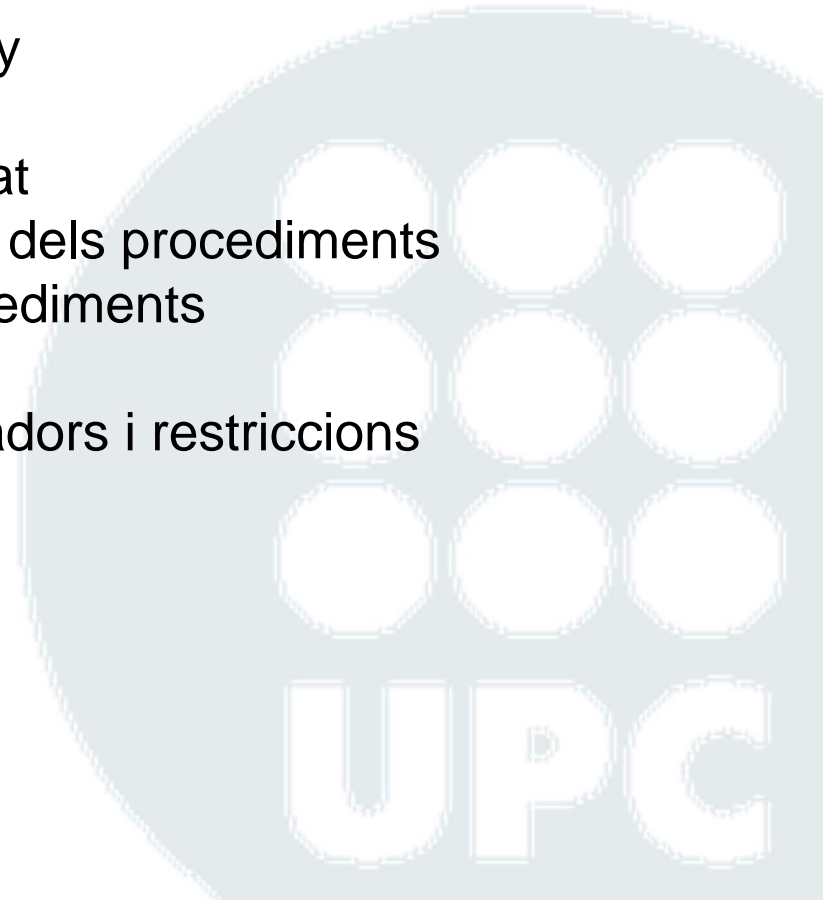


## 4.2 Disparadors en PostgreSQL

- Consideracions de disseny
- Sintaxis
- Ordre d'execució i visibilitat
- Variables accessibles des dels procediments
- Valors de retorn dels procediments
- Exemples
- Precedència entre disparadors i restriccions

