El lenguaje SQL II

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

PID_00171670

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico ampliaremos los conocimientos que tenemos del SQL estándar; más concretamente, estudiaremos los procedimientos almacenados y los disparadores (en inglés, *triggers*). Estos dos componentes lógicos, junto con otros componentes también lógicos, como las tablas y las vistas, están organizados en lo que denominamos *esquema de base de datos*. El esquema de base de datos no es el único componente que permite organizar otros; tenemos el catálogo, que contiene un conjunto de esquemas, y el servidor, que dispone de un conjunto de catálogos. El esquema, el catálogo y el servidor forman una jerarquía de componentes que denominaremos *entorno SQL* y que estudiaremos en este módulo didáctico.

Para que un usuario o una aplicación pueda ejecutar sentencias SQL sobre un conjunto de tablas en un servidor, hay que establecer una conexión previa. Las sentencias SQL que se ejecutan mientras hay una conexión abierta forman lo que denominamos *sesión*, y dentro de una sesión, se ejecutan transacciones. Las conexiones, las sesiones y las transacciones también serán objeto de estudio de este módulo didáctico.

Finalmente, hay que tener en cuenta que a menudo hay divergencias entre lo que dice el estándar SQL y las implementaciones de los diversos proveedores de SGBD relacionales. En nuestro caso, estudiaremos la manera en que PostgreSQL implementa los procedimientos almacenados y los disparadores. También analizaremos las diferencias que hay entre la jerarquía de componentes del entorno SQL definida en el SQL estándar y la que ofrece PostgreSQL.

Objetivos

Los materiales didácticos asociados a este módulo pretenden facilitar la consecución de los objetivos siguientes:

1. Saber utilizar las sentencias del SQL estándar para utilizar servidores, catálogos, esquemas, conexiones, sesiones y transacciones.

2. Conocer las diferencias que hay entre la jerarquía de componentes del entorno SQL definida en el SQL estándar y la que ofrece PostgreSQL.

3. Conocer las sentencias que ofrece PostgreSQL para utilizar esquemas y BD.

4. Completar el estudio de los componentes lógicos de una BD, es decir, los procedimientos almacenados y los disparadores.

5. Utilizar las sentencias que ofrece PostgreSQL para definir procedimientos almacenados y disparadores.

1. Entorno SQL

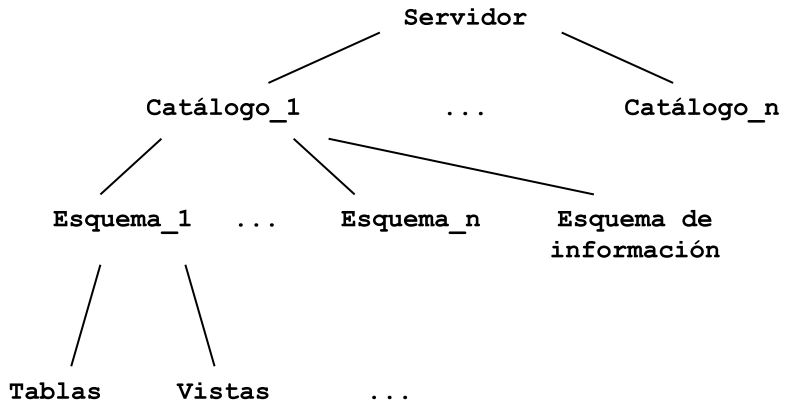
Los componentes lógicos de una BD, como las tablas y las vistas, y también los componentes lógicos que estudiaremos en este módulo didáctico (procedimientos almacenados y disparadores), pertenecen a una BD ubicada en un servidor. Tomemos, por ejemplo, una tabla que guarda datos de empleados. Para poder acceder a los datos de esta tabla, necesitamos conocer, entre otras cosas, el servidor de BD y la BD dentro de este servidor. El servidor de BD y la BD forman parte de lo que denominamos *entorno SQL*. Un entorno SQL es el marco en el que están los datos de una BD y las sentencias SQL para utilizarlos.

Un entorno SQL consta de una estructura jerárquica de componentes en la que cada componente tiene un papel. La figura siguiente muestra la estructura de componentes del entorno SQL definida en el SQL estándar.

Versión SQL

En este módulo didáctico, cuando hablemos del SQL estándar, siempre nos referiremos a la última versión del estándar, ya que tiene como subconjunto todas las anteriores.

Componentes del entorno SQL del SQL estándar



Antes de definir formalmente cada uno de los componentes del entorno SQL, empezaremos ilustrando para qué sirven mediante un ejemplo simple.

Ejemplo

Supongamos que la empresa ABC dispone de la versión 1.0 de una aplicación que accede a cierta BD. La BD se ubica en un servidor. Dentro de este servidor, se ha definido un catálogo para almacenar los datos de la versión 1.0 de la aplicación, y dentro de este catálogo, se han creado tres esquemas: desarrollo, preproducción y producción. A continuación, pasaremos a explicar cada uno de ellos:

- **Desarrollo**: contiene las tablas, las vistas y otros componentes lógicos que los desarrolladores de la aplicación utilizan para hacer pruebas de la misma.
- **Preproducción**: contiene las tablas y otros componentes lógicos de la versión de la aplicación que la empresa pondrá en el entorno de producción el mes próximo.
- **Producción**: contiene las tablas y los componentes lógicos de la versión de la aplicación que está en producción actualmente.

El año próximo, la empresa ABC planea sacar la versión 2.0 de su aplicación. Para empezar a preparar el entorno de la BD, hace un segundo catálogo en el que se crearán los tres esquemas anteriores: desarrollo, preproducción y producción. En cada uno de ellos se vuelven a crear las tablas y otros componentes lógicos utilizados en la versión 1.0 de la aplicación, y finalmente, se copian los datos a las nuevas tablas.

Del ejemplo anterior se puede deducir que el esquema de BD agrupa un conjunto de componentes lógicos: tablas, vistas, etc. El esquema es, por lo tanto, la unidad básica para organizar componentes lógicos. Un catálogo consta de un grupo de esquemas y un servidor (en inglés, *cluster*) es un conjunto de catálogos.

Para que un usuario o una aplicación pueda acceder a un servidor de BD, hay que establecer una conexión previa. Las sentencias SQL que se ejecutan mientras hay una conexión activa a un servidor forman una sesión, y en una sesión se ejecutan transacciones contra la BD.

En los apartados siguientes estudiaremos con más detalle los componentes del entorno SQL y también los conceptos relacionados de *conexión*, *sesión* y *transacción*.

1.1. Esquema, catálogo y servidor

El SQL estándar no dispone de ninguna sentencia de creación de BD; en vez de ésta, utiliza los componentes esquema y catálogo.

El esquema es el elemento que permite agrupar un conjunto de tablas y otros componentes lógicos que son propiedad de un usuario.

La instrucción `CREATE DATABASE`

Muchos de los sistemas relacionales comerciales (como es el caso de PostgreSQL, DB2, SQL Server y otros) han incorporado sentencias de creación de BD con la sintaxis siguiente: `CREATE DATABASE <nombre_BD>`. Una vez creada la BD, normalmente se ha de utilizar una sentencia para acceder a ella. Podéis consultar el manual de PostgreSQL para ver la sintaxis de esta sentencia.

La sentencia `CREATE SCHEMA` del SQL estándar permite crear un esquema y tiene la sintaxis siguiente:

```
CREATE SCHEMA [<nombre_catalogo>.]<nombre_esquema> |
              AUTHORIZATION <ident_usuario> |
              [<nombre_catalogo>.]<nombre_esquema>
              AUTHORIZATION <ident_usuario>
              [<lista_elementos_esquema>];
```

La nomenclatura utilizada en esta sentencia y de ahora en adelante es la siguiente:

- Las palabras en negrita son palabras reservadas del lenguaje.
- La notación `[...]` quiere decir que lo que hay entre los corchetes se podría poner o no.
- La notación `A|...|B` quiere decir que hemos de escoger entre todas las opciones, pero que tenemos que poner una de ellas obligatoriamente.

La sentencia de creación de esquemas permite que un conjunto de tablas y otros componentes lógicos (designados <lista_elementos_esquema>) se puedan agrupar bajo un mismo esquema (<nombre_esquema>) y que tengan un propietario (<ident_usuario>). Opcionalmente, se puede indicar el nombre del catálogo al que pertenece el esquema. Aunque muchos parámetros de la sentencia CREATE SCHEMA son opcionales, como mínimo hay que indicar el nombre del usuario propietario del esquema o el nombre del esquema.

Ejemplo de creación de esquemas

```
CREATE SCHEMA empresa AUTHORIZATION 'pedro';
  CREATE TABLE departamentos (
    codigo_dept integer primary key,
    nombre_dept varchar(30) not null);
CREATE TABLE empleados(
    codigo_empl integer primary key,
    nombre_empl varchar(50) not null,
    codigo_dept integer references departaments);
```

En este ejemplo se crea el esquema llamado empresa, que pertenece al usuario pedro. Dentro del esquema empresa, se crean dos tablas: departamentos y empleados.

No hace falta crear el esquema y todos los elementos correspondientes al mismo tiempo. Primero se puede crear el esquema; a continuación, definirlo como esquema de trabajo, y después añadir elementos. Para cambiar de esquema, el SQL estándar dispone de la sentencia SET SCHEMA con la sintaxis siguiente:

```
SET SCHEMA <nombre_esquema>;
```

Ejemplo de cambio de esquema

```
CREATE SCHEMA empresa AUTHORIZATION 'pere';
SET SCHEMA empresa;
CREATE TABLE departamentos (
    codigo_dept integer primary key,
    nombre_dept varchar(30) not null);
CREATE TABLE empleados(
    codigo_empl integer primary key,
    nombre_empl varchar(50) not null,
    codigo_dept integer references departaments);
```

En el ejemplo anterior, se crea el esquema empresa y, a continuación, se define como esquema de trabajo. Una vez hecho esto, las sentencias de creación de las tablas departamentos y empleados se ejecutan en el esquema de trabajo, es decir, en empresa.

La sentencia para borrar un esquema es DROP SCHEMA y tiene la sintaxis siguiente:

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

La opción `RESTRICT` permite borrar el esquema sólo si éste está completamente vacío. En cambio, la opción `CASCADE` permite borrarlo aunque contenga elementos. En este último caso, cuando se borra el esquema, también se borran todos sus elementos.

Las tablas y otros componentes lógicos se crean y se utilizan dentro de un esquema. De una manera análoga, los esquemas se crean, se modifican y se borran dentro de un catálogo. Por lo tanto, un catálogo es un componente que contiene un conjunto de esquemas.

Cada catálogo contiene, además de los esquemas definidos por los usuarios, un esquema de información, denominado *information schema*, que contiene toda la información sobre los esquemas que definen los usuarios, es decir, los nombres y los atributos de las tablas, las restricciones de integridad definidas en las mismas, la definición de las vistas, etc. Así pues, el esquema de información contiene metainformación (información relativa a los componentes lógicos definidos en cada esquema de usuario).

Para consultar la metainformación, el esquema de información consta de un conjunto de vistas definidas a partir de un conjunto de tablas de sistema, a las que sólo puede acceder el administrador de la BD.

Entre las vistas del esquema de información encontramos las siguientes:

- `SCHEMATA`: contiene información sobre cada esquema del catálogo.
- `DOMAINS`: contiene información sobre los dominios definidos en los esquemas del catálogo.
- `TABLES`: contiene información sobre las tablas definidas en los esquemas del catálogo.
- `VIEWS`: contiene información sobre las vistas definidas en los esquemas del catálogo.

Las vistas del esquema de información permiten a los usuarios consultar (pero no modificar de una manera directa) información relativa a los objetos de la BD sobre los que tienen privilegios.

Ejemplo

Por ejemplo, supongamos que el usuario `pedro` tiene un esquema en el que ha creado dos tablas: `empleados` y `departamentos`. Cuando `pedro` consulte la vista `TABLES` del esquema de información, obtendrá dos entradas con información, una por cada una de las dos tablas que tiene en el esquema. Cuando `pedro` cree una tabla nueva en su esquema, por ejemplo la tabla `proyectos`, y posteriormente vuelva a consultar la vista `TABLES` del esquema de información, le aparecerán tres entradas correspondientes a las tres tablas que ha creado en su esquema: `empleados`, `departamentos` y `proyectos`.

Del ejemplo anterior se desprende que la actualización de las vistas del esquema de información no la llevan a cabo los usuarios de una manera directa, sino que se produce cuando éstos hacen actualizaciones sobre los objetos de la BD.

Finalmente, hay que mencionar que cada usuario tiene acceso de consulta sólo a la información del esquema de información a la que está autorizado a acceder (por ejemplo, información sobre las tablas que ha creado); es decir, un usuario no podrá acceder, consultando las vistas del esquema de información, a información de objetos sobre los que no tiene privilegios.

Antes, hemos estudiado las sentencias del SQL estándar para crear y borrar esquemas del SQL estándar. Con respecto a los catálogos, el SQL estándar no dispone de sentencias para crearlos ni borrarlos. Estas sentencias son específicas de cada SGBD.

Finalmente, un conjunto de catálogos pertenece a un servidor. El SQL estándar tampoco dispone de sentencias para crear y borrar servidores. Estas sentencias son específicas de cada SGBD. Sin embargo, el SQL estándar proporciona las sentencias `CONNECT` y `DISCONNECT` para conectarse y desconectarse de un servidor.

