MANUAL DE COMANDOS para MySQL

"Para entender este manual se debe tener una idea base y mínima sobre las Bases de Datos Relacionales"

Xema Aceituno

ÍNDICE

Introducción a MySQL

A nivel teórico, existen 3 lenguajes para el manejo de bases de datos:

DDL (Data Definition Language) Que es el lenguaje de definición de datos. Es el lenguaje que se usa para crear bases de datos y tablas, y para modificar sus estructuras, así como los permisos y privilegios.
Este lenguaje trabaja sobre unas tablas especiales llamadas *diccionario de datos*.

DML (Data Manipulation Language) que es el lenguaje de manipulación de datos. Es el que se usa para modificar y obtener datos desde las bases de datos.

DCL (Data Control Language) que es el lenguaje de control de datos, que incluye una serie de comandos que permiten al administrador controlar el acceso a los datos contenidos en la base de datos.

A continuación vamos a ver las instrucciones más básicas en MySQL divididas en secciones. Pero antes de empezar te enseñaremos 2 auxiliares que te vendrán bien para comprobar los resultados de lo que vayas haciendo;

SELECT * FROM nombretabla;
se usa para ver el resultado de una tabla con todos sus registros.

DESCRIBE nombretabla;
se usa para ver la estructura de una tabla.

Crear una base de datos

CREATE DATABASE;

SHOW DATABASES;

USE prueba;

Desde el punto de vista de SQL, una base de datos es sólo un conjunto de relaciones (o tablas). A nivel de sistema operativo, cada base de datos se guarda en un directorio diferente.

Por tanto, crear una base de datos es una tarea muy simple. Claro que, en el momento de crearla, la base de datos estará vacía, es decir, no contendrá ninguna tabla.

Para crear una base de datos se usa una sentencia **CREATE DATABASE:**

```
mysql>CREATE DATABASE prueba;
Query OK, 1 row affected (0.03 sec)
```

Para saber cuántas bases de datos existen en nuestro sistema usamos la sentencia **SHOW DATABASES:**

```
mysql>SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| mysql              |
| prueba             |
| test               |
+--------------------+
3 rows in set (0.00 sec)
```

A partir de ahora, en los próximos capítulos, trabajaremos con esta base de datos, por lo tanto la seleccionaremos como base de datos por defecto.. Para seleccionar una base de datos concreta se usa el comando USE que no es exactamente una sentencia SQL, sino más bien de una opción de MySQL. Esto nos permitirá obviar el nombre de la base de datos en consultas:

```
mysql>USE prueba;
Database changed
```

Crear una tabla

CREATE TABLE ...
SHOW TABLES;

Veamos ahora la sentencia **CREATE TABLE** que sirve para crear tablas.
La sintaxis de esta sentencia es muy compleja, pero empezaremos con ejemplos sencillos.

En su forma más simple, la sentencia **CREATE TABLE** creará una tabla con las columnas que indiquemos. Crearemos, como ejemplo, una tabla que nos permitirá almacenar nombres de personas y sus fechas de nacimiento. Deberemos indicar el nombre de la tabla y los nombres y tipos de las columnas:

```
mysql> USE prueba
Database changed
mysql>CREATE TABLE gente (nombre VARCHAR(40), fecha DATE);
Query OK, 0 rows affected (0.53 sec)
```

Hemos creado una tabla llamada "gente" con dos columnas: "nombre" que puede contener cadenas de hasta 40 caracteres y "fecha" de tipo fecha.
Al final de este documento puedes ver algunos de los tipos de datos en MySQL.
(Pincha aquí → Diferentes tipos de dato con formato string)

Podemos consultar cuántas tablas y qué nombres tienen en una base de datos, usando la sentencia **SHOW TABLES:**

```
mysql>SHOW TABLES;
+------------------+

| Tables_in_prueba |
+------------------+
| gente            |
+------------------+
1 row in set (0.01 sec)
```

Además del tipo y el nombre, podemos definir valores por defecto, permitir o no que contengan valores nulos, crear una clave primaria, indexar...

La sintaxis para definir columnas es:

```
nombre_col tipo [NOT NULL | NULL] [DEFAULT valor_por_defecto]
[AUTO_INCREMENT] [[PRIMARY] KEY] [COMMENT 'string']
[definición_referencia]
```

Veamos cada una de las opciones por separado...

