

## 2. DDL

---

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, 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 <nom\_base\_dades> 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.

### **Distinción entre ámbitos LDD y LCD en el lenguaje SQL**

A menudo, los ámbitos LDD (lenguaje de definición de datos) y LCD (lenguaje para el control de los datos) se funden en un único ámbito y se habla únicamente de 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 <nom\_base\_dades> para decidir la base de datos en la que se trabajará (establecimiento de la base de datos de trabajo por defecto).

### 2.1. Reglas e indicaciones para nombrar objetos en MySQL

Dentro de MySQL, además de mesas, 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 mesa 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 / \y.

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 `SE UNA PROVA` ( 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 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, sin embargo, 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, sopesar 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 mesas** . Así, por ejemplo, cuando un atributo de una mesa es clave foránea de otra mesa, es muy conveniente llamarlo con el nombre que tiene en la mesa principal.

## 2.2. Comentarios en MySQL

El servidor MySQL soporta tres estilos de comentarios:

- # hasta el final de la línea.
- – <espai en blanc> hasta el final de la línea.
- /\* fins a la propera seqüència \*/. 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 ;
```

---

## 2.3. 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 hay que especificar 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 (o 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 ello, 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, puede ejecutar la orden SHOW ENGINES.

## 2.4. Creación de tablas

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

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 >
( < Nom_columna > < tipus_dada > [DEFAULT < expresión >] [ < llista_restriccions_pera_a_la_columna > ],
  < Nom_columna > < tipus_dada > [DEFAULT < expresión >] [ < llista_restriccions_per_a_la_columna > ],
  ...
  [ < llista_restriccions_addicionals_per_a_una_o_vàries_columnes > ] );
```

---

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 mesa es optativo y si no se indica, la mesa 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 siguiente sintaxis:

---

```
CREATE TABLE [< nombre_esquema >.] < Nombre_tabla > [ ( < noms_dels_camps > ) ]
AS < sentència_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.

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 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 un predeterminado.

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 > < tipus_dada > 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 siguiente sintaxis:

---

```
[ CONSTRAINT < nom_restricció > ] 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 > < tipus_dada > [ 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 > < tipus_dada > UNIQUE
```

---

En cambio, si la unicidad aplica sobre varias columnas simultáneamente, debe especificarse obligatoriamente en la zona final de restricciones sobre columnas de la tabla, con la siguiente sintaxis:

---

```
[ CONSTRAINT < nom_restricció > ] UNIQUE ( col1 , COL2 ... )
```

---

Este segundo procedimiento también se puede emplear para aplicar la unicidad en 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 > < tipus_dada > 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 añade un valor predeterminado consistente en el mayor ya introducido incrementado en una unidad.

---

```
< Columna > < tipus_dada > 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 > < tipus_dada > COMMENT 'comentario'
```

---

#### integridad referencial

Para definir la integridad referencial, hay que utilizar la *constraint* foreign key .

Si la clave foránea es formada por una única columna, se puede especificar en la línea de definición de la columna correspondiente, con la siguiente sintaxis:

---

```
< Columna > < tipus_dada > [ CONSTRAINT < nom_restricció > ] REFERENCES < tabla > [ ( columna ) ]
```

---

En cambio, si la clave foránea es 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 < nom_restricció > ] 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 <taula>.

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 se utilizan otros sintaxis, el SGBD las reconoce pero no las valida):



---

```
INDEX [< nom_index >] ( col1 , COL2 ... )  
[ CONSTRAINT < nom_restricció > ] 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 mesa 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 se refieren? 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 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 < nom_restricció > ] 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 **donde update** y **donde delete** son estas: RESTRICT | CASCADE | SET NULL | NO ACTION

- **NO ACTION** **NO 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 haberlas definido de manera que admitan valores nulos, claro.
- **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**. Pensamos 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 **com\_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**.

La [tabla 2.1](#) muestra las opciones proporcionadas por algunos SGBD actuales.

