Data Base Administrator PostgreSQL

Ernesto Quiñones A.
ernesto@eqsoft.net

Sesión 7

Pl/pgSQL
Parte 2

Triggers, Rules & Constraints

Funciones – Estructuras de Control

a) IF y CASE

```
create or replace function f_if_case(p_valor int) returns void as
$$
BEGIN
  IF     p_valor = 1  THEN  raise notice 'IF Valor UNO';
  ELSEIF p_valor = 2  THEN  raise notice 'IF Valor DOS';
  ELSE                      raise notice 'IF Valor %:', p_valor;
  END IF;

  CASE p_valor
    WHEN 1     THEN  raise notice 'CASE Valor UNO';
    WHEN 2,3   THEN  raise notice 'CASE Valor DOS o TRES';
    WHEN 2,3,4 THEN  raise notice 'CASE Valor DOS, TRES o CUATRO';
    ELSE             raise notice 'CASE Valor %:', p_valor;
  END CASE;

  CASE
    WHEN p_valor >= 1 and p_valor <= 6 THEN
      raise notice 'CASE-2 Valor entre UNO y SEIS';
    WHEN p_valor >=1 and (p_valor % 2) = 0  THEN
      raise notice 'CASE-2 Valor mayor a UNO y PAR';
    ELSE
      raise notice 'CASE-2 Valor %:', p_valor;
  END CASE;
END;
$$ language plpgsql;
```

Funciones – Estructuras de Control

b) LOOP, EXIT y CONTINUE

```
create or replace function f_loop(p_valor int) returns void as
$$
DECLARE
  contador integer := 0;
BEGIN
  LOOP
      contador := contador + 1;
      RAISE NOTICE 'Valor %:', contador;

      EXIT WHEN contador >= 3;

      IF contador >= p_valor THEN
        RAISE NOTICE 'Salió por el IF';
    EXIT;
      END IF;
  END LOOP;
END;
$$ language plpgsql;
```

```
prueba4=# select f_loop(2);
NOTICE:  Valor 1:
NOTICE:  Valor 2:
NOTICE:  Salió por el IF
```

```
prueba4=# select f_loop(5);
NOTICE:  Valor 1:
NOTICE:  Valor 2:
NOTICE:  Valor 3:
```

Funciones – Estructuras de Control

b) LOOP, EXIT y CONTINUE

```
create or replace function f_loop(p_valor int) returns void as
$$
DECLARE
  contador integer := 0;
BEGIN
  LOOP
      contador := contador + 1;
      RAISE NOTICE 'CONTADOR :%', contador;

      CONTINUE WHEN contador < 5; --fuerza el loop
        RAISE NOTICE 'Entro al continue :%', contador;

    IF contador >= p_valor THEN
      RAISE NOTICE 'Salió por el IF';
      EXIT;
    END IF;

  END LOOP;
END;
$$ language plpgsql;
```

Funciones – Estructuras de Control

b) LOOP, EXIT y CONTINUE

```
prueba4=# select f_loop(3);

NOTICE:  CONTADOR :1
NOTICE:  CONTADOR :2
NOTICE:  CONTADOR :3
NOTICE:  CONTADOR :4
NOTICE:  CONTADOR :5
NOTICE:  Entro al continue :5
NOTICE:  Salió por el IF
```

```
prueba4=# select f_loop(8);
NOTICE:  CONTADOR :1
NOTICE:  CONTADOR :2
NOTICE:  CONTADOR :3
NOTICE:  CONTADOR :4
NOTICE:  CONTADOR :5
NOTICE:  Entro al continue :5
NOTICE:  CONTADOR :6
NOTICE:  Entro al continue :6
NOTICE:  CONTADOR :7
NOTICE:  Entro al continue :7
NOTICE:  CONTADOR :8
NOTICE:  Entro al continue :8
NOTICE:  Salió por el IF
```

Funciones – Estructuras de Control

c) WHILE

```
create or replace function f_while(p_valor int) returns void as
$$
DECLARE
  contador integer := 0;
BEGIN
    WHILE contador <= p_valor LOOP
      contador := contador + 1;
      IF contador < 5 THEN
         CONTINUE;
      END IF;
      RAISE NOTICE ' Contador : %', contador;
    END LOOP;
END;
$$ language plpgsql;
```

```
prueba4=# select f_while(6);
NOTICE:   Contador : 5
NOTICE:   Contador : 6
NOTICE:   Contador : 7
```

```
prueba4=# select f_while(2);
 f_while
---------
(1 row)
```

