Programación con bases de datos

PID_00261968

Ignasi Lorente Puchades

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Ignasi Lorente Puchades

Ingeniero superior en Informática por la Universitat Oberta de Catalunya (UOC). Ejerce de jefe de proyectos de aplicaciones web en el ámbito público y privado. Es profesor colaborador del grado de Multimedia en la Universitat Oberta de Catalunya (UOC) y profesor asociado en la Universidad Pompeu Fabra (UPF).

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por la profesora: Àngels Rius Gavidia (2019)

Primera edición: febrero 2019

Avda. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Realización editorial: Oberta UOC Publishing, SL

Índice

Introducción

En este módulo didáctico aprenderemos a acceder a una base de datos relacional desde un programa de aplicación mediante el lenguaje estándar de acceso a bases de datos relacionales, el lenguaje SQL (del inglés *structured query language*). Concretamente veremos cómo acceder a la base de datos desde el entorno web.

Para ampliar nuestros conocimientos sobre SQL empezaremos viendo qué sentencias SQL permiten describir la base de datos y definir sus usuarios, asignar permisos de acceso y hacer consultas. También veremos cuáles son las diferentes estructuras de programación que el lenguaje SQL ofrece al desarrollador para facilitar el acceso a datos desde programa. Y finalmente, de qué manera se ejecutan estas operaciones, teniendo en cuenta la problemática de concurrencia que puede surgir cuando múltiples usuarios acceden a los mismos datos simultáneamente.

Antes de empezar a estudiar el lenguaje SQL con más profundidad, conviene tener clara la notación que utilizaremos:

- Toda sentencia SQL termina con punto y coma.
- Cada sentencia puede estar formada por una o más cláusulas.
- Hay cláusulas opcionales y obligatorias.
- Una cláusula opcional se indicará entre corchetes *[...]*.
- Una cláusula que permita elegir entre varias opciones se escribe separando cada opción por una barra vertical. Por ejemplo *A*|*B*|*C*, donde *A*, *B* y *C* son opciones.
- Las palabras reservadas se escriben en mayúscula.
- Entre símbolos menor (<) y mayor (>) se indican los valores que debe informar el usuario.

Objetivos

En los materiales didácticos de esta unidad encontraréis las herramientas indispensables para lograr los objetivos siguientes:

1. Entender la estructura de organización de una base de datos.

2. Comprender la necesidad de generar código de aplicación para acceder a la bases de datos desde un entorno web, teniendo en cuenta el sistema de gestor de bases de datos y el entorno de programación elegidos.

3. Conocer los mecanismos que ofrece una base de datos para encapsular datos con vistas, permitiendo el acceso a los datos de manera análoga a las tablas.

4. Conocer los mecanismos que ofrece una base de datos para combinar diferentes operaciones SQL: disparadores y procedimientos almacenados.

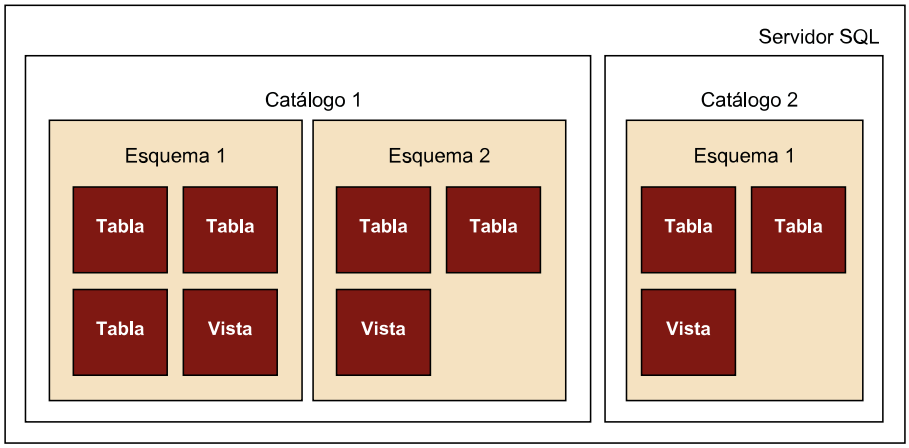
5. Saber cómo se gestionan las transacciones y cómo se facilita la concurrencia de acceso a los datos desde un SGBD.

1. Estructura de un SGBD relacional

Hasta ahora hemos visto que en una base de datos relacional los elementos que permiten almacenar la información son las tablas. Aún así, debemos tener en cuenta que un SGBD podrá gestionar más de una base de datos y que habrá que ver cómo se organizarán las tablas y las vistas dentro de un SGBD para diferenciar distintas tablas y vistas de bases de datos.

Un SGBD que soporte SQL estará formado por una serie de elementos estructurados de manera jerárquica:

- Servidor: sistema informático que gestionará los diferentes catálogos de datos.
- Catálogo: componente que contiene un conjunto de esquemas de bases de datos.
- Esquema: componente que permite agrupar un conjunto de tablas, vistas y otros elementos del modelo lógico de una base de datos que son propiedad de un usuario.



De acuerdo con esto, vemos que las bases de datos vienen descritas por sus esquemas y que varios esquemas pueden formar parte de un mismo catálogo. Estos catálogos son propiedad de algún usuario del servidor, que puede dar acceso a otros usuarios. No obstante, hay un usuario que gestiona el conjunto de catálogos y que tiene el control total de la información contenida en el servidor: el administrador de la base de datos.

Si bien el lenguaje SQL ofrece sentencias para la creación de la base de datos, sus tablas y vistas, cada uno de los diferentes SGBD tiene acciones propias para la creación y gestión del servidor, así como de los catálogos.

1.1. Usuarios de la base de datos

En una base de datos hay diferentes roles y usuarios cuyos privilegios permiten el acceso a la estructura de la base de datos. Por otro lado, los usuarios tienen acceso a los datos en función de los permisos que tengan. Así pues, la visibilidad de los diferentes objetos de la base de datos dependerá de estos permisos.

El **usuario administrador** (DBA, en inglés) es el usuario que gestionará los diferentes elementos de la base de datos, así como los otros usuarios.

Este rol será de vital importancia tanto para la instalación de la base de datos como para su mantenimiento en el día a día, ya sea controlando el espacio de almacenamiento o la modificación de la estructura de la base de datos, a criterio de los desarrolladores.

Será el usuario administrador el encargado de hacer copias de seguridad y, si es necesario, restaurar una versión estable y generar los informes necesarios para tener siempre informado al propietario de la base de datos.

No estará entre sus roles establecer el diseño de la base de datos, que será una de las atribuciones del usuario desarrollador, puesto que es quien conoce los requisitos funcionales de la aplicación a desarrollar.

El **usuario desarrollador** de la base de datos es el usuario que define la estructura de la base de datos, de acuerdo con los requisitos de la aplicación a la que da servicio.

Además, el usuario administrador será el encargado de dotar o de revocar los permisos (GRANTS) a los diferentes usuarios de la base de datos.

Esta es la sentencia de asignación de privilegios:

```
GRANT <privilegio>
ON <objeto>
TO {<user> | PUBLIC | <rol>}
[ WITH GRANT OPTION ];
```

Veamos algunos de los privilegios que se pueden asignar a los usuarios de la base de datos:

- **CREATE object**: permite a los usuarios crear o modificar el objeto especificado en su esquema.
- **CREATE ANY object**: permite a los usuarios crear o modificar el objeto especificado en cualquier esquema.
- **INSERT**: permite a los usuarios insertar filas en una tabla.
- **SELECT**: permite a los usuarios seleccionar datos de un objeto de base de datos.
- **UPDATE**: permite al usuario actualizar el contenido de las columnas de las filas de una tabla.
- **DELETE**: permite eliminar filas de una base de una tabla.
- **EXECUTE**: permite al usuario llamar procedimientos almacenados o funciones.