**Tabla 2.1.** Opciones de la restricción foreign key proporcionadas por algunos SGBD actuales

SGBD	donde update	donde delete	diferir actuación	no action	restrict	cascade	SET NULL	septiembre 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 . DEPT (
  DEPT_NO tiña ( 2 ) UNSIGNED ,
  DNOM   VARCHAR ( 14 ) NOT NULL UNIQUE ,
  LOC   VARCHAR ( 14 ) ,
  PRIMARY KEY ( DEPT_NO ) );
```

```
CREATE TABLE IF NOT EXISTS empresa . EMP (
EMP_NO    small ( 4 ) UNSIGNED ,
APELLIDO  VARCHAR ( 10 ) NOT NULL ,
OFICIO    VARCHAR ( 10 ) ,
CAP       small ( 4 ) UNSIGNED ,
DATA_ALTA DATE ,
SALARIO   INT UNSIGNED ,
COMISION  INT UNSIGNED ,
DEPT_NO   tiny ( 2 ) UNSIGNED NOT NULL ,
PRIMARY KEY ( EMP_NO ) ,
INDEX IDX_EMP_CAP ( CAP ) ,
INDEX IDX_EMP_DEPT_NO ( DEPT_NO ) ,
FOREIGN KEY ( DEPT_NO ) REFERENCES empresa . DEPT ( DEPT_NO ) ) ;
```

En primer lugar, cabe 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 mesa 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 predeterminado. Esto se puede hacer con la sentencia USE:

```
USE empresa;
```

En las dos definiciones de mesa, 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.

Fijense como se han definido dos índices para las columnas CAP 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 DEPT. No se crea, en cambio, la que hace referencia de EMP a la misma mesa y que sirve para referirse al cabo 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 hacia el 7902 y este no es todavía en la mesa, no lo podríamos insertar porque no existe el 7902 todavía en la mesa. 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\_node de la tabla EMP sobre la mesa DEPT si ésta aún no fuera creada.

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

En esta tabla, en cambio, 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 ,
DESCRIPCION   VARCHAR ( 30 ) NOT NULL UNIQUE );

CREATE TABLE IF NOT EXISTS empresa . PEDIDO (
COM_NUM        small ( 4 ) UNSIGNED PRIMARY KEY ,
COM_DATA       DATE ,
COM_TIPUS      CHAR ( 1 ) CHECK ( COM_TIPUS IN ( 'A' , 'B' , 'C' ) ) ,
CLIENT_COD     INT ( 6 ) UNSIGNED NOT NULL ,
DATA_TRAMESA   DATE ,
TOTAL         DECIMAL ( 8 , 2 ) UNSIGNED ,
INDEX IDX_COMANDA_CLIENT_COD ( CLIENT_COD ) ,
FOREIGN KEY ( CLIENT_COD ) REFERENCES empresa . CLIENTE ( CLIENT_COD ) );

CREATE TABLE IF NOT EXISTS empresa . DETALLE (
COM_NUM        small ( 4 ) UNSIGNED ,
DETALL_NUM     small ( 4 ) UNSIGNED ,
PROD_NUM       INT ( 6 ) UNSIGNED NOT NULL ,
PREU_VENDA     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   tiny ( 2 ) PRIMARY KEY ,
NOMBRE        VARCHAR ( 10 ) NOT NULL ,
DIRECCIÓN     VARCHAR ( 20 ) ,
TELEFONO      VARCHAR ( 8 ) ,
QTAT_LLITS    small ( 3 ) UNSIGNED DEFAULT 0 );

CREATE TABLE IF NOT EXISTS sanidad . SALA (
HOSPITAL_COD   tiny ( 2 ) ,
SALA_COD       tiny ( 2 ) ,
NOMBRE        VARCHAR ( 20 ) NOT NULL ,
QTAT_LLITS    small ( 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 ) );
```

---

Recordemos que la tabla se define con el nombre del esquema `sanitat`, pero que esto no es necesario si previamente se define este esquema como esquema predeterminado:

---

```
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 este, esta es la única opción y no es factible definir la constraint PRIMARY

KEY junto a cada columna, ya que una mesa sólo admite una definición de constraint PRIMARY KEY.

---