Valores nulos

… **NOT NULL / NULL …**

Al definir cada columna/atributo podemos decidir si podrá o no contener valores nulos.

Recordemos que aquellas columnas que son o forman parte de una clave primaria no pueden contener valores nulos.

Después veremos que si definimos una columna como clave primaria, automáticamente se impide que pueda contener valores nulos, pero este no es el único caso en que puede ser interesante impedir la asignación de valores nulos para una columna.

Cuando creamos una tabla la opción por defecto es que se permitan los valores nulos (***NULL)*** y para que no se permitan, se especifica con ***NOT NULL***. Por ejemplo:

```
mysql>CREATE TABLE ciudad1 (nombre CHAR(20) NOT NULL, poblacion INT NULL);
Query OK, 0 rows affected (0.98 sec)
```

Valores por defecto

… DEFAULT …

Para cada columna/atributo también se puede definir, opcionalmente, un valor por defecto. El valor por defecto se asignará de forma automática a un atributo cuando no se especifique un valor determinado al añadir filas. Para ello usamos el comando DEFAULT dentro de la sentencia CREATE TABLE.

Si a un atributo se le declara que puede ser nulo y no se especifica un valor, por defecto, se usará NULL como valor por defecto. En el ejemplo anterior, el valor por defecto para *poblacion* es NULL.

Por ejemplo, si queremos que el valor por defecto para *poblacion* sea 5000, podemos crear la tabla como:

```
mysql>CREATE TABLE ciudad2 (nombre CHAR(20) NOT NULL, poblacion INT NULL
DEFAULT 5000);
Query OK, 0 rows affected (0.09 sec)
```

Claves Primarias

... PRIMARY KEY...

Para *claves primarias t*ambién se puede definir una clave primaria sobre un atributo, usando la palabra clave *KEY* o *PRIMARY KEY*.

Sólo puede existir una clave primaria en cada tabla, y el atributo sobre el que se define una clave primaria no puede tener valores *NULL*. Si esto no se especifica de forma explícita, **MySQL** lo hará de forma automática.
Por ejemplo, si queremos asignar la clave primaria a "*nombre"* de la tabla de ciudades, crearemos la tabla así:

```
mysql>CREATE TABLE ciudad3 (nombre CHAR(20) NOT NULL PRIMARY KEY, poblacion
INT NULL DEFAULT 5000);
Query OK, 0 rows affected (0.20 sec)
```

Usar **NOT NULL PRIMARY KEY** equivale a **PRIMARY KEY**, **NOT NULL KEY** o sencillamente **KEY**.

Existe una sintaxis alternativa para crear claves primarias, que en general es preferible, ya que es más potente. De hecho, la anterior es un alias para la forma general, que no admite todas las funciones (como por ejemplo, crear claves primarias sobre varias columnas). Veremos esta otra alternativa más adelante.

Claves Primarias Compuestas

... PRIMARY KEY (clave1, clave2, ...)

Las *claves primarias compuestas* se usan cuando no hay ningún atributo en la tabla que identifique cada registro de manera única. Es posible que un conjunto de atributos pueda identificar cada registro de manera única, en estos casos utilizamos claves primarias compuestas.

En MySQL para establecer que una tabla tenga una clave compuesta, lo indicamos al final del create table:

```
mysql>mysql> create table ButacaCine (
    -> sala int not null,
    -> num_butaca int not null,
    -> color varchar(50),
    -> primary key(sala, num_butaca)
    -> );

Query OK, 0 rows affected (0.11 sec)

mysql> describe ButacaCine;

+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| sala            | int(11)     | NO   | PRI | NULL    |       |
| num_butaca      | int(11)     | NO   | PRI | NULL    |       |
| nombre_pelicula | varchar(50) | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
3 rows in set (0.04 sec)
```

En este ejemplo formamos la clave primaria con el número de sala y el número de butaca para identificar un asiento concreto en un cine.

En las claves primarias, solo 1 de las columnas puede tener la propiedad auto_increment.

Columnas Autoincrementadas

... AUTO_INCREMENT ...

Para *columnas autoincrementadas* en **MySQL** tenemos la posibilidad de crear una columna autoincrementada, aunque esta columna sólo puede ser de tipo entero.

