El llenguatge SQL II

Maria José Casany Guerrero
M. Elena Rodríguez González
Toni Urpí Tubella

PID_00171646

www.uoc.edu

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars del copyright.

Índex

Introducció 5

Objectius 6

1. Entorn SQL 7
- 1.1. Esquema, catàleg i servidor 8
- 1.2. Connexió, sessió i transacció 14

2. Procediments emmagatzemats 17
- 2.1. Sintaxi dels procediments emmagatzemats a PostgreSQL 18
  - 2.1.1. Paràmetres 21
  - 2.1.2. Variables 21
  - 2.1.3. Sentències condicionals 23
  - 2.1.4. Sentències iteratives 25
  - 2.1.5. Retorn de resultats 27
  - 2.1.6. Invocació de procediments emmagatzemats 30
  - 2.1.7. Gestió d'errors 31

3. Disparadors 37
- 3.1. Quan s'han de fer servir disparadors 38
- 3.2. Quan no s'han de fer servir disparadors 38
- 3.3. Sintaxi dels disparadors en PostgreSQL 39
  - 3.3.1. Procediments invocats per disparadors 41
  - 3.3.2. Exemples de disparadors 43
  - 3.3.3. Altres aspectes sobre els disparadors 48
  - 3.3.4. Consideracions de disseny 52

Resum 53

Activitats 55

Exercicis d'autoavaluació 55

Solucionari 57

Glossari 59

Bibliografia 60

Introducció

En aquest mòdul didàctic ampliarem els coneixements que tenim de l'SQL estàndard; més concretament, estudiarem els procediments emmagatzemats i els disparadors (en anglès, *triggers*). Aquests dos components lògics, juntament amb altres components també lògics, com les taules i les vistes, estan organitzats en el que anomenem *esquema de base de dades* (BD). L'esquema de BD no és l'únic component que permet organitzar-ne d'altres; tenim el catàleg, que conté un conjunt d'esquemes, i el servidor, que disposa d'un conjunt de catàlegs. L'esquema, el catàleg i el servidor formen una jerarquia de components que anomenem *entorn SQL* i que estudiarem en aquest mòdul didàctic.

Perquè un usuari o aplicació pugui executar sentències SQL sobre un conjunt de taules en un servidor, cal establir-hi una connexió prèvia. Les sentències SQL que es duen a terme mentre hi ha una connexió oberta formen el que anomenem *sessió* i, dins d'una sessió, s'executen transaccions. Les connexions, les sessions i les transaccions també seran objecte d'estudi d'aquest mòdul didàctic.

Finalment, cal tenir en compte que sovint hi ha divergències entre el que diu l'estàndard SQL i les implementacions dels diversos proveïdors de sistemes de gestió de bases de dades relacionals (SGBD). En el nostre cas, estudiarem la manera com implementa PostgreSQL els procediments emmagatzemats i els disparadors. També analitzarem les diferències que hi ha entre la jerarquia de components de l'entorn SQL definida en l'SQL estàndard i la que ofereix PostgreSQL.

Objectius

Els materials didàctics associats a aquest mòdul pretenen facilitar la consecució dels objectius següents:

1. Saber utilitzar les sentències de l'SQL estàndard per a emprar servidors, catàlegs, esquemes, connexions, sessions i transaccions.

2. Conèixer les diferències que hi ha entre la jerarquia de components de l'entorn SQL definida en l'SQL estàndard i la que ofereix PostgreSQL.

3. Conèixer les sentències que ofereix PostgreSQL per a fer servir esquemes i bases de dades.

4. Completar l'estudi dels components lògics d'una base de dades; és a dir, els procediments emmagatzemats i els disparadors.

5. Utilitzar les sentències que ofereix PostgreSQL per a definir procediments emmagatzemats i disparadors.

1. Entorn SQL

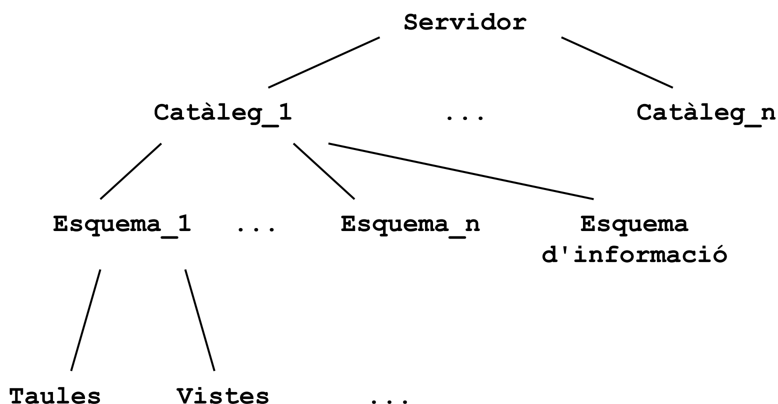
Els components lògics d'una base de dades (BD), com ara les taules i les vistes, i altres components lògics que estudiarem en aquest mòdul didàctic (procediments emmagatzemats i disparadors), pertanyen a una BD ubicada en un servidor. Agafem, per exemple, una taula que guarda dades d'empleats. Per a poder accedir a les dades d'aquesta taula, necessitem conèixer, entre altres coses, el servidor de BD i la BD dins d'aquest servidor. El servidor de BD i la BD formen part del que anomenem *entorn SQL*. Un entorn SQL és el marc on hi ha les dades d'una BD i les sentències SQL per a utilitzar-les.

Un entorn SQL consta d'una estructura jeràrquica de components, en què cada component té un paper. La figura següent mostra l'estructura de components de l'entorn SQL, definida en l'SQL estàndard.

Versió SQL

En aquest mòdul didàctic, quan parlem de l'SQL estàndard, sempre ens referirem a la darrera versió de l'estàndard, ja que té com a subconjunt totes les anteriors.

Components de l'entorn SQL de l'SQL estàndard



Abans de definir formalment cadascun dels components de l'entorn SQL, començarem il·lustrant per a què serveixen, mitjançant un exemple simple.

Exemple

Suposem que l'empresa ABC disposa de la versió 1.0 d'una aplicació que accedeix a una certa BD. La BD és ubicada en un servidor. Dins d'aquest servidor s'ha definit un catàleg per a emmagatzemar les dades de la versió 1.0 de l'aplicació i, dins d'aquest catàleg, s'han creat tres esquemes: desenvolupament, preproducció i producció. A continuació expliquem cadascun d'aquests esquemes:

- **Desenvolupament**: conté les taules, les vistes i altres components lògics que els desenvolupadors de l'aplicació utilitzen per a fer proves de l'aplicació.
- **Preproducció**: conté les taules i altres components lògics de la versió de l'aplicació que l'empresa posarà a l'entorn de producció el mes vinent.
- **Producció**: conté les taules i els components lògics de la versió de l'aplicació que està en producció actualment.

L'any vinent, l'empresa ABC planeja treure la versió 2.0 de la seva aplicació. Per començar a preparar l'entorn de la BD, fa un segon catàleg on es crearan els tres esquemes anteriors: desenvolupament, preproducció i producció. En cadascun d'aquests esquemes es tornen a crear les taules i altres components lògics emprats en la versió 1.0 de l'aplicació i, finalment, es copien les dades a les noves taules.

De l'exemple anterior es pot deduir que l'esquema de BD agrupa un conjunt de components lògics: taules, vistes, etc. L'esquema és, per tant, la unitat bàsica per a organitzar components lògics. Un catàleg consta d'un grup d'esquemes i un servidor (en anglès, *cluster*) és un conjunt de catàlegs.

Perquè un usuari o aplicació pugui accedir a un servidor de BD cal establir-hi una connexió prèvia. Les sentències SQL que s'executen mentre hi ha una connexió activa a un servidor formen una sessió, i en una sessió s'executen transaccions contra la BD.

En els subapartats següents estudiarem amb més detall els components de l'entorn SQL, i també els conceptes relacionats de connexió, sessió i transacció.

1.1. Esquema, catàleg i servidor

L'SQL estàndard no disposa de cap sentència de creació de BD; en comptes d'aquesta, utilitza el component esquema i el component catàleg.

L'esquema és l'element que permet agrupar un conjunt de taules i altres components lògics, que són propietat d'un usuari.

La instrucció `CREATE DATABASE`

Molts dels sistemes relacionals comercials (com és el cas de PostgreSQL, DB2, SQL Server i d'altres) han incorporat sentències de creació de BD amb la sintaxi següent: `CREATE DATABASE <nom_BD>`. Un cop creada la BD, normalment s'ha d'utilitzar una sentència per a accedir-hi. Podeu consultar el manual de PostgreSQL per veure la sintaxi d'aquesta sentència.

La sentència `CREATE SCHEMA` de l'SQL estàndard permet crear un esquema i té la sintaxi següent.

```
CREATE SCHEMA [<nom_cataleg>.]<nom_esquema> |
              AUTHORIZATION <ident_usuari> |
              [<nom_cataleg>.]<nom_esquema>
              AUTHORIZATION <ident_usuari>
              [<llista_elements_esquema>];
```

La nomenclatura utilitzada en aquesta sentència i d'ara endavant és la següent:

- Les paraules en negreta són paraules reservades del llenguatge.
- La notació `[...]` vol dir que el que hi ha entre els claudàtors es podria posar o no.
- La notació `A|...|B` vol dir que hem d'escollir entre totes les opcions, però n'hem de posar una obligatòriament.

La sentència de creació d'esquemes permet que un conjunt de taules i altres components lògics (designats `<llista_elements_esquema>`) es puguin agrupar sota un mateix esquema (`<nom_esquema>`) i que tinguin un propietari (`<ident_usuari>`). Opcionalment, es pot indicar el nom del catàleg al qual pertany l'esquema. Encara que molts paràmetres de la sentència `CREATE SCHEMA` són opcionals, com a mínim s'ha d'indicar el nom de l'usuari propietari de l'esquema o el nom de l'esquema.

