Guía de MySQL

Contenidos:

1.- Introducción:

SQL: Structure Query Language (Lenguaje de Consulta Estructurado). MySQL es un sistema de administración de bases de datos (Database Management System, DBMS) para bases de datos relacionales open source y está haciendo un competidor cada vez más directo de gigantes en la materia de las bases de datos como ORACLE. Así que, MySQL es una aplicación que permite gestionar archivos llamados de bases de datos.

Existen muchos tipos de bases de datos, desde un simple archivo hasta sistemas relacionales orientados a objetos. MySQL, como base de datos relacional, utiliza múltiples tablas para almacenar y organizar la información.

MySQL fue escrito en C y C++ y destaca por su gran adaptación a diferentes entornos de desarrollo, permitiendo su interacción con los lenguajes de programación más utilizados como PHP, Perl y Java y su integración en distintos sistemas operativos.

También es muy destacable, la condición de open source de MySQL, que hace que su utilización sea gratuita e incluso se pueda modificar con total libertad, pudiendo descargar su código fuente. Esto ha favorecido muy positivamente en su desarrollo y continuas actualizaciones, para hacer de MySQL una de las herramientas más utilizadas por los programadores orientados a Internet.

Este permite crear base de datos y tablas, insertar datos, modificarlos, eliminarlos, ordenarlos, hacer consultas y realizar muchas operaciones. Ingresando instrucciones en la línea de comandos o embebidas en un lenguaje como PHP nos comunicamos con el servidor.

2.- Creación de una Base de datos: CREATE DATABASE

Para comenzar a trabajar con una base de datos debemos conocer en primer lugar la definición: Una base de datos es un gran contenedor de información almacenada de forma organizada tal que se pueda evitar la redundancia de datos y se pueda establecer relaciones entre los mismos. La base de datos contendrá las tablas identificadas con un nombre único donde se guardarán los datos también identificados de forma única.

Para crear una base de datos utilizamos la siguiente sintaxis SQL:

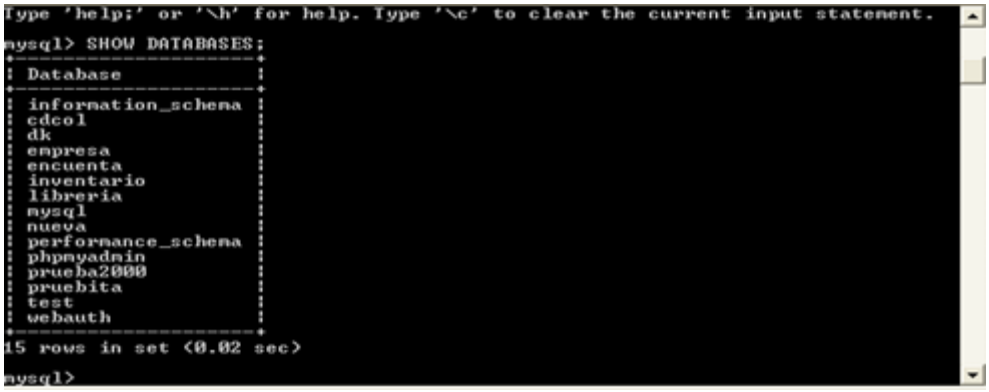
```
CREATE DATABASE tienda;
```

Con este query estamos creando la base de datos “tienda”. Nótese que las palabras “CREATE” y “DATABASE” están escritas en mayúsculas pues semánticamente es lo correcto. Se debe distinguir las palabras reservadas de SQL y las que como programadores estamos asignando a nuestros elementos. Al final de cada instrucción debe escribirse “;” ya que este símbolo garantiza el fin de la instrucción.

Una vez creada la BD podemos pedirle a MySQL que nos muestre todas las BD existentes. Para eso utilizamos la siguiente instrucción:

```
SHOW DATABASES;
```

Debe mostrarse una pantalla con el listado de las bd existentes para el momento tal y como lo muestra la siguiente imagen:



Bases de datos existentes en el sistema.

Para comenzar a trabajar con una base de datos se debe seleccionar para garantizar que todas las instrucciones se realicen dentro de la misma. Con la siguiente instrucción podemos hacer esa selección.

```
USE tienda;
```

Listo ya tenemos seleccionada la base de datos tienda.

3.- Creación de una tabla y mostrar sus campos (CREATE TABLE - SHOW TABLES - DESCRIBE - DROP TABLE)

Una base de datos almacena sus datos en tablas. Esta 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.

Cedula	Nombre	Apellido	Cargo	Tiempo_servicio
20345666	Mario	Perez	Administrador	4
12678987	Maria	Garcia	Gerente	10
17675823	Diego	Rodriguez	Vendedor	2

Gráficamente acá tenemos la tabla empleados, que contiene cuatro campos llamados: cedula, nombre, apellido y cargo. Luego tenemos tres registros almacenados en esta tabla, el primero almacena en el campo cedula con el valor “20345666”, seguido por el campo nombre con el valor "Mario", el campo apellido con el valor “Perez” y en el campo cargo con el valor " Administrador", y así sucesivamente con los otros dos registros.

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 "empleados", tipeamos:

```
CREATE TABLE empleados (

      cedula INT(8),
      nombre VARCHAR(30),
      apellido VARCHAR(30),
      cargo VARCHAR(30),
      tiempo_servicio INT;
);
```

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:

```
      SHOW TABLES;
```

Cuando se crea una tabla debemos indicar su nombre y definir sus campos con su tipo de dato. En esta tabla "empleados" definimos 4 campos:

- cedula: que contendrá una cadena de caracteres numérica, con longitud 8 (hasta 8 caracteres numéricos).

- nombre: que contendrá una cadena de hasta 30 caracteres de longitud, que almacenará el nombre del empelado

- nombre: que contendrá una cadena de hasta 30 caracteres de longitud, que almacenará el apellido del empleado

- cargo: otra cadena de caracteres de longitud 30, que guardará el nombre del cargo del empleado.

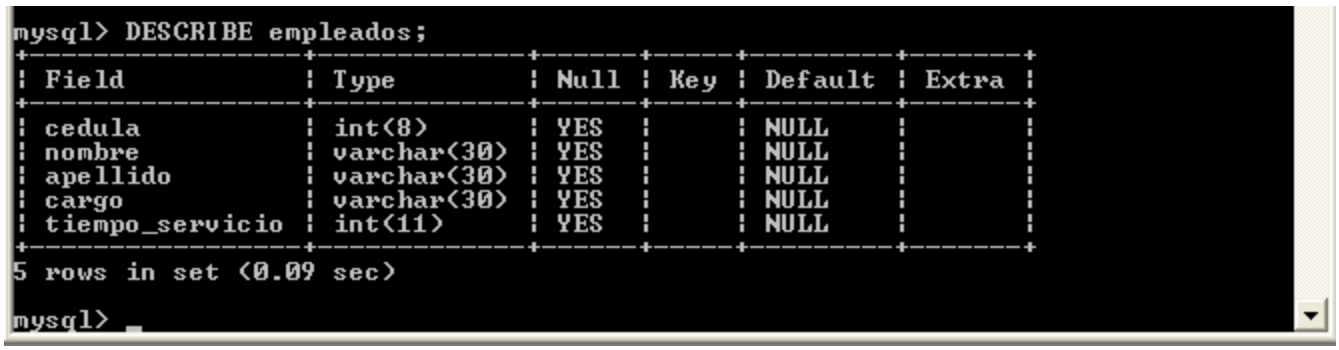
- tiempo_servicio: cadena de caracteres numéricos sin longitud definida que guardara los años de servicio del empleado.

