Bases de Datos

Prácticas: MySQL

Innovación en Formación Profesional

De:

Planeta Formación y Universidades

Olga M. Moreno Martín

Índice

1. Introducción
   1. Arquitectura
   2. Instalación
   3. Intérprete
2. Instrucciones básicas
3. Tipos de campos
   1. Textos
   2. Numéricos
   3. Fecha y hora
4. Operadores
   1. Relacionales
   2. Lógicos
   3. Búsqueda de patrones
5. Valores
   1. Null
   2. Default
   3. Invalid
6. Tipos de datos para englobe
   1. Enum
   2. Set
7. Instrucciones de modificación
   1. Update
   2. Truncate
   3. Alter
   4. Replace
8. Funciones y cálculos
   1. Columnas calculadas
   2. Manejo de cadenas
   3. Cálculos matemáticos
   4. Fecha y hora
   5. Alias y Limit
9. Indexación
   1. Primary key
   2. Index
   3. Unique
10. Combinación de tablas
    1. Join
    2. Left join & Right join
    3. Cross join
    4. Natural join
    5. Inner join & straight join
    6. Funciones de agrupación con join
    7. If & case con join
    8. Join con más de 2 tablas
11. Subconsultas
12. Vistas
13. Gestión de transacciones
    1. Commit, rollback, savepoint
14. Usuarios, Roles y Privilegios
    1. Grant & Revoke
    2. Roles

Introducción a MySQL

MySQL es un sistema de gestión de bases de datos relacionales.

Es un DBMS de código abierto, lo que significa que cualquiera puede utilizar y modificar el software.

Es un sistema cliente / servidor que consta de un servidor SQL multiproceso, que funciona con diferentes programas cliente y bibliotecas, herramientas administrativas y una amplia gama de interfaces de programación de aplicaciones (API).

Como cualquier otro servicio informático, requiere una óptima administración para su funcionamiento, lo que requiere tareas de instalación, configuración, monitorización y optimización.

El proceso de instalación y configuración de un DBMS como MySQL. Es una de las tareas más importantes de un administrador. Si bien la instalación es un proceso relativamente sencillo, es la configuración posterior la que conlleva la mayor dificultad, siendo además un proceso constante altamente dependiente de las circunstancias específicas de nuestras aplicaciones.

Arquitectura de MySQL

- Conectores: son clientes que permiten el acceso al servidor mediante las distintas API que proporciona cada lenguaje de programación
- Utilidades: programas auxiliares que facilitan el trabajo con el servidor proporcionando distntas funcionalides como copias de seguridad
- Gestor recursos: se encarga de asignar recursos a los diferentes hilos (conexiones) establecidos con el servidor
- Interfaz SQL: lenguajes de defínción, manipulación y control de datos
- Parser: preprocesa consultas
- Optimizador: realiza las consultas de manera que consuman los mínimos recursos.
- Cachés/Búfferes: partes de la memoria reservadas para cachear y/o realizar consultas.
- Sistemas de ficheros: almacenamiento físico de los archivos ustilizados en para los distintos objetos de las bases de datos.

Arquitectura de MySQL

MySQL: create database

SQL, Structure Query Language (Lenguaje de Consulta Estructurado) es un lenguaje de programación para trabajar con base de datos relacionales como MySQL, Oracle, etc.

MySQL es un interpretador de SQL, es un servidor de base de datos.

MySQL permite crear base de datos y tablas, insertar datos, modificarlos, eliminarlos, ordenarlos, hacer consultas y realizar muchas operaciones, etc., resumiendo: administrar bases de datos.

Ingresando instrucciones en la línea de comandos o embebidas en un lenguaje como PHP nos comunicamos con el servidor. Cada sentencia debe acabar con punto y coma (;).

La sensibilidad a mayúsculas y minúsculas, es decir, si hace diferencia entre ellas, depende del SO, Windows no es sensible, pero Linux sí. Ejemplo: Windows interpreta igualmente las siguientes sentencias:

create database administracion;

Create DataBase administracion; Pero Linux interpretará como un error la segunda.

Se recomienda usar siempre minúsculas.

MySQL: show databases

Una base de datos es un conjunto de tablas.

Una base de datos tiene un nombre con el cual accederemos a ella.

Vamos a trabajar en una base de datos, llamada "administracion".

Para que el servidor nos muestre las bases de datos existentes, se lo solicitamos enviando la instrucción:

show databases;

Nos mostrará los nombres de las bases de datos, debe aparecer en este sitio "administracion".

Para usar la base de datos “administración”, usaremos:

use administracion;

MySQL: create table

Una base de datos almacena sus datos en tablas.

Una tabla es una estructura de datos que organiza los datos en columnas y filas; cada columna es un campo (o atributo) y cada fila, un registro. La intersección de una columna con una fila, contiene un dato específico, un solo valor.

Cada registro contiene un dato por cada columna de la tabla.

Cada campo (columna) debe tener un nombre. El nombre del campo hace referencia a la información que almacenará.

Cada campo (columna) también debe definir el tipo de dato que almacenará.

Nombre	**clave**
MarioPerez	Marito
MariaGarcia	Mary
DiegoRodriguez	z8080

MySQL: create table

Al crear una tabla debemos resolver qué campos (columnas) tendrá y que tipo de datos almacenarán cada uno de ellos, es decir, su estructura.

La tabla debe ser definida con un nombre que la identifique y con el cual accederemos a ella.

Creamos una tabla llamada "usuarios", tipeamos:

```
create table usuarios (
nombre varchar(30),
clave varchar(10)
);
```

Si intentamos crear una tabla con un nombre ya existente (existe otra tabla con ese nombre), mostrará un mensaje de error indicando que la acción no se realizó porque ya existe una tabla con el mismo nombre.

MySQL: describe

Cuando se crea una tabla debemos indicar su nombre y definir sus campos con su tipo de dato. En esta tabla "usuarios" definimos 2 campos:

- nombre: que contendrá una cadena de hasta 30 caracteres, que almacena el nombre de usuario

- clave: otra cadena de caracteres de 10 de longitud, que guardará la clave de cada usuario

Cada usuario ocupará un registro de esta tabla, con su respectivo nombre y clave.

Para ver la estructura de una tabla usamos el comando "describe" junto al nombre de la tabla:

describe usuarios;

Aparece lo siguiente:

Field Type Null

nombre varchar(30) YES

clave varchar(10) YES

Esta es la estructura de la tabla "usuarios"; nos muestra cada campo, su tipo, lo que ocupa en bytes y otros datos como la aceptación de valores nulos etc., que veremos más adelante en detalle.

MySQL: drop table

Para eliminar una tabla usamos "drop table". Tipeamos:

drop table usuarios;

Si tipeamos nuevamente:

drop table usuarios;

Aparece un mensaje de error, indicando que no existe, ya que intentamos borrar una tabla inexistente.

Para evitar este mensaje podemos tipear:

drop table if exists usuarios;

En la sentencia precedente especificamos que elimine la tabla "usuarios" si existe.

MySQL: insert into

Un registro es una fila de la tabla que contiene los datos propiamente dichos. Cada registro tiene un dato por cada columna.

Recordemos como crear la tabla "usuarios":

```
create table usuarios (
nombre varchar(30),
clave varchar(10)
);
```

Al ingresar los datos de cada registro debe tenerse en cuenta la cantidad y el orden de los campos. Ahora vamos a agregar un registro a la tabla:

```
insert into usuarios (nombre, clave) values ('MarioPerez','Marito');
```

Usamos "insert into". Especificamos los nombres de los campos entre paréntesis y separados por comas y luego los valores para cada campo, también entre paréntesis y separados por comas.

MySQL: insert into

La tabla usuarios ahora la podemos graficar de la siguiente forma:

nombre	**clave**
MarioPerez	Marito

Es importante ingresar los valores en el mismo orden en que se nombran los campos, si ingresamos los datos en otro orden, no aparece un mensaje de error y los datos se guardan de modo incorrecto.

Notad que los datos ingresados, como corresponden a campos de cadenas de caracteres se colocan entre comillas simples. Las comillas simples son OBLIGATORIAS.

MySQL : select

Para ver los registros de una tabla usamos "select":

select nombre,clave from usuarios;

Aparece un registro.

El comando "select" recupera los registros de una tabla. Luego del comando select indicamos los nombres de los campos a rescatar.

¿Cómo ver todos los registros de una tabla?

select * from usuarios;

El comando "select" recupera los registros de una tabla. Con el asterisco (*) indicamos que seleccione todos los campos de la tabla que nombramos.

Podemos especificar el nombre de los campos que queremos ver separándolos por comas.

MySQL: select - where

Recuperación de registros específicos

Hemos aprendido cómo ver todos los registros de una tabla:

select nombre, clave from usuarios;

El comando "select" recupera los registros de una tabla. Detallando los nombres de los campos separados por comas, indicamos que seleccione todos los campos de la tabla que nombramos.

Existe una cláusula, "where" que es opcional, con ella podemos especificar condiciones para la consulta "select". Es decir, podemos recuperar algunos registros, sólo los que cumplan con ciertas condiciones indicadas con la cláusula "where". Por ejemplo, queremos ver el usuario cuyo nombre es "MarioPerez", para ello utilizamos "where" y luego de ella, la condición:

select nombre, clave from usuarios where nombre='MarioPerez';

Para las condiciones se utilizan operadores relacionales (tema que trataremos más adelante en detalle).

El signo igual(=) es un operador relacional. Para la siguiente selección de registros especificamos una condición que solicita los usuarios cuya clave es igual a 'bocajunior':

select nombre, clave from usuarios where clave='bocajunior';

Si ningún registro cumple la condición establecida con el "where", no aparecerá ningún registro.

MySQL: delete from table

Borrado de registros de una tabla (delete)

Para eliminar los registros de una tabla usamos el comando "delete":

delete from usuarios;

La ejecución del comando indicado en la línea anterior borra TODOS los registros de la tabla.

Si queremos eliminar uno o varios registros debemos indicar cuál o cuáles, para ello utilizamos el comando "delete" junto con la clausula "where" con la cual establecemos la condición que deben cumplir los registros a borrar. Por ejemplo, queremos eliminar aquel registro cuyo nombre de usuario es 'Leonardo':

delete from usuarios where nombre='Leonardo';

Si solicitamos el borrado de un registro que no existe, es decir, ningún registro cumple con la condición especificada, no se borrarán registros, pues no encontró registros con ese dato.

Tipos de los campos

Al crear una tabla debemos elegir la estructura adecuada, esto es, definir los campos y sus tipos más precisos, según el caso. Por ejemplo, si un campo numérico almacenará solamente valores enteros positivos el tipo "integer" con el atributo "unsigned" es más adecuado que, por ejemplo un "float".

Los valores que podemos guardar son:

A) TEXTO: Para almacenar texto usamos cadenas de caracteres. Las cadenas se colocan entre comillas simples. Podemos almacenar dígitos con los que no se realizan operaciones matemáticas, por ejemplo, códigos de identificación, números de documentos, números telefónicos. Tenemos los siguientes tipos: varchar, char y text.

B) NÚMEROS: Existe variedad de tipos numéricos para representar enteros, negativos, decimales. Para almacenar valores enteros, por ejemplo, en campos que hacen referencia a cantidades, precios, etc., usamos el tipo integer. Para almacenar valores con decimales utilizamos: float o decimal.

C) FECHAS Y HORAS: para guardar fechas y horas dispone de varios tipos: date (fecha), datetime (fecha y hora), time (hora), year (año) y timestamp (XX-XX-XX, 00:00:00).

D) OTROS TIPOS: enum y set representan una enumeración y un conjunto respectivamente.

Lo veremos más adelante.

E) El valor "null". El valor 'null' significa ? valor desconocido. O "dato inexistente".

No es lo mismo que 0 o una cadena vacía.

Tipos de los campos

Tipos de datos (texto)

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The size parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large OBjects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Tipos de los campos

Tipos de datos (texto)

Para almacenar TEXTO usamos cadenas de caracteres. Las cadenas se colocan entre comillas simples.

Podemos almacenar dígitos con los que no se realizan operaciones matemáticas, por ejemplo, códigos de identificación, números de documentos, números telefónicos.

1) varchar(x): define una cadena de caracteres de longitud variable en la cual determinamos el máximo de caracteres con el argumento "x" que va entre paréntesis. Su rango va de 1 a 255 caracteres. Un varchar(10) ocupa 11 bytes, pues en uno de ellos almacena la longitud de la cadena. Ocupa un byte más que la cantidad definida.

Tipos de los campos

Tipos de datos (texto)

2) char(x): define una cadena de longitud fija, su rango es de 1 a 255 caracteres. Si la cadena ingresada es menor a la longitud definida (por ejemplo cargamos 'Juan' en un char(10)), almacena espacios en blanco a la derecha, tales espacios se eliminan al recuperarse el dato. Un char(10) ocupa 10 bytes, pues al ser fija su longitud, no necesita ese byte adicional donde guardar la longitud. Por ello, si la longitud es invariable, es conveniente utilizar el tipo char; caso contrario, el tipo varchar.

Ocupa tantos bytes como se definen con el argumento "x". Si ingresa un argumento mayor al permitido (255) aparece un mensaje indicando que no se permite y sugiriendo que use "blob" o "text". Si omite el argumento, coloca 1 por defecto.

Tipos de los campos

Tipos de datos (texto)

3) blob o text: bloques de datos de 60000 caracteres de longitud aprox.

Para los tipos que almacenan cadenas, si asignamos una cadena de caracteres de mayor longitud que la permitida o definida, la cadena se corta. Por ejemplo, si definimos un campo de tipo varchar(10) y le asignamos la cadena 'Buenas tardes', se almacenará 'Buenas tar' ajustándose a la longitud de 10.

Es importante elegir el tipo de dato adecuado según el caso, el más preciso. Por ejemplo, si vamos a almacenar un caracter, conviene usar char(1), que ocupa 1 byte y no varchar(1), que ocupa 2 bytes.

Tipos de los campos

Tipos de datos (numéricos)

Data type	Description
BIT(size)	A bit-value type. The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1.
TINYINT(size)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(size)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255)
MEDIUMINT(size)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter specifies the maximum display width (which is 255)
INT(size)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)
INTEGER(size)	Equal to INT(size)
BIGINT(size)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)
FLOAT(size, d)	A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(p)	A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(size, d)	A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter
DOUBLE PRECISION(size, d)	
DECIMAL(size, d)	An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0.
DEC(size, d)	Equal to DECIMAL(size,d)

Tipos de los campos

Tipos de datos (numéricos)

Existe variedad de tipos numéricos para representar enteros, negativos, decimales. Para almacenar valores enteros, por ejemplo, en campos que hacen referencia a cantidades, precios, etc., usamos:

1) integer(x) o int(x): su rango es de -2000000000 a 2000000000 aprox. El tipo "int unsigned" va de 0 a 4000000000. El tipo "integer" tiene subtipos:

- mediumint(x): va de -8000000 a 8000000 aprox. Sin signo va de 0 a 16000000 aprox.
- smallint(x): va de -30000 a 30000 aprox., sin signo, de 0 a 60000 aprox.
- tinyint(x): define un valor entero pequeño, cuyo rango es de -128 a 127. El tipo sin signo va de 0 a 255.
- bool o boolean: sinónimos de tinyint(1). Un valor cero se considera falso, los valores distintos de cero, verdadero.
- bigint(x): es un entero largo. Va de -9000000000000000000 a 9000000000000000000 aprox. Sin signo es de 0 a 10000000000000000000.

Tipos de los campos

Tipos de datos (numéricos)

Para almacenar valores con decimales utilizamos:

2) float (t,d): número de coma flotante. Su rango es de -3.4e^+38 a -1.1e^-38 (9 cifras).

3) decimal o numeric (t,d): el primer argumento indica el total de dígitos y el segundo, la cantidad de decimales. El rango depende de los argumentos, también los bytes que ocupa. Si queremos almacenar valores entre 0.00 y 99.99 debemos definir el campo como tipo "decimal (4,2)". Si no se indica el valor del segundo argumento, por defecto es 0. *Para los tipos "float" y "decimal" se utiliza* ***el punto*** *como separador de decimales.*

Todos los tipos enteros pueden tener el atributo **"unsigned",** esto permite sólo valores positivos y duplica el rango. Los tipos de coma flotante también aceptan el atributo "unsigned", pero el valor del límite superior del rango no se modifica.

Es importante elegir el tipo de dato adecuado según el caso, el más preciso. Por ejemplo, si un campo numérico almacenará valores positivos menores a 10000, el tipo "int" no es el más adecuado, porque su rango va de -2000000000 a 2000000000 aprox., conviene el tipo "smallint unsigned", cuyo rango va de 0 a 60000 aprox. De esta manera usamos el menor espacio de almacenamiento posible.

Tipos de los campos

Tipos de datos (fechas y horas)

MySQL viene con los siguientes tipos de datos para almacenar una fecha o un valor de fecha / hora en la base de datos:

- A) DATE- formato AAAA-MM-DD
- B) DATETIME - formato: AAAA-MM-DD HH: MI: SS
- C) TIMESTAMP - formato: AAAA-MM-DD HH: MI: SS
- D) YEAR - formato YYYY o YY
- E) TIME - formato HH: MI: SS

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(fsp)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(fsp)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(fsp)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

Campo entero con autoincremento

Un campo de tipo entero puede tener otro atributo extra 'auto_increment'. Los valores de un campo 'auto_increment', se inician en 1 y se incrementan en 1 automáticamente.

Se utiliza generalmente en campos correspondientes a códigos de identificación para generar valores únicos para cada nuevo registro que se inserta.

Sólo puede haber un campo "auto_increment" y debe ser clave primaria (o estar indexado).

Para establecer que un campo autoincremente sus valores automáticamente, éste debe ser entero (integer) y debe ser clave primaria:

```
create table libros(
codigo int unsigned auto_increment  not null unique,
titulo varchar(20),
autor varchar(30),
editorial varchar(15),
primary key (codigo)
);
```

Para definir un campo autoincrementable colocamos "auto_increment" luego de la definición del campo al crear la tabla.

Campo entero con autoincremento

Hasta ahora, al ingresar registros, colocamos el nombre de todos los campos antes de los valores; es posible ingresar valores para algunos de los campos de la tabla, pero recuerde que al ingresar los valores debemos tener en cuenta los campos que detallamos y el orden en que lo hacemos.

Cuando un campo tiene el atributo "auto_increment" no es necesario ingresar valor para él, porque se inserta automáticamente tomando el último valor como referencia, o 1 si es el primero.

Para ingresar registros omitimos el campo definido como "auto_increment", por ejemplo:

```
insert into libros (titulo,autor,editorial) values('El aleph','Borges','Planeta');
```

Este primer registro ingresado guardará el valor 1 en el campo correspondiente al código.

Si continuamos ingresando registros, el código (dato que no ingresamos) se cargará automáticamente siguiendo la secuencia de autoincremento.

Un campo "auto_increment" funciona correctamente sólo cuando contiene únicamente valores positivos. Más adelante explicaremos cómo definir un campo con sólo valores positivos.

Está permitido ingresar el valor correspondiente al campo "auto_increment", por ejemplo:

```
insert into libros (codigo,titulo,autor,editorial)
values(6,'Martin Fierro','Jose Hernandez','Paidos');
```

Campo entero con autoincremento

Pero debemos tener cuidado con la inserción de un dato en campos "auto_increment". Debemos tener en cuenta que:

- si el valor está repetido aparecerá un mensaje de error y el registro no se ingresará.
- si el valor dado saltea la secuencia, lo toma igualmente y en las siguientes inserciones, continuará la secuencia tomando el valor más alto.
- si el valor ingresado es 0, no lo toma y guarda el registro continuando la secuencia.
- si el valor ingresado es negativo (y el campo no está definido para aceptar sólo valores positivos), lo ingresa.

Para que este atributo funcione correctamente, el campo debe contener solamente valores positivos; más adelante trataremos este tema.

Clave primaria

Una clave primaria es un campo (o varios) que identifica 1 solo registro (fila) en una tabla. Para un valor del campo clave existe solamente 1 registro. Los valores no se repiten ni pueden ser nulos.

Veamos un ejemplo, si tenemos una tabla con datos de personas, el número de documento puede establecerse como clave primaria, es un valor que no se repite; puede haber personas con igual apellido y nombre, incluso el mismo domicilio (padre e hijo por ejemplo), pero su documento será siempre distinto.

Si tenemos la tabla "usuarios", el nombre de cada usuario puede establecerse como clave primaria, es un valor que no se repite; puede haber usuarios con igual clave, pero su nombre de usuario será siempre distinto.

Establecemos que un campo sea clave primaria al momento de creación de la tabla:

```
create table usuarios (
nombre varchar(20),
clave varchar(10),
primary key(nombre)
);
```

Clave primaria

Para definir un campo como clave primaria agregamos "primary key" luego de la definición de todos los campos y entre paréntesis colocamos el nombre del campo que queremos como clave.

Si visualizamos la estructura de la tabla con "describe" vemos que el campo "nombre" es clave primaria y no acepta valores nulos (más adelante explicaremos esto detalladamente).

Ingresamos algunos registros:

```
insert into usuarios (nombre, clave) values ('Leonardo','payaso');
insert into usuarios (nombre, clave) values ('MarioPerez','Marito');
insert into usuarios (nombre, clave) values ('Marcelo','River');
insert into usuarios (nombre, clave) values ('Gustavo','River');
```

Si intentamos ingresar un valor para el campo clave que ya existe, aparece un mensaje de error indicando que el registro no se cargó pues el dato clave existe. Esto sucede porque los campos definidos como clave primaria no pueden repetirse.

Ingresamos un registro con un nombre de usuario repetido, por ejemplo:

```
insert into usuarios (nombre, clave) values ('Gustavo','Boca');
```

Una tabla sólo puede tener una clave primaria. Cualquier campo (de cualquier tipo) puede ser clave primaria, debe cumplir como requisito, que sus valores no se repitan.

Clave primaria compuesta

Las claves primarias pueden ser simples, formadas por un solo campo o compuestas, más de un campo.

Recordemos que una clave primaria identifica 1 solo registro en una tabla. Para un valor del campo clave existe solamente 1 registro. Los valores no se repiten ni pueden ser nulos.

El ejemplo de la playa de estacionamiento que almacena cada día los datos de los vehículos que ingresan en la tabla llamada "vehiculos" con los siguientes campos:

- patente char(6) not null,
- tipo char (4),
- horallegada time not null,
- horasalida time

Necesitamos definir una clave primaria para una tabla con los datos descriptos arriba. No podemos usar la patente porque un mismo auto puede ingresar más de una vez en el día a la playa; tampoco podemos usar la hora de entrada porque varios autos pueden ingresar a una misma hora. Tampoco sirven los otros campos.

Como ningún campo, por si solo cumple con la condición para ser clave, es decir, debe identificar un solo registro, el valor no puede repetirse, debemos usar 2 campos.

Definimos una clave compuesta cuando ningún campo por si solo cumple con la condición para ser clave.

Clave primaria compuesta

En este ejemplo, un auto puede ingresar varias veces en un día a la playa, pero siempre será a distinta hora. Usamos 2 campos como clave, la patente junto con la hora de llegada, así identificamos unívocamente cada registro.

Para establecer más de un campo como clave primaria usamos la siguiente sintaxis:

```
create table vehiculos(
patente char(6) not null,
tipo char(4),
horallegada time not null
horasalida time,
primary key(patente,horallegada) );
```

Nombramos los campos que formarán parte de la clave separados por comas.

Si vemos la estructura de la tabla con "describe" vemos que en la columna "key", en ambos campos aparece "PRI", porque ambos son clave primaria.

