

MANUAL DE MYSQL

FUENTES DE BIBLIOGRAFIA: - www.mysql.com.ar
- IESE

INDICE

1 - Introducción	5
2 – Lenguaje SQL	5
3 – Conexión con el servidor MySQL	7
4 – Definición de Bases de Datos	7
5 - Creación de una tabla y mostrar sus campos (create table - show tables - describe - drop table).....	8
6 - Tipos de datos	9
7 - Tipos de datos (texto)	9
8 - Tipos de datos (numéricos).....	10
9 - Tipos de datos (fechas y horas)	11
10 - Clave primaria (primary key)	11
11 - Clave primaria compuesta	12
12 - Campo entero con autoincremento.	13
13 - Valores null	14
14 - Valores numéricos sin signo (unsigned).....	15
15 - Valores por defecto.....	15
16 - Valores inválidos.....	16
17 - Atributo default en una columna de una tabla	17
18 - Atributo zerofill en una columna de una tabla.....	19
19 - Índice de una tabla.....	19
20 - Índice de tipo primary.....	20
21 - Índice común (index).....	20
22 - Índice único (unique).....	21
23 - Borrar índice (drop index).....	21
24 - Creación de índices a tablas existentes (create index)	22
25 - Carga de registros a una tabla (insert into)	22
26 - Remplazar registros (replace)	22
27 - Recuperación de registros (select).....	23
28 - Cláusula order by del select.....	24
29 - Recuperación de registros específicos (select - where)	24
30 - Operadores Relacionales (= <> < <= > >=).....	24
31 - Operadores Lógicos (and - or - not)	25
32 - Otros operadores relacionales (between - in).....	26
33 - Búsqueda de patrones (like y not like)	26
34 - Búsqueda de patrones (regexp).....	27

35 - Borrado de registros de una tabla (delete)	28
36 - Comando truncate table.....	28
37 - Modificación de registros de una tabla (update)	29
38 - Columnas calculadas.....	29
39 - Contar registros (count)	30
40 - Funciones de agrupamiento.....	30
41 - Agrupar registros (group by)	31
42 - Selección de un grupo de registros (having)	32
43 - Registros duplicados (distinct)	33
44 - Alias.....	34
45 - Cláusula limit del comando select	34
46 - Recuperación de registros en forma aleatoria(rand)	35
47 - Agregar campos a una tabla (alter table - add)	35
48 - Eliminar campos de una tabla (alter table - drop)	36
49 - Modificar campos de una tabla (alter table - modify)	36
50 - Cambiar el nombre de un campo de una tabla (alter table - change)	37
51 - Agregar y eliminar la clave primaria (alter table)	37
52 - Agregar índices(alter table - add index)	38
53 - Borrado de índices (alter table - drop index)	39
54 - renombrar tablas (alter table - rename - rename table)	39
55 - Tipo de dato enum.....	39
56 - Tipo de dato set.....	40
57 - Tipos de datos blob y text	42
58 - Funciones para el manejo de cadenas.....	43
59 - Funciones matemáticas	45
60 - Funciones para el uso de fecha y hora	46
61 - Funciones de control de flujo (if)	48
62 - Funciones de control de flujo (case)	48
63 - Varias tablas (join)	50
64 - Clave foránea	51
65 - Varias tablas (left join)	52
66 - Varias tablas (right join)	53
67 - Varias tablas (cross join).....	53
68 - Varias tablas (natural join)	54
69 - Varias tablas (inner join - straight join)	54
70 - join, group by y funciones de agrupamiento.	55
71 - join con más de dos tablas.....	55
72 - Función de control if con varias tablas.	56

73 - Variables de usuario.	57
74 - Crear tabla a partir de otra (create - insert)	57
75 - Crear tabla a partir de otras (create - insert - join).....	59
76 - Insertar datos en una tabla buscando un valor en otra (insert - select)	61
77 - Insertar registros con valores de otra tabla (insert - select)	62
78 - Insertar registros con valores de otra tabla (insert - select - join)	63
79 - Actualizar datos con valores de otra tabla (update).....	64
80 - Actualización en cascada (update - join).....	65
81 - Borrar registros consultando otras tablas (delete - join)	66
82 - Borrar registros buscando coincidencias en otras tablas (delete - join)	67
83 - Borrar registros en cascada (delete - join)	67
84 - Chequear y reparar tablas (check - repair).....	68
85 - Encriptación de datos (encode - decode).....	69
86 – Clave Foránea (foreign key).....	70

1 - Introducción

Las **BASES DE DATOS** constituyen el sistema de almacenamiento de la información en los sistemas computacionales actuales. Tanto para aquellos de gran envergadura como los sistemas bancarios, buscadores de Internet o comercio electrónico, como para aquellos de pequeñas empresas y organizaciones.

Desde hace décadas han reemplazado a los sistemas de almacenamiento de datos en archivos ya que permiten guardar todos los datos de una organización brindando seguridad, confiabilidad, permitiendo el acceso personalizado de múltiples usuarios en forma simultánea y segura.

El diseño eficiente de la base de datos es un punto central en la implementación de sistemas ya que asegurará poder procesar todas las consultas necesarias con un tiempo de respuesta adecuado.

SISTEMA GESTOR DE BASES DE DATOS (SGBD)

Un **SGBD** consiste en un conjunto de programas para acceder a bases de datos.

El objetivo principal de un SGBD es proporcionar una forma de almacenar, recuperar y administrar la información de una base de datos de manera que sea tanto práctica como eficiente.

Los SGBD se diseñan para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para procesar la información, respondiendo a consultas, generando informes, etc.

Además, deben proporcionar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o los intentos de acceso sin autorización. Si los datos van a ser compartidos entre diversos usuarios, el sistema debe evitar posibles resultados erróneos que podrían producirse si dos usuarios actualizan un mismo dato en forma simultánea.

Un SGBD también llamado motor de base de datos puede manejar en forma simultánea varias bases de datos.

2 – Lenguaje SQL

SQL es la sigla en inglés Structure Query Language (Lenguaje estructurado de consulta), es el lenguaje estándar para trabajar con bases de datos relacionales y es soportado prácticamente por todos los productos en el mercado aunque puede haber pequeñas diferencias de sintaxis.

SQL incluye instrucciones tanto de definición de datos como de manipulación y consulta de datos.

Las instrucciones pueden escribirse en mayúsculas o en minúsculas y finalizan con un punto y coma (;).

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

Comandos SQL

Existen dos tipos de comandos SQL:

- **DDL** (Data Definition Language – Lenguaje de Definición de Datos): permiten definir la estructura de la base de datos, crear nuevas bases de datos, campos e índices.
- **DML** (Data Manipulation Language – Lenguaje de Manipulación de Datos): permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.

Comandos DDL

Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices
DROP	Empleado para eliminar tablas e índices
ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Comandos DML

Comando	Descripción
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados
DELETE	Utilizado para eliminar registros de una tabla de una base de datos

Cláusulas SQL

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

Operadores

Lógicos:

Operador	Uso
AND	Es el "y" lógico. Evalúa dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdad si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

De comparación:

Operador	Uso
<	Menor que
>	Mayor que
<	Distinto de
<=	Menor ó Igual que
=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo
In	Utilizado para especificar registros de una base de datos

Funciones de agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

MySQL

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.

3 – Conexión con el servidor MySQL

El SGBD MySQL es un servidor de BD de libre distribución actualmente propiedad de la firma ORACLE con versiones para Windows y Linux.

Para Windows puede descargarse libremente como aplicación única (www.mysql.com), o como parte de un compilado como WAMPServer (www.wampserver.es) o XAMPP (www.xampp.es). Para el presente curso utilizaremos un clon de MySQL denominado MariaDB (www.mariadb.org).

La aplicación puede ser instalada localmente o en forma remota, y se puede acceder para su administración desde alguna aplicación de escritorio (www.heidisql.com), o aplicación web (www.phpmyadmin.net), o directamente desde la línea de comandos.

Al instalar el servidor solicitara la clave para el usuario “**root**” (administrador) debemos definirla y recordarla para conectarnos luego.

Luego de instalado para establecer la conexión desde la línea de comandos ejecutamos CMD y luego las sentencias:

- Si el servidor es local:

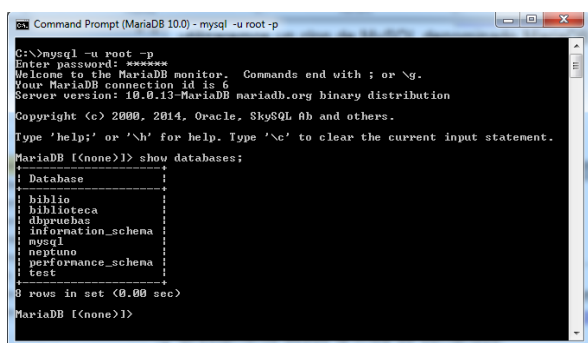
mysql -u nombre.usuario -p (Ej: mysql -u root -p)

- Si el servidor es remoto:

mysql -u nombre.usuario -h ip.servidor -p (Ej: mysql -u root -h 192.168.1.105 -p)

Una vez establecida la conexión podemos correr comandos sql.

Ej: **show databases;**



```

C:\>mysql -u root -p
Enter password: *****
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 6
Server version: 10.0.13-MariaDB mariadb.org binary distribution
Copyright (c) 2000, 2014, Oracle, SkySQL Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| biblio   |
| biblioteca |
| dbpruebas |
| information_schema |
| mysql    |
| neptuno  |
| performance_schema |
| test     |
+-----+
0 rows in set (0.00 sec)

MariaDB [(none)]>

```

4 – Definición de Bases de Datos

Instrucciones que permiten crear, modificar o eliminar bases de datos, tablas, vistas, restricciones o usuarios, asignar permisos, definir tipos de datos, etc. En un mismo servidor o SGBD, pueden almacenarse y consultarse muchas bases de datos simultáneamente.

Instrucción para crear una base de datos:

create database administracion;

Instrucción para eliminar una base de datos:

drop database administracion;

Instrucción para mostrar las bases de datos existentes en el servidor:

show databases;

Instrucción para poner en uso una base de datos:

use database administracion;

5 - Creación de una tabla y mostrar sus campos (create table - show tables - describe - drop 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

Gráficamente acá tenemos la tabla usuarios, que contiene dos campos llamados: nombre y clave. Luego tenemos tres registros almacenados en esta tabla, el primero almacena en el campo nombre el valor "MarioPerez" y en el campo clave "Marito", y así sucesivamente con los otros dos registros.

Las tablas forman parte de una base de datos.

Para ver las tablas existentes en una base de datos usamos el comando:

show tables;

Deben aparecer todas las tablas de la Base de Datos.

