Taller de Bases de Datos IN1078C

Prof: Mariella Gutiérrez Valenzuela

Contenidos

2. Procedimientos Almacenados y Triggers
    a. Disparadores (Triggers)
    b. Procedimientos almacenados
    c. Funciones en lenguaje Pl/Sql

Trigger

- Un trigger es una llamada o “disparador” de un procedimiento almacenado (plpgsql) que se ejecuta en el momento que una operación INSERT, UPDATE o DELETE es aplicada a una tabla.

- Es el procedimiento o Trigger function el que contiene el código que se va a ejecutar por el disparador.

- Se crea para producir actualizaciones de datos, validar valores o hacer respaldos en otras tablas cada vez que ocurre un evento determinado, de forma automática.

Trigger

Forma General:

CREATE TRIGGER *name*

{ BEFORE | AFTER|INSTEAD OF } { *event* [OR ...] } ON *table*

FOR EACH { ROW | STATEMENT }

EXECUTE PROCEDURE *Name_func* (*arguments*)

DROP TRIGGER name ON table;

Donde:

Name: nombre del trigger

Before|After|Instead of: indica si es antes, después o en lugar del evento.

Event: INSERT, UPDATE, DELETE

Table: el nombre de la tabla donde se produce el evento.

FOR EACH ROW: se dispara para cada fila.

FOR EACH STATEMENT: se dispara sólo una vez.

Name_func: es el procedimiento que se dispara cada vez que se ejecuta el trigger.

Trigger

Antes de crear un trigger se debe escribir previamente un procedimiento o Trigger Function que debe tener la siguiente estructura:

```
CREATE OR REPLACE FUNCTION Name_func ()
RETURNS trigger AS $BODY$
BEGIN
    Body…
    RETURN retorno;
END; $BODY$  LANGUAGE lang VOLATILE;
```

Donde:

Name_Func: es el nombre de la función.

RETURNS trigger: se usa para indicar que es una Trigger Function

BEGIN .. END: indica el inicio y fin del cuerpo de la función, respectivamente.

Retorno: puede ser NULL, o la nueva fila NEW o la antigua OLD.

Lang: en el caso de un Trigger function es ‘c’, ‘plpgsql’ o ‘internal’

VOLATILE: indica que la función puede modificar la BD y puede variar de una llamada a otra.

Trigger

Características:

- El procedimiento almacenado que se vaya a utilizar por el disparador debe de definirse e instalarse antes de definir el propio disparador.
- Un procedimiento que se vaya a utilizar por un disparador no puede tener argumentos y tiene que devolver el tipo "trigger".
- Un mismo procedimiento se puede utilizar por múltiples disparadores en diferentes tablas.
- Procedimientos utilizados por disparadores que se ejecutan una sola vez por comando SQL (statement-level) tienen que devolver siempre NULL.
- Procedimientos utilizados por disparadores que se ejecutan una vez por línea afectada por el comando SQL (row-level) pueden devolver una fila de tabla.
- Procedimientos utilizados por disparadores que se ejecutan DESPUES de ejecutar el comando SQL que lo lanzó, ignoran el valor de retorno, así que pueden retornar NULL sin problemas.

Martínez, Rafael. Postgresql-es. Recuperado el 01/05/2016 desde la URL: http://www.postgresql.org.es/node/301

Trigger

Características:

- Procedimientos utilizados por disparadores que se ejecutan una vez por fila afectada por el comando SQL (row-level) y ANTES de ejecutar el comando SQL que lo lanzó, pueden retornar:
  - NULL para saltarse la operación en la fila afectada.
  - Devolver una fila de tabla (RECORD)

- En resumen, independientemente de cómo se defina un disparador, el procedimiento utilizado por dicho disparador devuelve NULL o un valor RECORD con la misma estructura que la tabla que lanzó dicho disparador.

- Si una tabla tiene más de un disparador definido para un mismo evento (INSERT,UPDATE,DELETE), estos se ejecutarán en orden alfabético por el nombre del disparador.

  En el caso de disparadores del tipo ANTES / row-level, la fila retornada por cada disparador, se convierte en la entrada del siguiente. Si alguno de ellos retorna NULL, la operación será anulada para la fila afectada.