Cada empleado ocupará un registro de esta tabla, con su respectiva cedula, nombre, apellido, cargo y tiempo_servicio.

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

```
 DESCRIBE empleados;
```

Aparece lo siguiente:



Estructura de la tabla empleados.

Esta es la estructura de la tabla “empleados”; nos muestra cada campo, su tipo, lo que ocupa en bytes y otros datos como la aceptación de valores nulos etc, Para eliminar una tabla usamos:

```
      DROP TABLE empleados;
```

Si tipeamos nuevamente:

```
DROP TABLE empleados;
```

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 empleados;
```

En la sentencia precedente especificamos que elimine la tabla “empleados” si existe.

También podemos utilizar la instrucción DROP para eliminar una BD

```
DROP DATABASE nombre_base_de_datos;
```

nombre_base_de_datos es referente al nombre de la bd que queremos borrar.

4-. Instrucción ALTER TABLE:

La instrucción ALTER TABLE permite modificar la estructura de una tabla existente, Podemos agregar o eliminar columnas, agregar o eliminar índices, modificar el tipo de columna (tipo de dato), renombrar las columnas y renombrar la tabla. ALTER TABLE crea una copia de la original y realiza los cambios sobre esta copia, luego la tabla original se borra y es renombrada la nueva tabla (la tabla copia.). Cabe destacar que mientras se ejecuta el ALTER TABLE, otros clientes pueden visualizar la original sin presentar cambio alguno. Aplicaremos a manera de ejemplo ciertos cambios a la tabla “empleados”.

Comando a utilizar:

4.1.- Eliminar varias columnas específicas en una sola instrucción:

```
ALTER TABLE empleados DROP COLUMN tiempo_servicio, DROP COLUMN apellido;
```

En la instrucción anterior DROP COLUMN anticipa el nombre de la columna que se desea borrar.

4.2.- Renombrar una columna:

```
ALTER TABLE empleados CHANGE cargo cargo_em VARCHAR(30);
```

En esta instrucción especificamos el nombre actual de la columna seguido del nuevo nombre y el tipo de columna, “cargo” lo identificamos como el nombre de la columna que queremos modificar y “cargo_em” el nuevo nombre.

4.3 Cambiar el tipo de datos de una columna sin renombrarla:

```
ALTER TABLE empleados MODIFY cargo_em INT NOT NULL;
```

En esta instrucción estamos especificando la columna a la cual le cambiaremos el tipo. Escribimos el nombre de la columna seguido del nuevo tipo. Podemos incluir atributos.

4.4 Agregar una nueva columna:

```
ALTER TABLE empleados ADD tiempo_servicio INT;
```

4.5 Agregar una nueva columna indicándole la posición:

```
ALTER TABLE empleados ADD direccion VARCHAR(100) AFTER apellido;
```

Podemos sustituir AFTER por FIRST para colocar la nueva columna de primera.

4.6 Renombrar una tabla:

```
ALTER TABLE empleados RENAME empleado.
```

Necesitamos escribir el nombre de la columna actual, seguido por la palabra reservada RENAME y finalmente el nuevo nombre de la tabla.

5.- Carga de registros a una tabla y su recuperación (INSERT INTO – SELECT)

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

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 empleados (cedula, nombre, apellido, cargo, tiempo_servicio) VALUES (20345666,'Mario','Perez','Administrador',4);
```

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 representar de la siguiente forma:

cedula	nombre	apellido	cargo	Tiempo_servicio
20345666	Mario	Perez	Administrador	4

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

Note que los datos ingresados, como corresponden a campos de cadenas de caracteres se colocan entre comillas simples. Las comillas simples son OBLIGATORIAS, mientras que los datos numéricos podemos escribirlos sin o con comillas.

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

```
      SELECT cedula, nombre, apellido, cargo, tiempo_servicio FROM empleados;
```

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

La instrucción anterior es equivalente a escribir:

```
      SELECT * FROM empleados;
```

El asterisco (*) significa todos los campos de la tabla.

6.- Tipos de datos básicos de un campo de una tabla

Ya explicamos que al crear una tabla debemos resolver qué campos (columnas) tendrá y que tipo de datos almacenará cada uno de ellos, es decir, su estructura. Estos son algunos tipos de datos básicos:

- VARCHAR: se usa para almacenar cadenas de caracteres. Una cadena es una secuencia de caracteres. Se coloca entre comillas (simples): 'Hola'. El tipo "VARCHAR" define una cadena de longitud variable en la cual determinamos el máximo de caracteres. Puede guardar hasta 255 caracteres. Para almacenar cadenas de hasta 30 caracteres, definimos un campo de tipo VARCHAR(30). Si asignamos una cadena de caracteres de mayor longitud que la 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 caracteres.

- IINTEGER: se usa para guardar valores numéricos enteros, de -2000000000 a 2000000000 aprox. Definimos campos de este tipo cuando queremos representar, por ejemplo, cantidades.

- FLOAT: se usa para almacenar valores numéricos decimales. Se utiliza como separador el punto (.). Definimos campos de este tipo para precios, por ejemplo.

Antes de crear una tabla debemos pensar en sus campos y optar por el tipo de dato adecuado para cada uno de ellos. Por ejemplo, si en un campo almacenaremos números enteros, el tipo "FLOAT" sería una mala elección; si vamos a guardar precios, el tipo "FLOAT" es correcto, no así "INTEGER" que no tiene decimales.

7.- Recuperación de algunos campos (SELECT)

Como vimos en el punto número 4. Para ver todos los registros de una tabla usamos la instrucción SELECT.

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

```
SELECT nombre, apellido FROM empleados;
```

En la sentencia anterior la consulta mostrará sólo los campos "nombre" y "apellido". En la siguiente sentencia, veremos los campos correspondientes al cargo y cedula de todos los empleados:

```
SELECT cargo, cedula FROM empleados;
```

Como podemos ver podemos decidir incluso el orden en que se mostrarán los campos solicitados.

8.- Recuperación de registros específicos (SELECT - WHERE)

Hemos aprendido cómo ver todos los registros de una tabla. Pero podemos aplicar ciertas condiciones para realizar una búsqueda más específica.

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 "Mario", para ello utilizamos "WHERE" y luego de ella, la condición:

```
SELECT nombre, apellido, cargo FROM empleados WHERE nombre='Mario';
```

Para las condiciones se utilizan operadores relacionales. El signo igual (=) es un operador relacional. Para la siguiente selección de registros especificamos una condición que solicita los usuarios cuya cedula es igual a 17675823:

```
SELECT nombre, apellido, cargo FROM empleados WHERE cedula=17675823;
```

Si ningún registro cumple la condición establecida con el "WHERE", no devolverá ningún resultado.