Funciones – Estructuras de Control

d) FOR

```
create or replace function f_for(p_valor int) returns void as
$$
DECLARE
  contador integer := 0;
BEGIN
  FOR contador IN 1..p_valor LOOP
    RAISE NOTICE 'Contador A %', contador;
  END LOOP;

  FOR contador IN REVERSE p_valor..1 LOOP
    RAISE NOTICE 'Contador B %', contador;
  END LOOP;

  FOR contador IN REVERSE p_valor..1 BY 2 LOOP
    RAISE NOTICE 'Contador C %', contador;
  END LOOP;

  FOR contador IN 1..p_valor LOOP
    contador := contador + 2;
    RAISE NOTICE 'Contador D %', contador;
  END LOOP;

END;
$$ language plpgsql;
```

Funciones – Estructuras de Control

d) FOR

```
prueba4=# select f_for(3);
NOTICE:  Contador A 1
NOTICE:  Contador A 2
NOTICE:  Contador A 3
NOTICE:  Contador B 3
NOTICE:  Contador B 2
NOTICE:  Contador B 1
NOTICE:  Contador C 3
NOTICE:  Contador C 1
NOTICE:  Contador D 3
NOTICE:  Contador D 4
NOTICE:  Contador D 5
 f_for
-------
(1 row)
```

Funciones – Loop de set de datos

```
create or replace function f_prueba22() returns setof tpy_data3 as
$$
DECLARE
   data tpy_data3;
BEGIN
  FOR data IN
       select a.id,a.fecha,b.id,b.monto, '' from factura_cab a,
factura_det b where a.id = b.fac_id
  LOOP
    if data.monto > 200 then  data.mensaje = 'Muy caro';  end if;
    RETURN NEXT data;
  END LOOP;
  return;
END;
$$ language plpgsql;
```

Se puede hacer un query dinámico con:

```
  FOR data IN EXCUTE
      'select a.id,a.fecha,b.id,b.monto, '' from factura_cab a,
            factura_det b where a.id = b.fac_id'
  LOOP
```

Funciones – Mensajes

Para enviar mensajes a la consola de PostgreSQL utilizamos el comando RAISE, existen diversos niveles dependiendo el tipo de mensaje que deseamos enviar.

```
create or replace function f_raise(p_valor integer) returns void as
$$
DECLARE
  valor varchar;

BEGIN
  raise notice 'Mensaje Simple';

  raise notice 'El parametro es :%', p_valor;

  raise notice 'Este mensaje es' using HINT = '__una sugerencia__';

  raise sqlstate '23505' using MESSAGE = 'Cuando se de el ERROR 23505
saldra este mensaje';

END;
$$ language plpgsql;
```

Funciones – Control de errores

PostgreSQL tiene una relación de errores controlados bastante amplia.

https://www.postgresql.org/docs/current/static/errcodes-appendix.html

```
prueba5=# create table tbl_tabla2 ( id integer);
prueba5=# create index idx_1 on tbl_tabla2(id) unique;
```

Funciones – Control de errores

```
create or replace function f_error() returns void as $$
BEGIN
  BEGIN
    insert into tbl_tabla2 values(1);
    insert into tbl_tabla2 values(1);
    exception
      when unique_violation then
    raise notice 'Violo UNIQUE index';
  END;
  BEGIN
    insert into tbl_tabla2 values(1);
    insert into tbl_tabla2 values(1);
    exception
      when sqlstate '23505' then
    raise notice 'Violo UNIQUE index (2)';
  END;
END;
$$ language plpgsql;

prueba5=# select f_error(1);
NOTICE:  Violo UNIQUE index
NOTICE:  Violo UNIQUE index (2)
```

