Base de Datos

Unidad 3: Base de datos relacionales

Curso 2022-2023

Tabla de contenido

Aparte de las conocidas instrucciones para consultar y modificar los datos, 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, así como instrucciones para definir vistas.

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

Distinción entre ámbitos DDL y DCL en el lenguaje SQL

A menudo, los ámbitos DDL (lenguaje de definición de datos) y DCL (lenguaje para el control de los datos) se funden en un único ámbito y se habla únicamente de DDL.

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 DATABASE** son sinónimos.

Disponemos, también, de la instrucción **USE <nom_base_datos>** para decidir la base de datos en la que se va a trabajar (establecimiento de la base de datos de trabajo por defecto).

1. REGLAS E INDICACIONES PARA NOMBRAR OBJETOS EN MYSQL

Dentro de MySQL, aparte de tablas, encontraremos otros tipos de objetos: **índices**, **columnas**, **alias**, **vistas**, **procedimientos**, etc.

Los nombres de los objetos dentro de una base de datos, y las propias bases de datos, 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 tendrán que seguir unas reglas sintácticas que debemos conocer.

En general, las tablas y bases de datos son **not case sensitive**, es decir, que **podemos hacer referencia a ella 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 empleado;
```

O bien:

```
select * FROM EMPLEADO;
```

Sin embargo, lo que no solemos hacer es que dentro de una misma sentencia es referirnos a un mismo objeto en mayúsculas y en minúsculas a la vez:

```
SELECT * FROM EMPLEADO WHERE empleado.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 / \y.

De todas formas, 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 "ES_OTRA_PRUEBA" (a INT) ;
```

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

```
SELECT * FROM `empresa`.`emp` WHERE `emp`.`emp_no` = 7499 ;
```

Aunque no es recomendable, pueden llamarse objetos con palabras reservadas del mismo lenguaje como **SELECT**, **INSERT**, **DATABASE**, etc. Sin embargo, estos nombres se tendrán que poner obligatoriamente entre comillas.

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

Por último, veamos algunas indicaciones para nombrar objetos:

- **Utilizar nombres enteros, descriptivos y pronunciables y, en su defecto, buenas abreviaturas**. Al nombrar objetos, sopese 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 empezar 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 nombrarlo con el nombre que tiene en la tabla principal.

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 el final de la secuecia */. Estos tipos de comentarios admiten varias líneas de comentario.

Ejemplos de los distintos 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 +
/*
Éste es un
comentario
que se puede poner
en varias líneas*/
1 ;
```

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 es necesario especificar en qué sistema de los posibles lo 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 propiedatos que garantizan los sistemas transaccionales son las características denominadas **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** (o aislamiento) : cada transacción dentro del sistema debe ejecutarse como si fuera la única que se ejecuta en ese momento.
- **Definitividad**: si se confirma una transacción, en un SGBD, el resultado de ésta 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 el que a menudo interesará utilizar para las bases de datos que creamos, pero puede haber casos en los que sea interesante considerar otros tipos de motores de almacenamiento. Por eso, MySQL también ofrece otros sistemas 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 a agrupar deben ser de 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, puede ejecutar el mandato **SHOW ENGINES**.

4. CREACIÓN DE TABLAS

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

*Recuerde que los elementos que se ponen entre **corchetes** ([]) son **opcionales**.*

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_pera_a_la_columna>],
<nombre_columna> <tipo_dato> [DEFAULT <expresión>]
[<lista_restricciones_por_a_la_columna>],
...
[<lista_restricciones_adicionales_por_a_una_o_varias_columnas>]) ;
```

Fijémonos en 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 por defecto (opción **default**) a partir de una expresión, que 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 tales como **NOW()** 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 siguiente sintaxis:

```
CREATE TABLE [<nombre_esquema>.]<nombre_tabla> [(<nombres_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 nuevos campos. 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 SGBD posibilita.
- Las restricciones sobre los nombres de tablas y columnas.
- La integridad de los datos.

*En el apartado "**Consultas de selección simples**" de la unidad "Lenguaje SQL. Consultas", se presentan ampliamente los tipos de datos más importantes en el SGBD MySQL.*

El SGBD MySQL proporciona varios tipos de restricciones (***constraints*** en la nomenclatura que debe utilizarse en los SGBD) u opciones de restricción para facilitar la integridad de los datos. Por lo 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 puede especificarse en el momento de crear la restricción. Si no se especifica, el SGBD asigna uno por defecto.

Veamos, a continuación, los distintos tipos de restricciones:

1. Clave primaria

Para definir la clave primaria de una tabla, es necesario 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, debe especificarse obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la siguiente sintaxis:

```
[CONSTRAINT <nombre_restricción>] PRIMARY KEY (col1, col2,...)
```

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

2. Obligatoriedad de valor

Para definir la obligatoriedad de valor en una columna, es necesario utilizar la opción **not null**.

Esta restricción puede indicarse 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, puesto que formar parte de la clave primaria implica, automáticamente, la imposibilidad de tener valor nulos.

3. Unicidad de valor

Para definir la unicidad valiosa en una columna, es necesario utilizar la ***constraint*** **unique**.

Si la unicidad se 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, debe especificarse obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la siguiente sintaxis:

```
[CONSTRAINT <nombre_restricción>] UNIQUE (col1, col2...)
```

Este segundo procedimiento también puede utilizarse 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, puesto que la clave primaria implica, automáticamente, la unicidad de valores.