Privilegio CREATE

Este privilegio permitirá al usuario o rol especificado ejecutar sentencias CREATE, ALTER o DROP sobre objetos de la base de datos.

Para revocar privilegios, utilizaremos la sentencia REVOKE, con la sintaxis siguiente:

```
REVOKE <privilegio>
ON <objeto>
FROM {<user> | PUBLIC | <rol>};
```

Aun así, será el usuario administrador el encargado de crear y de mantener los diferentes roles de la base de datos, así como los roles asignados a cada usuario.

La sintaxis para crear roles será:

```
CREATE ROLE <rol>
[IDENTIFIED BY <password>];
```

Por ejemplo, para crear un rol llamado «desarrollador» con la contraseña como «pwd», el código será el siguiente:

```
CREATE ROLE desarrollador
IDENTIFIED BY pwd;
```

A continuación, si se quiere asignar el privilegio CREATE TABLE al rol «desarrollador», y asignar este rol a un usuario «user1», utilizaremos el código siguiente:

```
GRANT CREATE TABLE TO desarrollador;
GRANT desarrollador TO user1;
```

1.2. Los esquemas de una base de datos

Un esquema es la descripción de un objeto de base de datos que nos permite definir su propia estructura; es decir, tablas, vistas, índices, tipos, secuencias y reglas de integridad para asegurar la coherencia de los datos almacenados, entre otros objetos que la componen.

Para crear un esquema, utilizaremos la sentencia CREATE SCHEMA, que tiene la siguiente sintaxis:

```
CREATE SCHEMA <schema_name> [AUTHORIZATION <owner_name>]
```

```
[DEFAULT CHARACTER SET <char_set_name>]
[PATH <schema_name[, ...]>]
[ ANSI CREATE <statements [...]> ]
[ ANSI GRANT <statements [...]> ];
```

Como vemos en esta instrucción, se podrá indicar en diferentes cláusulas: el nombre del propietario del esquema (owner_name), la codificación de los caracteres con que se trabajará (char_set_name) —en el caso de que se desee otra diferente de la instalada en el servidor— y el fichero físico donde se almacenará la información del esquema (schema_name). También será posible indicar qué componentes (tablas, vistas, índices...) se desean crear; así como asignar permisos a diferentes usuarios de la base de datos para leer, escribir, etc.

Finalmente, con la cláusula GRANT podremos asignar permisos de lectura, de escritura, de acceso, entre otros, a los diferentes usuarios de la base de datos.

En la creación del esquema es importante darse cuenta de que hace falta que se informen las sentencias de creación de objetos de la base de datos, como las tablas, las vistas, etc. De este modo, la creación y la modificación de los elementos del esquema se podrá hacer *a posteriori*.

Dado que en un catálogo hay más de un esquema de bases de datos, hay que seleccionarlo para operar con los elementos. Esto lo haremos con la cláusula SET SCHEMA.

```
SIETE SCHEMA <schema_name>
```

Por ejemplo, si queremos crear un esquema denominado «UNIVERSIDAD» para almacenar información de estudiantes, podemos ejecutar las sentencias SQL siguientes:

```
CREATE SCHEMA UNIVERSIDAD AUTHORIZATION 'admin';
SET SCHEMA UNIVERSIDAD;
CREATE TABLE Estudiante (
  id INT,
  nombre VARCHAR (255),
  apellidos VARCHAR (255)
);
```

En este ejemplo vemos cómo crear un esquema de nombre «UNIVERSIDAD», que pertenece al usuario «admin». A continuación, se selecciona como esquema de trabajo («UNIVERSIDAD») y, finalmente, se crea una tabla («ESTUDIANTE»).

El esquema, como cualquier otro elemento de SGBD descrito por alguna sentencia de definición en SQL, no es inalterable y se podrá modificar y borrar (siempre que el usuario tenga permisos) con las sentencias ALTER SCHEMA y DROP SCHEMA.

En el caso de la modificación del esquema, la sintaxis será:

```
ALTER <schema_name> SCHEMA [RENAME TO <new_schema_name>] [OWNER TO <new_user_name>];
```

Como vemos, con la sentencia ALTER SCHEMA, podremos modificar tanto el nombre del esquema como el del propietario del mismo.

Para eliminar un esquema, la sintaxis será:

```
DROP SCHEMA <schema name> [RESTRICT | CASCADE};
```

En este caso, encontraremos dos opciones, RESTRICT y CASCADE, que nos impedirán, respectivamente, eliminar el esquema solo si está vacío o eliminarlo en cualquier caso, destruyendo también su contenido.

Hemos visto que un esquema tiene un nombre, un propietario, unos usuarios con permisos para acceder, unas tablas, etc. Esta información se guardará en una estructura de datos aparte del esquema de la base de datos, que contendrá solo datos orientados a la gestión.

Así pues, aparte de los esquemas creados por los diferentes usuarios del servidor SQL que serán accesibles a los usuarios que tengan permiso para ello, es necesario un esquema que sirva para organizar y mantener esta metainformación, la cual será solo accesible al administrador de la base de datos.

Estas sentencias nos servirán para conectar con un servidor y un esquema, desde la línea de comandos, para poder ejecutar operaciones. Cada SGBD tendrá una implementación diferente de las sentencias de creación y de conexión en un esquema propio de un servidor.

Los diferentes lenguajes de programación de alto nivel nos ofrecerán idealmente una única interfaz que permitirá acceder y ejecutar sentencias en una base de datos con independencia del SGBD.

Para lograr este objetivo, habitualmente cada lenguaje de programación proporcionará unos conectores (*drivers*) que permitirán que estas tareas sean transparentes para el desarrollador. Hay otros lenguajes de programación, como el PHP, que tendrán una biblioteca propia para cada base de datos. Sin embargo, el lenguaje PHP también ofrece una biblioteca propia, denominada PDO (PHP data object), que permite conectar con cualquier base de datos a partir de un conector.

Habitualmente, la conexión a una base de datos se hace mediante un URL, en el cual se especifica el protocolo de conexión, el SGBD al que se quiere conectar, la dirección IP (o nombre de dominio), el puerto de acceso y el nombre del esquema o de la base de datos con que se quiere conectar. En este URL se puede indicar, también, las credenciales de acceso del usuario.

La URL será de la forma

```
jdbc:[<subprotocol>]:[<node>]/[<databaseName>]
```

Donde el subprotocolo hará referencia al tipo de SGBD o especificación (Oracle, MySQL, ODBC), y el nodo hará referencia a la dirección del servidor.

Por ejemplo, si queremos conectar a una base de datos Oracle desde Java, ejecutaremos una instrucción del estilo:

```
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","user","password");
```

En caso de que se desee conectar desde Java con una base de datos MySQL, la instrucción será del estilo:

```
Connection con = DriverManager.getConnection
("jdbc:mysql://localhost/mydb?user=user&password=password");
```

Si, por el contrario, queremos conectar a una base de datos Oracle desde PHP, ejecutaremos una instrucción del estilo:

```
$con = oci_connect('user', 'password', 'localhost/xe');
```

En caso de que se desee conectar desde PHP con una base de datos MySQL, la instrucción será del estilo:

```
$con = mysqli_connect("localhost", "user", "password", "mydb");
```

En caso de que se desee conectar desde PHP a partir de un conector, la instrucción será del estilo:

```
$con = new PDO('mysql:dbname=mydb;host=localhost', 'user', 'password');
```

Una vez conectados a la base de datos, un lenguaje de programación de alto nivel permitirá realizar operaciones simples guardando la salida de cada operación para utilizarla como parámetro de la operación siguiente o, incluso, realizar otros tipos de operaciones sobre el resultado con el fin de utilizarlo como entrada de otras operaciones.

Así pues, las instrucciones que permitirán el control del flujo en los lenguajes de programación posibilitarán la concatenación de sentencias SQL en bucles, sentencias condicionales, etc.