Funciones – Cursores

Un CURSOR permite cargar información en memoria (generalmente proveniente de una tabla) y poder trabajar con ella sin afectar los datos reales almacenados en la base de datos.

```
create or replace function f_cursor1() returns void as $$
DECLARE
  c_tabla1 refcursor; data record;
BEGIN
  open c_tabla1 for execute 'select * from tbl_usuario';
  LOOP
      fetch c_tabla1 into data;
      IF NOT FOUND THEN
        exit;
      END IF;
      raise notice 'Nombre: %', data.nombre;
  END LOOP;
END;
$$ language plpgsql;

prueba5=# select f_cursor1();
NOTICE:  Nombre: ernesto
NOTICE:  Nombre: juan
NOTICE:  Nombre: pedro
```

Funciones – Cursores

```
create or replace function f_cursor2() returns void as $$
DECLARE
  c_tabla1 refcursor; data record; contador integer := 0;
BEGIN
   OPEN c_tabla1 SCROLL FOR EXECUTE 'select * from tbl_usuario';
   LOOP
      contador = contador +1;
      IF contador % 3 = 0 THEN
         FETCH PRIOR FROM c_tabla1 INTO data;
      ELSE
         FETCH NEXT FROM c_tabla1 INTO data;
      END IF;
      IF NOT FOUND THEN
         exit;
      END IF;
      raise notice 'Nombre: % %', contador, data.nombre;
  END LOOP;
END;
$$ language plpgsql;
```

Podemos ir hacia atrás o hacia adelante en un CURSOR, recuerda que el orden de los datos lo defines añadiendo un ORDER BY al Query.

Funciones – Cursores

```
prueba5=# select f_cursor2();
NOTICE:  Nombre: 1 ernesto
NOTICE:  Nombre: 2 juan
NOTICE:  Nombre: 3 ernesto
NOTICE:  Nombre: 4 juan
NOTICE:  Nombre: 5 pedro
NOTICE:  Nombre: 6 juan
NOTICE:  Nombre: 7 pedro
```

Para más variaciones de FETCH (siempre abrir el cursor con SCROLL para esto)
http://developer.postgresql.org/pgdocs/postgres/sql-fetch.html

Funciones – Cursores

```
create or replace function f_cursor4() returns void as $$
DECLARE
  c_tabla1 refcursor; data record;
BEGIN
   OPEN c_tabla1 FOR EXECUTE 'select * from tbl_usuario';
   LOOP
      FETCH c_tabla1 INTO data;
      IF NOT FOUND THEN
        exit;
      END IF;
      raise notice 'Nombre: %', data.nombre;
      move relative 1 FROM c_tabla1;
  END LOOP;
END;
$$ language plpgsql;
```

La característica principal de MOVE es que no devuelve data de la tabla a la cual esta unida el cursor, solo mueve el puntero de posición.

Variaciones: NEXT, PRIOR, FIRST, LAST, ABSOLUTE XX, RELATIVE XX, ALL, FORWARD XX ó ALL, BACKWARD XX ó ALL, donde XX es la cantidad de espacios

Funciones – Cursores

```
create or replace function f_cursor5( p_valor integer) returns void
as $$
DECLARE
  c_tabla1 cursor FOR select * from tbl_usuario where id = p_valor;
  data record; contador integer;
BEGIN
   OPEN c_tabla1;
   LOOP
      FETCH c_tabla1 INTO data;
      IF NOT FOUND THEN exit;  END IF;
      raise notice 'Nombre: %', data.nombre;

      update tbl_usuario set nombre='--' || data.nombre
                        where id = data.id;
      GET DIAGNOSTICS contador = ROW_COUNT;   también existe
                                              RESULT_OID y OID
      if contador = 1 then
         Raise notice 'Se proceso % registro', contador;
      end if;
  END LOOP;
END; $$ language plpgsql;
```

Funciones – Cursores

```
prueba5=# select * from f_cursor5(1);
NOTICE:  Nombre: --ernesto
NOTICE:  Se proceso 1 registro
 f_cursor5
-----------
 (1 fila)

prueba5=# select * from f_cursor5(5);
 f_cursor5
-----------
 (1 fila)
```

Funciones – LABELs y LOOPs

```
create or replace function saltos () returns integer as $$
Declare contador integer := 0 ;
Begin
     <<salto1>>
     loop
          contador := contador + 1; raise notice 'paso 1: %', contador;
          if contador < 10 then continue salto1;
          else raise notice 'PASA AL OTRO LOOP';
          end if;

          if contador > 20 then
               raise notice 'FIN DE TODO';
               return -1;
          end if;

          <<salto2>>
          loop
               contador := contador + 1; raise notice 'paso 2: %', contador;
               if contador >= 20 then continue salto1;
               else continue salto2;
               end if;
          end loop salto2;

     end loop salto1;
     return 1;
end;
$$ language plpgsql;
```