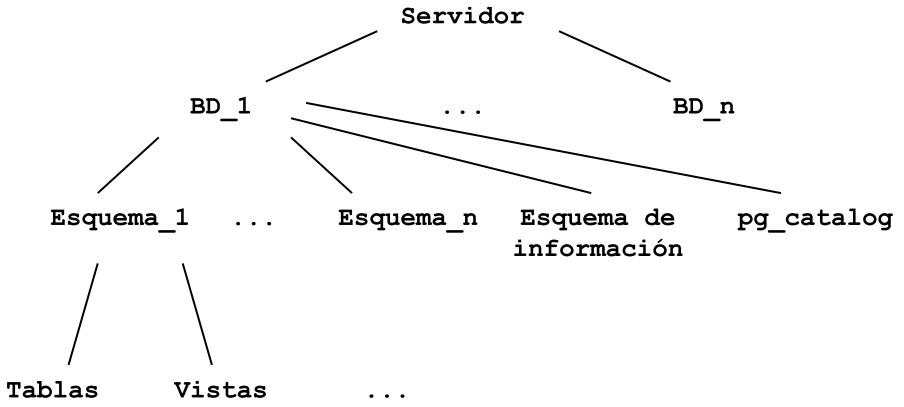
Ved también

Las sentencias `CONNECT` y `DISCONNECT` se estudian en subapartado 1.2.

Una vez estudiados los elementos del entorno SQL que ofrece el SQL estándar, veamos cuáles de estos componentes tiene PostgreSQL.

En PostgreSQL, un servidor contiene un conjunto de BD. En una conexión a un servidor, un usuario accede sólo a una BD que se especifica en el momento de establecer la conexión. Por su parte, una BD contiene un conjunto de esquemas. La figura siguiente muestra esta jerarquía de componentes:

Componentes del entorno SQL de PostgreSQL



Para crear una BD, PostgreSQL ofrece la sentencia `CREATE DATABASE`, que tiene la siguiente sintaxis simplificada:

```
CREATE DATABASE <nombre_bd>;
```

La sentencia `DROP DATABASE` permite borrar una BD y tiene la sintaxis siguiente:

```
DROP DATABASE <nombre_bd>;
```

Para manejar esquemas, PostgreSQL ofrece las sentencias `CREATE SCHEMA` y `DROP SCHEMA`. La primera, permite crear un esquema y la segunda, borrarlo. La sintaxis de la sentencia `CREATE SCHEMA` es la siguiente:

```
CREATE SCHEMA <nombre_esquema>;
```

La sentencia `DROP SCHEMA` de PostgreSQL tiene la misma sintaxis que la que tiene en el SQL estándar.

En PostgreSQL, todas las BD tienen un esquema por defecto denominado `public`. Si no se especifica lo contrario, todos los componentes lógicos que crea un usuario, como tablas o vistas, se crean dentro de este esquema.

En la jerarquía de objetos de PostgreSQL, se accede a un componente lógico, como una tabla, especificando el nombre de la BD a la que pertenece y el nombre del esquema dentro de esa BD. Por ejemplo, si tenemos la BD llamada `empresa`, que contiene el esquema `public`, en el que se han creado las tablas `empleados`, `departamentos` y `proyectos`, el nombre completo para acceder a la tabla `empleados` será `empresa.public.empleados`.

No obstante, PostgreSQL permite acceder a un objeto sin tener que utilizar su nombre completo (`nombre_bd.nombre_esquema.nombre_objeto`). Ello es posible porque PostgreSQL permite definir una lista de esquemas en la que se pueden buscar los objetos de la BD en una sesión de usuario. Esta lista de esquemas se define en la variable de sistema `search_path`. La sentencia `SET SEARCH_PATH <lista_esquemas>` permite definir los esquemas de la variable `search_path`.

Ejemplo de definición de search_path

La sentencia siguiente indica que los nombres de los objetos se buscarán en los esquemas llamados `mi_esquema` y, si no están ahí, en el esquema `public`. Además, el primer esquema especificado en el `search_path` es el esquema en el que se crearán los componentes lógicos que defina el usuario.

```
SET SEARCH_PATH TO mi_esquema, public;
```

Ejemplo de creación de tabla dentro de un esquema

La sentencia `SET SEARCH_PATH` define el esquema de trabajo durante la sesión del usuario. Por lo tanto, la tabla `departamentos` se creará en el esquema `empresa`.

```
SET SEARCH_PATH TO empresa;
CREATE TABLE Departamentos (
  codigo_dpt integer primary key,
  nombre_dpt varchar(30) not null,
  ciudad_dpt varchar(30),
  presupuesto float not null);
```

Cada BD de PostgreSQL tiene un esquema adicional: el esquema de información *(information schema)*. Como el esquema de información es definido por el SQL estándar, es portable y estable; por lo tanto, no contiene características o funcionalidades específicas de PostgreSQL.

Recordad

El esquema de información consta de un conjunto de vistas que contienen metainformación sobre la BD (por ejemplo, información relativa a las tablas, restricciones de integridad definidas en las mismas, etc.).

Además del esquema de información, PostgreSQL tiene un esquema específico de sistema, denominado `pg_catalog`, que contiene toda la información de los esquemas definidos por los usuarios dentro de la BD, es decir, los nombres y los atributos de las tablas, las restricciones de integridad, etc. El `pg_catalog` de PostgreSQL está formado por un conjunto de tablas como, por ejemplo, las siguientes:

- `pg_class`: almacena información sobre las tablas, los índices, las secuencias, las vistas y otros componentes lógicos definidos en la BD.
- `pg_attribute`: almacena información sobre las columnas de las tablas de la BD.
- `pg_triggers`: almacena información sobre los disparadores de la BD.
- `pg_proc`: almacena información sobre los procedimientos almacenados definidos en la BD.

Finalmente, PostgreSQL proporciona un conjunto de vistas definidas sobre las tablas de `pg_catalog` que permiten consultar los datos almacenados en el `pg_catalog`. La información que suministran, es la misma que la facilitada por las vistas del esquema de información. Como el esquema de información es estable, porque está definido en el SQL estándar, es recomendable consultar las vistas del esquema de información en vez de las vistas específicas de PostgreSQL.

Ved también

En los apartados 2 y 3 de este módulo, se estudian los componentes lógicos llamados *procedimientos almacenados* y *disparadores*.

1.2. Conexión, sesión y transacción

Para acceder a un servidor de BD, hay que establecer una conexión previamente.

Una conexión se puede definir como la asociación que se crea entre un cliente y un servidor cuando el cliente manifiesta que tiene la intención de trabajar con la BD solicitando una conexión.

El SQL estándar proporciona las sentencias siguientes para manejar conexiones: CONNECT y DISCONNECT.

La sentencia CONNECT permite establecer una conexión con un servidor. Tiene la sintaxis siguiente:

```
CONNECT TO <nombre_servidor> [AS <nombre_conexion>]
[USER <ident_usuario>];
```

Ejemplo de utilización de la sentencia CONNECT

La sentencia siguiente permite que el usuario pedro establezca una conexión llamada conexión1 al servidor llamado algún_servidor.

```
CONNECT TO 'algun_servidor' AS 'conexion1' USER 'pedro';
```

Un usuario puede establecer más de una conexión con un servidor. Cuando el usuario se conecta por segunda vez con el servidor, la última conexión se convierte en la conexión actual (en inglés, *current*). La primera conexión que se ha establecido queda dormida y se mantienen sus características y su información de contexto por medio del SGBD para poder restaurarla más adelante. Si el segundo intento de conexión falla, la primera conexión se mantiene como conexión actual.

Después de establecer la conexión con el servidor, se puede acceder a un esquema concreto con la sentencia siguiente:

```
SET SCHEMA <nombre_esquema>;
```

Para cerrar una conexión, el SQL estándar dispone de la sentencia DISCONNECT, que tiene la sintaxis siguiente:

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

Con la sentencia DISCONNECT se puede cerrar la conexión que tiene <nombre_conexión>, la conexión actual o todas las conexiones que el usuario tenía abiertas.

Las sentencias SQL que se ejecutan mientras hay una conexión activa con un servidor forman una sesión. Por lo tanto, una sesión es el contexto en el que un usuario o una aplicación ejecuta una secuencia de sentencias SQL mediante una conexión.

Cada sesión tiene un catálogo y un esquema de trabajo, que se pueden definir con las sentencias `SET CATALOG` y `SET SCHEMA` respectivamente. Adicionalmente, las características de las transacciones que se ejecutan en una sesión se pueden definir con la sentencia `SET SESSION CHARACTERISTICS`, que tiene la sintaxis siguiente:

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

en el que <modo_transaccion> puede ser:
modo_de acceso | ISOLATION LEVEL <nivel de aislamiento>
en el que modo_de acceso puede ser: READ ONLY | READ WRITE
en el que nivel de aislamiento puede ser: READ UNCOMMITTED |
READ COMMITTED | REPETEABLE READ | SERIALIZABLE
```

Una transacción es un conjunto de sentencias SQL de lectura (consultas) y actualización de la BD que confirma o cancela los cambios que se han llevado a cabo. Por lo tanto, se trata de una unidad indivisible de trabajo, de manera que, o bien todo el conjunto de sentencias SQL se ejecuta completamente, si es el caso de una serie de cambios en la BD, o bien no se ejecuta ninguna sentencia.

Volviendo a la sentencia `SET SESSION CHARACTERISTICS`, el modo de acceso permite especificar si la transacción sólo leerá datos de la BD (modo de acceso `READ ONLY`) o si, al contrario, además de leer, también modificará su contenido (modo de acceso `READ WRITE`). El nivel de aislamiento indica la independencia con la que trabajan las transacciones de la BD entre ellas. Si no se indica nada, el SGBD garantizará el aislamiento total de la transacción (nivel de aislamiento `SERIALIZABLE`).

Con respecto a las transacciones, el SQL estándar ofrece sentencias que permiten a los usuarios delimitar el inicio y la finalización de las transacciones. El inicio de una transacción se puede indicar de una manera explícita con la sentencia `SET TRANSACTION`.

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

La utilización de la sentencia `SET TRANSACTION` no es obligatoria. Si no se indica de una manera explícita el inicio de una transacción, implícitamente el SGBD empezará una con la ejecución de la primera sentencia SQL que se lleve a cabo dentro de la sesión. La transacción permanecerá activa hasta que, de una manera explícita y obligatoria, se indique su finalización.

Para indicar la finalización de una transacción, el SQL estándar nos ofrece la sentencia siguiente:

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

La diferencia entre COMMIT y ROLLBACK es que, mientras que la sentencia COMMIT confirma todos los cambios realizados contra la BD durante la ejecución de la transacción, la sentencia ROLLBACK los deshace y deja la BD como estaba antes de empezar. La palabra reservada WORK sólo sirve para explicar qué hace la sentencia y es opcional.

Ejemplo de establecimiento de las características de una sesión

En este ejemplo, el usuario pedro establece una conexión con el servidor_ABC. En la sesión de trabajo establece las características siguientes: la transacción leerá y escribirá datos de la BD (modo READ WRITE) y el nivel de aislamiento será SERIALIZABLE. Después de ejecutar las sentencias SQL, el usuario pedro acepta los cambios que ha llevado a cabo la transacción y la acaba con la sentencia COMMIT. Finalmente, finaliza la conexión.

```
CONNECT TO 'servidor_ABC' AS 'conexion1' USER 'pedro';
SET SESSION CHARACTERISTICS AS
  READ WRITE
  ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM departamentos;
INSERT INTO departamentos VALUES (...);
COMMIT;
DISCONNECT conexion1;
```

2. Procedimientos almacenados

Un procedimiento almacenado es una acción o función definida por un usuario que proporciona un servicio determinado. Una vez creado, el procedimiento se guarda en la BD y se trata como un objeto más de ésta.

Los procedimientos almacenados se pueden ejecutar:

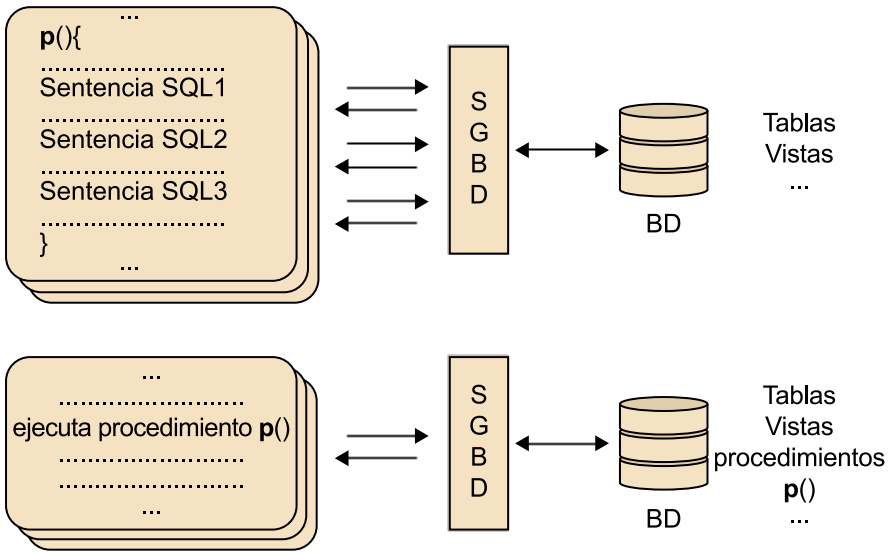
- De una manera directa, utilizando el SQL interactivo.
- Desde una aplicación que acceda a la BD.
- Desde otro procedimiento almacenado.

Los procedimientos almacenados sirven para:

- Simplificar el desarrollo de aplicaciones.
- Mejorar el rendimiento de las BD.
- Encapsular las operaciones que se ejecutan contra una BD.

Para ilustrar las características previas, nos fijaremos en la figura que mostramos a continuación. En ella se presentan dos posibles arquitecturas de desarrollo de aplicaciones que acceden a una BD, una que utiliza los procedimientos almacenados y la otra que no:

SGBD sin procedimientos almacenados y SGBD con procedimientos almacenados



Como se puede ver en la figura anterior, si `p()` es un método que conceptualmente resuelve una operación determinada que necesita ejecutar todo un conjunto de sentencias SQL, podemos trasladar su definición a la BD y transformarlo en un procedimiento almacenado. Además, también pueden utilizar el

procedimiento almacenado otras aplicaciones, con lo que conseguimos simplificar el desarrollo de aplicaciones: transferimos lógica relativa a la BD hacia el SGBD y favorecemos la reutilización de código.

Si pensamos en un entorno distribuido, el tráfico por la red disminuirá en el caso de utilizar procedimientos almacenados. Con el uso de estos procedimientos, se evitará que cada sentencia SQL se envíe de una manera individual por la red y que los resultados de la ejecución de cada sentencia SQL sean recogidos por la aplicación individualmente. Además, los procedimientos almacenados se guardan precompilados en la BD, de manera que su estrategia de ejecución se calcula en el momento de crearlos, lo que no sucede necesariamente cuando nuestra aplicación no utiliza procedimientos almacenados. Por lo tanto, la utilización de procedimientos almacenados mejora el rendimiento de las BD.

Finalmente, el procedimiento almacenado encapsula el conjunto de sentencias SQL que incorpora; el programador que utiliza este procedimiento sólo ha de saber que un procedimiento almacenado concreto le proporciona un servicio determinado y que, en el caso de no poder satisfacerlo, se reportará un error. No necesita conocer las sentencias SQL que incorpora el procedimiento ni los elementos del esquema de la BD que manipulan estas sentencias. Esto ayuda a controlar las operaciones que los usuarios efectúan en la BD.

procedimiento almacenado y control de errores

El procedimiento almacenado no sólo resuelve la operación, sino que también se responsabiliza del control de todas las situaciones de error que se produzcan.

2.1. Sintaxis de los procedimientos almacenados en PostgreSQL

En PostgreSQL, para poder escribir procedimientos almacenados, disponemos de dos tipos de sentencias:

- Sentencias propias del SQL.
- Sentencias propias de algún lenguaje procedimental, como por ejemplo el PL/pgSQL o el PL/Python.

PL

PL es la sigla de *procedural language*, que es la denominación que reciben los lenguajes procedimentales en PostgreSQL.

En este módulo estudiaremos el lenguaje procedimental PL/pgSQL, que es uno de los que vienen con la distribución estándar de PostgreSQL. El PL/pgSQL complementa el SQL, que no es un lenguaje computacionalmente completo. En esencia, el PL/pgSQL proporciona sentencias que permiten controlar el flujo de ejecución de nuestros procedimientos almacenados.

Nota

De momento, y hasta que no lleguemos a la sección de errores, se omite el tratamiento de errores en los ejemplos de procedimientos almacenados.

Ejemplo de un procedimiento almacenado escrito en PL/pgSQL

A partir de la tabla `clientes`,