```
CREATE TABLE IF NOT EXISTS sanidad . PLANTILLA (
HOSPITAL_COD tiña ( 2 ),
SALA_COD tiña ( 2 ),
EMPLEAT_NO small ( 4 ) NOT NULL ,
APELLIDO VARCHAR ( 15 ) NOT NULL ,
FUNCION VARCHAR ( 10 ),
TURNO VARCHAR ( 1 ) CHECK ( TURNO IN ( 'M', 'T', 'N' ) ),
SALARIO INT ( 10 ),
CONSTRAINT PLANTILLA_PK PRIMARY KEY ( HOSPITAL_COD , SALA_COD , EMPLEAT_NO ),
INDEX IDX_PLANTILLA_HOSP_SALA ( HOSPITAL_COD , SALA_COD ),
FOREIGN KEY ( HOSPITAL_COD , SALA_COD ) REFERENCES sanidad . SALA ( HOSPITAL_COD , SAL
```

La definición de la tabla PLANTILLA necesita declarar las restricciones PRIMARY KEY y FOREIGN KEY al final de la definición de la mesa para que ambas hacen referencia a una combinación de columnas.

---

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

CREATE TABLE IF NOT EXISTS sanidad . INGRESOS (
INSCRIPCION INT ( 5 ) PRIMARY KEY ,
HOSPITAL_COD tiña ( 2 ) NOT NULL ,
SALA_COD tiña ( 2 ) NOT NULL ,
CAMA small ( 4 ) UNSIGNED ,
INDEX IDX_INGRESSOS_INSCRIPCION ( INSCRIPCION ),
INDEX IDX_INGRESSOS_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_COI

CREATE TABLE IF NOT EXISTS sanidad . DOCTOR (
HOSPITAL_COD tiña ( 2 ),
DOCTOR_NO small ( 3 ),
APELLIDO 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 ) );
```

---

## 2.5. Eliminación de tablas

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

La sintaxis de la sentencia DROP TABLE es esta:

---

```
DROP TABLE [ < nombre_esquema > . ] < Nombre_tabla > [ IF EXISTS ] ;
```

---

La opción `if exists` puede especificar para evitar un error en caso de que la mesa no exista.

También se pueden añadir las opciones `cascade` 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` de la mesa que es referenciada por otras tablas (a nivel de definición, independientemente de que haya o no, en un momento determinado, filas referenciadas), el SGBD no la elimina.

En MySQL, sin embargo, no se tienen efecto las opciones `cascade` restrict 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 mesa `DEPT` está referenciada por la mesa `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;  
DROP TABLE pedido;  
DROP TABLE producto;  
DROP TABLE cliente;  
DROP TABLE emp;  
DROP TABLE dept;
```

---

## 2.6. 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 >  
< Cláusulas_de_modificació_de_taula >;
```

---

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 mesa que pueden ir acompañadas, en una misma sentencia alter table, por otras cláusulas de modificación, mientras que 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 mesa. 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 sea necesario 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: Data truncated for column telefon 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  
modiy 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  
modiy direccion varchar ( 7 );
```

---

Veamos, a continuación, las diferentes posibilidades de alteración de mesa, 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 ] nom_columna definició_columna [ FIRST | After nom_columna ]
```

---

O bien, si es necesario definir unas cuantas nuevas:

---

```
ADD [COLUMN] (nom_columna definició_columna, ...)
```

---

### 2. Para eliminar una columna

---

```
DROP [ COLUMN ] < nom_columna >
```

---

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

---

```
MODIFY [ COLUMN ] nom_columna definició_columna [ FIRST | AFTER col_name ]
```

---

O bien:

---

```
CHANGE [COLUMN] nom_columna_antic nom_columna_nou definició_columna  
[FIRST | AFTER nom_columna]
```

---

### 4. Para añadir restricciones

---

```
ADD [ CONSTRAINT < nom_restricció > ] < restricció >
```

---

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

---

```
ADD [CONSTRAINT [símbolo]] PRIMARY KEY [tipus_index] (nom_columna_index, ...) [opcions_index] ...  
ADD [CONSTRAINT [símbolo]] UNIQUE [INDEX | KEY] [nom_index] [tipus_index] (nom_columna_index, ...) [op  
ADD [CONSTRAINT [símbolo]] FOREIGN KEY [nombre] (nom_columna1, ...) REFERENCES tabla (columna1, ..
```