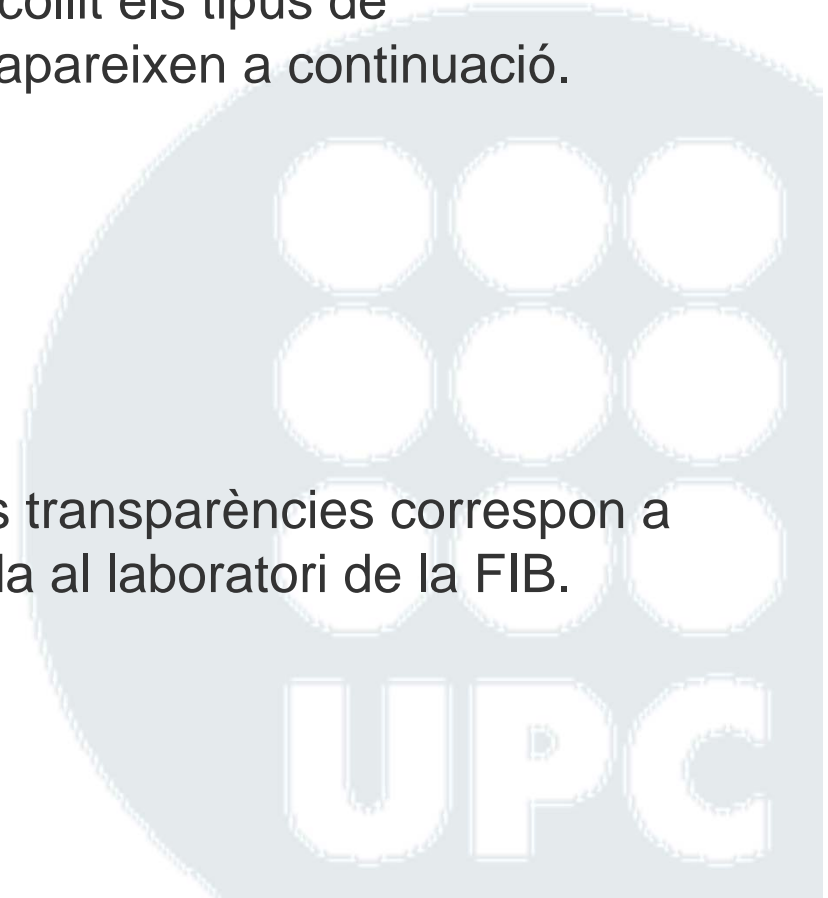


## Consideracions de disseny

Cal tenir en compte les consideracions de disseny explicades a la classes del tema ***Components lògics d'una base de dades*** per entendre com s'han escollit els tipus de disparadors en els exemples que apareixen a continuació.

## Disparadors en PostgreSQL

La sintaxis que s'explica aquestes transparències correspon a la versió de PostgreSQL instal·lada al laboratori de la FIB.



## Sintaxis

```
CREATE TRIGGER nom { BEFORE | AFTER }
{ esdeveniment [ OR ... ] } ON taula
[ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE nomFunc
```

Procediment emmagatzemat, específic per a triggers. Agrupa les accions que s'han de fer quan es produeix l'esdeveniment/s.

*esdeveniment* ::= [INSERT | DELETE | UPDATE [of column,.. ]]

Begin work

...

delete from empleats ...

...

commit

CREATE TRIGGER Esborrar1  
BEFORE delete ON empleats  
FOR EACH STATEMENT  
Execute procedure.....

CREATE TRIGGER Esborrar2  
AFTER delete ON empleats  
FOR EACH STATEMENT  
Execute procedure.....

## Ordre d'execució i visibilitat

### 1. BEFORE STATEMENT :

L'acció s'executa **1 sola vegada abans de l'execució de la sentència** que dispara el disparador.

### 2. BEFORE ROW

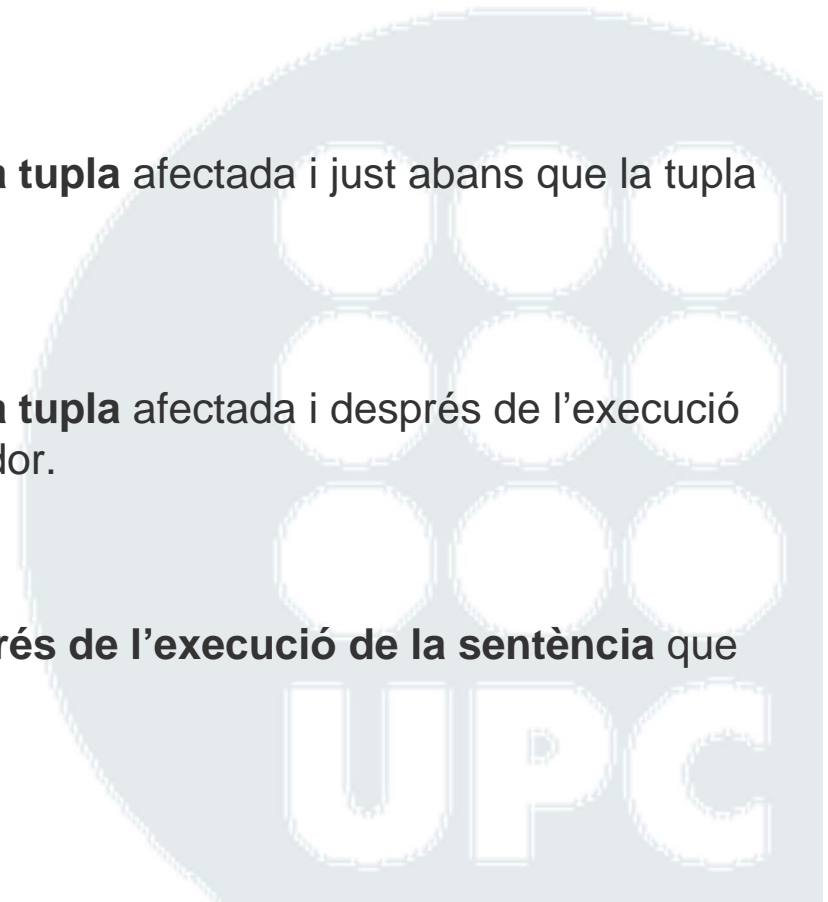
L'acció s'executa **1 vegada per a cada tupla** afectada i just abans que la tupla s'insereixi, modifiqui o esborri.