Un campo que es parte de una clave primaria puede ser auto_increment sólo si es el primer campo que compone la clave, si es secundario no se permite. Es posible eliminar un campo que es parte de una clave primaria, la clave queda con los campos restantes. Esto, siempre que no queden registros con clave repetida.

Atributo default en una columna de una tabla

Si al insertar registros no se especifica un valor para un campo, se inserta su valor por defecto implícito según el tipo de dato del campo. Por ejemplo:

```
insert into libros (titulo,autor,editorial,precio,cantidad)
values('Java en 10 minutos','Juan Pereyra','Paidos',25.7,100);
```

Como no ingresamos valor para el campo "codigo", MySQL insertará el valor por defecto, como "codigo" es un campo "auto_increment", el valor por defecto es el siguiente de la secuencia.

Si omitimos el valor correspondiente al autor:

```
insert into libros (titulo,editorial,precio,cantidad) values('Java en 10 minutos','Paidos',25.7,200);
```

MySQL insertará "null", porque el valor por defecto de un campo (de cualquier tipo) que acepta valores nulos, es "null".

Lo mismo sucede si no ingresamos el valor del precio:

```
insert into libros (titulo,autor,editorial,cantidad)
values('Java en 10 minutos','Juan Pereyra',Paidos',150);
```

MySQL insertará el valor "null" porque el valor por defecto de un campo (de cualquier tipo) que acepta valores nulos, es "null".

Atributo default en una columna de una tabla

Si omitimos el valor correspondiente al título:

```
insert into libros (autor,editorial,precio,cantidad) values ('Borges','Paidos',25.7,200);
```

MySQL guardará una cadena vacía, ya que éste es el valor por defecto de un campo de tipo cadena definido como "not null" (no acepta valores nulos). Si omitimos el valor correspondiente a la cantidad:

```
insert into libros (titulo,autor,editorial,precio)
values('Alicia a través del espejo', 'Lewis Carroll','Emece',34.5);
```

El valor que se almacenará será 0, porque el campo "precio" es de tipo numérico "not null" y el valor por defecto de los tipos numéricos que no aceptan valores nulos es 0. Podemos establecer valores por defecto para los campos cuando creamos la tabla. Para ello utilizamos "default" al definir el campo. Por ejemplo, queremos que el valor por defecto del campo "precio" sea 1.11 y el valor por defecto del campo "autor" sea "Desconocido":

```
create table libros(
codigo int unsigned auto_increment,
titulo varchar(40) not null,
autor varchar(30) default ‘Anónimo',
precio decimal(5,2) unsigned default 1.11,
cantidad int unsigned not null default 20,
primary key (codigo) );
```

Atributo default en una columna de una tabla

Si al ingresar un nuevo registro omitimos los valores para el campo "autor" y "precio", MySQL insertará los valores por defecto definidos con la palabra clave "default":

```
insert into libros (titulo,editorial,cantidad) values('Java en 10 minutos','Paidos',200);
```

MySQL insertará el registro con el siguiente valor de la secuencia en "codigo", con el título, editorial y cantidad ingresados, en "autor" colocará "Desconocido" y en precio "1.11".

Entonces, si al definir el campo explicitamos un valor mediante la cláusula "default", ése será el valor por defecto; sino insertará el valor por defecto implícito según el tipo de dato del campo.

Los campos definidos "auto_increment" no pueden explicitar un valor con "default", tampoco los de tipo "blob" y "text".

Los valores por defecto implícitos son los siguientes:

- para campos de cualquier tipo que admiten valores nulos, el valor por defecto "null";
- para campos que no admiten valores nulos, es decir, definidos "not null", el valor por defecto depende del tipo de dato:
  - para campos numéricos no declarados "auto_increment": 0;
  - para campos numéricos definidos "auto_increment": el valor siguiente de la secuencia, comenzando en 1;
  - para los tipos cadena: cadena vacía.
  - fecha 0000-00-00 00:00:00

Atributo default en una columna de una tabla

Ahora al visualizar la estructura de la tabla con "describe" podemos entender un poco más lo que informa cada columna:

describe libros;

"Field" contiene el nombre del campo; "Type", el tipo de dato; "NULL" indica si el campo admite valores nulos; "Key" indica si el campo está indexado (lo veremos más adelante); "Default" muestra el valor por defecto del campo y "Extra" muestra información adicional respecto al campo, por ejemplo, aquí indica que "codigo" está definido "auto_increment".

También se puede utilizar "default" para dar el valor por defecto a los campos en sentencias "insert", por ejemplo:

insert into libros (titulo,autor,precio,cantidad) values ('El gato con botas',default,default,100);

Atributo zerofill en una columna de una tabla

Cualquier campo numérico puede tener otro atributo extra "zerofill". "zerofill" rellena con ceros los espacios disponibles a la izquierda. Por ejemplo, creamos la tabla "libros", definiendo los campos "codigo" y "cantidad" con el atributo "zerofill":

```
create table libros(
codigo int(6) zerofill auto_increment,
titulo varchar(40) not null,
autor varchar(30),
editorial varchar(15),
precio decimal(5,2) unsigned,
cantidad smallint zerofill,
primary key (codigo) );
```

Note que especificamos el tamaño del tipo "int" entre paréntesis para que muestre por la izquierda ceros, cuando los valores son inferiores al indicado; dicho parámetro no restringe el rango de valores que se pueden almacenar ni el número de dígitos.

Al ingresar un valor de código con menos cifras que las especificadas (6), aparecerán ceros a la izquierda rellenando los espacios; por ejemplo, si ingresamos "33", aparecerá "000033". Al ingresar un valor para el campo "cantidad", sucederá lo mismo.

Si especificamos "zerofill" a un campo numérico, se coloca automáticamente el atributo "unsigned". Cualquier valor negativo ingresado en un campo definido "zerofill" es un valor inválido.

Claves foráneas

En MySQL sólo existe soporte para claves foráneas en tablas de tipo InnoDB. Sin embargo, esto no impide usarlas en otros tipos de tablas.

La diferencia consiste en que en esas tablas no se verifica si una clave foránea existe realmente en la tabla referenciada, y que no se eliminan filas de una tabla con una definición de clave foránea. Para hacer esto hay que usar tablas InnoDB. Hay dos modos de definir claves foráneas en bases de datos MySQL.

El primero, sólo sirve para documentar, y, no define realmente claves foráneas. Esta forma consiste en definir una referencia al mismo tiempo que se define una columna:

```
create table personas (
id int unsigned auto_increment primary key,
nombre varchar(40),
fecha date
primary key(id));
create table telefonos (
numero char(12),
id int not null references personas (id)
);
```

Claves foráneas

Hemos usado una definición de referencia para la columna 'id' de la tabla 'telefonos', indicando que es una clave foránea correspondiente a la columna 'id' de la tabla 'personas'. Sin embargo, aunque la sintaxis se comprueba, esta definición no implica ningún comportamiento por parte de MySQL. La otra forma es mucho más útil, aunque sólo se aplica a tablas InnoDB.

```
create table personas2 (
id int unsigned auto_increment primary key,
nombre varchar(40),
fecha date
primary key(id));
engine=InnoDB;
create table telefonos2 (
numero CHAR(12),
id int not null ,
key(id),
foreign key (id) refrences personas2 (id)
) ;
ENGINE=InnoDB;
```

Claves foráneas

Es imprescindible que la columna que contiene una definición de clave foránea esté indexada. Pero esto no debe preocuparnos demasiado, ya que si no lo hacemos de forma explícita, MySQL lo hará por nosotros de forma implícita.

Esta forma define una clave foránea en la columna 'id', que hace referencia a la columna 'id' de la tabla 'personas'. La definición incluye las tareas a realizar en el caso de que se elimine una fila en la tabla 'personas'.

- ON DELETE <opción>, indica que acciones se deben realizar en la tabla actual si se borra una fila en la tabla referenciada.
- ON UPDATE <opción>, es análogo pero para modificaciones de claves.

Existen cinco opciones diferentes. Veamos lo que hace cada una de ellas:

- RESTRICT: esta opción impide eliminar o modificar filas en la tabla referenciada si existen filas con el mismo valor de clave foránea.
- CASCADE: borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica borrar las filas con el mismo valor de clave foránea o modificar los valores de esas claves foráneas.
- SET NULL: borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica asignar el valor NULL a las claves foráneas con el mismo valor.
- NO ACTION: las claves foráneas no se modifican, ni se eliminan filas en la tabla que las contiene.
- SET DEFAULT: borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado implica asignar el valor por defecto a las claves foráneas con el mismo valor.

Claves foráneas

Para crear una FOREIGN KEY en la columna “idPersona" cuando la tabla “pedidos" ya está creada, usamos la siguiente sintaxis SQL:

```
ALTER TABLE pedidos
ADD FOREIGN KEY (idPersona) REFERENCES personas(id);
```

Operadores Relacionales = <> < <= > >=

Hemos aprendido a especificar condiciones de igualdad para seleccionar registros de una tabla; por ejemplo:

select titulo,autor,editorial from libros where autor='Borges';

Utilizamos el operador relacional de igualdad.

Los operadores relacionales vinculan un campo con un valor para que MySQL compare cada registro (el campo especificado) con el valor dado.

Los operadores relacionales son los siguientes:

= igual <> distinto > mayor < menor

>= mayor o igual <= menor o igual

Podemos seleccionar los registros cuyo autor sea diferente de 'Borges', para ello usamos la condición:

select titulo,autor,editorial from libros where autor<>'Borges';

Podemos comparar valores numéricos. Por ejemplo, queremos mostrar los libros cuyos precios sean mayores a 20 pesos:

select titulo,autor,editorial,precio from libros where precio>20;

También, los libros cuyo precio sea menor o igual a 30:

select titulo,autor,editorial,precio from libros where precio<=30;

Operadores Lógicos (and - or - not)

Hasta el momento, hemos aprendido a establecer una condición con "where" utilizando operadores relacionales. Podemos establecer más de una condición con la cláusula "where", para ello aprenderemos los operadores lógicos. Son los siguientes:

- and, significa "y",
- or, significa "y/o",
- xor, significa "o",
- not, significa "no", invierte el resultado
- (), paréntesis

Los operadores lógicos se usan para combinar condiciones.

Queremos recuperar todos los registros cuyo autor sea igual a "Borges" y cuyo precio no supere los 20 €, para ello necesitamos 2 condiciones:

```
select * from libros where (autor='Borges') and (precio<=20);
```

Los registros recuperados en una sentencia que une 2 condiciones con el operador "and", cumplen con las 2 condiciones.

Operadores Lógicos (and - or - not)

Queremos ver los libros cuyo autor sea "Borges" y/o cuya editorial sea "Planeta":

```
select * from libros where (autor='Borges') or (editorial='Planeta');
```

En la sentencia anterior usamos el operador "or", indicamos que recupere los libros en los cuales el valor del campo "autor" sea "Borges" y/o el valor del campo "editorial" sea "Planeta", es decir, seleccionará los registros que cumplan con la primera condición, con la segunda condición o con ambas condiciones.

Los registros recuperados con una sentencia que une 2 condiciones con el operador "or", cumplen 1 de las condiciones o ambas.

Queremos ver los libros cuyo autor sea "Borges" o cuya editorial sea "Planeta":

```
select * from libros where (autor='Borges') xor (editorial='Planeta');
```

En la sentencia anterior usamos el operador "xor", indicamos que recupere los libros en los cuales el valor del campo "autor" sea "Borges" o el valor del campo "editorial" sea "Planeta", es decir, seleccionará los registros que cumplan con la primera condición o con la segunda condición pero no los que cumplan con ambas condiciones. Los registros recuperados con una sentencia que une 2 condiciones con el operador "xor", cumplen 1 de las condiciones, no ambas.

Operadores Lógicos (and - or - not)

Queremos recuperar los libros que no cumplan la condición dada, por ejemplo, aquellos cuya editorial NO sea "Planeta":

```
select * from libros where not (editorial='Planeta');
```

El operador "not" invierte el resultado de la condición a la cual antecede.

Los registros recuperados en una sentencia en la cual aparece el operador "not", no cumplen con la condición a la cual afecta el "NO".

Los paréntesis se usan para encerrar condiciones, para que se evalúen como una sola expresión. Cuando explicitamos varias condiciones con diferentes operadores lógicos (combinamos "and", "or") permite establecer el orden de prioridad de la evaluación; además permite diferenciar las expresiones más claramente.

Por ejemplo, las siguientes expresiones devuelven un resultado diferente:

```
select * from libros where (autor='Borges') or (editorial='Paidos' and precio<20);
```

```
select * from libros where (autor='Borges' or editorial='Paidos') and (precio<20);
```

Si bien los paréntesis no son obligatorios en todos los casos, se recomienda utilizarlos para evitar confusiones.

Operadores Relacionales (between - in)

Para recuperar de nuestra tabla "libros" los registros que tienen precio mayor o igual a 20 y menor o igual a 40, usamos 2 condiciones unidas por el operador lógico "and":

```
select * from libros where precio>=20 and precio<=40;
```

Podemos usar "between":

```
select * from libros where precio between 20 and 40;
```

"between" significa "entre". Averiguamos si el valor de un campo dado (precio) está entre los valores mínimo y máximo especificados (20 y 40 respectivamente).

Para recuperar los libros cuyo autor sea 'Paenza' o 'Borges' usamos 2 condiciones:

```
select * from libros where autor='Borges' or autor='Paenza';
```

Podemos usar "in":

```
select * from libros where autor in('Borges','Paenza');
```

Con "in" averiguamos si el valor de un campo dado (autor) está incluido en la lista de valores especificada (en este caso, 2 cadenas).

Operadores de búsqueda de patrones (like y not like)

Imaginemos que tenemos registrados estos 2 libros: El Aleph de Borges; Antologia poetica de J.L. Borges; si queremos recuperar todos los libros cuyo autor sea "Borges", y especificamos la siguiente condición:

select * from libros where autor='Borges';

Sólo aparecerá el primer registro, ya que la cadena "Borges" no es igual a la cadena "J.L. Borges".

Esto sucede porque el operador "=" (igual), también el operador "<>" (distinto) comparan cadenas de caracteres completas. Para comparar porciones de cadenas utilizamos los operadores "like" y "not like".

Entonces, podemos comparar trozos de cadenas de caracteres para realizar consultas. Para recuperar todos los registros cuyo autor contenga la cadena "Borges" debemos introducir:

select * from libros where autor like "%Borges%";

El símbolo "%" (porcentaje) reemplaza cualquier cantidad de caracteres (incluyendo ningún carácter). Es un carácter comodín. "like" y "not like" son operadores de comparación que señalan igualdad o diferencia.

Operadores de búsqueda de patrones (like y not like)

Para seleccionar todos los libros que comiencen con "A":

```
select * from libros where titulo like 'A%';
```

Note que el símbolo "%" ya no está al comienzo, con esto indicamos que el título debe tener como primera letra la "A" y luego, cualquier cantidad de caracteres.

Para seleccionar todos los libros que no comiencen con "A":

```
select * from libros where titulo not like 'A%';
```

Así como "%" reemplaza cualquier cantidad de caracteres, el guion bajo "_" reemplaza un carácter, es el otro carácter comodín. Por ejemplo, queremos ver los libros de "Lewis Carroll" pero no recordamos si se escribe "Carroll" o "Carrolt", entonces usamos esta condición:

```
select * from libros where autor like "%Carrol_";
```

Si necesitamos buscar un patrón en el que aparezcan los caracteres comodines, por ejemplo, queremos ver todos los registros que comiencen con un guion bajo, si utilizamos '_%', mostrará todos los registros porque lo interpreta como "patrón que comienza con un caracter cualquiera y sigue con cualquier cantidad de caracteres". Debemos utilizar "_%", esto se interpreta como 'patrón que comienza con guion bajo y continúa con cualquier cantidad de caracteres". Es decir, si queremos incluir en una búsqueda de patrones los caracteres comodines, debemos anteponer al carácter comodín, la barra invertida "\", así lo tomará como carácter de búsqueda literal y no como comodín para la búsqueda. Para buscar el carácter literal "%" se debe colocar "\%".

Búsqueda de patrones (regexp)

Los operadores "regexp" y "not regexp" busca patrones de modo similar a "like" y "not like". Para buscar libros que contengan la cadena "Ma" usamos:

select titulo from libros where titulo regexp 'Ma';

Para buscar los autores que tienen al menos una "h" o una "k" o una "w" usamos:

select autor from libros where autor regexp '[hkw]';

Para buscar los autores que no tienen ni "h" o una "k" o una "w" introducimos:

select autor from libros where autor not regexp '[hkw]';

Para buscar autores que tienen por lo menos una de las letras de la "a" a la "d", "a,b,c,d", usamos:

select autor from libros where autor regexp '[a-d]';

Para ver los títulos que comienzan con "A" usamos:

select titulo from libros where titulo regexp '^A';

Para ver los títulos que terminan en "HP" usamos:

select titulo from libros where titulo regexp 'HP$';

Búsqueda de patrones (regexp) Expresiones regulares

Para buscar títulos que tengan una "a" luego un carácter cualquiera y luego una "e" utilizamos:

```
select titulo from libros where titulo regexp 'a.e';
```

El punto (.) identifica cualquier carácter.

Podemos mostrar los títulos que contienen una "a" seguida de 2 caracteres y luego una "e":

```
select titulo from libros where titulo regexp 'a..e';
```

Para buscar autores que tengan 6 caracteres exactamente usamos:

```
select autor from libros where autor regexp '^......$';
```

Para buscar autores que tengan al menos 6 caracteres usamos:

```
select autor from libros where autor regexp '......';
```

Para buscar títulos que contengan 2 letras "a" usamos:

```
select titulo from libros where titulo regexp 'a.*a';
```

El asterisco indica que busque el carácter inmediatamente anterior, en este caso cualquiera porque hay un punto.

Ordenación – order by

Podemos ordenar el resultado de un "select" para que los registros se muestren ordenados por algún campo, para ello usamos la cláusula "order by".Por ejemplo, recuperamos los registros de la tabla "libros" ordenados por el título:

```
select codigo,titulo,autor,editorial,precio from libros order by titulo;
```

Aparecen los registros ordenados alfabéticamente por el campo especificado. También podemos colocar el número de orden del campo por el que queremos que se ordene en lugar de su nombre. Por ejemplo, queremos el resultado del "select" ordenado por "precio":

```
select codigo,titulo,autor,editorial,precio from libros order by 5;
```

Por defecto, si no aclaramos en la sentencia, los ordena de manera ascendente (de menor a mayor). Podemos ordenarlos de mayor a menor, para ello agregamos la palabra clave "desc":

```
select codigo,titulo,autor,editorial,precio from libros order by editorial desc;
```

También podemos ordenar por varios campos, por ejemplo, por "titulo" y "editorial":

```
select codigo,titulo,autor,editorial,precio from libros order by titulo, editorial;
```

Incluso, podemos ordenar en distintos sentidos, por ejemplo, por "titulo" en sentido ascendente y "editorial" en sentido descendente:

```
select codigo,titulo,autor,editorial,precio from libros order by titulo asc, editorial desc;
```

Debe aclararse al lado de cada campo, pues estas palabras claves afectan al campo inmediatamente anterior.

Contar registros (count)

Imaginemos que nuestra tabla "libros" tiene muchos registros. Para averiguar la cantidad sin necesidad de contarlos manualmente usamos la función "count()": ***select count(*) from libros;***

La función "count()" cuenta la cantidad de registros de una tabla, incluyendo los nulos. Para saber la cantidad de libros de la editorial "Planeta" usamos:

select count(*) from libros where editorial='Planeta';

También podemos utilizar esta función junto con la clausula "where" para una consulta más específica. Por ejemplo, solicitamos la cantidad de libros que contienen la cadena "Borges":

select count(*) from libros where autor like '%Borges%';

Para contar los registros que tienen precio (sin los que tienen valor nulo), usamos la función "count()" y en los paréntesis colocamos el nombre del campo que queremos contar:

select count(precio) from libros;

"count(*)" devuelve la cantidad de registros de una tabla (incluyendo los "null") mientras que "count(precio)" devuelve la cantidad de registros en los cuales el campo "precio" no es nulo. No es lo mismo. "count(*)" cuenta registros, si en lugar de un asterisco colocamos como argumento el nombre de un campo, se contabilizan los registros cuyo valor en ese campo no es nulo.

No debe haber espacio entre el nombre de la función y el paréntesis, porque puede confundirse con una referencia a una tabla o campo. Las siguientes sentencias son distintas:

select count(*) from libros; select count (*) from libros;

La primera es correcta, la segunda incorrecta.

Selección de un grupo de registros (having)

Así como la cláusula "where" permite seleccionar (o rechazar) registros individuales; la cláusula "having" permite seleccionar (o rechazar) un grupo de registros.

Si queremos saber la cantidad de libros agrupados por editorial usamos la siguiente instrucción ya aprendida:

select editorial, count(*) from libros group by editorial;

Si queremos saber la cantidad de libros agrupados por editorial pero considerando sólo algunos grupos, por ejemplo, los que devuelvan un valor mayor a 2, usamos la siguiente instrucción:

select editorial, count(*) from libros group by editorial having count(*)>2;

Se utiliza "having", seguido de la condición de búsqueda, para seleccionar ciertas filas devuelvedas por la cláusula "group by".

Veamos otros ejemplos. Queremos el promedio de los precios de los libros agrupados por editorial:

select editorial, avg(precio) from libros group by editorial;

Selección de un grupo de registros (having)

Ahora, sólo queremos aquellos cuyo promedio supere los 25 euros:

```
select editorial, avg(precio) from libros group by editorial having avg(precio)>25;
```

En algunos casos es posible confundir las cláusulas "where" y "having". Queremos contar los registros agrupados por editorial sin tener en cuenta a la editorial "Planeta".

Analicemos las siguientes sentencias:

```
select editorial, count(*) from libros where editorial<>'Planeta' group by editorial;
```

```
select editorial, count(*) from libros group by editorial having editorial<>'Planeta';
```

Ambas devuelven el mismo resultado, pero son diferentes.

La primera, selecciona todos los registros rechazando los de editorial "Planeta" y luego los agrupa para contarlos. La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza la cuenta correspondiente a la editorial "Planeta".

Selección de un grupo de registros (having)

No debemos confundir la cláusula "where" con la cláusula "having"; la primera establece condiciones para la selección de registros de un "select"; la segunda establece condiciones para la selección de registros de una salida "group by".

Veamos otros ejemplos combinando "where" y "having".

Queremos la cantidad de libros, sin considerar los que tienen precio nulo, agrupados por editorial, sin considerar la editorial "Planeta":

select editorial, count(*) from libros where precio is not null group by editorial having editorial<>'Planeta';

Aquí, selecciona los registros rechazando los que no cumplan con la condición dada en "where", luego los agrupa por "editorial" y finalmente rechaza los grupos que no cumplan con la condición dada en el "having".

Selección de un grupo de registros (having)

Generalmente se usa la cláusula "having" con funciones de agrupamiento, esto no puede hacerlo la cláusula "where". Por ejemplo queremos el promedio de los precios agrupados por editorial, de aquellas editoriales que tienen más de 2 libros:

```
select editorial, avg(precio) from libros group by editorial having count(*) > 2;
```

Podemos encontrar el mayor valor de los libros agrupados por editorial y luego seleccionar las filas que tengan un valor mayor o igual a 30:

```
select editorial, max(precio) from libros group by editorial having
max(precio)>=30;
```

Esta misma sentencia puede usarse empleando un "alias", para hacer referencia a la columna de la expresión:

```
select editorial, max(precio) as 'mayor' from libros group by editorial having
mayor>=30;
```

Registros duplicados (distinct)

Con la cláusula "distinct" se especifica que los registros con ciertos datos duplicados sean obviadas en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

select autor from libros;

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

select distinct autor from libros; o ***select autor from libros group by autor;***

Note que en los tres casos anteriores aparece "null" como un valor para "autor"· Si sólo queremos la lista de autores conocidos, es decir, no queremos incluir "null" en la lista, podemos utilizar la sentencia siguiente:

select distinct autor from libros where autor is not null;

Para contar los distintos autores, sin considerar el valor "null" usamos:

select count(distinct autor) from libros;

Si contamos los autores sin "distinct", no incluirá los "null" pero si los repetidos:

select count(autor) from libros;

Esta sentencia cuenta los registros que tienen autor.