2. Estructuras de programación en SQL

SQL nos ofrece diferentes formas de encapsular los datos para facilitar su acceso, ya sea mediante la generación de subconjuntos de datos u otros mecanismos que permitan la combinación y concatenación de sentencias de manera transparente para el desarrollador.

Primero veremos vistas, un mecanismo que facilita el acceso a subconjuntos de datos como si estuvieran almacenados en una tabla. Después, los procedimientos almacenados y/o disparadores, también conocidos estos últimos como *triggers*. Ambos son mecanismos de programación que utilizan estructures algorítmicas para el control de flujo. Difieren en el mecanismo que desencadena su ejecución, bajo demanda del usuario o de forma autónoma, si se producen ciertas circunstancias especificadas por el programador.

Como la sintaxis de las sentencias SQL para las vistas, las funciones y los procedimientos almacenados depende de cada sistema gestor de base de datos, tendremos que elegir uno para ir estudiando y probando los ejemplos.

Nuestro SGBD de referencia será MySQL y la base de datos de la que partiremos e iremos ampliando a medida que lo necesitemos será la base de datos BD_ESTUDIANTE, que está formada por las siguientes tablas:

- ESTUDIANTE (codigoMatricula, nombre, apellidos, estudios, porcentaje)
- ESTUDIANTE_DE_GRADO (codigoMatricula, nombre, apellidos, porcentaje)

La extensión de ambas tablas es:

ESTUDIANTE

codigo-Matricula	nombre	apellidos	estudios	porcentaje	notaLetra
1	Joan	Pi Dot	Grado	0	NULL
2	Laura	Sentís Aguilar	Máster	0	NULL
3	Roc	Sánchez Gómez	Grado	0	NULL
4	Joana	Sauler Sunyer	Grado	0	NULL

ESTUDIANTES_DE_GRADO

codigo-Matricula	nombre	apellidos	porcentaje	notaFinal
1	Joan	Pi Dot	0	9
3	Roc	Sánchez Gómez	0	8
4	Joana	Sauler Sunyer	0	7

Y las instrucciones del DDL de creación y del DML para insertar los datos son las siguientes:

```
CREATE TABLE ESTUDIANTE
(
codigoMatricula int,
nombre varchar(50),
apellidos varchar(50),
estudios varchar(50),
porcentaje numérico(5,2),
notaLetra char(2),
primary key (codigoMatricula)
);

INSERT INTO ESTUDIANTE VALUES (1,"Joan","Pi Dot","Grado", 0, null);
INSERT INTO ESTUDIANTE VALUES (2,"Laura","Sentís Aguilar","Máster", 0, null);
INSERT INTO ESTUDIANTE VALUES (3,"Roc","Sánchez Gómez","Grado", 0, null);
INSERT INTO ESTUDIANTE VALUES (4,"Joana","Sauler Sunyer","Grado", 0, null);

CREATE TABLE ESTUDIANTE_DE_GRADO AS
SELECT codigoMatricula, nombre, apellidos, porcentaje
FROM ESTUDIANTE
WHERE estudios = "Grado";

ALTER TABLE ESTUDIANTE_DE_GRADO ADD notaFinal integer;
UPDATE ESTUDIANTE_DE_GRADO SET notaFinal = 9 WHERE codigoMatricula = 1;
UPDATE ESTUDIANTE_DE_GRADO SET notaFinal = 8 WHERE codigoMatricula = 3;
UPDATE ESTUDIANTE_DE_GRADO SET notaFinal = 7 WHERE codigoMatricula = 4;
```

2.1. Vistas en SQL

Hasta ahora hemos considerado que el acceso a la información almacenada en una base de datos se hace a partir de consultas contra las tablas que la forman.

Sin embargo, existe una estructura, llamada *vista*, que nos permitirá acceder a la información presente en una o más tablas como si se encontrara agrupada en una tabla virtual.

Una **vista** es un conjunto de datos, resultado de una consulta, al que se puede acceder del mismo modo que a una tabla.

Las vistas no serán parte del esquema físico de la base de datos, y su contenido se calculará siempre en el momento de acceder a ellas. Esto significa que no existen realmente como conjuntos de valores almacenados en la BD, sino que se derivan de la información existente en la base de datos por lo que pueden considerarse como tablas ficticias (no materializadas). La no existencia física de las vistas hace que no siempre puedan ser actualizables.

Cuando se actualiza una tabla, ya sea porque se insertan filas o modifican valores de las columnas, la nueva información debe reflejarse en las tablas que forman parte de la consulta que se utiliza para definirla.

Por este motivo, solo se podrán actualizar las vistas que cumplan las siguientes condiciones:

- La vista tiene que incluir todas las claves primarias de las tablas que se utilizan en la consulta que la define.

- La vista debe incluir todos los campos, que no pueden tener valor nulo, de las tablas utilizadas en la consulta que la define.

En este caso es posible hacer lo que llamamos *reverse-mapping*, un mapeo entre vista-tablas y tablas-vista, es decir, un mapeo en ambos sentidos. En caso contrario, no se podrá llevar a cabo la operación, ya que se violarían las reglas de integridad del modelo relacional.

2.1.1. Sintaxis

Para crear una vista, utilizaremos la sintaxis básica siguiente:

```
CREATE VIEW <nombre_vista> [(<columnas>)]
AS <expresion_consulta>
[WITH CHECK OPTION];
```

La expresión de consulta será una sentencia SELECT, que se puede hacer sobre una o más tablas.

A partir de la base de datos de ejemplo BD_ESTUDIANTE, se quiere crear una vista para obtener las notas cuantitativas y cualitativas de los estudiantes de grado, puesto que se consulta frecuentemente.

```
CREATE VIEW NotasEstudiantes (matrícula, nota, calificación) as
(
SELECT e.codigoMatricula, eg.notaFinal, e.notaLetra
FROM ESTUDIANTE e INNER JOIN ESTUDIANTE_DE_GRADO eg
ON e.codigoMatricula = eg.codigoMatricula
```

```
);
```

Observamos que es una vista que accede a columnas de diferentes tablas.

Si la ejecutamos mediante esta consulta SQL:

```
SELECT * FROM NotasEstudiantes;
```

Obtendríamos la extensión siguiente:

EXTENSIÓN DE LA VISTA:

matrícula	nota	calificación
1	9	Null
3	8	Null
4	7	Null

Para borrar una vista, utilizaremos la sentencia DROP:

```
DROP VIEW <nombre_vista> {RESTRICT|CASCADE};
```

Si utilizamos la opción RESTRICT, la vista no se borrará si está referenciada, por ejemplo, por otra vista. En cambio, si ponemos la opción CASCADE, todo lo que referencie la vista se borrará.

Si quisiéramos borrar la vista que hemos creado, deberíamos ejecutar:

```
DROP VIEW NotasEstudiantes;
```

Los resultados de las vistas pueden actualizar las tablas involucradas siempre y cuando la vista contenga la clave primaria, pero, para hacerlo, hay que definir la vista con la cláusula WITH CHECK OPTION.

Creamos la vista actualizable que devuelve las notas de los estudiantes de grado.

```
CREATE VIEW NotasEstudiantes (matrícula, nota, calificación) as
(
SELECT e.codigoMatricula, eg.notaFinal, e.notaLetra
FROM ESTUDIANTE e INNER JOIN ESTUDIANTE_DE_GRADO eg
WHERE e.codigoMatricula = eg.codigoMatricula
) WITH CHECK OPTION;
```

Imaginemos que queremos modificar la nota de un estudiante de grado del que conocemos el número de matrícula. Para hacerlo, ejecutaríamos las sentencias siguientes:

```
UPDATE NotasEstudiantes
SET nota = 8
WHERE matrícula = 1;
```

```
SELECT * FROM NotasEstudiantes;
```

La extensión de la vista se vería modificada así:

EXTENSIÓN DE LA VISTA:

matrícula	nota	calificación
1	8	Null
3	8	Null
4	7	Null

Finalmente, si se quiere modificar la estructura de una vista, utilizaremos la sentencia ALTER VIEW:

```
ALTER <nombre_vista> [(columnas)]
AS <expresion_consulta>
[WITH [RESTRICT|CASCADE] CHECK OPTION];
```

2.1.2. Ventajas en el uso de las vistas

Las vistas pueden proporcionar ciertas ventajas sobre las tablas:

- Pueden representar un subconjunto de los datos contenidos en una tabla, limitando el grado de exposición de los datos y permitiendo que un usuario determinado tenga permiso para consultar la vista, mientras que se deniega el acceso al resto de la tabla base.