4. Condiciones de comprobación

Para definir condiciones de comprobación en una columna, es necesario utilizar la opción **check** (<**condició**n>).

Esta restricción puede indicarse 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>)
```

5. 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 esa columna definida de este modo, se añada un valor por defecto consistente en el mayor ya introducido incrementado en una unidad.

```
<columna> <tipo_dato> AUTO_INCREMENT
```

6. *Comentarios de columnas*

Podemos añadir comentarios a las columnas de forma que puedan quedarse guardados en la base de datos y ser consultados. La forma de hacerlo es añadir la palabra **COMMENT** seguida del texto que haya que poner como comentario entre comillas simples.

```
<columna> <tipo_dato> COMMENT 'comentario'
```

7. *Integridad referencial*

Para definir la integridad referencial, es necesario utilizar la ***constraint*** **foreign key**.

Si la clave foránea 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> [CONSTRAINT <nombre_restricción>] REFERENCES
<tabla> [(columna)]
```

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

```
[CONSTRAINT <nombre_restricción>] 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. Esta sintaxis anterior es la más usada, y se suele poner después de la declaracion de las columnas.

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 se utiliza la sintaxis de **`constraint`** de la zona de definición de restricciones y, además, se define un INDEX para las columnas implicadas en la clave foránea.

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

```
INDEX [<nombre_index>] (col1, col2...)
[CONSTRAINT <nom_restricción>] FOREIGN KEY (col1, col2...) REFERENCES
<tabla> (col1, col2...)
```

La sintaxis que hemos presentado para definir la integridad referencial no está 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 existen filas en otras tablas que hacen referencia a ellas.

La ***constraint*** **foreign key** puede definirse 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 se refieren.
- **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 se refieren.

Por si no te ha quedado claro, pensamos en las tablas DEPARTAMENTO y EMPLEADO del esquema empresa. La tabla EMPLEADO contiene la columna DEPT_NO, que es clave foránea de la tabla DEPARTAMENTO. Por tanto, en la definición de la tabla EMPLEADO debemos tener definida una

constraint foreign key en la columna DEPT_NO haciendo referencia a la tabla DEPARTAMENTO. 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 DEPARTAMENTO si hay filas en la tabla EMPLEADO que hacen referencia? 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 DEPARTAMENTO si hay filas en la tabla EMPLEADO que se refieren a ella? Esto se define en el apartado **ON UPDATE *<acción>.***

Por lo general, los SGBD ofrecen diversas posibilidatos de acción, pero no siempre son las mismas. Antes de conocer estas posibilidatos, también debemos 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 no diferir la comprobación.

Por 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 <nom_restricción>] FOREIGN KEY (col1, col2,...) REFERENCES
<tabla> (col1,

col2,...) [ ON DELETE <acción>] [ ON UPDATE <acción>] _
```

Las opciones que podemos llegar a encontrar en referencia a la acción que acompañe a los apartados on update y donde 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 forma que admitan valores nulos, por supuesto.
- **SET DEFAULT**: cuando se actualiza o elimina la fila padre, las filas relacionadas (hijas) se actualizan en el valor predeterminado. MySQL soporta la sintaxis, pero no actúa, frente a esta opción.

La opción **CASCADE** es muy peligrosa al utilizarla con ***on delete***. Pensamos qué pasaría, en el esquema empresa, si alguien decidiera eliminar un departamento de la tabla DEPARTAMENTO y la clave foránea en la tabla EMPLEADO fuera definida con **on `delete cascade`**: todos los empleados del departamento serían, inmediatamente, eliminados de la tabla EMPLEADO.

A veces, sin embargo, es muy útil utilizar ***on delete***. Pensamos en la relación de integridad entre las tablas PEDIDO y DETALLE del esquema empresa. La tabla DETALLE contiene la columna *pedido_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 un pedido provocará la eliminación automática de sus líneas de detalle.

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

La siguiente tabla muestra las opciones proporcionadas por algunos SGBD actuales.

Tabla: Opciones de la restricción *foreign key* proporcionadas por algunos SGBD actuales

SGBD	on update	on delete	Dif. Act.	no action	restrict	cascade	set null	set default
Oracle	No	Sí	No	Sí	No	Sí	Sí	No

MySQL	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí
PostgreSQL	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
SQLServer 2005	Sí	Sí	No	Sí	No	Sí	Sí	Sí
MS-Access 2003	Sí	Sí	No	Sí	No	Sí	3	No

Ejemplo 1 de creación de tablas. Tablas del esquema empresa

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

```
CREATE TABLE IF NOT EXISTS empresa.EMPLEADO (
EMP_NO SMALLINT (4) UNSIGNED,
APELLIDO VARCHAR (10) NOT NULL,
CARGO VARCHAR (10),
JEFE_DE 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_CARGO (CARGO),
 INDEX IDX_EMP_DEPT_NO (DEPT_NO),
 FOREIGN KEY (DEPT_NO) REFERENCES empresa.DEPARTAMENTO (DEPT_NO)) ;
```

En primer lugar, cabe destacar la opción **IF NOT EXISTS**, opción muy utilizada a la hora de crear bases de datos para 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;
```

En ambas definiciones de tabla, la clave primaria se ha definido en la parte inferior de la sentencia, no en la misma definición de la columna, aunque se trataba de claves primarias que sólo tenían una única columna.

Fíjese cómo se han definido dos índices para las columnas JEFE_DE 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 EMPLEADO a DEPARTAMENTO. No se crea, en cambio, la que hace referencia EMPLEADO a la misma tabla y que sirve para referirse al jefe de un empleado. El motivo es que si se definiera 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 hacia el 7902 y éste no está todavía en la tabla, no podríamos insertarlo porque no existe el 7902 todavía 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 volver a activarla al terminar. Así pues, después de las inserciones habrá que modificar la tabla y añadir la restricción de clave foránea.

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

