3. Disparadors

Els components lògics d'una BD vistos fins ara són insuficients per a implementar adequadament algunes de les situacions que es produeixen en el món real.

Imaginem, per exemple, que tenim una taula d'estocs de productes d'una organització determinada que té una regla de negoci definida, segons la qual, quan l'estoc d'un producte queda per sota de cinquanta unitats, cal fer-ne una comanda de cent unitats. Amb els components vistos fins ara, es presenten dues solucions convencionals per a implementar aquesta situació:

- Afegir aquesta regla a tots els programes que actualitzen l'estoc dels productes.
- Fer un programa addicional que faci un sondeig (en anglès, *polling*) periòdic de la taula per comprovar la regla.

La primera solució té diversos inconvenients: la regla de negoci està amagada, distribuïda i replicada dins del codi dels programes i, per tant, és difícil de localitzar i canviar, i, a més, la seva eficàcia o correcció depèn del fet que cada programador insereixi la regla correctament.

La segona solució, tot i que la regla està concentrada en un sol lloc (el programa de sondeig), presenta els inconvenients clàssics del sondeig; si es fa ocasionalment, es pot perdre el bon moment per a reaccionar i si, en canvi, es fa molt sovint, es perd eficiència.

La bona solució és dotar el sistema d'activitat. S'incorpora a la BD un nou component, els disparadors[2], que modelen la regla de negoci i s'executen d'una manera automàtica. Per al problema plantejat (i sense entrar encara en la sintaxi concreta), el disparador que s'incorporaria a la BD és el següent: "Quan es modifiqui l'estoc d'un producte i esdevingui inferior a cinquanta, cal demanar-ne cent unitats noves."

[(2)] En anglès, *triggers*.

Com es pot veure, amb aquesta solució se superen els inconvenients de les dues solucions vistes anteriorment: la regla de negoci és només en un lloc, l'eficàcia no depèn del programador i s'executarà sempre que calgui i només quan calgui.

Resumint, els disparadors són uns components que s'executen d'una manera automàtica quan es produeix un esdeveniment determinat. Es diu que un disparador és una regla **ECA** (esdeveniment, condició, acció) quan es produeix un esdeveniment determinat, si es compleix una condició, aleshores s'executa una acció.

En l'exemple anterior, l'esdeveniment que activa el disparador és la modificació de l'estoc d'algun producte; la condició, que l'estoc sigui inferior a cinquanta, i l'acció que s'executa, el fet de demanar cent unitats noves.

3.1. Quan s'han de fer servir disparadors

Hi ha tota una sèrie de situacions en què és possible usar disparadors, i convé fer-ho, entre les quals esmentem les següents:

Actualitzacions

Hem d'entendre *actualitzacions* en el sentit ampli: insercions, esborrats i modificacions.

- Implementació d'una regla de negoci. Com a exemple d'això podem revisar el cas dels estocs mencionat anteriorment.
- Manteniment automàtic d'una taula d'auditoria de l'activitat en la BD. Es tracta d'enregistrar d'una manera automàtica els canvis que es fan en una taula determinada.
- Manteniment automàtic de columnes derivades. El valor d'una columna derivada es calcula a partir del valor d'altres columnes; possiblement, d'altres taules: quan es modifiquen les columnes base, cal recalcular d'una manera automàtica el valor de la columna derivada.
- Comprovació de restriccions d'integritat no expressables en el sistema mitjançant `CHECK` o mitjançant restriccions d'integritat referencial. Com a exemple d'això podem considerar les restriccions dinàmiques. La més clàssica, i, d'altra banda normalment benvinguda, és "un sou no pot baixar". Aquesta restricció fa referència a l'estat anterior i posterior a una modificació i, per tant, no es pot expressar com a `CHECK`. En són un altre exemple les assercions, que són restriccions expressables en SQL estàndard, però que cap SGBD no implementa.
- Reparació automàtica de restriccions d'integritat. Cal distingir entre *comprovació* i *reparació* de restriccions d'integritat. La semàntica de la comprovació d'una restricció d'integritat és pot resumir així: "si una actualització infringeix la restricció, cal avortar (desfer) l'actualització". La semàntica de la reparació és: "si la restricció és infringida, cal dur a terme accions compensatòries (noves actualitzacions) perquè no s'infringeixi". Els SGBD normalment implementen les comprovacions.