Martínez, Rafael. Postgresql-es. Recuperado el 01/05/2016 desde la URL: http://www.postgresql.org.es/node/301

Trigger

Características:

- Procedimientos utilizados por disparadores pueden ejecutar sentencias SQL que a su vez pueden activar otros disparadores. Esto se conoce como disparadores en cascada.

  Importante: No existe límite para el número de disparadores que se pueden llamar, pero es responsabilidad del programador el evitar una recursión infinita de llamadas en la que un disparador se llame a sí mismo de manera recursiva.

Martínez, Rafael. Postgresql-es. Recuperado el 01/05/2016 desde la URL: http://www.postgresql.org.es/node/301

Trigger

El procedimiento llamado por un trigger no recibe argumentos pasados como parámetros y siempre retornan “trigger”.

Sin embargo, automáticamente tienen variables especiales disponibles:

- NEW: Tipo de dato RECORD; Variable que contiene la nueva fila de la tabla para las operaciones INSERT/UPDATE en disparadores del tipo row level. Esta variable es NULL en disparadores del tipo statement level.

- OLD: Tipo de dato RECORD; Variable que contiene la antigua fila de la tabla para las operaciones UPDATE/DELETE en disparadores del tipo row level. Esta variable es NULL en disparadores del tipo statement level.

- TG_NAME: Tipo de dato name; variable que contiene el nombre del disparador que está usando la función actualmente.

- TG_WHEN: Tipo de dato text; una cadena de texto con el valor BEFORE o AFTER dependiendo de cómo fue definido el disparador que está usando la función.

Trigger

- TG_LEVEL: Tipo de dato text; con el valor ROW o STATEMENT dependiendo de como el disparador está usando la función actualmente.

- TG_OP: Tipo de dato text; con el valor INSERT, UPDATE o DELETE dependiendo de la operación que ha activado el disparador.

- TG_TABLE_NAME: Tipo de dato name; con el nombre de la tabla que ha activado el disparador.

Trigger

Como resumen, el procedimiento recibe los registros OLD y NEW según lo siguiente:

1. NEW
    a. registro insertado (INSERT)
    b. registro modificado (UPDATE)

2. OLD
    a. registro antes de modificar (UPDATE)
    b. registro eliminado (DELETE)

Puede retornar:

- NULL (detener la acción)
- OLD (eventos DELETE, UPDATE)
- NEW (eventos INSERT, UPDATE)

NOTA: modificaciones a los valores sólo pueden ser hechas con disparadores *BEFORE*

Trigger

En el cuerpo de la función es posible incluir estructuras de control: **LOOP, EXIT, IF–THEN–ELSE, CASE, FOR, WHILE**

Ciclo LOOP: Esta sentencia nos permite efectuar un ciclo, su estructura básica es :

```
LOOP
...sentencias a ejecutar en el ciclo...
END LOOP;
```

Exit: se puede utilizar para finalizar un loop.

```
EXIT [ label ] [ WHEN boolean-expression ];
```

Ej:

```
LOOP
        -- some computations
        IF count > 0 THEN EXIT; -- exit loop
        END IF;
END LOOP;
```

Trigger

IF–THEN–ELSE : permite ejecutar distintas sentencias a partir de si se cumple o no una condición:

IF *boolean-expression*
THEN *statements*
[ELSIF *boolean-expression* THEN *statements*]
[ELSE *statements*]
END IF;

Ej. IF number = 0 THEN
result := 'zero’;
ELSIF number > 0 THEN
result := 'positive’;
ELSIF number < 0 THEN
result := 'negative’;
ELSE
-- the only other possibility is that number is null
result := 'NULL';
END IF;

Trigger

CASE (Simple): Esta sentencia nos permite efectuar comparaciones y a partir del resultado ejecutar una de las opciones:

CASE *search-expression*

WHEN *expression* [, *expression* [...]] THEN *statements*

[WHEN *expression* [, *expression* [...]] THEN *statements* ...]