```
CREATE TABLE clientes (
  dni char(9) primary key,
  nombre varchar(15) not null,
  apellido1 varchar(15) not null,
  apellido2 varchar(15) not null,
  calle varchar(20) not null,
  num_calle varchar(4) not null,
  cp char(5) not null,
  ciudad varchar(15) not null);
```

podemos definir el procedimiento almacenado `encontrar_ciudad`:

```
CREATE FUNCTION encontrar_ciudad(dni_cliente char(9))
RETURNS varchar(15) AS $$
DECLARE
  ciudad_cliente varchar(15);
BEGIN
  SELECT ciudad into ciudad_cliente
  FROM clientes
  WHERE dni=dni_cliente;
  RETURN ciudad_cliente;
END;
$$LANGUAGE plpgsql;
```

Podemos ejecutar el procedimiento anterior con la sentencia siguiente:

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

El ejemplo anterior muestra algunos elementos básicos en la definición de un procedimiento almacenado:

- Presencia de la sentencia `CREATE FUNCTION`, que es la que permite crear procedimientos almacenados en PostgreSQL. La sentencia `CREATE FUNCTION` y la palabra `END` seguida de punto y coma delimitan el procedimiento.

- Nombre del procedimiento; en este caso `encontrar_ciudad`.

- Definición de los parámetros (nombre y tipo de datos) necesarios para invocar la ejecución del procedimiento; en este caso `dni_cliente`, que es de tipo `CHAR(9)`.

- Definición de los tipos de datos del parámetro devuelto por el procedimiento mediante la cláusula `RETURNS`; en este ejemplo `VARCHAR(15)`.

- Definición del cuerpo del procedimiento, que a menudo se delimita con las marcas `$$`.

La marca $$

La marca `$$` se utiliza habitualmente para delimitar el cuerpo del procedimiento almacenado. El cuerpo del procedimiento es una cadena de caracteres y, por lo tanto, tendría que estar delimitado por comillas simples. No obstante, el uso de comillas simples puede ser poco práctico si dentro hay otras comillas. Las comillas interiores se deberían duplicar

para diferenciarlas de las comillas que delimitan el inicio y el final del procedimiento. Como esto puede hacer que el código sea poco legible, a menudo se usa la marca `$$` para delimitar el inicio y el final del cuerpo del procedimiento.

- Definición de las variables del procedimiento; en el ejemplo `ciudad_cliente`. La sección de declaración de variables es precedida por la cláusula `DECLARE`. Si el procedimiento no tiene variables, se puede omitir esta cláusula.

Notación

Si un procedimiento no devuelve nada, sólo hay que poner la palabra `void` después de la cláusula `RETURNS`.

- Presencia de las palabras `BEGIN` y `END` para delimitar el bloque que contiene las sentencias del procedimiento; en este caso, el bloque contiene la sentencia `SELECT` del SQL.

- Parámetro devuelto por el procedimiento precedido por la cláusula `RETURN`; en este caso `ciudad_cliente`.

- Nombre del lenguaje procedimental utilizado para escribir el procedimiento; en este caso `plpgsql`.

Como se puede ver en el ejemplo anterior, una vez que se ha creado un procedimiento, se puede ejecutar mediante la sentencia del `SQL SELECT * FROM nombre_procedimiento (parámetros)`.

Lectura recomendada

La sentencia `SQL SELECT * FROM nombre_procedimiento (parámetros)` no es la única manera de ejecutar procedimientos almacenados en PostgreSQL. Podéis consultar las otras en el manual de PostgreSQL.

Para borrar un procedimiento almacenado, se utiliza la sentencia `DROP FUNCTION`, que tiene la sintaxis siguiente:

```
DROP FUNCTION
nombre_procedimiento(tipo_parametros_entrada);
```

En la sentencia `DROP FUNCTION`, hay que especificar los tipos de los parámetros de entrada al procedimiento, porque en PostgreSQL los procedimientos aceptan sobrecarga (diversos procedimientos con el mismo nombre).

En el ejemplo anterior, el procedimiento devuelve un solo campo. Más adelante explicaremos cómo se pueden devolver un conjunto de campos y un conjunto de filas.

El lenguaje PL/pgSQL proporciona básicamente tres tipos de sentencias:

- Sentencias para definir (`DECLARE`) variables.
- Sentencias para asignar valores a variables (`variable:=expresión`).
- Sentencias para controlar el flujo de ejecución de un procedimiento:
  - Sentencias condicionales: `IF` y `CASE`.
  - Sentencias iterativas: `FOR`, `LOOP` y `WHILE`.
  - Sentencias para gestionar errores: `EXCEPTION` y `RAISE EXCEPTION`.

2.1.1. Parámetros

Un procedimiento almacenado puede recibir un conjunto de parámetros de entrada y devolver un resultado. También puede tener parámetros, tanto de entrada como de salida.

Lectura recomendada

Podéis consultar la sintaxis para usar parámetros de entrada y de salida en el manual de PostgreSQL.

A cada parámetro de entrada le pondremos un nombre y le asociaremos un tipo de datos. Para asociar un tipo de datos a un parámetro de entrada, se pueden utilizar los mismos tipos de datos que en la definición de columnas de tablas. También es posible especificar que el tipo de datos de un parámetro de entrada sea idéntico al tipo de datos de una columna determinada de una tabla mediante la cláusula `%TYPE`.

Para devolver el resultado de la función, basta con indicar el tipo de datos en la cláusula `RETURNS`.

Ejemplo de uso de la cláusula `%TYPE`

Teniendo en cuenta lo que acabamos de explicar, podríamos haber definido el procedimiento almacenado `encontrar_ciudad` de la manera siguiente:

```
CREATE or REPLACE FUNCTION encontrar_ciudad(dni_cliente
  clientes.dni%type) RETURNS varchar(15) AS $$
DECLARE
  ciudad_cliente clientes.ciudad%type;
BEGIN
  SELECT ciudad INTO ciudad_cliente
  FROM clientes
  WHERE dni=dni_cliente;
  RETURN ciudad_cliente;
END;
$$LANGUAGE plpgsql;
```

Para volver a crear el procedimiento almacenado `encontrar_ciudad` hay dos alternativas. La primera consiste en borrar antes la versión antigua del procedimiento mediante la sentencia `DROP FUNCTION encontrar_ciudad (char(9))`, y la segunda, en crear el procedimiento con la sentencia `CREATE or REPLACE FUNCTION`. La cláusula `REPLACE` permite sustituir la versión antigua del procedimiento almacenado en la BD por la nueva.

2.1.2. Variables

Para definir variables en un procedimiento almacenado, utilizaremos la cláusula `DECLARE` de PL/pgSQL, que tiene la sintaxis siguiente:

```
nombre_variable [CONSTANT] tipo_datos [NOT NULL]
[{DEFAULT | :=}expresión];
```

La cláusula `CONSTANT` indica que la variable tendrá un valor constante durante la ejecución del procedimiento; la cláusula `NOT NULL` indica que la variable no puede tener un valor nulo, y finalmente, la cláusula `DEFAULT` indica que

la variable tendrá un valor por defecto, que se podrá modificar más adelante en la ejecución del procedimiento. Si no se especifica un valor por defecto, la variable se inicializa a NULL.

Cada variable ha de tener asociados un nombre y un tipo de datos. De forma análoga al caso de los parámetros de entrada de un procedimiento almacenado, se pueden utilizar los mismos tipos de datos que en la definición de columnas de una tabla, y también es posible indicar que el tipo de datos de una variable concreta es idéntico al tipo de datos de una columna determinada de una tabla específica mediante la cláusula `%TYPE`.

Otra posibilidad cuando se declara una variable, consiste en indicar que el tipo de datos de la variable es idéntico al tipo de datos de una fila de una tabla concreta. Esto se lleva a cabo mediante la cláusula `%ROWTYPE`. Con esta cláusula, la variable definida tiene un tipo compuesto, que consta de tantos campos como tenga la tabla utilizada en la declaración de la variable. Los tipos de los campos de la variable también coinciden con los de las columnas de la tabla a partir de la que se define la variable.

Ejemplo de uso de la cláusula `%ROWTYPE`

En el ejemplo siguiente, la variable `var_cliente` tiene el mismo tipo que una fila de la tabla `clientes`; es decir, la variable `var_cliente` es de tipo compuesto, ya que tiene tantos campos como campos tenga la tabla `clientes`.

```
CREATE FUNCTION…
DECLARE
var_cliente clientes%ROWTYPE;
BEGIN
…
END;
$$ LANGUAGE plpgsql;
```

Para acceder a cada uno de los campos de una variable definida con la cláusula `%ROWTYPE` se utiliza la notación siguiente: `nombre_variable.nombre_campo`. Los nombres de los campos coinciden con los nombres de los campos de la tabla a partir de la que se define la variable.

Si no se inicializan las variables definidas en el procedimiento, por defecto tienen valor nulo. Hay diversas posibilidades para asignar valores a las variables de los procedimientos:

Error de ejecución

Cuando a una variable declarada `NOT NULL` se le asigna un valor nulo durante la ejecución del procedimiento, se produce un error de ejecución.

- Sentencia de asignación de PL/PgSQL (`variable:=expresión`).
- Sentencia `SELECT ... INTO` del SQL.
- Asignación a una variable del resultado de ejecutar otro procedimiento.

Ejemplos de asignaciones

- Sentencia de asignación de PL/pgSQL:

```
Contador:=1;
```

- Sentencia `SELECT ... INTO` del SQL:

```
SELECT ciudad INTO ciudad_cliente
FROM clientes
WHERE dni=dni_cliente;
```

- Asignación a una variable del resultado de ejecutar otro procedimiento:

```
importe_pedido:=importe_un_pedido(num_pedido);
```

o bien

```
SELECT * FROM importe_un_pedido(num_pedido)
INTO importe_pedido;
```

Las variables definidas en un procedimiento almacenado son siempre variables locales; el ámbito de visibilidad de una variable local queda restringido al procedimiento en el que se haya definido.

Nota

`Contador`, `ciudad_cliente` e `importe_pedido` son variables.

Las variables **no se consideran objetos de la BD**. Los valores de las variables locales quedan descartados una vez finalizada la ejecución del procedimiento almacenado en el que han sido definidas.

2.1.3. Sentencias condicionales

La sentencia `IF` de PL/pgSQL sirve para establecer condiciones en el flujo de ejecución de un procedimiento almacenado. Tiene la sintaxis básica siguiente:

Lectura recomendada

PostgreSQL dispone de otra sentencia condicional, la sentencia `CASE`. Podéis consultar su sintaxis en el manual de PostgreSQL.

```
IF <condicion> THEN <bloque_de_sentencias>
ELSE <bloque_de_sentencias>
END IF;
```

Se pueden establecer diversos niveles de imbricación mediante la cláusula `ELSIF` como alternativa a la cláusula `ELSE`. Por ejemplo, para establecer un nivel de imbricación adicional, hay que hacer lo siguiente:

```
IF <condicion> THEN <bloque_de_sentencias>
  ELSIF <condicion> THEN <bloque_de_sentencias>
     ELSE <bloque_de_sentencias>
END IF;
```

Para especificar las condiciones, disponemos de los elementos siguientes:

- Operadores lógicos: `AND`, `OR`, `NOT`.
- Operadores de comparación: >, <, >=, <=, =, <>.

- Predicados propios del SQL: `BETWEEN`, `IN`, `IS NULL`, `IS NOT NULL` o `LIKE`.
- Consultas SQL.

Como hay columnas de tablas que pueden tener valor nulo, existen predicados propios del SQL para comparar con el valor nulo. Cuando se evalúan las expresiones de las condiciones, también hay que tener en cuenta los valores nulos, ya que estas expresiones se pueden evaluar a nulo. Hay que destacar que, si alguna expresión dentro de la condición evalúa a `NULL`, toda la condición no evalúa a cierto, salvo en el caso de que comprobemos de una manera explícita condiciones del tipo `IS NULL` o `IS NOT NULL`.

Ejemplo de uso de sentencias condicionales

Imaginemos que tenemos la siguiente tabla `clientes`, a la cual hemos añadido el campo `num_ped`. Este campo almacena el número de pedidos que ha hecho un cliente concreto.