3.2. Quan no s'han de fer servir disparadors

Si bé els disparadors, com hem vist, poden ser útils a l'hora d'implementar determinades situacions del món real, no cal abusar-ne: no s'haurien d'utilitzar disparadors en les situacions en què el sistema es pot resoldre amb els seus

propis mecanismes. Així, per exemple, no s'haurien de fer servir per a implementar les restriccions de clau primària, ni les restriccions d'integritat referencial, ni totes les expressables com a `CHECK`, etc.

3.3. Sintaxi dels disparadors en PostgreSQL

Actualment, la majoria dels SGBD disposen de disparadors. En trobem tant en els sistemes comercials (com ara l'Oracle, el DB2, l'SQLServer o l'Informix) com en els de lliure distribució (PostgreSQL o MySQL). De fet, l'SQL estàndard, des de l'SQL:1999 defineix els disparadors com a components essencials de les BD.

No obstant això, els llenguatges que usen els diversos sistemes per a definir- los i les capacitats d'expressivitat són lleugerament diferents no solament entre els SGBD sinó també entre aquests i l'SQL:1999. En aquest mòdul hem optat per explicar la sintaxi dels disparadors en PostgreSQL, per tal de poder executar els exemples que veurem i resoldre els exercicis en un sistema real.

A continuació descrivim la sintaxi dels disparadors en PostgreSQL, i en els subapartats següents presentarem exemples de com funciona.

```
CREATE TRIGGER <nom_disparador> {BEFORE | AFTER}
{<esdeveniment> [OR <esdeveniment>]} ON <taula>
 [FOR [EACH] {ROW | STATEMENT}]
 EXECUTE PROCEDURE <nom_procediment();>
```

Com es pot veure, un disparador té un nom, un esdeveniment (com a mínim) i una taula associats:

- El nom serveix per a identificar-lo en cas que es vulgui modificar[3] o esborrar.

[(3)] Per a modificar un disparador, cal esborrar-lo i tornar-lo a crear.

- L'esdeveniment especifica el tipus de sentència que activa el disparador. En particular, en PostgreSQL són:

```
INSERT | DELETE | UPDATE [OF columna[, columna…]]
```

Per a cada disparador, cal definir com a mínim un procediment emmagatzemat que conté les accions que es duran a terme quan s'activi. Les clàusules `BEFORE`, `AFTER`, `FOR EACH ROW` i `FOR EACH STATEMENT` serveixen per a especificar quan s'executarà el procediment associat al disparador.

Activació d'un disparador a la versió 9.0 de PostgreSQL

A partir de la versió 9.0 de PostgreSQL, es pot activar un disparador quan es modifica el valor d'alguna columna d'una taula concreta. Les versions anteriors a la 9.0 no suporten aquesta característica. És a dir, l'esdeveniment d'`UPDATE` no accepta que s'especifiqui la columna o columnes que es modifiquen en la sentència `CREATE TRIGGER`.

El procediment emmagatzemat que conté les accions que ha d'executar el disparador és especial de PostgreSQL i s'anomena *trigger procedure*. Es caracteritza perquè no rep paràmetres i retorna un tipus de dades especial anomenat *trigger*. Aquest procediment s'ha de definir abans que el disparador que l'invoca. Un mateix procediment pot ser invocat per diversos disparadors.

En PostgreSQL, els disparadors poden ser `BEFORE` o `AFTER`:

- Si el disparador és `BEFORE`, el procediment associat s'executa abans que la sentència que dispara el disparador.
- Si el disparador és `AFTER`, el procediment associat s'executa després que la sentència que dispara el disparador.

A més, els disparadors poden ser `FOR EACH ROW` o `FOR EACH STATEMENT`:

- Si el disparador és `FOR EACH ROW`, el procediment associat s'executa una vegada per a cada fila afectada per la sentència que dispara el disparador.
- Si el disparador és `FOR EACH STATEMENT`, el procediment s'executa una vegada, independentment del nombre de files afectades per la sentència que dispara el disparador.

