

# PL/pgSQL

## Banco de Dados II

**Prof. Guilherme Tavares de Assis**

**Universidade Federal de Ouro Preto – UFOP**  
**Instituto de Ciências Exatas e Biológicas – ICEB**  
**Departamento de Computação – DECOM**

## Introdução

---

- PL/pgSQL (*Procedural Language for the PostgreSQL*) é uma linguagem procedural carregável desenvolvida para o SGBD PostgreSQL, que possui as seguintes características:
  - é utilizada para criar funções e gatilhos;
  - possibilita a execução de processamentos complexos;
  - permite adicionar estruturas de controle à linguagem SQL;
  - é fácil de ser utilizada;
  - é compatível com a PL/SQL padrão.

# Introdução

- A linguagem PL/pgSQL é estruturada em blocos.

```
[ <<rótulo>> ]  
[ DECLARE  
    --declarações ]  
BEGIN  
    --instruções  
END
```

- As declarações e instruções dentro de um bloco devem ser finalizadas por ponto-e-vírgula.
  - Um bloco contido dentro de outro bloco, ou seja, um sub-bloco, deve conter um ponto-e-vírgula após o seu END; entretanto, o END final que conclui o corpo da função não requer o ponto-e-vírgula.

## Introdução

---

- Todos os identificadores podem ser escritos misturando letras maiúsculas e minúsculas.
  - As letras dos identificadores são convertidas implicitamente em minúsculas, a menos que estejam entre aspas.
- Existem dois tipos de comentários na PL/pgSQL.
  - O hífen duplo (--) começa um comentário que se estende até o final da linha.
  - O /\* começa um bloco de comentário que se estende até a próxima ocorrência de \*/.
- As variáveis declaradas na seção de declaração que precede um bloco são inicializadas com seus valores-padrão toda vez que o bloco é executado, e não somente uma vez a cada chamada da função.

# Introdução

```
CREATE FUNCTION func_escopo() RETURNS integer AS $$
DECLARE
    quantidade integer := 30;
BEGIN
    RAISE NOTICE 'Aqui a quantidade é %', quantidade;
    quantidade := 50;
    -- Criar um sub-bloco
    DECLARE
        quantidade integer := 80;
    BEGIN
        RAISE NOTICE 'Aqui a quantidade é %', quantidade;
    END;
    RAISE NOTICE 'Aqui a quantidade é %', quantidade;
    RETURN quantidade;
END;
$$ LANGUAGE plpgsql;
```

## Introdução

- Ao executar a função criada pela instrução

**SELECT func\_escopo();**

os seguintes resultados são obtidos:

- *Data Output:*

```
Func_escopo
-----
                    50
```

- *Messages:*

```
NOTA: Aqui a quantidade é 30
NOTA: Aqui a quantidade é 80
NOTA: Aqui a quantidade é 50
```

## Introdução

---

- Para remover a função criada, usa-se a instrução:

**DROP FUNCTION func\_escopo();**

## Declaração de Variáveis

- Todas as variáveis utilizadas em um bloco devem ser declaradas na seção de declaração do mesmo.
  - Exceção: variável de controle do FOR interagindo sobre um intervalo de valores inteiros, que é automaticamente declarada como sendo do tipo inteiro.
- As variáveis da linguagem podem possuir qualquer tipo de dado da linguagem SQL (*integer*, *varchar*, *char*, ...).
- Alguns exemplos simples de declaração de variáveis são:

```
id_funcionario    integer;  
quantidade        numeric(3);  
url               varchar;  
minha_linha       nome_da_tabela%ROWTYPE;  
meu_campo         nome_da_tabela.nome_da_coluna%TYPE;  
uma_linha         RECORD;
```



## Declaração de Variáveis

- A sintaxe geral para declaração de variáveis é:

*nome* [CONSTANT] *tipo* [NOT NULL] [{ DEFAULT | := } *expressão*];

Tal opção impede que seja atribuído valor a variável; assim, seu valor permanece constante pela duração do bloco.

A variável deve ter um valor-padrão não nulo especificado. Uma atribuição de valor nulo no bloco resulta em um erro em tempo de execução.

A cláusula DEFAULT, se fornecida, especifica o valor inicial da variável quando o processamento entra no bloco; caso contrário, a variável é inicializada com o valor nulo da SQL.