Funciones – LABELs y LOOPs

El ejemplo muestra como podemos anidar LOOPs y asignar un nombre a cada uno, leugo dentro de ellos podemos saltar entre uno interno a uno externo y viceversa, más no podemos saltar entre LOOPs de un mismo nivel, es decir, esto es valido:

```
<<salto1>>
loop
    Continue salto2
    <<salto2>>
    Loop
    End loop salto 2
End loop salto1
```

Pero esto NO es valido:

```
<<salto1>>
loop
    Continue salto2
End loop salto1
<<salto2>>
Loop
    Continue salto1
End loop salto 2
```

Los LABELs pueden ser usados con: LOOP, EXIT, CONTINUE, WHILE, FOR, FOREACH

Funciones – Tipos de funciones por los resultados que devuelve

IMMUTABLE

Indica que la función no modificará la base de datos, y que siempre devolverá el mismo resultado si se le pasan los mismos valores a sus parámetros.

STABLE
La función no modificará la base de datos, pero que haciendo búsquedas en las tablas siempre retornará el mismo resultado si es que se pasan los mismos valores a sus parámetros, pero que podría cambiar en alguno de los querys que tiene incluidos.

VOLATILE

La función cambiará su resultado siempre.

```
create or replace function saltos () returns integer as $$
Declare contador integer := 0 ;
Begin
     -- QUERYS
     -- QUERYS
     return 1;
end;
$$ language plpgsql IMMUTABLE;
```

Funciones – Tipos de funciones por los resultados que devuelve

CALLED ON NULL INPUT

Indica que la función podría recibir en un parámetro un NULL y debería procesarce sin revisión adicional.

RETURNS NULL ON NULL INPUT

Indica que si la función recibir un parámetro en NULL no se procesará y se mandará de resultado de la misma otro NULL.

```
create or replace function saltos (parametro INT) returns integer as $$
Declare contador integer := 0 ;
Begin
     -- QUERYS
     -- QUERYS
     return 1;
end;
$$ language plpgsql IMMUTABLE RETURNS NULL ON NULL INPUT ;
```

Funciones – Transacciones controladas por el cliente

Una transacción puede englobar a varias funciones que realizan sus transacciones por dentro, si la transacción principal no se culmina con un commit todas las transacciones anidadas dentro de ella se hacen rollback al detectarse un problema.

```
prueba5=# select * from tbl_usuario;
 id |   nombre
----+-------------
  2 | --juan
  3 | --pedro
  1 | ----ernesto

create or replace function f_commit1() returns void as $$
BEGIN
  insert into tbl_usuario (nombre) values ('lucho');
  insert into tbl_usuario (nombre) values ('pedro');
END;
$$ language plpgsql;

create or replace function f_commit2() returns void as $$
BEGIN
  update tbl_usuario set nombre = 'juan' where nombre = '--juan';
  update tbl_usuario set nombre = 'pedro' where nombre = '--pedro';
END;
$$ language plpgsql;
```

Funciones – Transacciones controladas por el cliente

```
<?php

$conn = pg_connect("host=127.0.0.1 port=5432 dbname=prueba5
user=dbadmin password=dbadmin");
if ($conn <> false ) {
  echo "entro en la db\r\n";

  pg_exec("begin");

  pg_exec("select f_commit1()");
  pg_exec("select f_commit2()");

  pg_exec("rollback");
}

?>
```

Nótese que la transacción principal no se esta cerrando en el código, más bien se está forzando a realizar un rollback.