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", con la siguiente instrucción:

```
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.

Para ver las tablas existentes en una base de datos tipeamos nuevamente:

show tables;

Ahora aparece "usuarios" entre otras que ya pueden estar creadas.

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 de longitud, que almacenará el nombre de usuario y
- 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.

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.

6 - Tipos de datos

Al crear una tabla debemos resolver qué campos (columnas) tendrá y que tipo de datos almacenará cada uno de ellos, es decir, su estructura. MySQL permite manejar varios **tipos y subtipos de datos**.

Podemos clasificar los tipos de datos en:

- 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**.
- NUMEROS:** 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**.
- FECHAS Y HORAS:** para guardar fechas y horas dispone de varios tipos: **date** (fecha), **datetime** (fecha y hora), **time** (hora), **year** (año) y **timestamp**.
- OTROS TIPOS:** **enum** y **set** representan una enumeración y un conjunto respectivamente.
- Otro valor que podemos almacenar es el valor "**null**". El valor '**null**' significa "valor desconocido" o "dato inexistente". No es lo mismo que 0 o una cadena vacía.

7 - 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. Tenemos los siguientes tipos:

- 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.
- 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.

- 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 carácter, conviene usar **char(1)**, que ocupa 1 byte y no **varchar(1)**, que ocupa 2 bytes.

Tipo	Bytes de almacenamiento
char(x)	x
varchar(x)	x+1

8 - 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.

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.

Tipo	Bytes de almacenamiento
tinyint	1
smallint	2
mediumint	3

int	4
bigint	8
float	4
decimal(t,d)	t+2 si d>0, t+1 si d=0 y d+2 si t<d

9 - Tipos de datos (fechas y horas)

Para guardar fechas y horas dispone de varios tipos:

- 1) **date**: representa una fecha con formato "**YYYY-MM-DD**". El rango va de "1000-01-01" a "9999-12-31".
- 2) **datetime**: almacena fecha y hora, su formato es "**YYYY-MM-DD HH:MM:SS**". El rango es de "1000-01-01 00:00:00" a "9999-12-31 23:59:59".
- 3) **time**: una hora. Su formato es "**HH:MM:SS**". El rango va de "-838:59:59" a "838:59:59".
- 4) **year(2)** y **year(4)**: un año. Su formato es "**YYYY**" o "**YY**". Permite valores desde 1901 a 2155 (en formato de 4 dígitos) y desde 1970 a 2069 (en formato de 2 dígitos).

Si ingresamos los valores como cadenas, un valor entre "00" y "69" es convertido a valores "**year**" en el rango de 2000 a 2069; si el valor está entre "70" y "99", se convierten a valores "**year**" en el rango 1970 a 1999.

Si ingresamos un valor numérico 0, se convierte en "0000"; entre 1 y 69, se convierte a valores "**year**" entre 2001 a 2069; entre 70 y 99, es convertido a valores "**year**" de 1970 a 1999.

Para almacenar valores de tipo fecha se permiten como separadores "/", "-" y ".".

Si ingresamos '06-12-31' (año de 2 dígitos), lo toma como '2006-12-31'.

Si ingresamos '2006-2-1' (mes y día de 1 dígito), lo toma como '2006-02-01'.

Si ingresamos '20061231' (cadena sin separador), lo toma como '2006-12-31'.

Si ingresamos 20061231 (numérico), lo toma como '2006-12-31'.

Si ingresamos '20061231153021' (cadena sin separadores), lo toma como '2006-12-31 15:30:21'.

Si ingresamos '200612311530' (cadena sin separadores con un dato faltante) no lo reconoce como fechahora y almacena ceros.

Si ingresamos '2006123' (cadena sin separadores con un dato faltante) no lo reconoce como fecha y almacena ceros.

Si ingresamos '2006-12-31 11:30:21' (valor date time) en un campo 'date', toma sólo la parte de la fecha, la hora se corta, se guarda '2006-12-31'.

Es importante elegir el tipo de dato adecuado según el caso, el más preciso. Por ejemplo, si sólo necesitamos registrar un año (sin día ni mes), el tipo adecuado es "year" y no "date".

Tipo	Bytes de almacenamiento
date	3
datetime	8
time	3
year	1

10 - Clave primaria (primary key)

Una **clave primaria** es un campo (o varios) que identifica un solo registro (fila) en una tabla. Para un valor del campo clave existe solamente un 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)  
);
```

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.

Al establecer una clave primaria estamos indexando la tabla, es decir, creando un índice para dicha tabla.

11 - 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.

Retomemos 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.

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 autoincrementable 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. Por ejemplo, podemos eliminar el campo "horallegada":

```
alter table vehiculos drop horallegada;
```

Siempre que no haya registros con "patente" duplicada, en ese caso aparece un mensaje de error y la eliminación del campo no se realiza.

En caso de ejecutarse la sentencia anterior, la clave queda formada sólo por el campo "patente".

12 - 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 autoincrementa sus valores automáticamente, éste debe ser entero (**integer**) y debe ser clave primaria:

```

create table libros(
        codigo int auto_increment,
        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.

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.

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

```

insert into libros (codigo,titulo,autor,editorial)
values(6,'Martín Fierro','Jose Hernandez','Paidós');

```

Pero 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.

13 - 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**".

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 ej., en la 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, 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 que aún no hemos definido el precio:

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

Note 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.

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"**.

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

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 retornan 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.

14 - Valores numéricos sin signo (unsigned)

Otro atributo que permiten los campos de tipo numérico es **"unsigned"**. El atributo **"unsigned"** (sin signo) permite sólo valores positivos.

Si necesitamos almacenar edades, por ejemplo, nunca guardaremos valores negativos, entonces sería adecuado definir un campo "edad" de tipo entero sin signo:

```
edad integer unsigned;
```

Si necesitamos almacenar el precio de los libros, definimos un campo de tipo **"float unsigned"** porque jamás guardaremos un valor negativo.

Hemos aprendido que al crear una tabla, es importante elegir el tipo de dato adecuado, el más preciso, según el caso. Si un campo almacenará sólo valores positivos, es útil definir dicho campo con este atributo.

En los tipos enteros, **"unsigned"** duplica el rango, es decir, el tipo **"integer"** permite valores de -2000000000 a 2000000000 aprox., si se define **"integer unsigned"** el rango va de 0 a 4000000000 aprox.

Los tipos de coma flotante (**float** por ejemplo) también aceptan el atributo **"unsigned"**, pero el valor del límite superior del rango se mantiene.

15 - Valores por defecto

Hemos visto los valores por defecto de los distintos tipos de datos. Ahora que conocemos más tipos de datos, vamos a ampliar la información referente a ellos y a repasar los conocidos.

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"**; tema que veremos más adelante.

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.

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"
caracter not null	cadena vacía	permite
numerico not null	0	permite
fecha not null	0000-00-00	permite
hora not null	00:00:00	permite
auto_increment	ultimo+1, empieza en 1	no permite
carac., numer., fecha, hora null	null	permite

16 - Valores inválidos

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 inválido. Un valor es inválido por tener un tipo de dato incorrecto para el campo o por estar fuera de rango.

Veamos los distintos tipos de datos inválidos.

Para **campos de tipo caracter**:

- **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.

Para **campos numéricos**:

- **cadena**: 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** el tratamiento es el siguiente:

- 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.

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 inválido 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.

Los valores inválidos para otros tipos de campos lo trataremos más adelante.

RESUMEN:

Tipo	Valor inválido	Resultado
caracter null/ not null	123	'123'
caracter null/ not null	mayor longitud	se corta
caracter not null	null	error
numérico null/ not null	'123'	0
numérico null/ not null	fuera de rango	límite más cercano
numérico not null	null	error
numérico decimal null/ not null	más decimales que los definidos	redondea al más cercano
num. auto_incr. c/signo null/not null	0	siguiente de la secuencia
num. auto_incr. s/signo null/not null	todos los menores a 1	siguiente de la secuencia
num. auto_incr. c/s signo null	null	siguiente de la secuencia
num. auto_incr. c/s signo null/not null	valor existente	error
fecha	fuera de rango	0000-00-00
fecha	'20-07-2006' (otro orden)	0000-00-00
hora	fuera de rango	límite más cercano
fecha y hora not null	null	error

17 - 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"**.

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 traves 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 'Desconocido',
    precio decimal(5,2) unsigned default 1.11,
    cantidad int unsigned not null,
    primary key (codigo)
);
```

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.

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);
```

18 - Atributo zerofill en una columna de una tabla

Cualquier campo numérico puede tener otro atributo extra **"zerofill"**, que 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.

19 - 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.

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.

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;
```

20 - 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 carácter.

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,
    titulo varchar(40) not null,
    autor varchar(30),
    editorial varchar(15),
    primary key(codigo)
);
```

21 - Índice común (index)

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.

Vamos a trabajar con nuestra tabla "libros". Un campo por el cual realizamos consultas frecuentemente es "editorial", indexar la tabla por ese campo sería útil.

Creemos un índice al momento de crear 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.

```
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),
primary key(codigo),
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.

Recuerde que **"index"** es sinónimo de **"key"**.

22 - 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.

```
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	Si	No	1
común	darlo o por defecto	"index" o "key"	No	Si	varios
único	darlo o por defecto	"unique"	Si	Si	varios

23 - 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).

24 - Creación de índices a tablas existentes (create index)

Podemos agregar un índice a una tabla existente.

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 común a la tabla "libros" usamos:

```
create index i_editorial on libros (editorial);
```

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.

25 - Carga de registros a una tabla (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.

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.

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

26 - Remplazar registros (replace)

La sentencia **"replace"** reemplaza un registro por otro.

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.

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.

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 se 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 ahí','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','Hawking','Emece',48);
```

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

27 - Recuperación de registros (select)

Para ver todos los registros de una tabla:

```
select * from libros;
```

El comando **"select"** recupera los registros de una tabla. Con el asterisco (*) indicamos que seleccione todos los campos de la tabla que nombramos, y con la cláusula **"from"** indicamos la tabla de la cual recuperar los registros.

Para recuperar solo algunos campos de la tabla, usamos el comando **"select"** e indicamos los nombres de los campos a rescatar.

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

```
select titulo,autor,editorial from libros;
```

En la sentencia anterior la consulta mostrará sólo los campos "titulo", "autor" y "editorial". En la siguiente sentencia, veremos los campos correspondientes al título y precio de todos los libros:

```
select titulo, precio from libros;
```

Para ver solamente la editorial y la cantidad de libros tipeamos:

```
select editorial, cantidad from libros;
```

28 - Cláusula order by del select

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.