- Permiten unir y simplificar múltiples tablas en una única tabla virtual. Además, permiten crear subconjuntos que simplifican la complejidad de los datos.

- Pueden actuar como tablas agregadas, en las cuales el motor de la base de datos agrega datos (suma, media, etc.) y presenta los resultados calculados como parte de los datos.

- Ocupan muy poco espacio de disco, de forma que la base de datos solo contiene la definición de una vista y no una copia de todos los datos que presenta.

2.2. Los procedimientos almacenados

Los procedimientos almacenados permitirán aplicar la metodología algorítmica en las consultas realizadas sobre una base de datos.

A pesar de que hemos visto que es posible realizar algunas consultas complejas combinando sentencias SQL y/o utilizando subconsultas, existen ciertas limitaciones para combinar sentencias SQL en una misma sentencia. Por ejemplo, podemos hacer inserciones de datos a partir de datos recuperados en una consulta, pero no crear una tabla cuyos nombres de columna provengan de los valores recuperados en una consulta. Para hacerlo, necesitamos un paso intermedio para almacenar dichos valores en una variable que posteriormente pueda ser consultada.

Para combinar distintas sentencias SQL como si fuera una sola, podemos utilizar los procedimientos almacenados.

Un **procedimiento almacenado** es una función definida por un usuario de la base de datos que proporciona un servicio determinado. El procedimiento se almacenará en la base de datos y se tratará como un objeto más.

Los procedimientos almacenados se podrán ejecutar bajo petición expresa de un usuario desde la línea de comandos, o bien desde una aplicación que acceda a la base de datos.

El uso de procedimientos almacenados nos permitirá agrupar operaciones que tienen un objetivo algorítmico común y, de este modo, simplificar el desarrollo de aplicaciones. Además, dado que funciona como una caja negra, el desarrollador que invoque el procedimiento no necesitará conocer qué operaciones se llevan a cabo dentro del procedimiento.

Otra ventaja de los procedimientos almacenados es que se guardan precompilados en la base de datos por lo que se mejora el rendimiento de la aplicación que los invoca.

2.2.1. Sintaxis

Para crear un procedimiento, utilizaremos la sintaxis básica siguiente:

```
CREATE {PROCEDURE | FUNCTION} <nombre_procedimiento>
AS
<sentencias_sql>
END;
```

Y esta será la manera como se invocará:

```
CALL <nombre_procedimiento>;
```

Crearemos un procedimiento que permita crear una tabla con los datos de los estudiantes de grado, concretamente su nombre y apellidos.

```
CREATE TABLE
ESTUDIANTES_DE_GRADO AS SELECT
nombre, apellidos FROM ESTUDIANTES
WHERE tipoEstudio = 'Grado';
```

Observemos que esta consulta nos permite crear la nueva tabla, a partir de valores de columnas de otra tabla. Sin embargo, no podríamos crear una tabla con un número de columnas variable o con nombres de columna obtenidos a partir de la tabla origen.

Si se quiere eliminar el procedimiento, utilizaremos la sentencia DROP:

```
DROP PROCEDURE <nombre_procedimiento>;
```

Finalmente, si se quiere modificar un procedimiento, tendremos la opción de eliminar el procedimiento para crearlo de nuevo, o bien utilizar la sentencia ALTER:

```
ALTER {PROCEDURE | FUNCTION} <nombre_procedimiento>
[( {<parameter_name> <datatype> }[, ...] )]
[NAME <new_object_name>]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL FECHA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT][DYNAMIC RESULT SETS int]
[CASCADE | RESTRICT]
```

Aún así, deberemos tener en cuenta que los procesos almacenados tendrán una sintaxis final diferente para cada SGBD de forma que, al cambiar de una base de datos a otra, se deberán reescribir para adaptarlos a la nueva sintaxis.

Diferenciaremos los procedimientos de las funciones porque, como en otros lenguajes de alto nivel, las funciones devolverán un único valor, mientras que los procedimientos no.

Cada SGBD, además, tendrá una serie de características de diferente complejidad, de forma que nos permitirá realizar operaciones más o menos sofisticadas con más o menos código.

La sentencia completa será:

```
CREATE {PROCEDURE | FUNCTION} <proc_name>
([{[IN | OUT | INOUT] [<parameter_name>] <datatype> [AS LOCATOR] [RESULT]} [, ...]] )
[ RETURNS <datatype> [AS LOCATOR]
LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}
```

```
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS int]<sentencias_sql>
```

2.2.2. Parámetros de entrada y de salida

Como podemos ver en la definición, los procedimientos almacenados podrán aceptar parámetros que, a su vez, podrán ser de entrada de valores, de regreso de un resultado o tener una función doble y ser parámetros tanto de entrada como de salida.

Cada parámetro tendrá un nombre y se le asociará un tipo de dato. Los tipos de datos de los parámetros de entrada deberán coincidir con los utilizados en la definición de columnas de tablas.

Será posible, además, definir un valor por defecto para los parámetros con la sentencia DECLARE PARAMETER.

Si queremos devolver un único valor, estaremos hablando de una función, en lugar de un procedimiento, e indicaremos el tipo de dato de regreso mediante la cláusula RETURNS.

Vamos a crear un procedimiento almacenado para calcular el porcentaje de la nota de un estudiante de grado a partir de su código de matrícula y sabiendo que, por defecto, cursa seis asignaturas.

```
CREATE PROCEDURE ValoracionEstudiantes (in estudiante int, in nota int)
begin
DECLARE numAssignatures int default 6;
UPDATE ESTUDIANTE_DE_GRADO
SET porcentaje = nota/numAssignatures
WHERE codigoMatricula = estudiante;
end;
```

Para invocarlo, lo haríamos así:

```
CALL ValoracionEstudiantes (1,8);
```

Podríamos comprobar el resultado de llevar a cabo este procedimiento ejecutando la sentencia:

```
SELECT * FROM estudiante_de_grado;
```

codigo-Matricula	nombre	apellidos	porcentaje	notaFinal
1	Joan	Pi Dot	1,33	8
3	Roc	Sánchez Gómez	0	8
4	Joana	Sauler Sunyer	0	7

2.2.3. Variables dentro del código

Una de las características principales de los procedimientos es que nos permitirán usar variables, definidas por el usuario, para almacenar información que utilizaremos en operaciones posteriores dentro del código del procedimiento.

Para definir variables dentro de un procedimiento almacenado usaremos la cláusula DECLARE. Esta cláusula servirá para definir variables locales y declarar parámetros de rutinas (procedimientos y funciones) que permitan comunicar cada rutina con los llamamientos correspondientes.

```
DECLARE <var_name> [, <var_name>] ... <type> [NOT NULL] [DEFAULT value]
```

Las variables tendrán un ámbito local y no tendrán visibilidad fuera del procedimiento. Tanto los valores como el espacio reservado para las variables se vaciarán y descartarán una vez haya finalizado la ejecución del procedimiento.

El valor se puede especificar como una expresión y no tiene que ser constante. Si no se especifica un valor para la cláusula DEFAULT, el valor inicial será NULL.

En caso de que se asigne un valor NULL a una variable indicada como NOT NULL, la ejecución del procedimiento devolverá un error y se finalizará la ejecución.

Cada variable deberá tener asociado un nombre y un tipo de datos, que serán siempre los mismos tipos de datos que los utilizados para la definición de los campos en la creación de tablas.