Així, per exemple, una operació de `DELETE` que afecta deu files de la taula `T` farà que el procediment associat a qualsevol disparador `FOR EACH ROW` definit sobre la taula `T` s'executi deu vegades, una vegada per cada fila esborrada. En canvi, si el disparador s'ha definit `FOR EACH STATEMENT`, el procediment en qüestió només s'executarà una sola vegada (amb independència del nombre de files esborrades).

Quan es defineix un disparador, cal especificar si el disparador s'executarà `BEFORE` o `AFTER` i si serà `FOR EACH ROW` o `FOR EACH STATEMENT`. Per tant, podem tenir quatre tipus de disparadors:

- Disparador `BEFORE/FOR EACH STATEMENT`. El procediment associat al disparador s'executa **una sola vegada abans** de l'execució de la sentència que dispara el disparador.

- Disparador `BEFORE/FOR EACH ROW`. El procediment associat al disparador s'executa **una vegada per a cada fila afectada** i just abans que la fila s'insereixi, es modifiqui o s'esborri.

- Disparador `AFTER/FOR EACH STATEMENT`. El procediment associat al disparador s'executa **una sola vegada després** de l'execució de la sentència que dispara el disparador.

Nota

Les "files afectades" en els disparadors `FOR EACH ROW` són les files inserides, esborrades o modificades per la sentència SQL que dispara el disparador.

- Disparador `AFTER/FOR EACH ROW`. El procediment associat al disparador s'executa **una vegada per a cada fila afectada i després** de l'execució de la sentència que dispara el disparador.

Per acabar, cal dir que un disparador s'esborra amb la sentència següent.

```
DROP TRIGGER <nom_disparador> ON <nom_taula>;
```

3.3.1. Procediments invocats per disparadors

Es tracta de procediments emmagatzemats especials que PostgreSQL anomena *trigger procedures*. Com hem dit abans, aquests procediments no poden rebre paràmetres de la manera habitual que hem explicat abans, i han de retornar un tipus de dades especial anomenat *trigger*.

La manera de passar paràmetres a aquests procediments emmagatzemats és per mitjà de variables especials de PostgreSQL que es creen i instancien automàticament quan s'executa el procediment. Algunes d'aquestes variables són les següents:

- `TG_OP`. És una cadena de text que conté el nom de l'operació que ha disparat el disparador. Pot tenir els valors següents: `INSERT`, `UPDATE` o `DELETE` (en majúscules).

- `NEW`. Per a disparadors del tipus `FOR EACH STATEMENT`, té el valor `NULL`. Per a disparadors del tipus `FOR EACH ROW` és una variable de tipus compost que conté la fila després de l'execució d'una sentència de modificació (`UPDATE`) o la fila que cal inserir (`INSERT`).

- `OLD`. Per a disparadors del tipus `FOR EACH STATEMENT`, té el valor `NULL`. Per a disparadors del tipus `FOR EACH ROW` és una variable de tipus compost que conté la fila abans de l'execució d'una sentència de modificació (`UPDATE`) o la fila que cal esborrar (`DELETE`).

Exemple d'instanciació de les variables `NEW` i `OLD`

Suposem que tenim definida la taula `t` següent. Sobre aquesta taula s'ha definit el disparador anomenat `trig`. Considerem també que s'executa la sentència de modificació que tenim més avall i que aquesta sentència dispara el disparador `trig`:

```
CREATE TABLE t(
  a integer PRIMARY KEY,
  b integer);

CREATE TRIGGER trig BEFORE UPDATE ON t
FOR EACH ROW
EXECUTE PROCEDURE prog();

UPDATE t
SET b=3
WHERE a=1;
```

Suposem que el contingut inicial de la taula `t` és el següent:

a	b
1	2

Durant l'execució del disparador, les variables `NEW` i `OLD` prenen els valors següents:

Valors abans que s'executi la sentència `UPDATE`:

```
OLD.a=1 i OLD.b=2
```

Valors després que s'executi la sentència `UPDATE`:

```
NEW.a=1 i NEW.b=3
```

Pel que fa al retorn de resultats, els procediments invocats per disparadors poden retornar `NULL` o bé una variable de tipus compost, que tingui la mateixa estructura que les files de la taula sobre la qual està definit el disparador. És el cas de les variables `OLD` i `NEW`.