[ELSE *statements*]

END CASE;

Ej. CASE x

```
        WHEN 1, 2 THEN
                msg := 'one or two';
   ELSE
                msg := 'other value than one or two';
   END CASE;
```

Trigger

CASE (Search):

CASE
WHEN *expression* [, *expression* [...]] THEN *statements*
[WHEN *expression* [, *expression* [...]] THEN *statements* ...]
[ELSE *statements*]
END CASE;

Ej. CASE
WHEN x BETWEEN 0 AND 10 THEN
msg := 'value is between zero and ten';
WHEN x BETWEEN 11 AND 20 THEN
msg := 'value is between eleven and twenty';
END CASE;

Trigger

FOR: permite ejecutar un conjunto de sentencias un número de veces determinado por un ciclo, que puede ser numérico o bien, en función de una sentencia Sql.

FOR (Numérico):

```
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
   statements
END LOOP [ label ];
```

Ej:

```
FOR i IN 1..10 LOOP
   -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
   -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
```

Trigger

FOR (SQL):

```
FOR target IN query LOOP
   statements
END LOOP [ label ];
```

Ej:

```
… Inicio de la Función
 DECLARE
   registro RECORD;
BEGIN
FOR registro IN SELECT * FROM productos LOOP
   stock := registro.sock_actual + 100;
   ... otras sentencias ...
END LOOP;
```

Trigger

WHILE: mientras se cumpla una condición se ejecuta un loop.

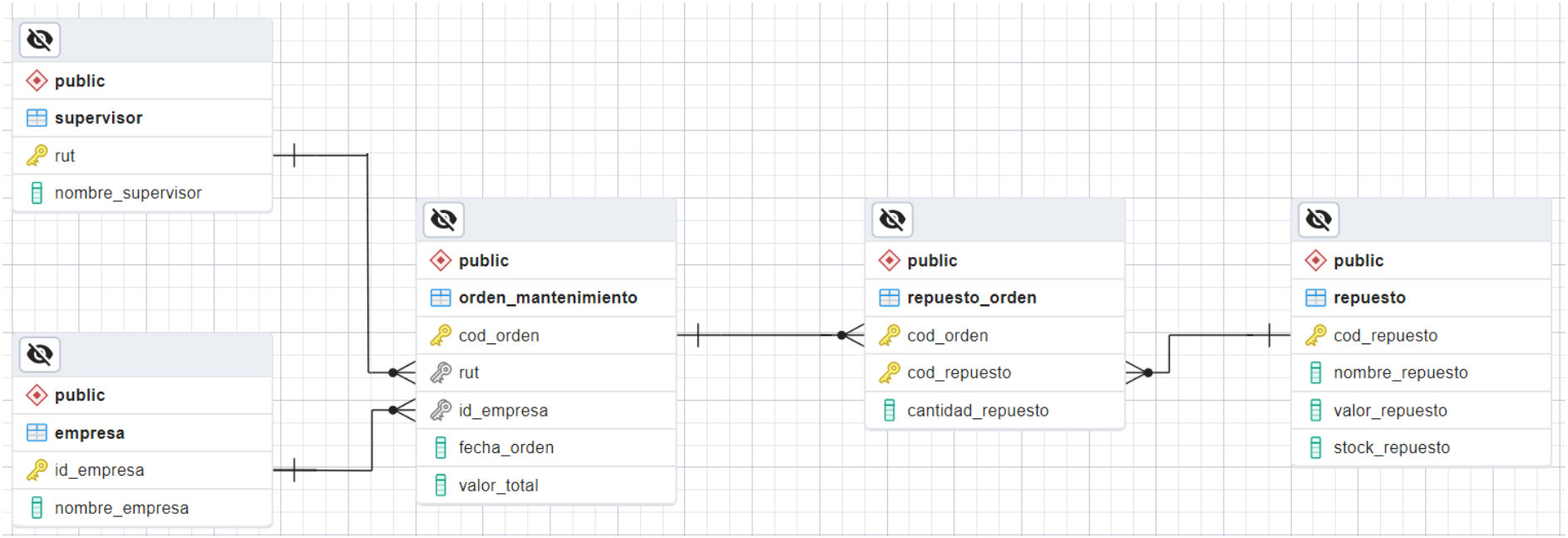
```
WHILE boolean-expression LOOP
   statements
END LOOP [ label ];
```