## Declaração de Variáveis

- O valor-padrão, uma vez definido, é atribuído à variável sempre que um bloco é inicialmente executado.
  - Por exemplo, atribuir `now()` a uma variável do tipo *timestamp* faz com que a variável possua a data e hora da chamada corrente à função, e não de quando a função foi pré-compilada.
- Alguns exemplos de declaração de variáveis são:

```
quantidade      integer DEFAULT 32;  
url             varchar := 'http://meu-site.com';  
id_funcionario  CONSTANT integer := 10;
```

## Parâmetros de Função

- Os parâmetros passados para as funções recebem, como nomes, os identificadores \$1, \$2, \$3, etc.
  - Para melhorar a legibilidade do código, podem ser declarados aliases para os nomes dos parâmetros.
  - Para fazer referência ao valor do parâmetro na função, pode ser utilizado tanto o alias quanto o identificador numérico.
- Uma maneira de criar um alias é fornecer um nome ao parâmetro no próprio cabeçalho da função.

```
CREATE FUNCTION taxa_venda(total real) RETURNS real AS $$  
BEGIN  
    RETURN total * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

## Parâmetros de Função

- Outra maneira é declarar explicitamente um alias na seção de declaração da função utilizando a sintaxe:

**nome** ALIAS FOR **\$n**;

```
CREATE FUNCTION taxa_venda(real) RETURNS real AS $$  
DECLARE  
    total ALIAS FOR $1;  
BEGIN  
    RETURN total * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

## Parâmetros de Função

```
-- Outro exemplo usando aliases explícitos
```

```
CREATE FUNCTION teste(vvarchar, integer) RETURNS integer AS $$  
DECLARE  
    cadeia ALIAS FOR $1;  
    numero ALIAS FOR $2;  
BEGIN  
    -- algum processamento qualquer  
END;  
$$ LANGUAGE plpgsql;
```

# Parâmetros de Função

-- Exemplo envolvendo tabelas reais de um banco de dados

```
CREATE FUNCTION concatenar_campos(tupla departamento)
```

```
RETURNS text AS $$
```

```
BEGIN
```

```
    RETURN tupla.id_depto || '---' || tupla.nomedepto;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

Operador de  
concatenação

Tuplas da relação departamento estão  
sendo passadas para o parâmetro  
"tupla" do tipo "departamento"

-- Ativação da função concatenar\_campos

```
SELECT concatenar_campos(departamento.*) FROM departamento;
```

```
SELECT concatenar_campos(t.*) FROM departamento t;
```

Retorno da função:

Output pane	
Data Output Explain Messages History	
	concatenar_campos_selecionados text
1	1---Pesquisa
2	2---Administracao
3	3---Construcao

## Parâmetros de Função

---

- Quando o tipo retornado por uma função é declarado como sendo de um tipo polimórfico (*anyelement* ou *anyarray*), é criado um parâmetro especial denominado \$0.
  - O parâmetro \$0 é inicializado como nulo, pode ser modificado pela função e é utilizado para armazenar o valor a ser retornado.
  - Pode ser criado, se desejado, um alias para o parâmetro \$0.
  - O tipo de dado real retornado pela função é deduzido a partir dos tipos de dado dos parâmetros de entrada.

## Parâmetros de Função

```

CREATE FUNCTION somar_tres_valores(v1 anyelement, v2
anyelement, v3 anyelement) RETURNS anyelement AS $$
DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função somar_valores
SELECT somar_tres_valores(10,20,30);
SELECT somar_tres_valores(1.1,2.2,3.3);

```

Somar\_tres\_valores  
-----  
60

Somar\_tres\_valores  
-----  
6.6

- A função somar\_tres\_valores funciona com qualquer tipo de dado que permita o operador +.



## Cópia de Tipo - %TYPE

- Na declaração de variáveis, a expressão **%TYPE** fornece o tipo de dado de uma variável já declarada ou da coluna de uma relação.
  - Geralmente, é utilizado para declarar variáveis que armazenam valores do banco de dados.
- Exemplos:

```
id_depto      departamento.id_depto%TYPE;  
quantidade    numeric(3);  
quantidade2   quantidade%TYPE;
```

Representa o tipo de dado da variável "quantidade"

Representa o tipo de dado da coluna "id\_depto" da relação "departamento"

## Cópia de Tipo - %TYPE

---

- Utilizando %TYPE, não é necessário conhecer o tipo de dado do item sendo referenciado e, se tal tipo de dado mudar no futuro, não será necessário mudar a definição na função.
- A expressão %TYPE é particularmente útil em funções polimórficas, uma vez que os tipos de dado das variáveis internas podem mudar de uma chamada para outra.
  - Ademais, podem ser criadas variáveis apropriadas aplicando %TYPE aos parâmetros.

## Tipo Linha - %ROWTYPE

- Uma variável do tipo-linha armazena toda uma linha de resultado de um comando SELECT ou FOR, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável.
  - Os campos individuais de uma variável do tipo-linha são acessados utilizando a notação usual "variável.campo".
- Uma variável do tipo-linha é declarada como tendo o tipo de dado das tuplas de uma relação ou de uma visão existente, por meio da notação **nome\_da\_tabela%ROWTYPE**. Ex.:

```
tupla      funcionario%ROWTYPE;
```

Representa o tipo de dado da relação "funcionario"

# Tipo Linha - %ROWTYPE

```
CREATE FUNCTION encontrar_gerentes(tuplad departamento)
RETURNS text AS $$
DECLARE
    tuplaf funcionario%ROWTYPE;
BEGIN
    SELECT * INTO tuplaf FROM funcionario where
                                tuplad.id_gerente = id_func;
    RETURN tuplad.nomedepcto || '---' || tuplaf.nomefunc;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função encontrar_gerentes
SELECT encontrar_gerentes(t.*) FROM departamento t;
```

Armazena em "tuplaf" a linha do funcionário que é gerente do departamento em questão ("tuplad")

Retorno da função:

Output pane	
Data Output	Explain Messages History
	encontrar_gerentes text
1	Pesquisa---Frank Santos
2	Administracao---Jaime Mendes
3	Construcao---Junia Mendes

## Tipo Registro - RECORD

---

- As variáveis do tipo registro, declaradas pela notação "**nome** RECORD", são semelhantes às variáveis do tipo-linha, mas não possuem uma estrutura pré-definida.
  - Uma variável registro assume a estrutura da linha que lhe é atribuída por meio dos comandos SELECT e FOR.
  - Uma atribuição à uma variável registro já preenchida faz com que tal variável assuma a estrutura da nova linha que lhe é atribuída.
  - Antes de ser utilizada em uma atribuição, a variável registro não possui estrutura; assim, qualquer tentativa de acessar um de seus campos produz um erro em tempo de execução.

# Tipo Registro - RECORD

```

CREATE FUNCTION encontrar_departamento(func funcionario.id_func%TYPE)
                                RETURNS text AS $$
DECLARE
    reg RECORD; depto funcionario.id_depto%TYPE;
BEGIN
    SELECT INTO reg * FROM funcionario WHERE id_func = func;
    IF NOT FOUND THEN --após SELECT INTO, a variável especial FOUND
                        --retorna falso se nenhum registro foi armazenado
        RAISE EXCEPTION 'Empregado % não encontrado', func;
    ELSE
        depto := reg.id_depto;
        SELECT INTO reg * FROM departamento WHERE id_depto = depto;
        RETURN reg.id_depto || '---' || reg.nomedepto;
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função encontra_departamento
SELECT encontra_departamento(1);

```

Armazena em "reg" o registro do funcionário de código "func"

Armazena em "reg" o registro do departamento de código "depto"

## Avaliação de Comandos SQL

- Os comandos SQL, tais como "SELECT expressão", usados em instruções da PL/pgSQL são processados pelo gerenciador da Interface de Programação do Servidor (SPI) do SGBD.
- Antes da avaliação de um comando SQL, as ocorrências de variável da PL/pgSQL são substituídas por parâmetros e os valores verdadeiros de tais variáveis são passados para a matriz de parâmetros do componente executor.
  - Isto permite que o plano de um comando SQL seja preparado uma só vez e, depois, reutilizado nas avaliações seguintes.
- A avaliação feita de comandos SQL pelo analisador principal do PostgreSQL pode produzir alguns efeitos colaterais na interpretação de valores constantes. Por exemplo, há diferença na execução das funções seguintes?

# Avaliação de Comandos SQL

```
create table logtable (  
    operacao VARCHAR(30) NOT NULL,  ts timestamp NOT NULL,  
    CONSTRAINT pk_log PRIMARY KEY (operacao, ts)  
);  
  