Registros duplicados (distinct)

Para obtener los nombres de las editoriales usamos: ***select editoriales from libros;***

Para una consulta en la cual los nombres no se repitan usamos:

select distinct editorial from libros;

Podemos saber la cantidad de editoriales distintas usamos:

select count(distinct editoriales) from libros;

Podemos combinarla con "where". Por ejemplo, queremos conocer los distintos autores de la editorial "Planeta": ***select distinct autor from libros where editorial='Planeta';***

También puede utilizarse con "group by":

select editorial, count(distinct autor) from libros group by editorial;

Para mostrar los títulos de los libros sin repetir títulos, usamos:

select distinct titulo from libros order by titulo;

La cláusula "distinct" afecta a todos los campos presentados. Para mostrar los títulos y editoriales de los libros sin repetir títulos ni editoriales, usamos:

select distinct titulo,editorial from libros order by titulo;

Note que los registros no están duplicados, aparecen títulos iguales pero con editorial diferente, cada registro es diferente.

Valores Null

Analizaremos la estructura de una tabla que vemos al utilizar el comando "describe". Tomamos como ejemplo la tabla "libros":

Field	Type		Null	Key	Default	Extra
codigo	int(11)	7 b..	NO	PRI	auto_increment	
titulo	varchar(20)	11 b..	YES		(NULL)	
autor	varchar(30)	11 b..	YES		(NULL)	
editorial	varchar(15)	11 b..	YES		(NULL)	
precio	float	5 b..	YES		(NULL)	

- La primera columna indica el tipo de dato de cada campo.
- La segunda columna "Null" especifica si el campo permite valores nulos; vemos que en el campo "codigo", aparece "NO" y en las demás "YES", esto significa que el primer campo no acepta valores nulos (porque es clave primaria) y los otros si los permiten.
- La tercera columna "Key", muestra los campos que son clave primaria; en el campo "codigo" aparece "PRI" (es clave primaria) y los otros están vacíos, porque no son clave primaria.
- La cuarta columna "Default", muestra los valores por defecto, esto es, los valores que MySQL ingresa cuando omitimos un dato o colocamos un valor inválido; para todos los campos, excepto para el que es clave primaria, el valor por defecto es "null".
- La quinta columna "Extra", muestra algunos atributos extra de los campos; el campo "codigo" es "auto_increment".

Valores Null

Vamos a explicar los valores nulos.

"null' significa "dato desconocido" o "valor inexistente". No es lo mismo que un valor 0, una cadena vacía o una cadena literal "null".

A veces, puede desconocerse o no existir el dato correspondiente a algún campo de un registro. En estos casos decimos que el campo puede contener valores nulos. Por ejemplo, en nuestra tabla de libros, podemos tener valores nulos en el campo "precio" porque es posible que para algunos libros no le hayamos establecido el precio para la venta.

En contraposición, tenemos campos que no pueden estar vacíos jamás, por ejemplo, los campos que identifican cada registro, como los códigos de identificación, que son clave primaria. Por defecto, es decir, si no lo aclaramos en la creación de la tabla, los campos permiten valores nulos.

Imaginemos que ingresamos los datos de un libro, para el cual aún no hemos definido el precio:

```
insert into libros (titulo,autor,editorial,precio) values ('El aleph','Borges',' Planeta', null);
```

Notamos que el valor "null" no es una cadena de caracteres, no se coloca entre comillas.

Si un campo acepta valores nulos, podemos ingresar "null" cuando no conocemos el valor.

Los campos establecidos como clave primaria no aceptan valores nulos. Nuestro campo clave primaria, está definido "auto_increment"; si intentamos ingresar el valor "null" para este campo, no lo tomará y seguirá la secuencia de incremento.

Valores Null

El campo "titulo", no debería aceptar valores nulos, para establecer este atributo debemos crear la tabla con la siguiente sentencia:

```
create table libros( codigo int auto_increment,
titulo varchar(20) not null
autor varchar(30),
editorial varchar(15),
precio float,
primary key (codigo) );
```

Entonces, para que un campo no permita valores nulos debemos especificarlo luego de definir el campo, agregando "not null". Por defecto, los campos permiten valores nulos, pero podemos especificarlo igualmente agregando "null".

Explicamos que "null" no es lo mismo que una cadena vacía o un valor 0 (cero).

Valores Null

Para recuperar los registros que contengan el valor "null" en el campo "precio" no podemos utilizar los operadores relacionales vistos anteriormente: = (igual) y <> (distinto); debemos utilizar los operadores "is null" (es igual a null) y "is not null" (no es null):

```
select * from libros where precio is null;
```

La sentencia anterior tendrá una salida diferente a la siguiente:

```
select * from libros where precio=0;
```

Con la primera sentencia veremos los libros cuyo precio es igual a "null" (desconocido); con la segunda, los libros cuyo precio es 0.

Igualmente para campos de tipo cadena, las siguientes sentencias "select" no devuelven los mismos registros:

```
select * from libros where editorial is null;
```

```
select * from libros where editorial='';
```

Con la primera sentencia veremos los libros cuya editorial es igual a "null", con la segunda, los libros cuya editorial guarda una cadena vacía.

Valores por defecto

Para campos de cualquier tipo no declarados "not null" el valor por defecto es "null" (excepto para tipos "timestamp" que no trataremos aquí).

Para campos declarados "not null", el valor por defecto depende del tipo de dato. Para cadenas de caracteres el valor por defecto es una cadena vacía. Para valores numéricos el valor por defecto es 0; en caso de ser "auto_increment" es el valor mayor existente+1 comenzando en 1. Para campos de tipo fecha y hora, el valor por defecto es 0 (por ejemplo, en un campo "date" es "0000-00-00").

Para todos los tipos, excepto "blob", "text" y "auto_increment" se pueden explicitar valores por defecto con la cláusula "default“.

Un valor por defecto se inserta cuando no está presente al ingresar un registro y en algunos casos en que el dato ingresado es inválido.

Valores por defecto

Los campos para los cuales no se ingresaron valores tomarán los valores por defecto según el tipo de dato del campo, en el campo "codigo" ingresará el siguiente valor de la secuencia porque es "auto_increment"; en el campo "titulo", ingresará una cadena vacía porque es "varchar not null"; en el campo "editorial" almacenará "null", porque no está definido "not null"; en el campo "precio" guardará "null" porque es el valor por defecto de los campos no definidos como "not null" y en el campo "cantidad" ingresará 0 porque es el valor por defecto de los campos numéricos que no admiten valores nulos.

Tipo	Valor por defecto	Cláusula	"default"
carácter	not null	cadena vacía	permite
numérico	not null	0	permite
fecha	not null	0000-00-00	permite
hora	not null	00:00:00	permite
auto_increment	siguiente de la sec.,	empieza en 1	no permite
carac.,numer.,fecha,hora	null	null	permite

Valores no válidos

Hemos visto los valores por defecto de los distintos tipos de datos. Un valor por defecto se inserta cuando no está presente al ingresar un registro y en algunos casos en que el dato ingresado es no válido. Un valor es no válido por tener un tipo de dato incorrecto para el campo o por estar fuera de rango.

Veamos los distintos tipos de datos no válidos.

- Para campos de tipo carácter:
  - valor numérico: si en un campo definido de tipo caracter ingresamos un valor numérico, lo convierte automáticamente a cadena. Por ejemplo, si guardamos 234 en un varchar, almacena '234'.
  - mayor longitud: si intentamos guardar una cadena de caracteres mayor a la longitud definida, la cadena se corta guardando sólo la cantidad de caracteres que quepa. Por ejemplo, si definimos un campo de tipo varchar(10) y le asignamos la cadena 'Buenas tardes', se almacenará 'Buenas tar’ ajustándose a la longitud de 10.

Valores no válidos

- Para campos numéricos:
  - cadenas: si en un campo numérico ingresamos una cadena, lo pasa por alto y coloca 0. Por ejemplo, si en un campo de tipo "integer" guardamos 'abc', almacenará 0.
  - valores fuera de rango: si en un campo numérico intentamos guardar un valor fuera de rango, se almacena el valor límite del rango más cercano (menor o mayor). Por ejemplo, si definimos un campo 'tinyint' (cuyo rango va de -128 a 127) e intentamos guardar el valor 200, se almacenará 127, es decir el máximo permitido del rango; si intentamos guardar -200, se guardará -128, el mínimo permitido por el rango. Otro ejemplo, si intentamos guardar el valor 1000.00 en un campo definido como decimal(5,2) guardará 999.99 que es el mayor del rango.
  - valores incorrectos: si cargamos en un campo definido de tipo decimal un valor con más decimales que los permitidos en la definición, el valor es redondeado al más cercano. Por ejemplo, si cargamos en un campo definido como decimal(4,2) el valor 22.229, se guardará 22.23, si cargamos 22.221 se guardará 22.22.
- Para campos definidos auto_increment:
  - Pasa por alto los valores fuera del rango, 0 en caso de no ser "unsigned" y todos los menores a 1 en caso de ser "unsigned".
  - Si ingresamos un valor fuera de rango continúa la secuencia.
  - Si ingresamos un valor existente, aparece un mensaje de error indicando que el valor ya existe.

Valores no válidos

- Para campos de fecha y hora:
  - valores incorrectos: si intentamos almacenar un valor que MySql no reconoce como fecha (sea fuera de rango o un valor inválido), convierte el valor en ceros (según el tipo y formato). Por ejemplo, si intentamos guardar '20/07/2006' en un campo definido de tipo "date", se almacena '0000-00-00'. Si intentamos guardar '20/07/2006 15:30' en un campo definido de tipo "datetime", se almacena '0000-00-00 00:00:00'. Si intentamos almacenar un valor no válido en un campo de tipo "time", se guarda ceros.Para "time", si intentamos cargar un valor fuera de rango, se guarda el menor o mayor valor permitido (según sea uno u otro el más cercano).
- Para campos de cualquier tipo:
  - valor "null": si un campo está definido "not null" e intentamos ingresar "null", aparece un mensaje de error y la sentencia no se ejecuta.

Tipo de dato Enum

El tipo de dato "enum" representa una enumeración. Puede tener un máximo de 65535 valores distintos. Es una cadena cuyo valor se elige de una lista enumerada de valores permitidos que se especifica al definir el campo. Puede ser una cadena vacía, incluso "null".

Los valores presentados como permitidos tienen un valor de índice que comienza en 1.

Una empresa necesita personal, varias personas se han presentado para cubrir distintos cargos. La empresa almacena los datos de los solicitantes a los puestos en una tabla llamada “solicitantes". Le interesa, entre otras cosas, conocer los estudios que tiene cada persona, si tiene estudios primario, secundario, terciario, universitario o ninguno. Para ello, crea un campo de tipo "enum" con esos valores. Para definir un campo de tipo "enum" usamos la siguiente sintaxis al crear la tabla:

```
create table solicitantes(
numero int unsigned auto_increment,
documento char(8),
nombre varchar(30),
estudios enum ('ninguno','primario','secundario', 'terciario','universitario'),
primary key(numero));
```

Los valores presentados deben ser cadenas de caracteres.

Tipo de dato Enum

Si un "enum" permite valores nulos, el valor por defecto el "null"; si no permite valores nulos, el valor por defecto es el primer valor de la lista de permitidos.

Si se ingresa un valor numérico, lo interpreta como índice de la enumeración y almacena el valor de la lista con dicho número de índice. Por ejemplo:

```
insert into solicitantes (documento,nombre,estudios) values('22255265','Juana Pereyra',5);
```

En el campo "estudios" almacenará "universitario" que es valor de índice 5.

Si se ingresa un valor inválido, puede ser un valor no presente en la lista o un valor de índice fuera de rango, coloca una cadena vacía. Por ejemplo:

```
insert into solicitantes (documento,nombre,estudios) values('22255265','Juana Pereyra',0);
insert into solicitantes (documento,nombre,estudios) values('22255265','Juana Pereyra',6);
insert into solicitantes (documento,nombre,estudios) values('22255265','Juana Pereyra','PostGrado');
```

En los 3 casos guarda una cadena vacía, en los 2 primeros porque los índices ingresados están fuera de rango y en el tercero porque el valor no está incluido en la lista de permitidos.

Tipo de dato Enum

Esta cadena vacía de error, se diferencia de una cadena vacía permitida porque la primera tiene el valor de índice 0; entonces, podemos seleccionar los registros con valores inválidos en el campo de tipo "enum" así:

select * from solicitantes where estudios=0;

El índice de un valor "null" es "null".

Para seleccionar registros con un valor específico de un campo enumerado usamos "where", por ejemplo, queremos todos los postulantes con estudios universitarios:

select * from solicitantes where estudios='universitario';

Los tipos "enum" aceptan cláusula "default".

Si el campo está definido como "not null" e intenta almacenar el valor "null" aparece un mensaje de error y la sentencia no se ejecuta.

Los bytes de almacenamiento del tipo "enum" depende del número de valores enumerados.

Tipo de dato Set

El tipo de dato "set" representa un conjunto de cadenas.

- Puede tener 1 ó más valores que se eligen de una lista de valores permitidos que se especifican al definir el campo y se separan con comas. Puede tener un máximo de 64 miembros. Ejemplo: un campo definido como set ('a', 'b') not null, permite los valores 'a', 'b' y ‘a, b'. Si carga un valor no incluido en el conjunto "set", se ignora y almacena cadena vacía. Es similar al tipo "enum" excepto que puede almacenar más de un valor en el campo.
- Una empresa necesita personal, varias personas se han presentado para cubrir distintos cargos. La empresa almacena los datos de los solicitantes a los puestos en una tabla llamada “solicitantes". Le interesa, entre otras cosas, saber los distintos idiomas que conoce cada persona; para ello, crea un campo de tipo "set" en el cual guardará los distintos idiomas que conoce cada postulante. Para definir un campo de tipo "set" usamos la siguiente sintaxis:

```
create table solicitantes(
numero int unsigned auto_increment,
documento char(8),
nombre varchar(30),
idioma set('ingles','italiano','portuges'),
primary key(numero) );
```

Ingresamos un registro:

```
insert into postulantes (documento,nombre,idioma) values('22555444','Ana Acosta','ingles');
```

Tipo de dato Set

Para ingresar un valor que contenga más de un elemento del conjunto, se separan por comas, p.ej.: ***insert into solicitantes (documento,nombre,idioma) values('23555444','Juana Pereyra','ingles,italiano');***

No importa el orden en el que se inserten, se almacenan en el orden que han sido definidos, p. ej.: ***insert into solicitantes (documento,nombre,idioma) values('23555444','Juana Pereyra','italiano,ingles');***

En el campo "idioma" guardará 'ingles,italiano'. Tampoco importa si se repite algún valor, cada elemento repetido, se ignora y se guarda una vez y en el orden que ha sido definido, p. ej.:

insert into solicitantes (documento,nombre,idioma) values('23555444','Juana Pereyra','italiano,ingles,italiano');

En el campo "idioma" guardará 'ingles,italiano'. Si ingresamos un valor que no está en la lista "set", se ignora y se almacena una cadena vacía, por ejemplo:

insert into solicitantes (documento,nombre,idioma) values('22255265','Juana Pereyra','frances');

Si un "set" permite valores nulos, el valor por defecto es "null"; si no permite valores nulos, el valor por defecto es una cadena vacía. Si se ingresa un valor de índice fuera de rango, coloca una cadena vacía. Por ejemplo:

insert into solicitantes (documento,nombre,idioma) values('22255265','Juana Pereyra',0);

insert into solicitantes (documento,nombre,idioma) values('22255265','Juana Pereyra',8);

Si se ingresa un valor numérico, lo interpreta como índice de la enumeración y almacena el valor de la lista con dicho número de índice.

Tipo de dato Set

Si se ingresa un valor numérico, lo interpreta como índice de la enumeración y almacena el valor de la lista con dicho número de índice. Los valores de índice se definen en el siguiente orden, en este ejemplo:

1='ingles',

2='italiano',

3='ingles,italiano',

4='portugues',

5='ingles,portugues',

6='italiano,portugues',

7='ingles,italiano,portugues'.

Ingresamos algunos registros con valores de índice:

```
insert into solicitantes (documento,nombre,idioma) values('22255265','Juana Pereyra',2);
insert into  solicitantes (documento,nombre,idioma) values('22555888','Juana Pereyra',3);
```

En el campo "idioma", con la primera inserción se almacenará "italiano" que es valor de índice 2 y con la segunda inserción, "ingles,italiano" que es el valor con índice 3.

Tipo de dato Set

Para búsquedas de valores en campos "set" se utiliza el operador "like" o la función "find_in_set()".

Para recuperar todos los valores que contengan la cadena "ingles" podemos usar cualquiera de las siguientes sentencias:

```
select * from postulantes where idioma like '%ingles%';
```

```
select * from postulantes where find_in_set('ingles',idioma)>0;
```

La función "find_in_set()" devuelve 0 si el primer argumento (cadena) no se encuentra en el campo set colocado como segundo argumento. Esta función no funciona correctamente si el primer argumento contiene una coma.

Para recuperar todos los valores que incluyan "ingles,italiano" introducimos:

```
select * from postulantes where idioma like '%ingles,italiano%';
```

Tipo de dato Set

Para realizar búsquedas, es importante respetar el orden en que se presentaron los valores en la definición del campo; por ejemplo, si se busca el valor "italiano,ingles" en lugar de "ingles,italiano", no devolverá registros.

Para buscar registros que contengan sólo el primer miembro del conjunto "set" usamos:

```
select * from solicitantes where idioma='ingles';
```

También podemos buscar por el número de índice:

```
select * from solicitantes where idioma=1;
```

Para buscar los registros que contengan el valor "ingles,italiano" podemos utilizar cualquiera de las siguientes sentencias:

```
select * from solicitantes where idioma='ingles,italiano';
```

```
select * from solicitantes where idioma=3;
```

También podemos usar el operador "not". Para recuperar todos los valores que no contengan la cadena "ingles" podemos usar cualquiera de las siguientes sentencias:

```
select * from solicitantes where idioma not like '%ingles%';
```

```
select * from solicitantes where not find_in_set('ingles',idioma)>0;
```

Los tipos "set" admiten cláusula "default". Los bytes de almacenamiento del tipo "set" depende del número de miembros, se calcula así: (cantidad de miembros+7)/8 bytes; entonces puede ser 1,2,3,4 u 8 bytes.

Modificación de registros de una tabla (update)

Para modificar uno o varios datos de uno o varios registros utilizamos "update" (actualizar).

Por ejemplo, en nuestra tabla "usuarios", queremos cambiar los valores de todas las claves, por "RealMadrid":

```
update usuarios set clave='RealMadrid';
```

Utilizamos "update" junto al nombre de la tabla y "set" junto con el campo a modificar y su nuevo valor. El cambio afectará a todos los registros.

Podemos modificar algunos registros, para ello debemos establecer condiciones de selección con "where".

Por ejemplo, queremos cambiar el valor correspondiente a la clave de nuestro usuario llamado 'MarioPerez', queremos como nueva clave 'Boca', necesitamos una condición "where" que afecte solamente a este registro:

```
update usuarios set clave='Boca' where nombre='MarioPerez';
```

Si no encuentra registros que cumplan con la condición del "where", ningún registro es afectado.

Modificación de registros de una tabla (update)

Las condiciones no son obligatorias, pero si omitimos la cláusula "where", la actualización afectará a todos los registros.

También se puede actualizar varios campos en una sola instrucción:

```
update usuarios set nombre='MarceloDuarte', clave='Marce' where nombre='Marcelo';
```

Para ello colocamos "update", el nombre de la tabla, "set" junto al nombre del campo y el nuevo valor y separado por coma, el otro nombre del campo con su nuevo valor.

Comando truncate table

Aprendimos que para borrar todos los registro de una tabla se usa "delete" sin condición "where". También podemos eliminar todos los registros de una tabla con "truncate table". Por ejemplo, queremos vaciar la tabla "libros", usamos:

truncate table libros;

La sentencia "truncate table" vacía la tabla (elimina todos los registros) y vuelve a crear la tabla con la misma estructura.

La diferencia con "drop table" es que esta sentencia borra la tabla, "truncate table" la vacía. La diferencia con "delete" es la velocidad, es más rápido "truncate table" que "delete" (se nota cuando la cantidad de registros es muy grande) ya que éste borra los registros uno a uno.

Otra diferencia es la siguiente: cuando la tabla tiene un campo "auto_increment", si borramos todos los registros con "delete" y luego ingresamos un registro, al cargarse el valor en el campo auto_increment, continúa con la secuencia teniendo en cuenta el valor mayor que se había guardado; si usamos "truncate table" para borrar todos los registros, al ingresar otra vez un registro, la secuencia del campo auto_increment vuelve a iniciarse en 1.

Por ejemplo, tenemos la tabla "libros" con el campo "codigo" definido "auto_increment", y el valor más alto de ese campo es "5", si borramos todos los registros con "delete" y luego ingresamos un registro sin valor de código, se guardará el valor "6"; si en cambio, vaciamos la tabla con "truncate table", al ingresar un nuevo registro sin valor para el código, iniciará la secuencia en 1 nuevamente.

Agregar campos a una tabla (alter table - add)

Para modificar la estructura de una tabla existente, usamos "alter table". "alter table" se usa para:

- agregar nuevos campos,
- eliminar campos existentes,
- modificar el tipo de dato de un campo,
- agregar o quitar modificadores como "null", "unsigned", "auto_increment",
- cambiar el nombre de un campo,
- agregar o eliminar la clave primaria,
- agregar y eliminar índices,
- renombrar una tabla.

"alter table" hace una copia temporal de la tabla original, realiza los cambios en la copia, luego borra la tabla original y renombra la copia.

Agregar campos a una tabla (alter table - add)

Aprenderemos a agregar campos a una tabla. Para ello utilizamos nuestra tabla "libros", definida con la siguiente estructura:

- código, int unsigned auto_increment, clave primaria,
- titulo, varchar(40) not null,
- autor, varchar(30),
- editorial, varchar (20),
- precio, decimal(5,2) unsigned.

Necesitamos agregar el campo "cantidad", de tipo int unsigned not null, usamos:

alter table libros add cantidad int unsigned not null;

Usamos "alter table" seguido del nombre de la tabla y "add" seguido del nombre del nuevo campo con su tipo y los modificadores. Agreguemos otro campo a la tabla:

alter table libros add edicion date;

Si intentamos agregar un campo con un nombre existente, aparece un mensaje de error indicando que el campo ya existe y la sentencia no se ejecuta.

Cuando se agrega un campo, si no especificamos, lo coloca al final, después de todos los campos existentes; podemos indicar su posición (luego de qué campo debe aparecer) con "after":

alter table libros add cantidad int unsigned after autor;

Eliminar campos de una tabla (alter table - drop)

"alter table" nos permite alterar la estructura de la tabla, podemos usarla para eliminar un campo. Continuamos con nuestra tabla "libros". Para eliminar el campo "edicion" utilizamos:

alter table libros drop edicion;

Entonces, para borrar un campo de una tabla usamos "alter table" junto con "drop" y el nombre del campo a eliminar.