```
CREATE TABLE clientes(
  dni varchar(9) primary key,
  nombre varchar(15) not null,
  apellido1 varchar(15) not null,
  apellido2 varchar(15) not null,
  calle varchar(20) not null,
  num_calle varchar(4) not null,
  cp char(5) not null,
  ciudad varchar(15) not null,
  num_ped integer);
```

El procedimiento `calculo_descuento_cliente` calcula el descuento que se ha de aplicar a cada cliente pasado como parámetro según el número de pedidos (campo `num_ped` de la tabla `clientes`) que ha hecho: el descuento es del 0% si ha hecho menos de cinco pedidos; del 3% si ha hecho entre cinco y nueve pedidos; del 5% si ha hecho entre diez y catorce pedidos, y del 10% si ha hecho quince o más.

```
CREATE FUNCTION calculo_descuento_cliente
    (dni_cliente clientes.dni%type)
RETURNS integer AS $$
DECLARE
  descuento integer;
  num_ped_cliente integer;
BEGIN
  IF((SELECT COUNT(*) FROM clientes
      WHERE dni=dni_cliente)=1)THEN
    num_ped_cliente=(SELECT núm_ped
      FROM clientes
      WHERE dni=dni_cliente);
    IF (num_ped_cliente IS NULL) THEN
      descuento=NULL;
    ELSIF (num_ped_cliente<5) THEN
      descuento=0;
    ELSIF (num_ped_cliente<10) THEN
      descuento=3;
    ELSIF (num_ped_cliente<15) THEN
      descuento=5;
    ELSE
      descuento=10;
    END IF;
  END IF;
  RETURN descuento;
END;
$$LANGUAGE plpgsql;
```

2.1.4. Sentencias iterativas

PL/pgSQL proporciona tres tipos de sentencias iterativas:

- Las sentencias `LOOP` y `WHILE` se utilizan para definir bucles cuya finalización esté definida por una expresión condicional.

- La sentencia `FOR` se puede usar cuando sabemos *a priori* el número de iteraciones que se han de ejecutar. También se puede utilizar para iterar sobre el conjunto de filas devueltas por una consulta SQL o una sentencia de ejecución de un procedimiento almacenado.

Lectura recomendada

Podéis consultar la sintaxis de las sentencias `LOOP`, `WHILE` y `FOR` (en la versión en la que se sabe *a priori* el número de iteraciones que se han de llevar a cabo) en los manuales de PostgreSQL.

Vista la tipología de las operaciones que se efectúan contra una BD, la sentencia iterativa más frecuente es `FOR`; concretamente, la versión que permite iterar sobre el conjunto de filas devueltas por una consulta SQL. Por esto nos centraremos en esta sentencia, cuya sintaxis se muestra a continuación:

```
FOR <lista_variables>
         IN <consulta_SQL_o_ejecucion_procedimiento> LOOP
        <bloque_sentencias>
END LOOP;
```

Cuando se ejecuta una sentencia `FOR` para acceder al conjunto de resultados de una consulta SQL, se llevan a cabo las acciones siguientes:

1) Se ejecuta la consulta SQL o el procedimiento almacenado asociado a la sentencia `FOR`.

2) Se asigna a cada variable de la lista de variables el conjunto de valores asociado a la fila actual, que forma parte del resultado de la consulta SQL o de la ejecución del procedimiento almacenado.

3) Se ejecuta el bloque de sentencias.

4) Se obtiene automáticamente la fila siguiente que forma parte del resultado de la consulta SQL o de la ejecución del procedimiento almacenado.

5) Se vuelven a ejecutar los pasos 2), 3) y 4) mientras queden filas que formen parte del resultado de la consulta SQL o de la ejecución del procedimiento almacenado.

Ejemplo de uso de sentencias iterativas

Imaginemos que tenemos las siguientes tablas pedidos, items e items_pedido:

```
CREATE TABLE pedidos(
  num_ped integer primary key,
  dni varchar(9) not null references clientes,
  fecha_llegada date not null,
  importe_total numeric);

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

CREATE TABLE items_pedido (
  num_item integer references items,
  num_ped integer references pedidos,
  cantidad integer not null,
  primary key(num_item,num_ped));
```

El procedimiento almacenado `importe_un_pedido` calcula el importe total del pedido pasado como parámetro.

```
CREATE or REPLACE FUNCTION importe_un_pedido
(ped_cliente pedidos.num_ped%TYPE) RETURNS numeric
AS $$
DECLARE
  total_pedido pedidos.importe_total%type;
  datos_item_precio items.precio_unidad%type;
  datos_item_cant items_pedido.cantidad%type;
BEGIN
  total_pedido=0.0;
  FOR datos_item_precio, datos_item_cant IN
    SELECT ic.cantidad, i.precio_unidad
      FROM items_pedido ic, items i
    WHERE i.núm_item=ic.núm_ítem AND
      ic.num_ped=ped_cliente LOOP
     total_pedido=total_pedido+
    (datos_item_cant*datos_item_precio);
  END LOOP;
  RETURN total_pedido;
END;
$$LANGUAGE plpgsql;
```

El ejemplo anterior muestra la utilización de la sentencia `FOR` para acceder al resultado de una consulta SQL. La cantidad y el precio unitario de cada ítem solicitado en el pedido pasado como parámetro se almacenan en las variables `datos_item_precio` y `datos_item_cant`. Con estos valores se va calculando el importe del pedido y se almacena en la variable `total_pedido`. Al final, el procedimiento devuelve este importe.

Es posible combinar la sentencia `FOR` con sentencias `UPDATE` o `DELETE` que actúen sobre la fila que se trata en una determinada iteración de un bucle. Este uso se ilustra en el ejemplo siguiente:

Ejemplo de uso de sentencias iterativas con la sentencia `UPDATE`

Imaginemos que tenemos las tablas anteriores (`pedidos`, `items`, `items_pedido` y `clientes`). El siguiente procedimiento almacenado, llamado `importe_todos_pedidos`, calcula el importe de todos los pedidos de un cliente, cuyo DNI se pasa como parámetro al procedimiento. El importe total de los pedidos del cliente se almacena en la tabla `pedidos` (`atributo importe_total`). Por esto, el procedimiento no devuelve ningún resultado.

```
CREATE FUNCTION importe_todos_pedidos
      (dni_cliente clientes.dni%type)
RETURNS void AS $$
DECLARE
  num_pedido pedidos.num_ped%type;
  importe_pedido pedidos.importe_total%type;
BEGIN
FOR num_pedido IN SELECT num_ped FROM pedidos
                  WHERE dni=dni_cliente LOOP
    importe_pedido:=importe_un_pedido(num_pedido);
    UPDATE pedidos SET importe_total=importe_pedido
    WHERE num_ped=num_pedido;
END LOOP;
END;
$$LANGUAGE plpgsql;
```

Para invocar el procedimiento almacenado `importe_un_pedido`, también podemos utilizar la siguiente sentencia `SELECT`:

```
SELECT * FROM importe_un_pedido(numpedido) INTO importe_pedido;
```

2.1.5. Retorno de resultados

En los ejemplos anteriores hemos visto cómo se devuelve un solo resultado (de tipo de datos simple, como por ejemplo `integer` o `char`) desde un procedimiento almacenado. A continuación veremos cómo se pueden retornar:

1) Un conjunto de campos que forman una sola fila (un solo resultado, pero de tipo de datos compuesto).

2) Un conjunto de filas (que pueden ser de tipo de datos simple o compuesto).

Retorno de un conjunto de campos

Un procedimiento almacenado puede devolver un conjunto de campos y formar una fila. Para hacer esto, se puede definir un tipo de datos compuesto que tenga la lista de campos que ha de devolver el procedimiento. En el procedimiento almacenado hay que indicar, a continuación de la cláusula `RETURNS`, el nombre del tipo de datos, que se tiene que haber definido previamente.

Ejemplo de uso de procedimientos para devolver un conjunto de atributos

Imaginemos que tenemos la tabla `clientes` que hemos usado anteriormente. Definimos un tipo de datos llamado `tipo_direccion` que tenga el nombre de la calle, el número correspondiente, el código postal y la ciudad de un cliente.

```
CREATE TYPE tipo_dirección AS (
  calle varchar(20),
  núm_calle varchar(4),
  código_postal char(5),
  ciudad varchar(15)
);
```

Finalmente, creamos un procedimiento almacenado llamado `encontrar_direccion_cliente` que, dado el DNI de un cliente, devuelva la dirección completa (nombre de la calle, número de la calle, código postal y ciudad).

```
CREATE FUNCTION encontrar_direccion_cliente
      (dni_cliente clientes.dni%type)
RETURNS tipo_direccion AS $$
DECLARE
    datos_cliente tipo_direccion;
BEGIN
  SELECT calle,num_calle,cp,ciudad
  INTO datos_cliente
  FROM clientes
  WHERE dni=dni_cliente;

  RETURN datos_cliente;
END;
$$LANGUAGE plpgsql;
```

Recordad

La sentencia `CREATE TYPE` permite crear tipos de datos compuestos. Hay que indicar el nombre del tipo de datos y, a continuación, la lista de campos que forman el tipo. PostgreSQL no permite utilizar la cláusula `%ROWTYPE` ni `%TYPE` en la sentencia `CREATE TYPE`.

Retorno de un conjunto de filas

La sentencia iterativa `FOR` se puede combinar con la cláusula `RETURN` para conseguir que un procedimiento devuelva un conjunto de filas como resultado de su ejecución. Hay que hacer dos cosas: en primer lugar, indicar que el procedimiento devuelve un conjunto de filas, para lo que se utiliza la cláusula `SETOF`, y, en segundo lugar, extender la cláusula `RETURN` del cuerpo del procedimiento con la cláusula `NEXT`. El uso de esta última cláusula hace que, después de devolver una fila (la fila actual de la sentencia `SELECT` asociada a la sentencia `FOR`), el procedimiento almacenado continúe la ejecución. El ejemplo siguiente muestra el uso de las cláusulas `SETOF` y `RETURN NEXT` para que un procedimiento almacenado devuelva un conjunto de filas.

Ejemplo de uso de las cláusulas `SETOF` y `RETURN NEXT`

Imaginemos que tenemos la tabla `clientes` del ejemplo anterior y un procedimiento llamado `encontrar_ciudad_cliente` que devuelve todas las ciudades de los clientes que tienen por nombre el parámetro de entrada.

```
CREATE FUNCTION encontrar_ciudad_cliente
      (nombre_cliente clientes.nom%type)
RETURNS SETOF clientes.ciudad%type AS $$
DECLARE
  ciudad_cliente clientes.ciudad%type;
BEGIN
  FOR ciudad_cliente IN SELECT ciudad
                        FROM clientes
                        WHERE nombre=nombre_cliente LOOP
    RETURN NEXT ciudad_cliente;
  END LOOP;
END;
$$LANGUAGE plpgsql;
```

Como se puede ver en el ejemplo anterior, la variable `ciudad_cliente` almacena en cada iteración del bucle un nombre de ciudad. Cuando se ejecuta la sentencia `RETURN NEXT ciudad_cliente`, el procedimiento devuelve el nombre de la ciudad y continúa la ejecución empezando una nueva iteración, o finalizándola si no quedan más filas por procesar.

Finalmente, se puede conseguir que un procedimiento almacenado devuelva un conjunto de filas y que cada una de estas filas esté formada por un conjunto de campos. Para hacer esto, combinamos las cláusulas `SETOF` y `RETURN NEXT` y un tipo de datos definido por el usuario. Este tipo de datos tiene la lista de campos que forman una fila a retornar por el procedimiento almacenado.

Ejemplo de uso de las cláusulas `SETOF`, `RETURN NEXT` y `CREATE TYPE` para devolver un conjunto de filas

Imaginemos que tenemos la tabla `clientes` del ejemplo anterior y un procedimiento llamado `clientes_ciudad` que devuelve el DNI, el nombre y el apellido de todos los clientes que viven en la ciudad que se pasa como parámetro en el procedimiento.

En primer lugar, definimos un tipo de datos que tenga la estructura de la fila que se ha de devolver para cada cliente, es decir, su DNI, el nombre y el apellido.

```
CREATE TYPE tipo_datos_cliente AS (
  dni_cliente VARCHAR(9),
  nombre_cliente VARCHAR(15),
  apellido1 VARCHAR(15));
```

Finalmente, definimos un procedimiento que devuelve `SETOF tipo_datos_cliente`. Esto indica que el procedimiento devuelve un conjunto de filas y que cada fila es del tipo `tipo_datos_cliente`.

```
CREATE FUNCTION clientes_ciudad
      (ciudad_cliente clientes.ciudad%type)
RETURNS SETOF tipo_datos_cliente AS $$
DECLARE
  datos_clientes tipo_datos_cliente;
BEGIN
FOR datos_clientes IN SELECT dni,nombre,apellido1
                      FROM clientes
                      WHERE ciudad=ciudad_cliente LOOP
  RETURN NEXT datos_clientes;
END LOOP;
RETURN;
END;
$$LANGUAGE plpgsql;
```

2.1.6. Invocación de procedimientos almacenados

PostgreSQL ofrece diversas formas de invocar procedimientos almacenados. Comentaremos un par de ellas. La primera consiste en invocar al procedimiento desde la cláusula `SELECT`, tal como se indica a continuación:

```
SELECT nombre_funcion(parametros) [AS alias];
```

Opcionalmente, se puede indicar un nombre alternativo o alias para el resultado del procedimiento. Esta forma de invocar procedimientos almacenados se utiliza normalmente cuando el procedimiento devuelve un solo resultado.

Otro modo de invocar procedimientos almacenados consiste en llamar al procedimiento desde la cláusula `FROM`, tal como se muestra a continuación:

```
SELECT * FROM nombre_funcion (parametros) [AS alias];
```

Esta segunda opción, normalmente se utiliza cuando el procedimiento almacenado devuelve un conjunto de resultados, aunque también se puede usar para invocar procedimientos que devuelven un solo resultado.

Por ejemplo, el procedimiento almacenado `encontrar_ciudad` que tenéis a continuación se podría invocar de las dos formas descritas anteriormente.