Para la asignación de valores a variables encontraremos diferentes posibilidades, como la asignación directa, tanto si es de una constante o el resultado de otro procedimiento, o bien la asignación a partir de la sentencia SELECT ... INTO.

Crearemos una función que devuelva la media de todos los porcentajes de los estudiantes de grado. Tened en cuenta que, al ser una función, se utiliza la cláusula RETURNS y se indica el tipo de dato que devuelve.

```
CREATE FUNCTION MediaPorcentajesEstudiantesGrado() RETURNS numeric(5,2) DETERMINISTIC
BEGIN
DECLARE mediaPorcentajes numeric(5,2);
SELECT AVG(porcentaje) INTO mediaPorcentajes
FROM ESTUDIANTE_DE_GRADO;
RETURN (mediaPorcentajes);
END;
```

Llama la función y el resultado:

```
select MediaPorcentajesEstudiantesGrado();
```

MediaPorcentajesEstudiantesGrado()
0,44

2.2.4. Sentencias condicionales

Como en la mayoría de los lenguajes estructurados, se podrán utilizar sentencias condicionales que nos permitan alterar el flujo de la ejecución del procedimiento.

Para especificar las condiciones, se usará la estructura IF ... ELSE utilizando la sintaxis básica siguiente:

```
IF ( <expresion_booleana> )
BEGIN
  <sentencias_sql>
END
ELSE
BEGIN
  <sentencias_sql>
END
```

Las condiciones IF se podrán anidar como si fueran una expresión SQL simple. No hay límite de número de niveles anidados.

Si la condición evaluada no se satisface y la expresión booleana devuelve FALSE, ejecutará la consulta de declaración SQL de la ELSE.

Al igual que en las cláusulas WHERE, se podrán especificar las condiciones utilizando operadores lógicos AND y OR, u operadores de comparación como, por ejemplo, =, <, >, <=, >=, <>. También se podrán utilizar predicados propios del SQL como BETWEEN, IN, IS NULL, IS NOT NULL o LIKE, así como otras consultas SQL que devuelvan una relación de un único campo de tipo booleano.

Si la estructura IF ... ELSE solo contiene una consulta SQL, no hace falta que se especifique la cláusula BEGIN ... END. En caso contrario, habrá que incluirla obligatoriamente para permitir ejecutar toda la consulta SQL.

Queremos crear una función para obtener la nota de un estudiante de grado en formato cualitativo. A partir del código de matrícula de un estudiante de grado, se obtendrá la nota en formato cualitativo de acuerdo con estas correspondencias: de 0 a 2 le corresponde D; de 3 a 4, C–; de 5 a 6, C+; de 7 a 8, B, y de 9 a 10, A.

```
CREATE FUNCTION CalculoNotaLetra (estudiante integer) RETURNS VARCHAR(2) DETERMINISTIC
BEGIN
   DECLARE notaLetra varchar(2);
   DECLARE quants integer;
   DECLARE nota integer;

   SELECT COUNT(*) INTO quants
   FROM ESTUDIANTE_DE_GRADO
   WHERE codigoMatricula = estudiante;

   IF (quants = 1) THEN
      SELECT notaFinal INTO nota
      FROM ESTUDIANTE_DE_GRADO
```

```
        WHERE codigoMatricula = estudiante;

        IF (nota <=2) THEN
           SET notaLetra = "D";
        ELSE
           IF (nota <=4) THEN
              SET notaLetra = "C-";
           ELSE
              IF (nota <=6) THEN
                 SET notaLletra = "C+";
              ELSE
                 IF (nota <=8) THEN
                    SET notaLetra = "B";
                 ELSE
                    SET notaLetra = "A";
                 END IF;
              END IF;
           END IF;
        END IF;
     ELSE
        SET notaLetra="";
     END IF;
     RETURN (notaLetra);
END;
```

Para invocar la función habría que ejecutar lo siguiente, suponiendo que nos interesa para el estudiante de código 3:

```
SELECT calculoNotaLetra (3);
```

Y el resultado sería:

CalculoNotaLetra(3)
B

2.2.5. Sentencias iterativas y uso de cursores de datos

Antes de ver cuáles son las sentencias iterativas que podemos utilizar cuando creamos un procedimiento o una función para repetir sentencias SQL, habrá que saber qué es un cursor y cómo se utiliza.

Un cursor es el mecanismo que tenemos para solucionar un problema *impedance missmatch*. Este problema se origina porque las sentencias SQL trabajan con conjuntos de tuplas y los programas de aplicación con filas una a una. Por esta razón ha sido necesario inventar un sistema que permitiera recoger todas las filas devueltas por el código SQL y gestionarlas para que el programa las tratase, una por una, mediante un conjunto de iteraciones. La fila que se trata en cada momento se dice que es la que está en la ventana del cursor. Por ejemplo, el cursor puede servir para actualizar (UPDATE) registros o filas con valores diferentes recuperados por una consulta SQL.

En las diferentes implementaciones del estándar SQL podremos encontrar más de un tipo de sentencias que permiten la iteración, como WHILE o FOR. En el primer caso, la expresión condicional permitirá la salida del bucle cuando la condición de iteración no se cumpla; en el segundo caso, en cambio, itera-

remos el conjunto de sentencias SQL tantas veces como se indique en la definición del bucle. Si se conoce el número de iteraciones que hay que hacer, lo más práctico es utilizar la sentencia FOR, cuya sintaxis es:

```
FOR <lista_variables>
IN <expresion_SQL>
   <sentencia_SQL>
END LOOP;
```

Para ver algún ejemplo de utilización de bucle y cómo se usa un cursor, crearemos un procedimiento que actualizará la nota de todos los estudiantes de grado utilizando el procedimiento para calcular la letra correspondiente a la nota y que hemos visto en el apartado anterior.

```
CREATE PROCEDURE NotaLetra ()
BEGIN

   DECLARE codigo integer;
   DECLARE letra char(2);
   DECLARE final integer DEFAULT 0;
   DECLARE curEstudiantes CURSOR FOR
SELECT codigoMatricula FROM ESTUDIANTE_DE_GRADO;
   DECLARO CONTINUE HANDLER FOR NOT FOUND SET final = 1;

   OPEN curEstudiantes;

   WHILE NOT final DO

      FETCH curEstudiantes INTO código;

      SELECT calculoNotaLetra(código) INTO letra;

      UPDATE ESTUDIANTE
      SET notaLetra = letra
      WHERE codigoMatricula = código;

   END WHILE;

   CLOSE curEstudiantes;

END;
```

En este procedimiento se declara un cursor (curEstudiantes) que permitirá obtener todos los códigos de matrícula de los estudiantes de grado. Una vez definido, puede usarse y para ello primero habrá que abrirlo con la sentencia SQL OPEN <Cursor>. A partir de aquí, con la estructura iterativa WHILE (mientras queden códigos de estudiantes) se obtiene el código de la fila en curso utilizando la sentencia SQL FETCH <nombre_cursor> INTO <variable>. Esta sentencia deposita el código de matrícula de la fila que está en la ventana del cursor en una variable, lo que permitirá utilizarla a continuación para invocar el procedimiento CalculoNotaLetra y obtener el valor de la letra con la que se actualizará el campo nombreLetra de la tabla ESTUDIANTE. Cuando no queden más códigos de matrícula por procesar, hay que cerrar el cursor con la sentencia CLOSE <cursor>. Tened en cuenta que, una vez declarado el cursor, indicamos la condición para continuar obteniendo la siguiente fila del cursor, la que se recuperará con el siguiente FETCH.