Nota

Per a accedir als valors de cada fila modificada per la sentència `UPDATE` utilitzarem la notació següent:

`OLD.nom_columna_taula` o `NEW.nom_columna_taula`

Per tant, els procediments invocats per disparadors poden retornar els valors següents:

1) Disparadors `BEFORE` / `FOR EACH ROW`:

- `NULL`, per a indicar al disparador que no ha d'acabar l'execució de l'operació per a la fila actual. En aquest cas, la sentència que dispara el disparador (`INSERT/UPDATE/DELETE`) no s'arriba a executar.

- `NEW`, per a indicar que l'execució del procediment per a la fila actual ha d'acabar normalment i que la sentència 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 variant directament la fila que s'inserirà o modificarà.

- OLD, per a indicar que l'execució del procediment per la fila actual ha d'acabar normalment i que la sentència que dispara el disparador (DELETE / UPDATE) s'ha d'executar. En el cas de l'UPDATE, si el procediment retorna OLD, no es fa la modificació de la fila actual.

2) Altres tipus de disparadors: Als disparadors AFTER/FOR EACH ROW i als disparadors FOR EACH STATEMENT (independentment que siguin BEFORE o AFTER), el valor retornat pel procediment que és invocat pel disparador és ignorat. Per aquest motiu, aquests procediments normalment retornen NULL.

Vegeu també

El disparador del subapartat 3.3.2 (exemple de manteniment automàtic d'un atribut derivat) il·lustra l'ús de la variable NEW per a modificar les dades de les files afectades pel disparador.

La motivació principal perquè els disparadors BEFORE i FOR EACH ROW retornin un valor diferent de nul, és que poden modificar les dades de les files que s'inseriran o modificaran en la taula associada al disparador. Vegem-ne un exemple.

Exemple de retorn de resultats en un `trigger-procedure`

Suposem que tenim definida la taula t següent. Sobre aquesta taula s'ha definit el disparador anomenat trig. Considerem també que s'executa la sentència d'inserció següent que dispara el disparador:

```
CREATE TABLE t(
  a integer PRIMARY KEY,
  b integer);

CREATE FUNCTION prog()
...
...
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trig BEFORE INSERT ON t
FOR EACH ROW
EXECUTE PROCEDURE prog();

INSERT INTO t VALUES (1,2);
```

El procediment prog() pot retornar dos valors:

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

3.3.2. Exemples de disparadors

A continuació presentem un seguit d'exemples que, a més d'il·lustrar la sintaxi de PostgreSQL, serviran per a mostrar alguns dels usos dels disparadors.

Manteniment automàtic d'una taula d'auditoria

En presentar aquest disparador exemplificarem, alhora, l'ús de la clàusula `FOR EACH ROW` i de la clàusula `AFTER`. També veurem l'ús de les variables especials `NEW` i `OLD`.

L'objectiu del disparador és mantenir d'una manera automàtica una taula d'auditoria (`log_record`) que contingui un registre de totes les modificacions de la columna `qtt` que fan els usuaris a la taula `Items` d'una certa BD. Així, imaginem que disposem de les taules següents.

```
CREATE TABLE log_record(
  item integer,
  username char(8),
  hora_modif timestamp,
  qtt_vella integer,
  qtt_nova integer);

CREATE TABLE items(
  item integer primary key,
  nom char(25),
  qtt integer,
  preu_total decimal(9,2));
```

Atès que el disparador ha de guardar una fila per cada modificació feta a la columna `qtt` de la taula `Items`, en aquest cas el més adequat és utilitzar un disparador del tipus `AFTER` i `FOR EACH ROW`. El disparador es defineix `AFTER` per a inserir tuples a la taula d'auditoria, només en el cas que s'hagin produït modificacions de la quantitat d'algun `item`. Si definíssim el disparador `BEFORE`, les insercions a la taula d'auditoria es farien abans que es produís la modificació de la quantitat d'algun `item`. Si, per algun motiu, la modificació de la quantitat d'algun `item` fallés, aleshores ja hauríem fet la inserció a la taula d'auditoria, realitzant més feina de la necessària.