```
CREATE FUNCTION encontrar_ciudad(dni_cliente
    clientes.dni%type) RETURNS varchar(15) AS $$
DECLARE
    ciudad_cliente clientes.ciudad%type;
BEGIN
    SELECT ciudad into ciudad_cliente
    FROM clientes
    WHERE dni=dni_cliente;
    RETURN ciudad_cliente;
END;
$$LANGUAGE plpgsql;
```

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

2.1.7. Gestión de errores

Durante la ejecución de un procedimiento almacenado se pueden producir errores. De una manera simplificada, podemos distinguir dos tipos de error:

- Errores predefinidos por el mismo SGBD; por ejemplo, el código de error `23503`, que en PostgreSQL significa `FOREIGN_KEY_VIOLATION`.

- Errores específicos del procedimiento; es decir, errores propios del universo del discurso que se modela y que, por lo tanto, son responsabilidad del creador del procedimiento almacenado.

De las muchas actuaciones alternativas en caso de que se produzca un error en un procedimiento almacenado, mencionamos dos:

1) No capturar el error en el procedimiento almacenado y dejar que el procedimiento cancele su ejecución de una manera inmediata. El error se reporta al nivel superior[1]; es decir, al nivel que había invocado la ejecución del procedimiento almacenado, que se responsabiliza de gestionarlo. Esta opción requiere que el nivel superior conozca los objetos de la BD que se manipulan en el procedimiento y las sentencias que se ejecutan sobre estos objetos.

(1)A una aplicación, a otro procedimiento almacenado o a la herramienta de ejecución del SQL de manera interactiva, si se trabaja con SQL interactivo.

2) Capturar el error en el procedimiento almacenado y dejar que éste se responsabilice de gestionarlo en primera instancia. De las opciones que se ofrecen con esta actuación, comentamos una: el procedimiento cancela su ejecución y el error se reporta en forma de excepción al nivel superior (de lo contrario, éste no se entera) después de haberlo gestionado dentro del mismo procedimiento almacenado. La aplicabilidad de esta opción requiere que el nivel superior sea capaz de capturar y tratar excepciones. Por ejemplo, podríamos tener una aplicación desarrollada en Java que ejecutara un procedimiento almacenado. Si el procedimiento almacenado gestiona los errores reportándolos a la aplicación

Java mediante excepciones, esta aplicación puede capturar las excepciones y hacer lo que corresponda (por ejemplo, se puede cancelar la transacción en curso, cerrar la conexión a la BD y finalizar la aplicación).

Con el fin de favorecer al máximo la encapsulación de los procedimientos almacenados, se recomienda capturar el error. Además, optaremos por hacer que el procedimiento cancele su ejecución y reporte el error en forma de excepción al nivel superior.

El nivel superior ha de saber necesariamente si la ejecución del procedimiento almacenado finaliza o no en una situación de error; de lo contrario, la BD puede quedar en un estado inconsistente. En caso de que se produzca una situación de error, en general, el nivel superior cancelará la transacción que invoca la ejecución del procedimiento almacenado. De hecho, PostgreSQL tiene este comportamiento y, siempre que un procedimiento falla, cancela la transacción en curso.

Para permitir el tratamiento de los errores que se producen durante la ejecución de un procedimiento almacenado, PL/pgSQL proporciona las sentencias `EXCEPTION` (para capturarlos) y `RAISE EXCEPTION` (para generarlos), que explicaremos a continuación.

La sentencia `EXCEPTION`

La sentencia `EXCEPTION` permite especificar las acciones que hay que ejecutar en caso de que se produzcan errores. Esta sentencia se ha de especificar, dentro del procedimiento almacenado, al final del bloque de sentencias, justo antes del `END` que marca el final del procedimiento. Así, la estructura de un procedimiento almacenado con la sentencia `EXCEPTION` es la siguiente:

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

La sentencia `EXCEPTION` consta de una o más cláusulas `WHEN`. La cláusula `WHEN` permite especificar el conjunto concreto de errores de interés que ha de tratar el procedimiento. En general, cada una de las condiciones de la cláusula `WHEN` se especifica utilizando constantes que PostgreSQL tiene definidas para iden-

tificar cada tipo de error. Por ejemplo, la constante FOREIGN_KEY_VIOLATION se utiliza para identificar un error de violación de la restricción de integridad referencial.

Si se produce un error que no pertenece a la lista de errores tratados con las cláusulas WHEN, el error se reporta al nivel superior y la ejecución del procedimiento almacenado se cancela de una manera inmediata.

El <bloque_de_sentencias>, que se ejecuta cuando se cumple una condición de error, representa el conjunto de sentencias que hay que ejecutar cuando se produce un error determinado.

En el <bloque_de_sentencias> se han de incluir todas las sentencias de tratamiento de errores. Hay que destacar que, si se produce un nuevo error cuando se ejecutan estas sentencias, la ejecución del procedimiento se cancelará y el error será reportado de una manera inmediata al nivel superior.

La sentencia RAISE EXCEPTION

La sentencia RAISE EXCEPTION sirve para que el programador pueda generar sus propios errores dentro de un procedimiento. Tiene la sintaxis general siguiente:

```
RAISE EXCEPTION 'mensaje de error';
```

El mensaje de error es una cadena de caracteres que podemos utilizar para explicar los motivos por los que se ha producido un error determinado. A la hora de generar y tratar los mensajes de error, PostgreSQL tiene dos variables especialmente útiles: SQLSTATE y SQLERRM.

Nota

La sentencia RAISE NOTICE ('mensaje') se utiliza para generar o escribir mensajes por pantalla. Esta sentencia se puede usar para consultar el valor de variables concretas del procedimiento en tiempo de ejecución.

- SQLSTATE contiene un código de cinco dígitos asociado al error producido. PostgreSQL sigue así la recomendación del SQL estándar, que aconseja utilizar este código para consultar los errores producidos en vez de consultar los mensajes textuales.

- SQLERRM contiene un mensaje explicativo asociado al error producido.

Ejemplo de gestión de errores

Queremos modificar el código del procedimiento calculo_descuento_cliente de manera que controle los siguientes errores específicos:

- El cliente no existe.
- El cliente no tiene pedidos.

Según lo que hemos explicado en este apartado, hay dos formas posibles de definir este procedimiento.

1) No capturar el error en el procedimiento almacenado.

```
CREATE FUNCTION calculo_descuento_cliente
      (dni_cliente clientes.dni%type)
RETURNS integer AS $$
DECLARE
  descuento INTEGER;
  num_pedidos_cliente INTEGER;
BEGIN
  IF ((SELECT COUNT(*)
    FROM clientes WHERE dni=dni_cliente)=1)THEN
    num_pedidos_cliente=(SELECT num_ped
        FROM clientes
        WHERE dni=dni_cliente);
    IF (num_pedidos_cliente IS NULL) THEN
      RAISE EXCEPTION
      'El cliente % no tiene pedidos',dni_cliente;
    ELSIF (num_pedidos_cliente<5) THEN
      descuento=0;
    ELSIF (num_pedidos_cliente<10) THEN
      descuento=3;
    ELSIF (num_pedidos_cliente<15) THEN
      descuento=5;
    ELSE
    descuento=10;
    END IF;
  ELSE
    RAISE EXCEPTION
      'El cliente % no existe',dni_cliente;
  END IF;
  RETURN descuento;
END;
$$LANGUAGE plpgsql;
```

Nota

En la cadena 'El cliente % no existe', el símbolo % es sustituido en tiempo de ejecución por el valor de la variable `dni_cliente`.

En esta alternativa, todos los errores se reportan al nivel superior y se cancela la ejecución del procedimiento y de la transacción en curso. Por ejemplo, si ejecutamos este procedimiento desde el editor de SQL de pgAdmin, en caso de que se produzca un error específico del procedimiento, en la parte inferior de la pantalla aparecerá una ventana que indicará que se ha producido el error con SQLSTATE P0001 y su mensaje asociado ("el cliente no tiene pedidos" o "el cliente no existe").

Pantalla de error en PostgreSQL

```
ERROR:  El cliente 45678000 no existe

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

ERROR:  El cliente 45678000 no existe
SQL state: P0001
```

Fijaos en que en este ejemplo el procedimiento almacenado no tiene la sentencia `EXCEPTION`, porque no gestiona los errores específicos del procedimiento.

2) Capturar el error en el procedimiento almacenado y reportarlo al nivel superior mediante excepciones.

```
CREATE FUNCTION calculo_descuento_cliente
      (dni_cliente clientes.dni%type)
RETURNS integer AS $$
DECLARE
  descuento INTEGER;
  num_pedidos_cliente INTEGER;
BEGIN
  IF ((SELECT COUNT(*)
    FROM clientes WHERE dni=dni_cliente)=1)THEN
    num_pedidos_cliente=(SELECT num_ped
        FROM clientes
```

```
        WHERE dni=dni_cliente);
  IF (num_pedidos_cliente IS NULL) THEN
    RAISE EXCEPTION
      'El cliente % no tiene pedidos',dni_cliente;
  ELSIF (num_pedidos_cliente<5) THEN
    descuento=0;
  ELSIF (num_pedidos_cliente<10) THEN
    descuento=3;
  ELSIF (num_pedidos_cliente<15) THEN
    descuento=5;
  ELSE
  descuento=10;
  END IF;
ELSE
  RAISE EXCEPTION
      'El cliente % no existe',dni_cliente;
END IF;
RETURN descuento;

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

En este ejemplo, el procedimiento trata los errores en la sentencia `EXCEPTION`, que se encuentra al final del mismo. En nuestro ejemplo hay dos casos de error que hay que tratar:

- El primer caso (`raise_exception`) captura los errores específicos del procedimiento, que son dos: el cliente no tiene pedidos o el cliente no existe.
- El segundo caso (`others`) captura cualquier otro error que no sea el anterior y el error de PostgreSQL llamado `query_cancelled`.

Las dos palabras reservadas que hay en la cláusula `WHEN` (`raise_exception` y `others`) son constantes predefinidas por PostgreSQL que tienen asociado un código `SQLSTATE` concreto. El nombre de estas palabras reservadas se puede escribir con letras mayúsculas o minúsculas.

Notación

En la cláusula `WHEN` también se puede utilizar directamente el código `SQLSTATE` en vez de las constantes predefinidas por PostgreSQL. Podéis consultar el manual para obtener más información.

En este último ejemplo, en caso de error, el procedimiento lo captura y, después de haberlo tratado, lo reporta mediante una excepción al nivel superior. Fijaos que el tratamiento del error consiste en encapsular los errores predefinidos de PostgreSQL, como `foreign_key_violation` o `not_null_violation`. Nos interesa que el procedimiento esconda todos estos errores y muestre sólo un mensaje genérico, que en nuestro ejemplo es `'error interno'`.

Nota

Si no interesa esconder los errores predefinidos de PostgreSQL, se puede trabajar con la opción 1), es decir, sin capturar errores dentro del procedimiento almacenado.

Como último ejemplo, supongamos que queremos hacer un procedimiento que borre un ítem de la tabla de ítems. El número del ítem a borrar se pasa como parámetro al procedimiento. En caso de que el ítem se pueda borrar, el procedimento no retornará nada. En caso de que se produzca alguno de los errores siguientes, el procedimiento tendrá que informarlo generando excepciones.

1) El ítem no existe.

2) El ítem tiene pedidos.

```
CREATE OR REPLACE FUNCTION eliminar_item (item integer)
RETURNS void AS $$
DECLARE
   mensaje varchar (50);

BEGIN
   DELETE FROM items WHERE num_item = item;
   IF NOT FOUND THEN
      RAISE EXCEPTION
         'El item no existe', mensaje;
   END IF;

EXCEPTION
   WHEN raise_exception THEN
      RAISE EXCEPTION '%', SQLERRM;
   WHEN foreign_key_violation THEN
      RAISE EXCEPTION 'El item tiene pedidos';
END;
$$LANGUAGE plpgsql;
```

El ejemplo anterior trata dos errores:

- Un error específico del procedimiento (caso `WHEN raise_exception`). En este caso se utiliza la variable especial de PostgreSQL `FOUND` para saber si la sentencia `delete` ha borrado alguna fila.

- Un error predefinido de PostgreSQL (caso `WHEN foreign_key_violation`) que es encapsulado por el procedimiento para poder mostrar al usuario un mensaje de error personalizado, en vez del mensaje de error predefinido de PostgreSQL.

Fijaos que en este ejemplo (si no ponemos el caso `WHEN others` en la sentencia `EXCEPTION`), si el procedimiento falla por algún error que no sea alguno de los dos anteriores, el usuario recibirá un mensaje de error predefinido de PostgreSQL informando del error producido.

Variable FOUND

La variable FOUND es una variable especial de tipo booleano de PosgreSQL. Su valor inicial es falso, pero este valor puede cambiar cuando se ejecutan sentencias SQL. Concretamente, cuando se ejecutan sentencias de actualización, la variable FOUND tiene valor cierto si como mínimo una fila se ve afectada por la sentencia de actualización. Si sucede lo contrario, su valor es falso. Podéis consultar el manual de PostgreSQL para obtener más información.

3. Disparadores

Los componentes lógicos de una BD vistos hasta ahora son insuficientes para implementar adecuadamente algunas de las situaciones que se producen en el mundo real.

Imaginemos, por ejemplo, que tenemos una tabla de existencias de productos de una organización determinada que tiene una regla de negocio definida según la cual, cuando el estoc de un producto queda por debajo de cincuenta unidades, hay que hacer un pedido de cien unidades. Con los componentes vistos hasta ahora, se presentan dos soluciones convencionales para implementar esta situación:

- Añadir esta regla a todos los programas que actualizan las existencias de los productos.
- Hacer un programa adicional que haga un sondeo (en inglés, *polling*) periódico de la tabla para comprobar la regla.

