TEMA 4: LENGUAJE SQL PARA LA MANIPULACIÓN Y DEFINICIÓN DE LOS DATOS. CONTROL DE TRANSACCIONES YCONCURRENCIA

David Bataller Signes

ÍNDICE

INTRODUCCIÓN

Sobre las bases de datos no sólo tenemos que aplicar instrucciones para extraer información, sino que es necesario poder manipular la información registrada (añadiendo nueva, eliminando y modificando la ya introducida).

Por ello en esta unidad aprenderemos las instrucciones SQL para la manipulación de datos de una base de datos.

Haremos los primeros pasos en el conocimiento del lenguaje SQL, y nos introduciremos en los tipos de datos que puede gestionar. Asimismo, hay que disponer de algún mecanismo para definir tablas, vistas, índices y otros objetos que conforman la base de datos, y también para modificar la estructura si es necesario. Y, por supuesto, también es muy importante poder controlar el acceso a la información que hay en la base de datos. El lenguaje SQL nos proporciona sentencias para alcanzar todos estos objetivos.

En un SGBD en explotación, se suele encomendar al administrador del SGBD la definición de las estructuras de datos. Pero eso no quita que todo informático -tanto si es especializado en desarrollo de aplicaciones informáticas o en administración de sistemas informáticos- debe conocer las principales sentencias que proporciona el lenguaje SQL para la definición de las estructuras de datos.

Piense que una persona que desarrolle aplicaciones debe ser capaz de crear la estructura de la base de datos (tablas, vistas, índices ...).

Así pues, comenzaremos conociendo los diferentes tipos de datos y las diversas posibilidades para manipular la información, para continuar definiendo las estructuras que permiten almacenar los datos y terminar controlando el concepto de transacción y también las herramientas para controlarlas y para controlar el acceso concurrente a los datos.

2. EL LENGUAJE SQL

2.1 Orígenes y evolución del lenguaje SQL bajo la guía de los SGBD

El modelo relacional en el que se basan los SGBD actuales fue presentado en 1970 por el matemático Edgar Frank Codd, que trabajaba en los laboratorios de investigación de la empresa de informática IBM. Uno de los primeros SGBD relacionales a aparecer fue el System R de IBM, que se desarrolló como prototipo para probar la funcionalidad del modelo relacional y que iba acompañado del lenguaje SEQUEL (acrónimo de Structured English Query Language) para manipular y acceder a los datos almacenados en el System R. Posteriormente, la palabra SEQUEL se condensó en SQL (acrónimo de SQL).

Una vez comprobada la eficiencia del modelo relacional y del lenguaje SQL, se inició una dura carrera entre diferentes marcas comerciales. Así, tenemos el siguiente:

• IBM comercializa diversos productos relacionales con el lenguaje SQL: System / 38 en 1979, SQL / DS en 1981, y DB2 en 1983.

• Relational Software, Inc. (Actualmente, Oracle Corporation) crea su propia versión de SGBD relacional para la Marina de los EE.UU., la CIA y otros, y el verano de 1979 libera Oracle V2 (versión 2) para las computadoras VAX (Las grandes competidoras de la época con las computadoras de IBM).

El lenguaje SQL evolucionó (cada marca comercial seguía su propio criterio) hasta que los principales organismos de estandarización intervinieron para obligar a los diferentes SGBD relacionales implementar una versión común del lenguaje y, así, en 1986 la ANSI (American National Standards Institute) publica el estándar SQL-86, que en 1987 es ratificado por la ISO (Organización internacional para la Normalización, o International Organization for Standardization en inglés).

2.2 Tipos de sentencias SQL

Los SGBD relacionales incorporan el lenguaje SQL para ejecutar diferentes tipos de tareas en las bases de datos: definición de datos, consulta de datos, actualización de datos, definición de usuarios, concesión de privilegios ... Por este motivo, las sentencias que aporta el lenguaje SQL suelen agrupar en las siguientes:

1. LDD. Sentencias destinadas a la definición de los datos (Data definition language, DDL), que permiten definir los objetos (tablas, campos, valores posibles, reglas de integridad referencial, restricciones ...).

2. LCD. Sentencias destinadas al control sobre los datos (Data control language, DCL), que permiten conceder y retirar permisos sobre los diferentes objetos de la base de datos.

3. LC. Sentencias destinadas a la consulta de los datos (Query language, QL), que permiten acceder a los datos en modo consulta.

4. LMD. Sentencias destinadas a la manipulación de los datos (Data manipulation language, DML), que permiten actualizar la base de datos (altas, bajas y modificaciones).

En algunos SGBD no hay distinción entre LC y LMD, y únicamente se habla de LMD para las consultas y actualizaciones. Del mismo modo, a veces se incluyen las sentencias de control (LCD) junto con las de definición de datos (LDD).

No tiene ninguna importancia que se incluyan en un grupo o que sean un grupo propio: es una simple clasificación.

Todos estos lenguajes suelen tener una sintaxis sencilla, similar a las órdenes de consola para un sistema operativo, llamada sintaxis auto-suficiente.

SQL alojado. Las sentencias SQL pueden presentar, sin embargo, una segunda sintaxis, sintaxis alojada, consistente en un conjunto de sentencias que son admitidas dentro de un lenguaje de programación llamado lenguaje anfitrión.

Así, podemos encontrar LC y LMD que pueden alojarse en lenguajes de tercera generación como C, Cobol, Fortran ..., y en lenguajes de cuarta generación. Los SGBD suelen incluir un lenguaje de tercera generación que permite alojar sentencias SQL en pequeñas unidades de programación (funciones o procedimientos). Así, el SGBD Oracle incorpora el lenguaje PL/SQL, el SGBD SQLServer incorpora el lenguaje Transact-SQL, el SGBD MySQL 5.x sigue la sintaxis SQL 2003 para la definición de rutinas del mismo modo que el SGBD DB2 de IBM.

2.3 Tipos de datos

La evolución anárquica que ha seguido el lenguaje SQL ha hecho que cada SGBD haya tomado sus decisiones en cuanto a los tipos de datos permitidas. Ciertamente, los diferentes estándares SQL que han ido apareciendo han marcado una cierta línea y los SGBD se acercan, pero tampoco pueden dejar de apoyar a los tipos de datos que han proporcionado a lo largo de su existencia, ya que hay muchas bases de datos repartidas por el mundo que las utilizan.

De todo ello tenemos que deducir que, a fin de trabajar con un SGBD, debemos conocer los principales tipos de datos que facilita (numéricas, alfanuméricas, momentos temporales ...) y debemos hacerlo centrándonos en un SGBD concreto teniendo en cuenta que el resto de SGBD también incorpora tipo de datos similares y, en caso de haber de trabajar, siempre tendremos que echar un vistazo a la documentación que cada SGBD facilita.

Cada valor manipulado por un SGBD determinado corresponde a un tipo de dato que asocia un conjunto de propiedades al valor. Las propiedades asociadas a cada tipo de dato hacen que un SGBD concreto trate de manera diferente los valores de diferentes tipo de datos.

En el momento de creación de una tabla, hay que especificar un tipo de dato para cada una de las columnas. En la creación de una acción o función almacenada en la base de datos, hay que especificar un tipo de dato para cada argumento. La asignación correcta del tipo de dato es fundamental para que los tipos de datos definen el dominio de valores que cada columna o argumento puede contener. Así, por ejemplo,

las columnas de tipo DATE no podrán aceptar el valor '30 de febrero' ni el valor 2 ni la cadena Hola.

Dentro de los tipos de datos básicos, podemos distinguir los siguientes:

• Tipo de datos para gestionar información alfanumérica.
• Tipo de datos para gestionar información numérica.
• Tipo de datos para gestionar momentos temporales (fechas y tiempo).
• Otros tipos de datos.

MySQL es el SGBD con el que se trabaja en estos materiales y el lenguaje SQL de MySQL lo descrito. La notación que se utiliza, en cuanto a sintaxis de definición del lenguaje, habitual, consiste en poner entre corchetes ([]) los elementos opcionales, y separar con el carácter | los elementos alternativos.

2.3.1 Tipos de datos string

Los tipos de datos string almacenan datos alfanuméricos en el conjunto de caracteres de la base de datos. Estos tipos son menos restrictivos que otros tipo de datos y, en consecuencia, tienen menos propiedades. Así, por ejemplo, las columnas de tipo carácter pueden almacenar valores alfanuméricos -letras y cifras-, pero las columnas de tipo numérico sólo pueden almacenar valores numéricos.

MySQL proporciona los siguientes tipos de datos para gestionar datos alfanuméricas:

• CHAR
• VARCHAR
• BINARY
• VARBINARY
• BLOB
• TEXTO
• ENUM
• SET

El tipo CHAR [(longitud)]

Este tipo especifica una cadena de longitud fija (indicada por longitud) y, por tanto, MySQL asegura que todos los valores almacenados en la columna tienen la longitud especificada. Si se inserta una cadena de longitud más corta, MySQL la rellena con espacios en blanco hasta la longitud indicada. Si se intenta insertar una cadena de longitud más larga, se trunca.

La longitud mínima y por defecto (no es obligatoria) para una columna de tipo CHAR es de 1 carácter, y la longitud máxima permitida es de 255 caracteres.

Para indicar la longitud, es necesario especificar con un número entre paréntesis, que indica el número de caracteres, que tendrá la string. Por ejemplo CHAR (10).

El tipo VARCHAR (longitud)

Este tipo especifica una cadena de longitud variable que puede ser, como máximo, la indicada por longitud, valor que es obligatorio introducir.

Los valores de tipo VARCHAR almacenan el valor exacto que indica el usuario sin añadir espacios en blanco. Si se intenta insertar una cadena de longitud más larga, VARCHAR devuelve un error.

La longitud máxima de este tipo de datos es de 65.535 caracteres. La longitud se puede indicar con un número, que indica el número de caracteres máximo que contendrá el string. Por ejemplo: VARCHAR (10).

El tipo de datos alfanumérico más habitual para almacenar strings en base de datos MySQL es VARCHAR.

El tipo BINARY (longitud)

El tipo de dato BINARY es similar al tipo CHAR, pero almacena caracteres en binario. En este caso, la longitud siempre se indica en bytes. La longitud mínima para una columna BINARY es de 1 byte. La longitud máxima permitida es de 255.

El tipo VARBINARY (longitud)

El tipo de dato VARBINARY es similar al tipo VARCHAR, pero almacena caracteres en binario. En este caso, la longitud siempre se indica en bytes. Los bytes que no se rellenan explícitamente rellenan con '\ 0'.

Así pues, por ejemplo, una columna definida como VARBINARY (4) a la que asigne el valor 'a' contendrá, realmente, 'a \ 0 \ 0 \ 0' y habrá que tenerlo en cuenta a la hora de hacer, por ejemplo, comparaciones, ya que no será el mismo comparar la columna con el valor 'a' que con el valor 'a \ 0 \ 0 \ 0'.

El valor '\ 0' en hexadecimal se corresponde con 0x00.

El tipo BLOB

El tipo de datos BLOB es un objeto que permite contener una cantidad grande y variable de datos de tipo binario.

De hecho, se puede considerar un dato de tipo BLOB como un dato de tipo VARBINARY, pero sin limitación en cuanto al número de bytes. De hecho, los valores de tipo BLOB almacenan en un objeto separado del resto de columnas de la tabla, debido a sus requerimientos de espacio.

Realmente, hay varios subtipos de BLOG: TINYBLOB, BLOB, MEDIUMBLOB y LONGBLOB.

• TINYBLOB puede almacenar hasta 28 + 2 bytes

• BLOB puede almacenar hasta 216 + 2 bytes

• MEDIUMBLOB puede almacenar hasta 224 + 3 bytes

• LONGBLOB puede almacenar hasta 232 + 4 bytes

El tipo TEXTO

El tipo de datos TEXTO es un objeto que permite contener una cantidad grande y variable de datos de tipo carácter.

De hecho, se puede considerar un dato de tipo TEXTO como un dato de tipo VARCHAR, pero sin limitación en cuanto al número de caracteres. De manera similar al tipo BLOB, los valores tipo TEXTO también se almacenan en un objeto separado de la resto de columnas de la tabla, debido a sus requerimientos de espacio.