Ej:

```
inc := 1;
WHILE inc <= 10 LOOP
   factorial := factorial * inc;
   inc:= inc +1;
END LOOP;
RETURN factorial;
```

Trigger

Ejercicios: Vamos a utilizar la Base de Datos que ya tiene creada de mantenimiento:



Trigger

Resuelva los siguientes ejercicios:

1. Construya un trigger en la tabla REPUESTO_ORDEN para que cada vez que se INSERTA un nuevo repuesto a una orden de mantenimiento, se actualice el campo valor_total correspondiente en la tabla ORDEN_MANTENIMIENTO.

2. Realice las modificaciones necesarias para que en el trigger anterior se verifique previamente que la cantidad de repuesto del nuevo registro, es menor o igual al stock_repuesto en la tabla REPUESTO.

3. Nuevamente realice las modificaciones necesarias para que en el trigger anterior, si la cantidad de repuesto es menor o igual al stock del repuesto, además de actualizar el valor_total, reste del stock la cantidad utilizada.

4. Realice un trigger similar al anterior pero que actualice los valores de valor_total y stock_repuesto, cada vez que se ELIMINA un registro en REPUESTO_ORDEN.

5. Realice un trigger similar al anterior pero que actualice el valor de valor_total en ORDEN_MANTENIMIENTO y stock_repuesto en REPUESTO, cada vez que se ACTUALIZA el valor de cantidad_repuesto en REPUESTO_ORDEN. Debe considerar que el valor de cantidad_repuesto puede disminuir o puede aumentar.

Trigger

1. Construya un trigger en la tabla REPUESTO_ORDEN para que cada vez que se INSERTA un nuevo repuesto a una orden de mantenimiento, se actualice el campo valor_total correspondiente en la tabla ORDEN_MANTENIMIENTO.

```
CREATE OR REPLACE FUNCTION inserta_rep_orden()
RETURNS trigger AS $$
DECLARE
valor_rep integer;
BEGIN
Select into valor_rep   valor_repuesto from repuesto
Where cod_repuesto = new.cod_repuesto;

update orden_mantenimiento
set valor_total = valor_total + (new.cantidad_repuesto* valor_rep)
where orden_mantenimiento.cod_orden = new.cod_orden;
return new;
END; $$ LANGUAGE 'plpgsql';

CREATE TRIGGER inserta_rep_orden
  BEFORE INSERT ON repuesto_orden
  FOR EACH ROW  EXECUTE PROCEDURE inserta_rep_orden();
```

Repuesto

cod_repuesto [PK] integer	nombre_repuesto character varying (2	valor_repuesto integer	stock_repuesto integer
123	motor	10000	5
124	manilla	300	10
125	estanque	5000	8
126	medidor	2500	2
127	polea	500	15

Orden_mantenimiento (antes de insertar)

cod_orden [PK] integer	rut integer	id_empresa integer	fecha_orden date	valor_total integer
4545	222	12	2016-03-30	[null]
5656	111	12	2016-06-12	7000
6767	111	13	2015-11-25	20000
8989	111	12	2014-06-12	[null]

20000+(2*2500)= 25000

Repuesto_orden INSERTA

	cod_orden [PK] integer	cod_repuesto [PK] integer	cantidad_repuesto integer
1	5656	125	1
2	5656	127	4
3	6767	123	2
4+	6767	126	2

new.cod_orden

new.cantidad_repuesto

new.cod_repuesto

	cod_orden [PK] integer	rut integer	id_empresa integer	fecha_orden date	valor_total integer
1	4545	222	12	2016-03-30	[null]
2	5656	111	12	2016-06-12	7000
3	6767	111	13	2015-11-25	25000
4	8989	111	12	2014-06-12	[null]

Orden_mantenimiento (después de insertar)