### 3. AFTER ROW

L'acció s'executa **1 vegada per a cada tupla** afectada i després de l'execució de la sentència que dispara el disparador.

### 4. AFTER STATEMENT

L'acció s'executa **1 sola vegada després de l'execució de la sentència** que dispara el disparador.



## Sintaxis dels procediments específics per a disparadors

```
CREATE FUNCTION disp_procediment()  
RETURNS trigger AS $$  
BEGIN  
...  
...  
...  
RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Procediment que no rep cap paràmetre i retorna un tipus especial de postgres anomenat trigger

Només pot retornar: NULL, NEW o OLD.  
(veure transparència següent)

## Variables accessibles des dels procediments

- **TG\_OP**: conté l'esdeveniment que llança el disparador. Els seus valors poden ser (en majúscules!): INSERT, DELETE i UPDATE.
- **OLD, NEW**:
  - Només tenen valor per triggers FOR EACH ROW.
  - Tenen valor NULL en triggers FOR EACH STATEMENT.
  - **NEW**: Valors de la tupla després de l'execució de la sentència update o insert. No té valor definit en sentències delete.
  - **OLD**: Valors de la tupla abans de l'execució de la sentència update o delete. No té valor definit en sentències insert.

**Exemple:** En un procediments que s'executa degut a un trigger FOR EACH ROW activat per la sentència:

```
UPDATE items SET qtt=qtt+10 WHERE item=1;
```

Els valors de les variables NEW i OLD dins del procediment seran:

```
OLD.item = 1  
OLD.qtt = 100  
NEW.item = 1  
NEW.qtt = 110
```

- Hi ha **altres variables**, que no usarem!!. Consultar el manual de PostgreSQL.

## Return dels procediments BEFORE, FOR EACH ROW

Aquests procediments poden retornar els valors següents:

- **NEW**, per indicar que l'execució del procediment per la fila actual ha d'acabar normalment i que l'operació que dispara el disparador (INSERT/UPDATE) s'ha d'executar. En aquest cas, el procediment pot retornar el valor original de la variable NEW o modificar-ne el contingut. Si el procediment modifica el contingut de la variable NEW, està modificant directament la fila que s'insertirà o modificarà.
- **OLD**, per indicar que l'execució del procediment per la fila actual ha d'acabar normalment i que l'operació que dispara el disparador (DELETE/UPDATE) s'ha d'executar. En el cas de l'UPDATE si el procediment retorna OLD el UPDATE no fa la modificació de la fila actual.
- **NULL**, per indicar que l'execució del procediment per la fila actual ha d'acabar normalment i que l'operació que dispara el disparador (INSERT/ UPDATE/ DELETE) no s'arriba a executar.

## Exemple: Return dels procediments BEFORE, FOR EACH ROW, INSERT

```
CREATE TABLE t(  
    a integer PRIMARY KEY,  
    b integer);  
  
CREATE FUNCTION prog()  
....  
....  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trig_i BEFORE INSERT ON t FOR EACH ROW  
EXECUTE PROCEDURE prog();  
  
INSERT INTO t VALUES (1,2);
```

El procediment `prog()` pot retornar dos valors, segons el return usat:

- **RETURN NULL.** En aquest cas, la fila `<1,2>` no s'insereix a la taula `t`.
- **RETURN NEW.** En aquest cas, tenim dues possibilitats. Si el valor de la variable `NEW` no ha sigut modificat pel procediment `prog()`, aleshores s'executa l'inserció de la fila `<1,2>` a la taula `t`. Si el valor de la variable `NEW` ha sigut modificat pel procediment `prog()`, per exemple s'ha executat l'operació `NEW.b=3`, aleshores s'executa l'inserció de la fila `<1,3>` a la taula `t`.



## Exemple: Return dels procediments BEFORE, FOR EACH ROW, UPDATE

```
CREATE TABLE t(  
    a integer PRIMARY KEY,  
    b integer);  
  