---

### 5. Para eliminar restricciones

---

```
DROP PRIMARY KEY  
  
DROP { INDEX | KEY } nom_index
```

---



`DROP FOREIGN KEY` nombre

---

## 6. Para añadir índices

---

```
ADD { INDEX | KEY } [ nom_index ]  
[ Tipus_index ] ( nom_columna , ... ) [ opciones_index ] ...
```

---

## 7. Para habilitar o deshabilitar los índices

---

`DISABLE KEYS`

`ENABLE KEYS`

---

## 8. Para renombrar una tabla

---

`RENAME [ TO ] nom_nou_taula`

---

## 9. Para reordenar las filas de una tabla

---

`ORDER BY` nom\_columna1 [ , nom\_columna2 ] ...

---

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

---

`ALTER [ COLUMN ] nom_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;  
Name      NULL  TYPE  
-----  
DEPT_NO    NOT NULL Tiny ( 2 )  
DNOM       NOT NULL VARCHAR ( 14 )  
LOC        VARCHAR ( 14 )
```

---

Se quiere modificar la estructura de la tabla DEPT del esquema empresa de manera que pase el 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 nombre.
- Ampliamos la anchura de la columna dnom a veinte caracteres.

Lo podemos conseguir haciendo lo siguiente:

---

```
ALTER TABLE dept  
MODIFY loc VARCHAR ( 14 ) NOT NULL ,  
ADD numEmps NUMBER ( 2 ) UNSIGNED ,  
MODIFY dnom VARCHAR ( 20 );
```

---

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

Tenga en cuenta 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 que el código de su cabeza fuera previamente insertado en la tabla.

---

## 2.7. Índices para mesas

### **Índice tipo B-tree y 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 buscarla dentro de una estructura lineal en que se haya de 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.

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 debe 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.

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 mesas, ya que es la persona que ha ideado la mesa 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 >. ] < Nom_index >  
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 | Spatial ] INDEX nom_index  
[ USING { BTREE | HASH }  
ON nombre_tabla ( columna1 [ longitud ] [ ASC | DESC ], .... )  
[ Opciones_administració ] ;
```

---

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

Las opciones USING BTREEy USING HASHpermiten forzar la creación de un índice de un tipo u otro (índice tipo B-tree o tipo hash).

La modificación ASCo DESCsobre 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 INDEXes la instrucción proporcionada por el lenguaje SQL para la eliminación de índices.

Su sintaxis es la siguiente:

---

```
DROP INDEX [ < nombre_esquema >. ] < Nom_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 índices siguientes:

---

- Para tener los empleados indexados por el apellido:

```
CREATE INDEX EMP_COGNOM ON EMP ( APELLIDO );
```

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

```
CREATE INDEX EMP_DEPT_NO_EMP ON EMP ( DEPT_NO , EMP_NO );
```

- Para tener los clientes indexados por el nombre:

```
CREATE INDEX CLIENT_NOM ON CLIENTE ( NOMBRE );
```

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

```
CREATE INDEX CLIENT_REPR_CLI ON CLIENTE ( REPR_COD , CLIENT_COD );
```

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

```
CREATE INDEX COMANDA_DATA_NUM ON PEDIDO ( COM_DATA , COM_NUM );
```

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

```
CREATE INDEX COMANDA_DATA_TRAMESA ON PEDIDO ( DATA_TRAMESA );
```

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

```
CREATE INDEX DETALL_PROD_COM_DET ON DETALLE ( PROD_NUM , COM_NUM , DETALL_NUM );
```

---

Todos estos índices se añaden a los existentes debido a 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 índices siguientes:

---

- 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 en cada hospital:

```
CREATE INDEX PLANTILLA_HOSP_COGNOM 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 , FUNCION );
```

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

```
CREATE INDEX PLANTILLA_FUNCIO_HOSP_SALA ON PLANTILLA ( FUNCION , HOSPITAL_COD , SALA_COD );
```

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