Funciones – Transacciones controladas por el cliente

```
ernesto@depeche:~/Documentos$ php prueba.php
entro en la db

prueba5=# select * from tbl_usuario;
 id |   nombre
----+------------
  2 | --juan
  3 | --pedro
  1 | ----ernesto
```

NO SE EJECUTO NADA!!, ahora cambiemos el rollback en el PHP por un commit y ejecutemos el código.

```
ernesto@depeche:~/Documentos$ php prueba.php
entro en la db

prueba5=# select * from tbl_usuario;
 id |   nombre
----+------------
  1 | ----ernesto
  8 | lucho
  9 | pedro
  2 | juan
  3 | pedro
```

TRIGGERS

Un TRIGEER es una función que se ejecuta cuando se realiza alguna acción sobre un tabla.

```
prueba5=# create table alumnos(id serial, nombre varchar(100), edad
integer, observ varchar(100));

create or replace function ft_prueba1() returns trigger as $$
begin
  IF NEW.edad is NULL THEN
      NEW.observ := 'NO especifica';
  ELSEIF NEW.edad < 18 THEN
      NEW.observ := 'menor edad';
  ELSEIF NEW.edad >= 18 THEN
      NEW.observ := 'mayor edad';
  END IF;
  RETURN NEW;
end;
$$ language plpgsql;
```

Una función TRIGGER siempre retornará un TRIGGER

TRIGGERS

Un TRIGEER tiene dos componente, la Función Trigger y el Trigger en si mismo, el Trigger especifica la tabla que afecta y la función quese debe ejecutar, además del evento que disparará el Trigger.

```
prueba5=# create trigger t_alumnos BEFORE INSERT ON alumnos FOR EACH
ROW EXECUTE Procedure ft_prueba1();

prueba5=# insert into alumnos (nombre,edad) values ('ernesto', null);
INSERT 0 1
prueba5=# insert into alumnos (nombre,edad) values ('juan', 15);
INSERT 0 1
prueba5=# insert into alumnos (nombre,edad) values ('pedro', 25);
INSERT 0 1
prueba5=# select * from alumnos;
 id | nombre  | edad |    observ
----+---------+------+---------------
  1 | ernesto |      | NO especifica
  2 | juan    |   15 | menor edad
  3 | pedro   |   25 | mayor edad
```

TRIGGERS

```
create or replace function ft_prueba2() returns trigger as $$
begin
  IF NEW.edad is NULL THEN
      NEW.observ := 'NO especifica';
  ELSEIF NEW.edad < 18 THEN
      NEW.observ := 'menor edad';
  ELSEIF NEW.edad > 18 and NEW.edad < 100 THEN
      NEW.observ := 'mayor edad';
  ELSEIF NEW.edad > 100 THEN
      raise notice 'Ya debería estar muerto';
      RETURN NULL;
  END IF;
  RETURN NEW;
end;
$$ language plpgsql;
```

Una FUNCION TRIGGER siempre devuelve un valor (NEW o OLD), sin embargo si en nuestra validación decidimos que la tabla no debe ser afectada podemos retornar un NULL.

TRIGGERS

```
Prueba5=# create trigger t_alumnos2 BEFORE INSERT ON alumnos FOR
EACH ROW EXECUTE Procedure ft_prueba2();

prueba5=# insert into alumnos (nombre,edad) values ('cesar', 115);
NOTICE:  Ya debería estar muerto
INSERT 0 0
prueba5=# select * from alumnos;
 id | nombre  | edad |    observ
----+---------+------+---------------
  1 | ernesto |      | NO especifica
  2 | juan    |   15 | menor edad
  3 | pedro   |   25 | mayor edad
(3 filas)
```

TRIGGERS

```
create or replace function ft_prueba3() returns trigger as $$
begin
  IF NEW.edad is NULL THEN
      raise notice 'NO ES VALIDO, NO ACTUALIZO';
      RETURN OLD;
  ELSEIF NEW.edad < 18 THEN
      NEW.observ := 'menor edad';
  ELSEIF NEW.edad >= 18 and NEW.edad < 100 THEN
      NEW.observ := 'mayor edad';
  ELSEIF NEW.edad > 100 THEN
      raise notice 'NO ES VALIDO, NO ACTUALIZO';
      RETURN OLD;
  END IF;
  RETURN NEW;
end;
$$ language plpgsql;
```

Una FUNCION TRIGGER siempre devuelve un NEW o un OLD, cuando se devuelve un NEW todos los nuevos datos ingresados se verán reflejados en la tabla, cuando se devuelve un OLD entonces se mantendrán los datos antiguso con los que ya contabla la tabla.