CREATE FUNCTION insere_log_1(op text) RETURNS timestamp AS $$  
BEGIN  
    INSERT INTO logtable VALUES (op, 'now');  RETURN 'now';  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE FUNCTION insere_log_2(op text) RETURNS timestamp AS $$  
DECLARE  
    ctime timestamp := 'now';  
BEGIN  
    INSERT INTO logtable VALUES (op, ctime);  RETURN ctime;  
END;  
$$ LANGUAGE plpgsql;
```



## Avaliação de Comandos SQL

- No caso da função `insere_log_1`, o analisador principal do PostgreSQL, ao analisar o comando `INSERT`, interpreta a cadeia de caracteres `'now'` como *timestamp* já que a coluna de destino na tabela `logtable` é deste tipo.
  - Assim, o analisador cria uma constante, a partir de `'now'`, contendo a data e hora da análise; tal constante é utilizada em todas as chamadas da função durante a sessão.
- No caso da função `insere_log_2`, o analisador principal do PostgreSQL retorna um valor do tipo *text* contendo a cadeia de caracteres `'now'` já que desconhece o tipo que `'now'` deve ser.
  - Nas atribuições à variável local `ctime`, o interpretador do PL/pgSQL converte a cadeia de caracteres `'now'` para o tipo *timestamp*; assim, a data e a hora são atualizadas a cada execução da função.

## Instruções Condicionais

- As instruções IF permitem executar comandos a partir da avaliação de condições. As formas de IF mais comuns são:

```
IF expressão_logica  
THEN  
    instruções  
END IF;
```

```
-----
```

```
IF expressão_logica  
THEN  
    instruções  
ELSE  
    instruções  
END IF;
```

```
IF expressão_logica THEN  
    instruções  
[ ELSIF expressão_logica  
  THEN  
    instruções  
  ... ]  
[ ELSE  
    instruções ]  
END IF;
```

# Instruções Condicionais

---

```
-- Validação de um número
IF numero = 0 THEN
    resultado := 'zero';
ELSIF numero > 0 THEN
    resultado := 'positivo';
ELSIF numero < 0 THEN
    resultado := 'negativo';
ELSE
    -- Outra possibilidade é o número nulo
    resultado := 'NULL';
END IF;
```

# Instrução LOOP

---

```
[<<rótulo>>]  
LOOP  
    instruções  
END LOOP;
```

- A instrução LOOP define um laço incondicional, repetido indefinidamente até ser finalizado por uma instrução EXIT ou RETURN.
  - Nos laços aninhados, pode ser utilizado um rótulo opcional na instrução EXIT para especificar o nível de aninhamento (ou seja, o LOOP) que deve ser finalizado.

## Instrução EXIT

```
EXIT [<<rótulo>>] [ WHEN expressão_lógica ] ;
```

- A instrução EXIT finaliza qualquer tipo de laço e blocos BEGIN-END.
- Quando a cláusula WHEN está presente, a saída do laço ocorre somente se a condição especificada na mesma for verdadeira; caso contrário, o controle passa para a instrução seguinte ao EXIT.
- Nos laços aninhados:
  - se não for especificado nenhum rótulo, o comando EXIT finaliza o laço mais interno e a primeira instrução após o END LOOP do mesmo será executada;
  - se o rótulo for especificado, o comando EXIT finaliza o laço corresponde ao rótulo e a primeira instrução após o END LOOP do laço finalizado será executada.

## Exemplos LOOP e EXIT

LOOP

-- algum processamento

IF contador > 0 THEN

EXIT; -- finalização do laço

END IF;

END LOOP;

LOOP

-- algum processamento

EXIT WHEN contador > 0; -- mesmo resultado do anterior

END LOOP;

BEGIN

-- algum processamento

IF quantidade > 100000 THEN

EXIT; -- finalização do bloco

END IF;

END;

# Instrução WHILE

---

```
[<<rótulo>>]  
WHILE expressão_lógica LOOP  
    instruções  
END LOOP;
```

- A instrução WHILE repete uma sequência de instruções (corpo do laço) enquanto a expressão lógica condicional definida for verdadeira.
  - A condição é verificada antes de cada entrada no corpo do laço.

# Instrução WHILE

---

```
WHILE quantidade > 0 AND quantidade < 1000 LOOP  
    -- algum processamento  
END LOOP;
```

```
WHILE NOT (quantidade <= 0 ) LOOP  
    -- algum processamento  
END LOOP;
```



# Instrução FOR

```
[<<rótulo>>]  
FOR nome IN [ REVERSE ] expressão .. expressão LOOP  
    instruções  
END LOOP;
```

- A instrução FOR cria um laço que interage num intervalo de valores inteiros.
  - A variável de controle **nome** é definida automaticamente como sendo do tipo integer, e somente existe dentro do laço.
  - As duas expressões que fornecem os limites inferior e superior do intervalo são avaliadas somente uma vez, ao iniciar o laço.
  - O passo da interação é 1 mas, quando a cláusula REVERSE é especificada, torna-se -1.
  - Se o limite inferior for maior do que o limite superior (ou menor, no caso do uso da cláusula REVERSE), o corpo do laço não é executado nenhuma vez e nenhum erro é gerado.

# Instrução FOR

---