D'altra banda, el disparador és defineix del tipus `FOR EACH ROW`, perquè ens interessa guardar un registre d'auditoria per a cada ítem del qual es modifica la quantitat.

A continuació, tenim el codi del disparador i del procediment emmagatzemat invocat per aquest disparador.

```
CREATE FUNCTION inserta_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 auditoria_items
AFTER UPDATE OF qtt ON items
FOR EACH ROW EXECUTE PROCEDURE inserta_log();
```

Cada vegada que s'actualitzi la columna `qtt` de la taula `items` s'inserirà una fila a la taula `log_record`. La variable `OLD.qtt` i `NEW.qtt` contindran, respectivament, els valors antics i nou d'aquesta columna.

Current_user i Current_date

Les funcions predefinides de PostgreSQL, `current_user` i `current_date` retornen, respectivament, el nom de l'usuari que ha executat la sentència que dispara el disparador i l'instant d'execució.

Considerem el contingut següent de la taula `items`:

ítem	nom	qtt	preu_total
1	sac	100	0,30
2	boli	5000	0,50
3	rat	500	0,60

Si executem la sentència `UPDATE Items SET qtt=qtt+10 WHERE item<>3` sobre la taula anterior, per a cada fila actualitzada (en aquest cas, les files 1 i 2) s'executarà la inserció corresponent, i en la taula `log_record` s'obtindrà el resultat següent:

ítem	username	hora_modif	qtt_vella	qtt_nova
1	joan	2010-04-15 15:00	100	110
2	joan	2010-04-15 15:00	5000	5010

Llegim la taula: l'usuari `joan` va actualitzar dues files de la taula `items`; va augmentar la quantitat de cadascuna en deu unitats el dia 15 d'abril de 2010 a les 15.00 hores.

Activitat

Com a activitat complementària, podeu tornar a implementar aquest exemple d'auditoria, considerant que ara només cal emmagatzemar l'`username` de l'usuari que fa la modificació i la data i l'hora en què es produeix aquesta modificació, a la taula d'auditoria. No cal emmagatzemar el codi de l'ítem modificat, ni la quantitat abans i després de la modificació. Canviaria el tipus de disparador (`BEFORE/AFTER`, `FOR EACH ROW/ FOR EACH STATEMENT`)?

Manteniment automàtic d'un atribut derivat

En aquest exemple es mostra com s'utilitzen els disparadors per a mantenir calculat d'una manera automàtica l'atribut derivat `preu_total` de la taula `ítems`, quan es produeixen modificacions de la quantitat d'estoc d'algun ítem.

El valor de l'atribut derivat es calcula a partir del valor d'altres columnes de la mateixa taula, o bé, d'altres taules; per tant, quan es modifiquen les columnes que es fan servir per a calcular l'atribut derivat, cal recalcular de forma automàtica el valor de l'atribut derivat.

Partim novament de la taula `items` següent.

```
CREATE TABLE items(
  item integer primary key,
  nom char(25),
  qtt integer,
  preu_total decimal(9,2));
```

Definim l'atribut `preu_total`, un atribut derivat, el valor del qual es calcularà de la manera següent.

```
CREATE FUNCTION calcula_nou_preu_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 calcula_nou_preu_total();
```

El procediment `calcula_nou_preu_total` retorna el nou preu total d'un ítem, calculat a partir del preu total anterior que teníem en la BD i la quantitat de l'ítem abans i després de la modificació.

Fixeu-vos, en primer lloc, que el procediment només recalcula el preu total d'un ítem quan es produeix una modificació de la quantitat en estoc d'un ítem. En segon lloc, el recàlcul del nou preu total s'assigna a la variable `NEW.preu_total`. Amb aquesta assignació modifiquem directament el contingut de la variable `NEW` abans que es produeixi la modificació de la quantitat en estoc d'un ítem. Finalment, el procediment retorna la variable `NEW`. Aquest retorn serveix perquè la modificació del preu total que realitza el procediment s'acabi efectuant a la taula `items`.

Recordeu

És més eficient que els procediments invocats per disparadors només duguin a terme les accions quan cal. En l'exemple d'aquest subapartat només es torna a calcular el preu total quan es modifica la quantitat d'algun ítem.