Si intentamos borrar un campo inexistente aparece un mensaje de error y la acción no se realiza.

Podemos eliminar 2 campos en una misma sentencia:

alter table libros drop editorial, drop cantidad;

Si se borra un campo de una tabla que es parte de un índice, también se borra el índice. Si una tabla tiene sólo un campo, éste no puede ser borrado. Hay que tener cuidado al eliminar un campo, éste puede ser clave primaria. Es posible eliminar un campo que es clave primaria, no aparece ningún mensaje:

alter table libros drop codigo;

Si eliminamos un campo clave, la clave también se elimina.

Modificar campos de una tabla (alter table - modify)

Con "alter table" podemos modificar el tipo de algún campo incluidos sus atributos.

Queremos modificar el tipo del campo "cantidad", como guardaremos valores que no superarán los 50000 usaremos smallint unsigned, usamos:

alter table libros modify cantidad smallint unsigned;

Usamos "alter table" seguido del nombre de la tabla y "modify" seguido del nombre del nuevo campo con su tipo y los modificadores.

Queremos modificar el tipo del campo "titulo" para poder almacenar una longitud de 40 caracteres y que no permita valores nulos, introducimos:

alter table libros modify titulo varchar(40) not null;

Hay que tener cuidado al alterar los tipos de los campos de una tabla que ya tiene registros cargados. Si tenemos un campo de texto de longitud 50 y lo cambiamos a 30 de longitud, los registros cargados en ese campo que superen los 30 caracteres, se cortarán. Igualmente, si un campo fue definido permitiendo valores nulos, se cargaron registros con valores nulos y luego se lo define "not null", todos los registros con valor nulo para ese campo cambiarán al valor por defecto según el tipo (cadena vacía para tipo texto y 0 para numéricos), ya que "null" se convierte en un valor inválido.

Modificar campos de una tabla (alter table - modify)

Si definimos un campo de tipo decimal(5,2) y tenemos un registro con el valor "900.00" y luego modificamos el campo a "decimal(4,2)", el valor "900.00" se convierte en un valor inválido para el tipo, entonces guarda en su lugar, el valor límite más cercano, "99.99".

Si intentamos definir "auto_increment" un campo que no es clave primaria, aparece un mensaje de error indicando que el campo debe ser clave primaria. Por ejemplo:

alter table libros modify codigo int unsigned auto_increment;

"alter table" combinado con "modify" permite agregar y quitar campos y atributos de campos. Para modificar el valor por defecto ("default") de un campo podemos usar también "modify" pero debemos colocar el tipo y sus modificadores, entonces resulta muy extenso, podemos cambiar sólo el valor por defecto con la siguiente sintaxis:

alter table libros alter autor set default 'Varios';

Para eliminar el valor por defecto podemos emplear:

alter table libros alter autor drop default;

Cambiar el nombre de un campo de una tabla (alter table - change)

Con "alter table" podemos cambiar el nombre de los campos de una tabla. Continuamos con nuestra tabla "libros", definida con la siguiente estructura:

- código, int unsigned auto_increment,
- nombre, varchar(40),
- autor, varchar(30),
- editorial, varchar (20),
- coste, decimal(5,2) unsigned,
- cantidad int unsigned,
- clave primaria: código.

Queremos cambiar el nombre del campo "coste" por "precio":

alter table libros change coste precio decimal (5,2);

Usamos "alter table" seguido del nombre de la tabla y "change" seguido del nombre actual y el nombre nuevo con su tipo y los modificadores.

Con "change" cambiamos el nombre de un campo y también podemos cambiar el tipo y sus modificadores. Por ejemplo, queremos cambiar el nombre del campo "nombre" por "titulo" y redefinirlo como "not null", usamos:

alter table libros change nombre titulo varchar(40) not null;

Agregar y eliminar la clave primaria (alter table)

Hasta ahora hemos aprendido a definir una clave primaria al momento de crear una tabla. Con "alter table" podemos agregar una clave primaria a una tabla existente.

Continuamos con nuestra tabla "libros", definida con la siguiente estructura:

- código, int unsigned auto_increment,
- titulo, varchar(40),
- autor, varchar(30),
- editorial, varchar (20),
- precio, decimal(5,2) unsigned,
- cantidad smallint unsigned.

Para agregar una clave primaria a una tabla existente usamos:

alter table libros add primary key (codigo);

Usamos "alter table" con "add primary key" y entre paréntesis el nombre del campo que será clave.

Si intentamos agregar otra clave primaria, aparecerá un mensaje de error porque (recuerde) una tabla solamente puede tener una clave primaria.

Agregar y eliminar la clave primaria (alter table)

Para que un campo agregado como clave primaria sea autoincrementable, es necesario agregarlo como clave y luego redefinirlo con "modify" como "auto_increment". No se puede agregar una clave y al mismo tiempo definir el campo auto_increment. Tampoco es posible definir un campo como auto_increment y luego agregarlo como clave porque para definir un campo "auto_increment" éste debe ser clave primaria.

También usamos "alter table" para eliminar una clave primaria. Para eliminar una clave primaria usamos:

alter table libros drop primary key;

Con "alter table" y "drop primary key" eliminamos una clave primaria definida al crear la tabla o agregada luego.

Si queremos eliminar la clave primaria establecida en un campo "auto_increment" aparece un mensaje de error y la sentencia no se ejecuta porque si existe un campo con este atributo DEBE ser clave primaria. Primero se debe modificar el campo quitándole el atributo "auto_increment" y luego se podrá eliminar la clave.

Si intentamos establecer como clave primaria un campo que tiene valores repetidos, aparece un mensaje de error y la operación no se realiza.

Remplazar registros (replace)

Cuando intentamos ingresar con "insert" un registro que repite el valor de un campo clave o indexado con índice único, aparece un mensaje de error indicando que el valor está duplicado. Si empleamos "replace" en lugar de "insert", el registro existente se borra y se ingresa el nuevo, de esta manera no se duplica el valor único.

Si tenemos la tabla "libros" con el campo "codigo" establecido como clave primaria e intentamos ingresar ("insert") un valor de código existente, aparece un mensaje de error porque no está permitido repetir los valores del campo clave. Si empleamos "replace" en lugar de "insert", la sentencia se ejecuta reemplazando el registro con el valor de código existente por el nuevo registro. Veamos un ejemplo. Tenemos los siguientes registros almacenados en "libros":

codigo	titulo	autor	editorial	precio
10	Alicia en ..	Lewis Carroll	Emece	15.4
15	Aprenda PHP	Mario Molina	Planeta	45.8
23	El aleph	Borges	Planeta	23.0

Intentamos insertar un registro con valor de clave repetida (código 23):

```
insert into libros values(23,'Java en 10 minutos','Mario Molina','Emece',25.5);
```

Aparece un mensaje de error indicando que hay registros duplicados.

Remplazar registros (replace)

Si empleamos "replace":

replace into libros values(23,'Java en 10 minutos','Mario Molina','Emece',25.5);

La sentencia se ejecuta y aparece un mensaje indicando que se afectaron 2 filas, esto es porque un registro se eliminó y otro se insertó. "replace" funciona como "insert" en los siguientes casos: - si los datos ingresados no afectan al campo único, es decir no se ingresa valor para el campo indexado:

replace into libros(titulo,autor,editorial,precio) values('Cervantes y el quijote','Borges','Paidos',28);

Aparece un mensaje indicando que afectó un solo registro, el ingresado, que se guarda con valor de código 0.

- Si el dato para el campo indexado que se ingresa no existe:

  replace into libros values(30,'Matematica estas ahi','Paenza','Paidos',12.8);

  Aparece un mensaje indicando que se afectó solo una fila, no hubo reemplazo porque el código no existía antes de la nueva inserción.

- Si la tabla no tiene indexación. Si la tabla "libros" no tuviera establecida ninguna clave primaria (ni índice único), podríamos ingresar varios registros con igual código:

  replace into libros values(1,'Harry Potter ya la piedra filosofal', 'J.K.Rowling','Salamandra',29);

  aparecería un mensaje indicando que se afectó 1 registro (el ingresado), no se reemplazó ninguno y ahora habría 2 libros con código 1.

Columnas calculadas

Es posible obtener salidas en las cuales una columna sea el resultado de un cálculo y no un campo de una tabla.

Si queremos ver los títulos, precio y cantidad de cada libro escribimos la siguiente sentencia:

select titulo, precio, cantidad from libros;

Si queremos saber el monto total en dinero de un título podemos multiplicar el precio por la cantidad por cada título, pero también podemos hacer que MySQL realice el cálculo y lo incluya en una columna extra en la salida:

select titulo, precio, cantidad, precio*cantidad from libros;

Si queremos saber el precio de cada libro con un 10% de descuento podemos incluir en la sentencia los siguientes cálculos:

select titulo, precio, precio*0.1,precio-(precio*0.1) from libros;

Funciones de agrupamiento (count - max - min - sum - avg)

La función "sum()" devuelve la suma de los valores que contiene el campo especificado. Por ejemplo, queremos saber la cantidad de libros que tenemos disponibles para la venta:

```
select sum(cantidad) from libros;
```

También podemos combinarla con "where". Por ejemplo, queremos saber cuántos libros tenemos de la editorial "Planeta":

```
select sum(cantidad) from libros where editorial ='Planeta';
```

Para averiguar el valor máximo o mínimo de un campo usamos las funciones "max()" y "min()“ respectivamente. Ejemplo, queremos saber cuál es el mayor precio de todos los libros:

```
select max(precio) from libros;
```

Queremos saber cuál es el valor mínimo de los libros de "Rowling":

```
select min(precio) from libros where autor like '%Rowling%';
```

Funciones de agrupamiento (count - max - min - sum - avg)

La función avg() devuelve el valor promedio de los valores del campo especificado. Por ejemplo, queremos saber el promedio del precio de los libros referentes a "PHP":

select avg(precio) from libros where titulo like '%PHP%';

Estas funciones se denominan "funciones de agrupamiento" porque operan sobre conjuntos de registros, no con datos individuales.

Tener en cuenta que no debe haber espacio entre el nombre de la función y el paréntesis, porque puede confundirse con una referencia a una tabla o campo. Las siguientes sentencias son distintas:

select count(*) from libros; ❤️

select count (*) from libros; 🚫

La primera es correcta, la segunda incorrecta.

Funciones matemáticas

Los operadores aritméticos son +,-,* y /. Todas las operaciones matemáticas devuelven "null" en caso de error.

MySQL tiene algunas funciones para trabajar con números. Aquí presentamos algunas. NO debe haber espacios entre un nombre de función y los paréntesis porque MySQL puede confundir una llamada a una función con una referencia a una tabla o campo que tenga el mismo nombre de una función.

- **abs(x):** devuelve el valor absoluto del argumento "x". Ejemplo: ***select abs(-20);*** devuelve 20.
- **ceiling(x):** redondea hacia arriba el argumento "x". Ejemplo: ***select ceiling(12.34),*** devuelve 13.
- **floor(x):** redondea hacia abajo el argumento "x". Ejemplo: ***select floor(12.34);*** devuelve 12.
- **greatest(x,y,..):** devuelve el argumento de máximo valor.
- **least(x,y,...):** con dos o más argumentos, devuelve el argumento más pequeño.
- **mod(n,m):** significa "módulo aritmético"; devuelve el resto de "n" dividido en "m". Ej.: ***select mod(10,3);*** devuelve 1. ***select mod(10,2);*** devuelve 0.
- **%:** Devuelve el resto de la división: ***select 10%3;*** devuelve 1. ***select 10%2;*** devuelve 0.
- **power(x,y):** potencia, devuelve "x" elevado a "y". Ejemplo: ***select power(2,3);*** devuelve 8.
- **rand():** devuelve un valor de coma flotante aleatorio dentro del rango 0 a 1.0.
- **round(x,y):** devuelve el argumento "x" redondeado al entero más cercano, “y”, opcional, número de cifras a redondear. Ej.: ***select round(12.34);*** devuelve 12. ***select round(12.64);*** devuelve 13.
- **sqrt():** devuelve la raíz cuadrada del valor enviado como argumento. En inglés “squareroot”.
- **truncate(x,d):** devuelve el número "x", truncado a "d" decimales. Si "d" es 0, el resultado no tendrá parte fraccionaria. Ej.: ***select truncate(123.4567,2);*** devuelve 123.45; ***select truncate (123.4567,0);*** devuelve 123.

Funciones para el manejo de cadenas

NO debe haber espacios entre un nombre de función y los paréntesis porque MySQL puede confundir una llamada a una función con una referencia a una tabla o campo que tenga el mismo nombre de una función. MySQL tiene algunas funciones para trabajar con cadenas de caracteres. Estas son algunas:

- **ord(caracter):** Retorna el código ASCII para el caracter enviado como argumento. Ejemplo: ***select ord('A');*** devuelve 65.
- **char(x,..):** devuelve una cadena con los caracteres en código ASCII de los enteros enviados como argumentos. Ejemplo: ***select char(65,66,67);*** devuelve "ABC".
- **concat(cadena1,cadena2,...):** devuelve la cadena resultado de concatenar los argumentos. Ejemplo: ***select concat('Hola,',' ','como esta?');*** devuelve "Hola, como esta?".
- **concat_ws(separador,cadena1,cadena2,...):** "ws" son las iniciales de "with separator". El primer argumento especifica el separador que utiliza para los demás argumentos; el separador se agrega entre las cadenas a concatenar. Ejemplo: ***select concat_ws('-','Juan','Pedro','Luis');*** devuelve "Juan-Pedro-Luis".
- **find_in_set(cadena,lista de cadenas):** devuelve un valor entre de 0 a n (correspondiente a la posición), si la cadena envidada como primer argumento está presente en la lista de cadenas enviadas como segundo argumento. La lista de cadenas enviada como segundo argumento es una cadena formada por subcadenas separadas por comas. Devuelve 0 si no encuentra "cadena" en la "lista de cadenas". Ejemplo: ***select find_in_set('hola','como esta,hola,buen dia');*** devuelve 2, porque la cadena "hola" se encuentra en la lista de cadenas, en la posición 2.
- **length(cadena):** devuelve la longitud de la cadena enviada como argumento. Ejemplo: ***select length('Hola');*** devuelve 4.

Funciones para el manejo de cadenas

- **locate(subcadena,cadena):** devuelve la posición de la primera ocurrencia de la subcadena en la cadena enviadas como argumentos. Devuelve "0" si la subcadena no se encuentra en la cadena. Ejemplo: ***select locale('o','como le va');*** devuelve 2.
- **position(subcadena in cadena):** funciona como "locate()". Devuelve "0" si la subcadena no se encuentra en la cadena. Ejemplo: ***select position('o' in 'como le va');*** devuelve 2.
- **locate(subcadena,cadena,posicioninicial):** devuelve la posición de la primera ocurrencia de la subcadena enviada como primer argumentos en la cadena enviada como segundo argumento, empezando en la posición enviada como tercer argumento. Devuelve "0" si la subcadena no se encuentra en la cadena. Ejemplos: ***select locate('ar','Margarita',1);*** devuelve 2. ***select locate('ar','Margarita',3);*** devuelve 5.
- **instr(cadena,subcadena):** devuelve la posición de la primera ocurrencia de la subcadena enviada como segundo argumento en la cadena enviada como primer argumento. Ejemplo: ***select instr('como le va','om');*** devuelve 2.
- **lpad(cadena,longitud,cadenarelleno):** devuelve la cadena enviada como primer argumento, rellenada por la izquierda con la cadena enviada como tercer argumento hasta que la cadena devuelta tenga la longitud especificada como segundo argumento. Si la cadena es más larga, la corta. Ejemplo: ***select lpad('hola',10,'0');*** devuelve "000000hola".
- **rpad(cadena,longitud,cadenarelleno):** igual que "lpad" excepto que rellena por la derecha.
- **left(cadena,longitud):** devuelve la cantidad (longitud) de caracteres de la cadena comenzando desde la izquierda, primer carácter. Ejemplo: ***select left('buenos dias',8);*** devuelve "buenos d".
- **right(cadena,longitud):** devuelve la cantidad (longitud) de caracteres de la cadena comenzando desde la derecha, último carácter. Ejemplo: ***select right('buenos dias',8);*** devuelve "nos dias".

Funciones para el manejo de cadenas

- **substring(cadena,posicion,longitud):** devuelve una subcadena de tantos caracteres de longitud como especifica en tercer argumento, de la cadena enviada como primer argumento, empezando desde la posición especificada en el segundo argumento. Ejemplo: ***select substring('Buenas tardes',3,5);*** devuelve "enas".
- **substring(cadena from posicion for longitud):** variante de "substring(cadena,posicion,longitud)". Ejemplo: ***select substring('Buenas tardes' from 3 for 5);***
- **mid(cadena,posicion,longitud):** igual que "substring(cadena,posicion,longitud)". Ejemplo: ***select mid('Buenas tardes' from 3 for 5);*** devuelve "enas".
- **substring(cadena,posicion):** devuelve la subcadena de la cadena enviada como argumento, empezando desde la posición indicada por el segundo argumento. Ejemplo: ***select substring('Margarita',4);*** devuelve "garita".
- **substring(cadena from posicion):** variante de "substring(cadena,posicion)". Ejemplo: ***select substring('Margarita' from 4);*** devuelve "garita".
- **substring_index(cadena,delimitador,ocurrencia):** subcadena de la cadena enviada como argumento antes o después de la "ocurrencia" de la cadena enviada como delimitador. Si "ocurrencia" es positiva, devuelve la subcadena anterior al delimitador (comienza desde la izda); si "ocurrencia" es negativa, devuelve la subcadena posterior al delimitador (comienza desde la dcha). Ejemplo: ***select substring_index('margarita','ar',2);*** devuelve "marg", todo lo anterior a la segunda ocurrencia de "ar". ***select substring_index('margarita','ar',-2);*** devuelve "garita", todo lo posterior a la segunda ocurrencia de "ar".

Funciones para el manejo de cadenas

- **ltrim y rtrim(cadena):** ltrim devuelve la cadena con los espacios de la izquierda eliminados y rtrim los de la derecha. Ejemplo: ***select ltrim(' Hola ');*** devuelve "Hola ", ***select rtrim(' Hola ');*** devuelve " Hola".
- **trim([[both|leading|trailing] [subcadena] from] cadena):** devuelve una cadena igual a la enviada pero eliminando la subcadena prefijo y/o sufijo. Si no se indica ningún especificador (both, leading o trailing) se asume "both" (ambos). Si no se especifica prefijos o sufijos elimina los espacios. Ejemplos:
  - ***select trim(' Hola ');*** devuelve 'Hola'.
  - ***select trim (leading '0' from '00hola00')***; devuelve "hola00".
  - ***select trim (trailing '0' from '00hola00');*** devuelve "00hola".
  - ***select trim (both '0' from '00hola00');*** devuelve "hola".
  - ***select trim ('0' from '00hola00');*** devuelve "hola".
  - ***select trim (' hola ');*** devuelve "hola".
- **replace(cadena,cadenareemplazo,cadenareemplazar):** devuelve la cadena con todas las ocurrencias de la subcadena reemplazo por la subcadena a reemplazar. Ejemplo: ***select replace('xxx.mysql.com','x','w');*** devuelve "www.mysql.com'.
- ***repeat(cadena,cantidad):*** devuelve una cadena consistente en la cadena repetida la cantidad de veces especificada. Si "cantidad" es menor o igual a cero, devuelve una cadena vacía. Ejemplo: ***select repeat('hola',3);*** devuelve "holaholahola".
- **reverse(cadena):** devuelve la cadena invirtiendo el order de los caracteres. Ejemplo: ***select reverse('Hola');*** devuelve "aloH".

Funciones para el manejo de cadenas

- ***insert(cadena,posicion,longitud,nuevacadena):*** devuelve la cadena con la nueva cadena colocándola en la posición indicada por "posicion" y elimina la cantidad de caracteres indicados por "longitud". Ejemplo: ***select insert('buenas tardes',2,6,'xx');*** devuelve ""bxxtardes".
- **lcase(cadena) y lower(cadena):** devuelven la cadena con todos los caracteres en minúsculas. Ejemplo: ***select lower('HOLA ESTUDIAnte');*** devuelve "hola estudiante". ***select lcase('HOLA ESTUDIAnte');*** devuelve "hola estudiante".
- **ucase(cadena) y upper(cadena):** devuelven la cadena con todos los caracteres en mayúsculas. Ejemplo: ***select upper('HOLA ESTUDIAnte');*** devuelve "HOLA ESTUDIANTE". ***select ucase('HOLA ESTUDIAnte');*** devuelve "HOLA ESTUDIANTE".
- **strcmp(cadena1,cadena2):** devuelve 0 si las cadenas son iguales, -1 si la primera es menor que la segunda y 1 si la primera es mayor que la segunda. Ejemplo: ***select strcmp('Hola','Chau');*** devuelve 1.

Funciones para el uso de fecha y hora

MySQL tiene algunas funciones para trabajar con fechas y horas. Estas son algunas:

- **adddate(fecha, interval expresion):** devuelve la fecha agregándole el intervalo especificado. Ejemplos: ***adddate('2006-10-10',interval 25 day)*** devuelve "2006-11-04". ***adddate('2006-10-10',interval 5 month)*** devuelve "2007-03-10".
- **adddate(fecha, dias):** devuelve la fecha agregándole a fecha "dias". Ejemplo: ***adddate('2006-10-10',25)***, devuelve "2006-11-04".
- **addtime(expresion1,expresion2):** agrega expresion2 a expresion1 y devuelve el resultado.
- **current_date:** devuelve la fecha de hoy con formato "YYYY-MM-DD" o "YYYYMMDD".
- **current_time:** devuelve la hora actual con formato "HH:MM:SS" o "HHMMSS".
- **date_add(fecha,interval expresion tipo)** y **date_sub(fecha,interval expresion tipo):** el argumento "fecha" es un valor "date" o "datetime", "expresion" especifica el valor de intervalo a ser añadido o substraído de la fecha indicada (puede empezar con "-", para intervalos negativos), "tipo" indica la medida de adición o substracción. Ejemplo: ***date_add('2006-08-10', interval 1 month)*** devuelve "2006-09-10"; ***date_add('2006-08-10', interval -1 day)*** devuelve "2006-09-09"; ***date_sub('2006-08-10 18:55:44', interval 2 minute)*** devuelve "2006-08-10 18:53:44"; ***date_sub('2006-08-10 18:55:44', interval '2:3' minute_second)*** devuelve "2006-08-10 18:52:41". Los valores para "tipo" pueden ser: second, minute, hour, day, month, year, minute_second (minutos y segundos), hour_minute (horas y minutos), day_hour (días y horas), year_month (año y mes), hour_second (hora, minuto y segundo), day_minute (dias, horas y minutos), day_second(dias a segundos).
- **datediff(fecha1,fecha2):** devuelve la cantidad de días entre fecha1 y fecha2.
- **dayname(fecha):** devuelve el nombre del día de la semana de la fecha. Ejemplo: ***dayname('2006-08-10')*** devuelve "thursday".