```
INSERT INTO EMPLEADO VALUES (100, ' Rodríguez ', ' Vendedor ', NULL,
sysdate, - 5000, NULL, 10)
```

Fijémonos, también, en la importancia del orden en el que se definen las tablas, puesto que no sería posible definir la integridad referencial en la columna DEPT_NO de la tabla EMPLEADOS sobre la tabla DEPARTAMENTO si ésta todavía no fuera creada.

```
CREATE TABLE IF NOT EXISTS empresa.CLIENTE (
CLIENTE_COD INT (6) UNSIGNED PRIMARY KEY,
NOMBRE VARCHAR (45) NOT NULL,
DIRECCIÓN VARCHAR (40) NOT NULL,
CIUDAD VARCHAR (30) NOT NULL,
ESTADO VARCHAR (2),
CÓDIGO_POSTAL VARCHAR (9) NOT NULL,
AREA SMALLINT (3),
TELEFON VARCHAR (9),
REPR_COD SMALLINT (4) UNSIGNED,
LIMIT_CREDIT DECIMAL (9, 2) UNSIGNED,
OBSERVACIONES TEXTO,
 INDEX IDX_CLIENT_REPR_COD (REPR_COD),
 FOREIGN KEY (REPR_COD) REFERENCES empresa.EMPLEADO (EMP_NO)) ;
```

En esta tabla, sin embargo, la clave primaria se ha definido en la misma definición de la columna.

```
CREATE TABLE IF NOT EXISTS empresa. PRODUCTO (
PROD_NUM INT (6) UNSIGNED PRIMARY KEY,
DESCRIPCIÓN VARCHAR (30) NOT NULL UNIQUE) ;
```

```
CREATE TABLE IF NOT EXISTS empresa.PEDIDO (
COM_NUM SMALLINT (4) UNSIGNED PRIMARY KEY,
```

```
COM_FECHA DATE,
COM_TIPUS CHAR (1) CHECK (COM_TIPUS IN ('A', 'B', 'C')),
CLIENTE_COD INT (6) UNSIGNED NOT NULL,
FECHA_ENVÍO DATE,
TOTAL DECIMAL (8, 2) UNSIGNED,
 INDEX IDX_COMANDA_CLIENT_COD (CLIENT_COD),
 FOREIGN KEY (CLIENTE_COD) REFERENCES empresa.CLIENTE (CLIENTE_COD)) ;
```

```
CREATE TABLE IF NOT EXISTS empresa.DETALLE (
COM_NUM SMALLINT (4) UNSIGNED,
DETALLE_NUM SMALLINT (4) UNSIGNED,
PROD_NUM INT (6) UNSIGNED NOT NULL,
PRECIO_VENTA DECIMAL (8, 2) UNSIGNED,
CANTIDAD INT (8),
IMPORTE DECIMAL (8, 2),
CONSTRAINT DETALL_PK PRIMARY KEY (COM_NUM, DETALL_NUM),
INDEX IDX_DETAL_COM_NUM (COM_NUM),
INDEX IDX_PROD_NUM (PROD_NUM),
FOREIGN KEY (COM_NUM) REFERENCES empresa.PEDIDO (COM_NUM),
FOREIGN KEY (PROD_NUM) REFERENCES empresa.PRODUCTO (PROD_NUM)) ;
```

Ejemplo 2 de creación de tablas. Tablas del esquema sanidad

```
CREATE TABLE IF NOT EXISTS sanidad.HOSPITAL (
HOSPITAL_COD TINYINT (2) PRIMARY KEY,
NOMBRE VARCHAR (10) NOT NULL,
DIRECCIÓN VARCHAR (20),
TELEFON VARCHAR (8),
NUM_CAMAS SMALLINT (3) UNSIGNED DEFAULT 0) ;
```

```
CREATE TABLE IF NOT EXISTS sanidad.SALA (
HOSPITAL_COD TINYINT (2),
SALA_COD TINYINT (2),
NOMBRE VARCHAR (20) NOT NULL,
NUM_CAMAS SMALLINT (3) UNSIGNED DEFAULT 0,
CONSTRAINT SALA_PK PRIMARY KEY (HOSPITAL_COD, SALA_COD),
INDEX IDX_SALA_HOSPITAL_COD (HOSPITAL_COD),
FOREIGN KEY (HOSPITAL_COD) REFERENCES sanidad.HOSPITAL (HOSPITAL_COD)) ;
```

Recordamos que la tabla se define con el nombre del esquema `sanidad`, pero que esto no es necesario si previamente se define este esquema como esquema por defecto:

```
USE sanidad;
```

La definición de la tabla SALA necesita declarar la constraint **PRIMARY KEY** al final de la definición de la tabla, ya que está formada por más de un campo. En casos como éste, ésta es la única opción y no es factible definir la constraint **PRIMARY KEY** junto a cada columna, ya que una tabla sólo admite una definición de constraint **PRIMARY KEY**.

