Triggers no PostgreSQL

Traduzido do manual do PostgreSQL

Pode-se utilizar PL/pqSQL para a definição de triggers (gatilhos). Um procedimento do tipo trigger é criado com o comando CREATE FUNCTION, declarando como uma função sem argumentos e retornando o tipo trigger. Perceba que a função deve ser declarada sem argumentos, mesmo que seja esperado que a mesma receba argumentos através da declaração CREATE TRIGGER — os argumentos de triggers são passados via TG_ARGV, como descrito abaixo.

Quando uma função PL/pgSQL é chamada como uma trigger, várias variáveis especiais são criadas automaticamente no bloco mais externos:

NEW: tipo de dado RECORD e armazena a nova tupĺa da tabela para operações de INSERT/UPDATE (para triggers do tipo EACH ROW). Esta variável recebe NULL em trigger do tipo STATEMENT.

OLD: tipo de dado RECORD. Semelhante a NEW porém armazena os valores antigos da tabela para UPDATE/DELETE.

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 INSERT, UPDATE ou DELETE indicando qual operação de atualização disparou a trigger.

TG_RELID: tipo de dado OID. O OID da tabela que disparou a trigger.

TG_RELNAME: tipo de dado NAME. Nome da tabela que disparou a trigger. Está fora de uso, foi substituída por TG_TABLE_NAME.

TG_TABLE_NAME: nome da tabela que disparou a trigger.

TG_TABLE_SCHEMA: tipo de dado NAME. Nome do esquema da tabela que disparou a trigger.

TG_NARGS: tipo de dado INTEGER. Número do argumentos dados no comando CREATE TRIGGER.

TG_ARGV[]: tipo de dados matriz de TEXT. Argumentos passados no comando CREATE TRIGGER. Começa da posição 0.

Uma função do tipo trigger deve retornar ou NULL ou um valor de registro/linha/tupla tendo exatamente a estrutura da tabela que causou o disparo da trigger.

Triggers do tipo nível de linha (FOR EACH ROW) disparadas por BEFORE podem retornar NULL para avisar ao gerenciador de triggers para pular o resto da operação para aquela linha (ou seja, triggers subsequentes não são disparadas e o INSERT/UPDATE/DELETE não ocorre para aquela

linha).
Se um valor não nulo é retornado então a operação continua com o valor da linha dado.
Retornando um valor de linha diferente do valor em NEW, a linha terá o seu valor alterado (não tem efeito para o comando DELETE). Para alterar a linha a ser armazenada, é possível alterar um valor através de uma atribuição à variável NEW e retornar NEW modificado, ou construir uma nova linha e retorná-la então.

O valor de retorno de um BEFORE ou AFTER para trigger do tipo comando (STATEMENT) é sempre ignorado; podendo ser null.

Exemplo 1

Este exemplo de trigger garante que todas as vezes que uma linha é inserida ou atualizada em uma tabela (no caso EMP), o nome do usuário logado e a hora são inseridos na linha. Também verifica se o nome do funcionário é informado e se o salário é um valor positivo

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

-- Primeiro cria-se um função que retorna o  tipo TRIGGER

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

-- Em seguida cria a trigger para disparar a função criada

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

Outra forma de registrar as alterações na tabela é criar uma nova tabela que armazenará as operações que foram feitas na tabela origem.

Exemplo 02

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS
$emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

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

Exemplo 3

Simular o armazenamento de sumarização de tabelas para criação de Data Warehouses.

```
--
-- Tabelas do data warehouse.
--
-- Dimensão tempo
CREATE TABLE time_dimension (
   time_key                integer NOT NULL,
   day_of_week             integer NOT NULL,
   day_of_month            integer NOT NULL,
   month                   integer NOT NULL,
   quarter                 integer NOT NULL,
   year                    integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

-- Fato
CREATE TABLE sales_fact (
   time_key                integer NOT NULL,
   product_key             integer NOT NULL,
   store_key               integer NOT NULL,
   amount_sold             numeric(12,2) NOT NULL,
   units_sold              integer NOT NULL,
   amount_cost             numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Tabela sumáriop, vendas por tempo.
--
CREATE TABLE sales_summary_bytime (
   time_key                integer NOT NULL,
   amount_sold             numeric(15,2) NOT NULL,
   units_sold              numeric(12) NOT NULL,
   amount_cost             numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
   DECLARE
      delta_time_key          integer;
      delta_amount_sold       numeric(15,2);
      delta_units_sold        numeric(12);
      delta_amount_cost       numeric(15,2);
   BEGIN

      -- Work out the increment/decrement amount(s).
      IF (TG_OP = 'DELETE') THEN

         delta_time_key = OLD.time_key;
         delta_amount_sold = -1 * OLD.amount_sold;
         delta_units_sold = -1 * OLD.units_sold;
         delta_amount_cost = -1 * OLD.amount_cost;
```

```
    ELSIF (TG_OP = 'UPDATE') THEN

       -- forbid updates that change the time_key -
       -- (probably not too onerous, as DELETE + INSERT is how most
       -- changes will be made).
       IF ( OLD.time_key != NEW.time_key) THEN
          RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key, NEW.time_key;
       END IF;

       delta_time_key = OLD.time_key;
       delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
       delta_units_sold = NEW.units_sold - OLD.units_sold;
       delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

    ELSIF (TG_OP = 'INSERT') THEN

       delta_time_key = NEW.time_key;
       delta_amount_sold = NEW.amount_sold;
       delta_units_sold = NEW.units_sold;
       delta_amount_cost = NEW.amount_cost;

    END IF;


    -- Insert or update the summary row with the new values.
    <<insert_update>>
    LOOP
       UPDATE sales_summary_bytime
          SET amount_sold = amount_sold + delta_amount_sold,
             units_sold = units_sold + delta_units_sold,
             amount_cost = amount_cost + delta_amount_cost
          WHERE time_key = delta_time_key;

       EXIT insert_update WHEN found;

       BEGIN
          INSERT INTO sales_summary_bytime (
                   time_key,
                   amount_sold,
                   units_sold,
                   amount_cost)
             VALUES (
                   delta_time_key,
                   delta_amount_sold,
                   delta_units_sold,
                   delta_amount_cost
                  );

          EXIT insert_update;

       EXCEPTION
          WHEN UNIQUE_VIOLATION THEN
             -- do nothing
       END;
    END LOOP insert_update;

    RETURN NULL;

 END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
   FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```