Exemple de creació d'esquemes

En aquest exemple es crea l'esquema anomenat `empresa` que pertany a l'usuari `pere`. Dins de l'esquema `empresa`, es creen dues taules: `departaments` i `empleats`.

```
CREATE SCHEMA empresa AUTHORIZATION 'pere';
  CREATE TABLE departaments (
    codi_dept integer primary key,
    nom_dept varchar(30) not null);
  CREATE TABLE empleats(
    codi_empl integer primary key,
    nom_empl varchar(50) not null,
    codi_dept integer references departaments);
```

No cal crear l'esquema i tots els elements corresponents alhora. Primer es pot crear l'esquema, a continuació definir-lo com a esquema de treball i després afegir-hi elements. Per canviar d'esquema, l'SQL estàndard té la sentència `SET SCHEMA` amb la sintaxi següent:

```
SET SCHEMA <nom_esquema>;
```

Exemple de canvi d'esquema

En aquest exemple es crea l'esquema `empresa` i, a continuació, es defineix com a esquema de treball. Un cop fet això, les sentències de creació de les taules `departaments` i `empleats` es fan en l'esquema de treball, és a dir, a `empresa`.

```
CREATE SCHEMA empresa AUTHORIZATION 'pere';
  SET SCHEMA empresa;
  CREATE TABLE departaments (
      codi_dept integer primary key,
      nom_dept varchar(30) not null);
  CREATE TABLE empleats(
      codi_empl integer primary key,
      nom_empl varchar(50) not null,
      codi_dept integer references departaments);
```

La sentència per a esborrar un esquema és `DROP SCHEMA` i té la sintaxi següent:

```
DROP SCHEMA <nom_esquema> [RESTRICT | CASCADE];
```

L'opció `RESTRICT` permet esborrar l'esquema només si aquest està completament buit. En canvi, l'opció `CASCADE` permet esborrar-lo encara que contingui elements. En aquest darrer cas, quan s'esborra l'esquema també se n'esborren tots els elements.

Les taules i altres components lògics es creen i es manegen dins d'un esquema. D'una manera anàloga, els esquemes es creen, es modifiquen i s'esborren dins d'un catàleg. Per tant, un catàleg és un component que conté un conjunt d'esquemes.

Cada catàleg conté, a més dels esquemes definits pels usuaris, un esquema d'informació anomenat *information schema*, que conté tota la informació sobre els esquemes que defineixen els usuaris, és a dir, els noms i atributs de les taules, les restriccions d'integritat definides en les taules, la definició de les vistes, etc. Així doncs, l'esquema d'informació conté metainformació (informació relativa als components lògics definits en cada esquema d'usuari).

Per a consultar la metainformació, l'esquema d'informació consta d'un conjunt de vistes definides a partir d'un conjunt de taules de sistema, que només són accessibles per l'administrador de la BD.

Entre les vistes de l'esquema d'informació trobem les següents:

- `SCHEMATA`: conté informació sobre cada esquema del catàleg.
- `DOMAINS`: conté informació sobre els dominis definits en els esquemes del catàleg.
- `TABLES`: conté informació sobre les taules definides en els esquemes del catàleg.

- `VIEWS`: conté informació sobre les vistes definides en els esquemes del catàleg.

Les vistes de l'esquema d'informació permeten als usuaris consultar (però no modificar d'una manera directa) informació relativa als objectes de la BD sobre els quals tenen privilegis.

Exemple

Suposem que l'usuari `Pere` té un esquema on ha creat dues taules: `empleats` i `departaments`. Quan en `Pere` consulti la vista `TABLES` de l'esquema d'informació, obtindrà dues entrades amb informació, una per cadascuna de les dues taules que té en l'esquema. Quan en `Pere` crea una taula nova en el seu esquema, per exemple la taula `projectes`, i posteriorment torna a consultar la vista `TABLES` de l'esquema d'informació, li surten tres entrades corresponents a les tres taules que ha creat en el seu esquema: `empleats`, `departaments` i `projectes`.

De l'exemple anterior, es desprèn que l'actualització de les vistes de l'esquema d'informació no és duta a terme pels usuaris d'una manera directa. Es produeix quan els usuaris fan actualitzacions sobre els objectes de la BD.

Finalment, cal esmentar que cada usuari té accés de consulta només a la informació de l'esquema d'informació a la qual està autoritzat a accedir (per exemple, informació sobre les taules que ha creat). És a dir, un usuari no podrà accedir, consultant les vistes de l'esquema d'informació, a informació sobre objectes sobre els quals no té privilegis.

Abans hem estudiat les sentències d'SQL estàndard per a crear i esborrar esquemes. Pel que fa als catàlegs, l'SQL estàndard no disposa de sentències per a crear-los ni esborrar-los. Aquestes sentències són específiques de cada sistema de gestió de bases de dades (SGBD).

Finalment, un conjunt de catàlegs pertany a un servidor. L'SQL estàndard tampoc no disposa de sentències per a crear i esborrar servidors. Aquestes sentències són específiques de cada SGBD; tanmateix, l'SQL estàndard proporciona les sentències `CONNECT` i `DISCONNECT` per a connectar-se i desconnectar-se d'un servidor.

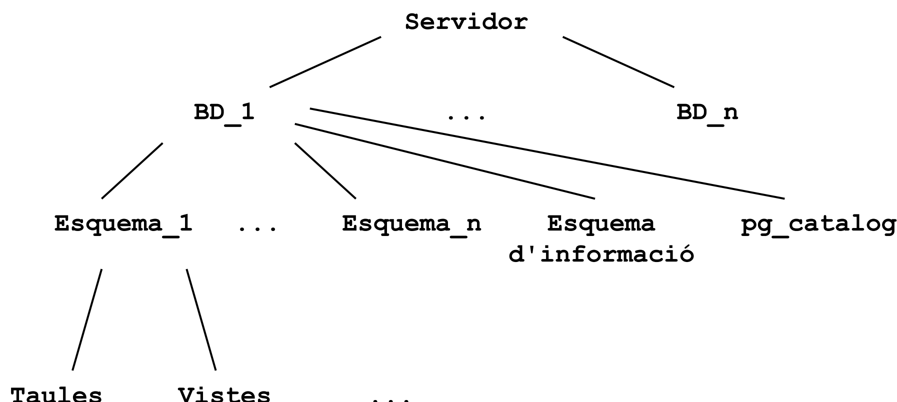
Vegeu també

Les sentencies `CONNECT` i `DISCONNECT` s'estudien al subapartat 1.2.

Un cop estudiats els elements de l'entorn SQL que ofereix l'SQL estàndard, vegem quins d'aquests components té PostgreSQL.

A PostgreSQL, un servidor conté un conjunt de BD. En una connexió a un servidor, un usuari accedeix només a una BD que s'especifica en el moment d'establir la connexió. Per la seva banda, una BD conté un conjunt d'esquemes. La figura següent mostra aquesta jerarquia de components.

Components de l'entorn SQL de PostgreSQL



Per crear una BD, PostgreSQL ofereix la sentència CREATE DATABASE, que té la sintaxi simplificada següent.

```
CREATE DATABASE <nom_bd>;
```

La sentència DROP DATABASE permet esborrar una BD i té la sintaxi següent.

```
DROP DATABASE <nom_bd>;
```

Per manejar esquemes, PostgreSQL ofereix les sentències CREATE SCHEMA i DROP SCHEMA. La primera sentència permet crear un esquema i la segona, esborrar-lo. La sintaxi de la sentència CREATE SCHEMA és la següent.

```
CREATE SCHEMA <nom_esquema>;
```

La sentència DROP SCHEMA de PostgreSQL té la mateixa sintaxi que en l'SQL estàndard.

En PostgreSQL, totes les BD tenen un esquema per defecte anomenat public. Si no s'especifica el contrari, tots els components lògics que crea un usuari, com ara taules o vistes, es creen dins d'aquest esquema.

Dins la jerarquia d'objectes de PostgreSQL, s'accedeix a un component lògic com ara una taula, especificant el nom de la BD a la qual pertany i el nom de l'esquema dins de la BD. Per exemple, si tenim la BD anomenada empresa, que conté l'esquema public, on s'han creat les taules d'empleats, departaments i projectes, el nom complet per a accedir a la taula d'empleats serà empresa.public.empleats.

No obstant això, PostgreSQL permet accedir a un objecte sense haver d'utilitzar el nom complet (`nom_bd.nom_esquema.nom_objecte`). Això és possible perquè PostgreSQL permet definir una llista d'esquemes on es poden buscar els objectes de la BD en una sessió d'usuari. Aquesta llista d'esquemes es defineix en la variable de sistema `search_path`. La sentència `SET SEARCH_PATH TO <llista_esquemes>` permet definir els esquemes de la variable `search_path`.

Exemple de definició de `search_path`

La sentència següent indica que els noms dels objectes es buscaran en els esquemes anomenats `el_meu_esquema` i, si no són aquí, en l'esquema `public`. A més a més, el primer esquema especificat en el `search_path` és l'esquema on es crearan els components lògics que defineixi l'usuari.

```
SET SEARCH_PATH TO el_meu_esquema, public;
```

Exemple de creació de taula dins d'un esquema

La sentència `SET SEARCH_PATH` defineix l'esquema de treball durant la sessió de l'usuari. Per tant, la taula `departaments` es crearà en l'esquema `empresa`.

```
SET SEARCH_PATH TO empresa;
CREATE TABLE Departaments (
  codi_dpt integer primary key,
  nom_dpt varchar(30) not null,
  ciutat_dpt varchar(30),
  pressupost float not null);
```

Cada BD de PostgreSQL té un esquema addicional: l'esquema d'informació (*information schema*). Com que l'esquema d'informació és definit per l'SQL estàndard, és portable i estable; per tant, no conté característiques o funcionalitats específiques de PostgreSQL.

Recordeu

L'esquema d'informació consta d'un conjunt de vistes que contenen metainformació sobre la BD (per exemple, informació relativa a les taules, les restriccions d'integritat definides en les taules, etc.).

A més de l'esquema d'informació, PostgreSQL té un esquema específic de sistema, anomenat `pg_catalog`, que conté tota la informació dels esquemes definits pels usuaris dins de la BD, és a dir, els noms i atributs de les taules, de les restriccions d'integritat, etc. El `pg_catalog` de PostgreSQL està format per un conjunt de taules, com ara les següents:

- `pg_class`: emmagatzema informació sobre les taules, els índexs, les seqüències, les vistes i altres components lògics definits en la BD.
- `pg_attribute`: emmagatzema informació sobre les columnes de les taules de la BD.
- `pg_triggers`: emmagatzema informació sobre els disparadors de la BD.
- `pg_proc`: emmagatzema informació sobre els procediments emmagatzemats definits en la BD.