```
FOR i IN 1..(quantidade*2) LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;
```

## Laço – Resultado de uma Consulta

```
[<<rótulo>>]  
FOR registro_ou_linha IN comando LOOP  
    instruções  
END LOOP;
```

- Por meio da instrução FOR acima, é possível manipular os dados retornados (tuplas) por uma consulta.
  - Cada linha de resultado do comando (que deve ser, necessariamente, um SELECT) é atribuída à variável do tipo registro ou do tipo-linha especificada.
  - O corpo do laço é executado uma vez para cada linha de resultado do comando.

## Laço – Resultado de uma Consulta

```
CREATE FUNCTION numero_de_employees
    (depto departamento.nomedeppto%TYPE) RETURNS integer AS $$
DECLARE
    reg RECORD;  contador integer := 0;
BEGIN
    SELECT INTO reg * FROM departamento WHERE nomedeppto = depto;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Departamento % não encontrado', depto;
    ELSE
        for reg IN SELECT * FROM funcionario
            WHERE id_depto = reg.id_depto ORDER BY nomefunc LOOP
            contador := contador + 1;
            RAISE NOTICE '%) % : %', contador, reg.nomefunc, reg.endereco;
            PERFORM insere_log_2('leitura do funcionario ' || reg.id_func);
        END LOOP;
        RETURN contador;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

A instrução PERFORM executa o comando especificado, desprezando algum resultado, caso haja.

## Laço – Resultado de uma Consulta

```
[<<rótulo>>]  
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP  
    instruções  
END LOOP;
```

- A instrução FOR-IN-EXECUTE é outra forma de manipular dados retornados (tuplas) por uma consulta.
  - Esta forma é semelhante à forma anterior, exceto que o código-fonte da instrução SELECT é especificado como uma expressão alfanumérica (cadeia de caracteres), sendo avaliada e replanejada a cada entrada no laço.
  - Assim, é possível escolher entre a velocidade da consulta pré-planejada e a flexibilidade da consulta dinâmica.

## Captura de Erros

- Por padrão, qualquer erro que ocorra em uma função interrompe a execução da mesma e da transação em questão.
  - Para capturar erros e se recuperar dos mesmos, utiliza-se a cláusula EXCEPTION dentro de um bloco ou sub-bloco BEGIN-END.

```
[<<rótulo>>]
BEGIN
    instruções
EXCEPTION
    WHEN condição [OR condição ...] THEN
        instruções_do_tratador
    [ WHEN condição [OR condição ...] THEN
        instruções_do_tratador
    ... ]
END;
```

## Captura de Erros

- Caso não ocorra erro em um bloco ou sub-bloco, todas as instruções do mesmo são executadas e o controle passa para a instrução seguinte ao END.
- Caso ocorra algum erro em um bloco ou sub-bloco, o processamento das instruções do mesmo é abandonado e o controle passa para a cláusula EXCEPTION; nesse momento, é feita a procura da condição, por meio das instruções WHEN, correspondente ao erro ocorrido.
  - Se for encontrada uma correspondência, as instruções referentes são executadas e o controle passa para a instrução seguinte ao END.
  - Se não for encontrada nenhuma correspondência, o erro se propaga para fora como se a cláusula EXCEPTION não existisse: o erro pode ser capturado por um bloco envoltório contendo EXCEPTION e, se não houver nenhum, o processamento da função é interrompido.

## Captura de Erros

- O nome da condição pode ser qualquer um dos erros definidos pela PL/pgSQL. Alguns exemplos são:

22008	DATETIME FIELD OVERFLOW
22012	DIVISION BY ZERO
22005	ERROR IN ASSIGNMENT
2200B	ESCAPE CHARACTER CONFLICT
22022	INDICATOR OVERFLOW
22015	INTERVAL FIELD OVERFLOW

- O nome de condição especial OTHERS representa qualquer erro.
- Caso ocorra um novo erro nas instruções `_do_` tratador, este não pode ser capturado pela cláusula `EXCEPTION` em questão, mas é propagado para fora do bloco ou sub-bloco; nesse caso, uma cláusula `EXCEPTION` envoltória, caso exista, pode capturá-lo.
- Quando um erro é capturado pela cláusula `EXCEPTION` de um bloco, todas as modificações feitas no banco de dados dentro do mesmo são desfeitas.



