1 - Introducción

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

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

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

SISTEMA GESTOR DE BASES DE DATOS (SGBD)

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

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

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

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

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

2 – Lenguaje SQL

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

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

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

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

Comandos SQL

Existen dos tipos de comandos SQL:

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

Comandos DLL

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

Comandos DML

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

Cláusulas SQL

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

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

Operadores

Lógicos:

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

De comparación:

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

Funciones de agregado

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

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

MySQL

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

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

Ingresando instrucciones en la linea de comandos o embebidas en un lenguaje como PHP nos comunicamos con el servidor.

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

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

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

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

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

Un mensaje nos muestra la cantidad de registros eliminados. Con "**limit**" indicamos la cantidad a eliminar.

Veamos cuántos hay ahora:

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

Sólo queda 1.

46 - Recuperación de registros en forma aleatoria(rand)

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

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

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

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

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

47 - Agregar campos a una tabla (alter table - add)

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

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

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

Aprenderemos a agregar campos a una tabla.

Para ello utilizamos nuestra tabla "libros", definida con la siguiente estructura:

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

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

```
alter table libros  add cantidad smallint unsigned not null;
```

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

Agreguemos otro campo a la tabla:

```
alter table libros  add edicion date;
```

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

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

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

48 - Eliminar campos de una tabla (alter table - drop)

"**alter table**" nos permite alterar la estructura de la tabla, podemos usarla para eliminar un campo.

Continuamos con nuestra tabla "libros".

Para eliminar el campo "edicion" tipeamos:

```
alter table libros  drop edicion;
```

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

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

Podemos eliminar 2 campos en una misma sentencia:

```
alter table libros  drop editorial, drop cantidad;
```

Si se borra un campo de una tabla que es parte de un índice, también se borra el índice.

Si una tabla tiene sólo un campo, éste no puede ser borrado.

Hay que tener cuidado al eliminar un campo, éste puede ser clave primaria. Es posible eliminar un campo que es clave primaria, no aparece ningún mensaje:

```
alter table libros  drop codigo;
```

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

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

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

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

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

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

```
alter table libros  modify cantidad smallint unsigned;
```

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

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

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

Hay que tener cuidado al alterar los tipos de los campos de una tabla que ya tiene registros cargados. Si tenemos un campo de texto de longitud 50 y lo cambiamos a 30 de longitud, los registros cargados en ese campo que superen los 30 caracteres, se cortarán.

Igualmente, si un campo fue definido permitiendo valores nulos, se cargaron registros con valores nulos y luego se lo define "**not null**", todos los registros con valor nulo para ese campo cambiarán al valor por defecto según el tipo (cadena vacía para tipo texto y 0 para numéricos), ya que "**null**" se convierte en un valor inválido.

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

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

```
alter table libros  modify codigo int unsigned auto_increment;
```

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

```
alter table libros  alter autor set default 'Varios';
```

Para eliminar el valor por defecto podemos emplear:

```
alter table libros  alter autor drop default;
```

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

Con "alter table" podemos cambiar el nombre de los campos de una tabla.

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

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

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

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

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

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

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

51 - Agregar y eliminar la clave primaria (alter table)

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

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

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

Para agregar una clave primaria a una tabla existente usamos:

```
alter table libros  add primary key (codigo);
```

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

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

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

También usamos "**alter table**" para eliminar una clave primaria.

Para eliminar una clave primaria usamos:

```
alter table libros  drop primary key;
```

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

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

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

52 - Agregar índices(alter table - add index)

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

Creamos la tabla "libros":

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

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

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

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

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

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

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

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

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

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

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

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

Para eliminar un índice usamos la siguiente sintaxis:

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

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

Para eliminar un índice único usamos la misma sintaxis:

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

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

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

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

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

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

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

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

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

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

55 - Tipo de dato enum

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

El tipo de dato "**enum**" representa una enumeración. Puede tener un máximo de 65535 valores distintos. Es una cadena cuyo valor se elige de una lista enumerada de valores permitidos que se especifica al definir el campo. Puede ser una cadena vacía, incluso "**null**". Los valores presentados como permitidos tienen un valor de índice que comienza en 1.