CREATE FUNCTION prog()  
...  
...  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trig_u BEFORE UPDATE ON t FOR EACH ROW  
EXECUTE PROCEDURE prog();  
  
UPDATE t SET b=3 WHERE a=1;
```

El procediment `prog()` pot retornar tres valors, segons el return usat:

- **RETURN NULL** o **RETURN OLD**. En aquest cas, la fila `a=1` no es modifica en la taula `t`.
- **RETURN NEW**. En aquest cas, tenim dues possibilitats. Si el valor de la variable `NEW` no ha sigut modificat pel procediment `prog()`, aleshores s'executa la modificació de la fila `a=1` passant a ser `<1,3>`. Si el valor de la variable `NEW` ha sigut modificat pel procediment `prog()`, per exemple s'ha executat l'operació `NEW.b=5`, aleshores s'executa la modificació de la fila `a=1` passant a ser `<1,5>`.

## Exemple: Return dels procediments BEFORE, FOR EACH ROW, DELETE

```
CREATE TABLE t(  
    a integer PRIMARY KEY,  
    b integer);  
  
CREATE FUNCTION prog()  
    ....  
    ....  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trig_d BEFORE DELETE ON t FOR EACH ROW  
EXECUTE PROCEDURE prog();  
  
DELETE FROM t WHERE a=1;
```

El procediment prog() pot retornar tres valors, segons el return usat:

- RETURN NULL. En aquest cas, no s'executa l'esborrat de la fila a=1 la taula t.
- RETURN OLD. En aquest cas, s'executa l'esborrat de la fila a=1 en la taula t.

## Return dels procediments AFTER o STATEMENT

Pels disparadors AFTER/FOR EACH ROW, i els disparadors FOR EACH STATEMENT (independentment de si són BEFORE o AFTER) el valor retornat pel procediment que és invocat pel disparador és ignorat. Per aquest motiu, sovint aquests procediments retornen NULL.



## Valors de les variables NEW i OLD: RESUM

	ROW			STATEMENT
	INSERT	UPDATE	DELETE	INSERT, UPDATE, DELETE
BEFORE	<ul style="list-style-type: none"> <li>■ OLD: No té valor definit.</li> <li>■ NEW: Valors de la tupla després de l'insert</li> </ul>	<ul style="list-style-type: none"> <li>■ OLD: Valors de la tupla abans de l'update.</li> <li>■ NEW: Valors de la tupla després de l'update</li> </ul>	<ul style="list-style-type: none"> <li>■ OLD: Valors de la tupla abans del delete.</li> <li>■ NEW: No té valor definit.</li> </ul>	Valor NULL
AFTER				

ATENCIO: La modificació del valor de la variable NEW dins d'un procediment d'un trigger BEFORE INSERT / UPDATE pot afectar a l'efecte de la sentència que ha activat el trigger.

## Return dels procediments: RESUM

	ROW	STATEMENT
BEFORE	<ul style="list-style-type: none"> <li>■ RETURN NEW               <ul style="list-style-type: none"> <li>– A continuació s'executa la sentència que activa el disparador en triggers INSERT, UPDATE. I si es modifica el valor de la variable NEW dins del procediment, s'executa la sentència amb el valor modificat.</li> <li>– No fa res en triggers DELETE</li> </ul> </li> <li>■ RETURN OLD               <ul style="list-style-type: none"> <li>– A continuació s'executa la sentència que activa el disparador en triggers DELETE, UPDATE. Si és un trigger UPDATE, no fa el canvi establert en la sentència.</li> <li>– No es fa res en triggers INSERT.</li> </ul> </li> <li>■ RETURN NULL               <ul style="list-style-type: none"> <li>– No executa la sentència que activa el disparador</li> </ul> </li> </ul>	<p>Valor de retorn ignorat. Millor: RETURN NULL</p>
AFTER	<p>Valor de retorn ignorat. Millor: RETURN NULL</p>	<p>Valor de retorn ignorat. Millor: RETURN NULL</p>

## Exemple 1: Auditoria

- Objectiu: Mantenir un registre de les modificacions que fan els usuaris a la taula Items. Cada vegada que es modifiqui la quantitat d'un ítem haurem de guardar un registre a la taula log\_record amb: l'identificador de l'ítem, l'usuari que ha fet la modificació, la data en que s'ha fet, i la quantitat inicial i quantitat final d'ítem.

