

# Sistemas Informáticos

---

Tema 3: Bases de datos distribuidas

## 3.3 Procedimientos almacenados y *Triggers*

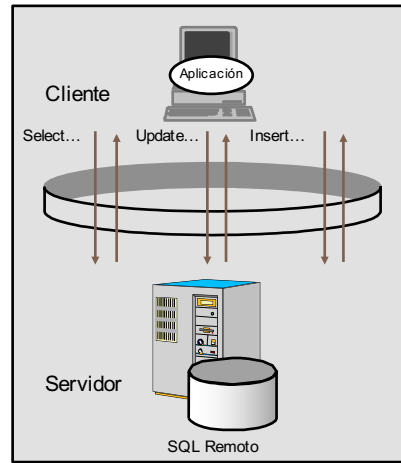
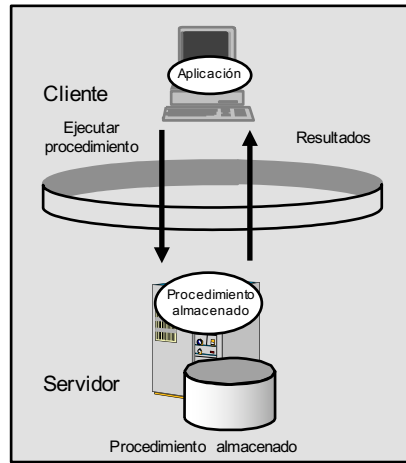


### Procedimientos almacenados

- Conjunto de sentencias SQL y lógica de programa compilado, verificado y almacenado en el servidor de base de datos.
- Tratado por el servidor como cualquier otro objeto de la base de datos y almacenado en el catálogo de la misma.
- Su acceso está controlado por los mecanismos de seguridad.
- Aceptan parámetros de entrada.
- Mejora de rendimiento al disminuir el tráfico por la red.
- No hay estándares. Implementación propia de cada fabricante.

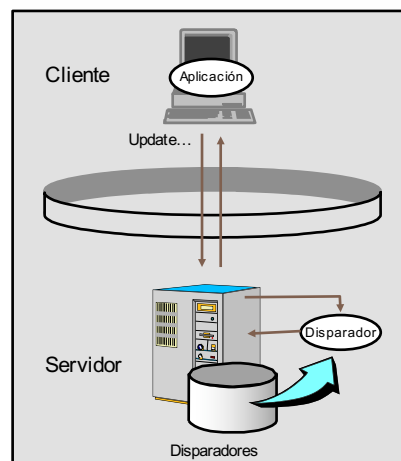


## Procedimientos almacenados / SQL remoto



## Caso particular: Disparador (trigger)

- Los disparadores (*triggers*) son casos particulares de procedimientos almacenados.
- Procedimientos invocados automáticamente por el servidor cuando ocurren determinados eventos sobre la base de datos (INSERT, DELETE, UPDATE).
- **Implementaciones dependientes del fabricante.**



## PL/pgSQL

- Desde 1997 PostgreSQL soporta el uso de procedimientos almacenados.
- Lenguaje: PL/pgSQL
- Permite crear **funciones** que se ejecutan en el servidor.
- La propia base de datos se encarga de compilar y gestionar estas funciones con lo que suelen ser eficientes
- Proporciona: uso de variables, bucles y evaluación condicional,



## PL/pgSQL: Generalidades

- Los tipos de datos pasados a la función están entre paréntesis
- El cuerpo de la función se pasa a la base de datos como una cadena de caracteres (véase que el cuerpo empieza y acaba con comillas simples)
- Tras la cadena se indica el lenguaje usado para crear la función, se define usando la orden "LANGUAGE" (otros lenguajes posibles son PL/PERL, PL/TCL, C, etc.)