El procediment `calcula_nou_preu_total` ha de retornar la variable `NEW` perquè la modificació del preu total es guardi a la taula `items`.

Implementació d'una regla de negoci

Amb aquest exemple il·lustrarem l'ús de més d'un disparador per a implementar una regla de negoci. A més, veurem la manera d'avortar (en anglès, *abort*) l'execució de la sentència que ha activat el disparador i de tota la transacció en curs.

Partim de la taula `items` de l'exemple del subapartat anterior.

```
CREATE TABLE items(
  item integer primary key,
  nom char(25),
  qtt integer,
  preu_total decimal(9,2));
```

L'objectiu del nou disparador és implementar la regla de negoci següent: "No pot ser que una sola sentència de modificació (UPDATE) augmenti la quantitat total dels productes en estoc més d'un 50%."

Atès que aquesta regla de negoci no requereix, *a priori*, un tractament per a cadascuna de les files, sembla natural implementar-la amb disparadors FOR EACH STATEMENT. Un primer disparador executat BEFORE pot sumar els ítems en estoc abans de l'actualització i un segon disparador AFTER pot sumar-los després i fer la comparació de les dues quantitats:

```
CREATE TRIGGER regla_negoci_abans BEFORE UPDATE ON items
FOR EACH STATEMENT
  EXECUTE PROCEDURE update_items_abans();

CREATE TRIGGER regla_negoci_despres AFTER UPDATE ON items
FOR EACH STATEMENT
  EXECUTE PROCEDURE update_items_despres();
```

Aquesta solució té un problema. En PostgreSQL, els procediments invocats per a disparadors no poden retornar qualsevol valor; per tant, no tenim manera de retornar la quantitat d'ítems en estoc abans que es produeixi la modificació. Per poder-ho fer, definirem la taula temporal següent:

```
CREATE TABLE TEMP(qtt_vella integer);
```

El procediment update_items_abans guardarà la quantitat en estoc en aquesta taula temporal i el procediment update_items_despres la consultarà i netejarà per a properes execucions.

```
CREATE FUNCTION update_items_abans()RETURNS
trigger AS $$
BEGIN
  INSERT INTO temp SELECT sum(qtt) FROM items;
  RETURN NULL;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION update_items_despres()RETURNS
trigger AS $$
DECLARE
  qtt_abans integer default 0;
  qtt_despres integer default 0;
BEGIN
  SELECT qtt_vella into qtt_abans FROM temp;
  DELETE FROM temp;
  SELECT sum(qtt) into qtt_despres FROM items;
  IF (qtt_despres>qtt_abans*1.5) THEN
    RAISE EXCEPTION 'Infraccio regla de negoci';
  END IF;
  RETURN NULL;
END
$$ LANGUAGE plpgsql;
```

Recordeu

Per a poder provar l'exemple, cal crear la taula temporal i els procediments emmagatzemats abans que els disparadors.

Fixeu-vos que, si la sentència de modificació no compleix la regla de negoci, el procediment emmagatzemat executat `AFTER` genera una excepció, la qual desfà la sentència que dispara el disparador i tota la transacció en curs. En canvi, si l'execució d'una sentència de modificació no provoca la violació de la regla de negoci, la sentència `UPDATE` s'executa correctament.

Un comentari final sobre aquest disparador. Com hem vist, el disparador calcula dues vegades l'estoc dels productes: una vegada abans de l'actualització i una altra, després. Pot ser interessant plantejar-se la possibilitat de definir un disparador alternatiu que faci la mateixa funció d'una manera més eficient, obviant una de les dues sumes totals. La clau està a tenir en compte les files que s'actualitzen: no caldrà la segona suma si es coneix el resultat de la primera i les actualitzacions que s'han executat. Només caldrà fer una vegada la suma, acumular les actualitzacions que es facin i deduir l'altra suma. És a dir, tant podem fer un disparador amb accions `FOR EACH ROW` (que acumuli) i `AFTER` (que calculi la suma) com el simètric amb accions `BEFORE` (que calculi la suma) i `FOR EACH ROW` (que acumuli).