# Captura de Erros

```
CREATE FUNCTION altera_salario() RETURNS integer AS $$  
BEGIN  
    UPDATE funcionario SET salario = salario * 1.1;  
    -- sub-bloco  
    DECLARE  
        x integer;  
    BEGIN  
        UPDATE funcionario SET salario = 5000;  
        -- geração do erro propositalmente  
        x := 1/0;  
        RETURN 1;  
    EXCEPTION  
        WHEN division_by_zero THEN RAISE NOTICE 'Divisão por zero';  
        RETURN 0;  
    END;  
END;  
$$ LANGUAGE plpgsql;
```

O erro é capturado pela cláusula EXCEPTION do sub-bloco, desfazendo a 2ª alteração do salário. A 1ª alteração não é desfeita.

## Gatilho

---

- A linguagem PL/pgSQL pode ser utilizada para definir gatilhos, por meio da criação do próprio gatilho (comando CREATE TRIGGER) e da função de gatilho a ser disparada.
  - A função de gatilho é criada pelo comando CREATE FUNCTION, não apresentando argumentos e retornando o tipo *trigger*.
  - A função deve ser declarada sem argumentos, mesmo que espere receber os argumentos especificados no comando CREATE TRIGGER correspondente: os argumentos para a função de gatilho são passados através da variável especial TG\_ARGV.

# Gatilho

- Quando uma função escrita em PL/pgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente, a saber:

Variável	Tipo	Descrição
NEW	RECORD	Contém a nova linha do banco de dados, para as operações de INSERT/UPDATE nos gatilhos no nível de linha; nos gatilhos no nível de instrução, contém o valor NULL.
OLD	RECORD	Contém a antiga linha do banco de dados, para as operações de UPDATE/DELETE nos gatilhos no nível de linha; nos gatilhos no nível de instrução, contém o valor NULL.
TG_NAME	name	Contém o nome do gatilho disparado.
TG_WHEN	text	Informa se o gatilho ocorre antes ou depois da efetivação da instrução disparadora do mesmo: cadeia de caracteres contendo BEFORE ou AFTER.
TG_LEVEL	text	Informa o nível do gatilho, podendo ser linha ou instrução: cadeia de caracteres contendo ROW ou STATEMENT.

# Gatilho

---

Variável	Tipo	Descrição
TG_OP	text	Informa para qual operação o gatilho foi disparado: cadeia de caracteres contendo INSERT, UPDATE, ou DELETE.
TG_RELID	oid	Contém o ID de objeto da tabela que causou o disparo do gatilho.
TG_RELNAME	name	Contém o nome da tabela que causou o disparo do gatilho.
TG_NARGS	integer	Contém o número de argumentos fornecidos à função de gatilho pela instrução CREATE TRIGGER.
TG_ARGV[]	Vetor de text	Contém os próprios argumentos da instrução CREATE TRIGGER fornecidos à função de gatilho. O contador do índice começa de 0. Índices inválidos (menor que 0 ou maior ou igual a TG_NARGS) resultam em um valor nulo.

## Gatilho

---

- Uma função de gatilho deve retornar nulo ou um valor do tipo registro/tipo-linha da tabela para a qual o gatilho foi disparado.
- As funções de gatilho no nível de linha, disparados antes (BEFORE) da efetivação da instrução causadora do gatilho, podem retornar nulo, no intuito de sinalizar ao gerenciador do gatilho para pular o restante da operação para esta linha.
  - Assim, os gatilhos posteriores não serão disparados e não ocorrerá o INSERT/UPDATE/DELETE para esta linha.

## Gatilho

---

- Se a função de gatilho no nível de linha, disparado antes (BEFORE) da efetivação da instrução causadora do gatilho, retornar um valor diferente de nulo, a operação prossegue com este valor de linha.
  - Se um valor de linha diferente do valor original de NEW for retornado, a linha que será inserida ou atualizada é alterada.
  - Neste caso, para alterar a linha a ser armazenada na função de gatilho, é possível substituir valores individuais diretamente em NEW e retornar o NEW modificado, ou construir um novo registro/linha completo a ser retornado.
- O valor retornado por um gatilho AFTER no nível de linha, ou por um gatilho BEFORE ou AFTER no nível de instrução, é sempre ignorado já que pode muito bem ser nulo.

