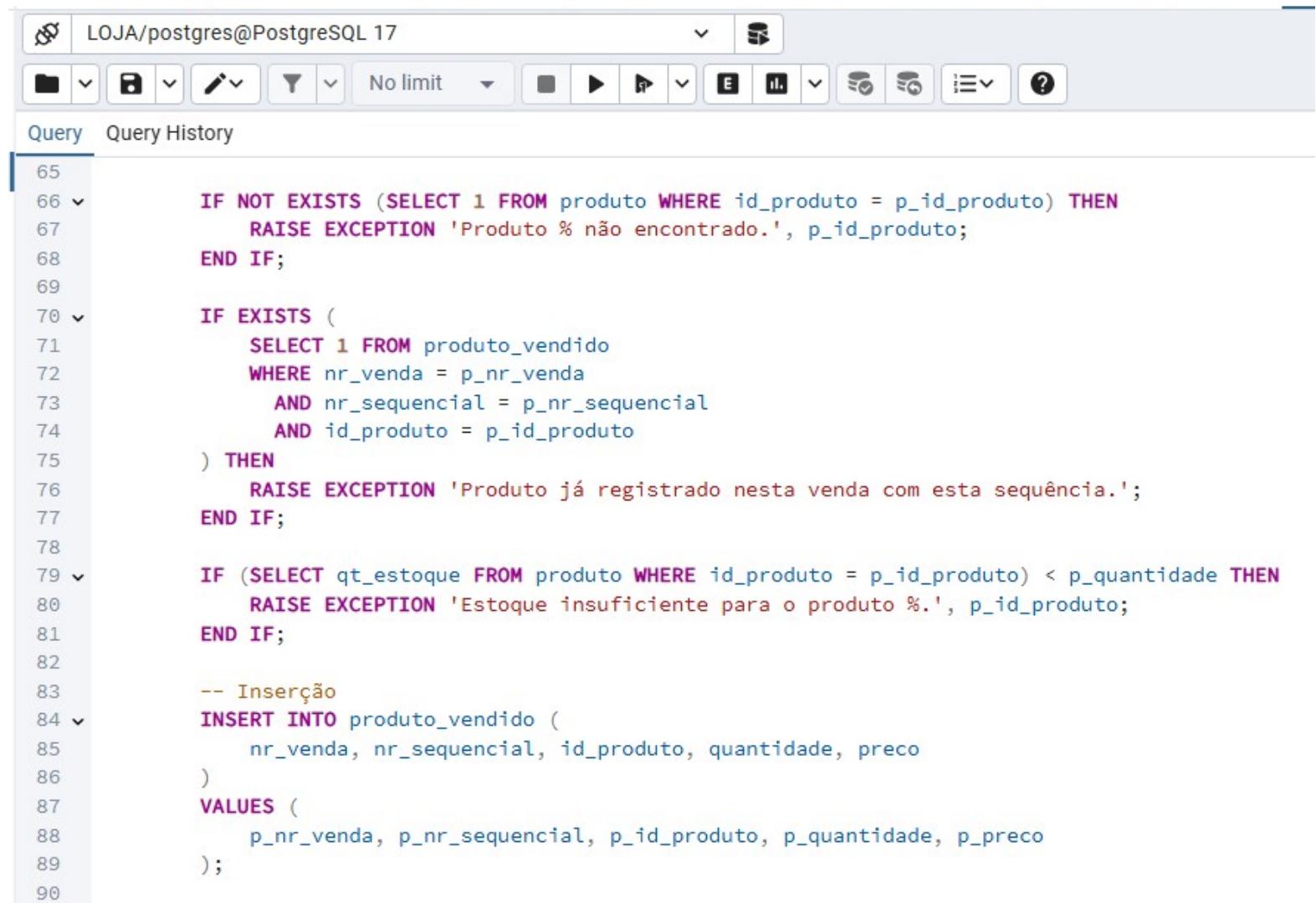
    -- Validações
    IF p_quantidade <= 0 THEN
        RAISE EXCEPTION 'A quantidade deve ser maior que zero.';
    END IF;

    IF p_preco <= 0 THEN
        RAISE EXCEPTION 'O preço deve ser maior que zero.';
    END IF;

    IF NOT EXISTS (SELECT 1 FROM venda WHERE nr_venda = p_nr_venda) THEN
        RAISE EXCEPTION 'Venda % não encontrada.', p_nr_venda;
    END IF;
```

# Stored Procedure com Transação

- Os dados de entrada são consistidos antes das atualizações de tabelas.

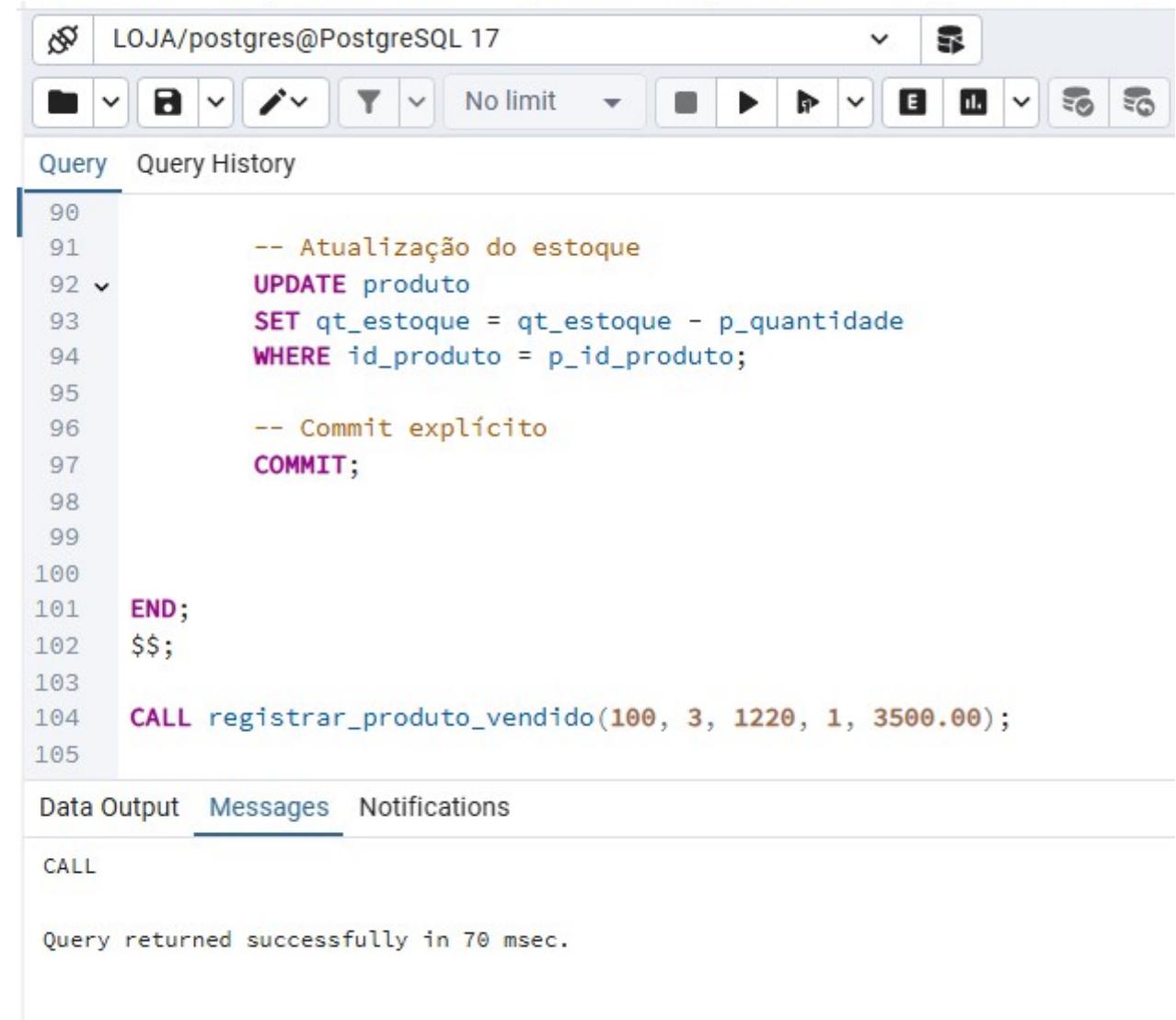


The screenshot shows a pgAdmin 4 interface with a query editor window. The title bar indicates the connection is to 'LOJA/postgres@PostgreSQL 17'. The toolbar has various icons for file operations, search, and execution. The main area is a code editor with tabs for 'Query' and 'Query History'. The 'Query' tab is selected, displaying the following PL/pgSQL code:

```
65
66 IF NOT EXISTS (SELECT 1 FROM produto WHERE id_produto = p_id_produto) THEN
67     RAISE EXCEPTION 'Produto % não encontrado.', p_id_produto;
68 END IF;
69
70 IF EXISTS (
71     SELECT 1 FROM produto_vendido
72     WHERE nr_venda = p_nr_venda
73     AND nr_sequencial = p_nr_sequencial
74     AND id_produto = p_id_produto
75 ) THEN
76     RAISE EXCEPTION 'Produto já registrado nesta venda com esta sequência.';
77 END IF;
78
79 IF (SELECT qt_estoque FROM produto WHERE id_produto = p_id_produto) < p_quantidade THEN
80     RAISE EXCEPTION 'Estoque insuficiente para o produto %.', p_id_produto;
81 END IF;
82
83 -- Inserção
84 INSERT INTO produto_vendido (
85     nr_venda, nr_sequencial, id_produto, quantidade, preco
86 )
87 VALUES (
88     p_nr_venda, p_nr_sequencial, p_id_produto, p_quantidade, p_preco
89 );
90
```

# Stored Procedure com Transação

- A **transação** é confirmada com o comando **COMMIT**.



The screenshot shows a pgAdmin 4 interface with a query editor window. The connection is set to LOJA/postgres@PostgreSQL 17. The query tab is active, displaying the following PostgreSQL code:

```
90 -- Atualização do estoque
91 UPDATE produto
92 SET qt_estoque = qt_estoque - p_quantidade
93 WHERE id_produto = p_id_produto;
94
95 -- Commit explícito
96 COMMIT;
97
98
99
100
101 END;
102 $$;
103
104 CALL registrar_produto_vendido(100, 3, 1220, 1, 3500.00);
105
```

The code performs an update on the 'produto' table, decreases the quantity by the value in 'p\_quantidade' where the product ID matches 'p\_id\_produto'. It then explicitly commits the transaction. Finally, it calls a function 'registrar\_produto\_vendido' with parameters 100, 3, 1220, 1, and 3500.00.

The messages tab at the bottom shows the output of the query:

```
CALL
```

Query returned successfully in 70 msec.

# Stored Procedure com Transação

- Confirmando a inserção na TABELA **PRODUTO\_VENDIDO**.

```
101  END;
102  $$;
103
104  CALL registrar_produto_vendido(100, 3, 1220, 1, 3500.00);
105
106  SELECT * FROM PRODUTO_VENDIDO;
```

Data Output Messages Notifications

	nr_venda [PK] bigint	nr_sequencial [PK] integer	id_produto [PK] bigint	quantidade integer	preco numeric (9,2)
1	100	3	1220	1	3500.00

# Stored Procedure com Transação

- Confirmando a alteração na tabela **PRODUTO**.
- Havia 50 unidades do **NOTEBOOK EPSTOLA**.
- Com a venda de 1 unidade agora existem 49 unidades.
- Como foi feita uma transação é impossível registrar um **PRODUTO\_VENDIDO** sem que **PRODUTO** seja atualizado.

The screenshot shows a MySQL Workbench interface. At the top, there is a code editor window containing the following SQL code:

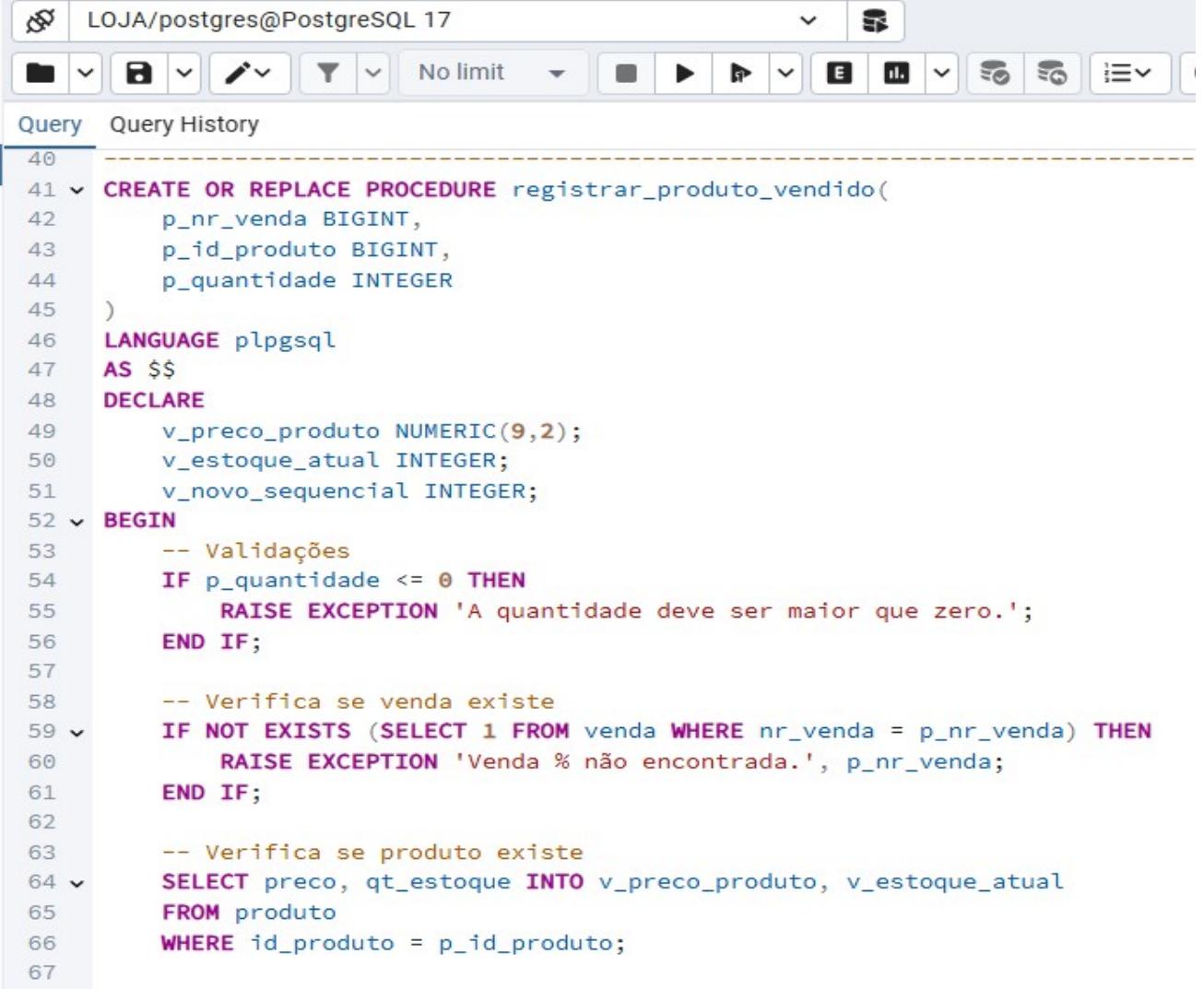
```
100
101 END;
102 $$;
103
104 CALL registrar_produto_vendido(100, 3, 1220, 1, 3500.00);
105
106 SELECT * FROM PRODUTO;
```

Below the code editor is a results grid titled "Data Output". The grid displays the following data:

	id_produto [PK] bigint	nome_produto character varying (40)	qt_estoque integer	qt_estoque_minimo integer	preco numeric (9,2)
1	1200	APARELHO DE TV EXCELSIOR	100	20	2450.00
2	1210	SMARTPHONE CELTOP	104	100	1999.99
3	1220	NOTEBOOK EPSTOLA	49	51	3500.00

# Stored Procedure com Transação (versão 2)

- Uma **nova versão do SP** em que **p\_preco** e **p\_nr\_sequencial** não precisem ser repassados como parâmetros de entrada.
- O **preço** pode ser conseguido através do acesso à **tabela PRODUTO** e o **número sequencial** pode ser **gerado automaticamente** pelo próprio stored procedure.

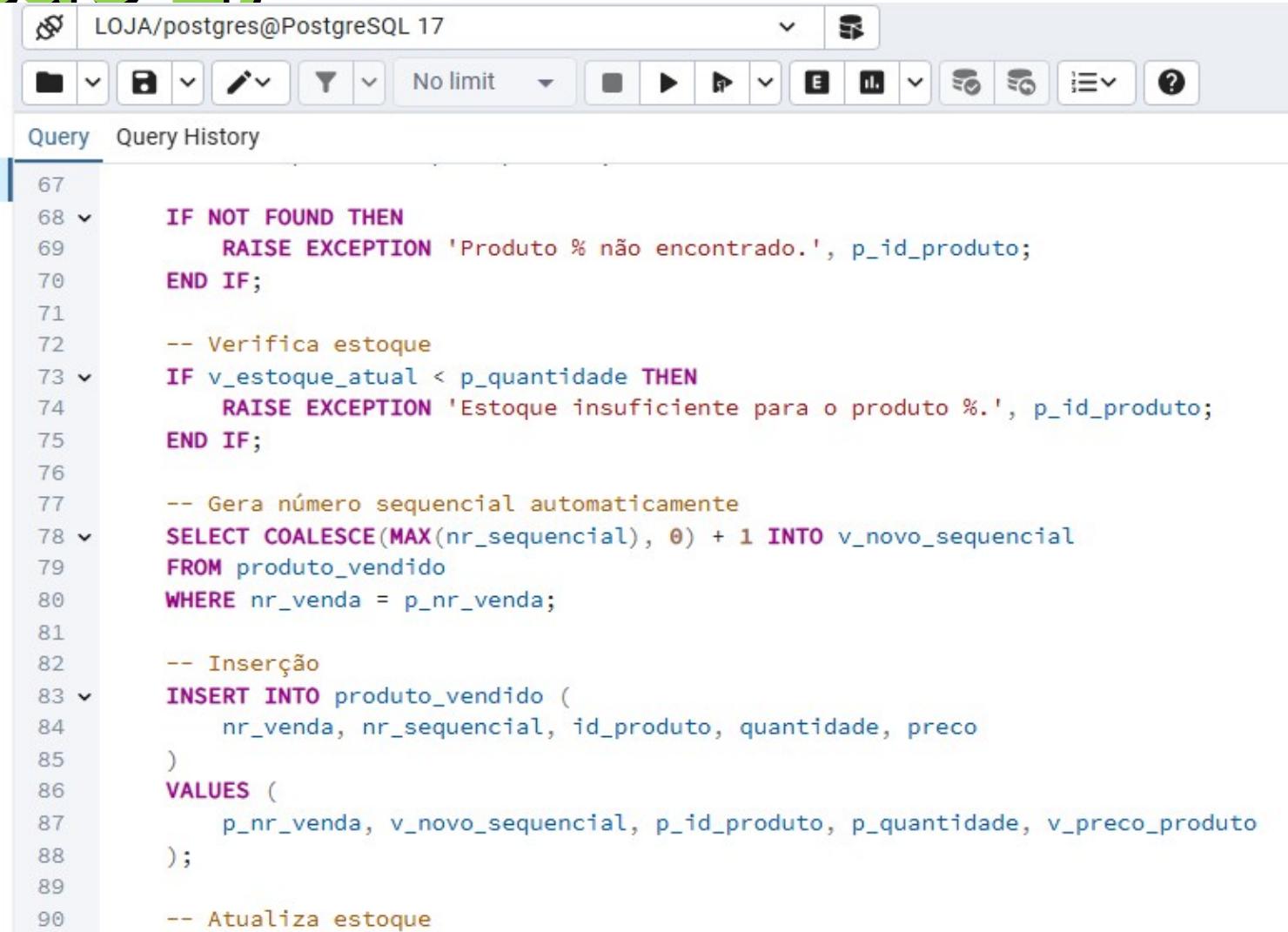