Vegeu també

En els apartats 2 i 3 d'aquest mòdul s'estudien els components lògics *procediments emmagatzemats* i *disparadors*.

Finalment, PostgreSQL proporciona un conjunt de vistes definides sobre les taules de `pg_catalog` que permeten consultar les dades emmagatzemades al `pg_catalog`. La informació que proporcionen és la mateixa que la facilitada per les vistes de l'esquema d'informació. Com que l'esquema d'informació és

estable, perquè està definit en l'SQL estàndard, és recomanable consultar les vistes de l'esquema d'informació en comptes de les vistes específiques de PostgreSQL.

1.2. Connexió, sessió i transacció

Per a accedir a un servidor de BD, cal establir-hi una connexió prèviament.

> Una connexió es pot definir com l'associació que es crea entre un client i un servidor quan el client manifesta que té la intenció de treballar amb la BD sol·licitant una connexió.

L'SQL estàndard proporciona les sentències següents per a manejar connexions: CONNECT i DISCONNECT. La sentència CONNECT permet establir una connexió amb un servidor. Té la sintaxi següent.

```
CONNECT TO <nom_servidor> [AS <nom_connexio>]
[USER <ident_usuari>];
```

Exemple d'utilització de la sentència CONNECT

La sentència següent permet que l'usuari pere estableixi una connexió anomenada connexio1 al servidor anomenat algun_servidor.

```
CONNECT TO 'algun_servidor' AS 'connexio1' USER 'pere'
```

Un usuari pot establir més d'una connexió amb un servidor. Quan l'usuari es connecta per segona vegada amb el servidor, la darrera connexió esdevé la connexió actual (en anglès, current). La primera connexió que s'ha establert queda adormida i se'n mantenen les característiques i la informació de context per mitjà de l'SGBD, per a poder-la restaurar més endavant. Si el segon intent de connexió falla, aleshores la primera connexió es manté com a connexió actual.

Després d'establir la connexió amb el servidor, es pot accedir a un esquema concret amb la sentència següent.

```
SET SCHEMA <nom_esquema>;
```

Per a tancar una connexió, l'SQL estàndard disposa de la sentència DISCONNECT, que té la sintaxi següent.

```
DISCONNECT <nom_connexio> | CURRENT | ALL;
```

Amb la sentència DISCONNECT es pot tancar la connexió que té <nom_connexió>, la connexió actual o totes les connexions que l'usuari tenia obertes.

Les sentències SQL que s'executen mentre hi ha una connexió activa a un servidor formen una sessió. Per tant, una sessió és el context en què un usuari o aplicació executa una seqüència de sentències SQL mitjançant una connexió.

Cada sessió té un catàleg i un esquema de treball, que es poden definir amb les sentències SET CATALOG i SET SCHEMA, respectivament. De manera addicional, les característiques de les transaccions que s'executen en una sessió es poden definir per mitjà de la sentència SET SESSION CHARACTERISTICS, que té la sintaxi següent.

```
SET SESSION CHARACTERISTICS AS
<mode_transaccio> [, <mode_transaccio> …];

en que <mode_transaccio> pot ser:
mode_d'acces | ISOLATION LEVEL <nivell d'aillament>
en que mode_d'acces pot ser: READ ONLY | READ WRITE
en que nivell d'aillament pot ser: READ UNCOMMITTED |
READ COMMITTED | REPETEABLE READ | SERIALIZABLE
```

Una transacció és un conjunt de sentències SQL de lectura (consultes) i actualització de la BD, que confirma o cancel·la els canvis que s'hi han dut a terme; per tant, es tracta d'una unitat indivisible de treball, de manera que, o bé tot el conjunt de sentències SQL s'executa completament, si és el cas d'una sèrie de canvis en la BD, o bé no s'executa cap sentència.

Tornant a la sentència SET SESSION CHARACTERISTICS, el mode d'accés permet especificar si la transacció només llegirà dades de la BD (mode d'accés READ ONLY) o si, per contra, a més de llegir-ne, també en modificarà el contingut (mode d'accés READ WRITE). El nivell d'aïllament indica la independència amb què treballen les transaccions de la BD entre elles. Si no s'hi indica res, l'SGBD garantirà l'aïllament total de la transacció (nivell d'aïllament SERIALIZABLE).

Pel que fa a les transaccions, l'SQL estàndard ofereix sentències que permeten als usuaris delimitar l'inici i l'acabament de les transaccions. L'inici d'una transacció es pot indicar d'una manera explícita amb la sentència SET TRANSACTION.

```
SET TRANSACTION <mode_transaccio> [, <mode_transaccio> …];
```

La utilització de la sentència SET TRANSACTION no és obligatòria. Si no s'indica d'una manera explícita l'inici d'una transacció, l'SGBD implícitament començarà una transacció amb l'execució de la primera sentència SQL que es dugui a terme dins de la sessió. La transacció romandrà activa fins que, d'una manera explícita i obligatòria, se n'indiqui l'acabament.

Per indicar l'acabament d'una transacció, l'SQL estàndard ens ofereix la sentència següent.

```
{COMMIT|ROLLBACK} [WORK];
```

La diferència entre COMMIT i ROLLBACK és que, mentre que la sentència COMMIT confirma tots els canvis produïts contra la BD durant l'execució de la transacció, la sentència ROLLBACK els desfà i deixa la BD com estava abans de començar la transacció. La paraula reservada WORK només serveix per a explicar què fa la sentència i és opcional.

Exemple d'establiment de les característiques d'una sessió

En aquest exemple, l'usuari pere estableix una connexió amb el servidor servidor_ABC. En la sessió de treball estableix les característiques següents: la transacció llegirà i escriurà dades de la BD (mode READ WRITE) i el nivell d'aïllament serà SERIALIZABLE. Després d'executar les sentències SQL, l'usuari Pere dóna per bons els canvis que ha dut a terme la transacció i acaba la transacció amb la sentència COMMIT. Finalment, posa fi a la connexió.

```
CONNECT TO 'servidor_ABC' AS 'connexio1' USER 'pere';
SET SESSION CHARACTERISTICS AS
  READ WRITE
  ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM departaments
INSERT INTO departaments VALUES (...);
COMMIT;
DISCONNECT connexio1;
```

2. Procediments emmagatzemats

> Un procediment emmagatzemat és una acció o funció definida per un usuari que proporciona un servei determinat. Un cop creat, el procediment es guarda en la BD i es tracta com un objecte més d'aquesta.

Els procediments emmagatzemats es poden executar:

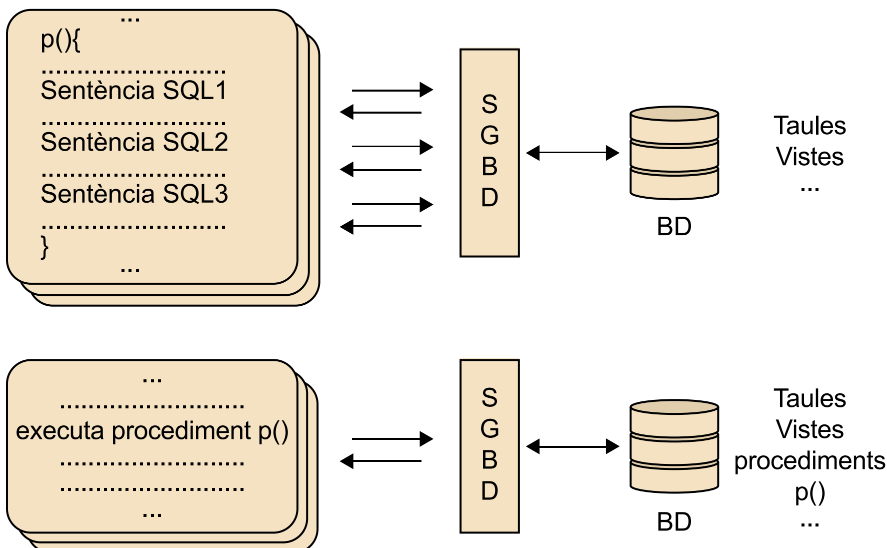
- D'una manera directa, fent ús de l'SQL interactiu.
- Des d'una aplicació que accedeixi a la BD.
- Des d'un altre procediment emmagatzemat.

Els procediments emmagatzemats serveixen per a:

- Simplificar el desenvolupament d'aplicacions.
- Millorar el rendiment de les BD.
- Encapsular les operacions que s'efectuen en una BD.

Per il·lustrar les característiques prèvies, ens fixarem en la figura següent. S'hi presenten dues possibles arquitectures de desenvolupament d'aplicacions que accedeixen a una BD, una de les quals fa ús dels procediments emmagatzemats i l'altra, no.

SGBD sense procediments emmagatzemats i SGBD amb procediments emmagatzemats



Com es pot veure en la figura anterior, si `p()` és un mètode que conceptualment resol una operació determinada que necessita executar tot un conjunt de sentències SQL, podem traslladar-ne la definició a la BD, i transformar-lo en un procediment emmagatzemat. A més, també poden fer ús del procediment

emmagatzemat altres aplicacions, amb la qual cosa aconseguim simplificar el desenvolupament d'aplicacions: transferim lògica relativa a la BD cap a l'SGBD i afavorim la reutilització de codi.

Si pensem en un entorn distribuït, el trànsit per mitjà de la xarxa disminuirà en cas d'utilitzar procediments emmagatzemats. Amb l'ús d'aquests procediments s'evitarà que cada sentència SQL s'enviï d'una manera individual per la xarxa i que els resultats de l'execució de cada sentència SQL siguin recollits per l'aplicació individualment. A més, els procediments emmagatzemats es guarden precompilats en la BD, de manera que la seva estratègia d'execució es calcula en el moment de crear-los; això no succeeix necessàriament quan la nostra aplicació no fa servir procediments emmagatzemats. Per tant, la utilització de procediments emmagatzemats millora el rendiment de les BD.

Finalment, el procediment emmagatzemat encapsula el conjunt de sentències SQL que incorpora; el programador que utilitza aquest procediment només ha de saber que un procediment emmagatzemat concret li proporciona un servei determinat, i que, en cas de no poder-lo satisfer, es reportarà un error. No necessita saber les sentències SQL que incorpora el procediment ni els elements de l'esquema de la BD que manipulen aquestes sentències. Això ajuda a controlar les operacions que els usuaris efectuen en la BD.