Funciones para el uso de fecha y hora

- **dayofmonth(fecha):** devuelve el día del mes para la fecha dada, dentro del rango 1 a 31. Ejemplo: ***dayofmonth('2006-08-10');*** devuelve 10.
- **dayofweek(fecha):** devuelve el índice del día de semana para la fecha pasada como argumento. Los valores de los índices son: 1=domingo, 2=lunes,... 7=sábado). Ejemplo: ***dayofweek('2006-08-10');*** devuelve 5, o sea jueves.
- **dayofyear(fecha):** devuelve el día del año para la fecha dada, dentro del rango 1 a 366. Ejemplo: ***dayofyear('2006-08-10');*** devuelve 222.
- **extract(tipo from fecha):** extrae partes de una fecha. Ejemplos: ***extract(year from '2006-10-10');*** devuelve "2006". ***extract(year_month from '2006-10-10 10:15:25');*** devuelve "200610". ***extract(day_minute from '2006-10-10 10:15:25');*** devuelve "101015".
- Los valores para tipo pueden ser: second, minute, hour, day, month, year, minute_second, hour_minute, day_hour, year_month, hour_second (horas, minutos y segundos), day_minute (días, horas y minutos), day_second (días a segundos).
- **hour(hora):** devuelve la hora para el dato dado, en el rango de 0 a 23. Ejemplo: ***hour('18:25:09');*** devuelve "18"
- **minute(hora):** devuelve los minutos de la hora dada, en el rango de 0 a 59.
- **monthname(fecha):** devuelve el nombre del mes de la fecha dada. Ej.: ***monthname('2006-08-10');*** devuelve "August".
- **month(fecha):** devuelve el mes de la fecha dada, en el rango de 1 a 12.

Funciones para el uso de fecha y hora

- **now()** y **sysdate()**: devuelven la fecha y hora actuales.
- **period_add(p,n):** agrega "n" meses al periodo "p", en el formato "YYMM" o "YYYYMM"; devuelve un valor en el formato "YYYYMM". El argumento "p" no es una fecha, sino un año y un mes. Ejemplo: ***period_add('200608',2);*** devuelve "200610".
- **period_diff(p1,p2):** devuelve el número de meses entre los períodos "p1" y "p2", en el formato "YYMM" o "YYYYMM". Los argumentos de período no son fechas sino un año y un mes. Ejemplo: ***period_diff('200608','200602');*** devuelve 6.
- **second(hora):** devuelve los segundos para la hora dada, en el rango de 0 a 59.
- **sec_to_time(segundos):** devuelve el argumento "segundos" convertido a horas, minutos y segundos. Ejemplo: ***sec_to_time(90);*** devuelve "1:30".
- **timediff(hora1,hora2):** devuelve la cantidad de horas, minutos y segundos entre hora1 y hora2.
- **time_to_sec(hora):** devuelve el argumento "hora" convertido en segundos.
- **to_days(fecha):** devuelve el número de día (el número de día desde el año 0).
- **weekday(fecha):** devuelve el índice del día de la semana para la fecha pasada como argumento. Los índices son: 0=lunes, 1=martes,... 6=domingo). Ejemplo***: weekday('2006-08-10');*** devuelve 3, o sea jueves.
- **year(fecha):** devuelve el año de la fecha dada, en el rango de 1000 a 9999. Ejemplo: ***year('06-08-10');*** devuelve "2006".

Funciones de control de flujo (if)

Trabajamos con las tablas "libros" de una librería.

No nos interesa el precio exacto de cada libro, sino si el precio es menor o mayor a $50. Podemos utilizar estas sentencias:

select titulo from libros where precio<50;

select titulo from libros where precio >=50;

En la primera sentencia mostramos los libros con precio menor a 50 y en la segunda los demás. También podemos usar la función "if".

"if" es una función a la cual se le envían 3 argumentos: el segundo y tercer argumento corresponden a los valores que devolverá en caso que el primer argumento (una expresión de comparación) sea "verdadero" o "falso"; es decir, si el primer argumento es verdadero, devuelve el segundo argumento, sino devuelve el tercero. Ejemplo:

select titulo, if (precio>50,'caro','economico') from libros;

Si el precio del libro es mayor a 50 (primer argumento del "if"), coloca "caro" (segundo argumento del "if"), en caso contrario coloca "economico" (tercer argumento del "if").

Funciones de control de flujo (if)

Si queremos mostrar los nombres de los autores y la cantidad de libros de cada uno de ellos; especificamos el nombre del campo a mostrar (autor), contamos los libros con autor conocido con la función count() y agrupamos por nombre de autor:

select autor, count(*) from libros group by autor;

El resultado nos muestra cada autor y la cantidad de libros de cada uno de ellos. Si solamente queremos mostrar los autores que tienen más de 1 libro, es decir, la cantidad mayor a 1, podemos usar esta sentencia:

select autor, count(*) from libros group by autor having count(*)>1;

Pero si no queremos la cantidad exacta sino solamente saber si cada autor tiene más de 1 libro, podemos usar "if":

select autor, if (count(*)>1,'Más de 1','1') from libros group by autor;

Si la cantidad de libros de cada autor es mayor a 1 (1e argumento del "if"), coloca "Más de 1“ (2º argumento del "if"), en caso contrario coloca "1" (3e argumento del "if"). Queremos saber si la cantidad de libros por editorial supera los 4 o no:

select editorial, if (count(*)>4,'5 o más','menos de 5') as cantidad from libros group by editorial order by cantidad;

Si la cantidad de libros de cada editorial es mayor a 4 (1e argumento del "if"), coloca "5 o más“ (2º argumento del "if"), en caso contrario coloca "menos de 5" (3e argumento del "if").

Funciones de control de flujo (case)

La función "case" es similar a la función "if", sólo que se pueden establecer varias condiciones a cumplir.

Queremos saber si la cantidad de libros de cada editorial es menor o mayor a 1, usamos:

select editorial, if (count(*)>1,'Mas de 2','1') as 'cantidad' from libros group by editorial;

vemos los nombres de las editoriales y una columna "cantidad" que especifica si hay más o menos de uno. Podemos obtener la misma salida usando un "case":

select editorial, case count(*) when 1 then 1 else 'mas de 1' end as 'cantidad' from libros group by editorial;

Por cada valor hay un "when" y un "then"; si encuentra un valor coincidente en algún "where" ejecuta el "then" correspondiente a ese "where", si no encuentra ninguna coincidencia, se ejecuta el "else", si no hay parte "else" devuelve "null". Finalmente se coloca "end" para indicar que el "case" ha finalizado.

Entonces, la sintaxis es: case when then ... else … end Se puede obviar la parte "else":

select editorial, case count(*) when 1 then 1 end as 'cantidad' from libros group by editorial;

Funciones de control de flujo (case)

Con el "if" solamente podemos obtener dos salidas, cuando la condición resulta verdadera y cuando es falsa, si queremos más opciones podemos usar "case". Vamos a extender el "case" anterior para mostrar distintos mensajes:

```
select editorial, case count(*) when 1 then 1 when 2 then 2 when 3 then 3 else
'Más de 3' end as 'cantidad' from libros group by editorial;
```

Incluso podemos agregar una cláusula "order by" y ordenar la salida por la columna "cantidad":

```
select editorial, case count(*) when 1 then 1 when 2 then 2  when 3 then 3 else
'Más de 3' end as 'cantidad' from libros group by editorial order by cantidad;
```

La diferencia con "if" es que el "case" toma valores puntuales, no expresiones. La siguiente sentencia provocará un error:

```
select editorial, case count(*) when 1 then 1 when >1 then 'mas de 1' end as
'cantidad' from libros group by editorial;
```

Pero existe otra sintaxis de case que permite condiciones: case when then... else end

```
select editorial, case when count(*)=1 then 1 else 'mas de uno' end as cantidad
from libros group by editorial;
```

Alias

Un "alias" se usa como nombre de un campo o de una expresión o para referenciar una tabla cuando se utilizan más de una tabla (tema que veremos más adelante). Cuando usamos una función de agrupamiento, por ejemplo:

select count(*) from libros where autor like '%Borges%';

La columna en la salida tiene como encabezado "count(*)", para que el resultado sea más claro podemos utilizar un alias:

select count(*) as librosdeborges from libros where autor like '%Borges%';

La columna de la salida ahora tiene como encabezado el alias, lo que hace más comprensible el resultado.

Un alias puede tener hasta 255 caracteres, acepta todos los caracteres. La palabra clave "as" es opcional en algunos casos, pero es conveniente usarla. Si el alias consta de una sola cadena las comillas no son necesarias, pero si contiene más de una palabra, es necesario colocarla entre comillas. Se pueden utilizar alias en las clásulas "group by", "order by", "having". Por ejemplo:

select editorial as 'Nombre de editorial' from libros group by 'Nombre de editorial';

select editorial, count(*) as cantidad from libros group by editorial order by cantidad;

select editorial, count(*) as cantidad from libros group by editorial having cantidad>2;

No está permitido utilizar alias de campos en las cláusulas "where".

Cláusula limit del comando select

La cláusula "limit" se usa para restringir los registros que se devuelven en una consulta "select".

Recibe 1 ó 2 argumentos numéricos enteros positivos; el primero indica el número del primer registro a devolver, el segundo, el número máximo de registros a devolver. El número de registro inicial es 0 (no 1). Si el segundo argumento supera la cantidad de registros de la tabla, se limita hasta el último registro.

Ejemplo:

select * from libros limit 0,4; Muestra los primeros 4 registros, 0,1,2 y 3.

Si introducimos:

select * from libros limit 5,4; Recuperamos 4 registros, desde el 5 al 8.

Si se coloca un solo argumento, indica el máximo número de registros a devolver, comenzando desde 0.

Ejemplo:

select * from libros limit 8; Muestra los primeros 8 registros.

Cláusula limit del comando select

Si se coloca un solo argumento, indica el máximo número de registros a retornar, comenzando desde 0. Ejemplo:

select * from libros limit 8;

Muestra los primeros 8 registros.

Para recuperar los registros desde cierto número hasta el final, se puede colocar un número grande para el segundo argumento:

select * from libros limit 6,10000;

Recupera los registros 7 al último.

"limit" puede combinarse con el comando "delete". Si queremos eliminar 2 registros de la tabla "libros“ . Usamos:

delete from libros limit 2;

Podemos ordenar los registros por precio (por ejemplo) y borrar 2:

delete from libros order by precio limit 2;

Esta sentencia borrará los 2 primeros registros, es decir, los de precio más bajo.

Cláusula limit del comando select

Podemos emplear la cláusula "limit" para eliminar registros duplicados. Por ejemplo, continuamos con la tabla "libros" de una librería, ya hemos almacenado el libro "El aleph" de "Borges" de la editorial "Planeta", pero nos equivocamos y volvemos a ingresar el mismo libro, del mismo autor y editorial 2 veces más, es un error que no controla MySQL.

Para eliminar el libro duplicado y que sólo quede un registro de él vemos cuántos tenemos:

```
select * from libros where titulo='El aleph' and autor='Borges' and editorial='Planeta';
```

Luego eliminamos con "limit" la cantidad sobrante (tenemos 3 y queremos solo 1):

```
delete from libros where titulo='El aleph' and autor='Borges' and editorial='Planeta' limit 2;
```

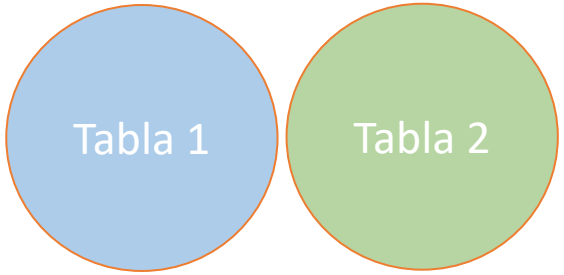
Un mensaje nos muestra la cantidad de registros eliminados.

Es decir, con "limit" indicamos la cantidad a eliminar. Veamos cuántos hay ahora:

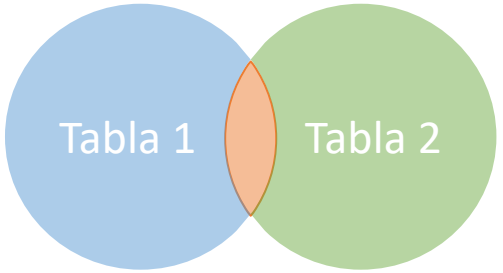
```
select * from libros where titulo='El aleph" and autor='Borges' and editorial='Planeta';
```

Sólo queda 1.

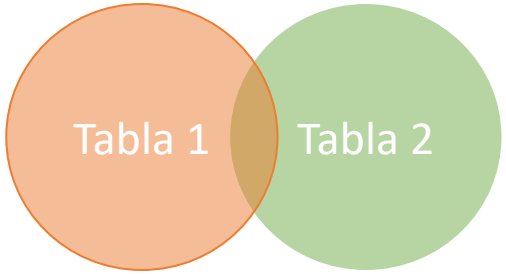
La operación JOIN



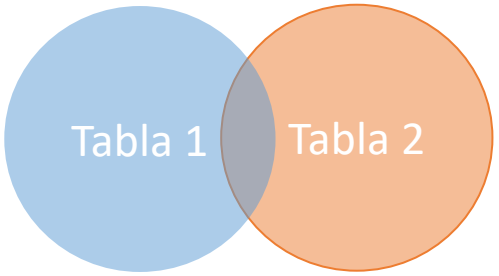

```
SELECT * FROM Tabla1;
SELECT * FROM Tabla2;
```



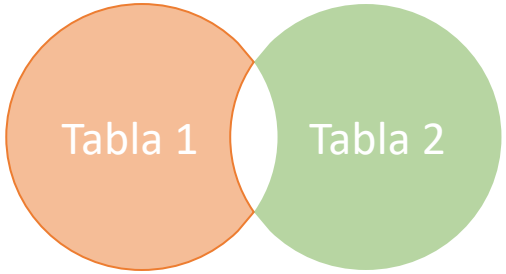

```
SELECT * FROM Tabla1 t1
INNER JOIN Table2 t2
ON t1.fk = t2.id;
```



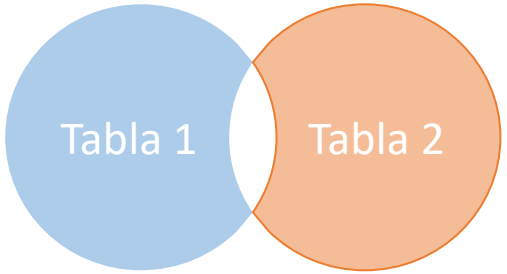

```
SELECT * FROM Tabla1 t1
LEFT JOIN Table2 t2
ON t1.fk = t2.id;
```




```
SELECT * FROM Tabla1 t1
RIGHT JOIN Table2 t2
ON t1.fk = t2.id;
```



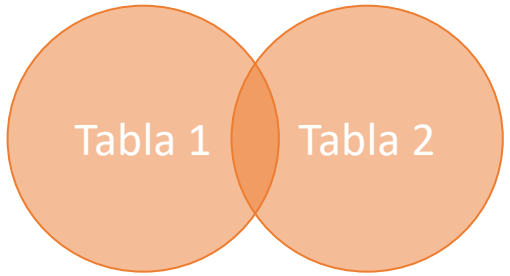

```
SELECT * FROM Tabla1 t1
LEFT JOIN Table2 t2
ON t1.fk = t2.id
where t2.id is null;
```



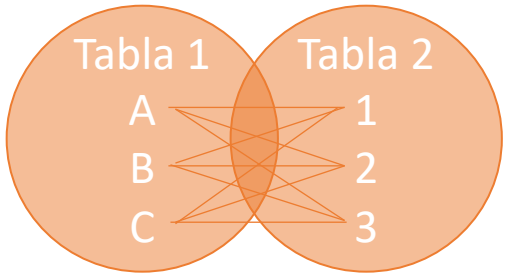

```
SELECT * FROM Tabla1 t1
RIGHT JOIN Table2 t2
ON t1.fk = t2.id
where t2.id is null;
```

La operación JOIN

FULL OUTER JOIN

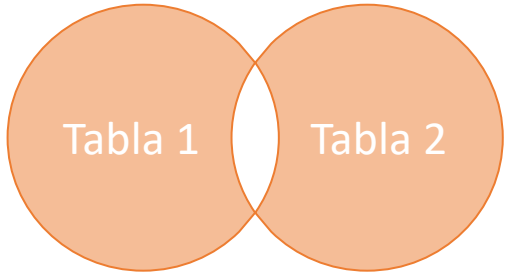



```
SELECT * FROM Tabla1 t1
LEFT JOIN Table2 t2
ON t1.fk = t2.id
UNION
SELECT * FROM Tabla1 t1
RIGHT JOIN Table2 t2
ON t1.fk = t2.id;
```

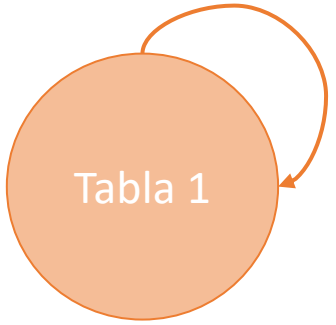



```
SELECT * FROM Tabla1 t1
CROSS JOIN Table2 t2
ON t1.fk = t2.id;
```

FULL OUTER JOIN with exclusion

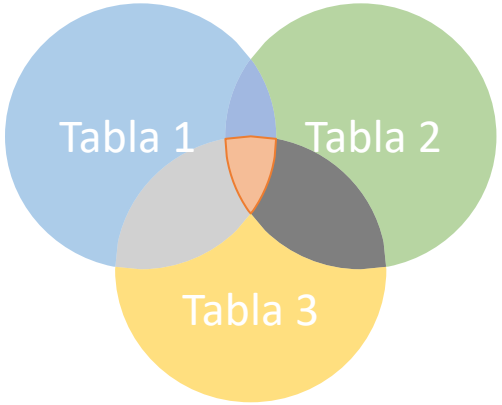



```
SELECT * FROM Tabla1 t1
LEFT JOIN Table2 t2
ON t1.fk = t2.id
where t2.id IS NOT NULL
UNION
SELECT * FROM Tabla1 t1
RIGHT JOIN Table2 t2
ON t1.fk = t2.id
where t1.id IS NOT NULL;
```

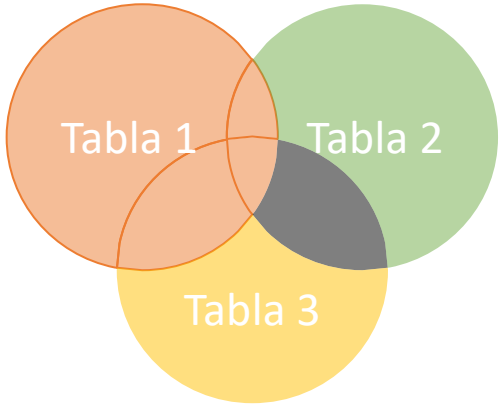



```
SELECT * FROM Tabla1 t1a
SELF JOIN Table1 t1b
ON t1a.fk = t1b.id;
```

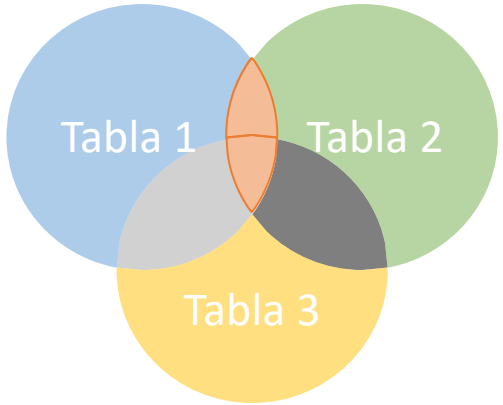
La operación JOIN




```
SELECT * FROM Tabla1 t1
JOIN Table2 t2
ON t1.fk_t2 = t2.id
JOIN Table3 t3
ON t1.fk_t3 = t3.id;
```




```
SELECT * FROM Tabla1 t1
LEFT JOIN Table2 t2
ON t1.fk_t2 = t2.id
LEFT JOIN Table3 t3
ON t1.fk_t3 = t3.id;
```




```
SELECT * FROM Tabla1 t1
JOIN Table2 t2
ON t1.fk_t2 = t2.id
LEFT JOIN Table3 t3
ON t1.fk_t3 = t3.id;
```

Varias tablas (join)

Hasta ahora hemos trabajado con una sola tabla, pero en general, se trabaja con varias tablas. Para evitar la repetición de datos y ocupar menos espacio, se separa la información en varias tablas. Cada tabla tendrá parte de la información total que queremos registrar.

Por ejemplo, los datos de nuestra tabla "libros" podrían separarse en 2 tablas, una "libros" y otra "editoriales" que guardará la información de las editoriales. En nuestra tabla "libros" haremos referencia a la editorial colocando un código que la identifique.

```
create table libros(
codigo int unsigned auto_increment,
titulo varchar(40) not null,
autor varchar(30) not null default 'Desconocido',
codigoeditorial tinyint unsigned not null,
precio decimal(5,2) unsigned,
cantidad smallint unsigned default 0,
primary key (codigo) );
```

```
create table editoriales(
codigo tinyint unsigned auto_increment,
nombre varchar(20) not null,
primary key(codigo) );
```

Varias tablas (join)

De este modo, evitamos almacenar tantas veces los nombres de las editoriales en la tabla "libros" y guardamos el nombre en la tabla "editoriales"; para indicar la editorial de cada libro agregamos un campo referente al código de la editorial en la tabla "libros" y en "editoriales".

Al recuperar los datos de los libros: ***select * from libros;***

vemos que en el campo "editorial" aparece el código, pero no sabemos el nombre de la editorial. Para obtener los datos de cada libro, incluyendo el nombre de la editorial, necesitamos consultar ambas tablas, traer información de las dos.

Cuando obtenemos información de más de una tabla decimos que hacemos un "join" (unión). Veamos un ejemplo:

select * from libros join editoriales on libros.codigoeditorial=editoriales.codigo;

Ahora, analicemos la consulta anterior.

Indicamos el nombre de la tabla luego del "from" ("libros"), unimos esa tabla con "join" y el nombre de la otra tabla ("editoriales"), luego especificamos la condición para enlazarlas con "on", es decir, el campo por el cual se combinarán. "on" hace coincidir registros de las dos tablas basándose en el valor de algún campo, en este ejemplo, los códigos de las editoriales de ambas tablas, el campo "codigoeditorial" de "libros" y el campo "codigo" de "editoriales" son los que enlazarán ambas tablas.

Varias tablas (join)

Cuando se combina (join, unión) información de varias tablas, es necesario indicar qué registro de una tabla se combinará con qué registro de la otra tabla.

Si no especificamos por qué campo relacionamos ambas tablas, por ejemplo:

select * from libros join editoriales;

el resultado es el producto cartesiano de ambas tablas (cada registro de la primera tabla se combina con cada registro de la segunda tabla), un "join" sin condición "on" genera un resultado en el que aparecen todas las combinaciones de los registros de ambas tablas. La información no sirve. Note que en la consulta:

select * from libros join editoriales on libros.codigoeditorial=editoriales.codigo;

al nombrar el campo usamos el nombre de la tabla también. Cuando las tablas referenciadas tienen campos con igual nombre, esto es necesario para evitar confusiones y ambigüedades al momento de referenciar un campo. En este ejemplo, si no especificamos "editoriales.codigo" y solamente usamos "codigo", MySQL no sabrá si nos referimos al campo "codigo" de "libros" o de "editoriales".

Varias tablas (join)

Si omitimos la referencia a las tablas al nombrar el campo "codigo" (nombre de campo que contienen ambas tablas):

select * from libros join editoriales on codigoeditorial=codigo;

aparece un mensaje de error indicando que "codigo" es ambiguo.

Entonces, si en las tablas, los campos tienen el mismo nombre, debemos especificar a cuál tabla pertenece el campo al hacer referencia a él, para ello se antepone el nombre de la tabla al nombre del campo, separado por un punto (.).

Entonces, se nombra la primer tabla, se coloca "join" junto al nombre de la segunda tabla de la cual obtendremos información y se asocian los registros de ambas tablas usando un "on" que haga coincidir los valores de un campo en común en ambas tablas, que será el enlace.