```
40
41 CREATE OR REPLACE PROCEDURE registrar_produto_vendido(
42     p_nr_venda BIGINT,
43     p_id_produto BIGINT,
44     p_quantidade INTEGER
45 )
46 LANGUAGE plpgsql
47 AS $$ 
48 DECLARE
49     v_preco_produto NUMERIC(9,2);
50     v_estoque_atual INTEGER;
51     v_novo_sequencial INTEGER;
52 BEGIN
53     -- Validações
54     IF p_quantidade <= 0 THEN
55         RAISE EXCEPTION 'A quantidade deve ser maior que zero.';
56     END IF;

57     -- Verifica se venda existe
58     IF NOT EXISTS (SELECT 1 FROM venda WHERE nr_venda = p_nr_venda) THEN
59         RAISE EXCEPTION 'Venda % não encontrada.', p_nr_venda;
60     END IF;

61     -- Verifica se produto existe
62     SELECT preco, qt_estoque INTO v_preco_produto, v_estoque_atual
63     FROM produto
64     WHERE id_produto = p_id_produto;
65
66     -- Atualiza estoque
67     UPDATE produto
68     SET qt_estoque = v_estoque_atual - p_quantidade
69     WHERE id_produto = p_id_produto;
70
71     -- Gera novo número sequencial
72     v_novo_sequencial := nextval('sequencial');
73
74     -- Insere venda
75     INSERT INTO venda (nr_venda, id_produto, quantidade, preco_unitario)
76     VALUES (p_nr_venda, p_id_produto, p_quantidade, v_preco_produto);
77
78     -- Atualiza estoque
79     UPDATE produto
80     SET qt_estoque = v_estoque_atual
81     WHERE id_produto = p_id_produto;
82
83     RETURN;
84 END;
```

# Stored Procedure com Transação (versão 2)

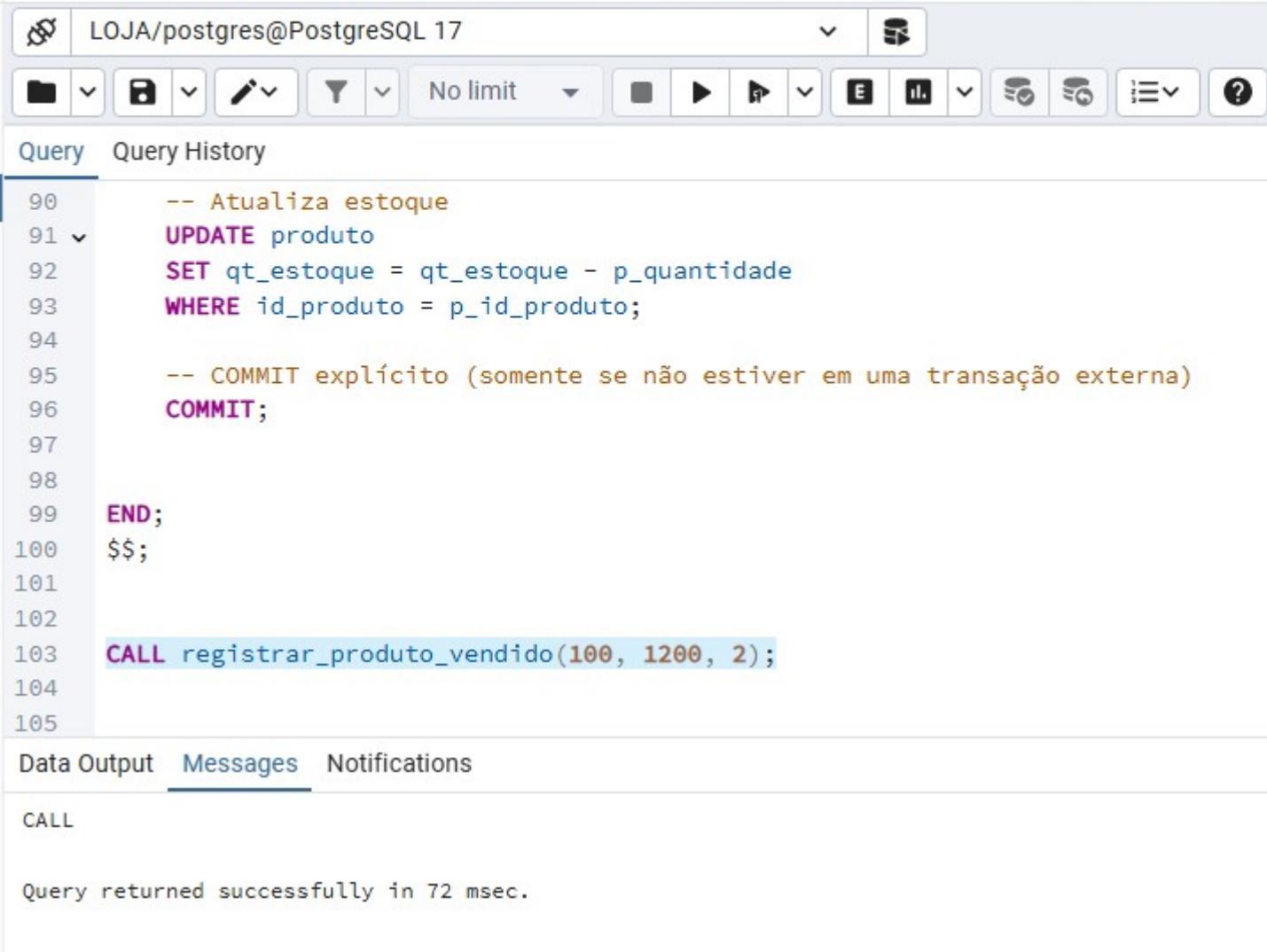


The screenshot shows a PostgreSQL query editor interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file operations, search, and various database functions.
- Tab:** The "Query" tab is selected, showing the query history.
- Code Area:** Displays a stored procedure script with line numbers from 67 to 90. The code handles product validation, checks stock levels, generates sequential numbers, inserts new sales records, and updates inventory.