Procediment emmagatzemat i control d'errors

El procediment emmagatzemat no solament resol l'operació, sinó que també es responsabilitza del control de totes les situacions d'error que es produeixin.

2.1. Sintaxi dels procediments emmagatzemats a PostgreSQL

En PostgreSQL, per a poder escriure procediments emmagatzemats, disposem de dos tipus de sentències:

- Sentències pròpies de l'SQL.
- Sentències pròpies d'algun llenguatge procedimental, com, per exemple, PL/pgSQL o PL/Python.

PL

PL és la sigla de *procedural language*, que és la denominació que reben els llenguatges procedimentals a PostgreSQL.

En aquest mòdul estudiarem el llenguatge procedimental PL/pgSQL, que és un dels que venen amb la distribució estàndard de PostgreSQL. El PL/pgSQL complementa l'SQL, que no és un llenguatge computacionalment complet. En essència, el PL/pgSQL proporciona sentències que permeten controlar el flux d'execució dels nostres procediments emmagatzemats.

Nota

De moment, i fins que no arribem a la secció d'errors, s'omet el tractament d'errors en els exemples de procediments emmagatzemats.

Exemple de procediment emmagatzemat escrit en PL/pgSQL

A partir de la taula `Clients`,

```
CREATE TABLE clients (
  dni char(9) primary key,
  nom varchar(15) not null,
  cognom1 varchar(15) not null,
  cognom2 varchar(15) not null,
  carrer varchar(20) not null,
  num_carrer varchar(4) not null,
  cp char(5) not null,
  ciutat varchar(15) not null);
```

podem definir el procediment emmagatzemat `trobar_ciutat`:

```
CREATE FUNCTION trobar_ciutat(dni_client char(9))
RETURNS varchar(15) AS $$
DECLARE
  ciutat_client varchar(15);
BEGIN
  SELECT ciutat into ciutat_client
  FROM clients
  WHERE dni=dni_client;
  RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;
```

Podem executar el procediment anterior amb la sentència següent:

```
SELECT * FROM trobar_ciutat('45678900J');
```

L'exemple anterior mostra alguns elements bàsics en la definició d'un procediment emmagatzemat.

- Presència de la sentència `CREATE FUNCTION`, que és la sentència que permet crear procediments emmagatzemats en PostgreSQL. La sentència `CREATE FUNCTION` i la paraula `END` seguida de punt i coma delimiten el procediment.

- Nom del procediment; en aquest cas, `trobar_ciutat`.

- Definició dels paràmetres (nom i tipus de dades) necessaris per a invocar l'execució del procediment; en aquest cas, `dni_client`, que és de tipus `CHAR(9)`.

- Definició dels tipus de dades del paràmetre retornat pel procediment, mitjançant la clàusula `RETURNS`; en aquest exemple, `VARCHAR(15)`.

- Definició del cos del procediment, que sovint és delimitat per les marques `$$`.

Marca $$

La marca `$$` s'utilitza habitualment per a delimitar el cos del procediment emmagatzemat. El cos del procediment és una cadena de caràcters i, per tant, hauria d'estar delimitada per cometes simples. No obstant això, l'ús de cometes simples pot ser poc pràctic, si a dins hi ha altres cometes. Les cometes interiors s'haurien de duplicar per a diferen-

ciar-les de les comentes que delimiten l'inici i el final del procediment. Com que això pot fer que el codi sigui poc llegible, sovint s'usa la marca $$ per a delimitar l'inici i el final del cos del procediment.

- Definició de les variables del procediment; en l'exemple, `ciutat_client`. La secció de declaració de variables és precedida per la clàusula `DECLARE`. Si el procediment no té variables, es pot ometre aquesta clàusula.

- Presència de les paraules `BEGIN` i `END`, per a delimitar el bloc que conté les sentències del procediment; en aquest cas, el bloc conté la sentència `SELECT` de l'SQL.

- Paràmetre retornat pel procediment, precedit per la clàusula `RETURN`; en aquest cas, `ciutat_client`.

- Nom del llenguatge procedimental utilitzat per a escriure el procediment; en aquest cas, `plpgsql`.

Notació

Si un procediment no retorna res, només cal posar la paraula `void` després de la clàusula `RETURNS`.

Com es pot veure en l'exemple anterior, un cop s'ha creat un procediment, es pot executar mitjançant la sentència de l'SQL `SELECT * FROM nom_procediment (paràmetres)`.

Lectura recomanada

La sentència `SELECT * FROM nom_procediment (paràmetres)` no és l'única manera d'executar procediments emmagatzemats en PostgreSQL. Podeu consultar les altres maneres de fer-ho al manual de PostgreSQL.

Per a esborrar un procediment emmagatzemat, s'utilitza la sentència `DROP FUNCTION`, que té la sintaxi següent.

```
DROP FUNCTION
nom_procediment(tipus_parametres_ entrada);
```

En la sentència `DROP FUNCTION`, cal especificar els tipus dels paràmetres d'entrada al procediment, perquè en PostgreSQL els procediments accepten sobrecàrrega (diversos procediments amb el mateix nom).

En l'exemple anterior, el procediment retorna un sol camp. Més endavant explicarem com es pot retornar un conjunt de camps i un conjunt de files.

El llenguatge PL/pgSQL proporciona bàsicament tres tipus de sentències:

- Sentències per a definir (`DECLARE`) variables.
- Sentències per a assignar valors a variables (`variable:=expressió`).
- Sentències per a controlar el flux d'execució d'un procediment:
  - Sentències condicionals: `IF` i `CASE`.
  - Sentències iteratives: `FOR`, `LOOP` i `WHILE`.
  - Sentències per a gestionar errors: `EXCEPTION` i `RAISE EXCEPTION`.

2.1.1. Paràmetres

Un procediment emmagatzemat pot rebre un conjunt de paràmetres d'entrada i retornar un resultat; també pot tenir paràmetres que són tant d'entrada com de sortida.

Lectura recomanada

Podeu consultar la sintaxi per a usar paràmetres d'entrada i sortida al manual de PostgreSQL.

A cada paràmetre d'entrada li posarem un nom i li associarem un tipus de dades. Per a associar un tipus de dades a un paràmetre d'entrada es poden fer servir els mateixos tipus de dades que en la definició de columnes de taules. També és possible especificar que el tipus de dades d'un paràmetre d'entrada és idèntic al tipus de dades d'una columna determinada d'una taula, mitjançant la clàusula `%TYPE`.

Per a retornar el resultat de la funció, n'hi ha prou d'indicar-ne el tipus de dades en la clàusula `RETURNS`.

Exemple d'ús de la clàusula `%TYPE`

Tenint en compte el que acabem d'explicar, podríem haver definit el procediment emmagatzemat trobar_ciutat de la manera següent:

```
CREATE or REPLACE FUNCTION trobar_ciutat(dni_client
  clients.dni%type) RETURNS varchar(15) AS $$
DECLARE
  ciutat_client clients.ciutat%type;
BEGIN
  SELECT ciutat INTO ciutat_client
  FROM clients
  WHERE dni=dni_client;
  RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;
```

Per a tornar a crear el procediment emmagatzemat `trobar_ciutat` hi ha dues alternatives. La primera consisteix a esborrar abans la versió antiga del procediment, mitjançant la sentència `DROP FUNCTION trobar_ciutat (char(9))`, i la segona, a crear el procediment amb la sentència `CREATE or REPLACE FUNCTION`. La clàusula `REPLACE` permet substituir la versió antiga del procediment emmagatzemat en la BD per la nova.

2.1.2. Variables

Per a definir variables dins d'un procediment emmagatzemat farem servir la clàusula `DECLARE` de PL/pgSQL, amb la sintaxi següent.

```
nom_variable [CONSTANT] tipus_dades [NOT NULL]
[{DEFAULT | :=}expressio];
```

La clàusula `CONSTANT` indica que la variable tindrà un valor constant durant l'execució del procediment; la clàusula `NOT NULL` assenyala que la variable no pot tenir un valor nul, i, finalment, la clàusula `DEFAULT` indica que la variable

tindrà un valor per defecte, que es podrà modificar més endavant en l'execució del procediment. Si no s'especifica un valor per defecte, la variable s'inicialitza a NULL.

Cada variable ha de tenir associat un nom i un tipus de dades. De manera anàloga al cas dels paràmetres d'entrada d'un procediment emmagatzemat, es poden utilitzar els mateixos tipus de dades que en la definició de columnes d'una taula i, també, és possible indicar que el tipus de dades d'una variable concreta és idèntic al tipus de dades d'una columna determinada d'una taula específica, mitjançant la clàusula `%TYPE`.

Una altra possibilitat quan es declara una variable, consisteix a indicar que el tipus de dades de la variable és idèntic al tipus de dades d'una fila d'una taula concreta. Això es fa mitjançant la clàusula `%ROWTYPE`. Amb aquesta clàusula, la variable definida té un tipus compost, que consta de tants camps com tingui la taula utilitzada en la declaració de la variable. Els tipus dels camps de la variable també coincideixen amb els de les columnes de la taula a partir de la qual es defineix la variable.

Exemple d'ús de `%ROWTYPE`

En l'exemple següent, la variable `var_client` té el mateix tipus que una fila de la taula de `clients`; és a dir, la variable `var_client` és de tipus compost, té tants camps com camps tingui la taula de `clients`.

```
CREATE FUNCTION…
DECLARE
var_client clients%ROWTYPE;
BEGIN
…
END;
$$ LANGUAGE plpgsql;
```

Per a accedir a cadascun dels camps d'una variable definida amb la clàusula `%ROWTYPE` es fa servir la notació següent: `nom_variable.nom_camp`. Els noms dels camps coincideixen amb els noms dels camps de la taula a partir de la qual es defineix la variable.

Si no s'inicialitzen les variables definides en el procediment, per defecte tenen valor nul. Hi ha diverses possibilitats per a assignar valors a les variables dels procediments:

- Sentència d'assignació de PL/PgSQL (`variable:=expressió`).
- Sentència `SELECT ... INTO` de l'SQL.
- Assignar a una variable el resultat d'executar un altre procediment.