TRIGGERS

```
prueba5=# create trigger t_alumnos3 BEFORE UPDATE ON alumnos FOR
EACH ROW EXECUTE Procedure ft_prueba3();

prueba5=# update alumnos set edad = 18 where id = 1;
UPDATE 1
prueba5=# update alumnos set edad = 118 where id = 2;
NOTICE:  NO ES VALIDO, NO ACTUALIZO
UPDATE 1
prueba5=# select * from alumnos;
 id | nombre  | edad |   observ
----+---------+------+------------
  3 | pedro   |   25 | mayor edad
  1 | ernesto |   18 | mayor edad
  2 | juan    |   15 | menor edad
```

TRIGGERS

```
create or replace function ft_prueba4() returns trigger as $$
begin
  IF TG_OP = 'INSERT' THEN
    IF NEW.edad is NULL  or NEW.edad < 18 or NEW.edad > 100 THEN
         raise notice 'Edad prohibidas'; RETURN NULL;
    ELSE
        NEW.observ = 'dato valido';   RETURN NEW;
    END IF;
  ELSEIF TG_OP = 'DELETE' THEN
    raise notice 'Prohibido borrar data de esta tabla';  RETURN NULL;
  ELSEIF TG_OP = 'UPDATE' THEN
    IF NEW.edad is NULL  or NEW.edad < 18 or NEW.edad > 100 THEN
         raise notice 'Edad prohibidas';   RETURN OLD;
    ELSE
        NEW.observ = 'En los esperado';  RETURN NEW;
    END IF;
  END IF;
end;
$$ language plpgsql;
```

Dentro de una FUNCION TRIGGER es posible controlar que evento lo disparó, los eventos son el Insert, Update y Delete.

TRIGGERS

Como se puede ver, al momento de crear el TRIGGER es posible especificar antes que eventos va a responder.

```
prueba5=# create trigger t_alumnos4 BEFORE INSERT OR UPDATE OR DELETE
ON alumnos FOR EACH ROW EXECUTE Procedure ft_prueba4();

prueba5=# insert into alumnos (edad,nombre) values (16,'luis');
NOTICE:  Edad prohibidas
prueba5=# insert into alumnos (edad,nombre) values (26,'luis');
prueba5=# update alumnos set edad = 29 where id = 1;
prueba5=# update alumnos set edad = 129 where id = 1;
NOTICE:  Edad prohibidas
prueba5=# delete from alumnos where id = 1;
NOTICE:  Prohibido borrar data de esta tabla

prueba5=# select * from alumnos;
 id | nombre  | edad |     observ
----+---------+------+-----------------
  3 | pedro   |   25 | mayor edad
  2 | juan    |   15 | menor edad
  6 | luis    |   26 | dato valido
  1 | ernesto |   29 | En los esperado
```

TRIGGERS

```
create or replace function ft_prueba5() returns trigger as $$
begin
  raise notice 'NEW: %', NEW;
  raise notice 'OLD: %', OLD;
  raise notice 'TG_NAME: %', TG_NAME;
  raise notice 'TG_WHEN: %', TG_WHEN;
  raise notice 'TG_LEVEL: %', TG_LEVEL;
  raise notice 'TG_OP: %', TG_OP;
  raise notice 'TG_RELID: %',TG_RELID;
  raise notice 'TG_RELNAME: %',TG_RELNAME;
  raise notice 'TG_TABLE_NAME: %',TG_TABLE_NAME;
  raise notice 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
  raise notice 'TG_NARGS: %',TG_NARGS;
  raise notice 'TG_ARGV[]: %',TG_ARGV[1];
end;
$$ language plpgsql;
```

Un TRIGGER tiene una serie de constantes que almacenan información sobre el.