## Un paso previo a usar PL/pgSQL

- Cuando se crea una base de datos nueva hace falta "autorizar" el uso de PL/pgSQL (a menos que template1 ya esté autorizada)

```
createdb mydatab  
createlang plpgsql mydatab
```

- En los laboratorios debería estar "autorizado" por defecto
- Ojo: no se comprueba la sintaxis de las funciones hasta que no son ejecutadas. (Es difícil depurar el código)



## PL/pgSQL: Estructura de las funciones

- PL/pgSQL presenta una estructura en "Bloques".
- Cada bloque se define usando

```
DECLARE  
    --variables inicializadas con NULL cada vez  
    --que se entra en el bloque  
    [...]  
BEGIN  
    --comandos;  
    [...]  
END;
```
- No se pueden definir transacciones dentro de una función



# Sistemas Informáticos

Tema 3.2: SQL

## Procedimientos almacenados



### Estructura de las funciones

```
CREATE FUNCTION nombre_funcion (tipos-argumentos)
  RETURNS integer AS $$
DECLARE
  -- declarations
BEGIN
  PERFORM my_function();
END;
$$ LANGUAGE plpgsql;
```

Puede ser '



## Lenguajes de los SP

- Lenguajes soportados por omisión: SQL, PL/pgSQL, C
  - SQL y PL/pgSQL son seguros, cualquier puede usarlos.
  - C es sólo para usuarios privilegiados, porque permiten acceder a recursos externos.
- Se pueden cargar en el sistema otros lenguajes:
  - Perl
  - Python
  - TCL
  - PostgreSQL



## SQL “puro”

- Ejecutan una lista arbitraria de sentencias SQL, retornando el resultado de la última consulta en la lista.
  - No puede contener comandos que alteren los catálogos del sistema (por ejemplo CREATE TABLE); si los contiene no tienen efecto.

```
CREATE FUNCTION clean_emp() RETURNS void AS '  
    DELETE FROM emp  
        WHERE salary < 0;  
' LANGUAGE SQL;  
  
SELECT clean_emp();  
  
clean_emp  
-----  
(1 row)
```



## Paso de argumentos

- Los argumentos pueden ser referenciados por nombre o por número (posición).

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$  
    SELECT $1 + $2;  
$$ LANGUAGE SQL;
```

```
SELECT add_em(1, 2) AS answer;
```

```
answer  
-----  
      3
```

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$  
    SELECT x + y;  
$$ LANGUAGE SQL;
```



## Argumentos complejos

- Suponga la siguiente tabla:

```
CREATE TABLE emp (  
    name      text,  
    salary    numeric,  
    age       integer,  
    cubicle   point  
);
```

- A la cual se pueden agregar filas con el siguiente comando:

```
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');
```



## Argumentos complejos (ii)

- Una función que recibe una fila de la tabla **emp** como argumento:

```
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

- Y luego puede ser invocada así:

```
SELECT name, double_salary(emp.*) AS dream
FROM emp
WHERE emp.cubicle ~= point '(2,1)';
```

name	dream
Bill	8400



## SQL "puro" (ii)

- Si la última consulta retorna más de un resultado, la función retorna la primera fila.
  - Debe recordarse que la primera fila no está definida, excepto que se use el modificador ORDER BY
- Si la última cláusula no es un SELECT
  - Puede ser INSERT, UPDATE, DELETE
- ... debe tener clausula RETURNING

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```





## Elementos adicionales

- Se puede declarar una función que retorne más de un elemento
  - SET OF, TABLE
- Se pueden declarar variables, retornos de función, o argumentos que contentan filas
  - RECORD, %ROWTYPE
- Se pueden declarar funciones con argumentos “de salida”
- Estas características se verán con el lenguaje PL/pgSQL, donde se usan de forma similar.



## PL/pgSQL

- Una extensión del SQL puro, que pretende:
  - Mantener una estructura muy similar y tan simple como SQL
  - Incorporar elementos que permitan mayor control sobre la ejecución:
    - Sentencias de control de flujo
    - Variables
    - Excepciones



## Ejemplo trivial sin pasar parámetros

- ¿Qué hace esta función?