Para simplificar la sentencia podemos usar un alias para cada tabla:

select * from libros as l join editoriales as e on l.codigoeditorial=e.codigo;

Cada tabla tiene un alias y se referencian los campos usando el alias correspondiente. En este ejemplo, el uso de alias es para fines de simplificación, pero en algunas consultas es absolutamente necesario.

Varias tablas (join)

En la consulta anterior vemos que el código de la editorial aparece 2 veces, desde la tabla "libros" y "editoriales". Podemos solicitar que nos muestre algunos campos:

```
select titulo,autor,nombre from libros as l join editoriales as e on
l.codigoeditorial=e.codigo;
```

Al presentar los campos, en este caso, no es necesario aclarar a qué tabla pertenecen porque los campos solicitados no se repiten en ambas tablas, pero si solicitáramos el código del libro, debemos especificar de qué tabla porque el campo "codigo" se repite en ambas tablas ("libros" y "editoriales"):

```
select l.codigo,titulo,autor,nombre from libros as l join editoriales as e on
l.codigoeditorial=e.codigo;
```

Si obviamos la referencia a la tabla, la sentencia no se ejecuta y aparece un mensaje indicando que el campo "codigo" es ambiguo.

Varias tablas (left join)

Hemos visto cómo usar registros de una tabla para encontrar registros de otra tabla, uniendo ambas tablas con "join" y enlazándolas con una condición "on" en la cual colocamos el campo en común. O sea, hacemos un "join" y asociamos registros de 2 tablas usando el "on", buscando coincidencia en los valores del campo que tienen en común ambas tablas.

- Trabajamos con las tablas de una librería:

  -libros: codigo (clave primaria), titulo, autor, codigoeditorial, precio, cantidad

  -editoriales: codigo (clave primaria), nombre.

Queremos saber de qué editoriales no tenemos libros. Para averiguar qué registros de una tabla no se encuentran en otra tabla necesitamos usar un “join” diferente. Necesitamos determinar qué registros no tienen correspondencia en otra tabla, cuáles valores de la primera tabla (de la izquierda) no están en la segunda (de la derecha). Para obtener la lista de editoriales y sus libros, incluso de aquellas editoriales de las cuales no tenemos libros usamos:

```
select * from editoriales left join libros on editoriales.codigo=libros.codigoeditorial;
```

Un "left join" se usa para hacer coincidir registros en una tabla (izquierda) con otra tabla (derecha), pero, si un valor de la tabla de la izquierda no encuentra coincidencia en la tabla de la derecha, se genera una fila extra (una por cada valor no encontrado) con todos los campos seteados a "null".

Varias tablas (left join)

Entonces, la sintaxis es la siguiente: se nombran ambas tablas, una a la izquierda del "join" y la otra a la derecha, y la condición para enlazarlas, es decir, el campo por el cual se combinarán, se establece luego de "on". Es importante la posición en que se colocan las tablas en un "left join", la tabla de la izquierda es la que se usa para localizar registros en la tabla de la derecha. Por lo tanto, estos "join" no son iguales:

select * from editoriales left join libros on editoriales.codigo=libros.codigoeditorial;

select * from libros left join editoriales on editoriales.codigo=libros.codigoeditorial;

- La primera sentencia opera así: por cada valor de codigo de "editoriales" busca coincidencia en la tabla "libros", si no encuentra coincidencia para algún valor, genera una fila seteada a "null".
- La segunda sentencia opera de modo inverso: por cada valor de "codigoeditorial" de "libros" busca coincidencia en la tabla "editoriales", si no encuentra coincidencia, setea la fila a "null". Usando registros de la tabla de la izquierda se encuentran registros en la tabla de la derecha.
- Luego del "on" se especifican los campos que se asociarán; no se deben colocar condiciones en la parte "on" para restringir registros que deberían estar en el resultado, para ello hay que usar la cláusula "where".

Varias tablas (left join)

Un "left join" puede tener clausula "where" que restringa el resultado de la consulta considerando solamente los registros que encuentran coincidencia en la tabla de la derecha:

```
select e.nombre,l.titulo from editoriales as e left join libros as l on
e.codigo=l.codigoeditorial where l.codigoeditorial is not null;
```

El anterior "left join" muestra los valores de la tabla "editoriales" que están presentes en la tabla de la derecha ("libros").

También podemos mostrar las editoriales que no están presentes en "libros":

```
select e.nombre,l.titulo from editoriales as e left join libros as l on
e.codigo=l.codigoeditorial where l.codigoeditorial is null;
```

El anterior "left join" muestra los valores de la tabla "editoriales" que no encuentran correspondencia en la tabla de la derecha, "libros".

Varias tablas (right join)

"right join" opera del mismo modo que "left join" sólo que la búsqueda de coincidencias la realiza de modo inverso, es decir, los roles de las tablas se invierten, busca coincidencia de valores desde la tabla de la derecha en la tabla de la izquierda y si un valor de la tabla de la derecha no encuentra coincidencia en la tabla de la izquierda, se genera una fila extra (una por cada valor no encontrado) con todos los campos seteados a "null".

- Trabajamos con las tablas de una librería:
  - libros: codigo (clave primaria), titulo, autor, codigoeditorial, precio, cantidad
  - editoriales: codigo (clave primaria), nombre.

Estas sentencias devuelven el mismo resultado:

```
select nombre,titulo from editoriales as e  left join libros as l on e.codigo=l.codigoeditorial;
select nombre,titulo from libros as l right join editoriales as e on e.codigo=l.codigoeditorial;
```

La primera busca valores de "codigo" de la tabla "editoriales" (tabla de la izquierda) coincidentes con los valores de "codigoeditorial" de la tabla "libros" (tabla de la derecha). La segunda busca valores de la tabla de la derecha coincidentes con los valores de la tabla de la izquierda.

Varias tablas (cross join)

"cross join" devuelve todos los registros de todas las tablas implicadas en la unión, devuelve el producto cartesiano. No es muy utilizado.

Un pequeño restaurante tiene almacenados los nombres y precios de sus comidas en una tabla llamada "comidas" y en una tabla denominada "postres" los mismos datos de sus postres. El restaurante quiere combinar los registros de ambas tablas para mostrar los distintos menús que ofrece. Podemos usar "cross join":

```
select c.*,p.* from comidas as c cross join postres as p;
```

es igual a un simple "join" sin parte "on":

```
select c.*,p.* from comidas as c join postres as p;
```

Podemos organizar la salida del "cross join" para obtener el nombre del plato principal, del postre y el precio total de cada combinación (menú):

```
select c.nombre,p.nombre, c.precio+p.precio as total from comidas as c cross join postres as p;
```

Para realizar un "join" no es necesario utilizar 2 tablas, podemos combinar los registros de una misma tabla. Para ello debemos utilizar 2 alias para la tabla.

Varias tablas (cross join)

Si los datos de las tablas anteriores ("comidas" y "postres") estuvieran en una sola tabla con la siguiente estructura:

```
create table comidas(
codigo tinyint unsigned auto_increment,
nombre varchar(30),
rubro varchar(20), # plato principal y postre
precio decimal (5,2) unsigned,
primary key(codigo) );
```

Podemos obtener la combinación de platos principales con postres empleando un "cross join" con una sola tabla:

```
select c1.nombre,c1.precio,c2.nombre,c2.precio from comidas as c1 cross join comidas
as c2 where c1.rubro='plato principal' and c2.rubro='postre';
```

Se empleó un "where" para combinar "plato principal" con "postre". Si queremos el total de cada combinación:

```
select c1.nombre,c2.nombre, c1.precio+c2.precio as total from comidas as c1 cross join
comidas as c2 where c1.rubro='plato principal' and c2.rubro='postre';
```

Varias tablas (natural join)

"natural join" se usa cuando los campos por los cuales se enlazan las tablas tienen el mismo nombre.

- Trabajamos con las tablas de una librería:
  - libros: codigo (clave primaria), titulo, autor, codigoeditorial, precio, cantidad
  - editoriales: codigo (clave primaria), nombre.

Como en ambas tablas, el código de la editorial se denomina "codigoeditorial", podemos omitir la parte "on" que indica los nombres de los campos por el cual se enlazan las tablas, empleando "natural join", se unirán por el campo que tienen en común:

```
select titulo,nombre from libros as l natural join editoriales as e;
```

La siguiente sentencia tiene la misma salida anterior:

```
select titulo,nombre from libros as l join editoriales as e on l.codigoeditorial=e.codigoeditorial;
```

También se puede usar "natural" con "left join" y "right join":

```
select nombre,titulo from editoriales as e natural left join libros as l;
```

que tiene la misma salida que:

```
select nombre,titulo from editoriales as e left join libros as l on e.codigoeditorial=l.codigoeditorial;
```

Varias tablas (natural join)

Es decir, con "natural join" no se coloca la parte "on" que especifica los campos por los cuales se enlazan las tablas, porque MySQL busca los campos con igual nombre y enlaza las tablas por ese campo.

Hay que tener cuidado con este tipo de "join" porque si ambas tablas tiene más de un campo con igual nombre, MySQL no sabrá por cual debe realizar la unión. Por ejemplo, si el campo "titulo" de la tabla "libros" se llamara "nombre", las tablas tendrían 2 campos con igual nombre ("codigoeditorial" y "nombre").

Otro problema que puede surgir es el siguiente. Tenemos la tabla "libros" con los siguientes campos:

codigo (del libro), titulo, autor y codigoeditorial, y la tabla "editoriales" con estos campos: codigo (de la editorial) y nombre. Si usamos "natural join", unirá las tablas por el campo "codigo", que es el campo que tienen igual nombre, pero el campo "codigo" de "libros" no hace referencia al código de la editorial sino al del libro, así que la salida será errónea.

Varias tablas (inner join - straight join)

Existen otros tipos de "join" además del simple "join", "left join", "right join", "cross join" y "natural join". Veámoslos.

"inner join" es igual que "join". Con "inner join", todos los registros no coincidentes son descartados, sólo los coincidentes se muestran en el resultado:

```
select nombre,titulo from editoriales as e inner join libros as l on e.codigo=l.codigoeditorial;
```

Tiene la misma salida que un simple "join":

```
select nombre,titulo from editoriales as e join libros as l on e.codigo=l.codigoeditorial;
```

"straight join" es igual a "join", sólo que la tabla de la izquierda es leída siempre antes que la de la derecha.

join, group by y funciones de agrupación

Para ver todas las editoriales, agrupadas por nombre, con una columna llamada "Cantidad de libros" en la que aparece la cantidad calculada con "count()" de todos los libros de cada editorial, introducimos:

```
select e.nombre, count(l.codigoeditorial) as 'Cantidad de libros' from editoriales as e left join libros as l on l.codigoeditorial=e.código group by e.nombre;
```

Si usamos "left join" la consulta mostrará todas las editoriales, y para cualquier editorial que no encontrara coincidencia en la tabla "libros" colocará "0" en "Cantidad de libros".

Si usamos "join" en lugar de "left join":

```
select e.nombre,count(l.codigoeditorial) as 'Cantidad de libros' from editoriales as e join libros as l on l.codigoeditorial=e.código group by e.nombre;
```

Solamente mostrará las editoriales para las cuales encuentra valores coincidentes para el código de la editorial en la tabla "libros". Para conocer el mayor precio de los libros de cada editorial usamos la función "max()", hacemos una unión y agrupamos por nombre de la editorial:

```
select e.nombre, max(l.precio) as 'Mayor precio' from editoriales as e left join libros as l on l.codigoeditorial=e.código group by e.nombre;
```

En la sentencia anterior, mostrará, para la editorial de la cual no haya libros, el valor "null" en la columna calculada; si realizamos un simple "join":

```
select e.nombre, max(l.precio) as 'Mayor precio' from editoriales as e join libros as l on l.codigoeditorial=e.código group by e.nombre;
```

Sólo mostrará las editoriales para las cuales encuentra correspondencia en la tabla de la derecha.

Función de control if con varias tablas

Podemos emplear "if" y "case" en la misma sentencia que usamos un "join".

Por ejemplo, tenemos las tablas "libros" y "editoriales" y queremos saber si hay libros de cada una de las editoriales:

```
select e.nombre, if (count(l.codigoeditorial)>0,'Si','No') as hay from editoriales as e
left join libros as l on e.codigo=l.codigoeditorial group by e.nombre;
```

Podemos obtener una salida similar usando "case" en lugar de "if":

```
select e.nombre, case count(l.codigoeditorial) when 0 then 'No' else 'Si' end as
'Hay' from editoriales as e left join libros as l on e.codigo=l.codigoeditorial group
by e.nombre;
```

Join con más de dos tablas

Podemos hacer un "join" con más de dos tablas.

Una biblioteca registra la información de sus libros en una tabla llamada "libros", los datos de sus socios en "socios" y los préstamos en una tabla "prestamos".

En la tabla "prestamos" haremos referencia al libro y al socio que lo solicita colocando un código que los identifique. Veamos:

```
create table libros(
codigo int unsigned
auto_increment,
titulo varchar(40) not null,
autor varchar(20) default
'Desconocido',
primary key (codigo) );
```

```
create socios(
documento char(8) not null,
nombre varchar(30),
domicilio varchar(30),
primary key (numero) );
```

```
create table prestamos(
documento char(8) not null,
codigolibro int unsigned,
fechaprestamo date not null,
fechadevolucion date,
primary key
(codigolibro,fechaprestamo) );
```

Join con más de dos tablas

Al recuperar los datos de los prestamos:

```
select * from prestamos;
```

aparece el código del libro pero no sabemos el nombre y tampoco el nombre del socio sino su documento. Para obtener los datos completos de cada préstamo, incluyendo esos datos, necesitamos consultar las tres tablas.

Hacemos un "join" (unión):

```
select nombre,titulo,fechaprestamo from prestamos as p join socios as s on
s.documento=p.documento join libros as l on codigolibro=codigo;
```

Analicemos la consulta anterior. Indicamos el nombre de la tabla luego del "from" ("prestamos"), unimos esa tabla con la tabla "socios" especificando con "on" el campo por el cual se combinarán: el campo "documento" de ambas tablas; luego debemos hacer coincidir los valores para la unión con la tabla "libros" enlazándolas por los campos "codigolibro" y "codigo" de "libros". Utilizamos alias para una sentencia más sencilla y comprensible.

Join con más de dos tablas

Especificamos a qué tabla pertenece el campos "documento" porque a ese nombre de campo lo tienen las tablas "prestamos" y "socios", esto es necesario para evitar confusiones y ambigüedades al momento de referenciar un campo. En este ejemplo, si omitimos la referencia a las tablas al nombrar el campo "documento" aparece un mensaje de error indicando que "documento" es ambiguo.

Para ver todos los prestamos, incluso los que no encuentran coincidencia en las otras tablas, usamos:

select nombre,titulo,fechaprestamo from prestamos as p left join socios as s on p.documento=s.documento left join libros as l on l.codigo=p.codigolibro;

Podemos ver aquellos prestamos con valor coincidente para "libros" pero para "socio" con y sin coincidencia:

select nombre,titulo,fechaprestamo from prestamos as p left join socios as s on p.documento=s.documento join libros as l on p.codigolibro=l.codigo;

Subconsultas

Una subconsulta es una consulta anidada dentro de otra consulta.

Debe tener en cuenta que no existe una única solución para resolver una consulta en SQL. En esta unidad vamos

a estudiar cómo podemos resolver haciendo uso de subconsultas, algunas de las consultas que hemos resuelto

en las unidades anteriores.

Tipos de subconsultas

El estándar SQL define tres tipos de subconsultas:

- **Subconsultas de fila**. Son aquellas que devuelven más de una columna pero una única fila.
- **Subconsultas de tabla**. Son aquellas que devuelve una o varias columnas y cero o varias filas.
- **Subconsultas escalares**. Son aquellas que devuelven una columna y una fila.

Subconsultas en la cláusula WHERE

Por ejemplo, suponga que queremos conocer el nombre del producto que tiene el mayor precio. En este caso podríamos realizar una primera consulta para buscar cuál es el valor del precio máximo y otra segunda consulta para buscar el nombre del producto cuyo precio coincide con el valor del precio máximo. La consulta sería la siguiente:

```
SELECT nombre FROM producto
WHERE precio = (SELECT MAX(precio) FROM producto)
```

En este caso sólo hay un nivel de anidamiento entre consultas pero pueden existir varios niveles de anidamiento.

Subconsultas en la cláusula HAVING

Ejemplo: Devuelve un listado con todos los nombres de los fabricantes que tienen el mismo número de productos que el fabricante Asus.

```
SELECT fabricante.nombre, COUNT(producto.codigo)
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
GROUP BY fabricante.codigo
HAVING COUNT(producto.codigo) >= (
SELECT COUNT(producto.codigo)
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
WHERE fabricante.nombre = 'Asus');
```

Subconsultas en la cláusula FROM

Ejemplo: Devuelve una listado de todos los productos que tienen un precio mayor o igual al precio medio de todos los productos de su mismo fabricante.

```
SELECT * FROM producto INNER JOIN (
SELECT codigo_fabricante, AVG(precio) AS media
FROM producto
GROUP BY codigo_fabricante) AS t
ON producto.codigo_fabricante = t.codigo_fabricante
WHERE producto.precio >= t.media;
```

Subconsultas en la cláusula SELECT

Las subconsultas que pueden aparecer en la cláusula SELECT tienen que ser subconsultas de tipo escalar, que devuelven una única fila y columna.

```
SELECT hoteles.etiqueta, numhabitacion FROM Habitaciones
INNER JOIN hoteles ON hoteles.idhotel =habitaciones.hotel
where tipohabitacion = (SELECT idtipohabitacion FROM tiposhabitacion
where descripcion = '1 cama individual con ducha');
```

Operadores que podemos usar en las subconsultas

Los operadores que podemos usar en las subconsultas son los siguientes:

- Operadores básicos de comparación (>,>=, <, <=, !=, <>, =).
- Predicados ALL y ANY.
- Predicado IN y NOT IN.
- Predicado EXISTS y NOT EXISTS.

Operadores básicos de comparación

Los operadores básicos de comparación (>,>=, <, <=, !=, <>, =) se pueden usar cuando queremos comparar una expresión con el valor que devuelve una subconsulta.

Los operadores básicos de comparación los vamos a utilizar para realizar comparaciones con subconsultas que devuelven un único valor, es decir, una columna y una fila.

Ejemplo: Devuelve todos los productos de la base de datos que tienen un precio mayor o igual al producto más caro del fabricante Asus.

```
SELECT * FROM producto
WHERE precio >= (SELECT MAX(precio)
FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante
WHERE fabricante.nombre = 'Asus');
```

Subconsultas con ALL y ANY

ALL y ANY se utilizan con los operadores de comparación (>,>=, <, <=, !=, <>, =) y nos permiten comparar una expresión con el conjunto de valores que devuelve una subconsulta.

ALL y ANY los vamos a utilizar para realizar comparaciones con subconsultas que pueden devolver varios valores, es decir, una columna y varias filas.

Ejemplo: Podemos escribir la consulta que devuelve todos los productos de la base de datos que tienen un precio mayor o igual al producto más caro del fabricante Asus, haciendo uso de ALL. Por lo tanto estas dos consultas darían el mismo resultado.

```
SELECT * FROM fabricante INNER JOIN producto  ON fabricante.codigo = producto.codigo_fabricante
WHERE precio >= (SELECT MAX(precio)  FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante  WHERE fabricante.nombre = 'Asus');
```

```
SELECT * FROM fabricante INNER JOIN producto ON fabricante.codigo = producto.codigo_fabricante
WHERE precio >= ALL (SELECT precio FROM fabricante INNER JOIN producto
ON fabricante.codigo = producto.codigo_fabricante WHERE fabricante.nombre = 'Asus');
```

Subconsultas con ALL y ANY

La palabra reservada SOME es un alias de ANY. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 <> ANY (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
```

Subconsultas con IN y NOT IN

IN y NOT IN nos permiten comprobar si un valor está o no incluido en un conjunto de valores, que puede ser el conjunto de valores que devuelve una subconsulta.

IN y NOT IN los vamos a utilizar para realizar comparaciones con subconsultas que pueden devolver varios valores, es decir, una columna y varias filas.

Ejemplo: Devuelve un listado de los clientes que no han realizado ningún pedido.

```
SELECT * FROM cliente WHERE id NOT IN (SELECT id_cliente FROM pedido);
```

Cuando estamos trabajando con subconsultas, IN y = ANY realizan la misma función. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
```

```
SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2);
```

Ocurre lo mismo con NOT IN y <> ALL. Por lo tanto, las siguientes consultas devolverían el mismo resultado:

```
SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2);
```

```
SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2);
```

Subconsultas con IN y NOT IN

Importante:

Tenga en cuenta que cuando hay un valor NULL en el resultado de la consulta interna, la consulta externa no devuelve ningún valor.

Ejemplo: Devuelve un listado con el nombre de los departamentos que no tienen empleados asociados.

```
SELECT nombre FROM departamento WHERE codigo NOT IN (
SELECT codigo_departamento FROM empleado);
```

La consulta interna SELECT codigo_departamento FROM empleado, devuelve algunas filas con valores NULL y por lo tanto la consulta externa no devuelve ningún valor. La forma de solucionarlo sería quitando los valores NULL de la consulta interna:

```
SELECT nombre FROM departamento WHERE codigo NOT IN (
SELECT codigo_departamento FROM empleado
WHERE codigo_departamento IS NOT NULL);
```

Subconsultas con EXISTS y NOT EXISTS

Ejemplo: Devuelve un listado de los clientes que no han realizado ningún pedido.

```
SELECT *  FROM cliente WHERE NOT EXISTS (
SELECT id_cliente FROM pedido WHERE cliente.id = pedido.id_cliente);
```

Índice de una tabla

Para facilitar la obtención de información de una tabla se utilizan índices.

- El índice de una tabla desempeña la misma función que el índice de un libro: permite encontrar datos rápidamente; en el caso de las tablas, localiza registros.
- Una tabla se indexa por un campo (o varios).
- El índice es un tipo de archivo con 2 entradas: un dato (un valor de algún campo de la tabla) y un puntero.
- Un índice posibilita el acceso directo y rápido haciendo más eficiente las búsquedas. Sin índice, se debe recorrer secuencialmente toda la tabla para encontrar un registro.
- El objetivo de un índice es acelerar la recuperación de información.
- La desventaja es que consume espacio en el disco.
- La indexación es una técnica que optimiza el acceso a los datos, mejora el rendimiento acelerando las consultas y otras operaciones. Es útil cuando la tabla contiene miles de registros.
- Los índices se usan para varias operaciones:
  - para buscar registros rápidamente.
  - para recuperar registros de otras tablas empleando "join".
- Es importante identificar el o los campos por los que sería útil crear un índice, aquellos campos por los cuales se realizan operaciones de búsqueda con frecuencia.

Índice de una tabla

Hay distintos tipos de índices, a saber:

1) "primary key": es el que definimos como clave primaria. Los valores indexados deben ser únicos y además no pueden ser nulos. MySQL le da el nombre "PRIMARY". Una tabla solamente puede tener una clave primaria.

2) "index": crea un índice común, los valores no necesariamente son únicos y aceptan valores "null". Podemos darle un nombre, si no se lo damos, se coloca uno por defecto. "key" es sinónimo de "index". Puede haber varios por tabla.

3) "unique": crea un índice para los cuales los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla. Podemos darle un nombre, si no se lo damos, se coloca uno por defecto.