```
67      IF NOT FOUND THEN
68          RAISE EXCEPTION 'Produto % não encontrado.', p_id_produto;
69      END IF;
70
71      -- Verifica estoque
72      IF v_estoque_atual < p_quantidade THEN
73          RAISE EXCEPTION 'Estoque insuficiente para o produto %.', p_id_produto;
74      END IF;
75
76      -- Gera número sequencial automaticamente
77      SELECT COALESCE(MAX(nr_sequencial), 0) + 1 INTO v_novo_sequencial
78      FROM produto_vendido
79      WHERE nr_venda = p_nr_venda;
80
81      -- Inserção
82      INSERT INTO produto_vendido (
83          nr_venda, nr_sequencial, id_produto, quantidade, preco
84      )
85      VALUES (
86          p_nr_venda, v_novo_sequencial, p_id_produto, p_quantidade, v_preco_produto
87      );
88
89      -- Atualiza estoque
90
```

# Stored Procedure com Transação (versão 2)



The screenshot shows a PostgreSQL client interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file operations, search, and various database management functions.
- Query Tab:** Active tab, showing the code for a stored procedure.

```
90      -- Atualiza estoque
91      UPDATE produto
92      SET qt_estoque = qt_estoque - p_quantidade
93      WHERE id_produto = p_id_produto;
94
95      -- COMMIT explícito (somente se não estiver em uma transação externa)
96      COMMIT;
97
98
99  END;
100 $$;
101
102
103 CALL registrar_produto_vendido(100, 1200, 2);
104
105
```
- Data Output Tab:** Shows the command `CALL`.
- Messages Tab:** Shows the message `Query returned successfully in 72 msec.`

# Stored Procedure com Transação (versão 2)

- Confirmando a inserção em PRODUTO\_VENDIDO.

```
99    END;
100   $$;
101
102
103   CALL registrar_produto_vendido(100, 1200, 2);
104
105
106   SELECT * FROM PRODUTO_VENDIDO;
107
```

Data Output    Messages    Notifications

≡+

	nr_venda [PK] bigint	nr_sequencial [PK] integer	id_produto [PK] bigint	quantidade integer	preco numeric (9,2)
1	100	3	1220	1	3500.00
2	100	4	1200	2	2450.00

# Stored Procedure com Transação (versão 2)

- Confirmando a alteração do estoque do aparelho de TV.
- Com a venda de 2 unidades o estoque foi reduzido de 100 para 98.

```
99    END;
100   $$;
101
102
103   CALL registrar_produto_vendido(100, 1200, 2);
104
105
106   SELECT * FROM PRODUTO;
107
```

Data Output Messages Notifications

	id_produto [PK] bigint	nome_produto character varying (40)	qt_estoque integer	qt_estoque_minimo integer	preco numeric (9,2)
1	1210	SMARTPHONE CELTOP	104	100	1999.99
2	1220	NOTEBOOK EPSTOLA	49	51	3500.00
3	1200	APARELHO DE TV EXCELSIOR	98	20	2450.00

# Trigger

- Um **trigger** (ou **gatilho**) é um **objeto de banco de dados** que define uma **ação automática** a ser **executada** em resposta a um determinado **evento** que ocorre em uma **tabela** ou **visão (view)**.
- Um **trigger** é um **código PL/pgSQL** ou **SQL** que é **executado automaticamente** pelo sistema de banco de dados quando uma **operação específica** acontece, como um **INSERT**, **UPDATE**, **DELETE** ou **TRUNCATE**.
- Um **trigger** nunca é **invocado pelo usuário** mas sim **executado automaticamente** sempre que um **evento** ocorre.

# Trigger

- Componentes principais de uma trigger:
  - **Evento que ativa a trigger:**
    - BEFORE INSERT, AFTER INSERT, BEFORE DELETE etc.
  - **Tabela associada:**
    - A tabela em que o evento ocorre.
  - **Condição opcional:**
    - Pode haver lógica para executar a trigger somente em certas condições.
  - **Ação definida:**
    - Código que será executado automaticamente quando a trigger for ativada.

# Trigger

- Quando nos referirmos a uma **operação** com uma trigger, esta é conhecida por **trigger de função** ou **trigger function**.
- **Trigger** e **função de trigger** são **duas coisas diferentes**, onde a primeira pode ser criada utilizando a instrução **CREATE TRIGGER**, enquanto que a última é definida pelo comando **CREATE FUNCTION**.
- Em linhas gerais, com as **triggers** definimos qual tarefa **executar**, e com as **triggers de função** definimos como essa tarefa será realizada.

# Trigger

- Ao termos uma **grande quantidade de acessos ao banco de dados** por múltiplas aplicações, a utilização das **triggers** é de grande utilidade, e com isso, podemos **manter a integridade de dados complexos**, além de **podermos acompanhar as mudanças ou o log a cada modificação ocorrida nos dados presentes numa tabela**.
- Enquanto em **funções comuns** você pode programar para que sejam recebidos valores e armazenados em variáveis para serem trabalhados no corpo da função, a **trigger** vai trabalhar com os **objetos** (variáveis) que armazenam os dados de acordo com determinada ação programada na **trigger**.

# Trigger

- As ações são:
  - Insert;
  - Update;
  - Delete;
  - Truncate.
- Para cada ação os objetos que ficam disponíveis são:
  - Insert: NEW;
  - Update: NEW e OLD;
  - Delete: OLD;

# Trigger

- Os objetos trabalham da seguinte forma:
  - **NEW**: Todos os dados que foram inseridos ou atualizados (**insert** e **update**, respectivamente);
  - **OLD**: Todos os dados que foram deletados ou sobrescritos (**delete** e **update**, respectivamente);
- Existem ainda outras variáveis que podem ser utilizadas na TRIGGER.

# Trigger

- Vamos então entender como funciona a criação de uma *trigger*:
- **CREATE OR REPLACE FUNCTION nome\_da\_funcao()**
- **RETURNS trigger AS \$\$**
- **begin**
- --// CORPO DE AÇÕES DA FUNÇÃO
- **return new;**
- **end;**
- **\$\$ LANGUAGE plpgsql;**

# Trigger

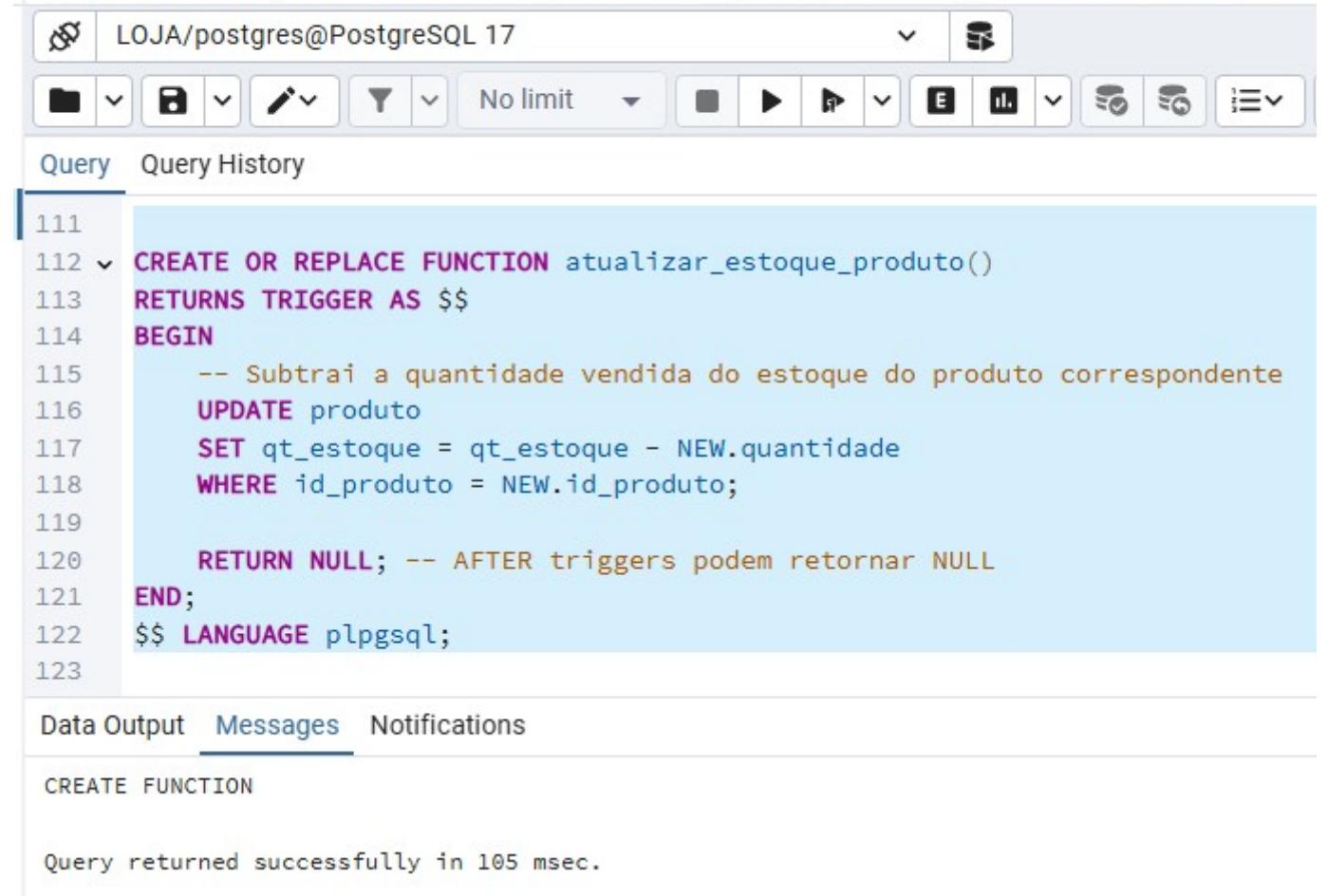
- Onde:
- ">Returns trigger" está diretamente declarando que sua função irá trabalhar com uma **trigger**.
- " \$\$ " delimita o corpo da **trigger**. Você pode trabalhar também com **\$BODY\$** (o **\$\$** é automaticamente convertido para **\$BODY\$** no momento da criação).

# Trigger

- "Return new" especifica qual o tipo de retorno da trigger você deseja trabalhar no corpo da procedure (**NEW** ou **OLD**).
- **plpgsql** é a linguagem nativa do PostgreSQL. Você também pode configurar outras linguagens para programar suas procedures.

# Trigger

- Aqui criamos uma **função de trigger**.
- Ela atualiza o **estoque** do **PRODUTO**.
- O registro **NEW** corresponde ao **registro** que acabou de ser **inserido** em **PRODUTO\_VENDIDO**.



The screenshot shows a pgAdmin 4 interface with a query editor window. The connection is set to LOJA/postgres@PostgreSQL 17. The query tab is selected, displaying the following PostgreSQL code:

```
111
112 CREATE OR REPLACE FUNCTION atualizar_estoque_produto()
113 RETURNS TRIGGER AS $$ 
114 BEGIN
115     -- Subtrai a quantidade vendida do estoque do produto correspondente
116     UPDATE produto
117     SET qt_estoque = qt_estoque - NEW.quantidade
118     WHERE id_produto = NEW.id_produto;
119
120     RETURN NULL; -- AFTER triggers podem retornar NULL
121 END;
122 $$ LANGUAGE plpgsql;
```

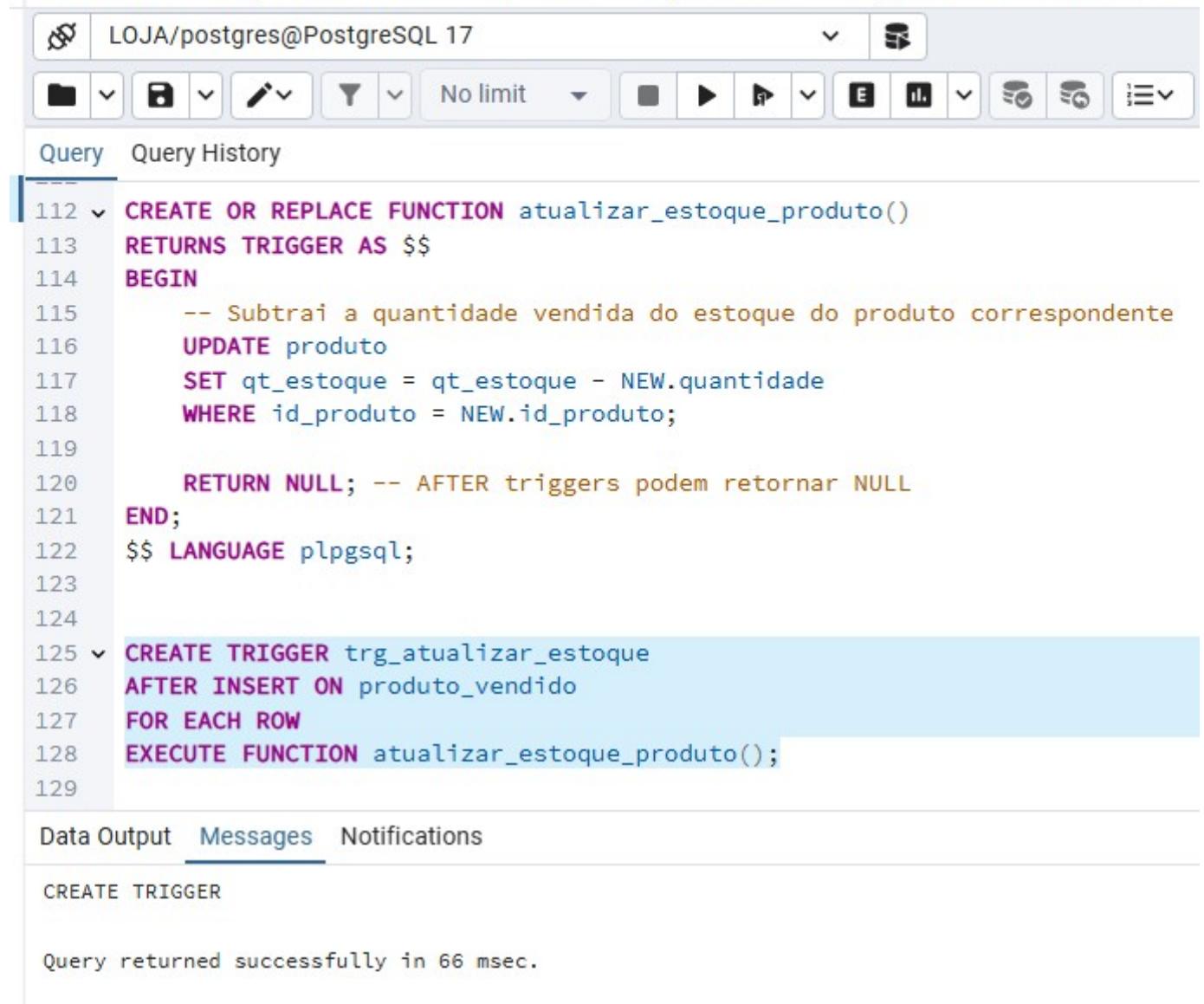
Below the code, the message tab shows:

CREATE FUNCTION

Query returned successfully in 105 msec.

# Trigger

- Criação da **trigger** que chama essa **função de trigger**.
- Nela identificamos o **evento** que **dispara** a **execução** da **função de trigger** (**AFTER INSERT PRODUTO\_VENDIDO**).



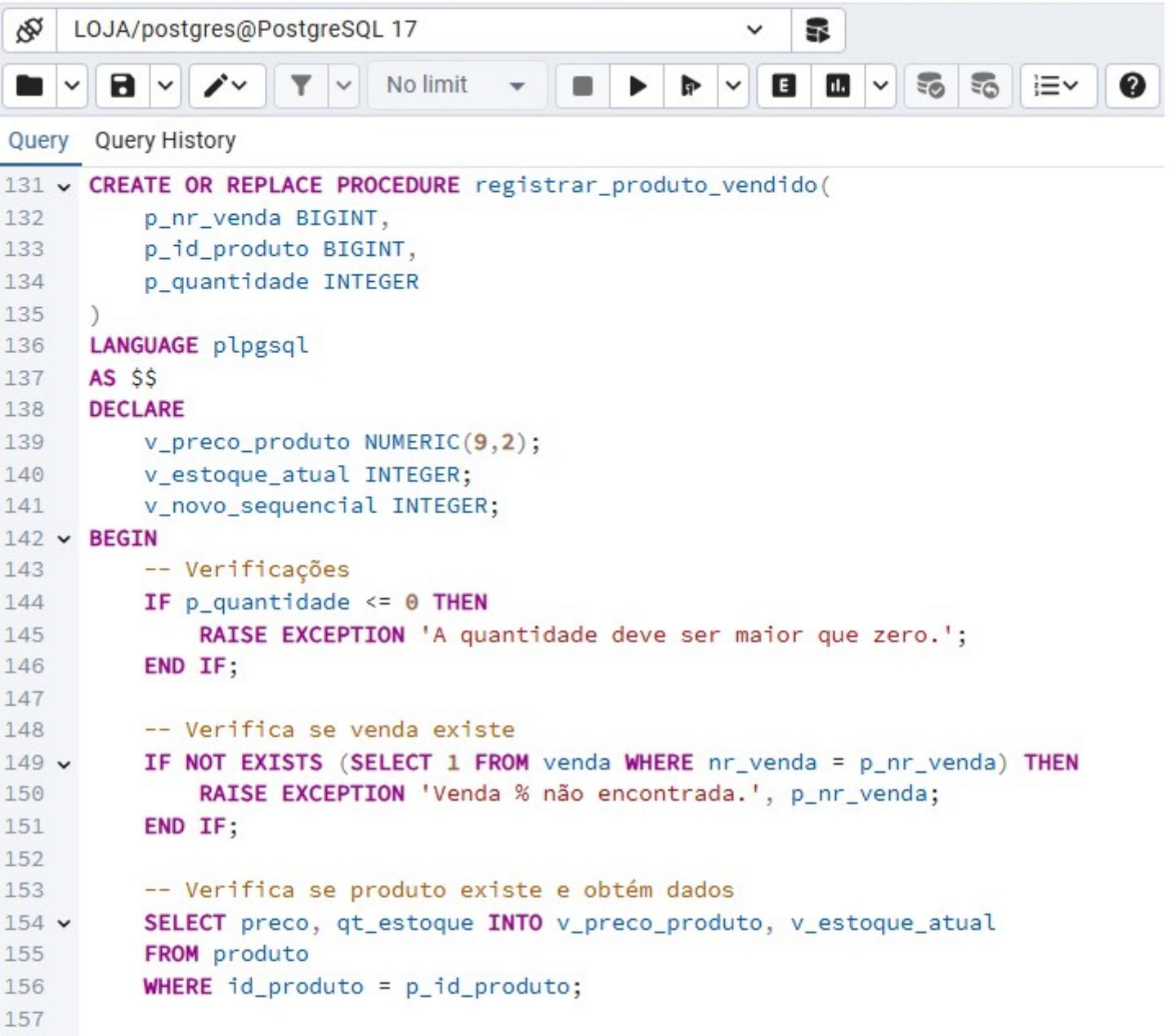
The screenshot shows a pgAdmin 4 interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file operations, search, filter, and various database management functions.
- Query Tab:** Active tab, showing the SQL code for creating a function and a trigger.
- Code Content:**