## Gatilho – Exemplo 01

---

```
/* Objetivo do gatilho: na inserção ou atualização de uma
tupla na tabela de empregados "emp", (a) garantir a validade
da tupla e (b) registrar o usuário que efetuou a operação e o
momento da mesma. */
```

```
-- Criação da tabela de empregados "emp"
```

```
CREATE TABLE emp (
    nome_emp      text,
    salario       integer,
    ultima_data    timestamp,
    ultimo_usuario text
);
```

# Gatilho – Exemplo 01

```
CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
BEGIN
    -- (a) Validação do nome e do salário do empregado
    IF NEW.nome_emp IS NULL THEN
        RAISE EXCEPTION 'Nome do empregado nulo';
    END IF;
    IF NEW.salario IS NULL THEN
        RAISE EXCEPTION 'Salário nulo de %', NEW.nome_emp;
    END IF;
    IF NEW.salario < 0 THEN
        RAISE EXCEPTION 'Salário negativo de %', NEW.nome_emp;
    END IF;

    -- (b) Registro do usuário e do momento
    NEW.ultima_data := 'now';
    NEW.ultimo_usuario := current_user;
    RETURN NEW;
END;
$emp_gatilho$ LANGUAGE plpgsql;
```

Variável especial que armazena o usuário corrente do PostgreSQL.



## Gatilho – Exemplo 01

-- Criação do gatilho propriamente dito

```
CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp  
FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();
```

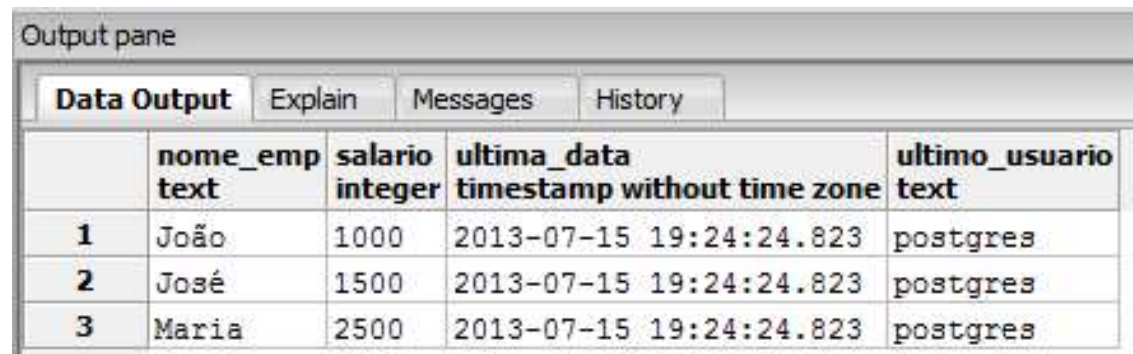
-- Inserção de três empregados na tabela "emp"

```
INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);  
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);  
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);
```

-- Consulta na tabela "emp"

```
SELECT * FROM emp;
```

Retorno da  
consulta:



The screenshot shows a database interface with an 'Output pane' at the top. Below the pane title are four tabs: 'Data Output' (selected), 'Explain', 'Messages', and 'History'. The 'Data Output' tab displays a table with five columns: an index column, 'nome\_emp' (text), 'salario' (integer), 'ultima\_data' (timestamp without time zone), and 'ultimo\_usuario' (text). The table contains three rows of data, numbered 1, 2, and 3 in the index column.

	nome_emp text	salario integer	ultima_data timestamp without time zone	ultimo_usuario text
1	João	1000	2013-07-15 19:24:24.823	postgres
2	José	1500	2013-07-15 19:24:24.823	postgres
3	Maria	2500	2013-07-15 19:24:24.823	postgres

## Gatilho – Exemplo 02

```
/* Objetivo do gatilho: garantir que todas as inserções, as atualizações e as exclusões de tuplas em uma tabela de empregados sejam registradas em uma tabela de auditoria, no intuito de que, futuramente, as operações efetuadas na tabela de empregados possam ser auditadas quando necessário. */
```

```
-- Criação das tabelas de empregados e de auditoria
```

```
CREATE TABLE emp2 (  
    nome_emp text NOT NULL,    salario integer  
);
```

```
CREATE TABLE emp_audit (  
    operacao      char(1)    NOT NULL, -- operação de alteração  
    usuario       text       NOT NULL, -- usuário promotor  
    data          timestamp  NOT NULL, -- data da alteração  
    nome_emp      text       NOT NULL, -- nome do empregado  
    salario       integer    -- salário do empregado  
);
```

