

Triggers



<https://www.postgresql.org/docs/>

Revisão sobre funções

Quero construir uma função capaz de aumentar o salário em 10% de todos os empregados da empresa a cada vez que for acionada.

Revisão sobre funções

```
CREATE OR REPLACE FUNCTION increase_salary()  
RETURNS VOID  
AS $$  
    DECLARE  
        employee_ RECORD;  
    BEGIN  
        RAISE NOTICE 'Aumento de 10% para todos os empregados', '%';  
        FOR employee_ IN SELECT * FROM employee LOOP  
            UPDATE employee  
            set salary = employee_.salary + employee_.salary / 10  
            WHERE id = employee_.id;  
        END LOOP;  
        RETURN;  
    END;  
$$ LANGUAGE plpgsql;
```

Revisão sobre funções

```
CREATE FUNCTION aumenta_salario_10() returns void AS
```

```
$$
```

```
BEGIN
```

```
    UPDATE employee SET salario = salario + (salario * 0.10) WHERE id >
```

```
0;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

Triggers

- Gatilhos são execuções disparadas pelo banco em função de EVENTOS que ocorrem
- Um evento ocorre
 - Em uma tabela
 - De acordo com uma operação, por DML (INSERT, UPDATE ou DELETE)
 - Antes ou depois (AFTER ou BEFORE)

Exemplo 1-a

Objetivo do trigger: Inserir um histórico de atualizações na tabela historico

```
CREATE TABLE emp (  
    empname text,  
    salary integer  
);  
CREATE TABLE historico(  
    data timestamp,  
    salary numeric,  
    empname text  
);
```

```
insert into emp(empname, salary) values ('joao',1000);
```

Exemplo 1-a

```
CREATE or REPLACE FUNCTION emp_time()  
RETURNS trigger  
AS $$  
    BEGIN  
        INSERT INTO historico VALUES (now(), New.salary, New.empname) ;  
        raise notice 'valor %', new.salary;  
        RETURN NEW;  
    END;  
$$ LANGUAGE plpgsql;
```

Exemplo 1-a

```
CREATE TRIGGER emp_time  
AFTER INSERT OR UPDATE  
ON emp  
FOR each row EXECUTE PROCEDURE  
emp_time();
```

```
insert into emp(empname, salary) values ('joao',1000);
```


Triggers: formato

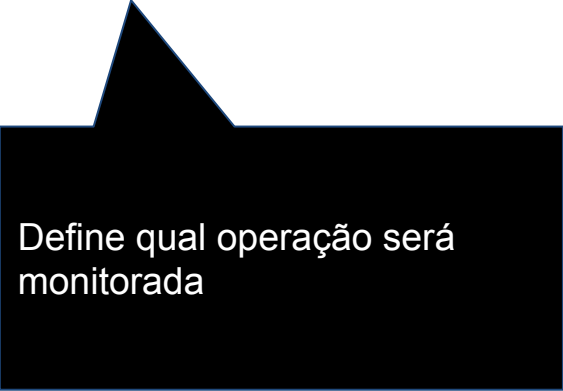
CREATE TRIGGER nome { **BEFORE** | **AFTER** }



Antes ou depois de executar
sobre a tabela

Triggers: formato

CREATE TRIGGER nome { **BEFORE** | **AFTER** } {**INSERT** | **UPDATE**| **DELETE**}



Define qual operação será monitorada

Triggers: formato

```
CREATE TRIGGER nome { BEFORE | AFTER } {INSERT | UPDATE| DELETE}  
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]
```



Define o escopo de execução

Triggers: formato

```
CREATE TRIGGER nome { BEFORE | AFTER } {INSERT | UPDATE| DELETE}  
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]  
EXECUTE PROCEDURE nome_da_função ( argumentos )
```



Nome da função

Triggers

- **Identificadores de correlação** – variáveis de vínculo PL/SQL
 - sempre vinculados à tabela desencadeadora do trigger

instrução	old	new
INSERT	NULL	valores que serão inseridos
UPDATE	valores antes da atualização	novos valores para a atualização
DELETE	valores antes da remoção	NULL

Exemplo: Old vs New

Considere os valores para cada execução de acordo com a tabela abaixo:

Nro	TipoVinc	Nome	Salario
-----	----------	------	---------