```
112 v CREATE OR REPLACE FUNCTION atualizar_estoque_produto()
113 RETURNS TRIGGER AS $$ 
114 BEGIN
115     -- Subtrai a quantidade vendida do estoque do produto correspondente
116     UPDATE produto
117     SET qt_estoque = qt_estoque - NEW.quantidade
118     WHERE id_produto = NEW.id_produto;
119
120     RETURN NULL; -- AFTER triggers podem retornar NULL
121 END;
122 $$ LANGUAGE plpgsql;
123
124
125 v CREATE TRIGGER trg_atualizar_estoque
126 AFTER INSERT ON produto_vendido
127 FOR EACH ROW
128 EXECUTE FUNCTION atualizar_estoque_produto();
129
```
- Data Output Tab:** Shows the result of the query: "CREATE TRIGGER".
- Messages Tab:** Shows the message: "Query returned successfully in 66 msec."

# Trigger

- Agora teremos de alterar nosso **Stored Procedure** para que não ocorra uma dupla atualização do estoque em **PRODUTO**.



The screenshot shows a PostgreSQL client interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file operations, search, filter, and various database management functions.
- Tab:** "Query" is selected, showing the query history.
- Code Area:** Displays a block of PostgreSQL SQL code. The code is a `CREATE OR REPLACE PROCEDURE` named `registrar_produto_vendido`. It includes parameters for `p_nr_venda`, `p_id_produto`, and `p_quantidade`. The procedure uses the `plpgsql` language and declares variables `v_preco_produto`, `v_estoque_atual`, and `v_novo_sequencial`. It performs several checks: ensuring `p_quantidade` is not zero, verifying the existence of the sale (`nr_venda`), and checking if the product exists. Finally, it performs an update on the `produto` table.

```
131 v CREATE OR REPLACE PROCEDURE registrar_produto_vendido(
132     p_nr_venda BIGINT,
133     p_id_produto BIGINT,
134     p_quantidade INTEGER
135 )
136 LANGUAGE plpgsql
137 AS $$
138 DECLARE
139     v_preco_produto NUMERIC(9,2);
140     v_estoque_atual INTEGER;
141     v_novo_sequencial INTEGER;
142 BEGIN
143     -- Verificações
144     IF p_quantidade <= 0 THEN
145         RAISE EXCEPTION 'A quantidade deve ser maior que zero.';
146     END IF;

147     -- Verifica se venda existe
148     IF NOT EXISTS (SELECT 1 FROM venda WHERE nr_venda = p_nr_venda) THEN
149         RAISE EXCEPTION 'Venda % não encontrada.', p_nr_venda;
150     END IF;

151     -- Verifica se produto existe e obtém dados
152     SELECT preco, qt_estoque INTO v_preco_produto, v_estoque_atual
153     FROM produto
154     WHERE id_produto = p_id_produto;
155
156     -- Atualiza o estoque
157     UPDATE produto
158     SET qt_estoque = v_estoque_atual - p_quantidade
159     WHERE id_produto = p_id_produto;
```

# Trigger

- Aqui terminamos a codificação de nossa terceira versão de **Stored Procedure**.

The screenshot shows a PostgreSQL client window with the connection set to 'LOJA/postgres@PostgreSQL 17'. The main area displays the SQL code for creating a stored procedure:

```
157      IF NOT FOUND THEN
158          RAISE EXCEPTION 'Produto % não encontrado.', p_id_produto;
159      END IF;
160
161      -- Verifica estoque disponível antes de inserir
162      IF v_estoque_atual < p_quantidade THEN
163          RAISE EXCEPTION 'Estoque insuficiente para o produto %.', p_id_produto;
164      END IF;
165
166      -- Gera número sequencial automaticamente
167      SELECT COALESCE(MAX(nr_sequencial), 0) + 1 INTO v_novo_sequencial
168      FROM produto_vendido
169      WHERE nr_venda = p_nr_venda;
170
171      -- Insere o produto vendido (trigger cuidará do estoque)
172      INSERT INTO produto_vendido (
173          nr_venda, nr_sequencial, id_produto, quantidade, preco
174      )
175      VALUES (
176          p_nr_venda, v_novo_sequencial, p_id_produto, p_quantidade, v_preco_produto
177      );
178
179      -- Não atualiza o estoque aqui. A trigger faz isso.
180
181      END;
182
183 $$;
```

Below the code, the status bar indicates 'CREATE PROCEDURE' and 'Query returned successfully in 135 msec.'

# Trigger

- Agora efetuaremos o teste de nosso trigger e nova versão do stored procedure.

```
182 END;
183 $$;
184
185 SELECT * FROM PRODUTO_VENDIDO;
186
```

Data Output Messages Notifications

	nr_venda [PK] bigint	nr_sequencial [PK] integer	id_produto [PK] bigint	quantidade integer	preco numeric (9,2)
1	100	3	1220	1	3500.00
2	100	4	1200	2	2450.00

# Trigger

```
182 END;  
183 $$;  
184  
185 SELECT * FROM PRODUTO;  
186
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for new table, file, dropdown, copy, delete, export, refresh, and SQL. Below the toolbar is a table with the following data:

	<b>id_produto</b> [PK] bigint	<b>nome_produto</b> character varying (40)	<b>qt_estoque</b> integer	<b>qt_estoque_minimo</b> integer	<b>preco</b> numeric (9,2)
1	1210	SMARTPHONE CELTOP	104	100	1999.99
2	1220	NOTEBOOK EPSTOLA	49	51	3500.00
3	1200	APARELHO DE TV EXCELSIOR	98	20	2450.00

# Trigger

```
179
180      -- Não atualiza o estoque aqui. A trigger faz isso.
181
182  END;
183  $$;
184
185  CALL registrar_produto_vendido(100, 1210, 3);
186
```

Data Output Messages Notifications

CALL

Query returned successfully in 103 msec.