Así es como llamaríamos al procedimiento NotaLetra y comprobaríamos su resultado:

```
CALL NotaLetra();
SELECT * FROM ESTUDIANTE;
```

codigo-Matricula	nombre	apellidos	estudios	porcentaje	notaLetra
1	Joan	Pi Dot	Grado	0	B
2	Laura	Sentís Aguilar	Máster	0	NULL

codigo-Matricula	nombre	apellidos	estudios	porcentaje	notaLetra
3	Roc	Sánchez Gómez	Grado	0	B
4	Joana	Sauler Sunyer	Grado	0	B

Justificación del uso de los cursores de datos

La mejor opción no siempre es utilizar un cursor de datos para iterar elementos de un conjunto de datos. Esto es así porque los sistemas de gestión de bases de datos incorporan mecanismos optimizados para hacer ciertos tratamientos sobre varias filas o tuplas. Como regla podemos decir que hay que emplear los cursores cuando es necesario un tratamiento diferente para cada fila o tupla. Si el tratamiento que hay que realizar para cada fila de datos es el mismo, no es necesario utilizar un cursor. Por ejemplo, si quisiéramos poner a 'N' todas las notas en forma de letra de los estudiantes de grado, no habría un cursor. Lo podríamos hacer con la sentencia SQL UPDATE, tal y como se muestra a continuación:

```
UPDATE ESTUDIANTE
SET notaLetra = 'N';
WHERE codigoMatricula IN (
   SELECT codigoMatricula
   FROM ESTUDIANTE_DE_GRADO
);
```

En cambio, si la nota cualitativa en forma de letra fuera diferente para cada estudiante de grado, entonces necesitaríamos un cursor que permitiera para cada estudiante calcular su calificación textual y actualizarla.

2.3. Los disparadores (*triggers*)

Si bien los procedimientos almacenados son bloques de código que ejecutaremos siempre de manera consciente, puede ser que, para mantener una cierta lógica de programación, necesitemos tener procedimientos que se ejecuten de manera transparente para el desarrollador cuando nuestra base de datos entra en un cierto estado o se da una serie de condiciones.

Imaginemos el caso, ya conocido, de la eliminación en cascada. Para mantener la integridad referencial, cuando se da la orden de eliminar un registro de una tabla de la base de datos, se dispara la ejecución de una serie de operaciones que tendrán como objetivo eliminar los registros de las tablas que hacían referencia en la clave primaria del registro eliminado.

Los **disparadores**, *triggers* en inglés, serán unos objetos de la base de datos que se ejecutarán de manera automática cuando se produzca un acontecimiento determinado.

Por ejemplo, imaginemos una base de datos que modela el departamento de ventas de una empresa. Supongamos, también, que en el momento en que un vendedor consigue vender una cantidad determinada de productos de un cierto tipo, por ejemplo, si a finales de año sobrepasa una determinada cifra de ventas, este vendedor recibirá un premio.

Necesitamos algún mecanismo que examine la cifra acumulada de ventas cada vez que se inserte una venta nueva en la base de datos, de modo que, si supera el umbral establecido, notifique que hay premio.

Para implementar situaciones como la descrita en el ejemplo, definiremos un disparador que se ejecutará siempre que se den las condiciones adecuadas.

Este disparador estará accesible para todos los objetos de la base de datos y su ejecución será autónoma, de forma que no habrá que establecer mecanismos de polling y, además, será transparente para el desarrollador, de forma que este no necesitará ser consciente de su existencia.

Polling

Acción síncrona repetitiva que tiene por objeto comprobar que se cumple una cierta acción o que un sistema logra un estado en particular.

Este último hecho es, también, la contraindicación de los disparadores. Si bien el desarrollador no necesita tener conciencia de la existencia de los disparadores, es posible que este hecho provoque que se realicen nuevas operaciones –tanto si se dan dentro de la base de datos como en las aplicaciones que acceden a ella– que invaliden o colisionen con los cambios realizados por el disparador.

Así pues, usaremos los disparadores en ocasiones concretas:

- Implementación de una regla de negocio, conocida, parte *core* del programa, y debidamente documentada.

- Mantenimiento automático de tablas de auditoría de actividad en la base de datos. Esta tarea será ajena al negocio y no interferirá con otros desarrollos que se puedan realizar. Las tareas de auditoría serán frecuentes cuando se trabaje con datos sensibles y se quiera añadir una trazabilidad de los cambios transparente respecto al negocio de la aplicación.

- Mantenimiento automático de columnas derivadas. A pesar de que el modelo relacional recomienda no utilizar columnas derivadas, es posible que su existencia facilite el acceso a cierta información. El uso de disparadores permitirá mantener los datos actualizados de manera transparente para el usuario que acceda a ellos.

- Comprobación de restricciones de integridad definidas por el diseñador de la base de datos y no asumibles por la integridad referencial básica. De este modo, el disparador cancelará la operación antes de que se modifiquen los datos. Siguiendo el ejemplo anterior, en el supuesto de que un vendedor supere una cierta cantidad en los premios acumulados durante el año, se puede hacer saltar una regla de integridad que no permita superarla.

- Reparación automática de restricciones de integridad. En caso de que no se hayan implementado los *triggers* que validarán las reglas de integridad referencial definidas por el diseñador de la base de datos, se podrán definir disparadores que, una vez infringida la regla de integridad, realizarán las operaciones necesarias para corregir los datos con el fin de que vuelvan a ser válidos.

2.3.1. Sintaxis

La sintaxis para la creación de un disparador será diferente según sea el SGBD, implementando más o menos opciones. Por ejemplo, en MySQL, la sintaxis será la siguiente:

```
CREATE [DEFINER = usuario] TRIGGER <nom_disparador>
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON <nombre_tabla> FOR EACH ROW
[{FOLLOWS | PRECEDES} <nombre_otro_disparador>]
<sentencias_sql>
```

Como se ve en la definición, los disparadores se activan solo en tres situaciones: al insertar en una tabla (INSERT), al borrarlos (DELETE) o al modificar valores contenidos en las tablas (UPDATE). Así pues, no será posible plantear disparadores que se ejecuten cuando se realiza una consulta de datos en una o más tablas (SELECT) o como consecuencia de ejecutar procedimientos almacenados.

Para borrar un disparador utilizaremos la sentencia DROP.

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

Si queremos modificar el contenido de un disparador, el procedimiento a seguir es eliminarlo y crear otro de nuevo.

Podrá suceder que haya más de un disparador asignado a más de un evento asociado a una tabla o, incluso, en una misma operación sobre un campo de una tabla.

Cada SGBD implementará un orden de ejecución propio. En el caso de MySQL, los disparadores asociados a una misma tabla y evento se ejecutarán sucesivamente por su orden de creación.

Se podrá modificar este flujo a partir de los modificadores FOLLOWS y PRECEDES de la cláusula FOR EACH ROW.

Los disparadores, por otra parte, se pueden activar antes (BEFORE) o después (AFTER) de haber procesado los acontecimientos que dan pie a ejecutarlos. Estos eventos estarán siempre fijados en una tabla, por lo que si queremos añadir una misma acción automática en diferentes tablas tendremos que crear tantos disparadores como tablas se quieran monitorizar.

De este modo, por ejemplo, podremos prever si se incumplirá una regla de integridad antes de que la base de datos alcance un estado incoherente o bien podremos actualizar una tabla de auditoría después de que se haya aplicado una modificación de un dato.

Las acciones que se activarán después de ejecutar un disparador solo podrán tener lugar una sola vez para cada evento. No obstante, se podrán encadenar disparadores para indicar si la ejecución se debe hacer antes (PRECEDES) o después (FOLLOWS) de que se haya ejecutado otro disparador.

El disparador, al activarse, adoptará los privilegios propios del usuario indicado como DEFINER, no los del usuario que ejecuta las acciones que provocan el inicio del disparador.

En la definición de las sentencias, y solo en el caso de los disparadores de tipos FOR EACH ROW, podremos hacer referencia a los valores de antes y después de la acción con las variables de sistema:

- **NEW**: es una variable de tipo compuesto que contiene la fila después de la ejecución de una sentencia de modificación (UPDATE) o de la fila que hay que insertar (INSERT).
- **OLD**: es una variable de tipo compuesto que contiene la fila antes de la ejecución de una sentencia de modificación (UPDATE) o de la fila que hay que borrar (DELETE).