Si al insertar una fila se omite el valor de la columna autoincrementada o si se inserta un valor nulo para esa columna, su valor se calcula automáticamente, tomando el valor más alto de esa columna y sumándole una unidad. Esto permite crear, de una forma sencilla, una columna con un valor único para cada fila de la tabla.

Generalmente, estas columnas se usan como claves primarias 'artificiales'. **MySQL** está optimizado para usar valores enteros como claves primarias, de modo que la combinación de clave primaria, que sea entera y autoincrementada es ideal para usarla como clave primaria artificial:

```
mysql>CREATE TABLE ciudad5 (clave INT AUTO_INCREMENT PRIMARY KEY,
-> nombre CHAR(20) NOT NULL,
-> poblacion INT NULL DEFAULT 5000);
Query OK, 0 rows affected (0.11 sec)
```

Para establecer un número específico del que partiren el auto_increment hay que crear la tabla de la siguiente manera:

```
mysql> create table prueba (id int key auto_increment, name
varchar(20)) auto_increment=20;
Query OK, 0 rows affected (0.28 sec)

mysql> describe prueba;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(20) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)
```

Comentarios

... COMMENT...

Para los *comentarios*, adicionalmente, al crear la tabla, podemos añadir un comentario a cada columna. Este comentario sirve como información adicional sobre alguna característica especial de la columna, y entra en el apartado de documentación de la base de datos:

```
mysql>CREATE TABLE ciudad6
-> (clave INT AUTO_INCREMENT PRIMARY KEY COMMENT 'Clave principal',
-> nombre CHAR(50) NOT NULL,
-> poblacion INT NULL DEFAULT 5000);
Query OK, 0 rows affected (0.08 sec)
```

Borrar una tabla

DROP TABLE ...

Para borrar una tabla usamos **DROP TABLE** de manera sencilla:

```
mysql> show tables;
+-----------------+
| Tables_in_mundo |
+-----------------+
| lista           |
| naciones        |
| prueba          |
| telef           |
+-----------------+
4 rows in set (0.00 sec)

mysql> DROP TABLE telef;
Query OK, 0 rows affected (0.32 sec)

mysql> show tables;
+-----------------+
| Tables_in_mundo |
+-----------------+
| lista           |
| naciones        |
| prueba          |
+-----------------+
3 rows in set (0.00 sec)
Query OK, 0 rows affected (0.08 sec)
```

Modificar estructura de una tabla

ALTER TABLE ... RENAME...
ALTER TABLE ... DROP COLUMN...
ALTER TABLE ... DROP PRIMARY KEY...
ALTER TABLE ... ADD...
ALTER TABLE ... ADD...AFTER...
ALTER TABLE ... ADD...FIRST;
ALTER TABLE ... ADD...PRIMARY KEY...
ALTER TABLE ... MODIFY COLUMN...

Para modificar una tabla podemos usar el comando **ALTER TABLE:**
Por ejemplo, si queremos cambiar el nombre de la tabla:

```
mysql>ALTER TABLE ciudad6 RENAME granciudad;
Query OK, 0 rows affected (0.03 sec)
```

Para eliminar una columna de una tabla:

```
mysql>ALTER TABLE granciudad DROP COLUMN poblacion;
Query OK, 0 rows affected (1.28 sec)
Records: 0 Duplicates: 0  Warnings: 0
```

Para eliminar varias columnas seguiríamos la misma sintaxis separando cada columna con una coma: ALTER TABLE nombre_tabla DROP COLUMN nombre1, DROP COLUMN nombre2;

Para eliminar la clave primaria de una columna:

```
mysql>ALTER TABLE granciudad DROP PRIMARY KEY;
Query OK, 1 row affected (1.20 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

Para insertar una nueva columna al final de una tabla:

```
mysql>ALTER TABLE granciudad ADD fecha date;
Query OK, 1 row affected (1.20 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

Para añadir una nueva columna después de otra:

```
mysql>ALTER TABLE granciudad ADD origen VARCHAR(50) AFTER nombre;
Query OK, 0 rows affected (0.70 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Para añadir una nueva columna en la primera posición de la tabla:

```
mysql>ALTER TABLE granciudad ADD id INT  FIRST;
Query OK, 0 rows affected (0.72 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Asignar como clave primaria a una columna:

```
mysql>ALTER TABLE granciudad ADD PRIMARY KEY(id);
```

También podemos reestructurar una columna:

```
mysql>ALTER TABLE granciudad MODIFY COLUMN id INT auto_increment;
```

En este caso le hemos dotado a la columna id de la propiedad de autoincrementarse automáticamente.