```
CREATE OR REPLACE FUNCTION una_funcion () RETURNS
int4 AS $$
DECLARE
    an_integer int4;          --variables
BEGIN
    an_integer := 10 * 10;    --comandos
    RETURN an_integer;
END;
$$ LANGUAGE plpgsql;
```

```
select una_funcion();
una_funcion
-----
        100
(1 row)
```



## Tipos de Variables

- Ejemplos de variables:

```
id_usuario INTEGER;
cantidad NUMERIC(5);
url VARCHAR;
```

```
-- tipos relativos a campos o tuplas de una base de datos
micampo mitabla.campo%TYPE;
mitupla mitabla%ROWTYPE;
```

- Todos los tipos de variable definidos para SQL son válidos en PL/pgSQL
- La sintaxis general para la declaración de variables:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := }
expression ];
```



## Más sobre Variables

- CREATE FUNCTION mifuncion(INTEGER, CHAR, ...)
- Se pueden pasar hasta 16 variables
  - \$1, \$2, ..., \$16
- ALIAS permite renombrar variables

```
CREATE FUNCTION cal_longitud (text) RETURNS int4 AS $$  
DECLARE  
  intext ALIAS FOR $1; --primer parametro  
  resultado int4;  
.  
.  
.
```



## Ejemplo trivial pasando variables

```
CREATE OR REPLACE FUNCTION cal_longitud (text)  
  RETURNS int4 AS $$  
DECLARE  
  intext ALIAS FOR $1; --primer parametro  
  resultado int4;  
BEGIN  
  resultado := (SELECT LENGTH(intext));  
  RETURN resultado;  
END;  
$$ LANGUAGE plpgsql;
```

```
SELECT cal_longitud('palabra');
```

```
cal_longitud  
-----  
              7  
(1 row)
```



## Argumentos por nombre

```
CREATE OR REPLACE FUNCTION get_sum(  
    a NUMERIC,  
    b NUMERIC)  
    RETURNS NUMERIC AS $$  
  
BEGIN  
    RETURN a + b;  
END; $$  
LANGUAGE plpgsql;
```



## Argumentos de salida

```
CREATE OR REPLACE FUNCTION min_max(  
    a NUMERIC,  
    b NUMERIC,  
    c NUMERIC,  
    OUT min NUMERIC,  
    OUT max NUMERIC)  
    AS $$  
  
BEGIN  
    min = LEAST(a, b, c);  
    max = GREATEST(a, b, c);  
END; $$  
LANGUAGE plpgsql;
```



## Argumentos de salida (ii)

- Cuidado: la salida crea un objeto del tipo record

```
1 SELECT hi_lo(10,20,30);
```

hi_lo
(30,10)

- Si lo quiero en columnas separadas:

```
1 SELECT * FROM hi_lo(10,20,30);
```

hi	lo
30	10



## Ejemplos argumentos de salida

- En SQL

```
CREATE OR REPLACE FUNCTION fn_sqltestout(param_subject text,
    OUT subject_scramble text, OUT subject_char text)
AS
$$
    SELECT substring($1, 1,CAST(random()*length($1) As integer)), substring($1, 1,1)
    $$
LANGUAGE 'sql';
```

- Esta función retorna 1 elemento (compuesto)



## Ejemplos argumentos de salida (ii)

- Usos:

```
SELECT (fn_sqltestout('This is a test subject')).subject_scramble;
```

```
-- Output
```

```
subject_scramble
```

```
-----
```

```
This is a test
```

```
SELECT (fn_sqltestout('This is a test subject')).*;
```

```
--Output
```

```
subject_scramble | subject_char
```

```
-----+-----
```

```
This is a test subje | T
```



## Ejemplos argumentos de salida

- Ahora lo mismo en PL/pgSQL:

```
CREATE OR REPLACE FUNCTION fn_plpgsqltestout(param_subject text,
  OUT subject_scramble text, OUT subject_char text)
AS
$$
BEGIN
  subject_scramble := substring($1, 1, CAST(random()*length($1) As integer));
  subject_char := substring($1, 1, 1);
END;
$$
LANGUAGE 'plpgsql';
```



## Copiando tipos

- Se puede definir una variable cuyo tipo sea una referencia al tipo de una columna o de otra variable

```
1 variable_name table_name.column_name%TYPE;
```

```
1 variable_name variable%TYPE;
```