Realmente, hay varios subtipos de TEXTO: TINYTEXT, TEXTO, MEDIUMTEXT y LONGTEXT:

• TINYTEXT puede almacenar hasta 28 + 2 bytes

• TEXTO puede almacenar hasta 216 + 2 bytes

• MEDIUMTEXT puede almacenar hasta 224 + 3 bytes

• LONGTEXT puede almacenar hasta 232 + 4 bytes

El tipo ENUM ('cadena1' [, 'cadena2'] ... [, 'cadena_n'])

El tipo ENUM define un conjunto de valores de tipo string con una lista prefijada de cadenas que se definen en el momento de la definición de la columna y que se corresponderán con los valores válidos de la columna.

Ejemplo de columna tipo ENUM

CREATE TABLE sizes (\\name ENUM ('small', 'medium', 'large') \\);

El conjunto de valores son obligatoriamente literales entre comillas simples.
Si una columna se declara de tipo ENUM y se especifica que no admite valores nulos, entonces, el valor por defecto será el primero de la lista de cadenas.

El número máximo de cadenas diferentes que puede soportar el tipo ENUM es 65535.

El tipo SET ('cadena1' [, 'cadena2'] ... [, 'cadena_n'])

Una columna de tipo SET puede contener cero o más valores, pero todos los elementos que contenga deben pertenecer a una lista especificada en el momento de la creación.
El número máximo de valores diferentes que puede soportar el tipo SET es 64.

Por ejemplo, se puede definir una columna de tipo SET (`one ',` two'). Y un elemento concreto puede tener cualquiera de los siguientes valores:
• ' '
• 'one'
• 'two'
• 'one, two' (el valor 'two, one' no se prevé que el +++++++ los elementos
no afecta las listas)

2.3.2 Tipos de datos numéricos

MySQL soporta todos los tipos de datos numéricos de SQL estándar:

• INTEGER (también abreviado por INT)
• SMALLINT
• DECIMAL (también abreviado por DEC o FIXED)
• NUMERIC

También soporta:

• FLOAT
• REAL
• DOUBLE PRECISION (también llamado DOUBLE, simplemente, o bien REAL)
• BIT
• BOOLEAN

Los tipos de datos INTEGER

El tipo INTEGER (comúnmente abreviado como INT) almacena valores enteros.

Hay varios subtipos de enteros en función de los valores admitidos (véase la tabla 1).

Tipo de entero	Almacenamiento (en bytes)	Valor mínimo (con signo/sin signo)	Valor máximo (con signo/sin signo)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	5	-9223372036854775808 / 0	-9223372036854775807 / 18446744073709551615

Los tipos de datos enteros admiten la especificación del número de dígitos que hay que mostrar de un valor concreto, utilizando la sintaxis:

INT (N), siendo N el número de dígitos visibles.

Así, pues, si se especifica una columna de tipo INT (4), en el momento de seleccionar un valor concreto, se mostrarán sólo 4 dígitos. Hay que tener en cuenta que esta especificación no condiciona el valor almacenado, tan sólo fija el valor que hay que mostrar.

También se puede especificar el número de dígitos visibles en los subtipos de INTEGER, utilizando la misma sintaxis.

Para almacenar datos de tipo entero en bases de datos MySQL el más habitual es utilizar el tipo de datos INT.

Tipo FLOAT, REAL y DOUBLE

FLOAT, REAL y DOUBLE son los tipos de datos numéricos que almacenan valores numéricos reales (es decir, que admiten decimales).

Los tipos FLOAT y REAL almacenan en 4 bytes y los DOUBLE en 8 bytes.
Los tipos FLOAT, REAL o DOUBLE PRECISION admiten que especifiquen los dígitos de la parte entera (E) y los dígitos de la parte decimal (D, que pueden ser 30 como máximo, y nunca más grandes que E-2). La sintaxis para esta especificación sería:
• FLOAT (E, D)
• REAL (E, D)
• DOUBLE PRECISION (E, D)

Ejemplo de almacenamiento en FLOAT

Por ejemplo, si se define una columna como FLOAT (7,4) y se quiere almacenar el valor 999.00009, el valor almacenado realmente será 999.0001, que es el valor más cercano (aproximado) al original.

Tipo de datos DECIMAL y NUMERIC

DECIMAL y NUMERIC son los tipos de datos reales de punto fijo que admite MySQL. Son sinónimos y, por tanto, se pueden utilizar indistintamente.

Los valores en punto fijo no se almacenarán nunca de manera redondeada, es decir, que si hay que almacenar un valor en un espacio que no es adecuado, emitirá un error. Este tipo de datos permiten asegurar que el valor es exactamente lo que se ha introducido. No se ha redondeado. Por tanto, se trata de un tipo de datos muy adecuado para representar valores monetarios, por ejemplo.

Ambos tipos de datos permiten especificar el total de dígitos (T) y la cantidad de dígitos decimales (D), con la siguiente sintaxis:

DECIMAL (T, D)
NUMERIC (T, D)

Valores posibles para NUMERIC

Por ejemplo un NUMERIC (5,2) podría contener valores desde -999.99 hasta 999.99. También se admite la sintaxis DECIMAL (T) y NUMERIC (T) que es equivalente a DECIMAL (T, 0) y NUMERIC (T, 0).

El valor predeterminado de T es 10, y su valor máximo es 65.

Tipo de datos BIT

BIT es un tipo de datos que permite almacenar bits, desde 1 (por defecto) hasta 64. Para especificar el número de bits que almacenará debe definirse, siguiendo la sintaxis siguiente:

BIT (M), en la que M es el número de bits que se almacenarán.

Los valores literales de los bits se dan siguiendo el formato: b'valor_binario'.
Por ejemplo, un valor binario admisible para un campo de tipo BIT sería b'0001 '.
Si damos el valor b'1010 'en un campo definido como BIT (6) el valor que almacenará será b'001010 '. Añadirá, pues, ceros a la izquierda hasta completar el número de bits definidos en el campo.

BIT es un sinónimo de TINYINT (1).

Otros tipos numéricos

BOOL o booleana es el tipo de datos que permite almacenar tipos de datos booleanas (que permiten los valores verdadero o falso). BOOL o booleana es sinónimo de Tiny (1). Almacenar un valor de cero se considera falso. En cambio, un valor distinto de cero se interpreta como cierto.

Modificadores de tipo numéricos

Hay algunas palabras clave que se pueden añadir a la definición de una columna numérica (entera o real) para acondicionar los valores que contendrán.

• UNSIGNED: con este modificador sólo se admitirán los valores de tipo numérico no negativos.
• ZEROFILL: se añadirán ceros a la izquierda hasta completar el total de dígitos del valor numérico, si es necesario.
• AUTO_INCREMENTv: cuando se añade un valor 0 o NULL en aquella columna, el valor que se almacena es el valor más alto incrementado en 1. El primer valor predeterminado es 1.

2.3.3 Tipos de datos para momentos temporales

El tipo de datos que MySQL dispone para almacenar datos que indiquen momentos temporales son:

• DATETIME
• DATE
• TIMESTAMP
• TIME
• YEAR

Tenga en cuenta que cuando hay que referirse a los datos referentes a un año es importante explicitar los cuatro dígitos. Es decir, que para expresar el año 98 del siglo XX lo mejor es referirse a ella explícitamente así: 1.998.

Si se utilizan dos dígitos en lugar de cuatro para expresar años, hay que tener en cuenta que MySQL los interpretará de la siguiente manera:

• Los valores de los años que van entre 00 a 69 se interpretan como los años: 2000-2069.
• Los valores de los años que van entre 70-99 interpretan como los años: 1970-1999.

El tipo de dato DATE

DATE permite almacenar fechas. El formato de una fecha en MySQL es 'AAAA-MM-DD ', en el que AAAA indica el año expresado en cuatro dígitos, MM indica el mes expresado en dos dígitos y DD indica el día expresado en dos dígitos.

La fecha mínima soportada por el sistema es '1000-01-01'. Y la fecha máxima admisible en MySQL es '9999-12-31'.

El tipo de dato DATETIME

DATATIME es un tipo de dato que permite almacenar combinaciones de días y horas.

El formato de DATETIME en MySQL es 'AAAA-MM-DD HH: MM: SS', en la que AAAA-MM-DD es el año, el mes y el día, y HH: MM: SS indican la hora, minuto y segundos, expresados en dos dígitos, separados por ':'.

Los valores válidos para los campos de tipo DATETIME van desde '1000-01-01 12:00:00 'hasta' 9999-12-31 23:59:59 '.

El tipo de dato TIMESTAMP

TIMESTAMP es un tipo de dato similar a DATETIME y tiene el mismo formato por defecto: 'AAAA-MM-DD HH: MM: SS'. TIMESTAMP, sin embargo, permite almacenar la hora y fecha actuales en un momento determinado.

Si se asigna el valor NULL en una columna TIMESTAMP o no se le asigna ninguno explícitamente, el sistema almacena por defecto la fecha y hora actuales. Si especifica una fecha-hora concretas en la columna, entonces la columna tomará aquel valor, como si se tratara de una columna DATETIME.

El rango de valores que admite TIMESTAMP es de '1970-01-01 12:00:01' a '2038-01-19 03:14:07 '.

Una columna TIMESTAMP es útil cuando se desea almacenar la fecha y hora en el momento de añadir o modificar un dato en la base de datos, por ejemplo.

Si en una tabla hay más de una columna de tipo TIMESTAMP, el funcionamiento de la primera columna TIMESTAMP es la esperable: se almacena la fecha y hora de la operación más reciente, a menos que explícitamente se asigne un valor concreto, y, entonces, prevalece el valor especificado. El resto de columnas TIMESTAMP de una misma tabla no se actualizarán con este valor. En este caso, si no se especifica un valor concreto, el valor almacenado será cero.

Ejemplo de creación de tabla utilizando TIMESTAMP

Para crear una tabla con una columna de tipo TIMESTAMP son equivalentes las sintaxis siguientes:

• CREATE TABLE t (ts TIMESTAMP);
• CREATE TABLE t (ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
• CREATE TABLE t (ts TIMESTAMP ON UPDATE CURRENT_TIMESTAMP\ newline DEFAULT CURRENT_TIMESTAMP);

Tipo de datos TIME

TIME es un tipo de dato específico que almacena la hora en el formato 'HH:MM:SS'. TIME, sin embargo, también permite expresar el tiempo transcurrido entre dos momentos (Diferencia de tiempo). Por ello, los valores que permite almacenar son de '-838:59:59 ' a ' 838:59:59 '. En esta caso, el formato será 'HHH: MM: SS'.

Otros formatos también admitidos para un dato de tipo TIME son: 'HH:MM', 'D HH:MM:SS', 'D HH:MM', 'D HH ', o 'SS', en el que D indica los días. Es posible, también, un formato que admite microsegundos 'HH: MM: SS.uuuuuu' en que uuuuuu son los microsegundos.

Tipo de datos YEAR

YEAR es un dato de tipo BYTE que almacena datos de tipo año. el formato predeterminado es AAAA (el año expresado en cuatro dígitos) o bien 'AAAA', expresado como string.

También se pueden utilizar los tipos YEAR(2) o YEAR(4), para especificar columnas de tipo año expresado con dos dígitos o año con cuatro dígitos.

Se admiten valores desde 1901 fin a 2155. También se admite 0000. En el formato de dos dígitos, se admiten los valores del 70 al 69, que representan los años de 1970 a 2069.

2.3.4 Otros tipos de datos

En MySQL hay extensiones que permiten almacenar otros tipos de datos: los datos poligonales.
Así pues, MySQL permite almacenar datos de tipo objeto poligonal y da implementación al modelo geométrico del estándar OpenGIS.

Por lo tanto, podemos definir columnas MySQL de tipo Polygon, Point, Curve o Line, entre otros.

3. Instrucciones para la manipulación de datos

El lenguaje SQL aporta una serie de instrucciones con las que se pueden realizar las acciones siguientes:

• La manipulación de los datos (instrucciones LMD que nos deben permitir efectuar altas, bajas y modificaciones).
• La definición de datos (instrucciones LDD que nos deben permitir crear, modificar y eliminar las tablas, los índices y las vistas).
• El control de datos (instrucciones LCD que nos deben permitir gestionar los usuarios y sus privilegios).