Una empresa necesita personal, varias personas se han presentado para cubrir distintos cargos. La empresa almacena los datos de los postulantes a los puestos en una tabla llamada "postulantes". Le interesa, entre otras

select trim (**trailing** '0' **from** '00hola00');
Retorna: "00hola".
select trim (**both** '0' **from** '00hola00');
Retorna: "hola".
select trim ('0' **from** '00hola00');
Retorna: "hola".

- **replace**(cadena,cadenareemplazo,cadenareemplazar): retorna la cadena con todas las ocurrencias de la subcadena reemplazo por la subcadena a reemplazar.
  Ejemplo: **select replace**('xxx.mysql.com','x','w');
  Retorna: "www.mysql.com'.
- **repeat**(cadena,cantidad): devuelve una cadena consistente en la cadena repetida la cantidad de veces especificada. Si "cantidad" es menor o igual a cero, retorna una cadena vacía.
  Ejemplo: **select repeat**('hola',3);
  Retorna: "holaholahola".
- **reverse**(cadena): devuelve la cadena invirtiendo el orden de los caracteres.
  Ejemplo: **select reverse**('Hola');
  Retorna: "aloH".
- **insert**(cadena,posicion,longitud,nuevacadena): retorna la cadena con la nueva cadena colocándola en la posición indicada por "posicion" y elimina la cantidad de caracteres indicados por "longitud".
  Ejemplo: **select insert**('buenas tardes',2,6,'xx');
  Retorna: "bxxtardes".
- **lcase**(cadena) y **lower**(cadena): retornan la cadena con todos los caracteres en minúsculas.
  Ejemplo: **select lower**('HOLA ESTUDIAnte');
  Retorna: "hola estudiante".
  select lcase('HOLA ESTUDIAnte');
  Retorna: "hola estudiante".
- **ucase**(cadena) y **upper**(cadena): retornan la cadena con todos los caracteres en mayúsculas.
  Ejemplo: **select upper**('HOLA ESTUDIAnte');
  Retorna: "HOLA ESTUDIANTE".
  select ucase('HOLA ESTUDIAnte');
  Retorna: "HOLA ESTUDIANTE".
- **strcmp**(cadena1,cadena2): retorna 0 si las cadenas son iguales, -1 si la primera es menor que la segunda y 1 si la primera es mayor que la segunda.
  Ejemplo: **select strcmp**('Hola','Chau');
  Retorna: 1.

59 - Funciones matemáticas

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

Ejemplo: **select** 5/0;

RECUERDE que NO debe haber espacios entre un nombre de función y los paréntesis porque MySQL puede confundir una llamada a una función con una referencia a una tabla o campo que tenga el mismo nombre de una función.

MySQL tiene algunas funciones para trabajar con números. Aquí presentamos algunas.

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

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

Todas retornan null en caso de error.

60 - Funciones para el uso de fecha y hora

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

- **adddate**(fecha, **interval** expresión **tipo**): retorna la fecha agregándole el intervalo especificado.
  Ejemplos: **adddate**('2006-10-10',**interval** 25 **day**)
  Retorna: "2006-11-04".
  adddate('2006-10-10',**interval** 5 **month**)
  Retorna: "2007-03-10".
- **addtime**(expresion1,expresion2): agrega expresion2 a expresion1 y retorna el resultado.
- **current_date**: retorna la fecha de hoy con formato "YYYY-MM-DD" o "YYYYMMDD".
- **current_time**: retorna la hora actual con formato "HH:MM:SS" o "HHMMSS".
- **date_add**(fecha,**interval** 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".