# Trigger

```
179
180      -- Não atualiza o estoque aqui. A trigger faz isso.
181
182  END;
183  $$;
184
185  CALL registrar_produto_vendido(100, 1210, 3);
186
187  SELECT * FROM PRODUTO_VENDIDO;
188
```

Data Output Messages Notifications



	nr_venda [PK] bigint	nr_sequencial [PK] integer	id_produto [PK] bigint	quantidade integer	preco numeric (9,2)
1	100	3	1220	1	3500.00
2	100	4	1200	2	2450.00
3	100	5	1210	3	1999.99

```
179
180      -- Não atualiza o estoque aqui. A trigger faz isso.
181
182  END;
183  $$;
184
185  CALL registrar_produto_vendido(100, 1210, 3);
186
187  SELECT * FROM PRODUTO;
188
```

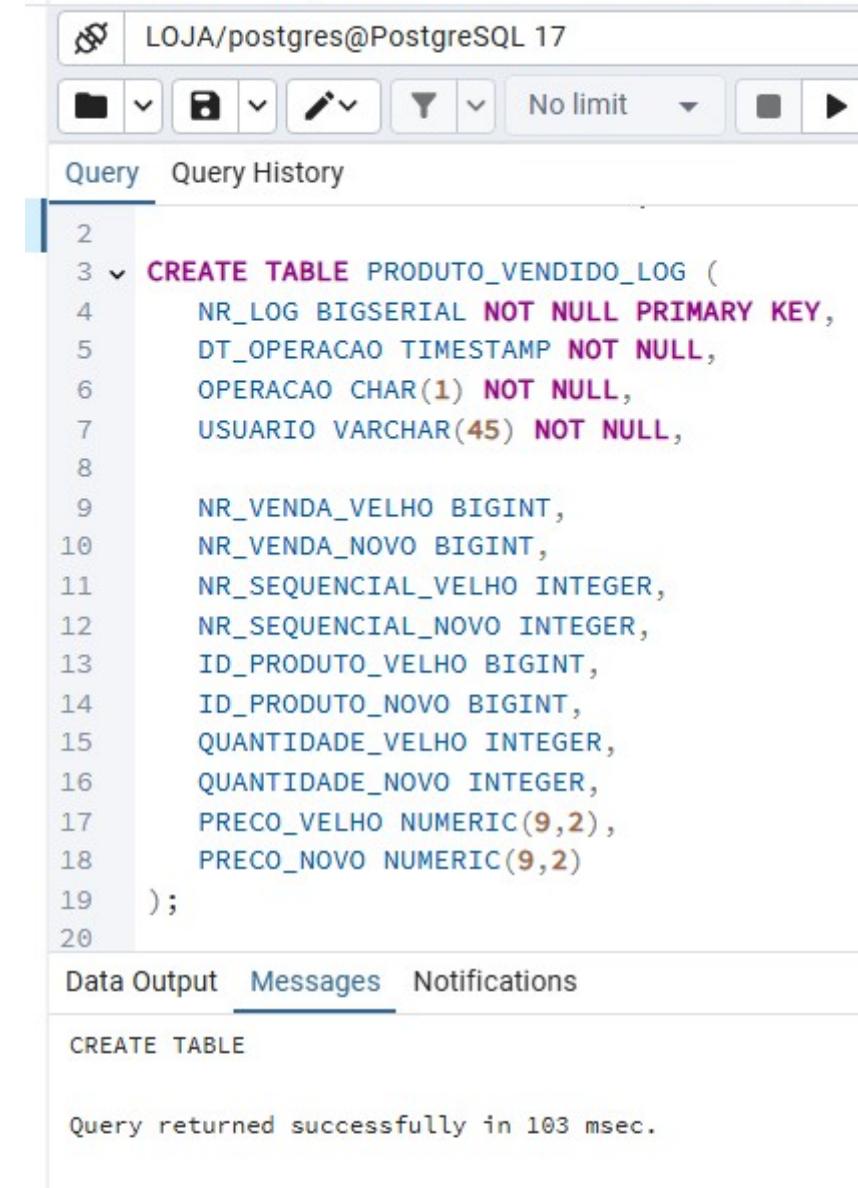
Data Output Messages Notifications



	id_produto [PK] bigint	nome_produto character varying (40)	qt_estoque integer	qt_estoque_minimo integer	preco numeric (9,2)
1	1220	NOTEBOOK EPSTOLA	49	51	3500.00
2	1200	APARELHO DE TV EXCELSIOR	98	20	2450.00
3	1210	SMARTPHONE CELTOP	101	100	1999.99

# Trigger

- Um outro uso muito comum para **Triggers (Gatilhos)** é o de manter **registros de operações de auditoria em tabelas de LOG** (Diário).



The screenshot shows a PostgreSQL database client interface. The title bar reads "LOJA/postgres@PostgreSQL 17". The toolbar includes icons for file, edit, search, and navigation. The main area has tabs for "Query" and "Query History", with "Query" selected. The code area contains the following SQL script:

```
2
3 CREATE TABLE PRODUTO_VENDIDO_LOG (
4     NR_LOG BIGSERIAL NOT NULL PRIMARY KEY,
5     DT_OPERACAO TIMESTAMP NOT NULL,
6     OPERACAO CHAR(1) NOT NULL,
7     USUARIO VARCHAR(45) NOT NULL,
8
9     NR_VENDA_VELHO BIGINT,
10    NR_VENDA_NOVO BIGINT,
11    NR_SEQUENCIAL_VELHO INTEGER,
12    NR_SEQUENCIAL_NOVO INTEGER,
13    ID_PRODUTO_VELHO BIGINT,
14    ID_PRODUTO_NOVO BIGINT,
15    QUANTIDADE_VELHO INTEGER,
16    QUANTIDADE_NOVO INTEGER,
17    PRECO_VELHO NUMERIC(9,2),
18    PRECO_NOVO NUMERIC(9,2)
19 );
20
```

The "Messages" tab is selected, showing the message "CREATE TABLE". Below it, the status message "Query returned successfully in 103 msec." is displayed.

# Trigger

```
20
21 ✓ CREATE OR REPLACE FUNCTION TRIG_LOG_VENDA ()  
22 RETURNS TRIGGER AS $$  
23 BEGIN  
24     IF TG_OP = 'INSERT'  
25     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_NOVO,  
26                                         NR_SEQUENCIAL_NOVO, ID_PRODUTO_NOVO,  
27                                         QUANTIDADE_NOVO, PRECO_NOVO) VALUES  
28                                         (NOW(), 'I', USER, NEW.NR_VENDA, NEW.NR_SEQUENCIAL,  
29                                         NEW.ID_PRODUTO, NEW.QUANTIDADE, NEW.PRECO);  
30     END IF;  
31  
32 ✓ IF TG_OP = 'UPDATE'  
33     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_NOVO,  
34                                         NR_VENDA_VELHO, NR_SEQUENCIAL_NOVO, NR_SEQUENCIAL_VELHO,  
35                                         ID_PRODUTO_NOVO, ID_PRODUTO_VELHO, QUANTIDADE_NOVO,  
36                                         QUANTIDADE_VELHO, PRECO_NOVO, PRECO_VELHO) VALUES  
37                                         (NOW(), 'U', USER, NEW.NR_VENDA, OLD.NR_VENDA,  
38                                         NEW.NR_SEQUENCIAL, OLD.NR_SEQUENCIAL, NEW.ID_PRODUTO,  
39                                         OLD.ID_PRODUTO, NEW.QUANTIDADE, OLD.QUANTIDADE,  
40                                         NEW.PRECO, OLD.PRECO);  
41     END IF;
```

# Trigger

The screenshot shows a PostgreSQL database client interface with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file, copy, paste, search, and various database operations.
- Query History:** Shows the history of queries run in the session.
- Current Query:** The code for a trigger function:

```
41     END IF;
42
43     IF TG_OP = 'DELETE'
44     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_VELHO,
45                                         NR_SEQUENCIAL_VELHO, ID_PRODUTO_VELHO,
46                                         QUANTIDADE_VELHO, PRECO_VELHO) VALUES
47                                         (NOW(), 'I', USER, OLD.NR_VENDA, OLD.NR_SEQUENCIAL,
48                                         OLD.ID_PRODUTO, OLD.QUANTIDADE, OLD.PRECO);
49     END IF;
50
51     RETURN NEW;
52 END;
53 $$ LANGUAGE PLPGSQL;
```

**Data Output:** CREATE FUNCTION

**Messages:** Query returned successfully in 79 msec.

# Trigger

The screenshot shows a PostgreSQL query editor interface. At the top, there is a connection bar with a user icon and the text "LOJA/postgres@PostgreSQL 17". Below the connection bar is a toolbar with several icons: a folder, a magnifying glass, a pencil, a funnel, and a dropdown menu set to "No limit". To the right of the toolbar are three navigation buttons: a square, a right-pointing triangle, and a left-pointing triangle.

The main area is divided into two tabs: "Query" (which is selected) and "Query History". The "Query" tab contains the following SQL code:

```
55
56 CREATE TRIGGER LOG_VENDA_INSERT
57 AFTER INSERT ON PRODUTO_VENDIDO
58 FOR EACH ROW
59 EXECUTE PROCEDURE TRIG_LOG_VENDA();
```

Below the code, the numbers 60 and 61 are visible, likely indicating the next lines of the script. At the bottom of the editor, there are three tabs: "Data Output", "Messages" (which is selected), and "Notifications". The "Messages" tab displays the output of the query:

CREATE TRIGGER

Query returned successfully in 89 msec.