La primera solución tiene diversos inconvenientes: la regla de negocio está escondida, distribuida y reproducida dentro del código de los programas y, por lo tanto, es difícil de localizar y cambiar. Además, su eficacia o corrección depende del hecho de que cada programador inserte la regla correctamente.

La segunda solución, aunque la regla está concentrada en un solo sitio (el programa de sondeo), presenta los inconvenientes clásicos del sondeo: si se hace ocasionalmente, se puede perder el buen momento para reaccionar, y si, en cambio, se hace muy a menudo, se pierde eficiencia.

La **solución correcta** es dotar al sistema de actividad. Se incorpora a la BD un nuevo componente, los disparadores (en inglés, *triggers*), que modelan la regla de negocio y se ejecutan de forma automática. Así, para el problema planteado (y sin entrar todavía en la sintaxis concreta), el disparador que se incorporaría a la BD es el siguiente: "Cuando se modifiquen las existencias de un producto y queden por debajo de cincuenta unidades, hay que pedir cien unidades nuevas".

Como se puede ver, con esta solución se superan los inconvenientes de las dos soluciones vistas anteriormente: la regla de negocio está sólo en un sitio, la eficacia no depende del programador y el disparador se ejecutará siempre que haga falta y sólo cuándo haga falta.

Resumiendo, los disparadores son unos componentes que se ejecutan de una manera automática cuando se produce un evento determinado. Se dice que un disparador es una regla **ECA** (**e**vento, condición, **a**cción): cuando se produce un evento determinado, si se cumple una condición, se ejecuta una acción.

En el ejemplo anterior, el evento que activa el disparador es la modificación de las existencias de algún producto; la condición, que las existencias queden por debajo de cincuenta unidades, y la acción que se ejecuta, el hecho de pedir cien unidades nuevas.

3.1. Cuándo se han de utilizar disparadores

Hay toda una serie de situaciones en las que es posible usar disparadores y conviene hacerlo. Entre ellas mencionamos las siguientes:

Las actualizaciones

Tenemos que entender el concepto *actualizaciones* en el sentido amplio: inserciones, borrados y modificaciones.

- Implementación de una regla de negocio. Como ejemplo, podemos revisar el caso de las existencias mencionado anteriormente.

- Mantenimiento automático de una tabla de auditoría de la actividad en la BD. Se trata de registrar de una manera automática los cambios que se hacen en una tabla determinada.

- Mantenimiento automático de columnas derivadas. El valor de una columna derivada se calcula a partir del valor de otras columnas; posiblemente, de otras tablas. Cuando se modifican las columnas base, hay que recalcular de una manera automática el valor de la columna derivada.

- Comprobación de restricciones de integridad no expresables en el sistema mediante `CHECK` o por medio de restricciones de integridad referencial. Como ejemplo, podemos considerar las restricciones dinámicas. La más clásica, y por otra parte normalmente bienvenida, es "un sueldo no puede bajar". Esta restricción hace referencia al estado anterior y posterior a una modificación y, por lo tanto, no se puede expresar como `CHECK`. Otro ejemplo de este tipo de restricciones son las aserciones, restricciones expresables en SQL estándar, pero que no implementa ningún SGBD.

- Reparación automática de restricciones de integridad. Hay que distinguir entre *comprobación* y *reparación* de restricciones de integridad. La semántica de la comprobación de una restricción de integridad se puede resumir así: "Si una actualización infringe la restricción, hay que cancelar anormalmente o deshacer (en inglés, *abort*) la actualización." La semántica de la reparación es la siguiente: "Si la restricción se infringe, hay que llevar

a cabo acciones compensatorias (nuevas actualizaciones) para que no se infrinja". Los SGBD normalmente implementan las comprobaciones.

3.2. Cuándo no se han de utilizar disparadores

Si bien los disparadores, como hemos visto, pueden ser útiles a la hora de implementar determinadas situaciones del mundo real, no hay que abusar de ellos: no se deberían utilizar disparadores en las situaciones en las que el sistema se pudiera resolver con sus propios mecanismos. Así, por ejemplo, no se tendrían que usar para implementar las restricciones de clave primaria, ni las restricciones de integridad referencial, ni todas las expresables como `CHECK`, etc.

3.3. Sintaxis de los disparadores en PostgreSQL

Actualmente, la mayoría de los SGBD disponen de disparadores. Los encontramos tanto en los sistemas comerciales (como Oracle, DB2, SQLServer o Informix) como en los de libre distribución (PostgreSQL o MySQL). De hecho, el SQL estándar, desde el SQL:1999, define los disparadores como componentes esenciales de las BD.

No obstante, los lenguajes que usan los diversos sistemas para definirlos y sus capacidades de expresividad son ligeramente diferentes no sólo entre los SGBD, sino también entre éstos y el SQL:1999. En este módulo hemos optado por explicar la sintaxis de los disparadores en PostgreSQL con el fin de poder ejecutar los ejemplos que veremos y resolver los ejercicios en un sistema real.

A continuación describimos la sintaxis de los disparadores en PostgreSQL. En los subapartados siguientes presentaremos ejemplos de cómo funcionan.

```
CREATE TRIGGER <nombre_disparador> {BEFORE | AFTER}
{<evento> [OR <evento>]} ON <tabla>
 [FOR [EACH] {ROW | STATEMENT}]
 EXECUTE PROCEDURE <nombre_procedimiento();>
```

Como se puede ver, un disparador tiene un nombre, un evento (como mínimo) y una tabla asociados:

- El nombre sirve para identificar el disparador en caso de que se quiera modificar[2] o borrar.

[(2)]Para modificar un disparador, hay que borrarlo y volverlo a crear.

- El evento especifica el tipo de sentencia que activa el disparador. En particular, en PostgreSQL son las siguientes:

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

Para cada disparador hay que definir, como mínimo, un procedimiento almacenado que contenga las acciones que se ejecutarán cuando se active. Las cláusulas `BEFORE`, `AFTER`, `FOR EACH ROW` y `FOR EACH STATEMENT` sirven para especificar cuándo se ejecutará el procedimiento asociado al disparador.

Activación de un disparador en la versión 9.0 de PostgreSQL

A partir de la versión 9.0 de PostgreSQL, se puede activar un disparador cuando se modifica el valor de alguna columna de una tabla concreta. Las versiones anteriores a la 9.0 no soportan esta característica; es decir, el evento de `UPDATE` no acepta que se especifique la columna o las columnas que se modifican en la sentencia `CREATE TRIGGER`.

El procedimiento almacenado que contiene las acciones que ha de ejecutar el disparador es especial de PostgreSQL y se denomina *trigger procedure*. Se caracteriza por que no recibe parámetros y devuelve un tipo de datos especial llamado *trigger*. Este procedimiento se ha de definir antes que el disparador que lo invoca. Un mismo procedimiento puede ser invocado por diversos disparadores.

En PostgreSQL, los disparadores pueden ser `BEFORE` o `AFTER`:

- Si el disparador es `BEFORE`, el procedimiento asociado se ejecuta antes que la sentencia que activa el disparador.
- Si el disparador es `AFTER`, el procedimiento asociado se ejecuta después que la sentencia que activa el disparador.

Además, los disparadores pueden ser `FOR EACH ROW` o `FOR EACH STATEMENT`:

- Si el disparador es `FOR EACH ROW`, el procedimiento asociado se ejecuta una vez por cada fila afectada por la sentencia que activa el disparador.

- Si el disparador es `FOR EACH STATEMENT`, el procedimiento se ejecuta una vez, independientemente del número de filas afectadas por la sentencia que activa el disparador.

Nota

Las "filas afectadas" en los disparadores `FOR EACH ROW` son las filas insertadas, borradas o modificadas por la sentencia SQL que dispara el disparador.

Así, por ejemplo, una operación de `DELETE` que afecte a diez filas de la tabla T hará que el procedimiento asociado a cualquier disparador `FOR EACH ROW` definido sobre la tabla T se ejecute diez veces, una vez por cada fila borrada. En cambio, si el disparador se ha definido `FOR EACH STATEMENT`, el procedimiento en cuestión se ejecutará una sola vez (con independencia del número de filas borradas).

Cuando se define un disparador, hay que especificar si se ejecutará `BEFORE` o `AFTER` y si será `FOR EACH ROW` o `FOR EACH STATEMENT`. Por lo tanto, podemos tener cuatro tipos de disparadores:

- Disparador `BEFORE / FOR EACH STATEMENT`. El procedimiento asociado al disparador se ejecuta **una sola vez antes** de la ejecución de la sentencia que activa el disparador.

- Disparador `BEFORE / FOR EACH ROW`. El procedimiento asociado al disparador se ejecuta **una vez por cada fila afectada y** justo **antes** de que la fila se inserte, se modifique o se borre.

- Disparador `AFTER / FOR EACH STATEMENT`. El procedimiento asociado al disparador se ejecuta **una sola vez después** de la ejecución de la sentencia que activa el disparador.

- Disparador `AFTER / FOR EACH ROW`. El procedimiento asociado al disparador se **ejecuta una vez por cada fila afectada y después** de la ejecución de la sentencia que activa el disparador.

Finalmente, hay que decir que un disparador se borra con la sentencia siguiente:

```
DROP TRIGGER <nombre_disparador> ON <nombre_tabla>;
```

3.3.1. Procedimientos invocados por disparadores

Se trata de procedimientos almacenados especiales que PostgreSQL denomina *trigger procedures*. Como ya hemos dicho, estos procedimientos no pueden recibir parámetros de la manera habitual que hemos explicado antes y han de devolver un tipo de datos especial llamado *trigger*.

La forma de pasar parámetros a estos procedimientos almacenados es a través de variables especiales de PostgreSQL que se crean y se instancian automáticamente cuando se ejecuta el procedimiento. Algunas de estas variables son las siguientes:

- `TG_OP`. Es una cadena de texto que contiene el nombre de la operación que ha activado el disparador. Puede tener los valores siguientes: `INSERT`, `UPDATE` o `DELETE` (con mayúsculas).

- `NEW`. Para disparadores del tipo `FOR EACH STATEMENT`, tiene el valor `NULL`. Para disparadores del tipo `FOR EACH ROW`, es una variable de tipo compuesto que contiene la fila después de la ejecución de una sentencia de modificación (`UPDATE`) o la fila que hay que insertar (`INSERT`).

- `OLD`. Para disparadores del tipo `FOR EACH STATEMENT`, tiene el valor `NULL`. Para disparadores del tipo `FOR EACH ROW`, es una variable de tipo com-

puesto que contiene la fila antes de la ejecución de una sentencia de modificación (UPDATE) o la fila que hay que borrar (DELETE).

Ejemplo de instanciación de las variables NEW y OLD

Supongamos que tenemos definida la siguiente tabla t. Sobre esta tabla se ha definido el disparador llamado trig. Consideremos también que se ejecuta la sentencia de modificación que tenemos más abajo y que esta sentencia activa 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;
```

Supongamos que el contenido inicial de la tabla t es el siguiente:

a	b
1	2

Durante la ejecución del disparador, las variables NEW y OLD tienen los valores siguientes:

- Valores antes de que se ejecute la sentencia UPDATE:

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

- Valores después de que se ejecute la sentencia UPDATE:

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

Con respecto al retorno de resultados, los procedimientos invocados por disparadores pueden devolver NULL o bien una variable de tipo compuesto que tenga la misma estructura que las filas de la tabla sobre la que está definido el disparador. Es el caso de las variables OLD y NEW.

Nota

Para acceder a los valores de cada fila modificada por la sentencia UPDATE, utilizaremos la notación siguiente:
OLD.nombre_columna_tabla
o
NEW.nombre_columna_tabla.

Por lo tanto, los procedimientos invocados por disparadores pueden devolver los valores siguientes:

1) Disparadores BEFORE / FOR EACH ROW:

- NULL, para indicar al disparador que no ha de acabar la ejecución de la operación para la fila actual. En este caso, la sentencia que activa el disparador (INSERT/UPDATE/DELETE) no se llega a ejecutar.

- NEW, para indicar que la ejecución del procedimiento para la fila actual ha de acabar normalmente y que la sentencia que activa el disparador (INSERT/UPDATE) se ha de ejecutar. En este caso, el procedimiento puede devolver el valor original de la variable NEW o modificar su contenido. Si

el procedimiento modifica el contenido de la variable NEW, está variando directamente la fila que se insertará o se modificará.

- OLD, para indicar que la ejecución del procedimiento por la fila actual ha de acabar normalmente y que la sentencia que activa el disparador (DELETE / UPDATE) se ha de ejecutar. En el caso de UPDATE, si el procedimiento retorna OLD, no hace la modificación de la fila actual.

2) Otros tipos de disparadores: en los disparadores AFTER / FOR EACH ROW y en los disparadores FOR EACH STATEMENT (independientemente de que sean BEFORE o AFTER), el valor devuelto por el procedimiento que es invocado por el disparador es ignorado. Por esto, estos procedimientos normalmente devuelven NULL.

Ved también

El disparador del apartado 3.3.2 (ejemplo de mantenimiento automático de un atributo derivado) ilustra el uso de la variable NEW para modificar los datos de las filas afectadas por el disparador.

La motivación principal para que los disparadores BEFORE y FOR EACH ROW devuelvan un valor diferente de nul, es que pueden modificar los datos de las filas que se insertarán o se modificarán en la tabla asociada al disparador. Veamos un ejemplo.

Ejemplo de retorno de resultados en un trigger-procedure

Supongamos que tenemos definida la siguiente tabla t. Sobre esta tabla se ha definido el disparador llamado trig. Consideremos también que se ejecuta la siguiente sentencia de inserción, que activa 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 procedimiento prog() puede devolver dos valores:

- NULL. En este caso, la fila <1,2> no se inserta en la tabla t.
- NEW. En este caso, tenemos dos posibilidades. Si el valor de la variable NEW no ha sido modificado por el procedimiento prog(), la fila <1,2> se inserta en la tabla t. Si el valor de la variable NEW ha sido modificado por el procedimiento prog() (por ejemplo, si se ha ejecutado la operación NEW.b=3), se inserta la fila <1,3> en la tabla t.

3.3.2. Ejemplos de disparadores

A continuación, presentamos una serie de ejemplos que, además de ilustrar la sintaxis de PostgreSQL, servirán para mostrar algunos de los usos de los disparadores.

Mantenimiento automático de una tabla de auditoría

Al presentar este disparador ilustraremos al mismo tiempo el uso de la cláusula `FOR EACH ROW` y el de la cláusula `AFTER`. También veremos el uso de las variables especiales `NEW` y `OLD`.

El objetivo del disparador es mantener de una manera automática una tabla de auditoría (`log_record`) que contenga un registro de todas las modificaciones de la columna `cant` que hacen los usuarios en la tabla `items` de cierta BD. Así, imaginemos que disponemos de las tablas siguientes:

```
CREATE TABLE log_record(
  item integer,
  username char(8),
  hora_modif timestamp,
  cant_vieja integer,
  cant_nueva integer);

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

Puesto que el disparador ha de guardar una fila por cada modificación hecha en la columna `cant` de la tabla `items`, en este caso lo más adecuado es utilizar un disparador del tipo `AFTER` y `FOR EACH ROW`. El disparador se define de tipo `AFTER` para insertar filas en la tabla de auditoría sólo en caso de que se hayan producido modificaciones de la cantidad de algún `ítem`. Si definiéramos el disparador como `BEFORE`, las inserciones en la tabla de auditoría se harían antes de que se produjera la modificación de la cantidad de algún `ítem`. Si por algún motivo la modificación de la cantidad de algún `ítem` fallara, ya habríamos hecho la inserción en la tabla de auditoría, realizando más trabajo del necesario.