- **datediff**(fecha1,fecha2): retorna la cantidad de días entre fecha1 y fecha2.
- **dayname**(fecha): retorna el nombre del día de la semana de la fecha.
  Ejemplo: **dayname**('2006-08-10')
  Retorna: "thursday".
- **dayofmonth**(fecha): retorna el día del mes para la fecha dada, dentro del rango 1 a 31.
  Ejemplo: **dayofmonth**('2006-08-10')
  Retorna: 10.
- **dayofweek**(fecha): retorna el índice del día de semana para la fecha pasada como argumento. Los valores de los índices son: 1=domingo, 2=lunes,... 7=sábado).
  Ejemplo: **dayofweek**('2006-08-10')
  Retorna: 5, o sea jueves.
- **dayofyear**(fecha): retorna el día del año para la fecha dada, dentro del rango 1 a 366.
  Ejemplo: **dayofmonth**('2006-08-10')
  Retorna: 222.
- **extract**(**tipo from** fecha): extrae partes de una fecha.
  Ejemplos: **extract**(**year from** '2006-10-10'),
  Retorna: "2006".
  extract(**year_month from** '2006-10-10 10:15:25')
  Retorna: "200610".
  extract(**day_minute from** '2006-10-10 10:15:25')
  Retorna: "101015";
- **hour**(hora): retorna la hora para el dato dado, en el rango de 0 a 23.
  Ejemplo: **hour**('18:25:09');
  Retorna "18".
- **minute**(hora): retorna los minutos de la hora dada, en el rango de 0 a 59.
- **monthname**(fecha): retorna el nombre del mes de la fecha dada.
  Ejemplo: **monthname**('2006-08-10')
  Retorna: "August".
- **month**(fecha): retorna el mes de la fecha dada, en el rango de 1 a 12.
- **now**() y **sysdate**(): retornan la fecha y hora actuales.
- **period_add**(p,n): agrega "n" meses al periodo "p", en el formato "YYMM" o "YYYYMM"; retorna un valor en el formato "YYYYMM". El argumento "p" no es una fecha, sino un año y un mes.
  Ejemplo: **period_add**('200608',2)
  Retorna: "200610".
- **period_diff**(p1,p2): retorna el número de meses entre los períodos "p1" y "p2", en el formato "YYMM" o "YYYYMM". Los argumentos de período no son fechas sino un año y un mes.
  Ejemplo: **period_diff**('200608','200602')
  Retorna: 6.
- **second**(hora): retorna los segundos para la hora dada, en el rango de 0 a 59.
- **sec_to_time**(segundos): retorna el argumento "segundos" convertido a horas, minutos y segundos.
  Ejemplo: **sec_to_time**(90)
  Retorna: "1:30".
- **timediff**(hora1,hora2): retorna la cantidad de horas, minutos y segundos entre hora1 y hora2.
- **time_to_sec**(hora): retorna el argumento "hora" convertido en segundos.
- **to_days**(fecha): retorna el número de día (el número de día desde el año 0).
- **weekday**(fecha): retorna el índice del día de la semana para la fecha pasada como argumento. Los índices son: 0=lunes, 1=martes,... 6=domingo).
  Ejemplo: **weekday**('2006-08-10') ;
  Retorna: 3, o sea jueves.
- **year**(fecha): retorna el año de la fecha dada, en el rango de 1000 a 9999.
  Ejemplo: **year**('06-08-10')
  Retorna: "2006".

Los valores para "**tipo**" pueden ser: **second**, **minute**, **hour**, **day**, **month**, **year**, **minute_second** (minutos y segundos), **hour_minute** (horas y minutos), **day_hour** (días y horas), **year_month** (año y mes), hour_second (hora, minuto y segundo), day_minute (dias, horas y minutos), **day_second** (dias a segundos).

61 - Funciones de control de flujo (if)

Trabajamos con las tablas "libros" de una librería. No nos interesa el precio exacto de cada libro, sino si el precio es menor o mayor a $50. Podemos utilizar estas sentencias:

```
select titulo from libros where precio<50;

select titulo from libros where precio >=50;
```

En la primera sentencia mostramos los libros con precio menor a 50 y en la segunda los demás.