Error d'execució

Quan a una variable declarada `NOT NULL` se li assigna un valor nul durant l'execució del procediment, es produeix un error d'execució.

Exemples d'assignacions

Sentència d'assignació de PL/pgSQL:

```
Comptador:=1;
```

Sentència `SELECT ... INTO`:

```
SELECT ciutat INTO ciutat_client
FROM clients
WHERE dni=dni_client;
```

Assignar a una variable el resultat d'executar un altre procediment:

```
import_comanda:=import_una_comanda(num_comanda);
```

o bé

```
SELECT * FROM import_una_comanda(num_comanda)
INTO import_comanda;
```

Les variables definides dins d'un procediment emmagatzemat són sempre variables locals; és a dir, l'àmbit de visibilitat d'una variable local queda restringit al procediment en què s'hagi definit.

Nota

Comptador, ciutat_client i import_comanda són variables.

Les variables **no es consideren objectes de la BD**. Els valors de les variables locals queden descartats un cop finalitza l'execució del procediment emmagatzemat en què han estat definides.

2.1.3. Sentències condicionals

La sentència `IF` de PL/pgSQL serveix per a establir condicions en el flux d'execució d'un procediment emmagatzemat. Té la sintaxi bàsica següent.

Lectura recomanada

PostgreSQL disposa d'una altra sentència condicional, la sentència `CASE`. Podeu consultar-ne la sintaxi al manual de PostgreSQL.

```
IF <condicio> THEN <bloc_de_sentencies>
ELSE <bloc_de_sentencies>
END IF;
```

Es poden establir diversos nivells d'imbricació mitjançant la clàusula `ELSIF`, com a alternativa a la clàusula `ELSE`. Per exemple, per a establir un nivell d'imbricació addicional, cal fer el següent.

```
IF <condicio> THEN <bloc_de_sentencies>
  ELSIF <condicio> THEN <bloc_de_sentencies>
     ELSE <bloc_de_sentencies>
END IF;
```

Per a especificar les condicions, disposem dels elements següents:

- Operadors lògics: `AND`, `OR`, `NOT`.
- Operadors de comparació: >, <, >=, <=, =, <>.

- Predicats propis de l'SQL: BETWEEN, IN, IS NULL, IS NOT NULL o LIKE.
- Consultes SQL.

Com que hi ha columnes de taules que poden tenir valor nul, existeixen predicats propis de l'SQL per a comparar amb el valor nul. Quan s'avaluen les expressions de les condicions també cal tenir en compte els valors nuls, ja que aquestes expressions es poden avaluar a nul. Cal destacar que, si alguna expressió dins de la condició avalua a NULL, tota la condició no avalua a cert, llevat del cas en què comprovem d'una manera explícita condicions del tipus IS NULL o IS NOT NULL.

Exemple d'ús de sentències condicionals

Imaginem que tenim la taula Clients següent a la qual hem afegit el camp num_com. Aquest camp emmagatzema el nombre de comandes que ha fet un client concret.

```
CREATE TABLE clients(
  dni varchar(9) primary key,
  nom varchar(15) not null,
  cognom1 varchar(15) not null,
  cognom2 varchar(15) not null,
  carrer varchar(20) not null,
  num_carrer varchar(4) not null,
  cp char(5) not null,
  ciutat varchar(15) not null,
  num_com integer);
```

El procediment calcul_descompte_client calcula el descompte que s'ha d'aplicar a cada client passat com a paràmetre, segons el nombre de comandes (camp num_com de la taula de clients) que ha fet: el descompte és del 0% si n'ha fet menys de cinc; del 3% si n'ha fet entre cinc i nou; del 5% si n'ha fet entre deu i catorze, i del 10% si n'ha fet quinze o més.

```
CREATE FUNCTION calcul_descompte_client
    (dni_client clients.dni%type)
RETURNS integer AS $$
DECLARE
  descompte integer;
  num_com_client integer;
BEGIN
  IF((SELECT COUNT(*) FROM clients
        WHERE dni=dni_client)=1)THEN
    num_com_client=(SELECT num_com
        FROM clients
        WHERE dni=dni_client);
    IF (num_com_client IS NULL) THEN
      descompte=NULL;
    ELSIF (num_com_client<5) THEN
      descompte=0;
    ELSIF (num_com_client<10) THEN
      descompte=3;
    ELSIF (num_com_client<15) THEN
      descompte=5;
    ELSE
      descompte=10;
    END IF;
  END IF;
  RETURN descompte;
END;
$$LANGUAGE plpgsql;
```

2.1.4. Sentències iteratives

El PL/pgSQL proporciona tres tipus de sentències iteratives:

- Les sentències `LOOP` i `WHILE` s'utilitzen per a definir bucles l'acabament dels quals estigui definit per una expressió condicional.

- La sentència `FOR` es pot emprar quan sabem *a priori* el nombre d'iteracions que s'han executar. També es pot fer servir per a iterar sobre el conjunt de files retornades per una consulta SQL o una sentència d'execució d'un procediment emmagatzemat.

Atesa la tipologia de les operacions que s'efectuen contra una BD, la sentència iterativa més freqüent és `FOR`; concretament, la versió que permet iterar sobre el conjunt de files retornades per una consulta SQL. Per aquest motiu ens centrarem en aquesta sentència, la sintaxi de la qual es mostra a continuació.

Lectura recomanada

Podeu consultar la sintaxi de les sentències `LOOP`, `WHILE` i `FOR` (en la versió en què se sap *a priori* el nombre d'iteracions que s'han de dur a terme) en els manuals de PostgreSQL.

```
FOR <llista_variables>
         IN <consulta_SQL_o_execucio_procediment> LOOP
        <bloc_sentencies>
END LOOP;
```

Quan s'executa una sentència `FOR` per a accedir al conjunt de resultats d'una consulta SQL, es duen a terme les accions següents:

1) S'executa la consulta SQL o el procediment emmagatzemat associat al `FOR`.

2) S'assigna a cada variable de la llista de variables el conjunt de valors associat a la fila actual, que forma part del resultat de la consulta SQL o de l'execució del procediment emmagatzemat.

3) S'executa el bloc de sentències.

4) S'obté automàticament la fila següent que forma part del resultat de la consulta SQL o de l'execució del procediment emmagatzemat.

5) Es tornen a executar els passos 2, 3 i 4 mentre quedin files que formin part del resultat de la consulta SQL o de l'execució del procediment emmagatzemat.

Exemple d'ús de sentències iteratives

Imaginem que tenim les taules `Comandes`, `items` i `Items_comanda` següents:

```
CREATE TABLE comandes(
  num_com integer primary key,
  dni varchar(9) not null references clients,
  data_arribada date not null,
  import_total numeric);

CREATE TABLE items(
  num_item integer primary key,
  preu_unitat numeric not null);

CREATE TABLE items_comanda(
  num_item integer references items,
  num_com integer references comandes,
  quantitat integer not null,
  primary key(num_item,num_com));
```

El procediment emmagatzemat `import_una_comanda` calcula l'import total de la comanda passada com a paràmetre.

```
CREATE or REPLACE FUNCTION import_una_comanda
(com_client comandes.num_com%TYPE) RETURNS numeric AS $$
DECLARE
    total_comanda comandes.import_total%type;
    dades_item_preu items.preu_unitat%type;
    dades_item_qtt items_comanda.quantitat%type;
  BEGIN
    total_comanda=0.0;
    FOR dades_item_preu, dades_item_qtt IN
            SELECT ic.quantitat, i.preu_unitat
              FROM items_comanda ic, items i
            WHERE i.num_item=ic.num_item AND
              ic.num_com=com_client LOOP
        total_comanda=total_comanda+
        (dades_item_qtt*dades_item_preu);
  END LOOP;
  RETURN total_comanda;
END;
$$LANGUAGE plpgsql;
```

L'exemple anterior mostra la utilització de la sentència `FOR` per a accedir al resultat d'una consulta SQL. La quantitat i el preu unitari de cada ítem demanat en la comanda passada com a paràmetre s'emmagatzemen en les variables `dades_item_preu` i `dades_item_qtt`. Amb aquests valors es va calculant l'import de la comanda i s'emmagatzema a la variable `total_comanda`. Al final, el procediment retorna aquest import.

És possible combinar la sentència `FOR` amb sentències `UPDATE` o `DELETE`, que actuïn sobre la fila que es tracta en una determinada iteració d'un bucle. Aquest ús s'il·lustra en l'exemple següent.

Exemple d'ús de sentències iteratives amb la sentència UPDATE

Imaginem que tenim les taules anteriors (`Comandes`, `items`, `Items_comanda` i `clients`). El procediment emmagatzemat següent, anomenat `import_totes_comandes`, calcula l'import de totes les comandes d'un client, el DNI del qual es passa com a paràmetre al procediment. L'import total de les comandes del client s'emmagatzema a la taula `Comandes` (atribut `import_total`). Per aquest motiu, el procediment no retorna cap resultat.

```
CREATE FUNCTION import_totes_comandes
      (dni_client clients.dni%type)
RETURNS void AS $$
DECLARE
  num_comanda comandes.num_com%type;
  import_comanda comandes.import_total%type;
BEGIN
FOR num_comanda IN SELECT num_com FROM comandes
                   WHERE dni=dni_client LOOP
  import_comanda:=import_una_comanda(num_comanda);
  UPDATE comandes SET import_total=import_comanda
  WHERE num_com=num_comanda;
END LOOP;
END;
$$LANGUAGE plpgsql;
```

Per a invocar el procediment emmagatzemat `import_una_comanda` també podem utilizar la sentència SELECT següent:

```
SELECT * FROM import_una_comanda(numcomanda) INTO import_comanda;
```

2.1.5. Retorn de resultats

En els exemples anteriors hem vist com es retorna un sol resultat (de tipus de dades simple, com, per exemple, `integer` o `char`) des d'un procediment emmagatzemat. A continuació veurem com es pot retornar:

1) un conjunt de camps que formen una sola fila (un sol resultat, però de tipus de dades compost) i
2) un conjunt de files (que poden ser de tipus de dades simple o compost).

Retorn d'un conjunt de camps

Un procediment emmagatzemat pot retornar un conjunt de camps i formar una fila. Per a fer això, es pot definir un tipus de dades compost, que tingui la llista de camps que ha de retornar el procediment. En el procediment emmagatzemat cal indicar, a continuació de la clàusula `RETURNS`, el nom del tipus de dades, que s'ha d'haver definit prèviament.