```
CREATE TABLE IF NOT EXISTS sanidad.PLANTILLA (
HOSPITAL_COD TINYINT (2),
SALA_COD TINYINT (2),
EMPLEADO_NUM SMALLINT (4) NOT NULL,
NOMBRE VARCHAR (15) NOT NULL,
CARGO  VARCHAR (10),
TURNO VARCHAR (1) CHECK (TURNO IN (' M ', ' T ', ' N')),
SALARIO INT (10),
CONSTRAINT PLANTILLA_PK PRIMARY KEY (HOSPITAL_COD, SALA_COD,
EMPLEADO_NUM),
INDEX IDX_PLANTILLA_HOSP_SALA (HOSPITAL_COD, SALA_COD),
FOREIGN KEY (HOSPITAL_COD, SALA_COD) REFERENCES sanidad.SALA
(HOSPITAL_COD, SALA_COD)) ;
```

La definición de la tabla PLANTILLA necesita declarar las restricciones **PRIMARY KEY** y **FOREIGN KEY** al final de la definición de la tabla porque ambas se refieren a una combinación de columnas.

```
CREATE TABLE IF NOT EXISTS sanidad.ENFERMO (
INSCRIPCION INT (5) PRIMARY KEY,
NOMBRE VARCHAR (15) NOT NULL,
DIRECCION VARCHAR (20),
FECHA_NAC DATE,
SEXO CHAR (1) NOT NULL CHECK (SEXO = 'H' OR SEXO = 'D'),
NSS CHAR (9)) ;
```

```
CREATE TABLE IF NOT EXISTS sanidad.INGRESOS (
INSCRIPCIÓN INT (5) PRIMARY KEY,
HOSPITAL_COD TINYINT (2) NOT NULL,
SALA_COD TINYINT (2) NOT NULL,
CAMA SMALLINT (4) UNSIGNED,
INDEX IDX_INGRESOS_INSCRIPCION (INSCRIPCIÓN),
INDEX IDX_INGRESOS_HOSP_SALA (HOSPITAL_COD, SALA_COD),
FOREIGN KEY (INSCRIPCION) REFERENCES sanidad.ENFERMO (INSCRIPCION),
```

```
 FOREIGN KEY (HOSPITAL_COD, SALA_COD) REFERENCES sanidad.SALA
(HOSPITAL_COD, SALA_COD)) ;
```

```
CREATE TABLE IF NOT EXISTS sanidad.DOCTOR (
HOSPITAL_COD TINYINT (2),
DOCTOR_NO SMALLINT (3),
NOMBRE VARCHAR (13) NOT NULL,
ESPECIALIDAD VARCHAR (16) NOT NULL,
CONSTRAINT DOCTOR_PK PRIMARY KEY (HOSPITAL_COD, DOCTOR_NO),
INDEX IDX_DOCTOR_HOSP (HOSPITAL_COD),
FOREIGN KEY (HOSPITAL_COD) REFERENCES sanidad.HOSPITAL (HOSPITAL_COD)) ;
```

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** puede especificarse 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 cascadela 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 lo elimina.

Sin embargo, en MySQL 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 DEPARTAMENTO del esquema empresa.

La ejecución de la siguiente sentencia es errónea:

```
DROP TABLE DEPARTAMENTO;
```

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

Si de verdad se quiere conseguir eliminar la tabla DEPARTAMENTO y provocar que todas las tablas que hacen referencia eliminen la definición correspondiente de clave foránea, habrá que eliminar EMPLEADO previamente. Y, antes que ésta, las otras tablas que hacen referencia a esa otra. De modo que el orden para eliminar las tablas suele ser el orden inverso en el que las hemos creado.

```
USE empresa;
DROP TABLE detalle;
DROP TABLE pedido;
DROP TABLE producto;
DROP TABLE cliente;
DROP TABLE empleado;
DROP TABLE departamento;
```

6. MODIFICACIÓN DE LA ESTRUCTURA DE LAS TABLAS

En ocasiones, es necesario realizar 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 ésta:

```
ALTER [IGNORE] TABLE [<nombre_esquema>.]<nombre_tabla>
<cláusulas_de_modificación_de_tabla> ;
```

Es decir, una sentencia **alter table** puede contener diferentes cláusulas (al menos 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 que algunas deben ir solas.

Hay que tener presente que, para efectuar una modificación, el SGBD no debería encontrar ninguna incongruencia entre la modificación que debe efectuarse y los datos que ya hay en la tabla. No todos los SGBD actúan de la misma forma 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 tipos varcharen un ancho inferior al ancho máximo de los valores contenidos en la columna.

Sin embargo, si se activa la opción **ignore**, la modificación especificada se intenta hacer, aunque sea necesario truncar o modificar datos de la tabla ya existente. Por ejemplo, si intentamos modificar la característica **not null** de la columna telefono de la tabla hospital, del esquema sanidad, dado que existe 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: Data truncated for column telefon at row 5.**

En cambio, si utilizamos la opción ignorepodemos ejecutar la siguiente sentencia que nos permitirá modificar la opción **not null** de teléfono de forma 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 forma 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: Data truncated for column adreca at row 1y 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  MODIFY direccion VARCHAR (7) ;
```

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

1. ***Para añadir una columna***

```
ADD [COLUMN] nombre_columna definición_columna [FIRST | ALFTER
nombre_columna]
```

O bien, si es necesario definir unas cuantas nuevas:

```
ADD [COLUMN] (nombre_columna definición_columna,...)
```