- Todos los índices pueden ser multicolumna, es decir, pueden estar formados por más de 1 campo. En las siguientes lecciones aprenderemos sobre cada uno de ellos.
- Una tabla puede tener hasta 64 índices. Los nombres de índices aceptan todos los caracteres y pueden tener una longitud máxima de 64 caracteres. Pueden comenzar con un dígito, pero no pueden tener sólo dígitos.
- Una tabla puede ser indexada por campos de tipo numérico o de tipo carácter. También se puede indexar por un campo que contenga valores NULL, excepto los PRIMARY.
- "show index" muestra información sobre los índices de una tabla. Por ejemplo:

show index from libros;

Índice de tipo primary

El índice llamado primary se crea automáticamente cuando establecemos un campo como clave primaria, no podemos crearlo directamente.

- El campo por el cual se indexa puede ser de tipo numérico o de tipo caracter.
- Los valores indexados deben ser únicos y además no pueden ser nulos. Una tabla solamente puede tener una clave primaria por lo tanto, solamente tiene un índice PRIMARY.
- Puede ser multicolumna, es decir, pueden estar formados por más de 1 campo.

Veamos un ejemplo definiendo la tabla "libros" con una clave primaria:

```
create table libros(
codigo int unsigned auto_increment not null,
titulo varchar(40) not null,
autor varchar(30),
editorial varchar(15),
primary key(codigo) );
```

Podemos ver la estructura de los índices de una tabla con "show index". Por ejemplo:

```
show index from libros;
```

Aparece el índice PRIMARY creado automáticamente al definir el campo "codigo" como clave primaria.

Índice común (index)

Dijimos que hay 3 tipos de índices. Hasta ahora solamente conocemos la clave primaria que definimos al momento de crear una tabla.

Vamos a ver el otro tipo de índice, común. Un índice común se crea con "index", los valores no necesariamente son únicos y aceptan valores "null". Puede haber varios por tabla.

Usamos la tabla "libros". Un campo por el cual realizamos consultas frecuentemente es "editorial", indexar la tabla por ese campo será útil. Creamos el índice cuando creamos la tabla:

```
create table libros(
codigo int unsigned auto_increment,
titulo varchar(40) not null,
autor varchar(30),
editorial varchar(15),
primary key(codigo),
index i_editorial (editorial) );
```

Luego de la definición de los campos colocamos "index" seguido del nombre que le damos y entre paréntesis el o los campos por los cuales se indexará dicho índice.

Índice común (index)

"show index" muestra la estructura de los índices: ***show index from libros;***

Si no le asignamos un nombre a un índice, por defecto tomará el nombre del primer campo que forma parte del índice, con un sufijo opcional (_2,_3,...) para que sea único. Ciertas tablas (MyISAM, InnoDB y BDB) soportan índices en campos que permiten valores nulos, otras no, debiendo definirse el campo como "not null".

Se pueden crear índices por varios campos:

```
create table libros(
codigo int unsigned auto_increment,
titulo varchar(40) not null,
autor varchar(30),
editorial varchar(15),
index i_tituloeditorial (titulo,editorial) );
```

Para crear índices por múltiple campos se listan los campos dentro de los paréntesis separados con comas. Los valores de los índices se crean concatenando los valores de los campos mencionados.

Índice único (unique)

Veamos el otro tipo de índice, único. Un índice único se crea con "unique", los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla. Podemos darle un nombre, si no se lo damos, se coloca uno por defecto.

Vamos a trabajar con nuestra tabla "libros".

Crearemos dos índices únicos, uno por un solo campo y otro multicolumna:

```
create table libros(
codigo int unsigned auto_increment,
titulo varchar(40) not null,
autor varchar(30),
editorial varchar(15),
unique i_codigo(codigo),
unique i_tituloeditorial (titulo,editorial) );
```

Luego de la definición de los campos colocamos "unique" seguido del nombre que le damos y entre paréntesis el o los campos por los cuales se indexará dicho índice.

Resumen tipos de índices

"show index" muestra la estructura de los índices:

show index from libros;

RESUMEN: Hay 3 tipos de índices con las siguientes características:

Tipo	Nombre	Palabra Clave	Valores únicos	Acepta null	Cantidad por tabla
Clave primaria	PRIMARY	no	sí	no	1
común	Darlo o por defecto	INDEX o KEY	no	sí	varios
único	Darlo o por defecto	UNIQUE	sí	sí	varios

Borrar índice (drop index)

Para eliminar un índice usamos "drop index". Ejemplo:

drop index i_editorial on libros;

drop index i_tituloeditorial on libros;

Se elimina el índice con "drop index" seguido de su nombre y "on" seguido del nombre de la tabla a la cual pertenece.

Podemos eliminar los índices creados con "index" y con "unique" pero no el que se crea al definir una clave primaria. Un índice PRIMARY se elimina automáticamente al eliminar la clave primaria (tema que veremos más adelante).

Creación de índices a tablas existentes (create index)

Podemos agregar un índice a una tabla existente.

Para agregar un índice común a la tabla "libros" usamos:

create index i_editorial on libros (editorial);

Entonces, para agregar un índice común a una tabla existente usamos "create index", indicamos el nombre, sobre qué tabla y el o los campos por los cuales se indexará, entre paréntesis.

Para agregar un índice único a la tabla "libros" usamos:

create unique index i_tituloeditorial on libros (titulo,editorial);

Para agregar un índice único a una tabla existente usamos "create unique index", indicamos el nombre, sobre qué tabla y entre paréntesis, el o los campos por los cuales se indexará.

Un índice PRIMARY no puede agregarse, se crea automáticamente al definir una clave primaria.

Vistas o views

Las vistas son tablas virtuales creadas a partir de una petición . No almacenan los datos que generan, sino solo la petición que permite crearlas. La petición que genera la vista hace referencia a una o varias tablas. La vista puede ser, por ejemplo, un join entre diferentes tablas, la agregación o la extracción de algunas columnas de una tabla. También puede crearse a partir de otra vista.

Las vistas son por lo general de solo lectura y solo permiten leer los datos. Sin embargo, MySQL permite la creación de vistas modificables en ciertas condiciones:

- La petición que genera la vista debe permitir a MySQL localizar la traza del registro que se va a modificar en la tabla o las tablas subyacentes, así como el de todos los valores de cada columna. La petición SELECT que crea la vista no debe contener la cláusula DISTINCT,GROUP BY, HAVING, ni otras funciones de agregación.
- La petición no debe acceder a vistas subyacentes no modificables.

Las vistas pueden utilizarse por diferentes razones; estas permiten:

- Comprobar la integridad al restringir el acceso a los datos para mejorar la privacidad con un particionado vertical y/u horizontal para ocultar los campos a los usuarios. Esto permite personalizar la visualización de información según el tipo de usuario.
- Ocultar la complejidad del esquema. La independencia lógica de los datos es útil para dar a los usuarios el acceso a un conjunto de relaciones representadas en la forma de una tabla. Los datos de la vista son entonces campos de distintas tablas agrupadas, o los resultados de operaciones en estos campos.
- Modificar de forma automática los datos seleccionados (sum() avg() max() ...). Esto permite manipular los valores calculados a partir de otros valores del esquema.
- Conservar la estructura de una tabla si debe modificarse. El esquema puede modificarse sin que sea necesario cambiar las peticiones del lado de la aplicación.

Ya que en general una vista sirve para ocultar la complejidad, no ayuda a mejorar el rendimiento. A menudo lo que ocurre es lo contrario: los desarrolladores pueden olvidar con facilidad que los registros de una vista son a veces difíciles de generar y que, por tanto, es preferible utilizarla lo menos posible si no se es consciente de la repercusión en el rendimiento.

Vistas- create view

Sintaxis de create view

CREATE [OR REPLACE] VIEW nombre_vista [column_list] AS consulta_SELECT

Explicación del código superior para crear vistas en MySQL:

- OR REPLACE: Reemplaza una vista existente en caso de coincidir en nombre.
- nombre_vista: Nombre de la vista a crear.
- column_list: Listado de columnas a crear.
- consulta_SELECT: Consulta SELECT que queremos realizar para obtener la información que contendrá la vista.

Ejemplos de uso del create view:

create view vista_futbolistas as select * from futbolistas;

create view vista_futbolistas as select futbolistas.id, nombre, apellidos from futbolistas
join tarjetas_amarillas on futbolistas.id = tarjetas_amarillas.id_futbolista;

Vistas - drop view

Sintaxis de drop view

```
DROP VIEW [IF EXISTS] nombre_vista1 , nombre_vista2 ...;
```

Explicación del código superior para borrar vistas en MySQL:

- Es recomendable usar la cláusula IF EXISTS, de esta manera no se mostrará error en caso de que la vista que queremos eliminar no exista.
- Se pueden borrar tantas vistas como deseemos, sus nombres deberán ir separadas de una coma.
- Hay que recordar que para poder eliminar una vista hay que tener los privilegios necesarios para realizar dicha acción.

Ejemplos de uso del drop view:

```
drop view if exists vista_usuarios;
```

Vistas – alter view

Sintaxis de alter view

ALTER VIEW nombre_vista [column_list] AS consulta_SELECT;

Tal y como podéis ver en la sintaxis solo hay que poner el nombre de la vista y columnas que queramos modificar seguidas de la consulta select.

También se puede usar CREATE VIEW con la sentencia [OR REPLACE] para modificar o actualizar una vista, en ese caso se borraría la vista para posteriormente crearse con el mismo nombre anterior. (Se considera una mejor opción)

Ejemplos de uso de alter view:

alter view or replace vista_usuarios as select * from usuarios where edad < 30;

Gestión de transacciones

En términos teóricos, una transacción es un conjunto de tareas relacionadas que se realizan de forma satisfactoria o incorrecta como una unidad. En términos de procesamiento, las transacciones se confirman o se anulan. Para que una transacción se confirme se debe garantizar la permanencia de los cambios efectuados en los datos. Los cambios deben conservarse aunque el sistema se bloquee o tengan lugar otros eventos imprevistos. Existen 4 propiedades necesarias, que son conocidas como propiedades ACID:

Atomicidad (atomicity) Coherencia (consistency) Aislamiento (isolation) y Permanencia (durability).

- Estas propiedades garantizan un comportamiento predecible, reforzando la función de las transacciones como proposiciones de todo o nada.
  - Atomicidad: Una transacción es una unidad de trabajo el cual se realiza en su totalidad o no se realiza en ningún caso. Las operaciones asociadas a una transacción comparten normalmente un objetivo común y son interdependientes. Si el sistema ejecutase únicamente una parte de las operaciones, podría poner en peligro el objetivo final de la transacción.
  - Coherencia: Una transacción es una unidad de integridad porque mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado de datos igualmente coherente.
  - Aislamiento: Una transacción es una unidad de aislamiento, permitiendo que transacciones concurrentes se comporten como si cada una fuera la única transacción que se ejecuta en el sistema. El aislamiento requiere que parezca que cada transacción sea la única que manipula el almacén de datos, aunque se puedan estar ejecutando otras transacciones al mismo tiempo. Una transacción nunca debe ver las fases intermedias de otra transacción.
  - Permanencia: Una transacción también es una unidad de recuperación. Si una transacción se realiza satisfactoriamente, el sistema garantiza que sus actualizaciones se mantienen aunque el equipo falle inmediatamente después de la confirmación. El registro especializado permite que el procedimiento de reinicio del sistema complete las operaciones no finalizadas, garantizando la permanencia de la transacción.

Una transacción comienza con la primera instrucción de modificación que se ejecute y finaliza con una operación COMMIT (si la transacción se confirma) o una operación ROLLBACK (si la operación se cancela). Hay que tener en cuenta que cualquier instrucción de definición o de control da lugar a un COMMIT implícito, es decir todas las instrucciones de modificación ejecutadas hasta ese instante pasan a ser definitivas.

Gestión de transacciones

COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos, irrevocables. Se dice que tenemos una transacción confirmada. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros. Además el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de lo que hacemos.

ROLLBACK

Esta instrucción regresa a la instrucción anterior al inicio de la transacción, normalmente el último COMMIT, la última instrucción de definición o de control o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación. Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema dan lugar a un ROLLBACK implícito.

SAVEPOINT

Esta instrucción permite establecer un punto de ruptura. El problema de la combinación ROLLBACK/COMMIT es que un COMMIT acepta todo y un ROLLBACK anula todo. SAVEPOINT permite señalar un punto intermedio entre el inicio de la transacción y la situación actual.

Para regresar a un punto de ruptura concreto se utiliza ROLLBACK TO SAVEPOINT seguido del nombre dado al punto de ruptura. También es posible hacer ROLLBACK TO nombre de punto de ruptura. Cuando se vuelve a un punto marcado, las instrucciones que siguieron a esa marca se anulan definitivamente.

```
create database if not exists transacciones;
use transacciones;
# set autocommit=0; set autocommit=1;
select @@autocommit;
set autocommit=0;
drop table if exists T;
CREATE TABLE if not exists T (FECHA DATE);

INSERT INTO T VALUES ('2017-01-01');
INSERT INTO T VALUES ('2017-02-01');

SAVEPOINT febrero;

INSERT INTO T VALUES ('2017-03-01');
INSERT INTO T VALUES ('2017-04-01');

SAVEPOINT abril;

INSERT INTO T VALUES ('2017-05-01');

ROLLBACK TO febrero;

select * from T;
```

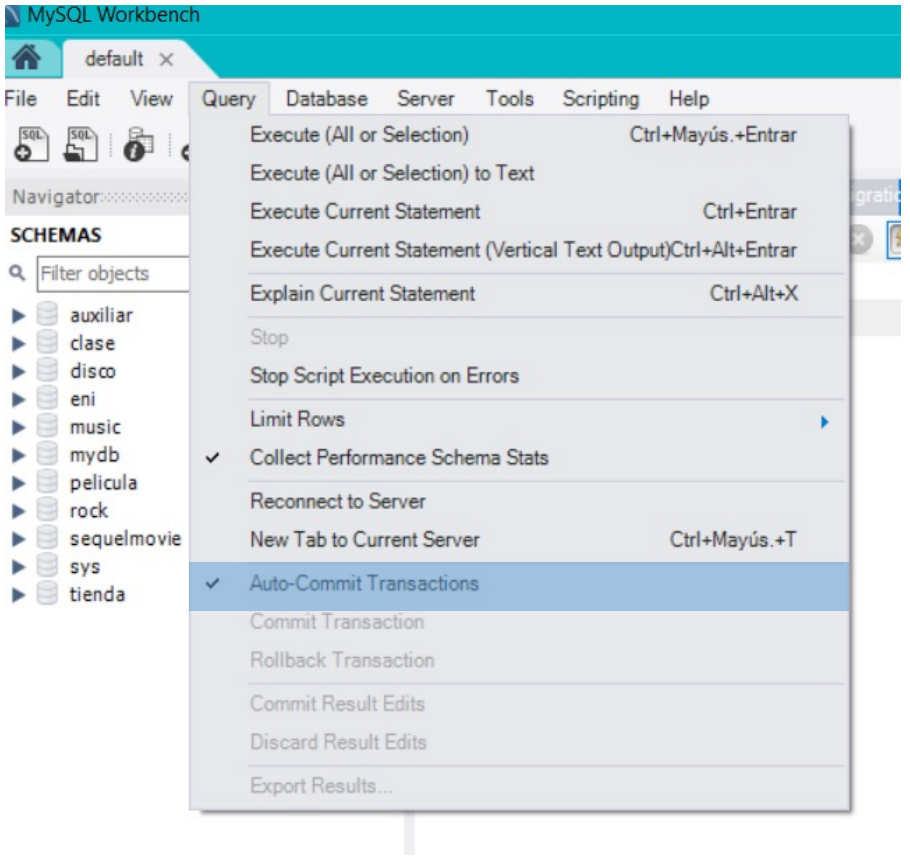
COMMIT; ¿?

Gestión de transacciones

Estado de los datos durante la transacción:

Si se inicia una transacción usando comandos de modificación de datos hay que tener en cuenta que:

- Se puede volver a la instrucción anterior a la transacción cuando se desee.
- Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción muestran los datos ya modificados por las instrucciones DML (Data Modification Language).
- DML = Data Modification Language – Modificaciones (insert, update, alter, delete/drop)
- DCL = Data Control Language – Control – grant, revoke
- DDL = Data Definition Language – Definición – créate
- Select (DML principalmente, pero siempre pueden actuar de varias formas)
- El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Esos usuarios no podrán modificar los valores de dichos registros.
- Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de transacción. Los bloqueos son liberados y los puntos de ruptura borrados.



Gestión de transacciones - Bloqueos

Concurrencia de varias transacciones (Bloqueos)

Cuando se realizan varias transacciones de forma simultánea, pueden darse diversas situaciones en el acceso concurrente a los datos, es decir, cuando se accede a un mismo dato en dos transacciones distintas. Estas situaciones son:

- Lectura sucia (Dirty Read). Una transacción lee datos que han sido escritos por otra transacción que aún no se ha confirmado.
- Lectura no repetible (Non-repeateable Read). Una transacción vuelve a leer los datos que ha leído anteriormente y descubre que otra transacción confirmada ha modificado o eliminado los datos.
- Lectura fantasma (Phantom Read). Una transacción vuelve a ejecutar una consulta que devuelve un conjunto de filas que satisface una condición de búsqueda y descubre que otra transacción confirmada ha insertado filas adicionales que satisfacen la condición.

Para una mejor gestión de estas situaciones debemos indicar el nivel de aislamiento que deseamos. De las cuatro propiedades de ACID de un SGBD, la propiedad de aislamiento es la más laxa. Un nivel de aislamiento bajo aumenta la capacidad de muchos usuarios para acceder a los mismos datos al mismo tiempo, pero también aumenta el número de efectos de concurrencia (como lecturas sucias). Un mayor nivel de aislamiento puede dar como resultado una pérdida de concurrencia y el aumento de las posibilidades de que una transacción bloquee a otra.

Gestión de transacciones - Bloqueos

Concurrencia de varias transacciones (Bloqueos)

Podemos solicitar al SGBD cuatro niveles de aislamiento. De menor a mayor nivel de aislamiento, tenemos:

- READ UNCOMMITTED (Lectura no confirmada). Las sentencias SELECT son efectuadas sin realizar bloqueos, por tanto, todos los cambios hechos por una transacción pueden verlos las otras transacciones. Permite que sucedan las 3 situaciones indicadas previamente: lecturas fantasma, no repetibles y sucias.
- READ COMMITTED (Lectura confirmada). Los datos leídos por una transacción pueden ser modificados por otras transacciones. Se pueden dar lecturas fantasma y lecturas no repetibles.
- REPEATEABLE READ (Lectura repetible). Consiste en que ningún registro leído con un SELECT se puede cambiar en otra transacción. Solo pueden darse lecturas fantasma.
- SERIALIZABLE. Las transacciones ocurren de forma totalmente aislada a otras transacciones. Se bloquean las transacciones de tal manera que ocurren unas detrás de otras, sin capacidad de concurrencia. El SGBD las ejecuta concurrentemente si puede asegurar que no hay conflicto con el acceso a los datos.

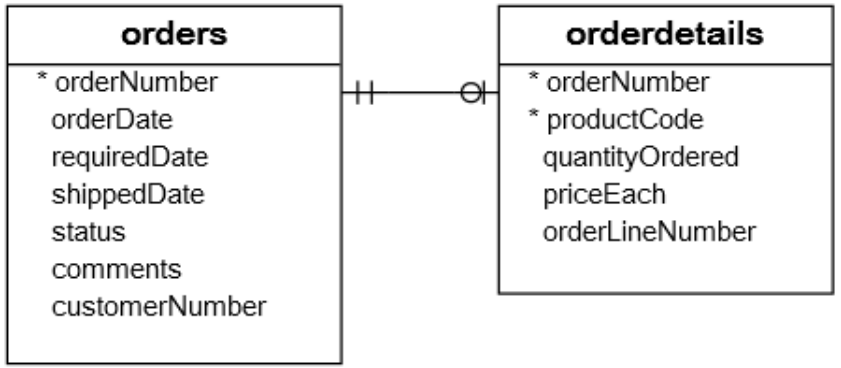
Nivel de aislamiento y Lecturas

Nivel de aislamiento	Lecturas sucias	Lecturas no repetibles	Lecturas fantasma
READ UNCOMMITTED	SÍ	SÍ	SÍ
READ COMMITTED	NO	SÍ	SÍ
REPEATEABLE READ	NO	NO	SÍ
SERIALIZABLE	NO	NO	NO

Internamente el SGBD proporciona dicho nivel de aislamiento mediante bloqueos en los datos.

```
set transaction isolation level serializable;
```

Gestión de transacciones - Bloqueos



Las variables globales del sistema se pueden usar poniendo :
@@nombreVariable
Y también podemos declarar nuevas variables usando
@nombreVariable:=valor

```
-- 1. start a new transaction
START TRANSACTION;

-- 2. Get the latest order number
SELECT
    @orderNumber:=MAX(orderNUmber)+1
FROM
    orders;

-- 3. insert a new order for customer 145
INSERT INTO orders(orderNumber,
                   orderDate,
                   requiredDate,
                   shippedDate,
                   status,
                   customerNumber)
VALUES(@orderNumber,
       '2005-05-31',
       '2005-06-10',
       '2005-06-11',
       'In Process',
        145);
```

```
-- 4. Insert order line items
INSERT INTO orderdetails(orderNumber,
                         productCode,
                         quantityOrdered,
                         priceEach,
                         orderLineNumber)
VALUES(@orderNumber,'S18_1749', 30, '136', 1),
      (@orderNumber,'S18_2248', 50, '55.09', 2);

-- 5. commit changes
COMMIT;
```

Usuarios y privilegios

Hasta ahora hemos usado sólo el usuario 'root', que es el administrador, y que dispone de todos los privilegios disponibles en MySQL.

Sin embargo, normalmente no será una buena práctica dejar que todos los usuario con acceso al servidor tengan todos los privilegios. Para conservar la integridad de los datos y de las estructuras será conveniente que sólo algunos usuarios puedan realizar determinadas tareas, y que otras, que requieren mayor conocimiento sobre las estructuras de bases de datos y tablas, sólo puedan realizarse por un número limitado y controlado de usuarios.

Los conceptos de usuarios y privilegios están íntimamente relacionados. No se pueden crear usuarios sin asignarle al mismo tiempo privilegios. De hecho, la necesidad de crear usuarios está ligada a la necesidad de limitar las acciones que tales usuarios pueden llevar a cabo.

MySQL permite definir diferentes usuarios, y además, asignar a cada uno determinados privilegios en distintos niveles o categorías de ellos.

Tipos de Privilegios

Niveles de privilegios

En MySQL existen cinco niveles distintos de privilegios:

- **Globales:** se aplican al conjunto de todas las bases de datos en un servidor. Es el nivel más alto de privilegio, en el sentido de que su ámbito es el más general.
- **De base de datos:** se refieren a bases de datos individuales, y por extensión, a todos los objetos que contiene cada base de datos.
- **De tabla:** se aplican a tablas individuales, y por lo tanto, a todas las columnas de esas tabla.
- **De columna:** se aplican a una columna en una tabla concreta.
- **De rutina:** se aplican a los procedimientos almacenados. Aún no hemos visto nada sobre este tema, pero en MySQL se pueden almacenar procedimientos consistentes en varias consultas SQL.

Crear usuarios

Crear usuarios

En general es preferible usar GRANT, ya que si se crea un usuario mediante CREATE USER, posteriormente hay que usar una sentencia GRANT para concederle privilegios.