Finalmente, podrá haber situaciones en las que los disparadores se ejecuten en cascada, porque un disparador realiza una acción que inicia otro disparador, y así sucesivamente.

Este tipo de situaciones pueden conducir a bucles infinitos y hay que tener mucho cuidado al diseñar las situaciones en las que se crearán los disparadores.

A continuación mostramos un par de ejemplos de disparadores que ilustran la utilización de las cláusulas AFTER y BEFORE.

Cuando cambie la nota de un estudiante de grado, queremos que se calcule de manera automática la nota cualitativa y se modifique convenientemente la columna notaLetra en la tabla ESTUDIANTE. Usaremos el procedimiento almacenado CalculoNotaLetra que ya hemos definido anteriormente.

```
CREATE TRIGGER actualizaNotaLetra AFTER UPDATE ON ESTUDIANT_DE_GRAU
FOR EACH ROW
BEGIN
DECLARE nota char(2);
SELECT calculoNotaLetra (NEW.codigoMatricula) INTO nota;
UPDATE ESTUDIANTE
SET notaLetra = nota
WHERE codigoMatricula = NEW.codigoMatricula;
END;
```

Si queremos probar su funcionamiento, podemos ejecutar:

```
UPDATE ESTUDIANTE_DE_GRADO
SET notaFinal = 4
WHERE codigoMatricula = 1;
```

```
SELECT * FROM ESTUDIANTE WHERE codigoMatricula = 1;
```

codigo-Matricula	nombre	apellidos	estudios	porcentaje	notaLetra
1	Joan	Pi Dot	Grado	0	C–

Queremos que, de manera automática, se compruebe que la nueva nota que se quiere poner a un estudiante de grado está dentro del rango de notas posibles; es decir, entre 0 y 10. En caso contrario, no se podrá modificar la nota.

```
CREATE TRIGGER controlNotaFinal BEFORE UPDATE ON ESTUDIANTE_DE_GRADO
FOR EACH ROW
BEGIN
IF NEW.notaFinal <0 or NEW.notaFinal> 10 THEN
SET NEW.notaFinal = OLD.notaFinal;
END IF;
END;
```

Podemos probar el resultado ejecutando esta sentencia:

```
SELECT * FROM ESTUDIANTE_DE_GRADO WHERE codigoMatricula = 1;
```

codigo-Matricula	nombre	apellidos	porcentaje	notaFinal
1	Joan	Pi Dot	1,33	4

Intentad modificar la nota final del estudiante (con código de matrícula 1) cambiándola de 4 a 100. Veréis que no es posible, que queda como estaba, ya que se ha ejecutado el disparador ControlNotaFinal que hemos creado.

```
UPDATE ESTUDIANTE_DE_GRADO
SET notaFinal = 100
WHERE codigoMatricula = 1;
SELECT * FROM ESTUDIANTE_DE_GRADO WHERE codigoMatricula = 1;
```

codigo-Matricula	nombre	apellidos	porcentaje	notaFinal
1	Joan	Pi Dot	1,33	4

3. Concurrencia y transacciones

A menudo nos encontraremos que habrá diferentes usuarios y aplicaciones que accederán de manera simultánea a los datos. El SGBD tendrá que ser capaz de permitir este tipo de acceso y asegurar que los datos recuperados son válidos y coherentes. Será parte de las tareas del SGBD mantener la integridad referencial en casos de acceso concurrente y de modificación simultánea de la información.

Tal y como hemos visto cuando hablábamos de los disparadores (caso de una acción que provoca la ejecución de más de un disparador), habrá que establecer una serie de mecanismos que nos permitan asegurar la validez de los datos en todo momento, dividiendo las acciones que se deben realizar en pequeños bloques indivisibles que se irán ejecutando de manera sincronizada.

Por este motivo, introduciremos el concepto de transacción, que representa la unidad de trabajo que será necesaria para controlar la concurrencia y la recuperación, si procede, de los datos.

Una **transacción** es un conjunto de operaciones contra la base de datos que deben ejecutarse a la vez, como si fueran una sola operación.

La utilización de las transacciones nos permitirá proteger los datos y las aplicaciones de las anomalías provocadas por el acceso simultáneo a la base de datos por parte de usuarios y de otras aplicaciones.

Las transacciones nos permitirán, además, dotar al usuario de la base de datos de la sensación de que solo él tiene acceso a la base de datos, de forma que sus acciones siempre generarán el resultado esperado, sin errores derivados del uso de un conjunto de datos incoherentes.

Por ejemplo, retomemos el caso de la base de datos que modela el departamento de ventas de una empresa. Podrá suceder que dos o más vendedores realicen una venta de un tipo de producto al mismo tiempo, de forma que se supere la cantidad disponible en *stock*.

En este caso, sin control de concurrencia y de transacciones, podría pasar que, sobre un *stock* de 30 unidades de producto, el vendedor A procesa un pedido de 25 y, el vendedor B, uno de 15. Las acciones que se realizarán en base a los datos serán:

1) El vendedor A consulta el *stock* para validar que se puede realizar el pedido (30 > 25).

2) El resultado es afirmativo y se guarda el valor del *stock* (30) para iniciar la operación.

3) El vendedor B consulta el *stock* para validar que se puede realizar el pedido (30 > 15).

4) El resultado es afirmativo y se guarda el valor del *stock* (30) para iniciar la operación.

5) El vendedor A modifica el valor del *stock*: 30 – 25 → 5.

6) El vendedor B modifica el valor del *stock*: 30 – 15 → 15.

El resultado es que el valor final del *stock* es 15, cuando en realidad se ha sobrepasado la cantidad de producto existente y no se tendría que haber permitido al vendedor B realizar la operación.

Otro caso que nos permitirá controlar el uso de transacciones es el de pérdida de conexión con la base de datos en medio de un conjunto de operaciones, de forma que los datos quedarían en un estado intermedio y no serían coherentes con el estado real.

Consideremos el caso que, en el proceso de venta, una vez se ha modificado el *stock*, se pasa a incrementar el valor de los ingresos por venta de este vendedor derivados de la venta del producto.

Si, una vez modificado el *stock*, se pierde la conexión con la base de datos, la venta no se acabará de procesar y devolverá un error, y habría que volver a procesar desde el principio comprobando el *stock*, etc.

El valor del *stock*, en este momento, no será correcto y esta incoherencia en los datos impedirá el correcto funcionamiento de la aplicación.

En definitiva, el SGBD deberá aportar mecanismos de recuperación que eviten la pérdida de datos existentes, así como reaccionar ante la incoherencia en la actualización de los datos.

Por este motivo, se definirán cuatro propiedades que deberán de implementar todos los SGBD para ofrecer la capacidad de trabajar con transacciones, conocidas por el nombre ACID:

1) Atomicidad: el conjunto de operaciones que constituirán una transacción se considerará como una unidad de ejecución atómica e indivisible. Si no se ejecutan todas las operaciones que constituyen la transacción, habrá que recuperar el estado previo al inicio.

Una vez haya finalizado la ejecución de las operaciones de la transacción, se ejecutará la sentencia COMMIT, que consolidará los cambios hechos en la base de datos.

Si hay algún problema en la ejecución y hay que abortarla, se realizará una operación de ROLLBACK, que se podrá ejecutar tanto de manera explícita como implícita.

2) Consistencia: la ejecución de una transacción preservará, por encima de todo, la consistencia y la coherencia de la base de datos, manteniendo la integridad referencial, de dominio, etc.

3) *Isolation*: (aislamiento) una transacción se ejecutará sin interferencia de ninguna otra acción que se pueda ejecutar de manera concurrente.

4) Definitividad: el resultado de una transacción será definitivo y, solo al finalizarse y una vez ejecutada la sentencia COMMIT, se podrán modificar las tablas y los datos involucrados en las operaciones que forman la transacción.