Aquesta tècnica per a augmentar l'eficiència dels disparadors que té en compte el que canvia i no fa comprovacions redundants es coneix amb el nom de *disparadors incrementals*. Sempre que es pugui, convé utilitzar aquesta tècnica.

Activitat

Com a activitat complementària, podeu implementar aquesta regla de negoci amb aquest enfocament.

3.3.3. Altres aspectes sobre els disparadors

Altres aspectes que cal considerar sobre els disparadors son el seu ordre d'execució, els errors, les restriccions d'integritat o els disparadors en cascada.

Ordre d'execució dels disparadors

Com hem vist anteriorment, és possible que per a implementar una semàntica o situació calgui més d'un disparador. En aquests casos, és possible que s'hagin d'implementar diversos disparadors per al mateix esdeveniment (`INSERT`, `UPDATE` o `DELETE` sobre la mateixa taula). Quan això passa, PostgreSQL fixa l'ordre d'execució dels disparadors de la manera següent:

- Els disparadors `BEFORE/FOR EACH STATEMENT` s'executen abans que els disparadors `BEFORE/FOR EACH ROW`.

- Els disparadors `AFTER/FOR EACH STATEMENT` s'executen després que els disparadors `AFTER/FOR EACH ROW`.

Només en el cas que tinguem dos disparadors per al mateix esdeveniment i que siguin del mateix tipus (per exemple, dos disparadors `BEFORE/FOR EACH ROW` sobre la mateixa taula), s'utilitzarà l'ordre alfabètic del nom del disparador per a determinar quin disparador s'executa abans. El disparador el nom del qual comenci abans segons l'ordre alfabètic és el que s'executarà abans.

Disparadors i errors

Quan un disparador falla a causa de l'execució d'una de les seves sentències SQL, el sistema retorna l'error concret SQL que s'ha produït. A més, en el subapartat anterior hem vist un exemple de com des d'un disparador es pot cridar un procediment emmagatzemat que provoqui el llançament d'una excepció (en l'exemple, es llançava l'excepció quan no es complia la regla de negoci especificada). Aquesta situació ens serveix per a simular un error SQL.

Recapitulant, tenim dues situacions que fan que el disparador falli:

- S'executa una sentència interna que provoca l'error.

- Es provoca l'error mitjançant el llançament de l'excepció.

Tant en un cas com en l'altre, totes les accions que ha pogut fer el disparador (i les que hagin pogut fer els disparadors activats per les accions del primer) i la transacció en curs es desfan d'una manera automàtica.

Exemple

Considerem l'esquema d'activació de disparadors següent. En aquest cas, es desfaran totes les accions dels disparadors `D1`, `D2` i `D3` i les sentències 1, 2 i 3.

```
BEGIN WORK;
Sentencia 1;
Sentencia 2;
Sentencia 3;
Provoca l'activacio del disparador D1;
D1:
Sentencia 4;
Provoca l'activacio del disparador D2;
D2:
Sentencia 5;
Provoca l'activacio del disparador D3;
D3:
Sentencia 6;
falla
```

Disparadors i restriccions d'integritat

Quan s'executa una sentència SQL contra una BD, pot passar que s'activin disparadors i que es violin restriccions d'integritat. Davant d'aquesta circumstància, el sistema ha de decidir què ha de fer en primer lloc: activar els disparadors o bé comprovar les restriccions d'integritat.

En PostgreSQL, les accions dels disparadors `BEFORE` s'executen abans de comprovar les restriccions d'integritat de la BD. En canvi, les accions dels disparadors `AFTER` s'executen després de comprovar les restriccions d'integritat de la BD.

Exemple

Considerem les taules següents, que expressen la restricció d'integritat referencial "tot fill ha de tenir un pare".

```
CREATE TABLE pare(
  a INTEGER PRIMARY KEY);
CREATE TABLE fill(
  b INTEGER PRIMARY KEY,
  c INTEGER REFERENCES pare);
```

Podem definir un disparador que s'activi quan s'insereixi un fill que no tingui especificat un pare, i que, com a reparació, l'insereixi a la taula `Pare`:

```
CREATE FUNCTION inserir() RETURNS trigger AS $$
BEGIN
  if ((SELECT count(*) FROM pare WHERE a=NEW.b)=0) THEN
    INSERT INTO pare VALUES (NEW.b);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER restriccions1 BEFORE INSERT ON fill
FOR EACH ROW EXECUTE PROCEDURE inserir();
```