Usando GRANT podemos crear un usuario y al mismo tiempo concederle también los privilegios que tendrá. La sintaxis simplificada que usaremos para GRANT, sin preocuparnos de temas de cifrados seguros que dejaremos ese tema para capítulos avanzados, es:

GRANT tipo_de_derechos ON alcance_de_ derechos TO cuenta_usuario

- tipo_de_derechos: derechos asignados (SELECT, PROCESS, EXECUTE...).
- alcance_de_derechos: todo el servidor (*.*), todos los objetos de un esquema: (nombreddbb.*), una tabla concreta (nombrebbdd.tabla)...
- cuenta_usuario: la cuenta o las cuentas a las que se asignan los derechos.

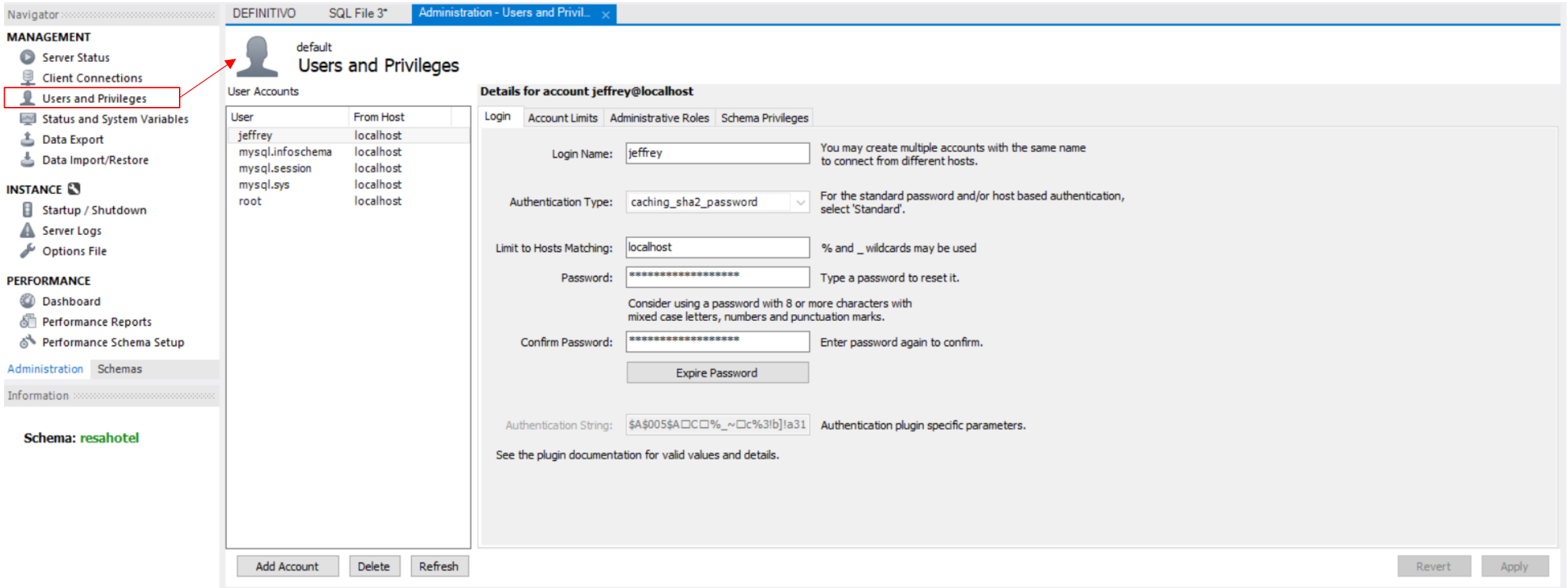
Para crear un usuario llamado ‘jeffrey’ sin privilegios usaremos la sentencia:

CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'password';

Le otorgaremos privilegios sobre la base de datos ‘resahotel’ con:

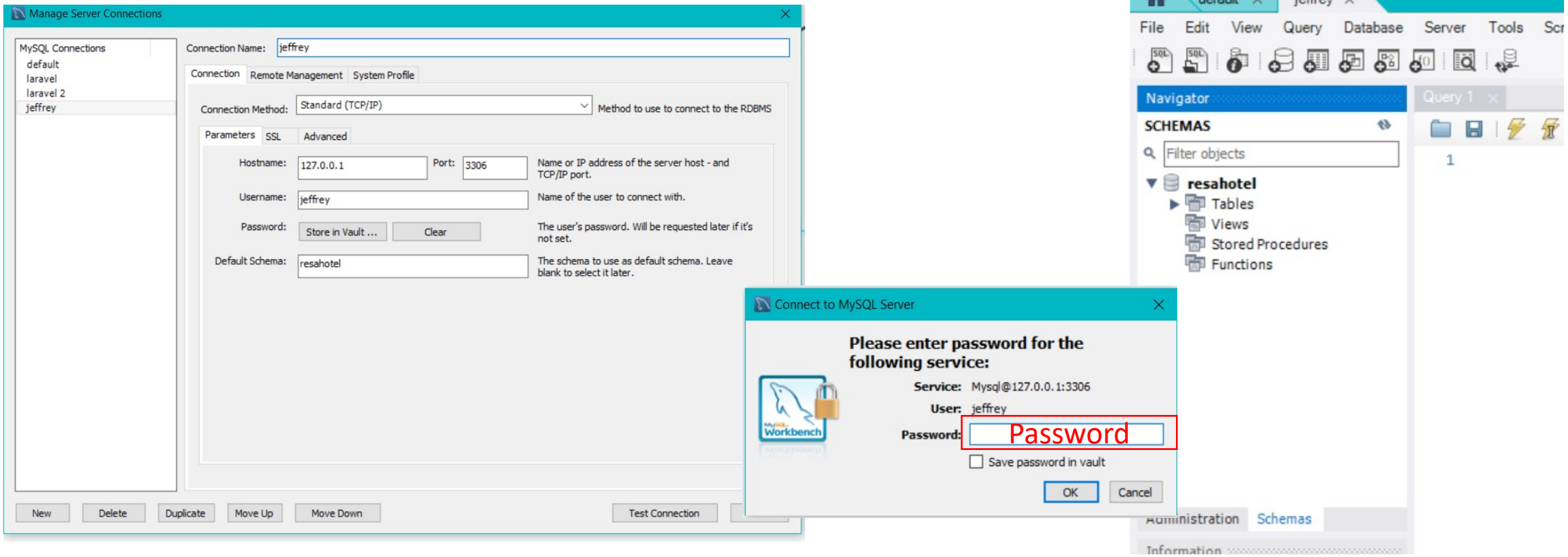
GRANT ALL ON resahotel.* TO 'jeffrey'@'localhost';

Usuarios y privilegios desde MySQL Workbench



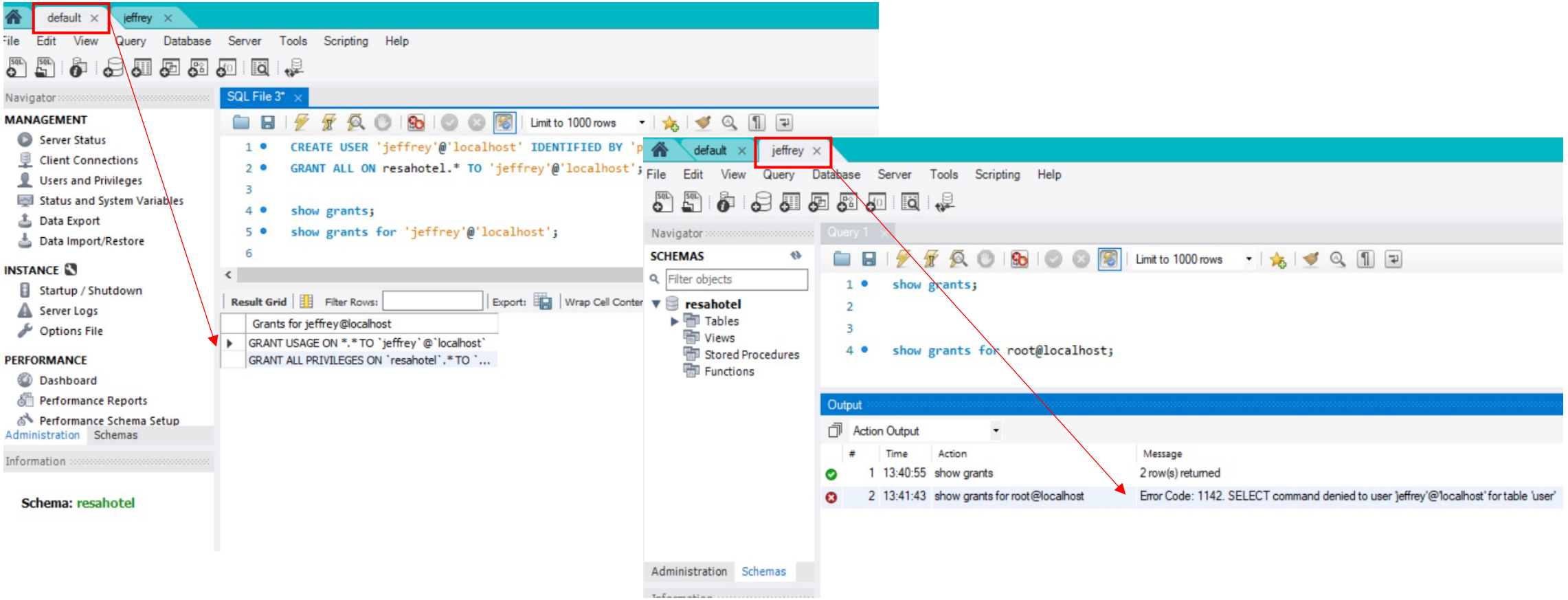
Usuarios y privilegios desde MySQL Workbench

Si lo hemos hecho todo bien, podremos conectarnos con el usuario que hemos creado y ver sus privilegios, en este caso podemos comprobar que tiene acceso a la BBDD llamada 'resahotel'.



Consultar los privilegios

El comando SHOW GRANTS [FOR user@host]; nos permite ver nuestros propios derechos o, si tenemos el privilegio en el esquema, los derechos de las demás cuentas de usuario.



Conceder Privilegios

Conceder privilegios GRANT y :

- Para conceder privilegios globales se usa ON *.*, para indicar que los privilegios se conceden en todas las tablas de todas las bases de datos.
- Para conceder privilegios en bases de datos se usa ON nombre_db.*, indicando que los privilegios se conceden sobre todas las tablas de la base de datos 'nombre_db'.
- Usando ON nombre_db.nombre_tabla, concedemos privilegios de nivel de tabla para la tabla y base de datos especificada.
- En cuanto a los privilegios de columna, para concederlos se usa la sintaxis tipo_privilegio (lista_de_columnas), [tipo_privilegio (lista_de_columnas)].
- Otros privilegios que se pueden conceder son:
  - ALL: para conceder todos los privilegios.
  - CREATE: permite crear nuevas tablas.
  - DELETE: permite usar la sentencia DELETE.
  - DROP: permite borrar tablas.
  - INSERT: permite insertar datos en tablas.
  - UPDATE: permite usar la sentencia UPDATE.
  - ALTER: permite actualizar tablas…… ETC.

Se pueden conceder varios privilegios en una única sentencia.

Revocar Privilegios

Revocar privilegios

Para revocar privilegios se usa la sentencia REVOKE.

REVOKE tipo_de_derechos ON alcance_de_derechos

FROM cuenta_usuario;

o

REVOKE ALL PRIVILEGES, GRANT OPTION FROM cuenta_usuario.

Su sintaxis es similar a la del comando GRANT. La diferencia es sutil y lingüística: la palabra clave TO es sustituida por FROM.

- tipo_de_derechos: derechos asignados (SELECT, PROCESS, EXECUTE...).
- alcance_de_derechos: todo el servidor (*.*), todos los objetos de un esquema: (nombreddbb.*), una tabla concreta (nombrebbdd.tabla)...
- cuenta_usuario: la cuenta o las cuentas a las que se asignan los derechos.

Usuarios y Privilegios - Resumen de los comandos

Los comandos necesarios para la gestión de los derechos son los siguientes:

- CREATE USER : permite crear una cuenta de usuario y asignarle una contraseña.
- CREATE ROLE: permite crear un rol.
- GRANT: permite otorgar derechos, proporcionar una contraseña, especificar los límites de utilización de recursos (como limitar el número de conexiones por hora) y configurar las opciones SSL También puede desempeñar el papel del comando CREATE USER.
- REVOKE: este comando es el opuesto a GRANT. Permite eliminar los derechos de una cuenta de usuario pero no borrar la cuenta.
- DROP USER: permite eliminar una cuenta de usuario (y en consecuencia, todos sus derechos).
- DROP ROLE: permite borrar un rol.
- SHOW GRANTS: permite ver los derechos.

Para saber los usuarios y los hosts en los que tienen permisos podemos usar:

select user, host from mysql.user ;

Ejercicio: ¿Qué está pasando?

```
mysql_fresh> SHOW GRANTS;
+-----------------------------------------------+
| Grants for fresh@%                            |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'fresh'@'%'             |
| GRANT SELECT, INSERT ON `world`.* TO 'fresh'@'%' |
+-----------------------------------------------+

mysql_fresh> SELECT * FROM world.City LIMIT 1;
+----+-------+-------------+----------+------------+
| ID | Name  | CountryCode | District | Population |
+----+-------+-------------+----------+------------+
| 1  | Kabul | AFG         | Kabol    | 1780000    |
+----+-------+-------------+----------+------------+

mysql_root> REVOKE SELECT ON world.* FROM fresh;

mysql_fresh> SHOW GRANTS;
+---------------------------------------+
| Grants for fresh@%                    |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'fresh'@'%'     |
| GRANT INSERT ON `world`.* TO 'fresh'@'%' |
+---------------------------------------+

mysql_fresh> SELECT * FROM world.City LIMIT 1;
+----+-------+-------------+----------+------------+
| ID | Name  | CountryCode | District | Population |
+----+-------+-------------+----------+------------+
| 1  | Kabul | AFG         | Kabol    | 1780000    |
+----+-------+-------------+----------+------------+

mysql_fresh> USE world
Database changed

mysql_fresh> SELECT * FROM world.City LIMIT 1;
ERROR 1142 (42000): SELECT command denied to user 'fresh'@'localhost'
for table 'City'

mysql_root> GRANT ALL ON *.* TO fresh;

mysql_fresh> SHOW GRANTS;
+---------------------------------------+
| Grants for fresh@%                    |
+---------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'fresh'@'%' |
+---------------------------------------+

mysql_fresh>> USE world
ERROR 1044 (42000): Access denied for user 'fresh'@'%' to database
'world'
```

Roles

Por lo general, hay varios usuarios con el mismo conjunto de permisos en la base de datos MySQL. En la versión anterior, solo otorgando y revocando permisos a varios usuarios se pueden cambiar los permisos de cada usuario individualmente. Cuando el número de usuarios es relativamente grande, esta operación requiere mucho tiempo.

MySQL 8.0 proporciona una nueva función de gestión de roles para facilitar la gestión de privilegios de usuario. Una función es un conjunto de permisos específicos. Al igual que una cuenta de usuario, los permisos se pueden otorgar y revocar. Si a un usuario se le otorgan permisos de rol, el usuario tiene los permisos de ese rol.

Las funciones de gestión de roles proporcionadas por MySQL 8.0 son las siguientes:

- CREATE ROL creación de roles
- DROP ROLE Eliminación de roles
- GRANT asigna permisos a usuarios y roles
- REVOKE revoca permisos para usuarios y roles
- SHOW GRANTS muestra los permisos de usuarios y roles
- SET DEFAULT ROLE especifica qué roles de cuenta están activos de forma predeterminada
- SET ROLE cambia el rol activo en la sesión actual
- CURRENT_ROLE () muestra el rol activo en la sesión actual

Crear roles y otorgar permisos de rol de usuario

Aquí tomamos varios escenarios comunes como ejemplos.

- La aplicación requiere permisos de lectura / escritura.
- El personal de operación y mantenimiento necesita acceso completo a la base de datos.
- Algunos desarrolladores necesitan permiso de lectura.
- Algunos desarrolladores necesitan permisos de lectura y escritura.

Si desea otorgar el mismo conjunto de permisos a varios usuarios, debe seguir los pasos a continuación.

- Crea un nuevo rol
- Otorgar permisos de rol
- Otorgar roles de usuario

Paso 1: creamos cuatro roles.

```
CREATE ROLE 'app', 'ops', 'dev_read', 'dev_write';
```

Para distinguir claramente los permisos de los roles, se recomienda nombrar los roles de manera intuitiva.

El formato del nombre del rol es similar a una cuenta de usuario compuesta por partes de usuario y host, como roleName@hostName. Si se omite la parte del host, el valor predeterminado es "%", que significa cualquier host.

Crear roles y otorgar permisos de rol de usuario

PASO 2: otorgamos los permisos correspondientes al rol. Para otorgar permisos de rol, podemos usar GRANT. (Vamos a llamar “mydb” a nuestra base de datos.

GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.* TO 'app’;

Otorgamos al rol de la aplicación los permisos de lectura y escritura de la base de datos “mydb”.

GRANT ALL PRIVILEGES ON mydb.* TO 'ops’;

Otorgamos todos los permisos de la base de datos “mydb” al rol de operaciones

GRANT SELECT ON mydb.* TO 'dev_read’;

Otorgamos permiso de solo lectura a la base de datos “mydb” al rol “dev_read”.

GRANT INSERT, UPDATE, DELETE ON mydb.* TO 'dev_write’;

Otorgamos el permiso de escritura de la base de datos “mydb” al rol “dev_write”.

Crear roles y otorgar permisos de rol de usuario

Paso 3: Agregaremos al usuario designado al rol correspondiente. Supongamos que necesitamos una cuenta utilizada por la aplicación, una cuenta de operador, una cuenta de desarrollador de solo lectura y dos cuentas de desarrollador de lectura y escritura.

Entonces, creamos los usuarios:

- Cuenta de aplicación

  CREATE USER 'app01'@'%' IDENTIFIED BY 'topsecretpass';

- Cuenta personal de operación y mantenimiento

  CREATE USER 'ops01'@'%' IDENTIFIED BY 'topsecretpass';

- Cuenta de desarrollador de solo lectura

  CREATE USER 'dev01'@'%' IDENTIFIED BY 'topsecretpass';

- Cuentas de lectura y escritura

  CREATE USER 'dev02'@'%' IDENTIFIED BY 'topsecretpass';

  CREATE USER 'dev03'@'%' IDENTIFIED BY 'topsecretpass';

Crear roles y otorgar permisos de rol de usuario

Paso 3 (Continuación):

Luego hay que asignar roles a usuarios:

```
GRANT app TO 'app01'@'%';
GRANT ops TO 'ops01'@'%';
GRANT dev_read TO 'dev01'@'%';
```

Y si se desea agregar varios usuarios a varios roles al mismo tiempo, se pueden usar declaraciones similares.

```
GRANT dev_read, dev_write TO 'dev02'@'%', 'dev03'@'%';
```

Verificar permisos de rol

Para verificar que los roles estén asignados correctamente, se puede usar SHOW GRANTS.

SHOW GRANTS FOR 'dev01'@'%';

Grants for dev01@%
GRANT USAGE ON *.* TO `dev01`@`%`
GRANT `dev_read`@`%` TO `dev01`@`%`

La diferencia con la versión anterior (5.7) es que SHOW GRANTS devuelve solo roles otorgados. Si se desea mostrar los permisos representados por el rol, debemos usar USING y el rol de autorización.

SHOW GRANTS FOR 'dev01'@'%' USING dev_read;

Grants for dev01@%
GRANT USAGE ON *.* TO `dev01`@`%`
GRANT SELECT ON `mydb`.* TO `dev01`@`%`
GRANT `dev_read`@`%` TO `dev01`@`%`

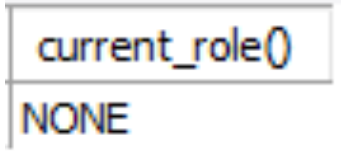
Establecer un rol predeterminado

Ahora, si nos conectamos a MySQL con la cuenta de usuario dev01 e intentamos acceder a la base de datos “mydb”, aparecerá el siguiente error:

ERROR 1044 (42000): Access denied for user 'dev01'@'%' to database ‘mysql’;

Esto se debe a que después de otorgar un rol a una cuenta de usuario, cuando la cuenta de usuario se conecta al servidor de la base de datos, no activa automáticamente el rol.

SELECT current_role();



Devuelve NONE (ninguno), aquí significa que no hay roles habilitados actualmente. Para especificar qué roles deben estar activos cada vez que una cuenta de usuario se conecta al servidor de la base de datos, debemos usar SET DEFAULT ROLE para especificar uno por defecto.

SET DEFAULT ROLE ALL TO 'dev01'@'%';

Establecer rol de actividad

La cuenta de usuario puede modificar los permisos efectivos del usuario actual en la sesión actual especificando qué rol autorizado está activo.

Establecer el rol activo en NONE, lo que significa que no hay ningún rol activo:

SET ROLE NONE;

Configurar el rol activo para todos los roles otorgados.

SET ROLE ALL;

Establecer rol activo como por defecto:

SET ROLE DEFAULT;

Configurar múltiples roles activos al mismo tiempo:

SET ROLE granted_role_1, granted_role_2, ...;

Revocar roles o permisos de rol

Así como se pueden autorizar los roles de un usuario, estos roles también se pueden revocar desde una cuenta de usuario. Para revocar un rol de una cuenta de usuario, debemos usar REVOKE.

REVOKE role FROM user;

REVOKE También se puede utilizar para modificar los permisos de funciones. Esto afecta no solo a los permisos del rol en sí, sino también a los permisos de usuario otorgados al rol. Si se desea que todos los usuarios de desarrollo sean de solo lectura temporalmente, se puede usar REVOKE. Entonces debemos de revocar el permiso de modificación del rol dev_write. Primero veamos los permisos de la cuenta de usuario dev02 antes de revocar.

SHOW GRANTS FOR 'dev02'@'%' USING 'dev_read', 'dev_write';

A continuación, revocamos el permiso de modificación del rol dev_write.

REVOKE INSERT, UPDATE, DELETE ON mydb.* FROM 'dev_write';

Y comprobamos de nuevo:

SHOW GRANTS FOR 'dev02'@'%' USING 'dev_read', 'dev_write';

De los resultados anteriores, se puede ver que revocar los permisos en un rol afectará los permisos de cualquier usuario en ese rol. Por lo tanto, dev02 ahora no tiene permisos de modificación de tablas (se han eliminado los permisos INSERT, UPDATE y DELETE). Si se desea restaurar los permisos de modificación de funciones, solo necesita volver a otorgarlos.

Eliminar rol

Para eliminar uno o más roles, puede usar DROP ROLE

DROP ROLE 'role_name', 'role_name', ...;

Por ejemplo, para eliminar los roles dev_read y dev_write, usamos:

DROP ROLE 'dev_read', 'dev_write';

Copiar los permisos de la cuenta de usuario a otro usuario

MySQL 8.0 trata cada cuenta de usuario como un rol, por lo que se puede otorgar una cuenta de usuario a otra cuenta de usuario. Por ejemplo: copiar los permisos de una cuenta de desarrollador a otra cuenta de desarrollador. Veamos un ejemplo:

Crear una nueva cuenta de usuario de desarrollo:

```
CREATE USER 'dev04'@'%' IDENTIFIED BY ‘topsecretpass’;
```

Copie los permisos de la cuenta de usuario dev02 a la cuenta de usuario dev04:

```
GRANT 'dev02'@'%' TO 'dev04'@'%’;
```

Ver los permisos de la cuenta de usuario dev04:

```
SHOW GRANTS FOR 'dev04'@'%' USING 'dev02';
```

FIN