9.- Operadores Relacionales = <> < <= > >=

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

SELECT * FROM empleados WHERE apellido='Perez';

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 nombre sea diferente de 'Mario', para ello usamos la condición:

```
SELECT nombre, apellido FROM empleados WHERE nombre<>'Mario';
```

Podemos comparar valores numéricos. Por ejemplo, queremos mostrar los empleados cuyo tiempo de servicio sea mayor a 5 años:

```
SELECT nombre, apellido FROM empleados WHERE tiempo_servicio>20;
```

También, los empleados cuyo tiempo de servicio sea menor o igual a 10:

```
SELECT nombre, apellido, tiempo_servicio FROM empleados WHERE tiempo_servicio<=10;
```

10.- Borrado de registros de una tabla (DELETE)

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

```
DELETE FROM empleados;
```

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 cláusula "WHERE" con la cual establecemos la condición que deben cumplir los registros a borrar. Por ejemplo, queremos eliminar aquel registro cuyo nombre es 'Diego':

```
DELETE FROM empleados WHERE nombre='Diego';
```

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. Por ejemplo:

```
DELETE FROM empleados WHERE nombre='Leonardo';
```

11.- 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 "empleados", queremos cambiar los valores de todos los años de servicio, por "6":

```
 UPDATE empleados SET tiempo_servicio=6;
```

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 al tiempo de servicio del empleado llamado 'Mario', queremos como nuevo tiempo de servicio “4”, necesitamos una condición "WHERE" que afecte solamente a este registro:

```
 UPDATE empleados SET tiempo_servicio=4 WHERE nombre='Mario';
```

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 empleados SET nombre='Marcelo', tiempo_servicio=5 WHERE nombre='Mario';
```

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.

12.- Clave primaria

Una clave primaria es un campo (o varios) que identifica un solo registro (fila) en una tabla. Los valores en este caso no se repiten ni pueden ser nulos.

Veamos un ejemplo, si tenemos una tabla con datos de personas, el número de cedula 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 cédula será siempre distinto.

Si tenemos la tabla "empleados", la cédula de cada empleado puede establecerse como clave primaria, es un valor que no se repite; puede haber usuarios con igual nombre, pero su cédula será siempre distinto. Establecemos que un campo sea clave primaria al momento de creación de la tabla:

```
CREATE TABLE empleados (
      cedula INT(8),
      nombre VARCHAR(30),
      apellido VARCHAR(30),
      cargo VARCHAR(30),
      tiempo_servicio INT;
      PRIMARY KEY(cedula)
);
```

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 "cedula" es clave primaria y no acepta valores nulos.

Ingresamos algunos registros utilizando un INSERT múltiple:

```
INSERT INTO empleados2 (cedula, nombre, apellido, cargo, tiempo_servicio) VALUES
(15856345,'Leonardo','Martinez',’vendedor’,6),
(13789123,'Jose','Bracho',’Auxiliar’,1),
(17321906,'Ana','Alvarado',’Gerente’,8);
```

La inserción múltiple permite insertar todos los registros que queramos con una sola instrucción INSERT. La forma es la siguiente:

1. Entre cada paréntesis colocamos los datos del registro en el mismo orden en que aparecen los nombre de los campos.
2. Luego Separamos con coma (,) para comenzar el siguiente grupo de datos. Repetimos el paso anterior.
3. Al finalizar la escritura de los datos del último registro, colocamos (;) para finalizar la instrucción INSERT.

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 una cédula de empleado existente, por ejemplo:

```
 INSERT INTO empleados2 (cedula, nombre, apellido, cargo, tiempo_servicio) VALUES
(17321906, 'Gustavo','Andrade',’Administrador’,3);
```

¿Cuál es el mensaje?...

Una tabla sólo puede tener una clave primaria y esta puede ser de cualquier tipo, así mismo al establecer una clave primaria estamos Indexando la tabla, es decir, creando un índice para dicha tabla.

13.- Campo entero con autoincremento

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

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

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

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

```
 CREATE TABLE proveedores(
      codigo INT AUTO_INCREMENT,
      nombre VARCHAR(20),
      direccion VARCHAR(50),
```

```
      pago FLOAT,
      PRIMARY KEY(codigo)
 );
```

Para definir un campo auto incrementable 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 proveedores (nombre, dirección, pago) VALUES ('provee1','Porlamar', 3000);
```

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 proveedores (codigo, nombre, dirección, pago) VALUES (6, 'provee2','Caracas,
34.56);
```

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

- si el valor está repetido aparecerá un mensaje de error y el registro no se ingresará.

- si el valor dado saltea la secuencia, lo toma igualmente y en las siguientes inserciones, continuará la secuencia tomando el valor más alto.

- si el valor ingresado es 0, no lo toma y guarda el registro continuando la secuencia.

- si el valor ingresado es negativo (y el campo no está definido para aceptar sólo valores positivos), lo ingresa.

Para que este atributo funcione correctamente, el campo debe contener solamente valores positivos.

14.- Clave Foránea o Clave Referenciada:

Las claves foráneas son referencias a registros de otras tablas estableciendo una relación entre ellas. La clave foránea apunta siempre a la clave primaria de la tabla a la cual hace referencia. Para trabajar con claves foráneas debemos establecer el tipo de tabla como InnoDB.

Para continuar con el ejercicio anterior crearemos la tabla “productos” que serán todos los productos vendidos en la tienda. Suponiendo que la tienda es de artículos para computadoras creamos

```
CREATE TABLE productos (codigo_p INT AUTO_INCREMENT, nombre VARCHAR(40), precio FLOAT,
disponible INT, PRIMARY KEY(codigo_p));
```

Analicemos ahora las estructuras que queremos relacionar: tenemos la tabla “proveedores”, y la tabla “productos”, los productos son adquiridos por medio de proveedor o un fabricante, en nuestro caso los fabricantes de los productos serán también proveedores de los mismos.

Ahora debemos identificar los productos comprados a un determinado proveedor en nuestra tabla productos. La manera de identificarlos es recordando que en la tabla “proveedores” existe un código de proveedor que es clave primaria en dicha tabla, entonces replicamos la columna “codigo” de la tabla proveedores en la tabla productos con el nombre de “codigo_provee”, de esta manera estaríamos relacionando ambas tablas evitando la redundancia de datos.

La columna “codigo” de la tabla proveedores y la columna “codigo_provee” de la tabla productos deben tener el mismo tipo de dato, es decir, sí “codigo” es INT entonces “código_provee” también es INT.

Ahora bien si nos fijamos un poco lo que acabamos de establecer es una clave foránea en la tabla de “productos” que hace referencia a la clave primaria de la tabla productos.

Como escribirla:

Podemos borrar la tabla creada anteriormente y crearla de nuevo:

```
CREATE TABLE productos (codigo_p INT AUTO_INCREMENT, codigo_provee INT, nombre
VARCHAR(40), precio FLOAT, disponible INT, PRIMARY KEY(codigo_p), FOREING
KEY(codigo_provee) REFERENCES proveedores(codigo));
```

FOREING KEY() encierra la columna que será clave foránea o referencial en la tabla “productos”, REFERENCES indica la tabla a la que hace referencia encerrando entre paréntesis el campo al que hace referencia, en nuestro caso REFERENCES proveedores(codigo) es el campo “código” de la tabla “proveedores”.

Si intentáramos insertar un producto de un proveedor inexistente MySQL nos daría un error advirtiendo de que no existe dicha clave de proveedor. Esto nos muestra que primero debemos insertar registros en la tabla “proveedores” (tabla referencia) y luego en la tabla “productos” (tabla que toma la referencia). Insertamos algunos registros en la tabla productos:

```
INSERT INTO productos (codigo_provee, nombre, precio,) VALUES (1, 'desktop', 3000), (1,
'desktop', 4500), (2, 'laptop', 3000); (3, 'laptop', 5000), (4, 'teclado', 500), (4,
'mouse', 300), (5, 'desktop', 5000), (6, 'desktop', 3200), (1, 'Monitor', 1000), (7,
'Monitor', 2000), (8, 'impresora', 3000),(9, 'impresora', 5000);
```

Si quisiéramos cambiar el código del proveedor por un nuevo código, debemos tomar en cuenta que ésta columna aparece en la tabla de “proveedores” y la tabla de “productos”. Trabajar con clave foránea permite realizar acciones en cascada, es decir si cambiáramos el código del “proveedor” en la tabla proveedores automáticamente se cambiaría en la tabla “productos” y en todas las demás tablas que aparezca la referencia. Igualmente si borramos un proveedor automáticamente quedaría fuera de la tabla de “productos”.

Para que una clave foránea pueda cumplir con estas características debemos agregar las siguientes instrucciones al momento de crear la tabla que tendrá clave foránea:

ON DELETE CASCADE ON UPDATE CASCADE

```
CREATE TABLE productos (codigo_p INT AUTO_INCREMENT, codigo_provee INT, nombre
VARCHAR(40), precio FLOAT, disponible INT, PRIMARY KEY(codigo_p), FOREING
KEY(codigo_provee) REFERENCES proveedores(codigo) ON DELETE CASCADE ON UPDATE CASCADE);
```

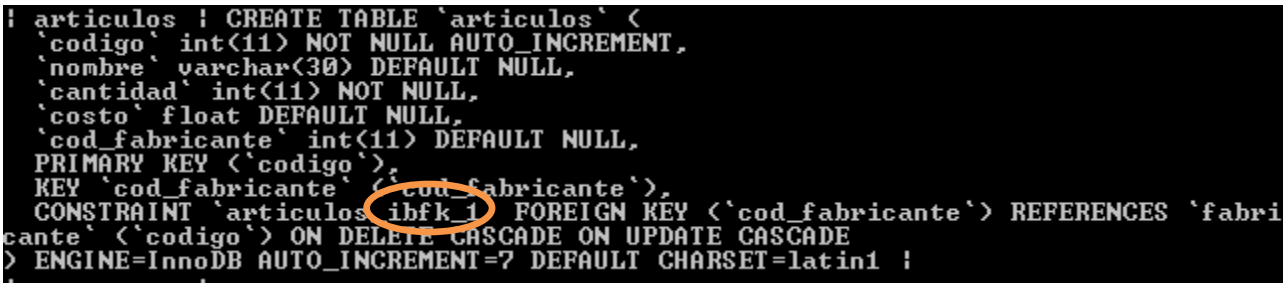
Una tabla puede tener más de una clave foránea con los atributos anteriores. Para eso se escribe la primera clave foránea con sus atributos seguida de una coma (,) y la siguiente clave foránea con sus atributos.

También podemos borrar una clave foránea que ya no necesitemos. Para eso necesitamos seguir los siguientes pasos:

1.- Debemos verificar el id asignado por MySQL a la clave foránea. Este es un id que asigna el sistema para identificar las claves foráneas que puedan existir en una misma tabla. Para poder visualizar dicho id utilizamos la siguiente instrucción:

```
SHOW CREATE TABLE productos;
```

Nos mostrará la siguiente información:



Nota: En el caso de la imagen la tabla se llama “artículos”

El óvalo naranja señala en donde se encuentra el id para la clave foránea del ejemplo en la imagen. Debemos tomar ese valor para el siguiente paso.

2.- Debemos escribir la siguiente sentencia tomando en cuenta el id de clave foránea a borrar:

```
ALTER TABLE productos DROP FOREIGN KEY productos_ibfk_1;
```

Al final de la sentencia estamos señalando cuál es la clave foránea a borrar perteneciente a la tabla “productos”**.**

También podemos crear una clave foránea a una tabla ya existente. Para eso utilizamos la siguiente instrucción:

```
ALTER TABLE productos ADD FOREIGN KEY(codigo_provee) REFERENCES proveedores(codigo) ON
DELETE CASCADE ON UPDATE CASCADE;
```

Es de notar que es prácticamente la definición de clave foránea que utilizamos en la creación de la tabla “productos” unas líneas arriba, pero combinada con la instrucción ALTER.

15.- comando TRUNCATE TABLE

Aprendimos que para borrar todos los registro de una tabla se usa "DELETE" sin condición "WHERE".

También podemos eliminar todos los registros de una tabla con "TRUNCATE TABLE". Por ejemplo, queremos vaciar la tabla "proveedores", usamos:

```
 TRUNCATE TABLE proveedores;
```

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

La diferencia con "DROP TABLE" es que esta sentencia borra la tabla, "TRUNCATE TABLE" la vacía.

La diferencia con "DELETE" es la velocidad, es más rápido "TRUNCATE TABLE" que "DELETE" (se nota cuando la cantidad de registros es muy grande) ya que éste borra los registros uno a uno.

Otra diferencia es la siguiente: cuando la tabla tiene un campo "AUTO_INCREMENT", si borramos todos los registros con "DELETE" y luego ingresamos un registro, al cargarse el valor en el campo 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 "proveedores" 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.

16.- valores NULL

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

Field	Type	NULL	Key	DEFAULT
codigo	int(11)	NO	PRI	auto_INcrement
nombre	varchar(20)	YES		(NULL)
direccion	varchar(50)	YES		(NULL)
pago	float	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".

Vamos a explicar los valores nulos.

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

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

En contraposición, tenemos campos que no pueden estar vacíos jamás, por ejemplo, los campos que identifican cada registro, como los códigos de identificación, que son clave primaria.

Por defecto, es decir, si no lo aclaramos en la creación de la tabla, los campos permiten valores nulos.

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

```
INSERT INTO proveedores (nombre, direccion, pago) VALUES ('provee3', 'La Asuncion',
NULL);
```

Note que el valor "NULL" no es una cadena de caracteres, no se coloca entre comillas, puesto que es una palabra reservada.

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

```
CREATE TABLE proveedores(
       codigo INt auto_INcrement,
       nombre varchar(20) NOT NULL
       direccion varchar(50),
       pago FLOAT,
       PRIMARY KEY(codigo)
);
```

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

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

Para recuperar los registros que contengan el valor "NULL" en el campo "pago" 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 proveedores WHERE pago IS NULL;
```

La sentencia anterior tendrá una salida diferente a la siguiente:

```
SELECT * FROM proveedores WHERE pago=0;
```

Con la primera sentencia veremos los proveedores cuyo pago es igual a "NULL" (desconocido); con la segunda, los proveedores cuyo pago es 0.

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

```
SELECT * FROM proveedores WHERE direccion is NULL;
SELECT * FROM proveedores WHERE direccion='' '';
```

Con la primera sentencia veremos los proveedores cuya dirección es igual a "NULL", con la segunda, los proveedores cuya dirección guarda una cadena vacía.