```
CREATE INDEX MALALT_NAIX_COGNOM ON ENFERMO ( DATA_NAIX , APELLIDO );
```

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

```
CREATE INDEX MALALT_COGNOM_NAIX ON ENFERMO ( APELLIDO , DATA_NAIX );
```

- Para tener los ingresos indexados por hospital sala:

```
CREATE INDEX INGRESSOS_HOSP_SALA ON INGRESOS ( HOSPITAL_COD , SALA_COD );
```

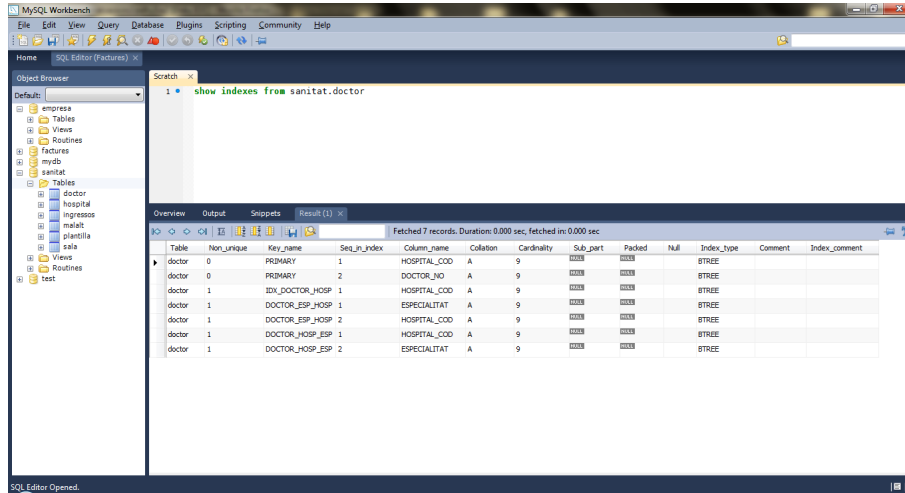
- Para tener los doctores indexados por su especialidad (entre todos los hospitales):

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

- Para tener los doctores indexados por su especialidad en cada hospital:

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

**Figura 2.1.** Índices de la tabla 'doctor' mostrados a través de Workbench de MySQL



The screenshot shows the MySQL Workbench interface. The SQL Editor contains the command 'show indexes from sanitat.doctor'. The Results window displays the output of this command, showing 7 records. The table has columns: Table, Non\_unique, Key\_name, Seq\_in\_index, Column\_name, Collation, Cardinality, Sub\_part, Packed, Null, Index\_type, Comment, and Index\_comment. The data shows various indexes on the 'doctor' table, including a primary key and several unique indexes.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
doctor	0	PRIMARY	1	HOSPITAL_COD	A	9				BTREE		
doctor	0	PRIMARY	2	DOCTOR_NO	A	9				BTREE		
doctor	1	IDX_DOCTOR_HOSP	1	HOSPITAL_COD	A	9				BTREE		
doctor	1	DOCTOR_ESP_HOSP	1	ESPECIALTAT	A	9				BTREE		
doctor	1	DOCTOR_ESP_HOSP	2	HOSPITAL_COD	A	9				BTREE		
doctor	1	DOCTOR_HOSP_ESP	1	HOSPITAL_COD	A	9				BTREE		
doctor	1	DOCTOR_HOSP_ESP	2	ESPECIALTAT	A	9				BTREE		

Todos 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 mesa concreta podemos utilizar el comando:

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

En el caso de la mesa doctor del esquema sanitat, podemos observar el resultado visualizado en la herramienta Workbench de MySQL, en la [figura 2.1](#).

## 2.8. Definición de vistas

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

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 VIEWes la instrucción proporcionada por el lenguaje SQL para la creación de vistas.

Su sintaxis es la siguiente:

---

```
CREATE [OR REPLACE] VIEW [< nombre_esquema >.] < Nom_vista > [( col1 , COL2 ... )]  
AS < sentència_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 optionindica al 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 replaceen 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 VIEWes 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 >.] < Nom_vista > [, [< nombre_esquema >.] < Nom_vista >];
```

---

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

---