Exemple d'ús de procediments per a retornar un conjunt de camps

Imaginem que tenim la taula `Clients` que hem usat anteriorment. Definim un tipus de dades anomenat `tipus_adr`, que tingui el nom del carrer, el número corresponent, el codi postal i la ciutat d'un client.

```
CREATE TYPE tipus_adr AS (
  carrer varchar(20),
  num_carrer varchar(4),
  codi_postal char(5),
  ciutat varchar(15)
);
```

Finalment, creem un procediment emmagatzemat anomenat `trobar_adr_client`, que, donat un DNI d'un client, en retorna l'adreça completa (nom del carrer, número del carrer, codi postal i ciutat).

```
CREATE FUNCTION trobar_adr_client
      (dni_client clients.dni%type)
RETURNS tipus_adr AS $$
DECLARE
    dades_client tipus_adr;
BEGIN
  SELECT carrer,num_carrer,cp,ciutat
  INTO dades_client
  FROM clients
  WHERE dni=dni_client;

  RETURN dades_client;
END;
$$LANGUAGE plpgsql;
```

Recordeu

La sentència `CREATE TYPE` permet crear tipus de dades compostos. Cal indicar el nom del tipus de dades i, a continuació, la llista de camps que formen el tipus. PostgreSQL no permet emprar la clàusula `%ROWTYPE` ni `%TYPE` en la sentència `CREATE TYPE`.

Retorn d'un conjunt de files

La sentència iterativa `FOR` es pot combinar amb la clàusula `RETURN` per a aconseguir que un procediment retorni un conjunt de files com a resultat de la seva execució. Cal fer dues coses; en primer lloc, indicar que el procediment retorna un conjunt de files, per a la qual cosa s'utilitza la clàusula `SETOF`, i, en segon lloc, estendre la clàusula `RETURN` del cos del procediment amb la clàusula `NEXT`. L'ús d'aquesta darrera clàusula fa que, després de retornar una fila (la fila actual de la sentència `SELECT` associada al `FOR`), el procediment emmagatzemat continuï l'execució. L'exemple següent mostra l'ús de la clàusula `SETOF` i `RETURN NEXT` perquè un procediment emmagatzemat retorni un conjunt de files.

Exemple d'ús de la clàusula `SETOF` i `RETURN NEXT`

Imaginem que tenim la taula de clients de l'exemple del subapartat anterior i un procediment anomenat `trobar_ciutat_client`, que retorna totes les ciutats dels clients que tenen per nom el paràmetre d'entrada.

```
CREATE FUNCTION trobar_ciutat_client
      (nom_client clients.nom%type)
RETURNS SETOF clients.ciutat%type AS $$
DECLARE
  ciutat_client clients.ciutat%type;
BEGIN
FOR ciutat_client IN SELECT ciutat
                     FROM clients
                     WHERE nom=nom_client LOOP
    RETURN NEXT ciutat_client;
  END LOOP;
END;
$$LANGUAGE plpgsql;
```

Com es pot veure en l'exemple anterior, la variable `ciutat_client` emmagatzema en cada iteració del bucle un nom de ciutat. Quan s'executa la sentència `RETURN NEXT ciutat_client`, el procediment retorna el nom de la ciutat i continua l'execució, començant una nova iteració o finalitzant-la si no queden més files per processar.

Finalment, es pot aconseguir que un procediment emmagatzemat retorni un conjunt de files i que cadascuna d'aquestes files estigui formada per un conjunt de camps. Per fer això, combinem les clàusules `SETOF`, `RETURN NEXT` i un tipus de dades definit per l'usuari. Aquest tipus de dades té la llista de camps que formen una fila a retornar pel procediment emmagatzemat.

Exemple d'ús de les clàusules SETOF, RETURN NEXT i CREATE TYPE per a retornar un conjunt de files

Imaginem que tenim la taula de clients de l'exemple anterior i un procediment anomenat clients_ciutat que retorna el dni, el nom i el cognom de tots els clients que viuen a la ciutat que es passa com a paràmetre al procediment.

En primer lloc definim un tipus de dades que tingui l'estructura de la fila que s'ha de retornar per a cada client, és a dir, el seu DNI, el nom i el cognom.

```
CREATE TYPE tipus_dades_client AS (
  dni_client VARCHAR(9),
  nom_client VARCHAR(15),
  cognom1 VARCHAR(15));
```

Finalment, definim un procediment que retorna SETOF tipus_dades_client. Això indica que el procediment retorna un conjunt de files i que cada fila és del tipus tipus_dades_client.

```
CREATE FUNCTION clients_ciutat
      (ciutat_client clients.ciutat%type)
RETURNS SETOF tipus_dades_client AS $$
DECLARE
  dades_clients tipus_dades_client;
BEGIN
FOR dades_clients IN SELECT dni,nom,cognom1
                     FROM clients
                     WHERE ciutat=ciutat_client LOOP
  RETURN NEXT dades_clients;
END LOOP;
RETURN;
END;
$$LANGUAGE plpgsql;
```

2.1.6. Invocació de procediments emmagatzemats

PostgreSQL ofereix diverses maneres d'invocar procediments emmagatzemats. En comentarem un parell. La primera consisteix a invocar el procediment des de la clàusula SELECT, tal com s'indica a continuació.

```
SELECT nom_funcio(parametres) [AS alies];
```

Opcionalment, es pot indicar un nom alternatiu o àlies per al resultat del procediment. Aquesta manera d'invocar procediments emmagatzemats s'utilitza normalment quan el procediment retorna un sol resultat.

Una altra manera d'invocar procediments emmagatzemats consisteix a invocar el procediment dels de la clàusula FROM, tal com es mostra a continuació.

```
SELECT * FROM nom_funcio (parametres) [AS alies];
```

Aquesta segona opció normalment s'utilitza quan el procediment emmagatzemat retorna un conjunt de resultats, tot i que també es pot fer servir per a invocar procediments que retornen un sol resultat.

Per exemple, el procediment emmagatzemat `trobar_ciutat` que teniu a continuació es podria invocar de les dues maneres descrites anteriorment.

```
CREATE FUNCTION trobar_ciutat(dni_client
  clients.dni%type) RETURNS varchar(15) AS $$
DECLARE
  ciutat_client clients.ciutat%type;
BEGIN
  SELECT ciutat into ciutat_client
  FROM clients
  WHERE dni=dni_client;
  RETURN ciutat_client;
END;
$$LANGUAGE plpgsql;

SELECT * FROM trobar_ciutat('45678900H');
o
SELECT trobar_ciutat('45678900H');
```

2.1.7. Gestió d'errors

Durant l'execució d'un procediment emmagatzemat es poden produir errors. D'una manera simplificada, podem distingir dos tipus d'error:

- Errors predefinits pel mateix SGBD; per exemple, el codi d'error `23503`, que en PostgreSQL significa `FOREIGN_KEY_VIOLATION`.
- Errors específics del procediment; és a dir, errors propis de l'univers del discurs que es modela i que, per tant, són responsabilitat del creador del procediment emmagatzemat.

De les moltes actuacions alternatives en cas de produir-se un error en un procediment emmagatzemat, n'esmentem dues:

- No capturar l'error dins del procediment emmagatzemat i deixar que el procediment en cancel·li l'execució d'una manera immediata. L'error es reporta al nivell superior[1]; és a dir, al nivell que havia invocat l'execució del procediment emmagatzemat, que es responsabilitza de gestionar-lo. Aquesta opció requereix que el nivell superior conegui els objectes de la BD que es manipulen dins del procediment i les sentències que s'executen sobre aquests objectes.

- Capturar l'error dins del procediment emmagatzemat i deixar que aquest es responsabilitzi de gestionar-lo en primera instància. De les opcions que s'ofereixen amb aquesta actuació, en comentem una: el procediment cancel·la l'execució i l'error és reportat en forma d'excepció al nivell superior (altrament, aquest no se n'assabenta) després d'haver-lo gestionat dins del mateix procediment emmagatzemat. L'aplicabilitat d'aquesta opció requereix que el nivell superior sigui capaç de capturar i tractar excepcions. Per exemple, podríem tenir una aplicació desenvolupada en Java que executés un procediment emmagatzemat. Si el procediment emmagatzemat gestiona els errors reportant-los a l'aplicació Java mitjançant excepcions,

(1)A una aplicació, a un altre procediment emmagatzemat o a l'eina d'execució de l'SQL de manera interactiva, si es treballa amb SQL interactiu.

aquesta aplicació pot capturar les excepcions i fer el que correspongui (per exemple, es pot cancel·lar la transacció en curs, tancar la connexió a la BD i finalitzar l'aplicació).

Per tal d'afavorir al màxim l'encapsulació dels procediments emmagatzemats, es recomana capturar l'error. A més a més, optarem per fer que el procediment cancel·li l'execució i reporti l'error en forma d'excepció al nivell superior.

El nivell superior ha de saber necessàriament si l'execució del procediment emmagatzemat finalitza o no en una situació d'error; altrament, la BD pot quedar en un estat inconsistent. En cas que es produeixi una situació d'error, en general, el nivell superior cancel·larà la transacció que invoca l'execució del procediment emmagatzemat. De fet, PostgreSQL té aquest comportament i, sempre que un procediment falla, cancel·la la transacció en curs.

Per permetre el tractament dels errors que es produeixen durant l'execució d'un procediment emmagatzemat, PL/pgSQL proporciona les sentències `EXCEPTION` (per a capturar-los) i `RAISE EXCEPTION` (per a generar-los), que explicarem a continuació.

La sentència EXCEPTION

La sentència `EXCEPTION` permet especificar les accions que cal executar en cas que es produeixin errors. Aquesta sentència s'ha d'especificar, dins del procediment emmagatzemat, al final del bloc de sentències, just abans de l'`END` que marca el final del procediment. Així, l'estructura d'un procediment emmagatzemat amb la sentència `EXCEPTION` és la següent.

```
CREATE FUNCTION …
BEGIN
  <bloc_de_sentencies>
EXCEPTION
    WHEN <condicio> [OR <condicio> …] THEN
          <bloc_de_sentències>
    [WHEN <condicio> [OR <condicio> …] THEN
          <bloc_de_sentencies>
END;
$$LANGUAGE plpgsql;
```

La sentència `EXCEPTION` consta d'una o més clàusules `WHEN`. La clàusula `WHEN` permet especificar el conjunt concret d'errors d'interès que ha de tractar el procediment. En general, cadascuna de les condicions de la clàusula `WHEN` s'especifiquen utilitzant constants que PostgreSQL té definides per a identificar cada tipus d'error. Per exemple, la constant `FOREIGN_KEY_VIOLATION` es fa servir per a identificar un error de violació de la restricció d'integritat referencial.