- Por ejemplo:

```
1 city_name city.name%TYPE := 'San Francisco';
```



## Ejemplo usando Rowtype

```
CREATE OR REPLACE FUNCTION trae_pelicula (integer)
  RETURNS text AS $$
  DECLARE
    pelicula_id ALIAS FOR $1;
    encontrada_pelicula pelicula%ROWTYPE;
  BEGIN
    SELECT INTO encontrada_pelicula * FROM pelicula
      WHERE id = pelicula_id;
    RETURN encontrada_pelicula.titulo || " (" ||
      encontrada_pelicula.agno || ")";
  END;
  $$ LANGUAGE plpgsql;
```

- Nota: Si SELECT INTO devuelve más de una tupla se ignoran todas menos la primera (la solución a esto más tarde)



## Ejemplo usando Rowtype

```
nueva=> select trae_pelicula(3);
```

trae_pelicula	
Blade Runner	(1982)

(1 row)



## Control de Flujo

- PL/pgSQL contiene estructuras de control que permiten seleccionar las líneas de código que serán ejecutadas en tiempo real.
- IF...THEN...ELSE...ELSE IF
  - ejecución condicional
- LOOPS, WHILE LOOPS, FOR LOOPS
  - iteraciones
  - bucles





## Ejemplo IF/ELSE

- Programa que calcula la longitud de dos cadenas y devuelve la longitud mayor.

```
CREATE OR REPLACE FUNCTION cadena_mas_larga(text, text)
  RETURNS int4 AS $$
DECLARE
  in_uno ALIAS FOR $1;
  in_dos ALIAS FOR $2;
  lon_uno int4;
  lon_dos int4;
  result int4;
BEGIN
  lon_uno := (SELECT LENGTH(in_uno));
  lon_dos := (SELECT LENGTH(in_dos));
  IF lon_uno > lon_dos THEN RETURN lon_uno;
  ELSE RETURN lon_dos;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

nueva=> SELECT cadena\_mas\_larga('hola','adios');

```
cadena_mas_larga
-----
                    5
(1 row)
```



## Ejemplo bucle WHILE

Función que cuenta cuantas veces aparece un carácter en una cadena

```
CREATE FUNCTION cuentac(text,text)
  RETURNS INT4 As $$
DECLARE
  intext ALIAS FOR $1; inchar ALIAS FOR $2;
  lon int4; resultado int4;
  i int4; tmp char;
BEGIN
  lon:= length(intext); i:=1;
  resultado:=0;
  WHILE i<= lon LOOP
    tmp := substr(intext,i,1);
    IF tmp = inchar THEN
      resultado := resultado +1;
    END IF;
    i:=i+1;
  END LOOP;
  RETURN resultado;
END
$$ LANGUAGE plpgsql;
```

-- SELECT cuentac('qwertyq','q');

```
cuentac
-----
                2
```



## Excepciones

- RAISE se usa para imprimir mensajes y, en el caso de excepción, abortar la transacción
- RAISE { NOTICE | EXCEPTION }
- RAISE NOTICE
  - RAISE NOTICE 'No hagas eso!';
  - RAISE NOTICE 'El señor' || id || 'no está en casa';
  - RAISE NOTICE 'el señor % no está en casa', id;



## Excepciones: Ejemplo

Calcular la suma de los enteros de n a m, usando la formula  $(p+1)*p/2$

```
CREATE OR REPLACE FUNCTION suma(int4, int4) RETURNS int4
AS $$
DECLARE
    inicio ALIAS FOR $1; fin ALIAS FOR $2;
    resultado int;
BEGIN
    IF (inicio < 1) THEN
        RAISE EXCEPTION "inicio debe ser mayor que 1";
    ELSE
        IF (inicio <= fin) THEN
            resultado := (fin+1)*fin/2 -
                        (inicio-1)*inicio/2;
        ELSE
            RAISE EXCEPTION "El valor inicial % debe ser menor
que el final %", inicio, fin;
        END IF;
    END IF;
    RETURN resultado;
END
$$ LANGUAGE plpgsql;
--SELECT suma(1,3);
--SELECT suma(-1,5);
--SELECT suma(8,5);
```

```
nueva=> SELECT suma(1,3);
        suma
        -----
           6
        (1 row)
```



## SELECT y Bucles

- Cuántas tuplas empiezan con una letra determinada