17.- 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 INT UNSIGNED;
```

Si necesitamos almacenar el precio de los productos, 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.

18.- Tipos de datos

Hasta ahora hemos visto 3 tipos de datos: VARCHAR, INTEGER (con y sin signo) y FLOAT (con y sin signo). Hay más tipos, incluso, subtipos.

Los valores que podemos guardar son:

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

B) 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.

C) FECHAS Y HORAS: para guardar fechas y horas dispone de varios tipos: DATE (fecha), DATETIME (fecha y hORa), TIME (hora), YEAR (año) y TIMESTAMP.

D) OTROS TIPOS: ENUM y SET representan una enumeración y un conjunto respectivamente.

19.- 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:

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

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

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

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

20.- 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

21.- 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 fecha-hora 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

22.- valores por defecto

Para campos declarados "NOT NULL", el valor por defecto depende del tipo de dato. Para cadenas de caracteres el valor por defecto es una cadena vacía. Para valores numéricos el valor por defecto es 0; en caso de ser "AUTO_INCREMENT"

es el valor mayor existente+1 comenzando en 1. Para campos de tipo fecha y hora, el valor por defecto es 0 (por ejemplo, en un campo "date" es "0000-00-00").

Para todos los tipos, excepto "BLOB", "TEXT" y "AUTO_INCREMENT" se pueden explicitar valores por defecto con la cláusula "DEFAULT".

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

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 "nombre", ingresará una cadena vacía porque es "VARCHAR NOT NULL"; en el campo "direccion" almacenará "NULL", porque no está definido "NOT NULL"; en el campo "pago" 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	siguiente de la sec., empieza en 1	no permite
carac.,numer.,fecha,hORa NULL	NULL	permite

23.- Valores Inválidos

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 carácter:

- valor numérico: si en un campo definido de tipo caracter ingresamos un valor numérico, lo convierte automáticamente a cadena. Por ejemplo, si guardamos 234 en un VARCHAR, almacena '234'.

- mayor longitud: si intentamos guardar una cadena de caracteres mayor a la longitud definida, la cadena se corta guardando sólo la cantidad de caracteres que quepa. Por ejemplo, si definimos un campo de tipo VARCHAR(10) y le asignamos la cadena 'Buenas tardes', se almacenará 'Buenas tar' ajustándose a la longitud de 10.

Para campos numéricos:

- Cadenas: si en un campo numérico ingresamos una cadena, lo pasa por alto y coloca 0. POR ejemplo, si en un campo de tipo "INTEGER" guardamos 'abc', almacenará 0.

- Valores fuera de rango: si en un campo numérico intentamos guardar un valor fuera de rango, se almacena el valor límite del rango más cercano (menor o mayor). Por ejemplo, si definimos un campo 'TINYINT' (cuyo rango va de -128 a 127) e intentamos guardar el valor 200, se almacenará 127, es decir el máximo permitido del rango; si intentamos guardar -200, se guardará -128, el mínimo permitido por el rango. Otro ejemplo, si intentamos guardar el valor 1000.00 en un campo definido como decimal (5,2) guardará 999.99 que es el mayor del rango.

- Valores incorrectos: si cargamos en un campo definido de tipo decimal un valor con más decimales que los permitidos en la definición, el valor es redondeado al más cercano. Por ejemplo, si cargamos en un campo definido como decimal (4,2) el valor 22.229, se guardará 22.23, si cargamos 22.221 se guardará 22.22.

Para campos definidos AUTO_INCREMENT 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 MySQLl 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

24.- 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 proveedores (nombre, direccion, pago) VALUES ('provee4', 'Maracaibo', 25.7);
```

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 nombre:

```
INSERT INTO proveedores (direccion, pago) VALUES ('Maracaibo', 25.7);
```

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 pago:

```
INSERT INTO proveedores (nombre, direccion) VALUES ('provee10','Valencia');
```

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

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 "pago" sea 1.11 y el valor por defecto del campo "direccion" sea "Desconocido":

```
CREATE TABLE proveedores2(
            codigo INt UNSIGNED AUTO_INCREMENT,
            nombre varchar(40) NOT NULL,
            direccion varchar(50) DEFAULT 'Desconocido',
            pago decimal(5,2) UNSIGNED DEFAULT 1.11,
            PRIMARY KEY(codigo)
);
```

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

```
INSERT INTO proveedores (nombre) VALUES ('prove11');
```

MySQL insertará el registro, con nombre “prove11”, en nombre colocará "Desconocido" y en pago "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 proveedores;
```

"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 proveedores (nombre, dirección, pago) VALUES ('provee12','DEFAULT',67.8);
```

25.- Atributo ZEROFILL en una columna de una tabla.

Cualquier campo numérico puede tener otro atributo extra "ZEROFILL", este rellena con ceros los espacios disponibles a la izquierda.

Por ejemplo, creamos la tabla "ventas", definiendo el campo "num_ventas" con el atributo "ZEROFILL":

```
CREATE TABLE ventas(
            codigo INt(6) ZEROFILL AUTO_INCREMENT,
            cod_empleado VARCHAR(40) NOT NULL,
            cod_producto VARCHAR(30),
            precio DECIMAL(5,2) UNSIGNED,
            cantidad SMALLINT ZEROFILL,
            fecha_hora DATETIME;
            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.

26.- 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 el código, fecha, precio y cantidad de cada venta escribimos la siguiente sentencia:

```
SELECT codigo, fecha_hora, precio, cantidad FROM ventas;
```

Si queremos saber el monto total en dinero de una venta realizada podemos multiplicar el precio por la cantidad y hacer que MySQL realice el cálculo y lo incluya en una columna extra en la salida:

```
SELECT codigo, fecha_hora, precio ,cantidad, precio*cantidad FROM ventas;
```

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

SELECT codigo, fecha_hora, precio, precio*0.1, precio-(precio*0.1) FROM ventas;

27.- Funciones para el manejo de cadenas

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 envidada como primer argumento está presente en la lista de cadenas enviadas como segundo argumento. La lista de cadenas enviada como segundo argumento es una cadena formada por subcadenas separadas por comas. Devuelve 0 si no encuentra "cadena" en la "lista de cadenas". Ejemplo:

```
SELECT FIND_IN_SET('hola','como esta,hola,buen dia');
```

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');
```

Devuelve 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 LOCALE('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');
```

Devuelve 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".

- RPAD (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 inquierda, primer caracter. 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 caracter. 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);
```

- 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"
SELECT TRIM ('  hola  ');   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 order 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.

28.- Funciones matemáticas

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

```
SELECT 5/0;
```

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.

-SRQT(): 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.

29.- Funciones para el uso de fecha y hora

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

-ADDDATE(fecha, interval expresion): 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".

-ADDDATE(fecha, dias): retorna la fecha agregándole a fecha "dias". Ejemplo: ADDDATE('2006-10-10',25), retorna "2006-11-04".

-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 expresion tipo) y DATE_SUB(fecha,interval expresion tipo): el argumento "fecha" es un valor "DATE" o "DATETIME", "expresion" especifica el valor de intervalo a ser añadido o substraído de la fecha indicada (puede empezar con "-", para intervalos negativos), "tipo" indica la medida de adición o substracción. Ejemplo:

```
DATE_ADD('2006-08-10', interval 1 month);
```

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".

Los valores para "tipo" pueden ser: second, minute, hour, day, month, year, MINUTE_SECOND (minutos y segundos), HOUR_MINUTE (horas y minutos), DAY_HOUR (días y horas), YEAR_MONTH (año y mes), HOUR_SECOND (hora, minuto y segundo), DAY_MINUTE (dias, horas y minutos), DAY_SECOND(dias a segundos).

```
-DATEDIFF(fecha1,fecha2);
```

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".

Los valores para tipo pueden ser: second, minute, hour, day, month, year, MINUTE_SECOND, HOUR_MINUTE, DAY_HOUR, YEAR_MONTH, HOUR_SECOND (horas, minutos y segundos), DAY_MINUTE (días, horas y minutos), DAY_SECOND (días a segundos).

-HOUR(hora): 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".

30.- Cláusula ORDER 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".