También podemos usar la función "**if**".

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

Veamos el ejemplo:

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

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

Veamos otros ejemplos. Queremos mostrar los nombres de los autores y la cantidad de libros de cada uno de ellos; para ello especificamos el nombre del campo a mostrar ("autor"), contamos los libros con "autor" conocido con la función "**count**()" y agrupamos por nombre de autor:

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

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

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

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

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

Si la cantidad de libros de cada autor es mayor a 1 (primer argumento del "**if**"), coloca "Más de 1" (segundo argumento del "**if**"), en caso contrario coloca "1" (tercer argumento del "**if**").

Queremos saber si la cantidad de libros por editorial supera los 4 o no:

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

Si la cantidad de libros de cada editorial es mayor a 4 (primer argumento del "**if**"), coloca "5 o más" (segundo argumento del "**if**"), en caso contrario coloca "menos de 5" (tercer argumento del "**if**").

62 - Funciones de control de flujo (case)

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

Trabajemos con la tabla "libros" de una librería.Queremos saber si la cantidad de libros de cada editorial es menor o mayor a 1, usamos:

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

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

```
select editorial,
```

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

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

Entonces, la sintaxis es:

```
case
    when … then …
    ...
    else ….
end
```

Se puede obviar la parte "**else**":

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

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

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

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

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

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

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

Pero existe otra sintaxis "**case when**" que permite condiciones:

```
case when  then  ...  else  end
```

Veamos un ejemplo:

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

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

63 - Varias tablas (join)

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

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

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



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

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

Al recuperar los datos de los libros:

```
select * from libros;
```

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

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

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

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

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

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

```
select * from libros join editoriales;
```

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

Note que en la consulta

```
join libros as l  on e.codigo=l.codigoeditorial;
```

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

70 - join, group by y funciones de agrupamiento.

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

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

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

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

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

Solamente mostrará las editoriales para las cuales encuentra valores coincidentes para el código de la editorial en la tabla "libros".

Para conocer el mayor precio de los libros de cada editorial usamos la función "**max**()", hacemos una unión y agrupamos por nombre de la editorial:

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

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

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

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

71 - join con más de dos tablas.

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

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

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

```
create table libros(
            codigo int unsigned auto_increment,
            titulo varchar(40) not null,
            autor varchar(20) default 'Desconocido',
      primary key (codigo)
 );
create table socios(
            documento char(8) not null,
            nombre varchar(30),
            domicilio varchar(30),
      primary key (numero)
 );
```

73 - Variables de usuario.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

select @editorial:=e.nombre  from libros as l
join editoriales as e  on l.codigoeditorial=e.codigo
where precio=@mayorprecio;

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

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

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

La tabla "libros" tiene esta estructura:

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

- clave primaria: codigo.

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

- nombre: nombre de la editorial.

La tabla libros contiene varios registros.

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

1º paso: crear la tabla "editoriales":

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

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

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

Obteniendo una salida como la siguiente:

```
editorial
_________
Emece
Paidos
Planeta
```

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

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

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

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

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

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

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

```
nombre
______
Emece
Paidos
Planeta
```

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

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

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

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

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

Podemos lograrlo en 3 pasos:

1º paso: crear la tabla "cantidadporeditorial":

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

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

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

Obteniendo una salida como la siguiente:

```
nombre      cantidad
_____________________
Emece       3
Paidos      4
Planeta     2
```

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

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

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

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

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

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

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

```
nombre      cantidad
_____________________
Emece       3
Paidos      4
Planeta     2
```

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

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

La tabla "editoriales" tiene esta estructura:

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

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

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

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

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

1º paso: crear la tabla "cantidadporeditorial":

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

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

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

Obteniendo una salida como la siguiente:

```
nombre          cantidad
_____________________
Emece           3
Paidos          4
Planeta         2
```

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

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

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

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

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

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

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

```
nombre          cantidad
_____________________
Emece           3
Paidos          4
Planeta         2
```

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene la siguiente estructura:

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

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

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

Queremos ingresar en "libros", el siguiente libro:

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

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

Podemos lograrlo en 2 pasos:

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

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

nos devuelve el valor "2".