El lenguaje SQL proporciona un conjunto de instrucciones, reducido pero muy potente, para manipular los datos, dentro del cual se ha de distinguir entre dos tipos de instrucciones:

• Las instrucciones que permiten ejecutar la manipulación de los datos, y que se reducen a tres: INSERT para la introducción de nuevas filas, UPDATE para la modificación de filas, y DELETE por el borrado de filas.
• Las instrucciones para el control de transacciones, que deben permitir asegurar que un conjunto de operaciones de manipulación de datos se ejecute con éxito en su totalidad o, en caso de problema, se aborte total o hasta un determinado punto en el tiempo.

Antes de introducirnos en el estudio de las instrucciones INSERT, UPDATE y DELETE, hay que conocer como el SGBD gestiona las instrucciones de inserción, eliminación y modificación que podamos ejecutar, ya que hay dos posibilidades de funcionamiento:

• Que queden automáticamente validadas y no haya posibilidad de tirar atrás. En este caso, los efectos de toda instrucción de actualización de datos que tenga éxito son automáticamente accesibles desde el resto de conexiones de la base de datos.
• Que queden en una cola de instrucciones, que permite echar atrás. En este caso, se dice que las instrucciones de la cola están pendientes de validación, y el usuario debe ejecutar, cuando lo cree conveniente, una instrucción para validarlas (llamada COMMIT) o una instrucción para echar atrás (llamada ROLLBACK).

Este funcionamiento implica que los efectos de las instrucciones pendientes de validación no se ven por el resto de conexiones de la base de datos, pero sí son accesibles desde la conexión donde se han efectuado. Al ejecutar la COMMIT, todas las conexiones acceden a los efectos de las instrucciones validadas. En caso de

ejecutar ROLLBACK, las instrucciones desaparecen de la cola y ninguna conexión (ni la propia ni el resto) no accede a los efectos correspondientes, es decir, es como si nunca hubieran existido.

Estos posibles funcionamientos forman parte de la gestión de transacciones que proporciona el SGBD y que hay que estudiar con más detenimiento. A la hora, pero, de ejecutar instrucciones INSERT, UPDATE y DELETE debemos conocer el funcionamiento del SGBD para poder actuar en consecuencia.

Así, por ejemplo, un SGBD MySQL funciona con validación automática después de cada instrucción de actualización de datos a menos que se indique lo contrario y, en cambio, un SGBD Oracle funciona con la cola de instrucciones pendientes de confirmación o rechazo que debe indicar al usuario.

En cambio, en MySQL, si se quiere desactivar la opción de autocommit que hay por defecto, habrá que ejecutar la siguiente instrucción:

SET autocommit = 0;

3.1 Sentencia INSERT

La sentencia INSERT es la instrucción proporcionada por el lenguaje SQL para insertar nuevas filas en las tablas. Admite dos sintaxis:

1. Los valores que se han de insertar explicitan en la misma instrucción en la cláusula values:

insert into <nombre_tabla> [(col1, COL2 …)] values (val1, val2 ...);

2. Los valores que se han de insertar consiguen por medio de una sentencia SELECT:

insert into <nombre_tabla> [(col1, COL2 …)] select ...;

En todo caso, se pueden especificar las columnas de la tabla que se han de rellenar y el orden en que se suministran los diferentes valores. En caso de que no se especifiquen las columnas, el SQL entiende que los valores se suministran para todas las columnas de la tabla y, además, en el orden en que están definidos en la tabla.

La lista de valores de la cláusula values y la lista de resultados de la sentencia SELECT deben coincidir en número, tipo y orden con la lista de columnas que se deben rellenar.

Ejemplo 1 de sentencia INSERT

En el esquema empresa, se pide insertar el departamento 50 de nombre INFORMÁTICA '. La posible sentencia para conseguir el objetivo es ésta:

insert into dept (dept_no, dnom) values (50, INFORMÁTICA ');

Si ejecutamos una consulta para comprobar el contenido actual de la tabla DEPT, encontraremos la nueva fila sin localidad asignada. El SGBD ha permitido dejar la localidad con valor NULL porque lo tiene permitido así, como se puede ver en el descriptor de la tabla DEPT:

SQL> desc dept;

Atributo	Null	Tipo
DEPT_NO	NOT NULL	INT(2)
DNOMBRE	NOT NULL	VARCHAR(14)
LOC		VARCHAR(14)

3 rows selected

Ejemplo 2 de sentencia INSERT

En el esquema sanidad, se pide dar de alta el doctor de código 100 y nombre 'SANCHEZ D.'. La solución parece que podría ser esta:

insert into doctor (doctor_no, apellido) values (100, 'SANCHEZ D.');

Al ejecutar esta sentencia, el SGBDR da un error.

Lo cierto es que la tabla DOCTOR no admite valores nulos en la columna hospital_cod, ya que esta columna forma parte de la clave primaria. Miremos el descriptor de la tabla DOCTOR:

SQL> desc doctor;

Atributo	Null	Tipo
HOSPITAL_COD	NOT NULL	INT(2)
DOCTOR_COD	NOT NULL	INT(3)
APELLIDOS	NOT NULL	VARCHAR(13)
ESPECIALIDAD	NOT NULL	VARCHAR(16)

4 rows selected

Aparte de la columna hospital_cod, también deberíamos dar un valor en la columna especialidad, ya que tampoco admite valores nulos.

Recordemos que, en nuestro esquema sanidad, la columna especialidad es una cadena que no tiene ningún tipo de restricción definida ni es clave foránea de ninguna tabla en la que haya todas las especialidades posibles. Por lo tanto, si queremos saber qué especialidades hay para escribir del doctor que queremos insertar, idénticamente a las ya introducidas en caso de que hubiera algún doctor con la misma especialidad del que queremos insertar, hacemos lo siguiente:

```
SQL> select distinct especialidad from doctor;

ESPECIALIDAD

Urología
pediatría
Cardiología
Neurología
Ginecología
Psiquiatría

6 rows selected
```

Supongamos que el doctor 'SANCHEZ D.' es psiquiatra. Como ya hay algún doctor con la especialidad 'Psiquiatría', correspondería hacer la inserción utilizando la misma grafía para la especialidad. Además, supongamos que queremos dar de alta el doctor en el hospital 66.

```
insert into doctor (doctor_no, apellido, hospital_cod, especialidad)
values (100, 'SANCHEZ D.', 66, 'Psiquiatría');
```

Esta vez, el SGBD también se nos queja con otro tipo de error: ha fallado la referencia a la clave foránea.

El error nos informa que una restricción de integridad definida en la tabla ha intentado ser violada y, por tanto, la instrucción no ha finalizado con éxito. El SGBD nos pasa dos informaciones para que tengamos pistas de dónde está el problema:

• Nos da una descripción breve del problema (no se puede añadir una fila hija -child row-), nos da a entender que se trata de un error de clave foránea, es decir, que no existe el código en la tabla referenciada.
• Nos dice la restricción que ha fallado (sanitat.doctor, CONSTRAINT … FOREIGN KEY (HOSPITA_COD) ...).

El SGBD tiene toda la razón. Recordemos que la columna hospital_cod de la tabla HOSPITAL es clave foránea de la tabla HOSPITAL. Esto significa que cualquier inserción en la tabla DOCTOR debe ser para hospitales existentes en la tabla

HOSPITAL, y esto no ocurre con el hospital 66, como se puede ver en consultar los hospitales existentes:

SQL> select * from hospital;

HOSPITAL_COD	NOMBRE	DIRECCIÓN	TELÉFONO	CDAD_CAMAS
13	Provincial	Donella 50	9644264	88
18	General	Atocha s/n	5953111	63
22	La Paz	Castellana 100	9235411	162
45	San Carlos	Ciudad Universitaria	5971500	92

4 rows selected

Así pues, o nos hemos equivocado de hospital o debemos dar de alta previamente el hospital 66. Supongamos que es el segundo caso y que, por tanto, tenemos que dar de alta el hospital 66:

insert into hospital (hospital_cod, nombre, direccion) values (66, 'General', 'De la fuente, 13');

El SGBD nos acepta la instrucción. Fijémonos que hemos informado del código de hospital, del nombre y de la dirección. Miremos el descriptor de la tabla HOSPITAL:

SQL> desc hospital;

Atributo	Null	Tipo
HOSPITAL_COD	NOT NULL	INT(2)
NOMBRE	NOT NULL	VARCHAR(10)
DIRECCION		VARCHAR(20)
TELEFONO		VARCHAR(8)
CDAD_CAMAS		INT(3)

5 rows selected

Vemos cinco campos, de los cuales sólo los dos primeros tienen marcada la obligatoriedad de valor. Por tanto, no se nos ha quejado porque no hayamos indicado el teléfono del hospital ni la cantidad de camas que tiene el hospital.

Comprobamos la información que ahora hay en la tabla HOSPITAL:

SQL> select * from hospital;

HOSPITAL_COD	NOMBRE	DIRECCIÓN	TELÉFONO	CDAD_CAMAS
13	Provincial	Donella 50	9644264	88
18	General	Atocha s/n	5953111	63
22	La Paz	Castellana 100	9235411	162
45	San Carlos	Ciudad Universitaria	5971500	92
66	General	De la Fuente, 13		0

5 rows selected

Para el nuevo hospital, la columna teléfono no tiene valor (valor NULL), pero la columna cdad_camas tiene el valor 0. De dónde ha salido? Esto se debe a que la columna cdad_camas de la tabla HOSPITAL tiene definido el valor por defecto (0) que el SGBD utiliza para rellenar la columna cdad_camas cuando se produce una inserción en la tabla sin indicar valor para esta columna.

Ahora parece que ya podemos insertar nuestro doctor 'SANCHEZ D.:

```
insert into doctor (doctor_no, apellido, hospital_cod, especialidad)
values (100, 'SANCHEZ D.', 66, 'Psiquiatría');
```

No nos olvidemos de registrar los cambios con la instrucción COMMIT o de hacer ROLLBACK, si tenemos la autocommit desactivado.

Ejemplo 3 de sentencia INSERT

Antes de empezar, desactivaremos el autocommit que tiene configurado por defecto MySQL para poder practicar el commit y el rollback:

```
SET AUTOCOMMIT = 0;
```

En el esquema empresa, se quiere insertar la instrucción identificada por número 1.000, con fecha de orden del 1 de septiembre de 2000 y para el cliente 500.

Quizás necesitamos conocer, en primer lugar, el descriptor de la tabla PEDIDO:

```
SQL> desc pedido;
```

Atributo	Null	Tipo
PED_NUM	NOT NULL	INT(4)
PED_FECHA		DATE
PED_TIPO		VARCHAR(1)
CLIENTE_COD	NOT NULL	INT(6)
FECHA_ENVIO		DATE
TOTAL		DECIMAL(8,2)

6 rows selected

Fijémonos que tenemos la información correspondiente a todos los campos obligatorios. Por lo tanto, podemos ejecutar el siguiente:

```
insert into pedido (ped_num, ped_fecha, cliente_cod)
values (1000, '2000/09/01', 500);
```

El SGBD nos reporta el error de restricción de integridad sobre la clave foránea. Y, por supuesto, el SGBD vuelve a tener razón, ya que en el esquema empresa la tabla PEDIDO tiene una restricción de clave foránea en la columna cliente_cod.