2. ***Para eliminar una columna***

```
DROP [COLUMN]<nombre_columna>
```

3. ***Para modificar la estructura de una columna***

```
MODIFY [COLUMN] nombre_columna definición_columna [FIRST | AFTER
col_name]
```

O bien:

```
CHANGE [COLUMN] nombre_columna_antiguo nombre_columna_nuevo
definición_columna

[FIRST|AFTER nombre_columna]
```

4. ***Para añadir restricciones***

```
ADD [CONSTRAINT <nombre_restricción>]<restricción>
```

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

```
ADD [CONSTRAINT [símbolo]] PRIMARY KEY [tipo_index]
(nombre_columna_index,...) [opciones_index]...

ADD [CONSTRAINT [símbolo]] UNIQUE [INDEX|KEY] [nombre_index]
[tipo_index] (nombre_columna_index,...) [opciones_index]...

ADD [CONSTRAINT [símbolo]] FOREIGN KEY [nombre] (nombre_columna1,...)
REFERENCES tabla (columna1,....)
```

5. ***Para eliminar restricciones***

```
DROP PRIMARY KEY

DROP { INDEX | KEY } nombre_index

DROP FOREIGN KEY nombre

ADD { INDEX | KEY } [nombre_index]

 [tipo_index] (nombre_columna,...) [opciones_index]...
```

6. ***Para Deshabilitar/Habilitar la revisión de las llaves extranjeras (foreign keys)***

Para desactivarlas hay que poner:

```
SET FOREIGN_KEY_CHECKS=0;
```

Y para activarlas otra vez:

```
SET FOREIGN_KEY_CHECKS=1;
```

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

Recordamos la estructura de la tabla DEPARTAMENTO del esquema empresa :

```
SQL> DESC DPTO;
Name NULL TYPE
------------  --------     -------------
DEPT_NO       NOT NULL     TINYINT (2)
DNOM          NOT NULL     VARCHAR (14)
LOC                        VARCHAR (14)
```

Se quiere modificar la estructura de la tabla DEPARTAMENTO del esquema empresa de forma que ocurra lo siguiente:

- La columna lugar 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 nombre.
- Ampliamos el ancho de la columna de nombre a veinte caracteres.

Lo podemos conseguir haciendo lo siguiente:

```
ALTER TABLE DEPARTAMENTO
MODIFY loc VARCHAR (14) NOT NULL,
ADD numEmps NUMBER (2) UNSIGNED,
MODIFY de nombre VARCHAR (20) ;
```

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

Para crear la estructura de las tablas DEPARTAMENTO y EMPLEADO de la empresa se crean las siguientes tablas y se añaden las filas con los valores introduciendo las siguientes sentencias:

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

```
INSERT INTO empresa.DEPARTAMENTO VALUES (10, 'CONTABILIDAD', 'SEVILLA')
;
INSERT INTO empresa.DEPARTAMENTO VALUES (20, 'INVESTIGACIÓN', 'MADRID')
;
INSERT INTO empresa.DEPARTAMENTO VALUES (30, 'VENTAS', 'BARCELONA') ;
INSERT INTO empresa.DEPARTAMENTO VALUES (40, 'PRODUCCIÓN', 'BILBAO') ;
```

```
CREATE TABLE IF NOT EXISTS empresa.EMPLEADO (
EMP_NO SMALLINT (4) UNSIGNED,
NOMBRE VARCHAR (10) NOT NULL,
OFICIO VARCHAR (10),
JEFE_DE 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_CAP (JEFE_DE),
 INDEX IDX_EMP_DEPT_NO (DEPT_NO),
 FOREIGN KEY (DEPT_NO) REFERENCES empresa.DEPARTAMENTO (DEPT_NO)) ;
```

```
INSERT INTO empresa.EMPLEADO VALUES (7369, 'SÁNCHEZ', ‘EMPLEADO’, 7902,
'1980-12-17', 104000, NULL, 20) ;
INSERT INTO empresa.EMPLEADO VALUES (7499, 'ARROYO', 'VENDEDOR', 7698,
'1980-02-20', 208000, 39000, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7521, 'SALA', 'VENDEDOR', 7698,
'1981-02-22', 162500, 65000, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7566, 'JIMÉNEZ', 'DIRECTOR', 7839,
'1981-04-02', 386750, NULL, 20) ;
INSERT INTO empresa.EMPLEADO VALUES (7654, 'MARTÍN', 'VENDEDOR', 7698,
'1981-09-29', 162500, 182000, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7698, 'NEGRO', 'DIRECTOR', 7839,
'1981-05-01', 370500, NULL, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7782, 'CEREZO', 'DIRECTOR', 7839,
'1981-06-09', 318500, NULL, 10) ;
INSERT INTO empresa.EMPLEADO VALUES (7788, 'GIL', 'ANALISTA', 7566,
'1981-11-09', 390000, NULL, 20) ;
INSERT INTO empresa.EMPLEADO VALUES (7839, 'REY', 'PRESIDENTE', NULL,
'1981-11-17', 650000, NULL, 10) ;
INSERT INTO empresa.EMPLEADO VALUES (7844, 'TOBAR', 'VENDEDOR', 7698,
'1981-09-08', 195000, 0, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7876, 'ALONSO', ‘EMPLEADO’, 7788,
'1981-09-23', 143000, NULL, 20) ;
INSERT INTO empresa.EMPLEADO VALUES (7900, 'JIMENO', ‘EMPLEADO’, 7698,
'1981-12-03', 123500, NULL, 30) ;
INSERT INTO empresa.EMPLEADO VALUES (7902, 'FERNÁNDEZ', 'ANALISTA',
7566, '1981-12-03', 390000, NULL, 20) ;
INSERT INTO empresa.EMPLEADO VALUES (7934, 'MUÑOZ', ‘EMPLEADO’, 7782,
'1982-01-23', 169000, NULL, 10) ;
```