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

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

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

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

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

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

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

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

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

Veamos un ejemplo. Queremos ingresar el siguiente registro:

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

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

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

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

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

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene la siguiente estructura:

- nombre: varchar(20).

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

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

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

obteniendo una salida como la siguiente:

```
editorial
_________
Emece
Paidos
Planeta
```

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

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

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

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

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

También podemos crear una tabla llamada "cantidadporeditorial":

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

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

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

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

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

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

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

La tabla "editoriales":

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

La tabla "cantidadporeditorial":

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

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

Podemos lograrlo en 2 pasos:

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

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

obteniendo una salida como la siguiente:

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

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

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

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

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

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

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene esta estructura:

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

Ambas tablas contienen registros.

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

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

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

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

Ahora debemos actualizar los valores para ese campo.

Podemos hacerlo en 2 pasos:

1º paso: consultamos los códigos de las editoriales:

```
select codigo,nombre
from editoriales;
```

obtenemos una salida similar a la siguiente:

```
codigo    nombre
_____________________
1         Planeta
2         Emece
3         Paidos
```

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

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

... con cada editorial...

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

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

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

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

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

80 - Actualización en cascada (update - join)

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

Las tablas tienen las siguientes estructuras:

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


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

Ambas tablas contienen registros.

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

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

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

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

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

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

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

```
update libros
set codigoeditorial=9
where codigoeditorial=2;
```

O podemos hacerlo en una sola sentencia:

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

El cambio se realizó en ambas tablas.

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene esta estructura:

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

Ambas tablas contienen registros.

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

Podemos hacerlo en 2 pasos:

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

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

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

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

```
delete libros
where codigoeditorial=2;
```

O podemos realizar todo *en un solo paso:*

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

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

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

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene esta estructura:

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

Ambas tablas contienen registros.

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

Podemos hacerlo en 2 pasos:

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

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

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

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

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

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

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

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

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

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

83 - Borrar registros en cascada (delete - join)

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

La tabla "libros" tiene la siguiente estructura:

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

La tabla "editoriales" tiene esta estructura:

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

Ambas tablas contienen registros.

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

Podemos hacerlo en 2 pasos:

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

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

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

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

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

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

O podemos hacerlo en una sola consulta:

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

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

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

84 - Chequear y reparar tablas (check - repair)

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

```
check table libros;
```

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

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

Los tipos de chequeo son:

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

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

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

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

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

```
repair table libros;
```

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

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

85 - Encriptación de datos (encode - decode)

Las siguientes funciones encriptan y desencriptan valores.

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

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

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

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

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

```
Select decode('§¡Ý7','dia');
```

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

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

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

Podemos ingresar registros encriptando la clave:

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

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

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

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

Veamos los registros ingresados:

```
select * from usuarios;
```

Desencriptamos la clave del usuario "MarioPerez":

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

Desencriptamos la clave del usuario "MariaGarcia":

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

86 – Clave Foránea (foreign key)

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

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

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

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

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

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

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

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

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

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

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

Las columnas correspondientes en la clave foránea y en la clave referenciada deben tener tipos de datos similares para que puedan ser comparadas sin la necesidad de hacer una conversión de tipos.
El tamaño y el signo de los tipos enteros debe ser el mismo. En las columnas de tipo caracter, el tamaño no tiene que ser el mismo necesariamente.
Si MySQL da un error cuyo número es el 1005 al momento de ejecutar una sentencia CREATE TABLE, y el mensaje de error se refiere al número 150, la creación de la tabla falló porque la
restricción de la clave foránea no se hizo de la manera adecuada.
De la misma manera, si falla una sentencia ALTER TABLE y se hace referencia al error número 150, esto significa que la definición de la restricción de la clave foránea no se hizo adecuadamente.