Si consultamos el contenido de la tabla cliente, veremos que no hay ningún cliente con código 500.
Por ello, el SGBD ha dado un error. Supongamos que era un error nuestro y el orden correspondía al cliente 109 (que sí existe en la tabla CLIENTE). Esta vez la instrucción siguiente no nos da ningún problema.

```
insert into pedido (com_num, com_data, client_cod)
values (1000, '2000/09/01', 109);
```

Podemos comprobar cómo ha quedado insertada la orden:

```
SQL> select * from pedido where ped_num = 1000;

PED_NUM  PED_FECHA  PED_TIPO  CLIENTE_COD  FECHA_ENVIO  TOTAL
1000     01/09/2000           109
```

Hacemos rollback para echar atrás la inserción efectuada y así poder comprobar que también la podríamos hacer de diferentes maneras. Recordemos que no es obligatorio indicar las columnas para las que se introducen los valores. En este caso, el SGBD espera todas las columnas de la tabla en el orden en que están definidas en la tabla. Así pues, podemos hacer lo siguiente:

```
insert into pedido
values (1000, '2000/09/01', NULL, 109, NULL, NULL);
```

Hacemos rollback para probar otra posibilidad. Fijémonos que el SGBD también nos deja introducir un precio total de orden cualquiera:

```
rollback;

insert into pedido
values (1000, DATE '2000  09  01', NULL, 109, NULL, 9999);

commit;
```

El SGBDR ha aceptado esta sentencia y ha insertado la fila correspondiente. Pero, hemos introducido un importe total que no se corresponde con la realidad, ya que no hay ninguna línea de detalle. Es decir, el valor 9999 no es válido. Los SGBD

proporcionan mecanismos (Disparadores) para controlar este tipo de incoherencias de los datos.

Ejemplo 4 de sentencia INSERT

En primer lugar, volvemos a activar la opción de autocommit para que sea más cómodo el trabajo.

SET AUTOCOMMIT = 1;

Como detalle de la orden 1000 insertada en el ejemplo anterior, en el esquema empresa se quieren insertar las mismas líneas que contiene la orden 620.

En este caso, ejecutaremos una instrucción INSERT tomando como valores que se insertarán los que nos da el resultado de una sentencia SELECT:

```
insert into detalle
select 1000, detall_num, prod_num, preu_venda, cantidad, importe
from detalle
where com_num = 620;
```

En esta instrucción, hemos seleccionado las filas de detalle de la orden 620 y las hemos insertado como filas de detalle de la orden 1000. Tenemos que ser conscientes de que el importe total de la orden 1000 sigue siendo, sin embargo, incorrecto.

Como ya hemos comentado, hemos utilizado una sentencia SELECT para insertar valores en una tabla. Es una coincidencia que las dos sentencias actúen sobre la misma tabla DETALLE.

Al no indicar, en la sentencia INSERT, las columnas en que se han de insertar los valores, ha sido necesario construir la sentencia SELECT de manera que las columnas de la cláusula select coincidieran, en orden, con las columnas de la tabla en que se ha de efectuar la inserción. Además, como para todas las filas de la orden 620 había que indicar 1000 como número de orden, la cláusula SELECT ha incorporado la constante 1000 como valor para la primera columna.

3.2 Sentencia UPDATE

La sentencia UPDATE es la instrucción proporcionada por el lenguaje SQL para modificar filas que hay en las tablas.

Su sintaxis es la siguiente:

update <nombre_tabla>

set col1 = val1, COL2 = val2, col3 = val3 ...
[where <condición>];

La cláusula optativa where selecciona las filas que deben actualizarse. En caso de inexistencia, se actualizan todas las filas de la tabla.

La cláusula set indica las columnas que deben actualizarse y el valor con que actualizarlas.
El valor de actualización de una columna puede ser el resultado obtenido por una sentencia SELECT que recupera una única fila:

update <nombre_tabla>
set col1 = (select exp1 from ...),
set col2 = (select exp2 from ...),
set col3 = val3,
...
[where <condición>];

En tales situaciones, la sentencia SELECT es una subconsulta de la sentencia UPDATE que puede utilizar valores de las columnas de la fila que se está modificando en la sentencia UPDATE.

Como a veces es posible que sea necesario actualizar los valores de más de una columna a partir de diferentes resultados de una misma sentencia SELECT, no sería nada eficiente ejecutar varias veces la misma sentencia SELECT para actualizar más de una columna. Por tanto, la sentencia UPDATE también admite la siguiente sintaxis:

update <nombre_tabla>
set (col1, col2) = (select exp1, exp2 from ...),
set col3 = val3,
...
[Where <condición>];

Ejemplo 1 de sentencia UPDATE

En el esquema empresa, se quiere modificar la localidad de los departamentos de manera que queden todos los caracteres en minúsculas. La instrucción para resolver la solicitud puede ser esta:

update dept
set loc = lower (loc);

Ahora se quiere modificar la localidad de los departamentos de forma que queden con la inicial en mayúscula y el resto de letras en minúsculas. La instrucción para resolver la solicitud puede ser esta:

```
update dept
set loc = concat (upper (left (loc, 1)), right (lower (loc), length (loc)-1));
```

Ejemplo 2 de sentencia UPDATE

En el esquema empresa, se quiere actualizar el importe total real del pedido 1000 a partir de los importes de las diferentes líneas de detalle que forman el pedido.

```
update pedido c
set total = (select sum (importe) from detalle where ped_num = c.ped_num)
where ped_num = 1000;
```

Ahora podemos comprobar la corrección de la información que hay en la base de datos sobre el pedido 1000:

```
SQL> select * from detalle where ped_num = 1000;
```

PED_NUM	DETALLE_NUM	PROD_NUM	PRECIO_VENTA	CANTIDAD	IMPORTE
1000	1	100860	35	10	350
1000	2	200376	2,4	1000	2400
1000	3	102130	3,4	500	1700

3 rows selected

```
SQL> select * from pedido where ped_num = 1000;
```

PED_NUM	PED_FECHA	PED_TIPO	CLIENTE_COD	FECHA_ENVIO	TOTAL
1000	01/09/2000		109		4450

1 rows selected

3.3 Sentencia DELETE

La sentencia DELETE es la instrucción proporcionada por el lenguaje SQL para borrar filas existentes que hay en las tablas. Su sintaxis es la siguiente:

```
delete from <nombre_tabla>
[where <condición>];
```

La cláusula optativa where selecciona las filas que se deben eliminar. En su defecto, se eliminan todas las filas de la tabla.

Ejemplo de sentencia DELETE

En el esquema empresa, se quiere eliminar el pedido 1000. La instrucción parece que podría ser esta:

```
delete from pedido
where ped_num = 1000;
```

Al ejecutar esta sentencia, sin embargo, nos encontramos con un error que nos indica que no se puede eliminar una fila padre (a parent row).

El motivo es que la columna ped_num de la tabla DETALLE es clave foránea de la tabla PEDIDO, lo que imposibilita eliminar una cabecera de pedido si hay líneas de detalle correspondientes. Estas se eliminarían de manera automática si hubiera definida la eliminación en cascada, pero no es el caso. Así pues, habrá que hacer lo siguiente:

```
delete from detalle where ped_num = 1000;
delete from pedido where ped_num = 1000;
```

3.4 Sentencia REPLACE

MySQL tiene una extensión del lenguaje SQL estándar que permite insertar una nueva fila que, en caso de que la clave primaria coincida con otra fila, previamente sea eliminada. Se trata de la sentencia REPLACE. Hay tres posibles sintaxis para la sentencia REPLACE:

```
REPLACE [INTO] nombre_tabla [(columna1, ...)]
{VALUES | VALUE} ({expr | DEFAULT}, ...), (...), ...

REPLACE [INTO] nombre_tabla
SET columna1 = {expr | DEFAULT}, ...

REPLACE [INTO] nombre_tabla [(columna1, ...)]
SELECT …
```

3.5 Sentencia LOAD XML

LOAD XML permite leer un fichero en formato xml y almacenar los datos contenidos en una tabla de la base de datos. Su sintaxis es:

LOAD XML [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'nombre_archivo'
[REPLACE | IGNORE]
INTO TABLE [nombre_base_datos.] nombre_tabla
[CHARACTER SET nombre_charset]
[ROWS IDENTIFIED BY <nombre_tag> ']
[IGNORE numero [LINES | ROWS]]
[(columnas, ...)]
[SET nombre_columna = expresión, …]

4. DDL

El lenguaje SQL aporta instrucciones para definir las estructuras en las que se almacenan los datos. Así, por ejemplo, tenemos instrucciones para la creación, eliminación y modificación de tablas e índices, y también instrucciones para definir vistas.

En el MySQL, el SQLServer y PostgreSQL cualquier instancia del SGBD gestiona un conjunto de bases de datos o esquemas, llamado cluster database, el cual puede tener definido un conjunto de usuarios con los privilegios de acceso y gestión que correspondan.

En el MySQL, el SQLServer y PostgreSQL, el lenguaje SQL proporciona una instrucción CREATE DATABASE <nombre_base_datos> que permite crear, dentro de la instancia, las diversas bases de datos. Esta instrucción CREATE DATABASE se puede considerar dentro del ámbito del lenguaje LDD.

La sentencia CREATE SCHEMA en el ámbito del lenguaje LDD está destinada a la creación de un esquema en el que se puedan definir tablas, índices, vistas, etc. En MySQL, CREATE SCHEMA y CREATE TABLE son sinónimos.

Disponemos, también, de la instrucción USE <nombre_base_datos> para decidir la base de datos en que se trabajará (establecimiento de la base de datos de trabajo por defecto).

4.1 Reglas e indicaciones para nombrar objetos en MySQL

Dentro de MySQL, además de tabas, encontraremos otros tipos de objetos: índices, columnas, alias, vistas, procedimientos, etc.

Los nombres de los objetos dentro de una base de datos, y las bases de datos mismas, en MySQL actúan como identificadores, y, como tales no se podrán repetir en un mismo ámbito. Por ejemplo, no podemos tener dos columnas de una misma tabla que se llamen igual, pero sí en tablas diferentes.

Los nombres con los que llamamos los objetos dentro de un SGBD deberán seguir unas reglas sintácticas que debemos conocer.

En general, las tablas y las bases de datos son *not case sensitive*, es decir, que podemos hacer referencia en mayúsculas o minúsculas y no encontraremos diferencia, si el sistema operativo sobre el que estamos trabajando soporta *not case sensitive*.

Por ejemplo, en Windows podemos ejecutar indiferentemente:

select * from emp;

O bien:

select * from EMP;

Lo que no acostumbramos a hacer, sin embargo, es que dentro de una misma sentencia nos referimos a un mismo objeto en mayúsculas y en minúsculas a la vez:

select * from emp where EMP.EMP_NO = 7499;

Los nombres de columnas, índices, procedimientos y disparadores (triggers), en cambio, siempre son *not case sensitive.*

Los nombres de los objetos en MySQL admiten cualquier tipo de carácter, excepto / \ i.

De todos modos, se recomienda utilizar caracteres alfabéticos estrictamente. Si el nombre incluye caracteres especiales es obligatorio hacer referencia entre comillas del tipo acento grave ('). Por ejemplo:

create table 'ES UNA PRUEBA' (a int);

Se admiten también las comillas dobles (“) si activamos el modo ANSI_QUOTES:

SET sql_mode = 'ANSI_QUOTES';
create table "SE OTRA PRUEBA" (a int);

Cualquier objeto puede ser referido utilizando las comillas, aunque no sea necesario, como ahora en el ejemplo:

select * from ‘empresa’.’emp’ where ‘emp’.’emp_no’= 7499;

Aunque no es recomendable, se pueden nombrar objetos con palabras reservadas del mismo lenguaje como SELECT, INSERT, DATABASE, etc. Estos nombres, pero, tendrán que poner obligatoriamente entre comillas.

La longitud máxima de los objetos de la base de datos es 64 caracteres, excepto para los alias, que pueden llegar a ser de 256.

Finalmente, veamos algunas indicaciones para nombrar objetos:

• Utilizar nombres enteros, descriptivos y pronunciables y, si no es factible, buenas abreviaturas. En nombrar objetos, sopesad el objetivo de conseguir nombres cortos y fáciles de utilizar ante el objetivo de tener nombres que sean descriptivos. En caso de duda, elija el nombre más descriptivo, ya que los objetos de la base de datos pueden ser utilizados por mucha gente a lo largo del tiempo.
• Utilizar reglas de asignación de nombres que sean coherentes. Así, por ejemplo, una regla podría consistir en comenzar con gc_ todos los nombres de las tablas que forman parte de una gestión comercial.
• Utilizar el mismo nombre para describir la misma entidad o el mismo atributo en diferentes tablas. Así, por ejemplo, cuando un atributo de una tabla es clave foránea de otra tabla, es muy conveniente llamarlo con el nombre que tiene en la tabla principal.