```
INSERT INTO table VALUES  
(21, 2, 'João', 2000),  
(22, 1, 'gilberto', 3000),  
(10, 3, 'Maria', 5000)
```

Quais são os valores de OLD e NEW para essa operação?

Exemplo: Old vs New

O que acontece com os valores considerando a operação:

Nro	TipoVinc	Nome	Salario
21	2	João	2000
22	1	Gilberto	3000
10	3	Alfredo	5000

```
UPDATE table SET salario = salario*1.3 WHERE TipoVinc=2;
```

Quais são os valores de OLD e NEW para cada a variável New.salario e Old.salario?

Exemplo: Old vs New

Os valores para cada execução considerando a operação serão:

Nro	TipoVinc	Nome	Salario
21	2	João	2000
22	1	Gilberto	3000
10	3	Alfredo	5000

```
DELETE FROM table WHERE Nro > 20;
```

Quais são os valores de OLD e NEW para cada a operação?

Exemplo 1-b

Objetivo do trigger: criar os campos 'date' e 'user' para inserção dos dados. Caso o salário ou o nome seja nulo deve ser disparado uma exceção.

```
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,  
    last_user text  
);
```

```
insert into emp(empname, salary) values ('joao',1000);
```

Exemplo 1-b

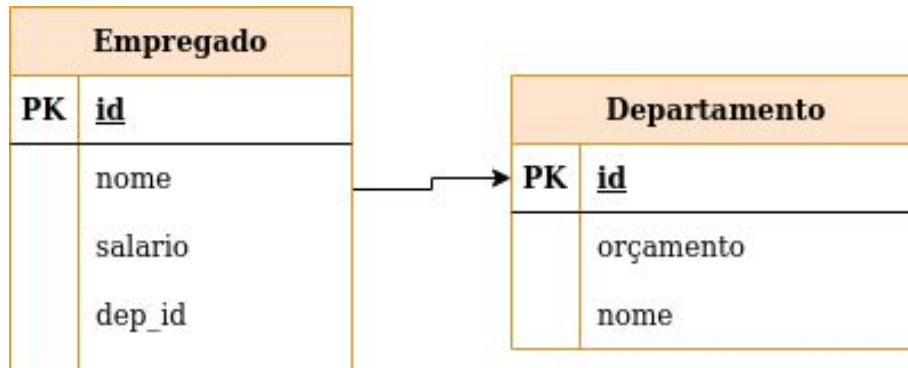
```
CREATE OR REPLACE FUNCTION emp_time()
RETURNS trigger AS $$
BEGIN
    IF NEW.empname IS NULL THEN
        RAISE EXCEPTION 'empname nao pode ser nulo';
    END IF;
    IF NEW.salary IS NULL THEN
        RAISE EXCEPTION 'Não pode ter salario nulo';
    END IF;
    NEW.last_date := current_date;
    NEW.last_user := current_user;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

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

insert into emp values ('joao', 1000);
```

Exemplo 1-c

Vamos implementar um trigger para que a cada nova inserção de empregado somente seja realizada se o orçamento do departamento permitir



Exemplo 1-c

```
create table departamento (id integer primary key,  
                           orçamento float,  
                           nome varchar(50));
```

```
create table empregado (id integer primary key,  
                        nome varchar(50),  
                        salario float,  
                        dep_id integer,  
                        CONSTRAINT fk_customer  
                        FOREIGN KEY(dep_id)  
                        REFERENCES departamento(id));
```

Exemplo 1-c

```
CREATE FUNCTION check_orcamento()
  RETURNS TRIGGER AS $$
DECLARE
  orc_permitido float;
  orc_gasto float;
BEGIN
  SELECT orcamento INTO orc_permitido FROM departamento WHERE id = NEW.dep_id;