```
CREATE TABLE log_record(  
    item integer,  
    username char(8),  
    update_time timestamp,  
    old_qtt integer,  
    new_qtt integer);  
  
create table items(  
    item integer primary key,  
    name char(25),  
    qtt integer,  
    preu_total decimal(9,2));  
  
CREATE FUNCTION insert_log() RETURNS trigger AS $$  
BEGIN  
    insert into log_record values (OLD.item,current_user,current_date,OLD.qtt,NEW.qtt);  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER restrict_after_empl  
AFTER UPDATE of qtt ON items  
FOR EACH ROW EXECUTE PROCEDURE insert_log();
```

## Exemple 1: Auditoria - Execució

```
select * from items;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

	item integer	name character(25)	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

Contingut inicial de la taula items

```
update items set qtt=qtt+10 where item<>3;  
select * from log_record;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

	item integer	username character(8)	update_time timestamp without time zone	old_qtt integer	new_qtt integer
1	1	bd	2013-06-20 00:00:00	100	110
2	2	bd	2013-06-20 00:00:00	5000	5010

Contingut de la taula log\_record després d'executar-se les accions del disparador.

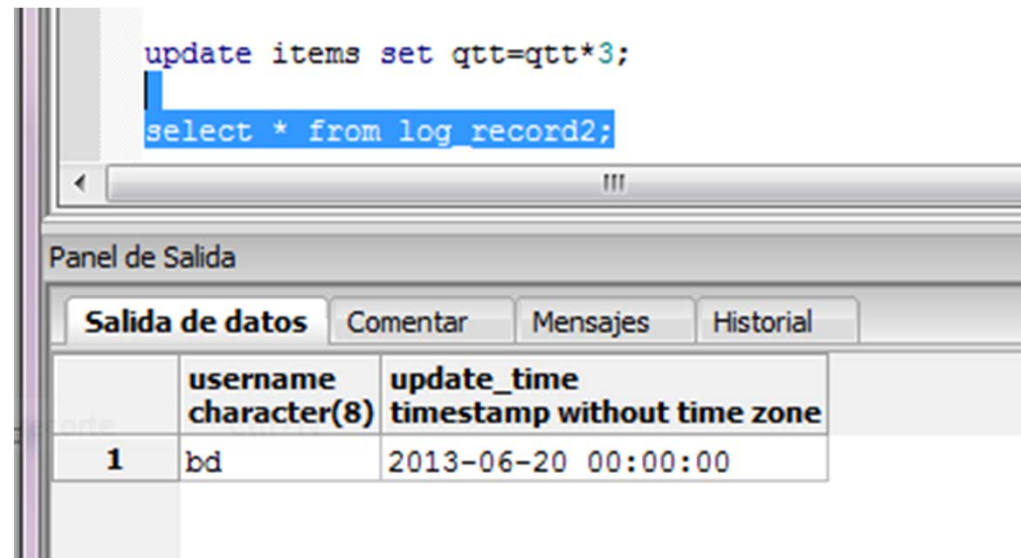
## Exemple 2: Auditoria

Objectiu: Mantenir un registre de les modificacions que fan els usuaris a la taula Items. Ara, només voldrem guardar l'usuari que fa la modificació i la data en que es produeix la modificació.

```
CREATE TABLE log_record2(  
    username char(8),  
    update_time timestamp);  
  
CREATE FUNCTION inserta_log() RETURNS trigger AS $$  
    BEGIN  
        insert into log_record2 values (current_user,current_date);  
        RETURN NULL;  
    END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER audit_items  
AFTER UPDATE ON items  
FOR EACH STATEMENT EXECUTE PROCEDURE inserta_log();
```



## Exemple 2: Auditoria - Execució



Després de modificar la taula items, hi ha una nova fila a la taula log\_record2 indicant que l'usuari "bd" ha modificat la taula items, i la data en què l'ha modificat.

## Exemple 3: Atribut derivat

- Objectiu: Donada la taula items mantenir de manera automàtica l'atribut derivat `preu_total` quan hi ha modificacions de la quantitat d'estoc.