4.2 Comentarios en MySQL

El servidor MySQL soporta tres estilos de comentarios:

• # hasta el final de la línea.
• -<espacio en blanco> hasta el final de la línea.
• / * hasta la próxima secuencia * /. Estos tipos de comentarios admiten varias líneas de comentario.

Ejemplos de los diferentes tipos de comentarios son los siguientes:

```
SELECT 1 + 1; # Este es el primer tipo de comentario

SELECT 1 + 1;     - Este es el segundo tipo de comentario

SELECT 1 / * Este es un tipo de comentario que se puede poner en medio de la línea
* / + 1;

SELECT 1+
/ *
Este es un
comentario
```

que se puede poner
en varias líneas * /
1;

4.3 Motores de almacenamiento en MySQL

MySQL soporta diferentes tipos de almacenamiento de tablas (motores de almacenamiento
o storage engines, en inglés). Y cuando se crea una tabla debe especificarse en qué sistema de los posibles queremos crear.

Por defecto, MySQL a partir de la versión 5.5.5 crea las tablas de tipo InnoDB, que es un sistema transaccional, es decir, que soporta las características que hacen que una base de datos pueda garantizar que los datos se mantendrán consistentes.

Las propiedades que garantizan los sistemas transaccionales son las características llamadas ACID. ACID es el acrónimo inglés de atomicity, consistency, isolation, durability:

• Atomicidad: se dice que un SGBD garantiza atomicidad si cualquier transacción o bien finaliza correctamente (commit), o bien no deja ningún rastro de su ejecución (rollback).
• Consistencia: se habla de consistencia cuando la concurrencia de diferentes transacciones no puede producir resultados anómalos.
• Aislamiento: cada transacción dentro del sistema se debe ejecutar como si fuera la única que se ejecuta en ese momento.
• Definitividad: si se confirma una transacción, en un SGBD, el resultado de esta debe ser definitivo y no se puede perder.

Sólo el motor InnoDB permite crear un sistema transaccional en MySQL. Los otros tipos de almacenamiento no son transaccionales y no ofrecen control de integridad en las bases de datos creadas.

Evidentemente, este sistema (InnoDB) de almacenamiento es lo que a menudo interesará utilizar para las bases de datos que creamos, pero puede haber casos en que sea interesante considerar otros tipos de motores de almacenamiento. Por esto, MySQL también ofrece otros sistemas tales como, por ejemplo:

• MyISAM: era el sistema por defecto antes de la versión 5.5.5 de MySQL. Se utiliza mucho en aplicaciones web y en aplicaciones de almacén de datos (Datawarehousing).
• Memory: este sistema almacena todo en memoria RAM y, por tanto, se utiliza para sistemas que requieran un acceso muy rápido a los datos.

• Merge: agrupa tablas de tipo MyISAM para optimizar listas y búsquedas. Las tablas que hay que agrupar deben ser similares, es decir, deben tener el mismo número y tipo de columnas.

Para obtener una lista de los motores de almacenamiento soportados por la versión MySQL que tenga instalada, se puede ejecutar el comando SHOW ENGINES.

4.4 Creación de tablas

La sentencia CREATE TABLE es la instrucción proporcionada por el lenguaje SQL para la creación de una tabla.

Es una sentencia que admite múltiples parámetros, y la sintaxis completa se puede consultar en la documentación del SGBD que corresponda, pero la sintaxis más simple y usual es ésta:

```
create table [<nombre_esquema>.] <Nombre_tabla>
(<nombre_columna> <tipo_dato> [default <expresión>]
[<lista_restricciones_para_la_columna>],
<nombre_columna> <tipo_dato> [default <expresión>]
[<lista_restricciones_para_la_columna>],
...
[<lista_restricciones_adicionales_para_una_o_varias_columnas>]);
```

Fijémonos que hay bastantes elementos que son optativos:

• Las partes obligatorias son el nombre de la tabla y, por cada columna, el nombre y el tipo de dato.
• El nombre del esquema en el que se crea la tabla es optativo y si no se indica, la tabla se intenta crear dentro del esquema en el que estamos conectados.
• Cada columna tiene permitido definir un valor predeterminado (opción default) a partir de una expresión, el cual utilizará el SGBD en las instrucciones de inserción cuando no se especifique un valor para las columnas que tienen definido el valor por defecto. En MySQL el valor por defecto debe ser constante, no puede ser, por ejemplo, una función como NOW () ni una expresión como CURRENT_DATE.
• La definición de las restricciones para una o más columnas también es optativa en el momento de proceder a la creación de la tabla.

También es muy usual crear una tabla a partir del resultado de una consulta, con la sintaxis:

```
create table [<nombre_esquema>.] <Nombre_tabla> [(<nombre_de_los_campos>]
as <sentencia_select>;
```

En esta sentencia, no se definen los tipos de campos que se corresponden con los tipos de las columnas recuperadas en la sentencia SELECT. La definición de los nombres de los campos es optativa; si no se efectúa, los nombres de las columnas recuperadas pasan a ser los nombres de los campos nuevos. Sin embargo, habrá que añadir las restricciones que correspondan. La tabla nueva contiene una copia de las filas resultantes de la sentencia SELECT.

A la hora de definir tablas, hay que tener en cuenta varios conceptos:

• Los tipos de datos que el SGBD posibilita.
• Las restricciones sobre los nombres de tablas y columnas.
• La integridad de los datos.

El SGBD MySQL proporciona varios tipos de restricciones (*constraints* en la nomenclatura que se utilizará en los SGBD) u opciones de restricción para facilitar la integridad de los datos. En general, se pueden definir en el momento de crear la tabla, pero también se pueden alterar, añadir y eliminar con posterioridad.

Cada restricción lleva asociado un nombre (único en todo el esquema) que se puede especificar en el momento de crear la restricción. Si no se especifica, el SGBD asigna uno por defecto. Veamos, a continuación, los diferentes tipos de restricciones:

Clave primaria. Para definir la clave primaria de una tabla, hay que utilizar la constraint primary key. Si la clave primaria está formada por una única columna, se puede especificar en la línea de definición de la columna correspondiente, con la siguiente sintaxis:

<columna> <tipo_dato> primary key

En cambio, si la clave primaria está formada por más de una columna, se debe especificar obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la sintaxis siguiente:

[constraint <nombre_restriccion>] primary key (col1, col2, …)

Las claves primarias que afectan a una única columna también se pueden especificar con este segundo procedimiento.

Obligatoriedad de valor. Para definir la obligatoriedad de valor en una columna, hay que utilizar la opción not null. Esta restricción se puede indicar en la definición de la columna correspondiente con esta sintaxis:

<columna> <tipo_dato> [not null]

Por supuesto, no es necesario definir esta restricción sobre columnas que forman parte de la clave primaria, ya que formar parte de la clave primaria implica, automáticamente, la imposibilidad de tener valor nulos.

Unicidad de valor. Para definir la unicidad de valor en una columna, hay que utilizar la constraint unique. Si la unicidad especifica para una única columna, se puede asignar en la línea de definición de la columna correspondiente, con la siguiente sintaxis:

<columna> <tipo_dato> unique

En cambio, si la unicidad se aplica sobre varias columnas simultáneamente, hay que especificar obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la siguiente sintaxis:

[Constraint <nombre_restriccion>] unique (col1, col2 …)

Este segundo procedimiento también se puede emplear para aplicar la unicidad a una única columna.

Por supuesto, no es necesario definir esta restricción sobre un conjunto de columnas que forman parte de la clave primaria, ya que la clave primaria implica, automáticamente, la unicidad de valores.

Condiciones de comprobación. Para definir condiciones de comprobación en una columna, hay que utilizar la opción check (<condición>). Esta restricción se puede indicar en la definición de la columna correspondiente:

<columna> <tipo_dato> check (<condición>)

También se puede indicar en la zona final de restricciones sobre columnas de la tabla, con la siguiente sintaxis:

check (<condición>)

AUTO_INCREMENT. Podemos definir las columnas numéricas con la opción AUTO_INCREMENT. Esta modificación permite que al insertar un valor null o no insertar valor explícitamente en aquella columna definida de esta manera, se añada un valor predeterminado consistente en el mayor ya introducido incrementado en una unidad.

<columna> <tipo_dato> AUTO_INCREMENT

Comentarios de columnas. Podemos añadir comentarios a las columnas de manera que puedan quedar guardados en la base de datos y ser consultados. La manera de

hacerlo es añadir la palabra COMMENT seguida del texto que haya que poner como comentario entre comillas simples.

<columna> <tipo_dato> COMMENT 'comentario'

Integridad referencial. Para definir la integridad referencial, hay que utilizar la constraint foreign key. Si la clave foránea esta formada por una única columna, se puede especificar en la línea de definición de la columna correspondiente, con la siguiente sintaxis:

<columna> <tipo_dato> [constraint <nombre_restriccion>] references <tabla> [(columna)]

En cambio, si la clave foránea es formada por más de una columna, hay que especificarla obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la sintaxis siguiente:

[constraint <nombre_restriccion>] foreign key (col1, col2 ...)
references <tabla> [(col1, col2 …)]

Las claves foráneas que afectan a una única columna también se pueden especificar con este segundo procedimiento.

En cualquiera de los dos casos, se hace referencia a la tabla principal de la que estamos definiendo la clave foránea, lo que se hace con la opción references <tabla>.

En MySQL la integridad referencial sólo se activa si se trabaja sobre el motor InnoDB y utiliza la sintaxis de constraint de la zona de definición de restricciones del final y, además, se define un índice para las columnas implicadas en la clave foránea.

La sintaxis para la integridad referencial activa en MySQL es la siguiente (si utilizan otras sintaxis, el SGBD las reconoce pero no las valida):

index [<nombre_indice>] (col1, col2 ...)
[constraint <nombre_restriccion>] foreign key (col1, col2 ...) references <tabla> (col1, col2 …)

La sintaxis que hemos presentado para definir la integridad referencial no es completa. Nos falta tratar un tema fundamental: la actuación que esperamos del SGBD ante posibles eliminaciones y actualizaciones de datos en la tabla principal, cuando hay filas en otras tablas que hacen referencia.

La constraint foreign key se puede definir acompañada de los siguientes apartados:

• on delete <acción>, que define la actuación automática del SGBD sobre las filas de nuestra tabla que se ven afectadas por una eliminación de las filas a las que hacen referencia.
• on update <acción>, que define la actuación automática del SGBD sobre las filas de nuestra tabla que se ven afectadas por una actualización del valor al que hacen referencia.

Por si no te ha quedado claro, pensamos en las tablas DEPT y EMP del esquema empresa. La tabla EMP contiene la columna dept_no, que es clave foránea de la tabla DEPT. Por tanto, en la definición de la tabla EMP debemos tener definida una constraint foreign key en la columna dept_no haciendo referencia a la tabla DEPT. Al definir esta restricción de clave foránea, el diseñador de la base de datos tuvo que tomar decisiones respecto a lo siguiente:

• ¿Cómo debe actuar el SGBD ante el intento de eliminación de un departamento en la tabla DEPT si hay filas en la tabla EMP que se refieren? Esto se define en el apartado on delete <acción>.
• ¿Cómo debe actuar el SGBD ante el intento de modificación del código de un departamento en la tabla DEPT si hay filas en la tabla EMP que hacen referencia? Esto se define en el apartado on update <acción>.

En general, los SGBD ofrecen varias posibilidades de acción, pero no siempre son las mismas. Antes de conocer estas posibilidades, también necesitamos saber que algunos SGBD permiten diferir la comprobación de las restricciones de clave foránea hasta la finalización de la transacción, en lugar de efectuar la comprobación -y actuar en consecuencia- después de cada instrucción. Cuando esto es factible, la definición de la constraint va acompañada de la palabra deferrable o not deferrable. La actuación por defecto suele ser no diferir la comprobación.

Por lo tanto, la sintaxis de la restricción de clave foránea se ve claramente ampliada. Si se efectúa en el momento de definir la columna, tenemos lo siguiente:

[constraint <nombre_restriccion>] foreign key (col1, col2, ...) references <tabla>
(Col1, col2, ...) [on delete <acción>] [on update <acción>]

Las opciones que nos podemos llegar a encontrar en referencia a la acción que acompañe los apartados on update y on delete son estas: RESTRICT | CASCADE | SET NULL | NO ACTION.