TRIGGERS

```
prueba5=# create trigger t_alumnos5 BEFORE UPDATE ON alumnos FOR
EACH ROW EXECUTE Procedure ft_prueba5();

prueba5=# update alumnos set edad = 14 where id = 1;
NOTICE:  NEW: (1,ernesto,14,"En los esperado")
NOTICE:  OLD: (1,ernesto,29,"En los esperado")
NOTICE:  TG_NAME: t_alumnos5
NOTICE:  TG_WHEN: BEFORE
NOTICE:  TG_LEVEL: ROW
NOTICE:  TG_OP: UPDATE
NOTICE:  TG_RELID: 22264
NOTICE:  TG_RELNAME: alumnos
NOTICE:  TG_TABLE_NAME: alumnos
NOTICE:  TG_TABLE_SCHEMA: public
NOTICE:  TG_NARGS: 0
NOTICE:  TG_ARGV[]: <NULL>
```

TRIGGERS

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

```
create trigger t_alumnos5
  BEFORE UPDATE
  ON alumnos
  FOR EACH ROW
  WHEN (NEW.* IS DISTINCT FROM OLD.* )   <-- solo en PostgreSQL 9.0
  EXECUTE PROCEDURE ft_prueba5();
```

TRIGGERS

Existen ocasiones en las que necesitamos deshabilitar triggers de las tablas:

```
BEGIN;
    -- desactivamos los triggers
    UPDATE pg_catalog.pg_class SET
        reltriggers = 0
        WHERE oid =    'nombre_tabla'::pg_catalog.regclass;
    -- operaciones sobre nombre_tabla ...
    -- activamos los triggers sobre nombre-tabla
    UPDATE pg_catalog.pg_class SET
        reltriggers = (SELECT pg_catalog.count(*)
                    FROM pg_catalog.pg_trigger
                    where pg_class.oid = tgrelid)
        WHERE oid = 'nombre_tabla'::pg_catalog.regclass;
COMMIT;
```

RULES

Las reglas nos permiten de una manera más limitada desarrollar alguna acción ante un evento producido en una tabla ó vista, no permiten desarrollar toda una lógica como en el caso de los triggers.

```
create table regla (id serial , nombre varchar(100));

create table log_regla( fecha timestamp , id integer,
                        nombre varchar(100), estado char(1));

CREATE RULE rul_i_fecha AS ON INSERT TO regla
  DO insert into log_regla values ( now(), NEW.id, NEW.nombre, 'I');

prueba4=# insert into regla (nombre) values ('Ernesto');

prueba4=# select * from regla;
 id | nombre
----+---------
  1 | Ernesto
```

RULES

```
prueba4=# select * from log_regla;
           fecha            | id | nombre  | estado
----------------------------+----+---------+--------
 2010-10-08 20:26:37.200866 |  2 | Ernesto | I
```

Ojo con el serial “id”, al reutilizarlo se relanza el nextval y devuelve un valor erroneo, lo correcto sería:

```
CREATE OR REPLACE RULE rul_i_fecha AS ON INSERT TO regla
  DO insert into log_regla values ( now(), currval('regla_id_seq'),
                                    NEW.nombre, 'I');
```

RULES

“Instead” nos permite que la acción solicitada ya no se lleve a cabo y se realice la acción alternativa declarada en la regla.

```
CREATE OR REPLACE RULE rul_i_fecha AS ON INSERT TO regla
  DO INSTEAD insert into log_regla values ( now(), NEW.id,
NEW.nombre, 'I');

prueba4=# insert into regla (nombre) values ('Ernesto4');
INSERT 0 1
prueba4=# select * from regla;
 id |  nombre
----+----------
  1 | Ernesto
  3 | Ernesto2
  5 | Ernesto3

prueba4=# select * from log_regla;
 fecha             | id |  nombre  | estado
---------------------------+----+----------+--------
 2010-10-08 20:26:37.200866 |  2 | Ernesto  | I
 2010-10-08 20:29:47.29924  |  4 | Ernesto2 | I
 2010-10-08 20:34:00.101965 |  6 | Ernesto3 | I
 2010-10-08 20:35:08.391818 |  7 | Ernesto4 | I
```