Por otro lado, el disparador se define del tipo `FOR EACH ROW` porque nos interesa guardar un registro de auditoría para cada `ítem` del que se modifica la cantidad.

A continuación, tenemos el código del disparador y del procedimiento almacenado invocado por este disparador.

```
CREATE FUNCTION inserta_log() RETURNS trigger AS $$ BEGIN
  INSERT INTO log_record VALUES
(OLD.item,current_user,current_date,OLD.cant,NEW.cant);
RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER auditoría_items
AFTER UPDATE OF cant ON items
FOR EACH ROW EXECUTE PROCEDURE inserta_log();
```

Cada vez que se actualice la columna `cant` de la tabla `items`, se insertará una fila en la tabla `log_record`. Las variables `OLD.cant` y `NEW.cant` contendrán respectivamente los valores antiguos y nuevos de esta columna.

`current_user` y `current_date`

Las funciones predefinidas de PostgreSQL `current_user` y `current_date` devuelven, respectivamente, el nombre del usuario que ha ejecutado la sentencia que activa el disparador y el instante de ejecución.

Consideremos el contenido siguiente de la tabla ítems:

ítem	nombre	cant	precio_total
1	saco	100	0,30
2	boli	5000	0,50
3	rat	500	0,60

Si ejecutamos la sentencia `UPDATE items SET cant=cant+10 WHERE item<>3` sobre la tabla anterior, para cada fila actualizada (en este caso, las filas 1 y 2) se ejecutará la inserción correspondiente, y en la tabla `log_record` se obtendrá el resultado siguiente:

ítem	username	hora_modif	cant_vieja	cant_nueva
1	juan	2010-04-15 15:00	100	110
2	juan	2010-04-15 15:00	5000	5010

Leamos la tabla: el usuario `juan` actualizó dos filas de la tabla `items` y aumentó la cantidad de cada una de ellas en diez unidades el día 15 de abril de 2010 a las 15:00 horas.

Actividad

Como actividad complementaria, podéis volver a implementar este ejemplo de auditoría considerando que ahora sólo hay que almacenar el `username` del usuario que hace la modificación y la fecha y la hora en las que se produce esta modificación en la tabla de auditoría. No hace falta almacenar el código del ítem modificado ni la cantidad antes y después de la modificación. ¿Cambiaría el tipo de disparador (`BEFORE/AFTER`, `FOR EACH ROW / FOR EACH STATEMENT`)?

Mantenimiento automático de un atributo derivado

En este ejemplo se muestra cómo se utilizan los disparadores para mantener calculado de una manera automática el atributo derivado `precio_total` de la tabla `ítems` cuando se producen modificaciones de la cantidad de existencias de algún ítem.

El valor del atributo derivado se calcula a partir del valor de otras columnas de la misma tabla o bien de otras tablas. Por lo tanto, cuando se modifican las columnas que se utilizan para calcular el atributo derivado, hay que recalcular de forma automática el valor del atributo derivado.

Partamos nuevamente de la siguiente tabla `items`:

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

Definimos el `atributo precio_total`, un atributo derivado cuyo valor se calculará tal como sigue:

```
CREATE FUNCTION calcula_nuevo_precio_total()
RETURNS trigger AS $$
BEGIN
  IF (old.cant<>0) then

NEW.precio_total=((OLD.precio_total/OLD.cant)*NEW.cant);
  END IF;
RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER atributo_derivado
BEFORE UPDATE OF cant ON items
FOR EACH ROW
EXECUTE PROCEDURE calcula_nuevo_precio_total();
```

El procedimiento `calcula_nuevo_precio_total` devuelve el nuevo precio total de un ítem, calculado a partir del precio total anterior que teníamos en la BD y la cantidad del ítem antes y después de la modificación.

Fijaos, en primer lugar, en que el procedimiento sólo recalcula el precio total de un ítem cuando se produce una modificación de la cantidad de existencias del mismo. En segundo lugar, el recálculo del nuevo precio total se asigna a la variable `NEW.precio_total`. Con esta asignación modificamos directamente el contenido de la variable `NEW` antes de que se produzca la modificación de la cantidad de existencias de un ítem. Finalmente, el procedimiento devuelve la variable `NEW`. Este retorno sirve para que la modificación del precio total que realiza el procedimiento se acabe efectuando en la tabla `items`.

Recordad

Es más eficiente que los procedimientos invocados por disparadores sólo lleven a cabo las acciones cuando hace falta. En el ejemplo anterior, sólo se vuelve a calcular el precio total cuando se modifica la cantidad de algún ítem.

El procedimiento `calcula_nuevo_precio_total` ha de devolver la variable `NEW` para que la modificación del precio total se guarde en la tabla `items`.

Implementación de una regla de negocio

Con este ejemplo ilustraremos el uso de más de un disparador para implementar una regla de negocio. Además, veremos la manera de cancelar (en inglés, *abort*) la ejecución de la sentencia que ha activado el disparador y toda la transacción en curso.

Partamos de la tabla `items` del ejemplo anterior:

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

El objetivo del nuevo disparador es implementar la regla de negocio siguiente: "No puede ser que una sola sentencia de modificación (`UPDATE`) aumente la cantidad total de las existencias de los productos más de un 50%".

Puesto que esta regla de negocio no requiere, *a priori*, un tratamiento para cada una de las filas, parece natural implementarla con disparadores `FOR EACH STATEMENT`. Un primer disparador ejecutado `BEFORE` puede sumar las existencias de los ítems antes de la actualización y un segundo disparador `AFTER` puede sumarlas después y hacer la comparación de las dos cantidades:

```
CREATE TRIGGER regla_negocio_antes BEFORE UPDATE ON items
FOR EACH STATEMENT
  EXECUTE PROCEDURE update_items_antes();

CREATE TRIGGER regla_negocio_despues AFTER UPDATE ON items
FOR EACH STATEMENT
  EXECUTE PROCEDURE update_items_despues();
```

Esta solución tiene un problema. En PostgreSQL, los procedimientos invocados para disparadores no pueden devolver cualquier valor; por lo tanto, no tenemos manera de devolver la cantidad de existencias de ítems antes de que se produzca la modificación. Para poderlo hacer, definiremos la siguiente tabla temporal:

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

El procedimiento `update_items_antes` guardará la cantidad de existencias en esta tabla temporal, y el procedimiento `update_items_despues` la consultará y la limpiará para próximas ejecuciones.

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

CREATE FUNCTION update_items_despues()RETURNS
trigger AS $$
DECLARE
  cant_antes integer default 0;
  cant_despues integer default 0;
BEGIN
  SELECT cant_vieja into cant_antes FROM temp;
  DELETE FROM temp;
  SELECT sum(cant) into cant_despues FROM items;
  IF (cant_despues>cant_antes*1.5) THEN
    RAISE EXCEPTION 'Infraccion regla de negocio';
  END IF;
  RETURN NULL;
END
$$ LANGUAGE plpgsql;
```

Recordad

Para poder probar el ejemplo, hay que crear la tabla temporal y los procedimientos almacenados antes que los disparadores.

Fijaos en que, si la sentencia de modificación no cumple la regla de negocio, el procedimiento almacenado ejecutado `AFTER` genera una excepción que deshace la sentencia que activa el disparador y toda la transacción en curso. En cambio, si la ejecución de una sentencia de modificación no provoca la violación de la regla de negocio, la sentencia `UPDATE` se ejecuta correctamente.

Debemos hacer un comentario final sobre este disparador. Como hemos visto, el disparador calcula dos veces las existencias de los productos: una vez antes de la actualización y la otra después. Puede ser interesante plantearse la posibilidad de definir un disparador alternativo que haga la misma función de forma más eficiente, obviando una de las dos sumas totales. La clave está en tener en cuenta las filas que se actualizan: no hará falta la segunda suma si se conoce el resultado de la primera y las actualizaciones realizadas. Sólo habrá que hacer una vez la suma, acumular las actualizaciones que se hagan y deducir la otra suma; es decir, tanto podemos hacer un disparador con acciones `FOR EACH ROW` (que acumule) y `AFTER` (que calcule la suma), como el simétrico con acciones `BEFORE` (que calcule la suma) y `FOR EACH ROW` (que acumule).

Esta técnica para aumentar la eficiencia de los disparadores que tiene en cuenta lo que cambia y no hace comprobaciones redundantes se conoce con el nombre de *disparadores incrementales*. Siempre que se pueda, conviene utilizarla.

Actividad

Como actividad complementaria, podéis implementar esta regla de negocio con este enfoque.

3.3.3. Otros aspectos que hay que tener en cuenta

Otros aspectos sobre los disparadores que conviene considerar son su orden de ejecución, los errores, las restriccions de integridad o los disparadores en cascada.

Orden de ejecución de los disparadores

Como hemos visto anteriormente, es posible que para implementar una semántica o situación haga falta más de un disparador. En estos casos, puede que se tengan que implementar diversos disparadores para el mismo evento (`INSERT`, `UPDATE` o `DELETE` sobre la misma tabla). Cuando esto pasa, PostgreSQL fija la orden de ejecución de los disparadores tal como sigue:

- Los disparadores `BEFORE / FOR EACH STATEMENT` se ejecutan antes que los disparadores `BEFORE / FOR EACH ROW`.

- Los disparadores `AFTER / FOR EACH STATEMENT` se ejecutan después que los disparadores `AFTER / FOR EACH ROW`.

Sólo en el caso de que tengamos dos disparadores para el mismo evento y de que sean del mismo tipo (por ejemplo, dos disparadores `BEFORE / FOR EACH ROW` sobre la misma tabla), se utilizará el orden alfabético del nombre del disparador para determinar qué disparador se ejecuta antes. El disparador cuyo nombre sea el primero según el orden alfabético es el que se ejecutará en primer lugar.

Disparadores y errores

Cuando un disparador falla a causa de la ejecución de una de sus sentencias SQL, el sistema devuelve el error concreto SQL que se ha producido. Además, en el subapartado anterior hemos visto un ejemplo de cómo desde un disparador se puede llamar a un procedimiento almacenado que provoque el lanzamiento de una excepción (en el ejemplo se lanzaba la excepción cuando no se cumplía la regla de negocio especificada). Esta situación nos sirve para simular un error SQL.

Recapitulando, tenemos dos situaciones que hacen que el disparador falle:

- Se ejecuta una sentencia interna que provoca el error.
- Se provoca el error mediante el lanzamiento de la excepción.

Tanto en un caso como en el otro, todas las acciones que ha podido hacer el disparador (y las que hayan podido hacer los disparadores activados por las acciones del primero) y la transacción en curso se deshacen de una manera automática.

Ejemplo

Consideremos el siguiente esquema de activación de disparadores:

```
BEGIN WORK;
Sentencia 1;
Sentencia 2;
Sentencia 3;
Provoca la activacion del disparador D1;
D1:
Sentencia 4;
Provoca la activacion del disparador D2;
D2:
Sentencia 5;
Provoca la activacion del disparador D3;
D3:
Sentencia 6;
falla
```

En este caso, se desharán todas las acciones de los disparadores `D1`, `D2` y `D3` y las sentencias 1, 2 y 3.

Disparadores y restricciones de integridad

Cuando se ejecuta una sentencia SQL contra una BD, puede suceder que se activen disparadores y que se violen restricciones de integridad. Ante esta circunstancia, el sistema ha de decidir qué tiene que hacer en primer lugar: activar los disparadores o bien comprobar las restricciones de integridad.

En PostgreSQL, las acciones de los disparadores `BEFORE` se ejecutan antes de comprobar las restricciones de integridad de la BD. En cambio, las acciones de los disparadores `AFTER` se ejecutan después de comprobar las restricciones de integridad de la BD. Veamos un ejemplo.

Ejemplo

Consideremos las tablas siguientes, que expresan la restricción de integridad referencial "todo hijo ha de tener un padre".

```
CREATE TABLE padre(
  a INTEGER PRIMARY KEY);
CREATE TABLE hijo(
  b INTEGER PRIMARY KEY padre,
  c INTEGER REFERENCES);
```

Podemos definir un disparador que se active cuando se inserte un hijo que no tenga especificado un padre y que, como reparación, lo inserte en la tabla padre.

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

CREATE TRIGGER restricciones1 BEFORE INSERT ON hijo
FOR EACH ROW EXECUTE PROCEDURE insertar();
```

La sentencia siguiente provoca la inserción del valor 1 en la tabla `padre`, puesto que el disparador se ejecuta antes de comprobar la restricción de integridad referencial.