• NO ACTION o RESTRICT: son sinónimos. Es la opción por defecto y no permite la eliminación o actualización de datos en la tabla principal.
• CASCADE: cuando se actualiza o elimina la fila padre, las filas relacionadas (Hijas) también se actualizan o eliminan automáticamente.

• SET NULL: cuando se actualiza o elimina la fila padre, las filas relacionadas (Hijas) se actualizan a NULL. Hay que haberlas definido de manera que admitan valores nulos.
• SET DEFAULT: cuando se actualiza o elimina la fila padre, las filas relacionadas (Hijas) se actualizan al valor predeterminado. MySQL soporta la sintaxis, pero no actúa, ante esta opción.

La opción CASCADE es muy peligrosa en utilizarla con on delete. Pensemos que pasaría, en el esquema empresa, si alguien decidiera eliminar un departamento de la tabla DEPT y la clave foránea en la tabla EMP fuera definida con on delete cascade: todos los empleados del departamento serían inmediatamente eliminados de la tabla EMP.

A veces, sin embargo, es muy útil acompañante on delete. Pensamos en la relación de integridad entre las tablas PEDIDO y DETALLE del esquema empresa. La tabla DETALLE contiene la columna ped_num, que es clave foránea de la tabla PEDIDO. En este caso, puede tener mucho sentido tener definida la clave foránea con on delete cascade, ya que la eliminación de una orden provocará la eliminación automática de sus líneas de detalle.

A diferencia de la caución en la utilización de la opción cascade para las actuaciones on delete, se suele utilizar mucho para las actuaciones on update.

Ejemplo 1 de creación de tablas. Tablas del esquema empresa

```
CREATE TABLE IF NOT EXISTS empresa.DEPT (
DEPT_NO  TINYINT(2) UNSIGNED,
DNOMBRRE  VARCHAR(14) NOT NULL UNIQUE,
LOC  VARCHAR(14),
PRIMARY KEY (DEPT_NO));

CREATE TABLE IF NOT EXISTS empresa.EMP (
EMP_NO  SMALLINT (4) UNSIGNED,
APELLIDO  VARCHAR (10) NOT NULL,
OFICIO  VARCHAR (10),
JEFE  SMALLINT (4) UNSIGNED,
FECHA_ALTA  DATE,
SALARIO  INT UNSIGNED,
COMISION  INT UNSIGNED,
DEPT_NO  TINYINT (2) UNSIGNED NOT NULL,
PRIMARY KEY (EMP_NO),
INDEX  IDX_EMP_JEFE (JEFE),
INDEX  IDX_EMP_DEPT_NO (DEPT_NO),
FOREIGN KEY (DEPT_NO) REFERENCES empresa.DEPT (DEPT_NO));
```

En primer lugar, hay que destacar la opción IF NOT EXISTS, opción muy utilizada a la hora de crear bases de datos con el fin de evitar errores en caso de que la tabla ya exista previamente. Por otra parte, cabe destacar que la tabla se define con el nombre del esquema ‘empresa’. Esto no es necesario si previamente se define este esquema como esquema por defecto. Esto se puede hacer con la sentencia USE:

USE empresa;

La clave primaria se ha definido en la parte inferior de la sentencia, no en la misma definición de la columna, a pesar de que se trataba de claves primarias que sólo tenían una única columna.

Fíjense como se han definido dos índices para las columnas JEFE y DEPT_NO para crear a posteriori las claves foráneas correspondientes. En el momento de la creación se crea la clave foránea que hace referencia de EMP a DPTO. No se crea, en cambio, la que hace referencia de EMP a la misma tabla y que sirve para referirse al jefe de un empleado. El motivo es que si se define esta clave foránea en el momento de la creación de la tabla, no podríamos insertar valores fácilmente, ya que no tendría el valor de referencia previamente introducido en la tabla. Por ejemplo, si queremos insertar el empleado número (EMP_NO) 7369 que tiene como empleado jefe al 7902 y este no esta todavía en la tabla, no lo podríamos insertar porque no existe el 7902 aún en la tabla. Una posible solución a este problema que se llama interbloqueo o deadlock, en inglés, es definir la clave foránea a posteriori de las inserciones. O bien, si el SGBD lo permite, desactivar la foreign key antes de insertarla y volverla a activar al terminar. Así pues, tras las inserciones habrá que modificar la tabla y añadir la restricción de clave foránea.

Obsérvese, también, que se ha utilizado el modificador de tipo UNSIGNED para definir los campos que necesariamente son considerados positivos. De modo que si se hace una inserción del tipo siguiente, el SGBD nos devolverá un error:

Insert into EMP values (100, 'Rodríguez', 'Vendedor', NULL, SYSDATE, -5000, NULL, 10)

Fijémonos, también, en la importancia del orden en que se definen las tablas, ya que no sería posible definir la integridad referencial en la columna dept_no de la tabla EMP sobre la tabla DEPT si ésta aún no estuviera creada.

4.5 Eliminación de tablas

La sentencia DROP TABLE es la instrucción proporcionada por el lenguaje SQL para la eliminación (datos y definición) de una tabla. La sintaxis de la sentencia DROP TABLE es ésta:

drop table [<nombre_esquema>.] <Nombre_tabla> [if exists];

La opción if exists se puede especificar para evitar un error en caso de que la tabla no exista.
También se pueden añadir las opciones cascade o restrict que en algunos SGBD hacen que se eliminen todas las definiciones de restricciones de otras tablas que hacen referencia a la tabla que se quiere eliminar antes de hacerlo, o que se impida la eliminación, respectivamente. Sin la opción cascade la tabla que es referenciada por otras tablas (a nivel de definición, independientemente de que haya o no, en un momento determinado, filas referenciadas), el SGBDR no la elimina.

En MySQL, sin embargo, no se tienen efecto las opciones cascade o restrict y siempre hay que eliminar las tablas referidas para poder eliminar la tabla referenciada.

Ejemplo de eliminación de tablas

Supongamos que queremos eliminar la tabla DEPT del esquema empresa. La ejecución de la sentencia siguiente es errónea:

drop table dept;

El SGBD informa que hay tablas que hacen referencia y que, por tanto, no se puede eliminar. Y es lógico, ya que la tabla DEPT está referenciada por la tabla EMP.

Si de verdad se quiere conseguir eliminar la tabla DEPT y provocar que todas las tablas que hacen referencia eliminen la definición correspondiente de clave foránea, habrá que eliminar EMP previamente. Y, antes de que ésta, las otras tablas que hacen referencia a esta otra.

De forma que la orden para eliminar las tablas suele ser el orden inverso en que las hemos creado.

use empresa;
drop table detalle;
….

4.6 Modificación de la estructura de las tablas

A veces, hay que hacer modificaciones en la estructura de las tablas (añadir o eliminar columnas, añadir o eliminar restricciones, modificar los tipos de datos ...).

La sentencia ALTER TABLE es la instrucción proporcionada por el lenguaje SQL para modificar la estructura de una tabla. Su sintaxis es la siguiente:

Alter [IGNORE] table [<nombre_esquema>.] <Nombre_tabla>
<clausulas_de_modificacion_de_tabla>;

Es decir, una sentencia alter table puede contener diferentes cláusulas (como mínimo una) que modifiquen la estructura de la tabla. Hay cláusulas de modificación de tabla que pueden ir acompañadas, en una misma sentencia alter table, por otras cláusulas de modificación, mientras las hay que deben ir solas.

Hay que tener presente que, para efectuar una modificación, el SGBD no debería encontrar ninguna incongruencia entre la modificación que se efectuará y los datos que ya hay en la tabla. No todos los SGBD actúan de la misma manera ante estas situaciones.

Así, el SGBD MySQL, por defecto, no permite especificar la restricción de obligatoriedad (Not null) a una columna que ya contiene valores nulos (algo lógico, no?) Ni tampoco disminuir el ancho de una columna de tipo varchar a una anchura inferior a la anchura máxima de los valores contenidos en la columna.

En cambio, sin embargo, si se activa la opción ignore, la modificación especificada se intenta hacer, aunque haya que truncar o modificar datos de la tabla ya existente. Por ejemplo, si intentamos modificar la característica not null de la columna teléfono de la tabla hospital, del esquema sanidad, dado que hay un valor nulo, la sentencia siguiente no se ejecutará:

```
Alter table hospital
modify telefono varchar (8) not null;
```

Y el resultado de la ejecución de esta modificación será un mensaje tipo Error: Fecha truncated for column telefono at row 5.

En cambio, si utilizamos la opción ignore podemos ejecutar la siguiente sentencia que nos permitirá modificar la opción not null de teléfono de manera que pondrá un string vacío en lugar de valor nulo en las columnas que no cumplan la condición:

```
Alter ignore table hospital
modify telefono varchar (8) not null;
```

De manera similar, si queremos disminuir el tamaño de la columna dirección de la tabla hospital:

```
Alter table hospital
modify direccion varchar (7);
```

Este código mostrará un error similar a Error: Fecha truncated for column direccion at row 1 y no se ejecutará la sentencia. En cambio, si añadimos la opción ignore, el

resultado será la modificación de la estructura de la tabla y el truncamiento de los valores de las columnas afectadas.

Alter ignore table hospital
modiy direccion varchar (7);

Veamos, a continuación, las diferentes posibilidades de alteración de tabla, teniendo en cuenta que en MySQL admiten varios tipos de alteraciones en una misma cláusula de alter table, separadas por comas:

1. Para añadir una columna

ADD [COLUMN] nombre_columna definicion_columna [FIRST | AFTER nombre_columna]

O bien, si es necesario definir unas cuantas nuevas:

ADD [COLUMN] (nombre_columna definicion_columna, …)

2. Para eliminar una columna

DROP [COLUMN] <nombre_columna>

3. Para modificar la estructura de una columna

MODIFY [COLUMN] nombre_columna definicion_columna [FIRST | AFTER nombre_columna]

O bien:

CHANGE [COLUMN] nombre_columna_antiguo nombre_columna_nuevo definicion_columna
[FIRST | AFTER nombre_columna]

4. Para añadir restricciones

ADD [CONSTRAINT <nombre_restriccion>] <restricción>

Concretamente, las restricciones que se pueden añadir en MySQL son las siguientes:

ADD [CONSTRAINT [símbolo]] PRIMARY KEY [tipo_indice] (nombre_columna_indice, ...) [
opciones_indice] …

ADD [CONSTRAINT [símbolo]] UNIQUE [INDEX | KEY] [nombre_indice] [tipo_indice] (
nombre_columna_indice, ...) [opciones_indice] ...

ADD [CONSTRAINT [símbolo]] FOREIGN KEY [nombre] (nombre_columna1, ...) REFERENCES tabla (Columna1, ….)

5. Para eliminar restricciones

DROP PRIMARY KEY

DROP {INDEX | KEY} nombre_indice

DROP FOREIGN KEY nombre

6. Para añadir índices

ADD {INDEX | KEY} [nombre_indice]
[tipo_indice] (nombre_columna, ...) [opciones_indice] …

7. Para habilitar o deshabilitar los índices

DISABLE KEYS

ENABLE KEYS

8. Para renombrar una tabla
RENAME [TO] nombre_nuevo_tabla

9. Para reordenar las filas de una tabla

ORDER BY nombre_columna1 [, nombre_columna2] …

10. Para cambiar o eliminar el valor predeterminado de una columna

ALTER [COLUMN] nombre_columna {SET DEFAULT literal | DROP DEFAULT}

Ejemplo 1 de modificación de la estructura de una tabla

Recordemos la estructura de la tabla DEPT del esquema empresa:

SQL> desc DEPT;

Atributo	Null	Tipo
DEPT_NO	NOT NULL	INT(2)
DNOMBRE	NOT NULL	VARCHAR(14)
LOC		VARCHAR(14)

Se quiere modificar la estructura de la tabla DEPT del esquema empresa de manera que pase lo siguiente:

• La columna loc pase a ser obligatoria.
• Añadimos una columna numérica de nombre numEmps destinada a contener el número de empleados del departamento.
• Eliminamos la obligatoriedad de la columna dnombre.
• Ampliamos el ancho de la columna dnombre a veinte caracteres.