RULES

```
CREATE OR REPLACE RULE rul_u_fecha AS ON UPDATE TO regla WHERE OLD.id < 6
  DO insert into log_regla values ( now(), NEW.id, NEW.nombre, 'U');

prueba4=# update regla set nombre = 'alejandro' where id = 1;
prueba4=# update regla set nombre = 'alejandro' where id = 9;

prueba4=# select * from regla;
 id |  nombre
----+-----------
  3 | Ernesto2
  5 | Ernesto3
  1 | alejandro
  9 | alejandro

prueba4=# select * from log_regla;
           fecha             | id |  nombre   | estado
-----------------------------+----+-----------+--------
 2010-10-08 20:26:37.200866 |  2 | Ernesto   | I
 2010-10-08 20:29:47.29924  |  4 | Ernesto2  | I
 2010-10-08 20:34:00.101965 |  6 | Ernesto3  | I
 2010-10-08 20:35:08.391818 |  7 | Ernesto4  | I
 2010-10-08 20:39:57.221768 |  8 | Ernesto4  | I
 2010-10-08 20:49:24.691783 |  8 | pedro     | I
 2010-10-08 20:49:43.711889 |  9 | pedro     | I
 2010-10-08 20:59:19.761879 |  1 | Ernesto   | U
 2010-10-08 21:00:08.529349 |  1 | alejandro | U
```

RULES

```
CREATE OR REPLACE RULE rul_d_fecha AS ON DELETE TO regla
  DO select sum(id) as quedan from regla;

prueba4=# select * from regla;
 id |  nombre
----+-----------
  3 | Ernesto2
  5 | Ernesto3
  9 | alejandro
(3 rows)

prueba4=# delete from regla where id = 3;
 quedan
--------
     17
(1 row)

prueba4=# select * from regla;
 id |  nombre
----+-----------
  5 | Ernesto3
  9 | alejandro
(2 rows)
```

RULES

```
CREATE OR REPLACE RULE rul_u_fecha AS ON UPDATE TO regla WHERE OLD.id
< 6  DO (insert into log_regla values ( now(), NEW.id, NEW.nombre,
'U'); select * from log_regla where id = NEW.id);

prueba4=# select * from regla;
 id |  nombre
----+-----------
  9 | alejandro
  5 | pepelucho

prueba4=# update regla set nombre = 'pepelucho1' where id = 5;
           fecha            | id |   nombre   | estado
----------------------------+----+------------+--------
 2010-10-08 21:21:55.82194  |  5 | pepelucho  | U
 2010-10-08 21:22:06.759216 |  5 | pepelucho1 | U

prueba4=# select * from regla;
 id |   nombre
----+------------
  9 | alejandro
  5 | pepelucho1
```

Constrains

Un CONSTRAIN es una validación a nivel de declaración de la estructura de a tabla, es una manera alterna a un TRIGGER o un RULE de mantener la consistencia de datos

```
prueba=# create table animales ( nombre varchar, tipo varchar check
(tipo = 'mamifero'));
prueba=# insert into animales values ('perro','mamifero');

prueba=# insert into animales values ('iguana','reptil');
ERROR:  new row for relation "animales" violates check constraint
"animales_tipo_check"
DETAIL:  Failing row contains (iguana, reptil).

prueba=# select * from animales;
 nombre |   tipo
--------+----------
 perro  | mamifero
(1 row)
```

Constrains

Se puede declarar un CONSTRAIN a nivel de tabla o de un campo (como en el ejemplo), si alguno de los CONSTRAINS declarados es violado entonces no se actualiza la data en la tabla.

```
prueba=# create table animales2 ( nombre varchar, tipo varchar, check
(tipo = 'mamifero'));

prueba=# insert into animales2 values ('iguana','reptil');
ERROR:  new row for relation "animales2" violates check constraint
"animales2_tipo_check"
DETAIL:  Failing row contains (iguana, reptil).
```

Constrains

El orden de ejecución de los CONSTRAINS es primero el de la tabla y luego el de los campos.

```
prueba=# create table animales6 ( nombre varchar constraint
solo_con_a check (substr(nombre,1,1) = 'A') , tipo varchar,
constraint solo_mamifero check  (tipo = 'mamifero'));

prueba=# insert into animales6 values ('Algo','mamifero');

prueba=# insert into animales6 values ('perro','mamifero');
ERROR:  new row for relation "animales6" violates check constraint
"solo_con_a"
DETAIL:  Failing row contains (perro, mamifero).

prueba=# insert into animales6 values ('Algo','ave');
ERROR:  new row for relation "animales6" violates check constraint
"solo_mamifero"
DETAIL:  Failing row contains (Algo, ave).

prueba=# insert into animales6 values ('perro','ave');
ERROR:  new row for relation "animales6" violates check constraint
"solo_mamifero"
DETAIL:  Failing row contains (perro, ave).
```