# Trigger

The screenshot shows a PostgreSQL client window with the following details:

- Connection:** LOJA/postgres@PostgreSQL 17
- Toolbar:** Includes icons for file, database, edit, search, and execution.
- Query Tab:** Active tab, showing the query history.
- Code Area:** Contains the following SQL code:

```
60
61 ✓ CREATE TRIGGER LOG_VENDA_UPDATE
62     AFTER UPDATE ON PRODUTO_VENDIDO
63     FOR EACH ROW
64     EXECUTE PROCEDURE TRIG_LOG_VENDA( );
65
66 ✓ CREATE TRIGGER LOG_VENDA_DELETE
67     AFTER DELETE ON PRODUTO_VENDIDO
68     FOR EACH ROW
69     EXECUTE PROCEDURE TRIG_LOG_VENDA( );
70
```
- Data Output Tab:** Shows the result of the query: "CREATE TRIGGER".
- Messages Tab:** Shows the message: "Query returned successfully in 106 msec."

# Trigger

- Testando nosso LOG.

```
72  
73     SELECT * FROM VENDA;  
74
```

Data Output    Messages    Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for file operations, a table view, a search bar, and other functions. Below the toolbar is a table titled 'VENDA'. The table has three columns: 'nr\_venda' (PK) of type bigint and 'data\_venda' of type date. There are two rows in the table: one with nr\_venda 100 and data\_venda 2025-08-02, and another with nr\_venda 101 and data\_venda 2025-08-03.

	nr_venda [PK] bigint	data_venda date
1	100	2025-08-02
2	101	2025-08-03

```
73     SELECT * FROM VENDA;  
74  
75     INSERT INTO VENDA (DATA_VENDA) VALUES ('2025-09-22');  
76
```

Data Output    Messages    Notifications

INSERT 0 1

Query returned successfully in 190 msec.

# Trigger

```
74  
75 INSERT INTO VENDA (DATA_VENDA) VALUES ('2025-09-22');  
76  
77 SELECT * FROM VENDA;
```

Data Output Messages Notifications



	nr_venda [PK] bigint	data_venda date
1	100	2025-08-02
2	101	2025-08-03
3	1	2025-09-22

```
77 SELECT * FROM PRODUTO_VENDIDO;  
78  
79 INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80  
81
```

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 204 msec.

# Trigger

```
76  
77   SELECT * FROM PRODUTO_VENDIDO;  
78  
79   INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80
```

Data Output Messages Notifications

	nr_venda [PK] bigint	nr_sequencial [PK] integer	id_produto [PK] bigint	quantidade integer	preco numeric (9,2)
1	100	3	1220	1	3500.00
2	100	4	1200	2	2450.00
3	100	5	1210	3	1999.99
4	1	1	1200	1	1225.00

```
78  
79   INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80  
81   SELECT * FROM PRODUTO_VENDIDO_LOG;
```

Data Output Messages Notifications

	nr_log [PK] bigint	dt_operacao timestamp without time zone	operacao character (1)	usuario character varying (45)	nr_venda_velho bigint	nr_venda_novo bigint	nr_sequencial_velho integer	nr_sequencial_novo integer	id_produto_velho bigint	id_produto_novo bigint
1	1	2025-08-03 16:50:19.593757	I	postgres	[null]	1	[null]	1	[null]	1200

# Trigger

```
78  
79 INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80  
81 SELECT * FROM PRODUTO_VENDIDO_LOG;  
82  
83 UPDATE PRODUTO_VENDIDO SET ID_PRODUTO = 1220 WHERE NR_VENDA = 1 AND ID_PRODUTO = 1200;
```

Data Output Messages Notifications

UPDATE 1

Query returned successfully in 96 msec.

```
78  
79 INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80  
81 SELECT * FROM PRODUTO_VENDIDO_LOG;  
82  
83 UPDATE PRODUTO_VENDIDO SET ID_PRODUTO = 1220 WHERE NR_VENDA = 1 AND ID_PRODUTO = 1200;
```

Data Output Messages Notifications

Showing rows: 1 to 2  

	nr_log [PK] bigint	dt_operacao timestamp without time zone	operacao character (1)	usuario character varying (45)	nr_venda_velho bigint	nr_venda_novo bigint	nr_sequencial_velho integer	nr_sequencial_novo integer	id_produto_velho bigint	id_produto_novo bigint
1	1	2025-08-03 16:50:19.593757	I	postgres	[null]	1	[null]	1	[null]	1200
2	2	2025-08-03 16:57:45.416352	U	postgres	1	1	1	1	1200	1220

# Trigger

```
78  
79  INSERT INTO PRODUTO_VENDIDO VALUES (1, 1, 1200, 1, 1225);  
80  
81  SELECT * FROM PRODUTO_VENDIDO_LOG;  
82  
83  UPDATE PRODUTO_VENDIDO SET ID_PRODUTO = 1220 WHERE NR_VENDA = 1 AND ID_PRODUTO = 1200;  
84  
85  DELETE FROM PRODUTO_VENDIDO WHERE NR_VENDA = 1 AND ID_PRODUTO = 1220;  
86
```

Data Output Messages Notifications

DELETE 1

Query returned successfully in 95 msec.

```
80  
81  SELECT * FROM PRODUTO_VENDIDO_LOG;  
82  
83  UPDATE PRODUTO_VENDIDO SET ID_PRODUTO = 1220 WHERE NR_VENDA = 1 AND ID_PRODUTO = 1200;  
84  
85  DELETE FROM PRODUTO_VENDIDO WHERE NR_VENDA = 1 AND ID_PRODUTO = 1220;  
86
```

Data Output Messages Notifications

	nr_log [PK] bigint	dt_operacao timestamp without time zone	operacao character (1)	usuario character varying (45)	nr_venda_velho bigint	nr_venda_novo bigint	nr_sequencial_velho integer	nr_sequencial_novo integer	id_produto_velho bigint	id_produto_novo bigint	quantidade_velho integer
1	1	2025-08-03 16:50:19.593757	I	postgres	[null]	1	[null]	1	[null]	1200	[null]
2	2	2025-08-03 16:57:45.416352	U	postgres	1	1	1	1	1200	1220	1
3	3	2025-08-03 17:02:57.975664	I	postgres	1	[null]	1	[null]	1220	[null]	1

# Trigger

- Note que ocorreu um erro no registro da exclusão no LOG.
- Em vez de operação = “D” operação ficou “I” de INSERT.
- É um erro causado por um Ctrl + C seguido por um Ctrl + V displicente.
- Mas é fácil de consertar.

Query Query History

```
21 ✓ CREATE OR REPLACE FUNCTION TRIG_LOG_VENDA ()
22 RETURNS TRIGGER AS $$ 
23 BEGIN
24     IF TG_OP = 'INSERT'
25     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_NOVO,
26                                         NR_SEQUENCIAL_NOVO, ID_PRODUTO_NOVO,
27                                         QUANTIDADE_NOVO, PRECO_NOVO) VALUES
28                                         (NOW(), 'I', USER, NEW.NR_VENDA, NEW.NR_SEQUENCIAL,
29                                         NEW.ID_PRODUTO, NEW.QUANTIDADE, NEW.PRECO);
29
30     END IF;
31
32 ✓     IF TG_OP = 'UPDATE'
33     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_NOVO,
34                                         NR_VENDA_VELHO, NR_SEQUENCIAL_NOVO, NR_SEQUENCIAL_VELHO,
35                                         ID_PRODUTO_NOVO, ID_PRODUTO_VELHO, QUANTIDADE_NOVO,
36                                         QUANTIDADE_VELHO, PRECO_NOVO, PRECO_VELHO) VALUES
37                                         (NOW(), 'U', USER, NEW.NR_VENDA, OLD.NR_VENDA,
38                                         NEW.NR_SEQUENCIAL, OLD.NR_SEQUENCIAL, NEW.ID_PRODUTO,
39                                         OLD.ID_PRODUTO, NEW.QUANTIDADE, OLD.QUANTIDADE,
40                                         NEW.PRECO, OLD.PRECO);
41
41     END IF;
42
43 ✓     IF TG_OP = 'DELETE'
44     THEN INSERT INTO PRODUTO_VENDIDO_LOG (DT_OPERACAO, OPERACAO, USUARIO, NR_VENDA_VELHO,
45                                         NR_SEQUENCIAL_VELHO, ID_PRODUTO_VELHO,
46                                         QUANTIDADE_VELHO, PRECO_VELHO) VALUES
47                                         (NOW(), 'D', USER, OLD.NR_VENDA, OLD.NR_SEQUENCIAL,
48                                         OLD.ID_PRODUTO, OLD.QUANTIDADE, OLD.PRECO);
48
49     END IF;
50
51     RETURN NEW;
52 END;
53 $$ LANGUAGE PLPGSQL;
```