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

En cambio, si el disparador `restricciones1` hubiera sido definido `AFTER`, la sentencia de inserción anterior habría producido un error de violación de la restricción de integridad referencial. Esto es así porque la restricción de integridad referencial se comprueba antes de ejecutar el disparador.

Disparadores en cascada

Normalmente, los SGBD permiten encadenar la ejecución de los disparadores en cascada; es decir, un disparador puede ejecutar una sentencia que, a su vez, active un disparador, el cual, de rebote, ejecute una sentencia, etc. PostgreSQL no pone límite al número de disparadores que se pueden ejecutar en cascada. Este comportamiento puede dar lugar a un bucle infinito. Veamos el ejemplo siguiente.

Ejemplo

```
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 sentencia siguiente provoca la activación de los disparadores anteriores, y se produce un bucle infinito de borrados sucesivos.

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

Estas secuencias de activación de disparadores en cascada son peligrosas porque pueden producir bucles infinitos. Según PostgreSQL, es responsabilidad del programador evitar que se produzcan estos bucles.

Por lo tanto, es importante que los programadores utilicen disparadores sólo en los casos necesarios y que documenten adecuadamente las situaciones que implementan mediante disparadores.

3.3.4. Consideraciones de diseño

A menudo hay diversas soluciones válidas para resolver un mismo problema. En general, se debe utilizar la solución que evite hacer trabajo y accesos innecesarios a la BD.

Según el manual de PostgreSQL, cuando se utilizan disparadores para implementar alguna situación y los disparadores pueden ser `BEFORE` o `AFTER`, normalmente se escoge el disparador `BEFORE` por motivos de eficiencia. El disparador `BEFORE` ejecuta las acciones antes que la operación que activa el disparador.

No obstante, hay muchas sutilezas a la hora de decidir qué tipo de disparador es mejor para implementar una situación. Veamos un ejemplo.

Ejemplo

Consideremos que queremos implementar con disparador la restricción siguiente: "El sueldo de un empleado no puede bajar." Disponemos del fragmento de la siguiente tabla `empleados`:

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

En principio, según lo que hemos dicho antes, comprobaríamos la restricción lo antes posible. Por lo tanto, utilizaríamos un disparador `BEFORE / FOR EACH ROW`.

Pero, ¿qué pasaría si la sentencia que activa el disparador viola la restricción `CHECK sueldo<50.000.0`? Por ejemplo, la sentencia `UPDATE empleados SET sueldo=60.000.0 WHERE núm_empl=10;`.

Como hemos definido el disparador de tipo `BEFORE`, se comprobaría en primer lugar la restricción de que el sueldo no puede bajar y, después, la restricción de que el sueldo ha de ser inferior a 50.000. Sin embargo, por motivos de eficiencia, quizás habría casos en los que sería mejor comprobar primero la restricción de que el sueldo tiene que ser inferior a 50.000. Por ejemplo, si esta restricción se viola muy a menudo en la BD, sería más eficiente comprobar en primer lugar la restricción `check sueldo<50.000,0`. En este caso, podríamos decidir definir nuestro disparador como disparador del tipo `AFTER / FOR EACH ROW`.

Por lo tanto, hay que ir con cuidado a la hora de escoger el tipo de disparador para implementar una determinada situación o semántica.

Resumen

En este módulo didáctico hemos completado el estudio de los diversos componentes lógicos de una BD: los procedimientos almacenados y los disparadores. Para cada uno de estos componentes, hemos detallado su definición utilizando un SGBD concreto, PostgreSQL.

Los componentes lógicos (tablas, vistas, procedimientos almacenados, disparadores, etc.) se organizan dentro de esquemas. El esquema es un componente definido en el SQL estándar que forma parte del entorno SQL. El resto de componentes del entorno SQL definidos en el SQL estándar son el catálogo y el servidor.

En este módulo también hemos comparado las diferencias que hay entre los componentes del entorno SQL definidos en el SQL estándar y los de un SGBD concreto, PostgreSQL.

Finalmente, hemos estudiado cómo se gestionan en el SQL estándar las conexiones a un servidor de BD y las sesiones. Dentro de una sesión, hemos visto las sentencias que permiten utilizar las transacciones.

Actividades

Además de las actividades que habéis encontrado a lo largo del módulo, os proponemos las siguientes:

1. Para cada uno de los elementos que hemos presentado en este módulo, consultad los manuales de PostgreSQL con el fin de ampliar la información.

2. Probad en PostgreSQL todos los ejemplos que se proponen en el módulo y también los ejercicios de autoevaluación.

Ejercicios de autoevaluación

1. Determinad si las afirmaciones siguientes son verdaderas o falsas y argumentad la respuesta brevemente.

a) Según el SQL estándar, una BD se crea con la sentencia `CREATE DATABASE nombre_bd`.
b) El esquema de información contiene información sobre las tablas y otros componentes lógicos de los esquemas de los usuarios.
c) La sentencia `SET SEARCH_PATH` de PostgreSQL sirve para definir el esquema de trabajo en el que se crearán los objetos del usuario.
d) Según el SQL estándar, cuando se inicia una sesión, siempre hay que explicitar el inicio de una transacción con la sentencia `SET TRANSACTION <modo_transacción>`.

2. A partir de estas tablas:

```
CREATE TABLE socios (
  nsocio char(10) primary key,
  sexo char(1) not null,
    check (sexo='M' or sexo='H'));
CREATE TABLE clubs (
  nclub char(20) primary key);
CREATE TABLE socios_clubs (
  nsocio char(10) not null references socios,
  nclub char(20) not null references clubs,
primary key(nsoci, nclub));
CREATE TABLE clubs_con_mas_de_5_socios (
  nclub char(20) primary key references clubs);
```

y de las restricciones de integridad siguientes:

```
RI1. Un club no puede tener más de veinte socios.
RI2. Un club ha de tener más mujeres que hombres.
```

implementad un procedimiento almacenado llamado `AsignarIndividual` que, dado un socio y un club, inserte la asignación en la tabla `socios_clubs`. Además, si el club pasa a tener más de cinco socios, lo tiene que dar de alta en `clubs_con_más_de_5_socios`. El procedimiento ha de informar de los errores por medio de excepciones y proporcionar los mensajes de error siguientes:

```
'Socio ya asignado a este club'
'El socio o el club no existen'
'El club tiene más de veinte socios'
'El club tiene menos mujeres que hombres'
'Error interno'
```

3. A partir de las tablas creadas con las sentencias siguientes, definid un disparador que implemente la regla de negocio: "Cuando la modificación de las existencias de un producto lo deje por debajo del punto de pedido, hay que insertar una petición pendiente, si no se había hecho previamente".

```
CREATE TABLE Productos(
  nProd INTEGER,
  existencias INTEGER NOT NULL,
  puntPedido INTEGER NOT NULL,
  cant_a_pedir INTEGER NOT NULL,
  PRIMARY KEY(nProd));

CREATE TABLE Peticiones_Pendientes(
  nProd INTEGER,
  cant INTEGER NOT NULL,
  fecha DATE,
  PRIMARY KEY (nProd));
```

Solucionario

Ejercicios de autoevaluación

1.

a) Falso. La sentencia `CREATE DATABASE` no existe en el SQL estándar. En cambio, sí que está en muchos SGBD comerciales, como por ejemplo en PostgreSQL.

b) Verdadero. El esquema de información está formado por un conjunto de vistas que contienen metainformació sobre los componentes lógicos de los esquemas de los usuarios.

c) Verdadero. Además, esta sentencia define los esquemas en los que se buscarán los objetos que intervienen en las sentencias SQL.

d) Falso. Normalmente, cuando se inicia una sesión, se empieza una transacción de forma automática.

2.

```
CREATE OR REPLACE FUNCTION asignar_individual
    (socio socios.nsocio%type, club clubs.nclub%type)
    RETURNS void AS $$
DECLARE
  mujeres integer default 0;
  hombres integer default 0;
  sexo_socio char(1);
BEGIN
  hombres= (SELECT COUNT(*)
    FROM socios s, socios_clubs c
    WHERE s.nsocio=c.nsocio AND s.sexo='H'
    AND c.nclub=club);
  mujeres= (SELECT COUNT(*)
    FROM socios s, socios_clubs c
    WHERE s.nsocio=c.nsocio AND s.sexo='M'
    AND c.nclub=club);
  IF (hombres + mujeres + 1 ) > 20 THEN
  RAISE EXCEPTION 'Un club no puede tener mas de 20 socios';
  ELSE
    Sexo_socio =
    (SELECT sexo FROM socios WHERE nsocio=socio);
    IF sexo_socio ='H' AND (hombres >= mujeres) THEN
      RAISE EXCEPTION
      'Un club ha de tener mas mujeres que hombres';
  END IF;
END IF;
INSERT into socios_clubs VALUES(socio,club);
  IF (hombres+mujeres+1>5) and
        (not exists
        (SELECT *
        FROM clubs_con_mas_de_5_socios
        WHERE nclub=club)) THEN
  INSERT into clubs_con_mas_de_5_socios VALUES(club);
END IF;
EXCEPTION
  WHEN raise_exception THEN
    RAISE EXCEPTION '%', SQLERRM;
  WHEN unique_violation THEN
    RAISE EXCEPTION 'Socio ya asignado a este club';
  WHEN foreign_key_violation THEN
    RAISE EXCEPTION 'El socio o el club no existen';
  WHEN OTHERS THEN
    RAISE EXCEPTION 'Error interno';
END;
$$LANGUAGE plpgsql;
```

Fijaos en que el procedimiento almacenado `assignar_individual` no devuelve ningún resultado. Si el procedimiento acaba con éxito, se inserta una fila en la tabla `socios_clubs` y, si hace falta, el club en la tabla `clubs_con_mas_de_5_socios`. Si el procedimiento acaba con error, se genera una excepción.

El tratamiento de excepciones incluye los casos `unique_violation` y `foreign_key_violation`. El primer caso se utiliza cuando el socio que se pasa como parámetro al procedimiento ya ha sido asignado al club que también se pasa como parámetro; es decir, cuando se viola la clave primaria de la tabla `socios_clubs`. El segundo caso se utiliza

cuando el socio o el club que se quieren insertar en la tabla `socios_clubs` no existen; es decir, cuando se viola la restricción de integridad referencial con la tabla `socios` o `clubs`.

Otra forma de comprobar estos dos errores sería hacer consultas en el procedimiento almacenado como las siguientes:

```
IF (SELECT count(*) FROM socios_clubs
    WHERE nsocio=socio AND nclub=club)>0) THEN
    RAISE EXCEPTION 'Socio ya asignado a este club'

IF (SELECT count(*) FROM socios WHERE nsocio=socio)>0)
THEN
RAISE EXCEPTION 'El socio o el club no existen'

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

Estas consultas son más ineficientes que la solución que proponemos, puesto que son consultas adicionales que se han de realizar contra la BD. Como estos dos errores se pueden comprobar mirando si se violan o no las restricciones de integridad de la BD, no hace falta hacer consultas adicionales para comprobarlas.

3.

```
CREATE or REPLACE FUNCTION poner_peticion()
RETURNS trigger AS $$
BEGIN
    IF (NEW.existencias<NEW.puntPedido) THEN
      IF ((SELECT count(*) FROM peticiones_pendientes
        WHERE nprod=NEW.nprod)=0) THEN
INSERT INTO peticiones_pendientes
      values(NEW.nprod,NEW.cant_a_pedir,current_date);
      END IF;
    END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER actualiza_existencias
AFTER UPDATE OF existencias ON Productos
FOR EACH ROW EXECUTE PROCEDURE poner_peticion();
```

El disparador `actualiza_existencias` es un disparador definido `AFTER`, porque sólo hemos de insertar una petición pendiente cuando se produce una modificación de las existencias de algún producto. Si lo definiéramos `BEFORE`, podría suceder que la sentencia de modificación sobre la tabla de productos fallara, y nosotros ya habríamos insertado la petición pendiente en la tabla `peticiones_pendientes`.

El disparador `actualiza_existencias` se define `FOR EACH ROW`, puesto que queremos insertar una petición pendiente por cada producto del cual se modifiquen las existencias (si las existencias son inferiores al punto de pedido). Si lo definiéramos `FOR EACH STATEMENT`, sólo insertaríamos una petición pendiente en la tabla `peticiones_pendientes`, independientemente de que se modificaran las existencias de un producto o de cien.

Glosario

catálogo *m* Componente del entorno SQL que contiene un conjunto de esquemas, uno de los cuales es el esquema de información, que recoge toda la información sobre los esquemas de los usuarios (nombres de tablas, columnas, restricciones, definiciones de vistas, etc.).

conexión *f* Asociación que se crea entre un cliente y un servidor.

disparador *m* Acción o procedimiento almacenado que se ejecuta automáticamente cuando se ejecuta una operación de inserción, de borrado o de modificación sobre alguna tabla de la base de datos.

esquema *m* Elemento que agrupa un conjunto de componentes lógicos (tablas, vistas, procedimientos almacenados, etc.).

PL/PgSQL *m* Lenguaje de PostgreSQL equivalente al PSM definido en el SQL estándar. Nota: La sigla PSM corresponde a la denominación inglesa *persistent stored module*.

procedimiento almacenado *m* Acción o función definida por un usuario que proporciona un servicio determinado. Una vez creada, se guarda en la base de datos y se trata como un objeto más de ésta.

servidor *m* Elemento superior de la jerarquía de componentes del entorno SQL que contiene un conjunto de catálogos.

sesión *f* Conjunto de sentencias SQL que se ejecutan mientras hay una conexión activa a un servidor.

transacción *f* Conjunto de peticiones SQL de lectura o actualización de la base de datos que confirma o cancela los cambios que ha llevado a cabo.

Bibliografía

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. Manuales accesibles en línea en: http://www.PostgreSQL.org/docs/.