```
CREATE OR REPLACE FUNCTION cuenta_letra (text) RETURNS
int4 AS $$
DECLARE
    caracter ALIAS FOR $1; temporal record;
    tmp_caracter text; resultado int4;
BEGIN
    resultado:=0;
    FOR temporal IN SELECT titulo FROM pelicula LOOP
        tmp_caracter :=substr(temporal.titulo,1,1);
        IF tmp_caracter = caracter THEN
            resultado := resultado +1;
        END IF;
    END LOOP;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

nueva=> SELECT cuenta\_letra('A');  
cuenta\_letra  
-----  
93  
(1 row)



## SELECT y Bucles

- Nombres de las películas que empiezan con una letra determinada (pelis\_con\_letra)

```
CREATE OR REPLACE FUNCTION pelis_con_letra (text) RETURNS text
AS $$
DECLARE
    caracter ALIAS FOR $1; temporal record;
    tmp_caracter text;
    total text;
BEGIN
    total := "";
    FOR temporal IN SELECT titulo FROM pelicula LOOP
        tmp_caracter :=substr(temporal.titulo,1,1);
        IF tmp_caracter = caracter THEN
            total := total || temporal.titulo || " . ";
        END IF;
    END LOOP;
    RETURN total;
END;
$$ LANGUAGE plpgsql;
```



```
--SELECT pelis_con_letra('A');
```

## Ejemplo de SELECT y Bucles

```

Aliens . Apocalypse Now . Alien . Apollo 13 . Amadeus . Abyss, The . Airplane!
. Army of Darkness . Aladdin . Air Force One . Alien 3 . Annie Hall . African Qu
een, The . Alien: Resurrection . Austin Powers: International Man of Mystery . A
kira . Adventures of Priscilla, Queen of the Desert, The . American President, T
he . Adventures of Buckaroo Banzai Across the 8th Dimension, The . American Graf
fici . Ace Ventura: Pet Detective . Awakenings . Adventures of Baron Munchausen,
The . As Good As It Gets . Arsenic and Old Lace . Addams Family, The . American
Werewolf in London, An . Angel Heart . All the President's Men . Airplane II: T
he Sequel . After Hours . Ace Ventura: When Nature Calls . Absolute Power . Acci
dental Tourist, The . Andromeda Strain, The . Adventures in Babysitting . Asachn
ophobia . Age of Innocence, The . Adventures of Ford Fairlane, The . Always . Am
istad . Arthur . Arrival, The . Anaconda . Amazon Women on the Moon . Addicted t
o Love . Addams Family Values . All of Me . About Last Night... . All About Eve
. Adventures of Robin Hood, The . All That Jazz . Accused, The . Altered States
. Apartment, The . Assassins . And Now for Something Completely Different . Anne
of Green Gables . Antonia . Alien Nation . Anastasia . Arizona Dream . Alive .
Au revoir les enfants . Annie . Amateur . Auntie Mame . Against All Odds . Airpo
rt . Absence of Malice . Affair to Remember, An . Anamcord . Atlantic City . Agn
es of God . American in Paris, An . All the Right Moves . Airheads . Around the
World in 80 Days . Air America . Aguirre, der Zorn Gottes . Amityville Horror, T
he . Angels and Insects . Anant, L . American Tail, An . American Werewolf in P
aris, An . Any Which Way You Can . Adam's Rib . Alice in Wonderland . Attack of
the Killer Tomatoes! . Another 48 HRS . Arthur 2: On the Rocks . Action Jackson
. Allan Quatermain and the Lost City of Gold .
<1 row>
melis=#

```



## Para retornar conjunto de valores (filas)

- Cuatro formas de retornar 1 fila
  - RETURNS RECORD
  - RETURNS nombreTabla%ROWTYPE
  - Como argumentos de salida (visto en los ejemplos anteriores)
  - RETURNS TABLE



## RETURNS TABLE, en SQL

- En SQL

```
CREATE OR REPLACE FUNCTION fn_sqltestout(param_subject text, pos integer)
RETURNS TABLE(subject_scramble text, subject_char text)
AS
$$
    SELECT substring($1, 1, CAST(random()*length($1) As integer)) ,
           substring($1, 1,1) As subject_char;
    $$
LANGUAGE 'sql';
-- example use
SELECT (fn_sqltestout('This is a test subject')).subject_scramble;
SELECT subject_scramble, subject_char FROM fn_sqltestout('This is a test subject');
```