```
alter view <nom_vista> [(columna1, ....)]  
as <sentència_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 , ninguna , data_alta , salario , comisión , y . dept_no , dnom  
FROM emp y , dept de  
WHERE y . dept_no = d . dept_no;
```

---

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

---

```
SQL > SELECT * FROM empd;
```

```
EMP_NO APELLIDO OFICIO CAP DATA_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 / 1.981	162 500	65000	30 VENTAS
7566	JIMÉNEZ DIRECTOR	7839	02 / 04 / 1981	386 750	20	INVESTIGACIÓN
7654	MARTÍN VENDEDOR	7698	29 / 09 / 1981	162 500	182000	30 VENTAS
7698	NEGRO DIRECTOR	7839	01 / 05 / 1981	370 500	30	VENTAS
7782	CEREZO DIRECTOR	7839	09 / 06 / 1981	318.500	10	CONTABILIDAD
7788	GIL ANALISTA	7566	09 / 11 / 1981	390000	20	INVESTIGACIÓN
7839	REY PRESIDENTE	17	11 / 11 / 1981	650000	10	CONTABILIDAD
7844	TOVAR 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	123 500	30	VENTAS
7902	FERNÁNDEZ ANALISTA	7566	03 / 12 / 1981	390000	20	INVESTIGACIÓN
7934	MUÑOZ EMPLEADO	7782	23 / 01 / 1.982	169000	10	CONTABILIDAD

---

## Ejemplo 2 de creación de vistas

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

La sentencia puede ser esta:

---

```
CREATE VIEW DEPT_PARELL  
AS SELECT * FROM DEPT WHERE MOD ( dept_no , 2 ) = 0 ;
```

---

### 2.8.1. 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 SELECTen que pueden intervenir muchas o pocas mesas 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, DELETEo INSERTcomo si se tratara de una mesa. 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 calculadas.

---

## Ejemplo de actualización en una vista

Recordemos una de las vistas creadas sobre el esquema *empresa* :

---

```
CREATE VIEW EMPD
AS SELECT emp_no , apellido , oficio , ninguna , data_alta , salario , comisión , y . dept_no , dnom
FROM emp y , dept de
WHERE y . dept_no = d . dept_no;
```

---

Si queremos modificar la comisión de un empleado concreto ( emp\_no=7782) mediante la vista EMPD lo podemos hacer como sigue:

---

```
UPDATE empd SET comisión = 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 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 DEPT, y quizás este no es el resultado que esperábamos.

---

## Ejemplo de eliminación e inserción en una vista

Recordemos la vista EMPD creada sobre el esquema *empresa* :

---

```
CREATE VIEW EMPD
AS SELECT emp_no , apellido , oficio , ninguna , data_alta , salario , comisión , y . dept_no , dnom
FROM emp y , dept de
WHERE y . 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 lo podremos hacer.

---

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

---

---

## Ejemplo de operaciones de actualización sobre vistas

Recordemos la vista DEPT\_PARELL creada sobre el esquema *empresa* :

---

```
CREATE VIEW DEPT_PARELL  
AS SELECT * FROM DEPT WHERE MOD ( dept_no , 2 ) = 0 ;
```

---

Ahora efectuaremos algunas inserciones, algunos borrados y algunas modificaciones de departamentos pares mediante la vista DEPT\_PARELL.

---

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

---

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

---

```
INSERT INTO DEPT_PARELL VALUES ( 55 , 'ALMACÉN' , 'LLEIDA' );
```

---

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 DEPT por medio de la vista DEPT\_PARELL 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 DEPT_PARELL  
AS SELECT * FROM DEPT WHERE MOD ( dept_no , 2 ) = 0 WITH CHECK OPTION ;  
  
INSERT INTO DEPT_PARELL VALUES ( 65 , 'MAGATZEM2' , 'GIRONA' );
```

---

La instrucción siguiente 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 hará el cambio de código querido:

---

```
CREATE OR REPLACE VIEW DEPT_PARELL  
AS SELECT * FROM DEPT WHERE MOD ( dept_no , 2 ) = 0 ;  
  