Para añadir la restricción que la columna ninguna hace referencia a un empleado, de la misma tabla, hay que añadir la restricción siguiente:

```
ALTER TABLE empresa.EMPLEADO
ADD FOREIGN KEY (JEFE_DE) REFERENCES EMPLEADO (EMP_NO) ;
```

Fíjese que no se podría haber definido esta restricción antes de insertar los valores en las filas porque ya la primera fila insertada ya no cumpliría la restricción de que el código de su cabeza fuera previamente insertado en la tabla.

7. ÍNDICES PARA TABLAS

INDEX tipo B-tree y hash

Los índices **B-tree** son una organización de los datos en forma de árbol, de modo que buscar un valor de un dato resulte más rápido que buscarlo dentro de una estructura lineal en la que deba buscarse 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 tanto, buscar un valor es muy rápido.

Los SGBD utilizan índices para acceder de forma más rápida a los datos. Cuando hay que acceder a un valor de una columna en la que no se ha definido ningún índice, el SGBD debe consultar todos los valores de todas las columnas desde la primera hasta la última. Esto resulta muy costoso en tiempo y cuanto más filas tiene la tabla en cuestión, más lenta es la operación. Sin embargo, si tenemos definido un **INDEX** en la columna de búsqueda, la operación de acceder a un valor concreto resulta **mucho más rápido**, porque no es necesario 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 **INDEX B-tree**. Sin embargo, para las tablas almacenadas en **MEMORY** se utilizan índices de tipo **HASH**.

Es lógico crear índices para facilitar el acceso a 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 nuevos índices. Pero también es tarea del analista y/o diseñador de la base de datos diseñar los índices adecuados para las distintas tablas, ya que es la persona que ha ideado la tabla pensando en las necesidatos 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 ésta:

```
CREATE INDEX [<nombre_esquema>.]<nombre_index>
USING <nombre_tabla> (col1 [ASC|DESC], col2 [ASC | DESC],...) ;
```

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

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX nombre_index
 [USING { BTREE | HASH }
 ON nombre_tabla (columna1 [longitud] [ASC | DESC],....)
 [opciones_administración] ;
```

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 tipos TEXT, si utilizamos un motor de almacenamiento de tipo **MyISAM**. MySQL soporta índices sobre los tipos de datos geométricos que soporta (**SPATIAL**).

Las opciones permiten forzar la creación de un **indice** de un tipo **USING BTRE**E u **USING HASH** otro (INDEX tipo B-tree o tipo hash).

Sin embargo, la modificación **ASC** o **DESC** sobre cada columna 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 ésta:

```
DROP INDEX [<nombre_esquema>.]<nombre_index> ON <nombre_tabla> ;
```

Ejemplo 1 de creación de índices. Tablas del esquema empresa

El diseñador de las tablas del esquema empresa consideró oportuno crear los siguientes índices:

```
-- Para tener los empleados indexados por el apellido:

CREATE INDEX EMP_APELLIDO ON EMPLEADO (APELLIDO) ;

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

CREATE INDEX EMP_DEPT_NO_EMPLEADO ON EMPLEADO (DEPT_NO, EMP_NO) ;

-- Para tener los clientes indexados por el nombre:

CREATE INDEX CLIENTE_NOMBRE ON CLIENTE (NOMBRE) ;

-- Para tener los clientes indexados por el representante (+ código de
cliente):

CREATE INDEX CLIENTE_REPR_CLI ON CLIENTE (REPR_COD, CLIENTE_COD) ;

-- Para tener los pedidos indexados por su fecha (+ número de pedido):

CREATE INDEX PEDIDO_DATA_NUM ON PEDIDO (COM_DATA, COM_NUM) ;

-- Para tener los pedidos indexados por la fecha de envío (+ número de
pedido):

CREATE INDEX PEDIDO_FECHA_ENVÍO ON PEDIDO (FECHA_ENVÍO) ;

-- Para tener las líneas de detalle indexadas por producto (+ pedido +
número de línea):

CREATE INDEX DETALLE_PROD_COM_DET ON DETALLE (PROD_NUM, COM_NUM,
DETALLE_NUM) ;
```

Todos estos índices se añaden a los existentes a causa de las restricciones de clave primaria y de unicidad.

Ejemplo 2 de creación de índices. Tablas del esquema sanidad

El diseñador de las tablas del esquema sanidad consideró oportuno crear los siguientes índices:

```
-- Para tener los hospitales indexados por el nombre:

CREATE INDEX HOSPITAL_NOM ON HOSPITAL (NOMBRE) ;

-- Para tener las salas indexadas por el nombre dentro de cada hospital:

CREATE INDEX SALA_HOSP_NOM ON SALA (HOSPITAL_COD, NOMBRE) ;

-- Para tener la plantilla indexada por apellido dentro de cada
hospital:

CREATE INDEX PLANTILLA_HOSP_APELLIDO ON PLANTILLA (HOSPITAL_COD,
APELLIDO) ;

-- Para tener la plantilla indexada por la función dentro de cada
hospital:

CREATE INDEX PLANTILLA_HOSP_FUNCIO ON PLANTILLA (HOSPITAL_COD, FUNCIO) ;

-- Para tener la plantilla indexada por la función (entre todos los
hospitales salas):

CREATE INDEX PLANTILLA_FUNCIO_HOSP_SALA ON PLANTILLA (FUNCIO,
HOSPITAL_COD, SALA_COD) ;

-- Para tener los enfermos indexados por fecha de nacimiento y apellido:

CREATE INDEX ENFERMO_NACIDO_APELLIDO ON ENFERMO (FECHA_NACIDO, APELLIDO)
;

-- Para tener los enfermos indexados por apellido y fecha de nacimiento:

CREATE INDEX ENFERMO_APELLIDO_NACE ON ENFERMO (APELLIDO, FECHA_NACIDO) ;

-- Para tener los ingresados indexados por hospital sala:

CREATE INDEX INGRESOS_HOSP_SALA ON INGRESOS (HOSPITAL_COD, SALA_COD) ;

-- Para tener los doctores indexados por su especialidad (entre todos
los hospitales):

CREATE INDEX DOCTOR_ESP_HOSP ON DOCTOR (ESPECIALIDAD, HOSPITAL_COD) ;
```

```
-- Para tener los doctores indexados por su especialidad dentro de cada
hospital:

CREATE INDEX DOCTOR_HOSP_ESP ON DOCTOR (HOSPITAL_COD, ESPECIALIDAD) ;
```

Todos estos índices se añaden a los existentes a causa de las restricciones de clave primaria y de unicidad. Para visualizar todos los índices existentes sobre una tabla concreta podemos utilizar el comando:

```
SHOW indexes FROM [nombre_esquema.] nombre_tabla
```

En el caso de la tabla doctor del esquema sanidad, podemos observar el resultado visualizado en la herramienta Workbench de MySQL, en la figura.

8. DEFINICIÓN DE VISTAS

Las vistas se corresponden con los distintos tipos de consultas que proporciona el SGBDR MS-Access.

Una **vista** es una tabla virtual por la que 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 similar 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 ésta:

```
CREATE [OR  REPLACE] VIEW [<nombre_esquema>.]<nombre_vista> [(col1, col2...)]
AS <sentencia_select>
[WITH [cascaded | LOCAL] CHECK OPTION] ;
```

Como observará, 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 nuevos campos. La sentencia SELECT puede basarse en otras tablas y/o vistas.

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

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

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 por **modificar vistas**.

Su sintaxis es ésta:

```
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, cargo, jefe_de, fecha_alta, salario,
comision, e. DEPT_NO, nombre
 FROM EMPLEADO e, DEPARTAMENTO d
 WHERE e. DEPT_NO = d. DEPT_NO;
```

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

```
SQL> SELECT * FROM empd;

EMP_NO APELLIDO CARGO JEFE_DE FECHA_ALTA SALARIO COMISIÓN DEPT_NO DNOM
------- ---------- ---------- ---- ----------- ------- - --------- -------- ----
------------
7369 SÁNCHEZ EMPLEADO  7902   17 / 12 / 1980 104000 20 INVESTIGACIÓN
7499 ARROYO VENDEDOR   7698   20 / 02 / 1980 208000 39000 30 VENTAS
7521 SALA VENDEDOR     7698   22 / 02 / 1981 162500 65000 30 VENTAS
7566 JIMÉNEZ DIRECTOR  7839   02 / 04 / 1981 386750 20 INVESTIGACIÓN
7654 MARTÍN VENDEDOR   7698   29 / 09 / 1981 162500 182000 30 VENTAS
7698 NEGRO DIRECTOR    7839   01 / 05 / 1981 370500 30 VENTAS
7782 CEREZO DIRECTOR   7839   09 / 06 / 1981 318500 10 CONTABILIDAD
7788 GIL ANALISTA      7566   09 / 11 / 1981 390000 20 INVESTIGACIÓN
7839 REY PRESIDENTE           17 / 11 / 1981 650000 10 CONTABILIDAD
7844 ZOBAR VENDEDOR    7698   08 / 09 / 1981 195000 0 30 VENTAS
7876 ALONSO EMPLEADO   7788   23 / 09 / 1981 143000 20 INVESTIGACIÓN
7900 JIMENO EMPLEADO   7698   03 / 12 / 1981 123500 30 VENTAS
7902 FERNÁNDEZ ANALISTA7566    03 / 12 / 1981 390000 20 INVESTIGACIÓN
7934 MUÑOZ EMPLEADO    7782   23 / 01 / 1982 169000 10 CONTABILIDAD
```

Ejemplo 2 de creación de vistas

En el esquema empresa, se solicita una vista para visualizar los departamentos de código par.

La sentencia puede ser ésta:

```
CREATE VIEW DEPARTAMENTO_PAR
AS SELECT * FROM DEPARTAMENTO WHERE MOD (DEPT_NO, 2) = 0 ;
```

8.1. Operaciones de actualización sobre vistas en MySQL

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

Para cada SGBD, será necesario 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 o DELETEcomo INSERT 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, existen 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 sentenciaselect
- Algunos tipos dejoin
- Otras vistas no actualizables en la cláusulafrom
- Subconsultas en la sentencia whereque hagan referencia a tablas de la cláusulaFROM

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

Ejemplo de actualización en una vista

Recordamos una de las vistas creadas sobre el esquema empresa :

```
CREATE VIEW EMPD
AS SELECT emp_no, apellido, cargo, jefe_de, fecha_alta, salario,
comision, e. DEPT_NO, nombre
 FROM EMPLEADO e, DEPARTAMENTO d
 WHERE e. DEPT_NO = d. DEPT_NO;
```

Si queremos modificar la comisión de un empleado concreto (emp_no=7782) mediante la vista EMPD lo podemos hacer tal y 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 EMPLEADO.

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

```
UPDATE empd SET dnom = '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 de DEPARTAMENTO, y quizás éste no es el resultado que esperábamos.