Por ejemplo, recuperamos los registros de la tabla "proveedores" ordenados por el nombre:

```
SELECT codigo, nombre, direccion  FROM proveedores ORDER BY nombre;
```

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 "pago":

```
SELECT * FROM proveedores ORDER BY pago;
```

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 * FROM proveedores ORDER BY nombre DESC;
```

También podemos ordenar por varios campos, por ejemplo, por "nombre" y "pago":

```
SELECT *  FROM proveedores ORDER BY nombre, pago;
```

Incluso, podemos ordenar en distintos sentidos, por ejemplo, por "nombre" en sentido ascendente y "direccion" en sentido descendente:

```
SELECT *  FROM proveedores ORDER BY nombre ASC, direccion DESC;
```

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

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", exclusivo
- NOT, significa "no", invierte el resultado
- (), paréntesis.

Los operadores lógicos se usan para combinar condiciones.

Queremos recuperar todos los registros cuyo proveedor sea igual a "prove1" y cuyo pago no supere los 3000 bolívares, para ello necesitamos 2 condiciones:

```
 SELECT * FROM proveedores  WHERE (nombre='provee1') AND (pago<=3000);
```

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

Queremos ver los proveedores cuyo nombre sea "provee2" y/o cuya dirección sea "Maracaibo":

```
 SELECT * FROM proveedores  WHERE nombre='provee2' OR direccion='Maracibo';
```

En la sentencia anterior usamos el operador "OR", indicamos que recupere los proveedores en los cuales el valor del campo "nombre" sea "provee2" y/o el valor del campo "direccion" sea "Maracaibo", 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 proveedores cuyo nombre sea "provee3" o cuya dirección sea "Porlamar":

```
 SELECT * FROM proveedores WHERE (nombre='provee3') XOR (direccion='Porlamar');
```

En la sentencia anterior usamos el operador "XOR", indicamos que recupere los proveedores en los cuales el valor del campo "nombre" sea "provee3" o el valor del campo "direccion" sea "Porlamar", 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 proveedores que no cumplan la condición dada, por ejemplo, aquellos cuya dirección NO sea "Porlamar":

```
 SELECT * FROM proveedores WHERE NOT (direccion='Porlamar');
```

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 proveedores WHERE (nombre='provee1') OR (direccion='Maracaibo' AND
pago<3000);
```

```
SELECT * FROM proveedores  WHERE (nombre='provee2' OR direccion='Valencia') AND
(pago<3000);
```

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)

Hemos visto los operadores relacionales:

= (igual), <> (distinto), > (mayor), < (menor), >= (mayor o igual), <= (menor o igual), is NULL/IS NOT NULL (si un valor es NULL o no).

Existen otros que simplifican algunas consultas:

Para recuperar de nuestra tabla "empleados" los registros que tienen tiempo de servicio mayor o igual a 5 años y menor o igual a 9 años, usamos 2 condiciones unidas por el operador lógico "AND":

```
 SELECT * FROM empleados WHERE tiempo_servicio>=5 AND tiempo_servicio<=9;
```

Podemos usar "BETWEEN":

```
SELECT * FROM empleados WHERE tiempo_servicio BETWEEN 5 AND 9;
```

"BETWEEN" significa "entre". Averiguamos si el valor de un campo dado (precio) está entre los valores mínimo y máximo especificados (5 y 9 respectivamente).

Si agregamos el operador "NOT" antes de "BETWEEN" el resultado se invierte.

Para recuperar los empleados cuyo nombre sea 'Mario' o 'Ana' usamos 2 condiciones:

```
SELECT * FROM empleados WHERE nombre='Mario' OR nombre='Ana';
```

Podemos usar "IN":

```
SELECT * FROM empleados WHERE nombre IN('Mario','Ana');
```

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

Para recuperar los empleados cuyo nombre no sea 'Mario' ni 'Ana' usamos:

```
SELECT * FROM empleados WHERE nombre<>'Mario' AND nombre<>'Ana';
```

También podemos usar "IN":

```
SELECT * FROM empleados WHERE nombre NOT IN ('Mario','Ana');
```

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)

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 nombre contenga la cadena "Mario" debemos tipear:

```
SELECT * FROM empleados WHERE nombre LIKE "%Mario%";
```

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

Para seleccionar todos los empleados cuyo nombre que comiencen con "A":

```
SELECT * FROM empleados WHERE nombre LIKE 'A%';
```

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

Para seleccionar todos los empleados cuyo nombre no comience con "A":

```
SELECT * FROM empleados WHERE nombre NOT LIKE 'A%';
```

Así como "%" reemplaza cualquier cantidad de caracteres, el guión bajo "_" reemplaza un caracter, es el otro caracter comodín. Por ejemplo, queremos ver los empleados cuyo apellido es López o Lopes pero no estamos seguros de cuál de ellos está bien escrito, entonces utilizamos:

```
SELECT * FROM empelados WHERE apellido LIKE "%Lope_";
```

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 caracter 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 caracter comodín, la barra invertida "\", así lo tomará como caracter de búsqueda literal y no como comodín para la búsqueda. Para buscar el caracter 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 proveedores que contengan la cadena "Ma" usamos:

```
SELECT * FROM proveedores WHERE direccion REGEXP 'Ma';
```

Para buscar los proveedores que tienen al menos una "h" o una "k" o una "w" tipeamos:

```
SELECT nombre FROM proveedores WHERE nombre REGEXP '[hkw]';
```

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

```
SELECT nombre FROM proveedores WHERE nombre NOT REGEXP '[hkw]';
```

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

```
SELECT nombre FROM proveedores WHERE nombre REGEXP '[a-d]';
```

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

```
SELECT * FROM proveedores WHERE nombre REGEXP '^A';
```

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

```
SELECT *  FROM proveedores WHERE nombre REGEXP 'HP$';
```

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

```
SELECT * FROM proveedores WHERE direccion REGEXP 'a.e';
```

El punto (.) identifica cualquier caracter.

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

```
SELECT * FROM proveedores WHERE nombre REGEXP 'a..e';
```

Para buscar proveedores que tengan 6 caracteres exactamente usamos:

```
SELECT * FROM proveedores WHERE nombre REGEXP '^......$';
```

Para buscar proveedores que tengan al menos 6 caracteres usamos:

```
SELECT * FROM proveedores WHERE nombre REGEXP '......';
```

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

```
SELECT * FROM proveedores WHERE direccion REGEXP 'a.*a';
```

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

35.- 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 "productos" contiene muchos registros. Para averiguar la cantidad sin necesidad de contarlos manualmente usamos la función "COUNT()":

```
SELECT COUNT(*) FROM productos;
```

La función "COUNT()" cuenta la cantidad de registros de una tabla, incluyendo los que tienen valor nulo.