29 - Recuperación de registros específicos (select - where)

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.

30 - Operadores Relacionales (= <> < <= > >=)

Vimos como especificar condiciones de igualdad para seleccionar registros de una tabla; p.ej.:

```
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', con la condición:

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

Podemos comparar valores numéricos. P.ej., mostrar los libros de precios 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;
```

31 - 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 pesos, 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.

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.

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.

El orden de prioridad de los operadores lógicos es el siguiente: "**not**" se aplica antes que "**and**" y "**and**" antes que "**or**", si no se especifica un orden de evaluación mediante el uso de paréntesis. El orden en el que se evalúan los operadores con igual nivel de precedencia es indefinido, por ello se recomienda usar los paréntesis.

32 - Otros 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).

Si agregamos el operador "**not**" antes de "**between**" el resultado se invierte.

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).

Para recuperar los libros cuyo autor no sea 'Paenza' ni 'Borges' usamos:

```
select * from libros where ((autor<>'Borges') and (autor<>'Paenza'));
```

También podemos usar "**in**" :

```
select * from libros where (autor not in ('Borges','Paenza'));
```

Con "**in**" averiguamos si el valor del campo está incluido en la lista, con "**not**" antecediendo la condición, invertimos el resultado.

33 - Búsqueda de patrones (like y not like)

Hemos realizado consultas utilizando operadores relacionales para comparar cadenas. Por ejemplo, sabemos recuperar los libros cuyo autor sea igual a la cadena "Borges":

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

Los operadores relacionales nos permiten comparar valores numéricos y cadenas de caracteres. Pero al realizar la comparación de cadenas, busca coincidencias de cadenas completas.

Imaginemos que tenemos registrados estos 2 libros:

El Aleph de Borges;
Antología 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 tipear:

```
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.

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 guión 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 tipeamos 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 guión bajo, si utilizamos '_%', mostrará todos los registros porque lo interpreta como "patrón que comienza con un carácter cualquiera y sigue con cualquier cantidad de caracteres". Debemos utilizar "_%", esto se interpreta como "patrón que comienza con guión 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 "%\".

34 - Búsqueda de patrones (regexp)

Los operadores **"regexp"** y **"not regexp"** buscan 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" tipeamos:

```
select autor from libros where (autor regexp '[hkw]');
```

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

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

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

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

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

```
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$');
```

Para buscar títulos que contengan una "a" luego un caracter cualquiera y luego una "e" utilizamos la siguiente sentencia:

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

El punto (.) identifica cualquier caracter.

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 caracter inmediatamente anterior, en este caso cualquiera porque hay un punto.

35 - 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.

36 - Comando truncate table.

Aprendimos que para borrar todos los registros 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 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 autoincrementable, 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 autoincrementable 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.

37 - 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. 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.

38 - 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) as total 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;
```

39 - Contar registros (count)

Existen en MySQL funciones que nos permiten contar registros, calcular sumas, promedios, obtener valores máximos y mínimos. Veamos algunas de ellas.

Imaginemos que nuestra tabla "libros" contiene 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 que tienen valor nulo.

Para saber la cantidad de libros de la editorial "Planeta" tipeamos:

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

También podemos utilizar esta función junto con la cláusula **"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 tener en cuenta los que tienen valor nulo), usamos la función **"count()"** y en los paréntesis colocamos el nombre del campo que necesitamos contar:

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

Note que **"count(*)"** retorna la cantidad de registros de una tabla (incluyendo los que tienen valor **"null"**) mientras que **"count(precio)"** retorna 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.

Tenga 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.

40 - Funciones de agrupamiento

Ya hemos aprendido "count()", veamos otras.

La función **"sum()"** retorna 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%');
```

La función **"avg()"** retorna 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.

41 - Agrupar registros (group by)

Hemos aprendido que las funciones de agrupamiento permiten contar registros, calcular sumas y promedios, obtener valores máximos y mínimos. También dijimos que dichas funciones operan sobre conjuntos de registros, no con datos individuales.

Generalmente estas funciones se combinan con la sentencia "**group by**", que agrupa registros para consultas detalladas.

Queremos saber la cantidad de visitantes de cada ciudad, usamos la siguiente sentencia:

```
select count(*) from visitantes where (ciudad='Cordoba');
```

y repetirla con cada valor de "ciudad":

```
select count(*) from visitantes where (ciudad='Alta Gracia');
```

```
select count(*) from visitantes where (ciudad='Villa Dolores');
```

etc.

Pero hay otra manera, utilizando la cláusula "**group by**":

```
select ciudad, count(*) from visitantes group by ciudad;
```

Entonces, para saber la cantidad de visitantes que tenemos en cada ciudad utilizamos la función "**count()**", agregamos "**group by**" y el campo por el que deseamos que se realice el agrupamiento, también colocamos el nombre del campo a recuperar.

La instrucción anterior solicita que muestre el nombre de la ciudad y cuente la cantidad agrupando los registros por el campo "ciudad". Como resultado aparecen los nombres de las ciudades y la cantidad de registros para cada valor del campo.

Para obtener la cantidad visitantes con teléfono no nulo, de cada ciudad utilizamos la función "**count()**" enviándole como argumento el campo "telefono", agregamos "**group by**" y el campo por el que deseamos que se realice el agrupamiento (ciudad):

```
select ciudad, count(telefono) from visitantes group by ciudad;
```

Como resultado aparecen los nombres de las ciudades y la cantidad de registros de cada una, sin contar los que tienen teléfono nulo. Recuerde la diferencia de los valores que retorna la función "**count()**" cuando enviamos como argumento un asterisco o el nombre de un campo: en el primer caso cuenta todos los registros incluyendo los que tienen valor nulo, en el segundo, los registros en los cuales el campo especificado es no nulo.

Para conocer el total de las compras agrupadas por sexo:

```
select sexo, sum(montocompra) from visitantes group by sexo;
```

Para saber el máximo y mínimo valor de compra agrupados por sexo:

```
select sexo, max(montocompra) from visitantes group by sexo;
```

```
select sexo, min(montocompra) from visitantes group by sexo;
```

Se pueden simplificar las 2 sentencias anteriores en una sola sentencia, ya que usan el mismo "**group by**":

```
select sexo, max(montocompra), min(montocompra) from visitantes group by sexo;
```

Para calcular el promedio del valor de compra agrupados por ciudad:

```
select ciudad, avg(montocompra) from visitantes group by ciudad;
```

Podemos agrupar por más de un campo, por ejemplo, vamos a hacerlo por "ciudad" y "sexo":

```
select ciudad, sexo, count(*) from visitantes group by ciudad,sexo;
```

También es posible limitar la consulta con "**where**".

Vamos a contar y agrupar por ciudad sin tener en cuenta "Cordoba":

```
select ciudad, count(*) from visitantes where (ciudad<>'Cordoba') group by ciudad;
```

Podemos usar las palabras claves "**asc**" y "**desc**" para una salida ordenada:

```
select ciudad, count(*) from visitantes group by ciudad desc;
```

42 - 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 retornadas 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;
```

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

```
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".

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**".

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;
```

43 - Registros duplicados (distinct)

Con la cláusula "**distinct**" se especifica que los registros con ciertos datos duplicados sean obviados 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;
```

También podemos utilizar:

```
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;
```

Note que si contamos los autores sin "**distinct**", no incluirá los valores "**null**" pero si los repetidos:

```
select count(autor) from libros;
```

Esta sentencia cuenta los registros que tienen autor.

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.

44 - 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 cláusulas **"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"**.

Los alias serán de suma importancia cuando rescate datos desde el lenguaje PHP.

45 - Cláusula limit del comando select

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

Recibe 1 ó 2 argumentos numéricos enteros positivos; el primero indica el número del primer registro a retornar, el segundo, el número máximo de registros a retornar. 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.

```
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 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.

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. 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.

46 - Recuperación de registros en forma aleatoria(rand)

Una librería que tiene almacenados los datos de sus libros en una tabla llamada "libros" quiere donar a una institución 5 libros tomados al azar.

Para recuperar de una tabla registros aleatorios se puede utilizar la función **"rand()"** combinada con **"order by"** y **"limit"**:

```
select * from libros order by rand() limit 5;
```

Nos devuelve 5 registros tomados al azar de la tabla "libros".

Podemos ejecutar la sentencia anterior varias veces seguidas y veremos que los registros recuperados son diferentes en cada ocasión.

47 - 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.

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,
- título, **varchar(40) not null**,
- autor, **varchar(30)**,
- editorial, **varchar (20)**,
- precio, **decimal(5,2) unsigned**.

Necesitamos agregar el campo "cantidad", de tipo **smallint unsigned not null**, tipeamos:

```
alter table libros add cantidad smallint 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 tinyint unsigned after autor;
```

48 - 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" tipeamos:

```
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.

49 - Modificar campos de una tabla (alter table - modify)

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

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

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

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

```
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, tipeamos:

```
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.

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 setear 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;
```

50 - 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),
- costo, decimal(5,2) unsigned,
- cantidad int unsigned,
- clave primaria: código.

Queremos cambiar el nombre del campo "costo" por "precio", tipeamos:

```
alter table libros change costo 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"**, tipeamos:

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

51 - 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,
- título, 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.

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 autoincrementable. Tampoco es posible definir un campo como autoincrementable 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.

52 - Agregar índices(alter table - add index)

Aprendimos a crear índices al momento de crear una tabla. También a crearlos luego de haber creado la tabla, con "**create index**". También podemos agregarlos a una tabla usando "**alter table**".

Creamos la tabla "libros":

```
create table libros(
    codigo int unsigned,
    título varchar(40),
    autor varchar(30),
    editorial varchar (20),
    precio decimal(5,2) unsigned,
    cantidad smallint unsigned
);
```

Para agregar un índice común por el campo "editorial" usamos la siguiente sentencia:

```
alter table libros add index i_editorial (editorial);
```

Usamos "**alter table**" junto con "**add index**" seguido del nombre que le daremos al índice y entre paréntesis el nombre de el o los campos por los cuales se indexará.

Para agregar un índice único multicampo, por los campos "título" y "editorial", usamos la siguiente sentencia:

```
alter table libros add unique index i_tituloeditorial (título,editorial);
```

Usamos "**alter table**" junto con "**add unique index**" seguido del nombre que le daremos al índice y entre paréntesis el nombre de el o los campos por los cuales se indexará.

En ambos casos, para índices comunes o únicos, si no colocamos nombre de índice, se coloca uno por defecto, como cuando los creamos junto con la tabla.

53 - Borrado de índices (alter table - drop index)

Los índices común y únicos se eliminan con "**alter table**".

Trabajamos con la tabla "libros" de una librería, que tiene los siguientes campos e índices:

```
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),  
    unique i_tituloeditorial (titulo,editorial)  
);
```

Para eliminar un índice usamos la siguiente sintaxis:

```
alter table libros drop index i_editorial;
```

Usamos "**alter table**" y "**drop index**" seguido del nombre del índice a borrar.

Para eliminar un índice único usamos la misma sintaxis:

```
alter table libros drop index i_tituloeditorial;
```

54 - renombrar tablas (alter table - rename - rename table)

Podemos cambiar el nombre de una tabla con "**alter table**".

Para cambiar el nombre de una tabla llamada "amigos" por "contactos" usamos esta sintaxis:

```
alter table amigos rename contactos;
```

Entonces usamos "**alter table**" seguido del nombre actual, "**rename**" y el nuevo nombre.

También podemos cambiar el nombre a una tabla usando la siguiente sintaxis:

```
rename table amigos to contactos;
```

La renombración se hace de izquierda a derecha, con lo cual, si queremos intercambiar los nombres de dos tablas, debemos tipear lo siguiente:

```
rename table amigos to auxiliar, contactos to amigos, auxiliar to contactos;
```

55 - Tipo de dato enum

Además de los tipos de datos ya conocidos, existen otros que analizaremos ahora, los tipos "**enum**" y "**set**".

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 postulantes a los puestos en una tabla llamada "postulantes". 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 postulantes(
    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.

Si un **"enum"** permite valores nulos, el valor por defecto es **"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 postulantes (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 postulantes (documento,nombre,estudios) values('22255265','Juana Pereyra',0);
```

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

```
insert into postulantes (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.

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 postulantes 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 postulantes 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"** dependen del número de valores enumerados.

56 - 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 postulantes a los puestos en una tabla llamada "postulantes". 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 postulantes(
    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');
```

Para ingresar un valor que contenga más de un elemento del conjunto, se separan por comas, por ejemplo:

```
insert into postulantes (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, por ejemplo, si ingresamos:

```
insert into postulantes (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, por ejemplo, si ingresamos:

```
insert into postulantes (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 postulantes (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 postulantes (documento,nombre,idioma) values('22255265','Juana Pereyra',0);
insert into postulantes (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. Los valores de índice se definen en el siguiente orden, en este ejemplo:

```
1='ingles',
2='italiano',
3='ingles,italiano',
4='portuges',
5='ingles,portuges',
6='italiano,portuges',
7='ingles,italiano,portuges'.
```

Ingresamos algunos registros con valores de índice:

```
insert into postulantes (documento,nombre,idioma) values('22255265','Juana Pereyra',2);
insert into postulantes (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.

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()"** retorna 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" tipeamos:

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

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 retornará registros.

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

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

También podemos buscar por el número de índice:

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

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

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

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

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

Los tipos **"set"** admiten cláusula **"default"**.

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

57 - Tipos de datos blob y text

Los tipos **"blob"** o **"text"** son bloques de datos. Tienen una longitud de 65535 caracteres.

Un **"blob"** (**B**inary **L**arge **O**bject) puede almacenar un volumen variable de datos. La diferencia entre **"blob"** y **"text"** es que **"text"** diferencia mayúsculas y minúsculas y **"blob"** no; esto es porque **"text"** almacena cadenas de caracteres no binarias (caracteres), en cambio **"blob"** contiene cadenas de caracteres binarias (de bytes).

No permiten valores **"default"**.

Existen subtipos:

- **tinyblob** o **tinytext**: longitud máxima de 255 caracteres.
- **mediumblob** o **mediumtext**: longitud de 16777215 caracteres.
- **longblob** o **longtext**: longitud para 4294967295 caracteres.

Se utiliza este tipo de datos cuando se necesita almacenar imágenes, sonidos o textos muy largos.

Un video club almacena la información de sus películas en alquiler en una tabla denominada "peliculas". Además del título, actor y duración de cada película incluye un campo en el cual guarda la sinopsis de cada una de ellas.

La tabla contiene un campo de tipo **"text"** llamado "sinopsis":

- código: **int unsigned auto_increment**, clave primaria,
- nombre: **varchar(40)**,
- actor: **varchar(30)**,
- duración: **tinyint unsigned**,
- sinopsis: **text**,

Se ingresan los datos en un campo **"text"** o **"blob"** como si fuera de tipo cadena de caracteres, es decir, entre comillas:

insert into peliculas **values**(1,'Mentes que brillan','Jodie Foster',120, 'El no entiende al mundo ni el mundo lo entiende a él; es un niño superdotado. La escuela especial a la que asiste tampoco resuelve los problemas del niño. Su madre hará todo lo que esté a su alcance para ayudarlo. Drama');

Para buscar un texto en un campo de este tipo usamos "like":

select * from peliculas **where** sinopsis **like** '%Drama%';

No se pueden establecer valores por defecto a los campos de tipo "blob" o "text", es decir, no aceptan la cláusula "default" en la definición del campo.

58 - Funciones para el manejo de cadenas

RECUERDE que 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');
Retorna: 65.
- **char**(x,...): retorna una cadena con los caracteres en código ASCII de los enteros enviados como argumentos.
Ejemplo: **select char**(65,66,67);
Retorna: "ABC".
- **concat**(cadena1,cadena2,...): devuelve la cadena resultado de concatenar los argumentos.
Ejemplo: **select concat**('Hola',' ','como esta?');
Retorna: "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');
Retorna: "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 enviada 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');
Retorna: 2, porque la cadena "hola" se encuentra en la lista de cadenas, en la posición 2.
- **length**(cadena): retorna la longitud de la cadena enviada como argumento.
Ejemplo: **select length**('Hola');
Retorna: 4.
- **locate**(subcadena,cadena): retorna 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 locate**('o','como le va');
Retorna: 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');
Retorna: 2.
- **locate**(subcadena,cadena,posicioninicial): retorna 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);
Retorna: 1.
select locate('ar','Margarita',3);
Retorna: 5.

- **instr**(cadena,subcadena): retorna 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');
Retorna: 2.
- **lpad**(cadena,longitud,cadenarelleno): retorna la cadena enviada como primer argumento, rellenada por la izquierda con la cadena enviada como tercer argumento hasta que la cadena retornada tenga la longitud especificada como segundo argumento. Si la cadena es más larga, la corta.
Ejemplo: **select lpad**('hola',10,'0');
Retorna: "000000hola".
- **rpadd**(cadena,longitud,cadenarelleno): igual que "lpad" excepto que rellena por la derecha.
- **left**(cadena,longitud): retorna la cantidad (longitud) de caracteres de la cadena comenzando desde la izquierda (primer carácter).
Ejemplo: **select left**('buenos dias',8);
Retorna: "buenos d".
- **right**(cadena,longitud): retorna la cantidad (longitud) de caracteres de la cadena comenzando desde la derecha (último carácter).
Ejemplo: **select right**('buenos dias',8);
Retorna: "nos dias".
- **substring**(cadena,posicion,longitud): retorna 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);
Retorna: "enas".
- **substring**(cadena **from** posicion **for** longitud): variante de "substring(cadena,posicion,longitud)".
Ejemplo: **select substring**('Buenas tardes' **from** 3 **for** 5);
Retorna: "enas".
- **mid**(cadena,posicion,longitud): igual que "substring(cadena,posicion,longitud)".
Ejemplo: **select mid**('Buenas tardes' **from** 3 **for** 5);
Retorna: "enas".
- **substring**(cadena,posicion): retorna la subcadena de la cadena enviada como argumento, empezando desde la posición indicada por el segundo argumento.
Ejemplo: **select substring**('Margarita',4);
Retorna: "garita".
- **substring**(cadena **from** posicion): variante de "substring(cadena,posicion)".
Ejemplo: **select substring**('Margarita' **from** 4);
Retorna: "garita".
- **substring_index**(cadena,delimitador,ocurrencia): retorna la subcadena de la cadena enviada como argumento antes o después de la "ocurrencia" de la cadena enviada como delimitador. Si "ocurrencia" es positiva, retorna la subcadena anterior al delimitador (comienza desde la izquierda); si "ocurrencia" es negativa, retorna la subcadena posterior al delimitador (comienza desde la derecha).
Ejemplo: **select substring_index**('margarita','ar',2);
Retorna: "marg", todo lo anterior a la segunda ocurrencia de "ar".
select substring_index('margarita','ar',-2);
Retorna: "garita", todo lo posterior a la segunda ocurrencia de "ar".
- **ltrim**(cadena): retorna la cadena con los espacios de la izquierda eliminados.
Ejemplo: **select ltrim**(' Hola ');
Retorna: "Hola".
- **rtrim**(cadena): retorna la cadena con los espacios de la derecha eliminados.
Ejemplo: **select rtrim**(' Hola ');
Retorna: " Hola".
- **trim**([[**both**|**leading**|**trailing**] [subcadena] **from**] cadena): retorna 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 ');
Retorna: 'Hola'.
select trim (leading '0' from '00hola00');
Retorna: "hola00".

- select trim (trailing '0' from '00hola00');**
Retorna: "00hola".
- select trim (both '0' from '00hola00');**
Retorna: "hola".
- select trim ('0' from '00hola00');**
Retorna: "hola".
- **replace**(cadena,cadenareemplazo,cadenareemplazar): retorna la cadena con todas las ocurrencias de la subcadena reemplazo por la subcadena a reemplazar.
Ejemplo: **select replace('xxx.mysql.com','x','w');**
Retorna: "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, retorna una cadena vacía.
Ejemplo: **select repeat('hola',3);**
Retorna: "holaholahola".
 - **reverse**(cadena): devuelve la cadena invirtiendo el orden de los caracteres.
Ejemplo: **select reverse('Hola');**
Retorna: "aloH".
 - **insert**(cadena,posicion,longitud,nuevacadena): retorna 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');**
Retorna: "bxxtardes".
 - **lcase**(cadena) y **lower**(cadena): retornan la cadena con todos los caracteres en minúsculas.
Ejemplo: **select lower('HOLA ESTUDIante');**
Retorna: "hola estudiante".
select lcase('HOLA ESTUDIante');
Retorna: "hola estudiante".
 - **ucase**(cadena) y **upper**(cadena): retornan la cadena con todos los caracteres en mayúsculas.
Ejemplo: **select upper('HOLA ESTUDIante');**
Retorna: "HOLA ESTUDIANTE".
select ucase('HOLA ESTUDIante');
Retorna: "HOLA ESTUDIANTE".
 - **strcmp**(cadena1,cadena2): retorna 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');**
Retorna: 1.