- En PL/pgSQL

```
CREATE OR REPLACE FUNCTION fn_plpgsqltestout(param_subject text)
RETURNS TABLE(subject_scramble text, subject_char text)
AS
$$
BEGIN
    subject_scramble := substring($1, 1, CAST(random()*length($1) As integer));
    subject_char := substring($1, 1,1);
    RETURN NEXT;
END;
$$
LANGUAGE 'plpgsql';
```

## Retornar conjuntos de valores

- En los ejemplos anteriores, si la consulta cuyo valor es devuelto produce más de una resultado (múltiples filas), se devuelve la primera fila de ese resultado.
- Si queremos que se devuelvan todas las filas, se puede usar:
  - RETURNS SET OF tipo\_de\_valor
  - RETURNS TABLE(columnas)
  - Argumentos de Salida





```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;
```

name	parent
Top	
Child1	Top
Child2	Top
Child3	Top
SubChild1	Child1
SubChild2	Child1

(6 rows)

```
SELECT listchildren('Top');
listchildren
```

Child1
Child2
Child3

(3 rows)


```
SELECT name, listchildren(name) FROM nodes;
```

name	listchildren
Top	Child1
Top	Child2
Top	Child3
Child1	SubChild1
Child1	SubChild2

(5 rows)

```
CREATE OR REPLACE FUNCTION storeopeninghours_tostring(numeric)
    RETURNS SETOF RECORD AS $$
DECLARE
    open_id ALIAS FOR $1;
    result RECORD;
BEGIN
    RETURN QUERY SELECT '1', '2', '3';
    RETURN QUERY SELECT '3', '4', '5';
    RETURN QUERY SELECT '3', '4', '5';
END
$$;
```

If you have a record or row variable to return (instead of a query result), use "RETURN NEXT" rather than "RETURN QUERY".



### Store procedure: devolver tabla

```

1 CREATE OR REPLACE FUNCTION get_film (p_pattern VARCHAR)
2 RETURNS TABLE (
3   film_title VARCHAR,
4   film_release_year INT
5 )
6 AS $$
7 BEGIN
8   RETURN QUERY SELECT
9     title,
10    cast( release_year as integer)
11 FROM
12   film
13 WHERE
14   title LIKE p_pattern ;
15 END; $$
16
17 LANGUAGE 'plpgsql';

```

```

--SQL returning multiple records
CREATE OR REPLACE FUNCTION fn_sqltestmulti(param_subject varchar,
OUT test_id integer,
OUT test_stuff text)
RETURNS SETOF record
AS
$$
SELECT test_id, test_stuff
FROM testtable where test_stuff LIKE $1;
$$
LANGUAGE 'sql' VOLATILE;

--example
SELECT * FROM fn_sqltestmulti('%stuff%');
--example
--OUTPUT--
test_id | test_stuff
-----+-----
1 | this is more stuff
2 | this is new stuff

```

# Sistemas Informáticos

---

Tema 3.2: SQL

## Triggers (disparadores)



### Triggers o "gatillos"

- Un **trigger o gatillo** es una orden SQL que se ejecuta automáticamente como resultado de una modificación de la base de datos:
  - Se activan cuando cierto eventos (especificados por el creador de la base de datos) ocurren
  - El primer paso requiere comprobar una condición
  - Si la condición se satisface el trigger se ejecuta





## Triggers en PostgreSQL

- El comando CREATE TRIGGER en postgresql implementa solo un subconjunto del estandar SQL-99

- En particular:

- Postgresql sólo permite la ejecución de una función (procedure) como acción "disparada" por el trigger.
- Dentro de un trigger no se pueden declarar transacciones

```
begin
```

```
...
```

```
end
```

- Se sugiere probar los siguientes ejemplos de triggers
- Para ello, mejor probar cada ejemplo en una base de datos nueva y vacía.
- Es interesante ver que producen en cada ejemplo las inserciones o modificaciones que se proponen al final de uno.