La sentència següent provoca la inserció del valor 1 a la taula `Pare`, atès que el disparador s'executa abans de comprovar la restricció d'integritat referencial.

```
INSERT INTO fill VALUES (1,1);
```

En canvi, si el disparador `restriccions1` hagués estat definit `AFTER`, la sentència d'inserció anterior hauria produït un error de violació de la restricció d'integritat referencial. Això és així perquè la restricció d'integritat referencial es comprova abans d'executar el disparador.

Disparadors en cascada

Normalment, els SGBD permeten encadenar l'execució dels disparadors en cascada; és a dir, un disparador pot executar una sentència que, al seu torn, activi un disparador, el qual, de retruc, executi una sentència, etc. PostgreSQL no posa límit al nombre de disparadors que es poden executar en cascada. Aquest comportament pot donar lloc a un bucle infinit.

Exemple

```
CREATE FUNCTION p1()
BEGIN
  DELETE FROM B...
END;
CREATE TRIGGER del_a
AFTER DELETE ON A
FOR EACH ROW (execute procedure p1());

CREATE FUNCTION p2()
BEGIN
  DELETE FROM C...
END;
CREATE TRIGGER del_b
AFTER DELETE ON B
FOR EACH ROW (execute procedure p2());

CREATE FUNCTION p3()
BEGIN
  DELETE FROM A...
END;
CREATE TRIGGER del_c
AFTER DELETE ON C
FOR EACH ROW (execute procedure p3());
```

La sentència següent provoca l'activació dels disparadors anteriors, i es produeix un bucle infinit d'esborraments successius.

```
DELETE FROM A...
```

Aquestes seqüències d'activació de disparadors en cascada són perilloses perquè poden produir bucles infinits. Segons PostgreSQL, és responsabilitat del programador evitar que es produeixin bucles infinits.

Per tant, és important que els programadors utilitzin disparadors només en els casos necessaris i que documentin adequadament les situacions que implementen mitjançant disparadors.

3.3.4. Consideracions de disseny

Sovint hi ha diverses solucions vàlides per a resoldre un mateix problema. En general, cal emprar la solució que eviti fer feina i accessos innecessaris a la BD.

Segons el manual de PostgreSQL, quan s'utilitzen disparadors per a implementar alguna situació i els disparadors poden ser `BEFORE` o `AFTER`, normalment es tria el disparador `BEFORE` per motius d'eficiència. El disparador `BEFORE` executa les accions abans que l'operació que dispara el disparador.

No obstant això, hi ha moltes subtileses a l'hora de decidir quin tipus de disparador és millor per a implementar una situació.

Exemple

Considerem que volem implementar amb disparador la restricció següent: "el sou d'un empleat no pot baixar". Disposem del fragment de la taula d'`empleats` següent:

```
CREATE TABLE empleats (
  num_empl INTEGER PRIMARY KEY,
  sou numeric NOT NULL (CHECK sou<50.000.0),
...);
```

En principi, segons el que hem dit abans, comprovaríem la restricció al més aviat possible. Per tant, utilitzaríem un disparador `BEFORE/ FOR EACH ROW`.

Però, què passaria si la sentència que dispara el disparador viola la restricció `CHECK sou<50.000.0`? Per exemple, la sentència `UPDATE empleats SET sou=60.000.0 WHERE num_empl=10;`

Com que hem definit el disparador de tipus `BEFORE`, es comprovaria en primer lloc la restricció que el sou no pot baixar i, després, la restricció que el sou ha de ser inferior a 50.000. Però, per motius d'eficiència, potser hi hauria casos en què seria millor comprovar primer la restricció que el sou ha de ser inferior a 50.000. Per exemple, si aquesta restricció es viola molt sovint en la BD, aleshores seria més eficient comprovar en primer lloc la restricció `check`. En aquest cas, podríem decidir definir el nostre disparador del tipus `AFTER/FOR EACH ROW`.

Per tant, cal anar amb compte a l'hora d'escollir el tipus de disparador per a implementar una situació o semàntica.