59 - Funciones matemáticas

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

Ejemplo: **select 5/0;**

RECUERDE que 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 números. Aquí presentamos algunas.

- **abs**(x): retorna el valor absoluto del argumento "x".
Ejemplo: **select abs(-20);**
Retorna: 20.
- **ceiling**(x): redondea hacia arriba el argumento "x".
Ejemplo: **select ceiling(12.34);**
Retorna: 13.
- **floor**(x): redondea hacia abajo el argumento "x".
Ejemplo: **select floor(12.34);**
Retorna: 12.

- **greatest(x,y,...)**: retorna el argumento de máximo valor.
- **least(x,y,...)**: con dos o más argumentos, retorna el argumento más pequeño.
- **mod(n,m)**: significa "módulo aritmético"; retorna el resto de "n" dividido en "m".
Ejemplos: **select mod(10,3);**
Retorna: 1.
select mod(10,2);
Retorna: 0.
- **%**: devuelve el resto de una división.
Ejemplos: **select 10%3;**
Retorna: 1.
select 10%2;
Retorna: 0.
- **power(x,y)**: retorna el valor de "x" elevado a la "y" potencia.
Ejemplo: **select power(2,3);**
Retorna: 8.
- **rand()**: retorna un valor de coma flotante aleatorio dentro del rango 0 a 1.0.
- **round(x)**: retorna el argumento "x" redondeado al entero más cercano.
Ejemplos: **select round(12.34);**
Retorna: 12.
select round(12.64);
Retorna: 13.
- **sqrt(x)**: devuelve la raíz cuadrada del valor enviado como argumento.
- **truncate(x,d)**: retorna el número "x", truncado a "d" decimales. Si "d" es 0, el resultado no tendrá parte fraccionaria.
Ejemplos: **select truncate(123.4567,2);**
Retorna: 123.45;
select truncate (123.4567,0);
Retorna: 123.

Todas retornan null en caso de error.

60 - Funciones para el uso de fecha y hora

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

- **adddate(fecha, interval expresión tipo)**: retorna la fecha agregándole el intervalo especificado.
Ejemplos: **adddate('2006-10-10',interval 25 day)**
Retorna: "2006-11-04".
adddate('2006-10-10',interval 5 month)
Retorna: "2007-03-10".
- **addtime(expresion1,expresion2)**: agrega expresion2 a expresion1 y retorna el resultado.
- **current_date**: retorna la fecha de hoy con formato "YYYY-MM-DD" o "YYYYMMDD".
- **current_time**: retorna la hora actual con formato "HH:MM:SS" o "HHMMSS".
- **date_add(fecha,interval expresión tipo)** y **date_sub(fecha, interval expresión 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)**
Retorna: "2006-09-10";
date_add('2006-08-10', interval -1 day)
Retorna: "2006-09-09";
date_sub('2006-08-10 18:55:44', interval 2 minute)
Retorna: "2006-08-10 18:53:44";
date_sub('2006-08-10 18:55:44', interval '2:3' minute_second)
Retorna: "2006-08-10 18:52:41".

- **datediff**(fecha1,fecha2): retorna la cantidad de días entre fecha1 y fecha2.
- **dayname**(fecha): retorna el nombre del día de la semana de la fecha.
Ejemplo: **dayname**('2006-08-10')
Retorna: "thursday".
- **dayofmonth**(fecha): retorna el día del mes para la fecha dada, dentro del rango 1 a 31.
Ejemplo: **dayofmonth**('2006-08-10')
Retorna: 10.
- **dayofweek**(fecha): retorna 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')
Retorna: 5, o sea jueves.
- **dayofyear**(fecha): retorna el día del año para la fecha dada, dentro del rango 1 a 366.
Ejemplo: **dayofmonth**('2006-08-10')
Retorna: 222.
- **extract**(tipo from fecha): extrae partes de una fecha.
Ejemplos: **extract(year from '2006-10-10')**,
Retorna: "2006".
extract(year_month from '2006-10-10 10:15:25')
Retorna: "200610".
extract(day_minute from '2006-10-10 10:15:25')
Retorna: "101015";
- **hour**(hora): retorna la hora para el dato dado, en el rango de 0 a 23.
Ejemplo: **hour**('18:25:09');
Retorna "18".
- **minute**(hora): retorna los minutos de la hora dada, en el rango de 0 a 59.
- **monthname**(fecha): retorna el nombre del mes de la fecha dada.
Ejemplo: **monthname**('2006-08-10')
Retorna: "August".
- **month**(fecha): retorna el mes de la fecha dada, en el rango de 1 a 12.
- **now()** y **sysdate()**: retornan la fecha y hora actuales.
- **period_add**(p,n): agrega "n" meses al periodo "p", en el formato "YYMM" o "YYYYMM"; retorna un valor en el formato "YYYYMM". El argumento "p" no es una fecha, sino un año y un mes.
Ejemplo: **period_add**('200608',2)
Retorna: "200610".
- **period_diff**(p1,p2): retorna 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')
Retorna: 6.
- **second**(hora): retorna los segundos para la hora dada, en el rango de 0 a 59.
- **sec_to_time**(segundos): retorna el argumento "segundos" convertido a horas, minutos y segundos.
Ejemplo: **sec_to_time**(90)
Retorna: "1:30".
- **timediff**(hora1,hora2): retorna la cantidad de horas, minutos y segundos entre hora1 y hora2.
- **time_to_sec**(hora): retorna el argumento "hora" convertido en segundos.
- **to_days**(fecha): retorna el número de día (el número de día desde el año 0).
- **weekday**(fecha): retorna 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') ;
Retorna: 3, o sea jueves.
- **year**(fecha): retorna el año de la fecha dada, en el rango de 1000 a 9999.
Ejemplo: **year**('06-08-10')
Retorna: "2006".

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** (días, horas y minutos), **day_second** (días a segundos).

61 - 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 retornará en caso que el primer argumento (una expresión de comparación) sea "verdadero" o "falso"; es decir, si el primer argumento es verdadero, retorna el segundo argumento, sino retorna el tercero.

Veamos el 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").

Veamos otros ejemplos. Queremos mostrar los nombres de los autores y la cantidad de libros de cada uno de ellos; para ello 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, usamos "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 (primer argumento del "if"), coloca "Más de 1" (segundo argumento del "if"), en caso contrario coloca "1" (tercer 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 (primer argumento del "if"), coloca "5 o más" (segundo argumento del "if"), en caso contrario coloca "menos de 5" (tercer argumento del "if").

62 - Funciones de control de flujo (case)

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

Trabajemos con la tabla "libros" de una librería. 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"** retorna **"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;

```

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 **"case when"** que permite condiciones:

```

case when then ... else end

```

Veamos un ejemplo:

```

select editorial,
  case when count(*)=1
    then 1
    else 'mas de uno'
  end as cantidad

```

```
from libros group by editorial;
```

63 - 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. Veamos:

```
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)
);
```

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;
```

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.

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 tipeamos "codigo", MySQL no sabrá si nos referimos al campo "codigo" de "libros" o de "editoriales".

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 primera 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.

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.

64 - Clave foránea

Un campo que se usa para establecer un "join" (unión) con otra tabla en la cual es clave primaria, se denomina "**clave foránea**".

En el ejemplo de la librería en que utilizamos las tablas "libros" y "editoriales" con los campos:

```
libros: codigo (clave primaria), titulo, autor, codigoeditorial, precio, cantidad y
editoriales: codigo (clave primaria), nombre.
```

El campo "codigoeditorial" de "libros" es una clave foránea, se emplea para establecer una **relacion** de la tabla "libros" con "editoriales" y es clave primaria en "editoriales" con el nombre "codigo".

Cuando alteramos una tabla, debemos tener cuidado con las claves foráneas. Si modificamos el tipo, longitud o atributos de una clave foránea, se puede romper la relación.

Las claves foráneas y las claves primarias deben ser del mismo tipo para poder relacionarlas. Si modificamos una, debemos modificar la otra para que los valores se correspondan.

65 - 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".

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".

Un "left join" puede tener clausula "where" que restrinja 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".

66 - 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.

67 - Varias tablas (cross join)

"**cross join**" retorna 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.

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 monto 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');
```

68 - Varias tablas (natural join)

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

Tenemos las tablas "libros" y "editoriales" de una librería. Las tablas tienen las siguientes estructuras:

- libros: codigo (clave primaria), titulo, autor, *codigoeditorial*, precio.
- editoriales: *codigoeditorial*(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;
```

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.

69 - Varias tablas (inner join - straight join)

"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.

70 - join, group by y funciones de agrupamiento.

Podemos usar "**group by**" y las funciones de agrupamiento con "**join**".

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 usamos:

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

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

71 - 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 table 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)
);

```

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.

Note que especificamos a qué tabla pertenece el campo "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.codigo=l.codigo;

```

72 - 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;

```


73 - Variables de usuario.

Cuando buscamos un valor con las funciones de agrupamiento, por ejemplo "**max()**", la consulta nos devuelve el máximo valor de un campo de una tabla, pero no nos muestra los valores de otros campos del mismo registro. Por ejemplo, queremos saber todos los datos del libro con mayor precio de la tabla "libros" de una librería, tipeamos:

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

Para obtener todos los datos del libro podemos emplear una variable para almacenar el precio más alto:

```
select @mayorprecio:=max(precio) from libros;
```

y luego mostrar todos los datos de dicho libro empleando la variable anterior:

```
select * from libros where precio=@mayorprecio;
```

Es decir, guardamos en la variable el precio más alto y luego, en otra sentencia, mostramos los datos de todos los libros cuyo precio es igual al valor de la variable.

Las variables nos permiten almacenar un valor y recuperarlo más adelante, de este modo se pueden usar valores en otras sentencias.

Las variables de usuario son específicas de cada conexión, es decir, una variable definida por un cliente no puede ser vista ni usada por otros clientes y son liberadas automáticamente al abandonar la conexión.

Las variables de usuario comienzan con "@" (arroba) seguido del nombre (sin espacios), dicho nombre puede contener cualquier carácter.

Para almacenar un valor en una variable se coloca "==" (operador de asignación) entre la variable y el valor a asignar.

En el ejemplo, mostramos todos los datos del libro con precio más alto, pero, si además, necesitamos el nombre de la editorial podemos emplear un "**join**":

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

La utilidad de las variables consiste en que almacenan valores para utilizarlos en otras consultas.

Por ejemplo, queremos ver todos los libros de la editorial que tenga el libro más caro. Debemos buscar el precio más alto y almacenarlo en una variable, luego buscar el nombre de la editorial del libro con el precio igual al valor de la variable y guardarlo en otra variable, finalmente buscar todos los libros de esa editorial:

```
select @mayorprecio:=max(precio) from libros;  
  
select @editorial:=e.nombre from libros as l  
join editoriales as e on l.codigoeditorial=e.codigo  
where precio=@mayorprecio;  
  
select l.titulo,l.autor,e.nombre from libros as l  
join editoriales as e on l.codigoeditorial=e.codigo  
where e.nombre=@editorial;
```

74 - Crear tabla a partir de otra (create - insert)

Tenemos la tabla "libros" de una librería y queremos crear una tabla llamada "editoriales" que contenga los nombres de las editoriales.

La tabla "libros" tiene esta estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(40) not null,
- autor: varchar(30),
- editorial: varchar(20) not null,
- precio: decimal(5,2) unsigned,

- clave primaria: codigo.

La tabla "editoriales", que no existe, debe tener la siguiente estructura:

- nombre: nombre de la editorial.

La tabla libros contiene varios registros.

Para guardar en "editoriales" los nombres de las editoriales, podemos hacerlo en 3 pasos:

1º paso: crear la tabla "editoriales":

```
create table editoriales(
    nombre varchar(20)
);
```

2º paso: realizar la consulta en la tabla "libros" para obtener los nombres de las distintas editoriales:

```
select distinct editorial as nombre from libros;
```

Obteniendo una salida como la siguiente:

```
editorial
-----
Emece
Paidos
Planeta
```

3º paso: insertar los registros necesarios en la tabla "editoriales":

```
insert into editoriales (nombre) values('Emece');
insert into editoriales (nombre) values('Paidos');
insert into editoriales (nombre) values('Planeta');
```

Pero existe otra manera simplificando los pasos. Podemos crear la tabla "editoriales" con los campos necesarios consultando la tabla "libros" y en el mismo momento insertar la información:

```
create table editoriales
select distinct editorial as nombre
from libros;
```

La tabla "editoriales" se ha creado con el campo llamado "nombre" seleccionado del campo "editorial" de "libros".

Entonces, se realiza una consulta de la tabla "libros" y anteponiendo "**create table ...**" se ingresa el resultado de dicha consulta en la tabla "editoriales" al momento de crearla.

Si seleccionamos todos los registros de la tabla "editoriales" aparece lo siguiente:

```
nombre
-----
Emece
Paidos
Planeta
```

Si visualizamos la estructura de "editoriales" con "describe editoriales" vemos que el campo "nombre" se creó con el mismo tipo y longitud del campo "editorial" de "libros".

También podemos crear una tabla a partir de una consulta cargando los campos con los valores de otra tabla y una columna calculada. Veamos un ejemplo.

Tenemos la misma tabla "libros" y queremos crear una tabla llamada "librosporeditorial" que contenga la cantidad de libros de cada editorial.

La tabla "cantidadporeditorial", que no está creada, debe tener la siguiente estructura:

- nombre: nombre de la editorial,
- cantidad: cantidad de libros.

Podemos lograrlo en 3 pasos:

1º paso: crear la tabla "cantidadporeditorial":

```
create table editoriales(
    nombre varchar(20),
    cantidad smallint
);
```

2º paso: realizar la consulta en la tabla "libros" para obtener la cantidad de libros de cada editorial agrupando por "editorial" y calculando la cantidad con "count()":

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

Obteniendo una salida como la siguiente:

nombre	cantidad
Emece	3
Paidos	4
Planeta	2

3º paso: insertar los registros necesarios en la tabla "editoriales":

```
insert into cantidadporeditorial values('Emece',3);
insert into cantidadporeditorial values('Paidos',4);
insert into cantidadporeditorial values('Planeta',2);
```

Pero existe otra manera *simplificando los pasos*. Podemos crear la tabla "cantidadporeditorial" con los campos necesarios consultando la tabla "libros" y en el mismo momento insertar la información:

```
create table cantidadporeditorial
select editorial as nombre,count(*) as cantidad from libros
group by editorial;
```

La tabla "cantidadporeditorial" se ha creado con el campo llamado "nombre" seleccionado del campo "editorial" de "libros" y con el campo "cantidad" con el valor calculado con count() de la tabla "libros".

Entonces, se realiza una consulta de la tabla "libros" y anteponiendo "create table ..." se ingresa el resultado de dicha consulta en la tabla "cantidadporeditorial" al momento de crearla.

Si seleccionamos todos los registros de la tabla "cantidadporeditorial" aparece lo siguiente:

nombre	cantidad
Emece	3
Paidos	4
Planeta	2

Si visualizamos la estructura de "cantidadporeditorial" con "describe cantidadporeditorial", vemos que el campo "nombre" se creó con el mismo tipo y longitud del campo "editorial" de "libros" y el campo "cantidad" se creó como "bigint".

75 - Crear tabla a partir de otras (create - insert - join)

Tenemos las tablas "libros" y "editoriales" y queremos crear una tabla llamada "cantidadporeditorial" que contenga la cantidad de libros de cada editorial.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(40) not null,
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,

- precio: decimal(5,2) unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene esta estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

Las tablas "libros" y "editoriales" contienen varios registros.

La tabla "cantidadporeditorial", que no existe, debe tener la siguiente estructura:

- nombre: nombre de la editorial,
- cantidad: cantidad de libros.

Podemos guardar en la tabla "cantidadporeditorial" la cantidad de libros de cada editorial en 3 pasos:

1º paso: crear la tabla "cantidadporeditorial":

```
create table cantidadporeditorial(
    nombre varchar(20),
    cantidad smallint
);
```

2º paso: realizar la consulta en la tabla "libros" y "editoriales", con un "join" para obtener la cantidad de libros de cada editorial agrupando por el nombre de la editorial y calculando la cantidad con "count()":

```
select e.nombre,count(*) from libros as l
join editoriales as e on l.codigoeditorial=e.codigo
group by e.nombre;
```

Obteniendo una salida como la siguiente:

nombre	cantidad
Emece	3
Paidos	4
Planeta	2

3º paso: insertar los registros necesarios en la tabla "editoriales":

```
insert into editoriales values('Emece',3);
insert into editoriales values('Paidos',4);
insert into editoriales values('Planeta',2);
```

Pero existe otra manera simplificando los pasos. Podemos crear la tabla "cantidadporeditorial" con los campos necesarios consultando la tabla "libros" y "editoriales" y en el mismo momento insertar la información:

```
create table cantidadporeditorial
select e.nombre,count(*) as cantidad from libros as l
join editoriales as e on l.codigoeditorial=e.codigo
group by e.nombre;
```

La tabla "cantidadporeditorial" se ha creado con el campo llamado "nombre" seleccionado del campo "nombre" de "editoriales" y con el campo "cantidad" con el valor calculado con count() de la tabla "libros".

Entonces, se realiza una consulta de la tabla "libros" y "editoriales" (con un "join") anteponiendo "create table ..." se ingresa el resultado de dicha consulta en la tabla "cantidadporeditorial" al momento de crearla.

Si seleccionamos todos los registros de la tabla "cantidadporeditorial" aparece lo siguiente:

nombre	cantidad
Emece	3
Paidos	4
Planeta	2

Si visualizamos la estructura de "cantidadporeditorial", vemos que el campo "nombre" se creó con el mismo tipo y longitud del campo "nombre" de "editoriales" y el campo "cantidad" se creó como "bigint".

76 - Insertar datos en una tabla buscando un valor en otra (insert - select)

Trabajamos con las tablas "libros" y "editoriales" de una librería.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(40) not null,
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- precio: decimal(5,2) unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene la siguiente estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- domicilio: varchar(30),
- clave primaria: codigo.

Ambas tablas contienen registros. La tabla "editoriales" contiene los siguientes registros:

- 1, Planeta, San Martin 222
- 2, Emece, San Martin 590
- 3, Paidos, Colon 245

Queremos ingresar en "libros", el siguiente libro:

Harry Potter y la piedra filosofal, J.K. Rowling, Emece, 45.90.

pero no recordamos el código de la editorial "Emece".

Podemos lograrlo en 2 pasos:

1º paso: consultar en la tabla "editoriales" el código de la editorial "Emece":

```
select codigo from editoriales where nombre='Emece';
```

nos devuelve el valor "2".

2º paso: ingresar el registro en "libros":

```
insert into libros (titulo,autor,codigoeditorial,precio)
values('Harry Potter y la piedra filosofal','J.K.Rowling',2,45.90);
```

O en una sola consulta podemos realizar la consulta del código de la editorial al momento de la inserción:

```
insert into libros (titulo,autor,codigoeditorial,precio)
select 'Harry Potter y la camara secreta','J.K.Rowling',codigo,45.90
from editoriales
where nombre='Emece';
```

Entonces, para realizar una inserción y al mismo tiempo consultar un valor en otra tabla, colocamos "**insert into**" junto al nombre de la tabla ("libros") y los campos a insertar y luego un "**select**" en el cual disponemos todos los valores, excepto el valor que desconocemos, en su lugar colocamos el nombre del campo a consultar ("codigo"), luego se continúa con la consulta indicando la tabla de la cual extraemos el código ("editoriales") y la condición, en la cual damos el "nombre" de la editorial para que localice el código correspondiente.

El registro se cargará con el valor de código de la editorial "Emece".

Si la consulta no devuelve ningún valor, porque buscamos el código de una editorial que no existe en la tabla "editoriales", aparece un mensaje indicando que no se ingresó ningún registro. Por ejemplo:

```
insert into libros (titulo,autor,codigoeditorial,precio)
select 'Cervantes y el quijote','Borges',codigo,35
from editoriales
where nombre='Plaza & Janes';
```

Hay que tener cuidado al establecer la condición en la consulta, el "**insert**" ingresará tantos registros como filas retorne la consulta. Si la consulta devuelve 2 filas, se insertarán 2 filas en el "**insert**". Por ello, el valor de la condición (o condiciones), por el cual se busca, debe retornar un sólo registro.

Veamos un ejemplo. Queremos ingresar el siguiente registro:

Harry Potter y la camara secreta, J.K. Rowling,54.

pero no recordamos el código de la editorial ni su nombre, sólo sabemos que su domicilio es en calle "San Martin". Si con un **"select"** localizamos el código de todas las editoriales que tengan sede en "San Martin", el resultado retorna 2 filas, porque hay 2 editoriales en esa dirección ("Planeta" y "Emece"). Tipeemos la sentencia:

```
insert into libros (titulo,autor,codigoeditorial,precio)
select 'Harry Potter y la camara secreta','J.K. Rowling',codigo,54
from editoriales
where domicilio like 'San Martin%';
```

Se ingresarán 2 registros con los mismos datos, excepto el código de la editorial.

Recuerde entonces, el valor de la condición (condiciones), por el cual se busca el dato desconocido en la consulta debe retornar un sólo registro.

También se pueden consultar valores de varias tablas incluyendo en el **"select"** un **"join"**.

77 - Insertar registros con valores de otra tabla (insert - select)

Tenemos las tabla "libros" y "editoriales" creadas. La tabla "libros" contiene registros; "editoriales", no.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- editorial: varchar(20),
- precio: decimal(5,2) unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene la siguiente estructura:

- nombre: varchar(20).

Queremos insertar registros en la tabla "editoriales", los nombres de las distintas editoriales de las cuales tenemos libros. Podemos lograrlo en 2 pasos, con varias sentencias:

1º paso: consultar los nombres de las distintas editoriales de "libros":

```
select distinct editorial from libros;
```

obteniendo una salida como la siguiente:

```
editorial
-----
Emece
Paidos
Planeta
```

2º paso: insertar los registros uno a uno en la tabla "editoriales":

```
insert into editoriales (nombre) values('Emece');
insert into editoriales (nombre) values('Paidos');
insert into editoriales (nombre) values('Planeta');
```

O podemos lograrlo en un solo paso, realizando el **"insert"** y el **"select"** en una misma sentencia:

```
insert into editoriales (nombre)
select distinct editorial from libros;
```

Entonces, se puede insertar registros en una tabla con la salida devuelta por una consulta; para ello escribimos la consulta y le anteponeamos **"insert into"**, el nombre de la tabla en la cual ingresaremos los registros y los campos que se cargarán.

También podemos crear una tabla llamada "cantidadporeditorial":

```
create table cantidadporeditorial(
    nombre varchar(20),
    cantidad smallint unsigned
);
```

e ingresar registros a partir de una consulta a la tabla "libros":

```
insert into cantidadporeditorial (nombre,cantidad)
select editorial,count(*) as cantidad
from libros
group by editorial;
```

Si los campos presentados entre paréntesis son menos (o más) que las columnas devueltas por la consulta, aparece un mensaje de error y la sentencia no se ejecuta.

78 - Insertar registros con valores de otra tabla (insert - select - join)

Tenemos las tabla "libros" y "editoriales", que contienen registros, y la tabla "cantidadporeditorial", que no contiene registros. La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- precio: decimal(5,2) unsigned,
- clave primaria: codigo.

La tabla "editoriales":

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

La tabla "cantidadporeditorial":

- nombre: varchar(20),
- cantidad: smallint unsigned.

Queremos insertar registros en la tabla "cantidadporeditorial", los nombres de las distintas editoriales de las cuales tenemos libros y la cantidad de libros de cada una de ellas.

Podemos lograrlo en 2 pasos:

1º paso: consultar con un **"join"** los nombres de las distintas editoriales de "libros" y la cantidad:

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

obteniendo una salida como la siguiente:

editorial	cantidad
Emece	3
Paidos	1
Planeta	1
Plaza & Janes	0

2º paso: insertar los registros uno a uno en la tabla "cantidadporeditorial":

```

insert into cantidadporeditorial values('Emece',3);
insert into cantidadporeditorial values('Paidos',1);
insert into cantidadporeditorial values('Planeta',1);
insert into cantidadporeditorial values('Plaza & Janes',0);

```

O podemos lograrlo en un solo paso, realizando el "insert" y el "select" en una misma sentencia:

```

insert into cantidadporeditorial
select e.nombre,count(l.codigoeditorial)
from editoriales as e
left join libros as l
on e.codigo=l.codigoeditorial
group by e.nombre;

```

Entonces, se puede insertar registros en una tabla con la salida devuelta por una consulta que incluya un "join" o un "left join"; para ello escribimos la consulta y le anteponeamos "insert into", el nombre de la tabla en la cual ingresaremos los registros y los campos que se cargarán (si se ingresan todos los campos no es necesario listarlos).

Recuerde que la cantidad de columnas devueltas en la consulta debe ser la misma que la cantidad de campos a cargar en el "insert".

79 - Actualizar datos con valores de otra tabla (update)

Tenemos la tabla "libros" en la cual almacenamos los datos de los libros de nuestra biblioteca y la tabla "editoriales" que almacena el nombre de las distintas editoriales y sus códigos.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene esta estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

Ambas tablas contienen registros.

Queremos unir los datos de ambas tablas en una sola: "libros", es decir, alterar la tabla "libros" para que almacene el nombre de la editorial y eliminar la tabla "editoriales".

En primer lugar debemos alterar la tabla "libros", vamos a agregarle un campo llamado "editorial" en el cual guardaremos el nombre de la editorial.

```
alter table libros add editorial varchar(20);
```

La tabla "libros" contiene un nuevo campo "editorial" con todos los registros con valor "null".

Ahora debemos actualizar los valores para ese campo.

Podemos hacerlo en 2 pasos:

1º paso: consultamos los códigos de las editoriales:

```

select codigo,nombre
from editoriales;

```

obtenemos una salida similar a la siguiente:

codigo	nombre
1	Planeta
2	Emece
3	Paidos

2º paso: comenzamos a actualizar el campo "editorial" de los registros de "libros" uno a uno:


```

update libros set editorial='Planeta'
where codigoeditorial=1;
update libros set editorial='Emece'
where codigoeditorial=2;
update libros set editorial='Paidos'
where codigoeditorial=3;

```

... con cada editorial...

Luego, eliminamos el campo "codigoeditorial" de "libros" y la tabla "editoriales".

Pero podemos simplificar la tarea actualizando el campo "editorial" de todos los registros de la tabla "libros" al mismo tiempo que realizamos el **"join"** (*paso 1 y 2 en una sola sentencia*):

```

update libros
join editoriales
on libros.codigoeditorial=editoriales.codigo
set libros.editorial=editoriales.nombre;

```

Luego, eliminamos el campo "codigoeditorial" de "libros" con **"alter table"** y la tabla "editoriales" con **"drop table"**.

Entonces, se puede actualizar una tabla con valores de otra tabla. Se coloca **"update"** junto al nombre de la tabla a actualizar, luego se realiza el **"join"** y el campo por el cual se enlazan las tablas y finalmente se especifica con **"set"** el campo a actualizar y su nuevo valor, que es el campo de la otra tabla con la cual se enlazó.

80 - Actualización en cascada (update - join)

Tenemos la tabla "libros" en la cual almacenamos los datos de los libros de nuestra biblioteca y la tabla "editoriales" que almacena el nombre de las distintas editoriales y sus códigos.

Las tablas tienen las siguientes estructuras:

```

create table libros(
    codigo int unsigned auto_increment,
    titulo varchar(30),
    autor varchar(30),
    codigoeditorial tinyint unsigned,
    precio decimal(5,2) unsigned,
    primary key(codigo)
);

create table editoriales(
    codigo tinyint unsigned auto_increment,
    nombre varchar(20),
    primary key(codigo)
);

```

Ambas tablas contienen registros.

Queremos modificar el código de la editorial "Emece" a "9" y también todos los "codigoeditorial" de los libros de dicha editorial. *Podemos hacerlo en 3 pasos:*

1) buscar el código de la editorial "Emece":

```

select * from editoriales
where nombre='Emece';

```

recordamos el valor devuelto (valor 2) o lo almacenamos en una variable;

2) actualizar el código en la tabla "editoriales":

```

update editoriales
set codigo=9
where nombre='Emece';

```

3) y finalmente actualizar todos los libros de dicha editorial:

```

update libros
set codigoeditorial=9
where codigoeditorial=2;

```

O podemos hacerlo en una sola sentencia:

```

update libros as l
join editoriales as e
on l.codigoeditorial=e.codigo
set l.codigoeditorial=9, e.codigo=9
where e.nombre='Emece';

```

El cambio se realizó en ambas tablas.

Si modificamos algún dato de un registro que se encuentra en registros de otras tablas (generalmente campos que son clave ajena) debemos modificar también los registros de otras tablas en los cuales se encuentre ese dato (generalmente clave primaria). Podemos realizar la actualización en cascada (es decir, en todos los registros de todas las tablas que contengan el dato modificado) en una sola sentencia, combinando **"update"** con **"join"** y seteando los campos involucrados de todas las tablas.

81 - Borrar registros consultando otras tablas (delete - join)

Tenemos la tabla "libros" en la cual almacenamos los datos de los libros de nuestra biblioteca y la tabla "editoriales" que almacena el nombre de las distintas editoriales y sus códigos.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene esta estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

Ambas tablas contienen registros.

Queremos eliminar todos los libros de la editorial "Emece" pero no recordamos el código de dicha editorial.

Podemos hacerlo en 2 pasos:

1º paso: consultamos el código de la editorial "Emece":

```

select codigo
from editoriales
where nombre='Emece';

```

recordamos el valor devuelto (valor 2) o lo almacenamos en una variable.

2º paso: borramos todos los libros con código de editorial "2":

```

delete libros
where codigoeditorial=2;

```

O podemos realizar todo en un solo paso:

```

delete libros
from libros
join editoriales
on libros.codigoeditorial=editoriales.codigo
where editoriales.nombre='Emece';

```

Es decir, usamos **"delete"** junto al nombre de la tabla de la cual queremos eliminar registros, luego realizamos el **"join"** correspondiente nombrando las tablas involucradas y agregamos la condición **"where"**.

82 - Borrar registros buscando coincidencias en otras tablas (delete - join)

Tenemos la tabla "libros" en la cual almacenamos los datos de los libros de nuestra biblioteca y la tabla "editoriales" que almacena el nombre de las distintas editoriales y sus códigos.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene esta estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

Ambas tablas contienen registros.

Queremos eliminar todos los libros cuyo código de editorial no exista en la tabla "editoriales".

Podemos hacerlo en 2 pasos:

1º paso: realizamos un **left join** para ver qué "codigoeditorial" en "libros" no existe en "editoriales":

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

recordamos el valor de los códigos de libro devueltos (valor 5) o lo almacenamos en una variable.

2º paso: borramos todos los libros mostrados en la consulta anterior (uno solo, con código 5):

```
delete libros
where codigo=5;
```

O podemos realizar la eliminación en una sola consulta en el mismo momento que realizamos el "left join":

```
delete libros
from libros
left join editoriales
on libros.codigoeditorial=editoriales.codigo
where editoriales.codigo is null;
```

Es decir, usamos "**delete**" junto al nombre de la tabla de la cual queremos eliminar registros, luego realizamos el "**left join**" correspondiente nombrando las tablas involucradas y agregamos la condición "**where**" para que seleccione solamente los libros cuyo código de editorial no se encuentre en "editoriales".

Ahora queremos eliminar todas las editoriales de las cuales no haya libros:

```
delete editoriales
from editoriales
left join libros
on libros.codigoeditorial=editoriales.codigo
where libros.codigo is null;
```

83 - Borrar registros en cascada (delete - join)

Tenemos la tabla "libros" en la cual almacenamos los datos de los libros de nuestra biblioteca y la tabla "editoriales" que almacena el nombre de las distintas editoriales y sus códigos.

La tabla "libros" tiene la siguiente estructura:

- codigo: int unsigned auto_increment,
- titulo: varchar(30),
- autor: varchar(30),
- codigoeditorial: tinyint unsigned,
- clave primaria: codigo.

La tabla "editoriales" tiene esta estructura:

- codigo: tinyint unsigned auto_increment,
- nombre: varchar(20),
- clave primaria: codigo.

Ambas tablas contienen registros.

La librería ya no trabaja con la editorial "Emece", entonces quiere eliminar dicha editorial de la tabla "editoriales" y todos los libros de "libros" de esta editorial.

Podemos hacerlo en 2 pasos:

1º paso: buscar el código de la editorial "Emece" y almacenarlo en una variable:

```
select @valor:= codigo from editoriales
where nombre='Emece';
```

2º paso: eliminar dicha editorial de la tabla "editoriales":

```
delete editoriales
where codigo=@valor;
```

3º paso: eliminar todos los libros cuyo código de editorial sea igual a la variable:

```
delete libros
where codigoeditorial=@valor;
```

O podemos hacerlo en una sola consulta:

```
delete libros,editoriales
from libros
join editoriales on libros.codigoeditorial=editoriales.codigo
where editoriales.nombre='Emece';
```

La sentencia anterior elimina de la tabla "editoriales" la editorial "Emece" y de la tabla "libros" todos los registros con código de editorial correspondiente a "Emece".

Es decir, podemos realizar la eliminación de registros de varias tablas (en cascada) empleando "**delete**" junto al nombre de las tablas de las cuales queremos eliminar registros y luego del correspondiente "**join**" colocar la condición "**where**" que afecte a los registros a eliminar.

84 - Chequear y reparar tablas (check - repair)

Para chequear el estado de una tabla usamos "**check table**":

```
check table libros;
```

"**check table**" chequea si una o más tablas tienen errores. Esta sentencia devuelve la siguiente información: en la columna "**Table**" muestra el nombre de la tabla; en "**Op**" muestra siempre "**check**"; en "**Msg_type**" muestra "**status**", "**error**", "**info**" o "**warning**" y en "**Msg_text**" muestra un mensaje, generalmente es "**OK**".

Existen distintas opciones de chequeo de una tabla, si no se especifica, por defecto es "**medium**".

Los tipos de chequeo son:

- **quick**: no controla los enlaces incorrectos de los registros.
- **fast**: controla únicamente las tablas que no se han cerrado en forma correcta.

- **changed:** únicamente controla las tablas que se han cambiado desde el último chequeo o que no se cerraron correctamente.
- **medium:** controla los registros para verificar que los enlaces borrados están bien.
- **extended:** realiza una búsqueda completa para todas las claves de cada registro.

Se pueden combinar las opciones de control, por ejemplo, realizamos un chequeo rápido de la tabla "libros" y verificamos si se cerró correctamente:

```
check table libros fast quick;
```

Para reparar una tabla corrupta usamos "**repair table**":

```
repair table libros;
```

"**repair table**" puede recuperar los datos de una tabla.

Devuelve la siguiente información: en la columna "**Table**" muestra el nombre de la tabla; en "**Op**" siempre muestra "**repair**"; en "**Msg_type**" muestra "**status**", "**error**", "**info**" o "**warning**" y en "**Msg_text**" muestra un mensaje que generalmente es "**OK**".

85 - Encriptación de datos (encode - decode)

Las siguientes funciones encriptan y desencriptan valores.

Si necesitamos almacenar un valor que no queremos que se conozca podemos encriptar dicho valor, es decir, transformarlo a un código que no pueda leerse.

Con "**encode**" se encripta una cadena. La función recibe 2 argumentos: el primero, la cadena a encriptar; el segundo, una cadena usada como contraseña para luego desencriptar:

```
select encode('feliz','dia');
```

El resultado es una cadena binaria de la misma longitud que el primer argumento.

Con "**decode**" desencriptamos una cadena encriptada con "**encode**". Esta función recibe 2 argumentos: el primero, la cadena a desencriptar; el segundo, la contraseña:

```
Select decode('§iY7','dia');
```

Si la cadena de contraseña es diferente a la ingresada al encriptar, el resultado será incorrecto.

Retomamos la tabla "usuarios" que constaba de 2 campos:

- nombre del usuario: varchar de 30;
- clave: varchar de 10

Podemos ingresar registros encriptando la clave:

```
insert into usuarios values ('MarioPerez',encode('Marito','octubre'));
```

La forma más segura es no transferir la contraseña a través de la conexión, para ello podemos almacenar la clave en una variable y luego insertar la clave encriptada:

```
select @clave:=encode('RealMadrid','ganador');
```

```
insert into usuarios values ('MariaGarcia',@clave);
```

Veamos los registros ingresados:

```
select * from usuarios;
```

Desencriptamos la clave del usuario "MarioPerez":

```
select decode(clave,'octubre') from usuarios  
where nombre='MarioPerez';
```

Desencriptamos la clave del usuario "MariaGarcia":

```
select decode(clave,'ganador') from usuarios  
where nombre='MariaGarcia';
```

86 – Clave Foránea (foreign key)

Estrictamente hablando, para que un campo sea una clave foránea, éste necesita ser definido como tal al momento de crear una tabla. Se pueden definir claves foráneas en cualquier tipo de tabla de MySQL, pero únicamente tienen sentido cuando se usan tablas del tipo InnoDB.

A partir de la versión 3.23.43b, se pueden definir restricciones de claves foráneas con el uso de tablas InnoDB. InnoDB es el primer tipo de tabla que permite definir estas restricciones para garantizar la integridad de los datos.

Para trabajar con claves foráneas, necesitamos hacer lo siguiente:
Crear ambas tablas del tipo InnoDB.

Usar la sintaxis `FOREIGN KEY(campo_fk) REFERENCESnombre_tabla (nombre_campo)`

Crear un índice en el campo que ha sido declarado clave foránea.

InnoDB no crea de manera automática índices en las claves foráneas o en las claves referenciadas, así que debemos crearlos de manera explícita. Los índices son necesarios para que la verificación de las claves foráneas sea más rápida. A continuación se muestra como definir las dos tablas de ejemplo con una clave foránea.

```
CREATE TABLE cliente
(
id_cliente INT NOT NULL,
nombre VARCHAR(30),
PRIMARY KEY (id_cliente)
) TYPE = INNODB;
```

```
CREATE TABLE venta
(
id_factura INT NOT NULL,
id_cliente INT NOT NULL,
cantidad INT,
PRIMARY KEY(id_factura),
INDEX (id_cliente),
FOREIGN KEY (id_cliente) REFERENCES
cliente(id_cliente)
) TYPE = INNODB;
```

La sintaxis completa de una restricción de clave foránea es la siguiente:

```
[CONSTRAINT símbolo] FOREIGN KEY (nombre_columna,
...)
REFERENCES nombre_tabla
(nombre_columna, ...)
```

```
[ON DELETE {CASCADE | SET NULL
| NO ACTION
| RESTRICT}]
[ON UPDATE {CASCADE | SET NULL
| NO ACTION
| RESTRICT}]
```

Las columnas correspondientes en la clave foránea y en la clave referenciada deben tener tipos de datos similares para que puedan ser comparadas sin la necesidad de hacer una conversión de tipos.

El tamaño y el signo de los tipos enteros debe ser el mismo. En las columnas de tipo carácter, el tamaño no tiene que ser el mismo necesariamente.

Si MySQL da un error cuyo número es el 1005 al momento de ejecutar una sentencia `CREATE TABLE`, y el mensaje de error se refiere al número 150, la creación de la tabla falló porque la restricción de la clave foránea no se hizo de la manera adecuada.

De la misma manera, si falla una sentencia `ALTER TABLE` y se hace referencia al error número 150, esto significa que la definición de la restricción de la clave foránea no se hizo adecuadamente.

A partir de la versión 4.0.13 de MySQL, se puede usar la sentencia SHOW INNODB STATUS para ver una explicación detallada del último error que se generó en relación a la definición de una clave foránea.

Si en una tabla, un registro contiene una clave foránea con un valor NULO, significa que no existe ninguna relación con otra tabla.

A partir de la versión 3.23.50, se pueden agregar restricciones de clave foránea a una tabla con el uso de la sentencia ALTER TABLE. La sintaxis es:

```
ALTER TABLE nombre_tabla ADD [CONSTRAINT símbolo]
FOREIGN KEY(...)
REFERENCES otra_tabla(...) [acciones_ON_DELETE]
[acciones_ON_UPDATE]
```

Por ejemplo, la creación de la clave foránea en la tabla venta que se mostró anteriormente pudo haberse hecho de la siguiente manera con el uso de una sentencia ALTER TABLE:

```
CREATE TABLE venta
(
id_factura INT NOT NULL,
id_cliente INT NOT NULL,
cantidad INT,
PRIMARY KEY(id_factura),
INDEX (id_cliente)
) TYPE = INNODB;
```

```
ALTER TABLE venta ADD FOREIGN KEY(id_cliente)
REFERENCES cliente(id_cliente);
```

En las versiones 3.23.50 y menores no deben usarse las sentencias ALTER TABLE o CREATE INDEX en tablas que ya tienen definidas restricciones de claves foráneas o bien, que son referenciadas en restricciones de claves foráneas: cualquier sentencia ALTER TABLE elimina todas las restricciones de claves foráneas definidas para la tabla.

No debe usarse una sentencia ALTER TABLE en una tabla que está siendo referenciada, si se quiere modificar el esquema de la tabla, se recomienda eliminar la tabla y volverla a crear con el nuevo esquema. Cuando MySQL hace un ALTER TABLE, puede que use de manera interna un RENAME TABLE, y por lo tanto, se confundan las restricciones de clave foránea que se refieren a la tabla. Esta restricción aplica también en el caso de la sentencia CREATE INDEX, ya que MySQL la procesa como un ALTER TABLE.

Cuando se ejecute un script para cargar registros en una base de datos, es recomendable agregar las restricciones de claves foráneas vía un ALTER TABLE. De esta manera no se tiene el problema de cargar los registros en las tablas de acuerdo a un orden lógico (las tablas referenciadas deberían ir primero)