UPDATE DEPT_PARELL SET dept_no = dept_no + 1 WHERE dept_no = 50 ;
```

---

---

```
delete from DEPT_PARELL where dept_no IN (50, 55);
```

---

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

---

## 2.9. 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 < nom_actual > TO < nou_nom > [, < nom_actual2 > TO < nou_nom2 >, ....];
```

---

## 2.10. 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).

## 2.11. 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 ] nom_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 mapada la base de datos o el conjunto de caracteres:

---

```
ALTER { DATABASE | SCHEMA } nom_bd  
{ UPGRADE FECHA DIRECTORY NAME  
| [ DEFAULT ] CHARACTER SET [=] nom_charset  
| [ DEFAULT ] COLLATE [=] nom_collation };
```

---

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

---

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

---

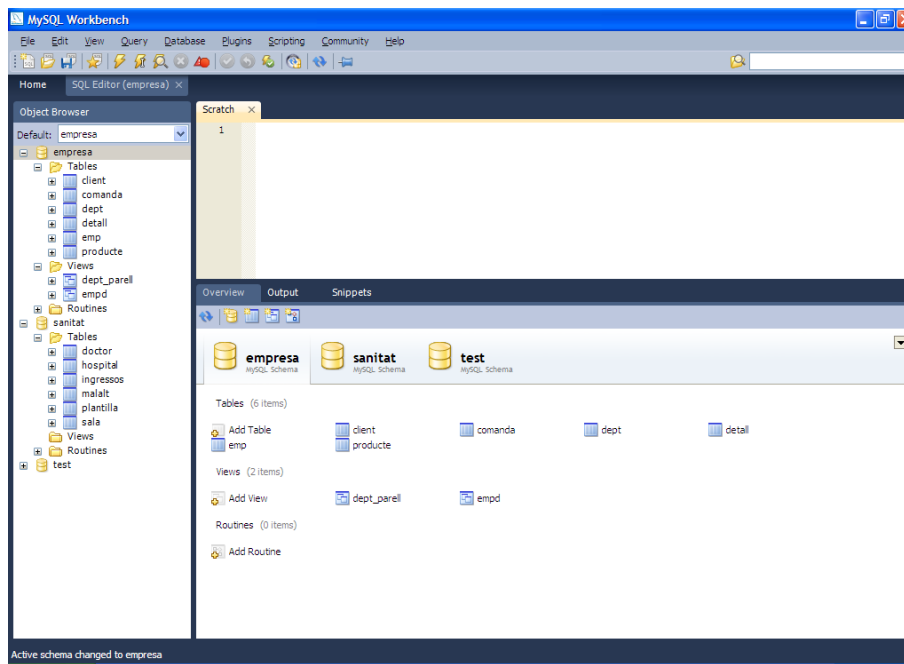
## 2.12. Como se pueden conocer los objetos definidos en un esquema de MySQL

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** 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.

Se puede ver en la [figura 2](#) la herramienta MySQL Workbench en el apartado SQL Development cómo se puede navegar por los diferentes objetos de las bases de datos que gestiona MySQL.

**Figura 2.2.** MySQL Workbench: herramienta gráfica de MySQL.  
Navegación por los diferentes objetos de la base de datos



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 [tabla 2](#) . Todas 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 2.2.** Vistas del 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.	<code>select * from information_schema.schemata;</code>
information_schema.tables	Información sobre las tablas de las diferentes bases de datos de MySQL.	<code>select * from information_schema.TABLES where table_schema = 'sanidad';</code>
information_schema.columns	Información sobre columnas de las tablas de las diferentes bases de datos de MySQL.	<code>SELECT column_name, data_type, IS_NULLABLE, COLUMN_DEFAULT FROM INFORMATION_SCHEMA.COLUMNS WHERE table_name = 'doctor' AND table_schema = 'sanidad';</code>
information_schema.table_constraints	Información sobre las restricciones	<code>SELECT * from information_schema.TABLE_CONSTRAINTS where table_schema = 'sanidad';</code>

	de las tablas de la base de datos.	
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 foráneas 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 nom_base_datos ]
```

---