## Ejemplo

```
DROP TABLE producto2 cascade;  
CREATE TABLE producto2(id int PRIMARY KEY, vendido timestamp);
```

```
DROP FUNCTION modificacion1() cascade;  
CREATE FUNCTION modificacion1() RETURNS TRIGGER AS $$  
BEGIN  
  NEW.vendido := "now()";  
  RETURN NEW; END; $$ LANGUAGE plpgsql;
```

```
DROP TRIGGER t_modificacion1 on producto2;  
CREATE TRIGGER t_modificacion1 BEFORE INSERT ON producto2  
FOR EACH ROW EXECUTE PROCEDURE modificacion1();
```

```
INSERT into producto2 VALUES (1);  
SELECT * FROM producto2;
```



### Ejemplo

```
DROP TABLE producto2 cascade;  
CREATE TABLE producto2(id int PRIMARY KEY, vendido timestamp);  
  
DROP FUNCTION modificacion2() cascade;  
CREATE FUNCTION modificacion2() RETURNS TRIGGER AS $$  
BEGIN  
    NEW.vendido := "now()";  
    RETURN NEW; END; $$ LANGUAGE plpgsql;  
  
DROP TRIGGER t_modificacion2 on producto2;  
CREATE TRIGGER t_modificacion2 AFTER INSERT ON producto2  
FOR EACH ROW EXECUTE PROCEDURE modificacion2();  
  
INSERT into producto2 VALUES (28);  
SELECT * FROM producto2;
```



### Ejemplo

```
DROP TABLE producto2 cascade;  
CREATE TABLE producto2(id int PRIMARY KEY, vendido timestamp);  
  
DROP FUNCTION modificacion3() cascade;  
CREATE FUNCTION modificacion3() RETURNS TRIGGER AS $$  
BEGIN  
    OLD.vendido := "now";  
    RETURN NEW; END; $$ LANGUAGE plpgsql;  
  
DROP TRIGGER t_modificacion3 on producto2;  
CREATE TRIGGER t_modificacion3 AFTER INSERT ON producto2  
FOR EACH ROW EXECUTE PROCEDURE modificacion3();  
  
INSERT into producto2 VALUES (11);  
SELECT * FROM producto2;
```



## Ejemplo

```
DROP TABLE producto2 cascade;
CREATE TABLE producto2( id int PRIMARY KEY, vendido timestamp, otro
int);
```

```
DROP FUNCTION modificacion4() cascade;
CREATE FUNCTION modificacion4() RETURNS TRIGGER AS $$
BEGIN
  RAISE NOTICE " VALOR OLD: % ", OLD.otro;
  RAISE NOTICE " VALOR NEW: % ", NEW.otro;
  NEW.otro := 10;
  NEW.vendido:="now";
  RETURN NEW; END; $$ LANGUAGE plpgsql;
```

```
DROP TRIGGER t_modificacion4 on producto2;
CREATE TRIGGER t_modificacion4 BEFORE UPDATE ON producto2
FOR EACH ROW EXECUTE PROCEDURE modificacion4();
```

```
INSERT into producto2 VALUES (1);
UPDATE producto2 SET otro=8 WHERE id=1;
SELECT * FROM producto2;
```



## Ejemplo

```
DROP TABLE producto2 cascade;
CREATE TABLE producto2( id int PRIMARY KEY, vendido timestamp, otro
int);
```

```
DROP FUNCTION modificacion4() cascade;
CREATE FUNCTION modificacion4() RETURNS TRIGGER AS $$
BEGIN
  RAISE NOTICE " VALOR OLD: % ", OLD.otro;
  RAISE NOTICE " VALOR NEW: % ", NEW.otro;
  NEW.otro := 10;
  NEW.vendido:="now";
  RETURN NEW; END; $$ LANGUAGE plpgsql;
```

```
DROP TRIGGER t_modificacion4 on producto2;
CREATE TRIGGER t_modificacion4 AFTER UPDATE ON producto2
FOR EACH ROW EXECUTE PROCEDURE modificacion4();
```

```
INSERT into producto2 VALUES (1);
UPDATE producto2 SET otro=8 WHERE id=1;
SELECT * FROM producto2;
```