```
CREATE FUNCTION calcular_nou_total() RETURNS trigger AS $$
BEGIN
    IF (old.qtt <> 0) THEN
        NEW.preu_total := ((OLD.preu_total / OLD.qtt) * NEW.qtt);
    END IF;

    RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER atribut_derivat
BEFORE UPDATE OF qtt ON items
FOR EACH ROW EXECUTE PROCEDURE calcular_nou_total();
```

Modificació de la variable `NEW` dins del procediment.

El procediment fa que, a part de modificar-se la quantitat en estoc d'un ítem, es modifiqui al mateix temps el `preu_total` de l'ítem.

### Exemple 3: Atribut derivat - Execució

```
select * from items;
```

Panel de Salida				
Salida de datos Comentar Mensajes Historial				
	item integer	name character(25)	qtt integer	preu_total numeric(9,2)
1	1	sac	100	5.00
2	2	bolis	5000	10.00
3	3	rats	500	15.00

Contingut inicial  
de la taula items

La sentència que activa el trigger és:

UPDATE items SET qtt=qtt+1000 WHERE item=3;

```
select * from items;
```

Panel de Salida				
Salida de datos Comentar Mensajes Historial				
	item integer	name character(25)	qtt integer	preu_total numeric(9,2)
1	1	sac	100	5.00
2	2	bolis	5000	10.00
3	3	rats	1500	45.00

Contingut final de la taula  
items després d'executar la  
sentència UPDATE

## Exemple 4 : Regla de negoci

- Objectiu: Una única sentència de modificació no pot augmentar la quantitat total en estoc dels productes en més d'un 50%.

```
CREATE TABLE TEMP(old_qtt integer);

CREATE FUNCTION update_items_before()RETURNS trigger AS $$
BEGIN
    INSERT INTO temp SELECT sum(qtt) FROM items;
    return null;
END $$ LANGUAGE plpgsql;

CREATE FUNCTION update_items_after()RETURNS trigger AS $$
DECLARE
    oldqtt integer default 0;
    newqtt integer default 0;
BEGIN
    SELECT old_qtt into oldqtt FROM temp;
    DELETE FROM temp;
    SELECT sum(qtt) into newqtt FROM items;
    IF (newqtt>oldqtt*1.5) THEN RAISE EXCEPTION 'Violació regla de negoci';
    END IF;
    RETURN NULL;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER regla_negociBS BEFORE UPDATE ON items
    FOR EACH STATEMENT EXECUTE PROCEDURE update_items_before();

CREATE TRIGGER regla_negociAS AFTER UPDATE ON items
    FOR EACH STATEMENT EXECUTE PROCEDURE update_items_after();
```

## Exemple 4 : Execució

```
select * from items;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

	item integer	name character(25)	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

```
update items set qtt=qtt*3;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

ERROR: Violació regla de negoci

\*\*\*\*\* Error \*\*\*\*\*

ERROR: Violació regla de negoci  
Estado SQL:P0001

Contingut inicial de la taula items

Les accions del disparador eviten que es violi la regla de negoci, però

HI HA UNA MILLOR SOLUCIÓ!!

per evitar haver d'executar dues vegades la sentència SQL  
`SELECT sum(qtt) from ITEMS`

## Exemple 4: Regla de negoci – Solució Incremental

- Objectiu: No pot ser que una única sentència de modificació augmenti la quantitat total en estoc dels productes en més d'un 50%

```
CREATE TABLE TEMP(  
    old_qtt integer,  
    incr integer);  
  
CREATE FUNCTION update_items_before() RETURNS trigger AS $$  
BEGIN  
    DELETE From temp;  
    INSERT INTO temp(old_qtt,incr) select sum(qtt),0 from  
items;  
    return null;  
END $$ LANGUAGE plpgsql;  
  
CREATE FUNCTION update_items_inc() RETURNS trigger AS $$  
DECLARE  
    oldqtt integer default 0;  
    suma_incr integer default 0;  
BEGIN  
    UPDATE temp  
    SET incr=incr+(NEW.qtt-OLD.qtt);  
    SELECT old_qtt,incr INTO oldqtt,suma_incr FROM temp;  
    IF (suma_incr>oldqtt*0.5) THEN  
        RAISE EXCEPTION 'Violacio regla de negoci';  
    END IF;  
    return NEW;  
END $$ LANGUAGE plpgsql;
```

A diferència de l'anterior solució en aquesta no s'accedeix a totes les files de la taula *items*, sinó només a les que s'ha de modificar.

Per aquest motiu es diu que aquesta és una solució incremental.

## Exemple 4: Regla de negoci – Solució Incremental