Lo conseguimos haciendo lo siguiente:

```
alter table dept
modify loc varchar(14) not null,
add numEmps number(2) unsigned,
modify dnom varchar(20);
```

Ejemplo 2 de modificación de la estructura de una tabla, por problemas de deadlock

Una vez creada la estructura de las tablas DEPT y EMP (anteriormente) y introducidos los datos correspondientes (mediante la sentencia INSERT), para añadir la restricción que la columna jefe hace referencia a un empleado, de la misma tabla, hay que añadir la restricción siguiente:

```
ALTER TABLE empresa.EMP
ADD FOREIGN KEY (JEFE) REFERENCES EMP(EMP_NO);
```

Tenga en cuenta que no se podría haber definido esta restricción antes de insertar los valores en
las filas porque las filas insertadas ya no cumplirían la restricción que el código de su jefe fuera previamente insertado en la tabla.

4.7 Índices para tablas

Los SGBD utilizan índices para acceder de manera más rápida a los datos. Cuando hay que acceder a un valor de una columna en la que no hay definido ningún índice, el SGBD ha de consultar todos los valores de todas las columnas desde la primera hasta la última.

Esto resulta muy costoso en tiempo y, como más filas tiene la tabla en cuestión, más lenta es la operación. En cambio, si tenemos definido un índice en la columna de búsqueda, la operación de acceder a un valor concreto resulta mucho más rápido, porque no hay que acceder a todos los valores de todas las filas para encontrar lo que se busca.

MySQL utiliza índices para facilitar el acceso a columnas que son PRIMARY KEY o UNIQUE y suele almacenar los índices utilizando el tipo de índice B-tree. Para las tablas almacenadas en MEMORY utilizan, sin embargo, índices de tipo HASH.

Los índices B-tree son una organización de los datos en forma de árbol, por lo que buscar un valor de un dato resulte más rápido que buscarlo dentro de una estructura lineal que debe buscar desde el inicio hasta el final pasando por todos los valores. Los índices tipo hash tienen como objetivo acceder directamente a un valor concreto mediante una función llamada función de hash. Por lo tanto, buscar un valor es muy rápido.

Es lógico crear índices para facilitar el acceso para las columnas que necesiten accesos rápidos o muy frecuentes. El administrador del SGBD tiene, entre sus tareas, evaluar los accesos que se efectúan en la base de datos y decidir, en su caso, el establecimiento de índices nuevos. Pero también es tarea del analista y/o diseñador de la base de datos diseñar los índices adecuados para las diferentes tablas, ya que es la persona que ha ideado la tabla pensando en las necesidades de gestión que tendrán los usuarios.

La sentencia CREATE INDEX es la instrucción proporcionada por el lenguaje SQL para la creación de índices. Su sintaxis simple es esta:

create index [<nombre_esquema>.] <Nombre_índice>
on <nombre_tabla> (col1 [asc | desc], COL2 [asc | desc], ...);

Aunque la creación de un índice tiene asociadas muchas opciones, que en MySQL pueden ser las siguientes:

CREATE [UNIQUE | FULLTEXT | SPATIALl] INDEX [nombre_esquema.] Nombre_indice
[USING {BTREE | HASH}
ON nombre_tabla (columna1 [longitud] [asc | desc], ...)
[opciones_administracion];

En MySQL podemos definir índices que mantengan valores no repetidos, especificando la cláusula UNIQUE. También podemos indicar que indexen teniendo en cuenta el campo entero de una columna de tipo TEXTO, si utilizamos un motor de almacenamiento de tipo MyISAM. MySQL soporta índices sobre los tipos de datos geométricos que soporta (Spatial).

Las opciones USING BTREE y USING HASH permiten forzar la creación de un índice de un tipo u otro (índice tipo B-tree o tipo hash).

La modificación ASC o DESC sobre cada columna, sin embargo, es soportada sintácticamente apoyando el estándar SQL pero no tiene efecto: todos los índices en MySQL son ascendentes.

La sentencia DROP INDEX es la instrucción proporcionada por el lenguaje SQL para la eliminación de índices. Su sintaxis es la siguiente:

drop index [<nombre_esquema>.] <Nombre_índice> on <nombre_tabla>;

Ejemplo 1 de creación de índices. Tablas del esquema empresa

Para tener los empleados indexados por el apellido:

create index EMP_APELLIDO donde EMP (APELLIDO);

Para tener los empleados indexados por el departamento al que están asignados:

create index EMP_DEPT_NO_EMP donde EMP (DEPT_NO, EMP_NO);

Estos índices se añaden a los existentes debido a las restricciones de clave primaria y de unicidad.

Para ver todos los índices existentes sobre una tabla concreta podemos utilizar la orden:

show indexes from [nombre_esquema.] Nombre_tabla

4.8 Definición de vistas

Una vista es una tabla virtual mediante la cual se puede ver y, en algunos casos cambiar, información de una o más tablas.

Una vista tiene una estructura similar a una tabla: filas y columnas. Nunca contiene datos, sino una sentencia SELECT que permite acceder a los datos que se quieren presentar por medio de la vista. La gestión de vistas es parecida a la gestión de tablas.

La sentencia CREATE VIEW es la instrucción proporcionada por el lenguaje SQL para la creación de vistas. Su sintaxis es la siguiente:

create [or replace] view [<nombre_esquema>.] <Nombre_vista> [(col1, col2 ...)]
as <sentencia_select>
[with [cascaded | local] check option];

Como observaréis, esta sentencia es similar a la sentencia para crear una tabla a partir del resultado de una consulta. La definición de los nombres de los campos es optativa; si no se efectúa, los nombres de las columnas recuperadas pasan a ser los nombres de los campos nuevos. La sentencia SELECT puede basarse en otras tablas y/o vistas.

La opción with check option indica al SGBD que las sentencias INSERT y UPDATE que se puedan ejecutar sobre la vista deben verificar las condiciones de la cláusula where de la vista.

La opción or replace en la creación de la vista permite modificar una vista existente con una nueva definición. Hay que tener en cuenta que esta es la única vía para modificar una vista sin eliminarla y volverla a crear.

La sentencia DROP VIEW es la instrucción proporcionada por el lenguaje SQL para la eliminación de vistas. Su sintaxis es ésta, que permite eliminar una o varias vistas:

drop view [<nombre_esquema>.] <Nombre_vista> [, [<nombre_esquema>.] <Nombre_vista>];

La sentencia ALTER VIEW es la instrucción proporcionada para modificar vistas. Su sintaxis es la siguiente:

alter view <nombre_vista> [(columna1, ...)]
as <sentencia_select>;

Ejemplo 1 de creación de vistas

En el esquema empresa, se pide una vista que muestre todos los datos de los empleados acompañados del nombre del departamento al que pertenecen. La sentencia puede ser la siguiente:

create view EMPD
as select emp_no, apellido, oficio, jefe, fecha_alta, salario, comisión, e.dept_no, dnombre from emp e, dept d
where e.dept_no = d.dept_no;

Una vez creada la vista, se puede utilizar como si fuera una tabla, al menos para ejecutar sentencias SELECT:

SQL> select * from empd;

4.8.1 Operaciones de actualización sobre vistas en MySQL

Las operaciones de actualización (INSERT, DELETE y UPDATE) son, para los diversos SGBD, un tema conflictivo, ya que las vistas se basan en sentencias SELECT en que pueden intervenir muchas o pocas tablas y, incluso, otras vistas, y por tanto hay que decidir a cuál de estas tablas y/o vistas corresponde la operación de actualización solicitada.

Para cada SGBD, habrá que conocer muy bien las operaciones de actualización que permite sobre las vistas.

Cabe destacar que las vistas en MySQL pueden ser actualizables o no actualizables: las vistas en MySQL son actualizables, es decir, admiten operaciones UPDATE, DELETE o INSERT como si se tratara de una tabla. De lo contrario son vistas no actualizables.

Las vistas actualizables deben tener relaciones uno a uno entre las filas de la vista y las filas de las tablas a las que hacen referencia. Así, pues, hay cláusulas y expresiones que hacen que las vistas en MySQL sean no actualizables, por ejemplo:

• Funciones de agregación (SUM (), MIN (), MAX (), COUNT (), etc.)
• DISTINCT
• GROUP BY
• HAVING
• UNION
• Subconsultas en la sentencia select
• Algunos tipos de join
• Otras vistas no actualizables en la cláusula from
• Subconsultas en la sentencia where que hagan referencia a tablas de la cláusula FROM

Una vista que tenga varias columnas calculadas no es insertable, pero sí se pueden actualizar las columnas que contienen datos no calculados.

Ejemplo de actualización en una vista

Dada la vista anterior EMPD, si queremos modificar la comisión de un empleado concreto (emp_no = 7782) mediante la vista lo podemos hacer como sigue:

```
update empd set comision = 10000 where emp_no = 7782;
```

Con el resultado esperado, que se habrá cambiado la comisión del empleado, también, en la tabla EMP.

Si queremos cambiar, sin embargo, el nombre de departamento de este empleado (emp_no = 7782) y lo hacemos con la siguiente instrucción:

update empd set dnombre = ‘ASESORÍA CONTABLE’ where emp_no = 7782;

El resultado será que también se habrán cambiado los nombres de los departamento de contabilidad de los compañeros del mismo departamento, ya que, efectivamente, se ha cambiado el nombre del
departamento dentro de la tabla DEPT, y quizás este no es el resultado que esperábamos.

Ejemplo de eliminación e inserción en una vista

Si intentamos ejecutar una sentencia DELETE sobre la vista EMPD, el sistema no lo permitirá, al tratarse de una vista que contiene una join y, por tanto, datos de dos tablas diferentes.

delete from empd where emp_no = 7782;

Podemos ejecutar INSERT sobre la vista EMPD y tampoco lo podremos hacer.

Insert into empd values (7777, 'PLAZA', 'VENDEDOR', 7698,' 1984 05 01 ', 200000, NULL, 10, NULL);

4.9 Sentencia RENAME

La sentencia RENAME es la instrucción proporcionada por el lenguaje SQL para modificar el nombre de una o varias tablas del sistema. Su sintaxis es la siguiente:

rename <nombre_actual> to <nuevo_nombre> [, <nombre_actual2> to <nuevo_nombre2>, ...];

4.10 Sentencia TRUNCATE

La sentencia TRUNCATE es la instrucción proporcionada por el lenguaje SQL para eliminar todas las filas de una tabla. Su sintaxis es la siguiente:

TRUNCATE [table] <nombre_tabla>;

TRUNCATE es similar a delete de todas las filas (sin cláusula where). Funciona, pero, eliminando la tabla (DROP TABLE) y volviéndola a crear (CREATE TABLE).

4.11 Creación, actualización y eliminación de esquemas o bases de datos en MySQL

Recordemos que en MySQL un SCHEMA es sinónimo de DATABASE y que consiste en una agrupación lógica de objetos de base de datos (tablas, vistas, procedimientos, etc.).