  SELECT INTO orc_gasto SUM(salario) FROM empregado WHERE dep_id = NEW.dep_id;
  IF (orc_gasto+NEW.salario) > orc_permitido
  THEN
    RAISE EXCEPTION 'Orçamento acima do teto [id:%] by [%]', NEW.id, (orc_gasto - orc_permitido);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Exemplo 1-c

```
create trigger check_orc before insert on empregado for each row execute  
procedure check_orcamento();
```

Atividade A

```
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,  
    last_user text);
```

- 1) Na tabela **emp**, crie uma trigger para aumentar 10 % do salário bruto (devido a impostos) antes de adicionar o valor na tabela.
- 2) Sobre a tabela **emp**, crie uma trigger para evitar que os salários sejam aumentados em mais de 30%. Deve ser disparado uma exceção com o erro.
- 3) Crie uma tabela auxiliar para armazenar os históricos de salários. Ou seja, armazenar os salários antigos dos funcionários.

Variáveis implícitas

TG_NAME: tipo de dado NAME. Variável que contém o nome da trigger disparada.

TG_WHEN: tipo de dado TEXT. Contém os valores BEFORE ou AFTER dependendo da definição da trigger e de como a trigger foi disparada.

TG_LEVEL: tipo de dado TEXT. Contém ROW ou STATEMENT dependendo do tipo declarado da trigger.

TG_OP: tipo de dado TEXT. Contém a string 'INSERT', 'UPDATE' ou 'DELETE' indicando qual operação de atualização disparou a trigger.

TG_TABLE_NAME: nome da tabela que disparou a trigger.

Exemplo

```
CREATE TABLE emp (  
    empname      text,  
    salary       integer  
);
```

```
CREATE TABLE emp_audit(  
    operation     varchar(1) ,  
    stamp         timestamp,  
    username      text    ,  
    empname       text ,  
    salary        integer  
);
```

Explique o que acontecerá no cenário abaixo?

-

```
CREATE or replace FUNCTION process_emp_audit() RETURNS TRIGGER AS $$  
    BEGIN  
        OLD.empname='nao sei o que colocar';  
        RETURN OLD;  
    END;  
$$ LANGUAGE plpgsql;
```

```
CREATE or replace TRIGGER emp_audit before INSERT OR UPDATE OR DELETE ON emp FOR EACH ROW EXECUTE  
PROCEDURE process_emp_audit();
```

Exemplo

Na tabela de EMPREGADO abaixo, faça um trigger de auditoria que armazene as informações do empregado, bem como, o evento (insert, delete ou update) que disparou este trigger, o usuário responsável pela alteração e a data do sistema.

```
Empregado( id integer primary key, nome varchar(50), cpf varchar(15),  
Num_Departamento integer, Salario DECIMAL(10,2 ), Supervisor  
varchar(50));
```

```
Auditoria( id integer primary key, nome varchar(50), cpf varchar(15),  
Num_Departamento integer, Salario DECIMAL(10,2 ), Supervisor  
varchar(50), op char(2), data date, currentUser varchar(40))
```

evento=1 caso insert
evento=2 caso update
evento=3 caso delete

Exemplo

```
CREATE FUNCTION process_emp_audit()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF (TG_OP = 'DELETE') THEN  
        INSERT INTO Auditoria values (old.*, 'D', now(), current_user);  
        RETURN OLD;  
    ELSIF (TG_OP = 'UPDATE') THEN  
        INSERT INTO Auditoria values ( new.*, 'U', now(), current_user);  
        RETURN NEW;  
    ELSIF (TG_OP = 'INSERT') THEN  
        INSERT INTO Auditoria values ( new.*, 'I', now(), current_user);  
        RETURN NEW;  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER emp_audit  
AFTER INSERT OR UPDATE OR DELETE ON empregado  
FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Respostas...)