```
CREATE TRIGGER regla_negociBS
  BEFORE UPDATE ON items
  FOR EACH STATEMENT
  EXECUTE PROCEDURE update_items_before();

CREATE TRIGGER regla_negociBR
  BEFORE UPDATE OF qtt ON items
  FOR EACH ROW
  EXECUTE PROCEDURE update_items_inc();
```

Cal notar que el segon disparador en la solució incremental és **BEFORE, FOR EACH ROW**. Amb aquesta solució s'estalvia l'accés a tots els items per a calcular l'estoc després de l'update, ja que el procediment **update\_items\_inc** utilitza la informació sobre les tuples modificades que hi ha a les variables NEW i OLD abans de cada update.

## Exemple 4: Regla de negoci – Solució Incremental - Execució

```
select * from items;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

	item integer	name character(25)	qtt integer
1	1	sac	100
2	2	bolis	5000
3	3	rats	500

Contingut inicial de la taula items

```
update items set qtt=qtt*3;
```

Panel de Salida

Salida de datos Comentar Mensajes Historial

ERROR: Violació regla de negoci

\*\*\*\*\* Error \*\*\*\*\*

ERROR: Violació regla de negoci  
Estado SQL:P0001

Com és natural, el resultat és el mateix que per la solució no incremental.



## Exemple 5: Auditoria

- Objectiu: Auditar els esborrats i modificacions de la taula items. Cada vegada que s'executi una sentència d'esborrat o modificació de files de la taula, cal registrar a la taula log\_record2 el nom de l'usuari que ha invocat la sentència, l'instant en què s'ha produït, i l'operació concreta que s'ha executat (delete o update).

```
CREATE TABLE log_record2(  
    username char(8),  
    update_time timestamp,  
    operacio varchar(6));  
  
CREATE FUNCTION inserta_log() RETURNS trigger AS $$  
    BEGIN  
        insert into log_record2  
            values (current_user,current_date,TG_OP);  
        RETURN NULL;  
    END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER audit_items  
AFTER UPDATE OR DELETE ON items  
FOR EACH STATEMENT EXECUTE PROCEDURE inserta_log();
```

Veure  
transparència  
*Variables  
accessibles des  
dels procediments*

## Exemple 5: Auditoria - Execució

```
select * from items;
```

Panel de Salida

	item integer	name character(25)	qtt integer	preu_total numeric(9,2)
1	1	sac	100	300.00
2	2	bolis	5000	50000.00
3	3	rats	500	5000.00

Contingut inicial  
de la taula items

```
select * from log_record2
```

Panel de Salida

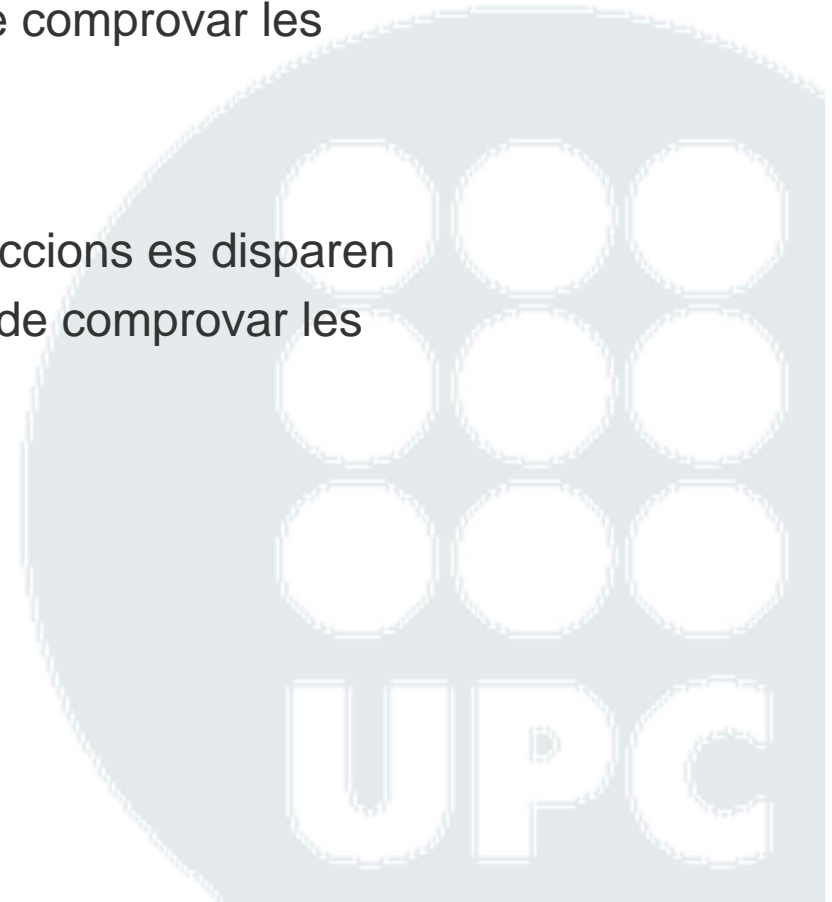
	username character(8)	update_time timestamp without time zone	operacio character varying(6)
1	bd	2013-06-20 00:00:00	UPDATE
2	bd	2013-06-20 00:00:00	DELETE