Para saber la cantidad de productos comprados al proveedor “provee1” tipeamos:

```
SELECT COUNT(*) FROM productos WHERE codigo_provee=1;
```

También podemos utilizar esta función junto con la cláusula "WHERE" para una consulta más específica. Por ejemplo, solicitamos la cantidad de productos que contienen la cadena "monitor":

```
SELECT COUNT(*) FROM productos WHERE nombre LIKE '%monitor%';
```

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 productos;
```

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.

36.- Funciones de agrupamiento (COUNT - MAX - MIN - SUM - AVG)

Existen en MySQL funciones que nos permiten contar registros, calcular sumas, promedios, obtener valores máximos y mínimos. 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 productos que tenemos disponibles para la venta:

```
SELECT SUM(disponible) FROM productos;
```

También podemos combinarla con "WHERE". Por ejemplo, queremos saber cuántos productos tenemos de un determinado proveedor:

```
SELECT SUM(disponible) FROM productos WHERE codigo_provee ='4';
```

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 productos:

```
SELECT MAX(precio) FROM productos;
```

Queremos saber cuál es el valor mínimo de los productos de un determinado proveedor:

```
SELECT MIN(precio) FROM productos WHERE codigo_provee LIKE '%4%';
```

La función AVG() retorna el valor promedio de los valores del campo especificado. Por ejemplo, queremos saber el promedio del precio de los productos que son monitores

```
SELECT AVG(precio) FROM productos WHERE nombre LIKE '%monitor%';
```

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

37.- Agrupar registros (GROUP BY)

Generalmente estas funciones se combinan con la sentencia "GROUP BY", que agrupa registros para consultas detalladas.

Queremos saber la cantidad de productos por proveedor, podemos tipear la siguiente sentencia:

```
SELECT COUNT(*) FROM productos WHERE codigo_provee=1;
```

y repetirla con cada valor de "código_provee":

```
SELECT COUNT(*) FROM productos WHERE codigo_provee=2;

SELECT COUNT(*) FROM productos WHERE codigo_provee=3';
```

Pero hay otra manera, utilizando la cláusula "GROUP BY":

```
SELECT nombre, COUNT(*) FROM productos GROUP BY codigo_provee;
```

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 del producto y cuente la cantidad agrupando los registros por el campo "codigo_provee". Como resultado aparecen los nombres de los productos y la cantidad de registros para cada valor del campo.

Para conocer el total de pago a proveedores agrupados por ciudad del proveedor:

```
SELECT direccion, SUM(pago) FROM proveedores  GROUP BY direccion;
```

Para saber el máximo y mínimo pago a proveedores agrupados por direccion:

```
SELECT direccion, MAX(pago) FROM proveedores  GROUP BY direccion;

SELECT direccion, MIN(pago) FROM proveedores  GROUP BY direccion;
```

Se pueden simplificar las 2 sentencias anteriores en una sola sentencia, ya que usan el mismo "GROUP BY":

```
SELECT direccion, MIN(pago), MAX(pago) FROM proveedores  GROUP BY direccion;
```

Para calcular el promedio del valor de pago agrupados por direccion:

```
SELECT direccion, AVG(pago) FROM proveedores  GROUP BY direccion;
```

Podemos agrupar por más de un campo, por ejemplo, vamos a hacerlo por "direccion" y "nombre":

```
 SELECT nombre,direccion, COUNT(*) FROM proveedores GROUP BY nombre,direccion;
```

También es posible limitar la consulta con "WHERE".

Vamos a contar y agrupar por dirección sin tener en cuenta "Porlamar":

```
 SELECT nombre, direccion, COUNT(*) FROM proveedores WHERE direccion<>'Porlamar' GROUP BY
direccion;
```

Podemos usar las palabras claves "ASC" y "DESC" para una salida ordenada:

```
 SELECT nombre, direccion, COUNT(*) FROM proveedores WHERE direccion<>'Porlamar' GROUP BY
direccion DESC;
```

38.- 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 proveedores agrupados por dirección usamos la siguiente instrucción ya aprendida:

```
 SELECT nombre,direccion, COUNT(*) FROM proveedores GROUP BY nombre, direccion;
```

Si queremos saber la cantidad de proveedores agrupados por pago pero considerando sólo algunos grupos, por ejemplo, los que devuelvan un pago mayor a 37.6 bolivares, usamos la siguiente instrucción:

```
 SELECT nombre, direccion, COUNT(*) FROM proveedores GROUP BY direccion HAVING pago>37.6;
```

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 productos agrupados por proveedor:

```
 SELECT nombre, AVG(precio) FROM productos GROUP BY codigo_provee;
```

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

```
 SELECT nombre, AVG(precio) FROM productos GROUP BY codigo_provee HAVING AVG(precio)>25;
```

En algunos casos es posible confundir las cláusulas "WHERE" y "HAVING". Queremos contar los registros agrupados por código_provee sin tener en cuenta al proveedor “provee1”.

Analicemos las siguientes sentencias:

```
SELECT nombre, COUNT(*) FROM productos WHERE codigo_provee<>2 GROUP BY codigo_provee;


SELECT nombre, COUNT(*) FROM productos GROUP BY codigo_provee HAVING codigo_provee<>2;
```

Ambas devuelven el mismo resultado, pero son diferentes.

La primera, selecciona todos los registros rechazando los del proveedor con código 2 y luego los agrupa para contarlos. La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza la cuenta correspondiente al código de proveedor igual 2.

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 productos, sin considerar los que tienen precio nulo, agrupados por código de proveedor, sin considerar el código de proveedor 3:

```
 SELECT nombre, COUNT(*) FROM producto WHERE precio is NOT NULL GROUP BY codigo_provee
HAVING codigo_provee<>3;
```

Aquí, selecciona los registros rechazando los que no cumplan con la condición dada en "WHERE", luego los agrupa por "código_provee" 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 de los productos agrupados por “cod_provee”, de aquellos proveedores que tienen 2 o más productos:

```
 SELECT codigo_provee, AVG(precio) FROM productos GROUP BY codigo_provee HAVING
COUNT(cod_fabricante) >=2;
```

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

```
 SELECT codigo_provee, MAX(precio) FROM productos GROUP BY codigo_provee HAVING
MAX(precio)>=30;
```

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

```
SELECT codigo_provee, MAX(precio) AS  'mayor' FROM productos GROUP BY codigo_provee HAVING
mayor>=30;
```

39.- 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 productos, si utilizamos esta sentencia:

```
 SELECT nombre FROM productos;
```

Aparecen repetidos. Para obtener la lista de productos SIN repetición usamos:

```
 SELECT DISTINCT nombre FROM productos;
```

También podemos tipear:

```
 SELECT nombre FROM productos GROUP BY nombre;
```

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

```
SELECT DISTINCT nombre FROM productos WHERE nombre IS NOT NULL;
```

Para contar los distintos productos, sin considerar el valor "NULL" usamos:

```
SELECT COUNT(DISTINCT nombre) FROM productos;
```

Note que si contamos los productos sin "DISTINCT", no incluirá los valores "NULL" pero si los repetidos:

```
SELECT COUNT(nombre) FROM productos;
```

La sentencia anterior sentencia cuenta los registros que tienen nombre.

Para obtener los nombres de los proveedores usamos:

```
SELECT nombre FROM proveedores;
```

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

```
SELECT DISTINCT nombre FROM proveedores;
```

Podemos saber la cantidad de productos distintas usamos:

```
SELECT COUNT(DISTINCT dispobible) FROM productos;
```

Podemos combinarla con "WHERE". Por ejemplo, queremos conocer los distintos productos de un determinado proveedor:

```
SELECT DISTINCT nombre FROM productos WHERE codigo_provee=4;
```

También puede utilizarse con "GROUP BY":

```
SELECT nombre, COUNT(DISTINCT nombre) FROM productos GROUP BY codigo_provee;
```

Para mostrar los cargos de los empleados sin repetición, usamos:

```
SELECT DISTINCT cargo FROM empleados ORDER BY cargo;
```

La cláusula "DISTINCT" afecta a todos los campos presentados. Para mostrar los nombres y cargos de los empleados sin repetir nombre ni cargos, usamos:

```
SELECT DISTINCT nombre, cargo FROM empleados ORDER BY nombre;
```

Note que los registros no están duplicados, aparecen nombres iguales pero con cargos diferentes, cada registro es diferente.

40.- Consultas a varias tablas INNER JOIN:

El comando INNER JOIN permite realizar una consulta a varias tablas que estén relacionadas en una sola ejecución. Podemos cruzar dos consultas en una para un resultado más completo. Recordamos que esto es posible sólo si las tablas están relacionadas. La relación la podemos establecer a través de una clave foránea como vimos anteriormente.

Hasta ahora en nuestro ejercicio hemos trabajado con el código de proveedor que se encuentra en la tabla de “productos” pero nuestro reporte está incompleto pues para saber cuál es el proveedor necesitamos consultar la tabla de “proveedores” y buscar el código que obtenemos en la consulta a la tabla “productos” y verificar el nombre del proveedor.

Podemos obtener todos los datos del producto y el todos los datos del proveedor en una sola consulta, ¿Cómo? Utilizando la siguiente sentencia:

```
SELECT *  FROM proveedores INNER JOIN productos ON
proveedores.codigo=productos.codigo_provee;
```

En la sentencia anterior utilizamos el comando INNER JOIN para unir la tabla de "proveedores" con la tabla de "productos", la instrucción ON señala la condición, en nuestro caso pedimos que la clave primaria de la tabla proveedores (codigo) sea igual a la clave foránea de la tabla productos (código_provee). Las claves tanto primaria como foránea están identificadas con el nombre de la tabla a la que pertenecen, separadas por un punto respectivamente, proveedores.codigo=productos.codigo_provee;

Podemos especificar los nombres de las columnas de ambas tablas con las que trabajaremos. Por ejemplo queremos el codigo del producto y el código del proveedor:

```
SELECT codigo, codigo_p FROM proveedores INNER JOIN productos ON
proveedores.codigo=productos.codigo_provee;
```

Nos devolverá dos columnas, la columna codigo sacada de la tabla "proveedores" y la columna código_p sacada de la tabla "productos".

Esta misma consulta la podemos escribir de forma equivalente cambiando la notación del INNER JOIN:

```
SELECT codigo, codigo_p FROM proveedores,productos WHERE
proveedores.codigo=productos.codigo_provee;
```

La coma (,) es equivalente a escribir INNER JOIN y WHERE es equivalente a escribir ON. No puede combinarse las notaciones. Si se trabaja con la (,) se utiliza WHERE, si se trabaja con INNER JOIN se utiliza ON.

¿Qué sucede si las columnas con las que queremos trabajar se llaman iguales en ambas tablas? Pues en este caso debemos identificar a que tabla pertenece cada columna. Por ejemplo queremos el nombre del proveedor y el nombre del producto. En ambos caso las columnas se llaman iguales, entonces nuestra consulta viene dada de la siguiente forma:

```
SELECT proveedores.nombre, productos.nombre FROM proveedores, productos WHERE
proveedores.codigo=productos.codigo_provee;
```

Al igual que las claves, colocamos el nombre de la tabla seguido por un punto y el nombre de la columna.

Este tipo de consulta permite utilizar todos los comandos y operadores que hemos visto anteriormente, Por ejemplo si queremos ver el total de productos por proveedores, ahora podemos identificar el proveedor utilizando la columna nombre de la tabla "proveedores"

```
SELECT proveedores.nombre, COUNTt( * ) FROM proveedores INNER JOIN productos ON
proveedores.codigo = productos.codigo_provee GROUP BY codigo_provee;
```

Si queremos el total de productos organizados por proveedor pero de aquellos que tengan más de dos productos:

```
SELECT proveedores.nombre, COUNT( * ) FROM proveedores INNER JOIN productos ON
proveedores.codigo = productos.codigo_provee GROUP BY codigo_provee HAVING COUNT(*) >=2;
```

Si queremos el total de productos organizados por proveedor pero que no incluya al proveedor "provee1":

```
SELECT proveedores.nombre, COUNT( * ) FROM proveedores INNER JOIN productos ON
proveedores.codigo = productos.codigo_provee GROUP BY codigo_provee HAVING
proveedores.nombre <> 'provee1';
```

También podemos unir más de dos tablas, por ejemplo si queremos imprimir un reporte de factura y queremos saber el nombre del producto, nombre del proveedor o fabricante, precio, cantidad, total a pagar y hora y fecha de la venta entonces tendríamos que unir la tabla “proveedores” con la tabla “productos” y finalmente la tabla ventas.

Pero tenemos un problema, la tabla “ventas” no tiene ninguna relación con las demás tablas, sólo tenemos la columna “cod_producto” pero no está referenciada, entonces en primer lugar vamos agregarle la columna “cod_provee” que será el código de proveedor o fabricante:

```
ALTER TABLE ventas ADD cod_provee INT;
```

Ahora vamos a definir las claves foráneas, en este caso tenemos dos, la clave foránea que hará referencia a la tabla “proveedores” y la clave foránea que hará referencia a la tabla “productos”.

Cave foránea referente a la tabla “proveedores”:

```
ALTER TABLE ventas ADD FOREING KEY(cod_provee) REFERENCES proveedores(codigo) ON DELETE
CASCADE ON UPDATE CASCADE;
```

Clave Foránea referente a la tabla “productos”:

```
ALTER TABLE ventas ADD FOREING KEY(cod_producto) REFERENCES productos(codigo_p) ON DELETE
CASCADE ON UPDATE CASCADE;
```

Tenemos un problema de redundancia de datos, la columna precio del producto la tenemos en la tabla productos y en la tabla ventas, debemos eliminarla de la tabla ventas porque gracias a la clave foránea ya podemos establecer una relación entre ambas tablas y con el INNER JOIN podemos tomar las columnas que queramos de ambas tablas. No sucede lo mismo con la columna cantidad ya que esta columna representa es la cantidad de productos que se está comprando y la cantidad de productos disponibles. Para eliminar la columna precio de la tabla “ventas” utilizamos la siguiente instrucción:

```
ALTER TABLE ventas DROP COLUMN precio;
```

Ahora vamos a crear nuestra consulta, recordemos como era: queremos saber el nombre del producto, nombre del proveedor o fabricante, precio, cantidad, total a pagar y hora y fecha de la venta.

El nombre del producto y el precio los obtenemos de la tabla producto

El nombre del proveedor lo obtenemos de la tabla proveedores

La cantidad la hora y fecha la obtenemos de la tabla “ventas”

El total a pagar lo obtenemos como una columna calculada.

Entonces nuestra instrucción sería:

```
SELECT proveedores.nombre, productos.nombre, ventas.precio, ventas.cantidad,
ventas.fecha_hora, precio*cantidad AS 'Total a pagar'FROM proveedores INNER JOIN
productos ON proveedores.codigo=productos.codigo_provee INNER JOIN ventas ON
proveedores.codigo=ventas.cod_provee AND productos.codigo_p=ventas.cod_producto;
```

En la consulta anterior hemos unido en primer lugar la tabla proveedores con la tabla productos a través de un INNER JOIN y verificamos que la clave primaria de la tabla “proveedores” (codigo) sea igual a la clave foránea de la tabla de “productos” (codigo_provee), luego esta consulta la unimos con la tabla “ventas” utilizando un segundo INNER JOIN y finalmente verificamos que la clave primaria de proveedores (codigo) sea igual a la primera clave foránea de ventas (cod_provee), luego verificamos que la clave primaria de productos (codigo_p) sea igual a la segunda clave foránea de ventas (cod_producto). Hemos terminado las consultas.