Si es produeix un error que no pertany a la llista d'errors tractats amb les clàusules `WHEN`, l'error es reporta al nivell superior i l'execució del procediment emmagatzemat es cancel·la d'una manera immediata.

El `<bloc_de_sentencies>`, que s'executa quan es compleix una condició d'error, representa el conjunt de sentències que cal dur a terme quan es produeix un error determinat.

En el `<bloc_de_sentencies>` s'han d'incloure totes les sentències de tractament dels errors. Cal destacar que, si es produeix un nou error quan es duen a terme aquestes sentències, l'execució del procediment es cancel·larà i l'error serà reportat d'una manera immediata al nivell superior.

La sentència RAISE EXCEPTION

La sentència `RAISE EXCEPTION` serveix perquè el programador pugui generar els seus propis errors dins d'un procediment. Té la sintaxi general següent.

```
RAISE EXCEPTION 'missatge d'error';
```

El missatge d'error és una cadena de caràcters que podem utilitzar per a explicar els motius pels quals s'ha produït un error determinat. A l'hora de generar i tractar els missatges d'error, PostgreSQL té dues variables especialment útils:

Sentència RAISE NOTICE

La sentència `RAISE NOTICE` ('missatge') s'utilitza per a generar o escriure missatges per pantalla. Aquesta sentència es pot fer servir per a consultar el valor de variables concretes del procediment en temps d'execució.

- `SQLSTATE`. La variable `SQLSTATE` conté un codi de cinc dígits associat a l'error produït. PostgreSQL segueix així la recomanació de l'SQL estàndard, que recomana utilitzar aquest codi per a consultar els errors produïts, en comptes de consultar els missatges textuals.

- `SQLERRM`. La variable `SQLERRM` conté un missatge explicatiu associat a l'error produït.

Exemple de gestió d'errors

Volem modificar el codi del procediment `calcul_descompte_client` de manera que controli els errors específics següents:

- El client no existeix.
- El client no té comandes.

Segons el que hem explicat en aquest subapartat, hi ha dues maneres possibles de definir aquest procediment:

1) No capturar l'error dins del procediment emmagatzemat.

WHEN i el codi SQLSTATE

En la clàusula `WHEN` també es pot emprar directament el codi `SQLSTATE`, en comptes de les constants predefinides per PostgreSQL. Podeu consultar el manual per obtenir-ne més informació.

```
CREATE FUNCTION calcul_descompte_client
        (dni_client clients.dni%type)
RETURNS integer AS $$
DECLARE
  descompte INTEGER;
  num_comandes_client INTEGER;
BEGIN
  IF ((SELECT COUNT(*)
```

```
    FROM clients WHERE dni=dni_client)=1)THEN
    num_comandes_client=(SELECT num_com
          FROM clients
          WHERE dni=dni_client);
    IF (num_comandes_client IS NULL) THEN
      RAISE EXCEPTION
      'El client % no te comandes',dni_client;
    ELSIF (num_comandes_client<5) THEN
      descompte=0;
    ELSIF (num_comandes_client<10) THEN
      descompte=3;
    ELSIF (num_comandes_client<15) THEN
      descompte=5;
    ELSE
    descompte=10;
    END IF;
  ELSE
    RAISE EXCEPTION
        'El client % no existeix',dni_client;
  END IF;
  RETURN descompte;
END;
$$LANGUAGE plpgsql;
```

Nota

En la cadena 'El client % no existeix', el símbol % és substituït en temps d'execució pel valor de la variable dni_client.

En aquesta alternativa, tots els errors es reporten al nivell superior, i es cancel·la l'execució del procediment i de la transacció en curs. Per exemple, si executem aquest procediment des de l'editor de l'SQL de `pgAdmin`, en cas que es produeixi un error específic del procediment, apareixerà a la part inferior de la pantalla una finestra que indicarà que s'ha produït l'error amb `SQLSTATE` P0001 i el seu missatge associat (el client no té comandes o el client no existeix).

Pantalla d'error en PostgreSQL

```
ERROR:  El client 45678000 no existeix

********** Error **********

ERROR:  El client 45678000 no existeix
SQL state: P0001
```

Fixeu-vos que, en aquest exemple, el procediment emmagatzemat no té la sentència `EXCEPTION`, perquè no gestiona els errors específics del procediment.

2) Capturar l'error dins del procediment emmagatzemat i reportar-lo al nivell superior mitjançant excepcions.

```
CREATE FUNCTION calcul_descompte_client
        (dni_client clients.dni%type)
RETURNS integer AS $$
DECLARE
  descompte INTEGER;
  num_comandes_client INTEGER;
BEGIN
  IF ((SELECT COUNT(*)
    FROM clients WHERE dni=dni_client)=1)THEN
    num_comandes_client=(SELECT num_com
          FROM clients
          WHERE dni=dni_client);
  IF (num_comandes_client IS NULL) THEN
    RAISE EXCEPTION
    'El client % no té comandes',dni_client;
  ELSIF (num_comandes_client<5) THEN
    descompte=0;
  ELSIF (num_comandes_client<10) THEN
    descompte=3;
  ELSIF (num_comandes_client<15) THEN
```

```
      descompte=5;
    ELSE
    descompte=10;
    END IF;
  ELSE
    RAISE EXCEPTION
        'El client % no existeix',dni_client;
    END IF;
    RETURN descompte;

    EXCEPTION
      WHEN raise_exception THEN
        RAISE EXCEPTION' %: %',SQLSTATE, SQLERRM;
      WHEN others THEN
        RAISE EXCEPTION' P0001: Error intern';
  END;
  $$LANGUAGE plpgsql;
```

En aquest exemple, el procediment tracta els errors en la sentència `EXCEPTION`, que es troba al final del procediment. En el nostre exemple hi ha dos casos d'error que cal tractar:

- El primer cas (`raise_exception`) captura els errors específics del procediment, que són dos: `el client no té comandes` o `el client no existeix`.
- El segon cas (`others`) captura qualsevol altre error que no sigui l'anterior i l'error de PostgreSQL anomenat `query_cancelled`.

Les dues paraules reservades que hi ha a la clàusula `WHEN` (`raise_exception` i `others`) són constants predefinides per PostgreSQL, que tenen associat un codi `SQLSTATE` concret. El nom d'aquestes paraules reservades es pot escriure en lletres majúscules o minúscules.

En aquest darrer exemple, en cas d'error, el procediment captura l'error i, després d'haver-lo tractat, el reporta mitjançant una excepció al nivell superior. Fixeu-vos que el tractament de l'error consisteix a amagar els errors predefinits de PostgreSQL, com ara `foreign_key_violation` o `not_null_violation`. Ens interessa que el procediment amagui tots aquests errors i mostri només un missatge genèric, que en el nostre exemple és '`error intern`'.

Nota

Si no interessa amagar els errors predefinits de PostgreSQL, es pot treballar amb l'opció 1), és a dir, sense capturar errors dins del procediment emmagatzemat.

Com a últim exemple, suposem que volem fer un procediment que esborri un ítem de la taula d'ítems. El número de l'ítem que s'ha d'esborrar es passa com a paràmetre al procediment. En cas que l'ítem es pugui esborrar, el procediment no retornarà res. En cas que es produeixi algun dels errors següents, el procediment haurà d'informar-ne generant excepcions.

1) L'ítem no existeix.

2) L'ítem té comandes.

```
CREATE REPLACE FUNCTION eliminar_item (item integer)
RETURNS void AS $$
DECLARE
   missatge varchar (50);
BEGIN
   DELETE FROM items WHERE num_item = item;
   IF NOT FOUND THEN
      RAISE EXCEPTION
       'L'item no existeix', missatge;
   END IF;
EXCEPTION
   WHEN raise_exception THEN
      RAISE EXCEPTION '%', SQLERRM;
   WHEN foreign_key_violation THEN
      RAISE EXCEPTION 'L'item te comandes;
END;
$$LANGUAGE plpgsql;
```

L'exemple anterior tracta dos errors:

- Un error específic del procediment (cas `WHEN raise_exception`). En aquest cas s'utilitza la variable especial de PostgreSQL `FOUND` per a saber si la sentència `delete` ha esborrat alguna fila.

- Un error predefinit de PostgreSQL (cas `WHEN foreign_key_violation`) que és encapsulat pel procediment per a poder mostrar a l'usuari un missatge d'error personalitzat, en lloc del missatge d'error predefinit de PostgreSQL.

Fixeu-vos que en aquest exemple (si no posem el cas `WHEN others` a la sentència `EXCEPTION`), si el procediment falla per algun error que no sigui algun dels dos anteriors, l'usuari rebrà un missatge d'error predefinit de PostgreSQL que l'informa de l'error produït.

Variable `FOUND`

La variable `FOUND` és una variable especial de tipus booleà de PostgreSQL. El seu valor inicial és fals, però aquest valor pot canviar quan s'executen sentències SQL. Concretament, quan s'executen sentències d'actualització, la variable `FOUND` té valor cert, si com a mínim una fila es veu afectada per la sentència d'actualització; altrament, el valor és fals. Podeu consultar el manual de PostgreSQL per obtenir més informació.

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.

Resum

En aquest mòdul didàctic hem completat l'estudi dels diversos components lògics d'una BD: els procediments emmagatzemats i els disparadors. Per a cadascun d'aquests components, n'hem detallat la definició utilitzant un SGBD concret, PostgreSQL.

Els components lògics (taules, vistes, procediments emmagatzemats, disparadors, etc.) s'organitzen dins d'esquemes. L'esquema és un component definit en l'SQL estàndard, que forma part de l'entorn SQL. La resta de components de l'entorn SQL definits en l'SQL estàndard són el catàleg i el servidor.