Files de la taula log\_record2  
després d'executar  
una sentència de modificació  
i una d'esborrat.

UPC

## Precedència entre disparadors i restriccions

- En un disparador BEFORE, les accions es disparen abans d'executar l'operació i de comprovar les restriccions
- En un disparador AFTER, les accions es disparen després d'executar l'operació i de comprovar les restriccions



## Exemple 6 – Manteniment de restriccions

- Objectiu: Tot estudiant ha de ser usuari.

```
CREATE TABLE usuari(  
    id_u integer primary key);  
CREATE TABLE estudiant(  
    id_e integer references usuari);  
  
CREATE FUNCTION inserir() RETURNS trigger AS $$  
BEGIN  
    IF (NOT EXISTS (SELECT * FROM usuari WHERE id_u=NEW.id_e))  
    THEN  
        INSERT INTO usuari VALUES (NEW.id_e);  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER restrict1 BEFORE INSERT ON estudiant  
FOR EACH ROW EXECUTE PROCEDURE inserir();
```

**Precedència de restriccions:** Tenint en compte la precedència entre disparadors i restriccions, penseu què passaria si el disparador de l'exemple s'hagués definit AFTER, en lloc de BEFORE.

## Exemple 6 – Manteniment de restriccions - Execució

Files de la taula usuari  
abans de la inserció  
de l'estudiant "1"

```
select * from usuari;
```

Panel de Salida

Salida de datos	Comentar	Mensajes	H
	id_u		
	integer		

Files de la taula usuari  
després de la inserció  
de l'estudiant "1"

```
insert into estudiant values (1);  
select * from usuari;
```

Panel de Salida

Salida de datos	Comentar	Mensajes	Historial
	id_u		
	integer		
1	1		

## Exemple 7 – Manteniment de restriccions

- Objectiu: Tot estudiant que acaba els estudis ja no pot ser becari. Quan passi aquesta situació caldrà esborrar l'estudiant de la taula de becaris.


```
CREATE TABLE estudiants (  
    dni char(9) primary key,  
    data_fi_estudis date default null);  
  
CREATE TABLE becaris (  
    dni char(9) primary key references estudiants);  
  
CREATE FUNCTION proc_manteniment() RETURNS trigger AS $$  
BEGIN  
    IF (OLD.data_fi_estudis is null and  
        NEW.data_fi_estudis is not null) THEN  
        DELETE FROM becaris where dni=NEW.dni;  
    END IF;  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER disp_manteniment AFTER UPDATE ON estudiants  
FOR EACH ROW EXECUTE PROCEDURE proc_manteniment();
```

## Exemple 7 – Manteniment de restriccions - Execució

- Sentència que dispara el trigger:

```
update estudiants  
set data_fi_estudis= current_date  
where dni='1';
```

Contingut inicial de  
la taula becaris  
abans de l'execució  
del trigger



	dni character(9)
1	1

Contingut final de la  
taula de becaris  
després de l'execució  
del trigger



dni character(9)
---------------------

## INSTEAD OF: un altre tipus de trigger per actualitzar vistes

```
create table emp(nemp integer primary key, sou integer);  
create view emp32 as select * from emp where nemp>32;
```

```
CREATE or replace FUNCTION insert32() RETURNS trigger AS $$  
BEGIN  
insert into emp values (new.nemp+1,new.sou+1);  
RETURN new;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER view_insert  
  INSTEAD OF INSERT ON emp32  
  FOR EACH ROW  
  EXECUTE PROCEDURE insert32();
```