Para crear una base de datos o un esquema se puede utilizar la sintaxis básica siguiente:

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] nombre_bd;
```

Para modificar una base de datos o un esquema se puede utilizar la sintaxis siguiente, que permite cambiar el nombre del directorio en el que está mapada la base de datos o el conjunto de caracteres:

```
ALTER {DATABASE | SCHEMA} nombre_bd
{UPGRADE DATA DIRECTORY NAME
| [DEFAULT] CHARACTER SET [=] nombre_charset
| [DEFAULT] COLLATE [=] nombre_collation};
```

Para eliminar una base de datos o un esquema se puede utilizar la sintaxis básica siguiente:

```
DROP {DATABASE | SCHEMA} [IF NOT EXISTS] nombre_bd;
```

4.12 Cómo se pueden conocer los objetos definidos en un esquema de MySQL

Una vez que sabemos definir tablas, vistas, índices o esquemas, y cómo modificar, en algunos casos, las definiciones existentes nos surge un problema: cómo podemos acceder de manera rápida a los objetos existentes?

La herramienta MySQL Workbench (dentro del apartado SQL Development) es una herramienta gráfica que permite, entre otras cosas, ver los objetos definidos dentro del SGBD MySQL y explorar las bases de datos que integra.

Es importante saber, también, que el SGBD MySQL nos proporciona un conjunto de tablas (que forman el diccionario de datos del SGBD) que permiten acceder a las definiciones existentes. Hay muchas, pero nos interesa conocer las de la siguiente tabla. Todas incorporan una gran cantidad de columnas, lo que hace necesario averiguar su estructura, mediante la instrucción desc, antes de intentar encontrar una información.

TABLA	CONTENIDO	EJEMPLO DE USO
information_schema.schemata	Información sobre las bases de datos del SGBD	select * from information_schema.schemata;
information_schema.tables	Información sobre las tablas de las diferentes bases de datos de MySQL	select * from information_schema.TABLES where table_schema='sanitat';
information_schema.columns	Información sobre columnas de las tablas de las diferentes bases de datos de MySQL	SELECT COLUMN_NAME, DATA_TYPE, IS_NULLABLE, COLUMN_DEFAULT FROM INFORMATION_SCHEMA.COLUMNS WHERE table_name = 'doctor' AND table_schema = 'sanitat';
information_schema.table_constraints	Información sobre las restricciones de las tablas de la base de datos	SELECT * from information_schema.TABLE_CONSTRAINTS where table_schema='sanitat' ;
information_schema.views	Información sobre las vistas de las diferentes bases de datos de MySQL	select * from information_schema.views where table_schema='empresa';
information_schema.referential_constraints	Información sobre las claves foraneas de las tablas de la base de datos	select * from information_schema.REFERENTIAL_CONSTRAINTS;

Hay formas abreviadas, pero, de mostrar la información de estas tablas del diccionario; por ejemplo, para mostrar las tablas o las columnas de las tablas, podemos utilizar estas formas simplificadas:

SHOW TABLES

SHOW COLUMNS
FROM nombre_tabla
[FROM nombre_base_datos]

5. Control de transacciones y concurrencias

Una transacción es una secuencia de instrucciones SQL que el SGBD gestiona como una unidad. Las sentencias COMMIT y ROLLBACK permiten indicar un fin de transacción.

Las transacciones en MySQL sólo tienen sentido bajo el motor de almacenamiento InnoDB, que es el único motor transaccional de MySQL. Recuerde que los otros sistemas de almacenamiento son no transaccionales y, por tanto, cada instrucción que se ejecuta es independiente y funciona, siempre, de manera autocommitiva.

Ejemplo de transacción: operación en el cajero automático

Una transacción típica es una operación en un cajero automático, por ejemplo: si vamos a un cajero automático a sacar dinero de una cuenta bancaria esperamos que si la operación termina bien (Y obtenemos el dinero extraído) se refleje esta operación en la cuenta, y, en cambio, si ha habido algún error y el sistema no nos ha podido dar

el dinero, esperamos que no se refleje esta extracción en nuestra cuenta bancaria. La operación, pues, es necesario que se considere como una unidad y termine bien (commit), sin embargo, que si acaba mal (rollback) todo quede como estaba inicialmente antes de empezar.

Una transacción habitualmente comienza en la primera sentencia SQL que se produce después de establecer conexión en la base de datos, después de una sentencia COMMIT o después de una sentencia ROLLBACK.

Una transacción finaliza con la sentencia COMMIT, con la sentencia ROLLBACK o con la desconexión (intencionada o no) de la base de datos.

Los cambios realizados en la base de datos en el transcurso de una transacción sólo son visibles para el usuario que los ejecuta. Al ejecutar una COMMIT, los cambios realizados en la base de datos pasan a ser permanentes y, por tanto, visibles para todos los usuarios.

Si una transacción finaliza con ROLLBACK, se deshacen todos los cambios realizados en la base de datos para las sentencias de la transacción.

Recordemos que MySQL tiene la autocommit definido por defecto, por lo que efectúa un COMMIT automático después de cada sentencia SQL de manipulación de datos. Para desactivarlo ejecutar:

set autocommit = 0;
Con el fin de volver a activar el sistema de autocommit:

set autocommit = 1;

Hay que tener en cuenta que una transacción sólo tiene sentido si no está definido el autocommit.
El funcionamiento de transacciones no es el mismo en todos los SGBD y, por tanto, habrá que averiguar el tipo de gestión que proporciona antes de querer trabajar.

5.1 Sentencia START TRANSACTION en MySQL

START TRANSACTION define explícitamente el inicio de una transacción. Por tanto, el código que haya entre start transaction y commit o rollback formará la transacción.

Iniciar una transacción implica un bloqueo de tablas (LOCK TABLES), así como la finalización de la transacción provoca el desbloqueo de las tablas (UNLOCK TABLES).

En MySQL start transaction es sinónimo de begin y, también, de begin work. Y la sintaxis para start transaction es la siguiente:

START TRANSACTION [WITH CONSISTENT SNAPSHOT] | BEGIN [WORK]

La opción WITH CONSISTENTE SNAPSHOT inicia una transacción que permite lecturas consistentes de los datos.

5.2 Sentencias COMMIT y ROLLBACK en MySQL

COMMIT define explícitamente la finalización esperada de una transacción. La sentencia ROLLBACK define la finalización errónea de una transacción. La sintaxis de commit y rollback en MySQL es la siguiente:

COMMIT [WORK] [AND [NO] CHAIN | [NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN | [NO] RELEASE]

La opción AND CHAIN provoca el inicio de una nueva transacción que comenzará apenas termine la actual.

La opción REALEASE causará la desconexión de la sesión actual.

Cuando se hace un rollback es posible que el sistema procese de manera lenta las operaciones, ya que un rollback es una instrucción lenta. Si se quiere visualizar el conjunto de procesos que se ejecutan se puede ejecutar la orden SHOW PROCESSLIST y visualizar los procesos que se están deshaciendo debido al rollback.

El SGBDR MySQL realiza una COMMIT implícita antes de ejecutar cualquier sentencia LDD (lenguaje de definición de datos) o LCD (lenguaje de control de datos), o en ejecutar una desconexión que no haya sido precedida de un error.

Por tanto, no tiene sentido incluir este tipo de sentencias dentro de las transacciones.

5.3 Sentencias SAVEPOINT y ROLLBACK TO SAVEPOINT en MySQL

Existe la posibilidad de marcar puntos de control (savepoints) en medio de una transacción, por lo que si se efectúa ROLLBACK este puede ser total (Toda la transacción) o hasta uno de los puntos de control de la transacción.

La instrucción SAVEPOINT permite crear puntos de control. Su sintaxis es la siguiente:

savepoint <nombre_punto_control>;

La sentencia ROLLBACK para deshacer los cambios hasta un determinado punto de control tiene esta sintaxis:

rollback [work] to [savepoint] <nombre_punto_control>;

Si en una transacción se crea un punto de control con el mismo nombre que un punto de control que ya existe, este queda sustituido por el nuevo.

Si se quiere eliminar el punto de control sin ejecutar un commit ni un rollback podemos ejecutar la siguiente instrucción:

release savepoint <nombre_punto_control>

Ejemplo de utilización de puntos de control

Consideremos la situación siguiente:

SQL> instrucción_A;
SQL> savepoint PB;
SQL> instrucción_B;
SQL> savepoint PC;
SQL> instrucción C;
SQL> instrucción_consulta_1;
SQL> rollback to PC;
SQL> instrucción_consulta_2;
SQL> rollback; o commit;

La instrucción de consulta 1 ve los cambios efectuados por las instrucciones A, B y C, pero el ROLLBACK TO PC deshace los cambios producidos desde el punto de control PC, por lo que la instrucción de consulta 2 sólo ve los cambios efectuados por las instrucciones A y B (los cambios para C han desaparecido), y el último ROLLBACK deshace todos los cambios efectuados por A y B, mientras que el último COMMIT los dejaría como permanentes.

5.4 Sentencias LOCK TABLES y UNLOCK TABLES

Con el fin de prevenir la modificación de ciertas tablas y vistas en algunos momentos, cuando se requiere acceso exclusivo a las mismas, en sesiones paralelas (o concurrentes), es posible bloquear el acceso a las tablas.

El bloqueo de tablas (LOCK TABLES) protege contra accesos inapropiados de lecturas o escrituras de otras sesiones.

La sintaxis para bloquear algunas tablas o vistas, e impedir que otros accesos puedan cambiar simultáneamente los datos, es la siguiente:
lock tables <nombre_tabla1> [[as] <alias1>] read | write
[, <nombre_tabla1> [[as] <alias1>] read | write] …

La opción read permite leer sobre la tabla, pero no escribir. La opción write permite que la sesión que ejecuta el bloqueo pueda escribir sobre la tabla, pero el resto de sesiones sólo la puedan leer, hasta que termine el bloqueo.

A veces, los bloqueos se utilizan para simular transacciones (en motores de almacenamiento que no sean transaccionales, por ejemplo) o bien para conseguir acceso más rápido a la hora de actualizar las tablas.

Cuando se ejecuta lock tables se hace un commit implícito, por lo tanto, si había alguna transacción abierta, ésta termina. Si termina la conexión (normalmente o anormalmente) antes de desbloquear las tablas, automáticamente se desbloquean las tablas.

Es posible, también, en MySQL bloquear todas las tablas de todas las bases de datos del SGBD, para hacer, por ejemplo, copias de seguridad. La sentencia que lo permite es FLUSH TABLES WITH READ LOCK.

Para desbloquear las tablas (todas las que estuvieran bloqueadas) hay que ejecutar la sentencia unlock TABLES.

5.4.1 Funcionamiento de los bloqueos

Cuando se crea un bloqueo para acceder a una tabla, dentro de esta zona de bloqueo no se puede acceder a otras tablas (a excepción de las tablas del diccionario del SGBD -information_schema-) hasta que no finalice el bloqueo. Por ejemplo:

```
mysql> LOCK TABLES t1 READ;
mysql> SELECT COUNT (*) FROM t1;
+---------------+
| COUNT (*) |
+---------------+
| 3 |
+---------------+
mysql> SELECT COUNT (*) FROM t2;
ERROR 1100 (HY000): Table 't2' was not locked with LOCK TABLES
```

No se puede acceder más de una vez en la tabla bloqueada. Si se necesita acceder dos veces en la misma tabla, hay que definir un alias para el segundo acceso a la hora de hacer el bloqueo.

```
mysql> LOCK TABLE t WRITE, t AS t1 READ;
mysql> INSERT INTO t SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> INSERT INTO t SELECT * FROM t AS t1;
```

Si se bloquea una tabla especificando un alias, hay que hacer referencia con este alias. Provoca un error acceder directamente con su nombre:

```
mysql> LOCK TABLE t AS myalias READ;
mysql> SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> SELECT * FROM t AS myalias;
```

Si se quiere acceder a una tabla (bloqueada) con un alias, hay que definir el alias en el momento de establecer el bloqueo:

```
mysql> LOCK TABLE t READ;
mysql> SELECT * FROM t AS myalias;
ERROR 1100: Table 'myalias' was not locked with LOCK TABLES
```

5.5 Sentencia SET TRANSACTION

MySQL permite configurar el tipo de transacción con la sentencia set transaction. La sintaxis para configurar las transacciones es:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}
```

Esta sentencia permite definir determinados parámetros para la transacción en curso o para la transacción siguiente que se abrirá. Las características que se definen pueden afectar globalmente (GLOBAL) o bien afectar a la sesión en curso (SESSION).

• READ UNCOMMITTED: se permite acceder a los datos de las tablas, aunque no se haya hecho un commit. Por lo tanto, es posible acceder a datos no consistentes (Dirty read).
• READ COMMITTED: sólo se permite acceder a datos que se hayan aceptado (Commit).
• REPEATABLE READ (opción por defecto): permite acceder a los datos de manera consistente dentro de las transacciones, de manera que todas las lecturas de los datos, dentro de una transacción de tipo REPEATABLE READ, permitirán obtener los datos como al inicio de la transacción, aunque ya hubieran cambiado.
• SERIALIZABLE: permite acceder a los datos de manera consistente en cualquier lectura de los datos, aunque no nos encontramos dentro de una transacción.