3.1. Nivel de concurrencia

Con el aislamiento de las transacciones puede ocurrir que el conjunto de operaciones que las formen sea demasiado extenso y bloquee tanto las tablas que serán modificadas como los datos y, por lo tanto, que este hecho afecte al rendimiento de las aplicaciones que accedan a ellas.

El **nivel de concurrencia** indicará cómo se aprovecharán los recursos disponibles, según se permita el encabalgamiento de ciertas operaciones dentro de diferentes transacciones.

Continuando con el ejemplo anterior, se pueden plantear dos transacciones: una para consultar el total de productos por tipo previa modificación de la tabla y otra para calcular el nuevo valor al modificar el *stock*. Dado que son operaciones diferentes, se puede plantear un encabalgamiento de forma que se puedan realizar, de manera concurrente, partes de ambas transacciones.

El SGBD tendrá que ser capaz de detectar estos posibles encabalgamientos para optimizar el uso de los recursos disponibles y favorecer la velocidad de ejecución de las operaciones.

3.2. Sintaxis

Las transacciones no se definirán como bloques de código cerrados, sino que solo se indicarán marcando el inicio y el final de las operaciones que se considerarán como parte de la transacción.

Una vez activada la transacción, todas las operaciones que se realicen a continuación formarán parte de ella hasta que se indique la finalización de manera explícita.

Para indicar el inicio de una transacción, utilizaremos la sintaxis siguiente:

```
SET TRANSACTION <modo_de_acceso>;
```

El modo de acceso podrá ser de dos tipos: READ ONLY, en caso de que solo se quiera acceder a las bases de datos para realizar consultas, y READ WRITE, en caso de que alguna de las operaciones incluidas modifique los datos.

Nota

En otros SGBD, la instrucción que indica inicio de transacción puede ser BEGIN TRANSACTION o START TRANSACTION.

Para indicar la finalización de una transacción, utilizaremos la sintaxis siguiente:

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

Como hemos visto previamente, la sentencia COMMIT consolidará los cambios realizados durante la transacción, mientras que ROLLBACK revertirá los cambios realizados durante la transacción.

La palabra reservada WORK será solo indicativa y, además, será opcional.

3.3. Responsabilidades del SGBD y del desarrollador

Según hemos visto, para evitar los impactos en el rendimiento de la base de datos será importante planificar correctamente el orden de las operaciones que se ejecutarán dentro de una transacción.

Así pues, por parte del desarrollador habrá que determinar si una cierta operación tendrá que formar parte de una transacción o no, o si una transacción se puede dividir en diferentes transacciones para favorecer la concurrencia.

Esta tarea formará parte de las responsabilidades del desarrollador, quien tendrá que garantizar que las transacciones tendrán una duración mínima, asegurando también la coherencia y la consistencia de los datos.

El SGBD tendrá que garantizar, definiendo un plan de acceso a partir del estudio de las operaciones que forman las diferentes transacciones, que las peticiones de escritura y de lectura se ejecuten en concurrencia de manera óptima.

Así mismo, estará entre las responsabilidades del SGBD garantizar que se cumplan todas las reglas de integridad definidas en la base de datos, tanto si es como validación previa al cierre de la transacción con un COMMIT, como si es inmediatamente después de la ejecución de cada una de las operaciones.

Obviamente, en caso de error o de violación de una regla de integridad, el SGBD tendrá que ser capaz de devolverla al estado previo al inicio de la transacción en que estaba la base de datos.

Resumen

En este módulo didáctico hemos visto cómo se organiza un servidor de base de datos, representado como un conjunto de catálogos y, dentro de ellos, un conjunto de esquemas. En dichos esquemas encontraremos el resto de objetos de la base de datos, como pueden ser las tablas y vistas.

También hemos visto distintos objetos de la base de datos que ofrece el sistema gestor para facilitar el acceso a los datos. Para empezar, las vistas, que permiten acceder a subconjuntos de datos de una o más tablas como si estuvieran almacenados en una única tabla y en qué casos es modificable dicho conjunto de datos.

Después, hemos visto otros objetos que se almacenan en el esquema de la base de datos y que permiten encapsular el acceso a datos, haciéndolo transparente al desarrollador de aplicaciones: los procedimientos almacenados y los disparadores.

Estos objetos nos permitirán definir algoritmos y conjuntos de operaciones que se ejecutarán de manera explícita. En el caso de los procedimientos y las funciones, o de manera implícita, en el caso de los disparadores, en el momento en que se den una serie de condiciones establecidas.

Finalmente, hemos visto cómo trabajaría el SGBD en caso de que haya un acceso concurrente a las tablas y los datos, por medio de transacciones, para garantizar la coherencia y la integridad de la información.

Actividades

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

a) Según el SQL estándar, una BD se crea con la sentencia CREATE DATABASE <nombre_bd>.

b) El esquema de la base de datos contiene información sobre las tablas y otros componentes lógicos de los esquemas de los usuarios.

c) Según la 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 SOCIO (
  nSocio char(10) primary key,
  sexo char(1) not null,
  check (sexo='M' or sexo='H'));

CREATE TABLE CLUB (
  nClub char(20) primary key);

CREATE TABLE SOCIO_CLUB (
  nSocio char(10) not null references SOCIO,
  nClub char(20) not null references CLUB,
  primary key(nSocio, nClub));

CREATE TABLE clubs_con_mas_de_5_socios (
  nClub char(20)
  primary key references CLUB);
```

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 SOCIO_CLUB.

Además, si el club pasa a tener más de cinco socios, lo tiene que dar de alta en clubs_con_mas_de_cinco_socios. El procedimiento tiene que 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 del *stock* 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 PRODUCTO(
  nProd INTEGER,
```

```
    stock INTEGER NOT NULL,
    puntoPedido INTEGER NOT NULL,
    qtatPedir INTEGER NOT NULL,
    PRIMARY KEY(nProd));

  CREATE TABLE PETICION_PENDIENTE(
    nProd INTEGER,
    qtt INTEGER NOT NULL,
    fecha DATE,
    PRIMARY KEY (nProd));
```

4. A partir de las tablas creadas en el ejercicio anterior, definid una vista que implemente la regla de negocio: «obtener el identificador de producto y *stock* siempre que el *stock* sea menor que el valor de punto de pedido».

¿Se pueden insertar datos en esta vista? ¿Cuáles son las condiciones que hay que evaluar antes de la inserción?

Glosario

ACID *m* Acrónimo formado por las palabras en inglés *atomicity*, *consistency*, *isolation* y *durability* (en castellano, atomicidad, consistencia, aislamiento y definitividad), que indica las propiedades que ha de tener toda transacción.

administrador de la base de datos *m* Usuario de la base de datos que gestionará los objetos, la estructura y los privilegios de acceso del resto de usuarios.

cancelación de una transacción *f* Finalización de una transacción sin que se confirmen las actualizaciones hechas en la BD.

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

confirmación de una transacción *f* Finalización de una transacción que hace que los cambios realizados se vuelvan definitivos en la BD.

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

control de concurrencia *m* Conjunto de técnicas que utiliza un SGBD para evitar que haya interferencias entre transacciones que se ejecutan concurrentemente.

DBA *m* Siglas en inglés del usuario administrador de la base de datos.

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

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

nivel de concurrencia *m* Grado de aprovechamiento de los recursos de proceso, disponibles según el encabalgamiento de ejecución de las transacciones, que acceden concurrentemente a la BD y consiguen confirmar.

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

rol *m* Conjunto de privilegios que tiene asignado un usuario de la base de datos, definido por el usuario administrador.

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 en un servidor.

transacción *f* Secuencia de operaciones de lectura y de actualización de la BD que cumple las propiedades ACID.

vista *f* Conjunto de datos, resultado de una consulta, al cual se puede acceder igual que a una tabla.