En aquest mòdul didàctic també hem comparat les diferències que hi ha entre els components de l'entorn SQL definits en l'SQL estàndard i en un SGBD concret, PostgreSQL.

Finalment, hem estudiat com es gestionen en l'SQL estàndard les connexions i sessions a un servidor de BD. Dins d'una sessió hem estudiat les sentències que permeten utilitzar les transaccions.

Activitats

1. Per a cadascun dels elements que hem presentat en aquest mòdul, consulteu els manuals de PostgreSQL a fi d'ampliar-ne la informació.

2. Proveu en PostgreSQL tots els exemples que es proposen en el mòdul i també els exercicis d'autoavaluació.

Exercicis d'autoavaluació

1. Determineu si les afirmacions següents són vertaderes o falses i argumenteu la resposta breument.

a) Segons l'SQL estàndard, una BD es crea amb la sentència `CREATE DATABASE nom_bd`.
b) L'esquema d'informació conté informació sobre les taules i altres components lògics dels esquemes dels usuaris.
c) La sentència `SET SEARCH_PATH` de PostgreSQL serveix per a definir l'esquema de treball on es crearan els objectes de l'usuari.
d) Segons l'SQL estàndard, quan s'inicia una sessió sempre cal explicitar l'inici d'una transacció amb la sentència `SET TRANSACTION <mode_transacció>`.

2. A partir d'aquestes taules:

```
CREATE TABLE socis (
  nsoci char(10) primary key,
  sexe char(1) not null,
    check (sexe='D' or sexe='H'));

CREATE TABLE clubs (
  nclub char(20) primary key);

CREATE TABLE socis_clubs (
  nsoci char(10) not null references socis,
  nclub char(20) not null references clubs,
  primary key(nsoci, nclub));

CREATE TABLE clubs_amb_mes_de_5_socis (
  nclub char(20) primary key references clubs);
```

i de les restriccions d'integritat següents:

```
RI1: Un club no pot tenir més de vint socis.
RI2: Un club ha de tenir més dones que no pas homes.
```

implementeu un procediment emmagatzemat anomenat `AssignarIndividual` que, donat un soci i un club, insereixi l'assignació a la taula `socis_clubs`. A més, si el club passa a tenir més de cinc socis l'ha de donar d'alta a `clubs_amb_mes_de_cinc_socis`. El procediment ha d'informar dels errors per mitjà d'excepcions, i proporcionar els missatges d'error següents:

```
'Soci ja assignat a aquest club'
'El soci o el club no existeixen'
'El club té més de vint socis'
'El club té menys dones que homes'
'Error intern'
```

3. A partir de les taules creades amb les sentències següents, definiu un disparador que implementi la regla de negoci: "quan la modificació de l'estoc d'un producte el deixi per sota del punt de comanda, cal inserir-hi una petició pendent, si no s'havia fet prèviament".

```
CREATE TABLE Productes(
  nProd INTEGER,
  estoc INTEGER NOT NULL,
  puntComanda INTEGER NOT NULL,
  qtt_a_demanar INTEGER NOT NULL,
  PRIMARY KEY(nProd));

CREATE TABLE Peticions_Pendents(
  nProd INTEGER,
  qtt INTEGER NOT NULL,
  data DATE,
  PRIMARY KEY (nProd));
```

Solucionari

Exercicis d'autoavaluació

1.

a) Fals. La sentència `CREATE DATABASE` no existeix en l'SQL estàndard. En canvi, sí que hi és en molts SGBD comercials, com, per exemple, PostgreSQL.

b) Vertader. L'esquema d'informació és format per un conjunt de vistes que contenen metainformació sobre els components lògics dels esquemes dels usuaris.

c) Vertader. A més, aquesta sentència defineix els esquemes on es buscaran els objectes que intervenen en les sentències SQL.

d) Fals. Normalment, quan s'inicia una sessió es comença una transacció d'una manera automàtica.

2.

```
CREATE OR REPLACE FUNCTION assignar_individual
    (soci socis.nsoci%type, club clubs.nclub%type) RETURNS void AS $$
DECLARE
  dones integer default 0;
  homes integer default 0;
  sexe_soci char(1);
BEGIN
  homes= (SELECT COUNT(*)
      FROM socis s, socis_clubs c
      WHERE s.nsoci=c.nsoci AND s.sexe='H'
      AND c.nclub=club);
  dones= (SELECT COUNT(*)
      FROM socis s, socis_clubs c
      WHERE s.nsoci=c.nsoci AND s.sexe='D'
      AND c.nclub=club);
  IF (homes + dones + 1 ) > 20 THEN
RAISE EXCEPTION 'Un club no pot tenir mes 20 socis';
  ELSE
    Sexe_soci =
    (SELECT sexe FROM socis WHERE nsoci=soci);
    IF sexe_soci ='H' AND (homes >= dones) THEN
      RAISE EXCEPTION
      'Un club ha de tenir mes dones que homes ';
  END IF;
END IF;
INSERT into socis_clubs VALUES(soci,club);
IF (homes+dones+1>5) and
            (not exists
            (SELECT *
            FROM clubs_amb_mes_de_5_socis
            WHERE nclub=club)) THEN
INSERT into clubs_amb_mes_de_5_socis VALUES(club);
END IF;
EXCEPTION
  WHEN raise_exception THEN
    RAISE EXCEPTION '%', SQLERRM;
  WHEN unique_violation THEN
    RAISE EXCEPTION 'Soci ja assignat a aquest club';
  WHEN foreign_key_violation THEN
    RAISE EXCEPTION 'El soci o el club no existeixen';
  WHEN OTHERS THEN
    RAISE EXCEPTION 'Error intern';
END;
$$LANGUAGE plpgsql;
```

Fixeu-vos que el procediment emmagatzemat `assignar_individual` no retorna cap resultat. Si el procediment acaba amb èxit s'insereix una fila a la taula `socis_clubs` i, si cal, el club a la taula `clubs_amb_mes_de_5_socis`. Si el procediment acaba amb error es genera una excepció.

El tractament d'excepcions inclou els casos `unique_violation` i `foreign_key_violation`. El primer cas s'utilitza quan el soci que es passa com a paràmetre al procediment ja ha estat assignat al club que també es passa com a paràmetre; és a dir, quan es viola la clau primària de la taula `socis_clubs`. El segon cas s'utilitza quan el soci o el club

que es volen inserir a la taula socis_clubs no existeixen; és a dir, quan es viola la restricció d'integritat referencial amb la taula socis o clubs.

Una altra manera de comprovar aquests dos errors seria fer consultes al procediment emmagatzemat, com ara les següents:

```
IF (SELECT count(*) FROM socis_clubs
   WHERE nsoci=soci AND nclub=club)>0) THEN
   RAISE EXCEPTION 'Soci ja assignat a aquest club'

IF (SELECT count(*) FROM socis WHERE nsoci=soci)>0)
THEN
RAISE EXCEPTION 'El soci o el club no existeixen'

IF (SELECT count(*) FROM clubs WHERE nclub=club)>0)
THEN
RAISE EXCEPTION 'El soci o el club no existeixen'
```

Aquestes consultes són més ineficients que la solució que proposem, atès que es tracta de consultes addicionals que s'han de realitzar contra la BD. Com que aquests dos errors es poden comprovar mirant si es violen o no les restriccions d'integritat de la BD, no cal fer consultes addicionals per a comprovar-les.

3.

```
CREATE or REPLACE FUNCTION posar_peticio()
RETURNS trigger AS $$
BEGIN
    IF (NEW.estoc<NEW.puntComanda) THEN
      IF ((SELECT count(*) FROM peticions_pendents
        WHERE nprod=NEW.nprod)=0) THEN
        INSERT INTO peticions_pendents
    values(NEW.nprod,NEW.qtt_a_demanar,current_date);
    END IF;
    END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER actualitza_estoc
AFTER UPDATE OF estoc ON Productes
FOR EACH ROW EXECUTE PROCEDURE posar_peticio();
```

El disparador actualitza_estoc és un disparador definit AFTER perquè només hem d'inserir-hi una petició pendent quan es produeix una modificació de l'estoc d'algun producte. Si el definíssim BEFORE, podria succeir que la sentència de modificació sobre la taula de productes fallés i nosaltres ja hauríem inserit la petició pendent a la taula peticions_pendents.

El disparador actualitza_estoc es defineix FOR EACH ROW, atès que volem inserir-hi una petició pendent per cada producte del qual es modifica l'estoc (si l'estoc és inferior al punt de comanda). Si el definíssim FOR EACH STATEMENT, només inseriríem una petició pendent a la taula peticions_pendents, independentment que es modifiqués un estoc o cent estocs de productes.

Glossari

catàleg *m* Component de l'entorn SQL que conté un conjunt d'esquemes, un dels quals és l'esquema d'informació que conté tota la informació dels esquemes dels usuaris (noms de taules, columnes, restriccions, definicions de vistes, etc.).

connexió *m* Associació que es crea entre un client i un servidor.

disparador *m* Acció o procediment emmagatzemat que s'executa automàticament quan s'executa una operació d'inserció, d'esborrament o de modificació sobre alguna taula de la base de dades.

esquema *m* Element que agrupa un conjunt de components lògics (taules, vistes, procediments emmagatzemats, etc.).

PL/PgSQL *m* Llenguatge de PostgreSQL equivalent al PSM definit en l'SQL estàndard.
Nota: La sigla *PSM* correspon a la denominació anglesa *persistent stored module*.

procediment emmagatzemat *m* Acció o funció definida per un usuari que proporciona un servei determinat. Un cop creat, es guarda en la base de dades i es tracta com un objecte més d'aquesta.

servidor *m* Element superior de la jerarquia de components de l'entorn SQL que conté un conjunt de catàlegs.

sessió *f* Conjunt de sentències SQL que s'executen mentre hi ha una connexió activa en un servidor.

transacció *f* Conjunt de peticions SQL de lectura o actualització de la base de dades que confirma o cancel·la els canvis que ha dut a terme.

Bibliografia

García-Molina, H.; Ullman, J. D.; Widom, J. (2002). *Database systems: the complete book*. Prentice Hall.

Gulutzan, P.; Pelzer, T. (1999). *SQL-99 Complete, Really. An example-Based Reference Manual of the New Standard*. R&D Books.

PostgreSQL. Manuals accessibles en línia a: http://www.PostgreSQL.org/docs/