## Gatilho – Exemplo 02

```
CREATE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
    -- Cria uma linha na tabela emp_audit para refletir a operação
    -- realizada na tabela emp2, usando a variável especial TG_OP
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'E', current_user, now(), OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'A', current_user, now(), NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', current_user, now(), NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- o resultado é ignorado já que é um gatilho AFTER
END;
$emp_audit$ language plpgsql;
```

## Gatilho – Exemplo 02

```
-- Criação do gatilho propriamente dito
CREATE TRIGGER emp_audit AFTER INSERT OR UPDATE OR DELETE ON emp2
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

-- Operações de alteração da tabela "emp2"
INSERT INTO emp2 (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp2 (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp2 (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp2 SET salario = 2500 WHERE nome_emp = 'Maria';
DELETE FROM emp2 WHERE nome_emp = 'João';

-- Consulta na tabela "emp_audit"
SELECT * FROM emp_audit;
```

Retorno da  
consulta:

Output pane					
Data Output Explain Messages History					
	operacao character(1)	usuario text	data timestamp without time zone	nome_emp text	salario integer
1	I	postgres	2013-07-17 14:55:53.964	João	1000
2	I	postgres	2013-07-17 14:55:53.964	José	1500
3	I	postgres	2013-07-17 14:55:53.964	Maria	250
4	A	postgres	2013-07-17 14:55:53.964	Maria	2500
5	E	postgres	2013-07-17 14:55:53.964	João	1000

## Gatilho – Exemplo 03

```
/* Objetivo do gatilho: garantir que as atualizações realizadas nas
colunas da tabela de empregados sejam registradas em uma tabela de
auditoria, permitindo a auditoria futura das colunas e não apenas das
tuplas (obs: não é permitido atualizar a chave do empregado).*/
```

```
-- Criação das tabelas de empregados e de auditoria
```

```
CREATE TABLE emp3 (  
    id          serial PRIMARY KEY,  
    nome_emp    text    NOT NULL,    salario integer  
);
```

```
CREATE TABLE emp_audit_col (  
    usuario      text    NOT NULL, -- usuário promotor  
    data         timestamp NOT NULL, -- data da atualização  
    id           integer  NOT NULL, -- id do empregado  
    coluna       text    NOT NULL, -- coluna atualizada  
    valor_antigo text    NOT NULL, -- valor antigo da coluna  
    valor_novo   text    NOT NULL  -- valor novo da coluna  
);
```

## Gatilho – Exemplo 03

```
CREATE FUNCTION processa_emp_audit_col() RETURNS TRIGGER AS $emp_aud$
BEGIN
    -- Não permite atualizar a chave primária
    IF (NEW.id <> OLD.id) THEN
        RAISE EXCEPTION 'Não é permitido atualizar o campo ID';
    END IF;
    -- Atualiza emp_audit_col para refletir as alterações de emp3
    IF (NEW.nome_emp <> OLD.nome_emp) THEN
        INSERT INTO emp_audit_col SELECT current_user, current_timestamp,
                                         NEW.id, 'nome_emp', OLD.nome_emp, NEW.nome_emp;
    END IF;
    IF (NEW.salario <> OLD.salario) THEN
        INSERT INTO emp_audit_col SELECT current_user, current_timestamp,
                                         NEW.id, 'salario', OLD.salario, NEW.salario;
    END IF;
    RETURN NULL; -- o resultado é ignorado já que é um gatilho AFTER
END;
$emp_aud$ language plpgsql;
```

## Gatilho – Exemplo 03

```
-- Criação do gatilho propriamente dito
CREATE TRIGGER emp_audit_col AFTER UPDATE ON emp3
FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit_col();

-- Operações de alteração da tabela "emp3"
INSERT INTO emp3 (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp3 (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp3 (nome_emp, salario) VALUES ('Maria',2500);
UPDATE emp3 SET salario = 2500 WHERE id = 2;
UPDATE emp3 SET nome_emp = 'Maria Cecília' WHERE id = 3;
UPDATE emp3 SET id=100 WHERE id=1;

-- Consulta na tabela "emp_audit_col"
SELECT * FROM emp_audit_col;
```

ERRO: Não é permitido atualizar o campo ID.

Retorno da  
consulta:

Output pane						
Data Output Explain Messages History						
	usuario text	data timestamp without time zone	id integer	coluna text	valor_antigo text	valor_novo text
1	postgres	2013-07-17 16:03:29.381	2	salario	1500	2500
2	postgres	2013-07-17 16:03:29.381	3	nome emp	Maria	Maria Cecília