Ejemplo de eliminación e inserción en una vista

Recordamos la vista EMPD creada sobre el esquema empresa :

```
CREATE VIEW EMPD
AS SELECT emp_no, apellido, oficio, ninguno, fecha_alta, salario,
comision, e. DEPT_NO, nombre
 FROM EMPLEADO e, DEPARTAMENTO d
 WHERE e. DEPT_NO = d. DEPT_NO;
```

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 podremos hacerlo.

INSERT INTO empd VALUES (7777, 'PLAZA', 'VENDEDOR', 7698, '1984-05-01', 200000, NULL, 10, NULL) ;

Ejemplo de operaciones de actualización sobre vistas

Recordamos la vista DEPARTAMENTO_PAR creada sobre el esquema empresa :

```
CREATE VIEW DEPARTAMENTO_PAR
AS SELECT * FROM DEPARTAMENTO WHERE MOD (DEPT_NO, 2) = 0 ;
```

Ahora vamos a efectuar algunas inserciones, algunos borramientos y algunas modificaciones de departamentos pares por medio de la vista DEPARTAMENTO_PARELL.

```
INSERT INTO DEPARTAMENTO_PAR VALUES (60, 'INFORMÁTICA', 'BARCELONA') ;
```

Esta instrucción provoca la inserción de una fila sin ningún problema en la tabla DEPARTAMENTO.

```
INSERT INTO DEPARTAMENTO_PAR VALUES (55, ' ALMACÉN ', ' LEIDA ') ;
```

Esta instrucción provoca la inserción de una fila sin ningún problema, pero esta inserción no se produciría si la vista hubiera sido creada con la opción **with check option**, ya que en esta situación los departamentos insertados en la tabla DEPARTAMENTO por medio de la vista DEPARTAMENTO_PAR deberían verificar la cláusula where de la definición de la vista.

Podemos comprobar que si hacemos esta modificación, nos da un error en la inserción de un código no par:

```
CREATE OR REPLACE VIEW DEPARTAMENTO_PAR
 AS SELECT * FROM DEPARTAMENTO WHERE MOD (DEPT_NO, 2) = 0 WITH CHECK
OPTION ;
```

```
INSERT INTO DEPARTAMENTO_PAR VALUES (65, ' ALMACÉN2 ', ' GIRONA ') ;
```

La siguiente instrucción provoca error porque se ha definido la opción **with check option**, ya que en esta situación el departamento 50 ha sido seleccionado pero no ha

podido cambiar a 51 porque no cumple la cláusula where de la vista. Si volvemos a evitar la opción **with check option** en la definición de la vista, la actualización del departamento 50 (seleccionable por la vista, al ser par) hacia departamento 51 no dará ningún error y realizará el cambio de código:

```
CREATE OR REPLACE VIEW DEPARTAMENTO_PAR
 AS SELECT * FROM DEPARTAMENTO WHERE MOD (DEPT_NO, 2) = 0 ;
```

```
UPDATE DEPARTAMENTO_PAR SET DEPT_NO = DEPT_NO + 1 WHERE DEPT_NO = 50 ;
DELETE FROM DEPARTAMENTO_PAR WHERE DEPT_NO IN (50, 55);
```

Esta instrucción no borra ningún departamento, puesto que el 50 no existe (lo hemos cambiado a 51), y el 55 existe pero no es seleccionable por medio de la vista ya que no es par.

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 ésta:

```
RENAME <nombre_actual> TO <nuevo_nombre> [, <nombre_actual2> TO
<nuevo_nombre2>,....] ;
```

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 ésta:

```
TRUNCATE [TABLE]<nombre_tabla> ;
```

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

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 siguiente sintaxis básica:

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

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

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

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

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

12.Cómo 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 forma rápida a los objetos existentes?

La herramienta **MySQL Workbench** 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 SGBD MySQL nos proporciona un conjunto de tablas (que forman el diccionario de datos de SGBD) que permiten acceder a las definiciones existentes. Hay muchas, pero nos interesa conocer las de la tabla. Todas ellas incorporan una gran cantidad de columnas, lo que hace necesario averiguar su estructura, por medio de la instrucción desc, antes de intentar encontrar una información.

Tabla: Vistas de SGBD MySQL que proporcionan información sobre los objetos definidos en el esquema

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 distintas bases de datos de MySQL.	select * from information_schema.TABLES where table_schema='sanidad';
information_schema.columns	Información sobre columnas de las tablas de las distintas 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 = 'sanidad';
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='sanidad' ;
information_schema.views	Información sobre las vistas de las distintas bases de datos de MySQL.	select * from information_schema.views where table_schema='empresa';
information_schema.referential_constraints	Información sobre las claves foráneas de las tablas de la base de datos.	select * from information_schema.REFERENTIAL_CONSTRAINTS;

Sin embargo, hay formas abreviadas 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];
```