```
CREATE OR REPLACE FUNCTION emp_time()  
RETURNS trigger AS $$  
    BEGIN  
        *****  
        RETURN NEW;  
    END;  
$$ LANGUAGE plpgsql;
```

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

```
insert into emp values ('joao', 1000);
```

Atividade B

```
create table mesas(id int, lugares int, livre int, primary key (id));
```

```
create table reservas(nome varchar(50), npessoas int, nmesa int, CONSTRAINT fk_mesas  
FOREIGN KEY(nmesa) REFERENCES mesas(id));
```

Faça um trigger capaz de atribuir uma mesa automaticamente para uma reserva observando a necessidade de encontrar uma mesa que otimize o número de lugares ocupados pelo restaurante. Após alocar uma mesa, a variável ‘livre’ deve ser setada. Se não tiver mesa livre, deve ser retornada uma mensagem e não deve ser inserido na tabela reservas.

Exemplo de inserção dos dados:

```
insert into mesas values (1, 5, 1), (2,6,1),(3,7,1);
```

```
insert into reservas (nome, npessoas) values ('joao', 3);
```

Id da Mesa selecionada: 1

Atividade C

AvailableFlights(Num_flight int, date date, numberOfFreeSeats int, price float)

Bookings(Num_flight int, date date, passenger int, price float)

Este trigger deve cuidar do armazenamento de lugares livre de voos. O trigger é disparado quando ocorre uma inserção de um passageiro em um voo (Bookings). Os efeitos devem ser os seguintes:

- 1- Se o número de poltronas do voo for positivo (AvailableFlights), o mesmo deve ser decrementado;
- 2- Se não existir poltronas, a emissão do ticket falha;
- 3- Se a emissão do ticket for ok, é necessário incrementar o preço da passagem em 50 reais para o próximo passageiro

OBS: para inserir uma linha no booking antes deve-se inserir o voo na tabela

AvailableFlights

Criando tabelas

```
DROP TABLE IF EXISTS AvailableFlights;  
CREATE TABLE AvailableFlights (  
    Num_flight INTEGER,  
    date DATE,  
    numberOfFreeSeats INTEGER,  
    price FLOAT  
);
```

```
INSERT INTO AvailableFlights VALUES (0, now(), 2, 50);
```

```
-- Bookings(Num_flight int, date date, passenger int, price float)  
DROP TABLE IF EXISTS Bookings;  
CREATE TABLE Bookings (  
    Num_flight INTEGER,  
    date DATE,  
    passenger INTEGER,  
    price FLOAT  
);
```


Rules

- São execuções disparadas pelo banco em função de EVENTOS que ocorrem
- São mais simples que triggers para implementar no entanto limitadas para alteração de dados
- Triggers são disparadas a cada linha afetada. Já o operador de rules modificam a consulta ou produz uma nova consulta.

Rules

```
CREATE [ OR REPLACE ] RULE NomeDaRule AS ON evento  
    TO Tabela [ WHERE CondiçãoLógica]  
    DO [ ALSO | INSTEAD ] { NOTHING | comando | ( comando ; comando ... ) }
```

*evento= SELECT | INSERT | UPDATE | DELETE

Exemplo 1-a

Objetivo: criar os campos 'date' e 'user' para inserção dos dados

```
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,  
    last_user text  
);
```

```
insert into emp(empname, salary) values ('joao',1000);
```

Rules

```
create or replace rule a1 as  
  on insert to emp  
  where new.salary > 10000  
  do instead NOTHING;
```

```
insert into emp (empname, salary) values ('maria', 100000 );
```

```
select * from emp;
```

Pq não chamar de Procedimento?

Segundo a documentação do postgres:

“In the syntax of CREATE TRIGGER, the keywords FUNCTION and PROCEDURE are equivalent, **but the referenced function must in any case be a function, not a procedure.** The use of the keyword PROCEDURE here is historical and deprecated.”

Procedures

```
create procedure procedure_name()  
as $$  
declare  
    -- variable declaration  
begin  
    -- stored procedure body  
end;  
$$  
language plpgsql
```