Trigger

2. Realice las modificaciones necesarias para que en el trigger anterior se verifique previamente que la cantidad de repuesto del nuevo registro, es menor o igual al stock_repuesto en la tabla REPUESTO

```
CREATE OR REPLACE FUNCTION inserta_rep_orden()
RETURNS trigger AS $$
DECLARE
valor_rep integer;
stock_rep integer;
BEGIN
Select into valor_rep  stock_rep  valor_repuesto, stock_repuesto
from repuesto
Where cod_repuesto = new.cod_repuesto;

IF (new.cantidad_repuesto <= stock_rep) THEN
    update orden_mantenimiento
   set valor_total = valor_total + (new.cantidad_repuesto* valor_rep)
    where orden_mantenimiento.cod_orden = new.cod_orden;
    return new;
ELSE
    RAISE EXCEPTION 'No hay suficiente stock';
    return NULL;
END IF;
END; $$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER inserta_rep_orden
  BEFORE INSERT ON repuesto_orden
  FOR EACH ROW  EXECUTE PROCEDURE inserta_rep_orden();
```

Repuesto

cod_repuesto [PK] integer	nombre_repuesto character varying (2	valor_repuesto integer	stock_repuesto integer
123	motor	10000	5
124	manilla	300	10
125	estanque	5000	8
126	medidor	2500	2
127	polea	500	15

Orden_mantenimiento (antes de insertar)

cod_orden [PK] integer	rut integer	id_empresa integer	fecha_orden date	valor_total integer
4545	222	12	2016-03-30	[null]
5656	111	12	2016-06-12	7000
6767	111	13	2015-11-25	20000
8989	111	12	2014-06-12	[null]

Repuesto_orden **INSERTA**

	cod_orden [PK] integer	cod_repuesto [PK] integer	cantidad_repuesto integer
1	5656	125	1
2	5656	127	4
3	6767	123	2
4+	6767	126	2

Trigger

3. Nuevamente realice las modificaciones necesarias para que en el trigger anterior, si la cantidad de repuesto es menor o igual al stock del repuesto, además de actualizar el valor_total, reste del stock la cantidad utilizada.

¿Qué se debe modificar y/o agregar a esta función?

```
CREATE OR REPLACE FUNCTION inserta_rep_orden()
RETURNS trigger AS $$
DECLARE
valor_rep integer;
stock_rep integer;
BEGIN
Select into valor_rep  stock_rep  valor_repuesto, stock_repuesto
from repuesto
Where cod_repuesto = new.cod_repuesto;

IF (new.cantidad_repuesto <= stock_rep) THEN
    update orden_mantenimiento
   set valor_total = valor_total + (new.cantidad_repuesto* valor_rep)
    where orden_mantenimiento.cod_orden = new.cod_orden;
    return new;
ELSE
    RAISE EXCEPTION 'No hay suficiente stock';
    return NULL;
END IF;
END; $$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER inserta_rep_orden
  BEFORE INSERT ON repuesto_orden
  FOR EACH ROW  EXECUTE PROCEDURE inserta_rep_orden();
```

Repuesto

cod_repuesto [PK] integer	nombre_repuesto character varying (2	valor_repuesto integer	stock_repuesto integer
123	motor	10000	5
124	manilla	300	10
125	estanque	5000	8
126	medidor	2500	2
127	polea	500	15

Orden_mantenimiento (antes de insertar)

cod_orden [PK] integer	rut integer	id_empresa integer	fecha_orden date	valor_total integer
4545	222	12	2016-03-30	[null]
5656	111	12	2016-06-12	7000
6767	111	13	2015-11-25	20000
8989	111	12	2014-06-12	[null]

Repuesto_orden **INSERTA**

	cod_orden [PK] integer	cod_repuesto [PK] integer	cantidad_repuesto integer
1	5656	125	1
2	5656	127	4
3	6767	123	2
4+	6767	126	2

Trigger

4. Realice un trigger similar al anterior pero que actualice los valores de valor_total y stock_repuesto, cada vez que se ELIMINA un registro en REPUESTO_ORDEN.

5. Realice un trigger similar al anterior pero que actualice el valor de valor_total en ORDEN_MANTENIMIENTO y stock_repuesto en REPUESTO, cada vez que se ACTUALIZA el valor de cantidad_repuesto en REPUESTO_ORDEN. Debe considerar que el valor de cantidad_repuesto puede disminuir o puede aumentar.