En este caso le hemos dotado a la columna id de la propiedad de autoincrementarse automáticamente. Cuando se añade una columna AUTO_INCREMENT, los valores de columna se llenan con una secuencia numérica automáticamente.

Se puede establecer el auto_increment usando ALTER TABLE de la siguiente manera:

```
mysql> alter table granciudad auto_increment = 40;
Query OK, 0 rows affected (0.08 sec)
```

O también con la opción **SET INSERT_ID**=value

```
mysql> set insert_ID=50;
Query OK, 0 rows affected (0.00 sec)
```

Inserción de Filas

INSERT INTO... VALUES(...);
INSERT INTO... SET...

La forma más directa de insertar una fila nueva en una tabla es mediante una sentencia **INSERT**. En la forma más simple de esta sentencia debemos indicar la tabla a la que queremos añadir filas, y los valores de cada columna. Las columnas de tipo cadena o fechas deben estar entre comillas sencillas o dobles, para las columnas númericas esto no es imprescindible, aunque también pueden estar entrecomilladas.

```
mysql>INSERT INTO granciudad VALUES (1, 1, “barcelona”, “barcino”, “524-1-5”);
Query OK, 1 row affected (0.07 sec)
```

Introducir más de una fila a la vez:

```
mysql>INSERT INTO granciudad VALUES
→ (1, 1, “barcelona”, “barcino”, “524-1-5”),
→(2,2, “madrid”, “madrilus”, 1210-3-4”);
Query OK, 2 rows affected (0.09 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

También es posible especificar la lista de columnas/atributos donde deseamos ingresar ciertos valores. La ventaja de esta forma es que no precisamos de conocer el orden exacto de los atributos en la tabla:

```
mysql>INSERT INTO granciudad (id, nombre) VALUES (4, "zaragoza");
Query OK, 1 row affected (0.13 sec)

mysql> SELECT * from granciudad;
+------+------+-----------+----------+------------+
| id2  | id   | nombre    | origen   | fecha      |
+------+------+-----------+----------+------------+
|    1 |    1 | barcelona | barcino  | 2011-01-01 |
|    2 |    2 | madrid    | madrilus | 1111-01-02 |
|    3 |    3 | valencia  | valentia | 3214-01-03 |
| NULL |    4 | zaragoza  | NULL     | NULL       |
+------+------+-----------+----------+------------+
4 rows in set (0.00 sec)
```

Además existe una alternativa que consiste en indicar un valor para cada columna:

```
mysql>INSERT INTO granciudad SET
    -> id = 5,
    ->nombre = "Bilbao";
Query OK, 1 row affected (0.08 sec)

mysql> select * from granciudad;
+------+------+-----------+----------+------------+
| id2  | id   | nombre    | origen   | fecha      |
+------+------+-----------+----------+------------+
|    1 |    1 | barcelona | barcino  | 2011-01-01 |
|    2 |    2 | madrid    | madrilus | 1111-01-02 |
|    3 |    3 | valencia  | valentia | 3214-01-03 |
| NULL |    4 | zaragoza  | NULL     | NULL       |
| NULL |    5 | Bilbao    | NULL     | NULL       |
+------+------+-----------+----------+------------+
5 rows in set (0.01 sec)
```

Reemplazar Filas

REPLACE INTO ... VALUES(...);

Con la sentencia **REPLACE** podemos modificar un registro anterior. Realizamos un ejemplo, si partimos de esta tabla:

```
mysql> select * from granciudad;
+----+----------+----------+------------+
| id | nombre   | origen   | fecha      |
+----+----------+----------+------------+
|  2 | madrid   | madrilus | 1111-01-02 |
|  3 | valencia | valentia | 3214-01-03 |
|  4 | zaragoza | NULL     | NULL       |
|  5 | Bilbao   | NULL     | NULL       |
+----+----------+----------+------------+
4 rows in set (0.00 sec)
```

Siendo “id” la PRIMARY KEY, queremos reemplazar la columna ‘origen’ de Zaragoza de NULL a ‘caesaraugusta’:

```
mysql>REPLACE INTO granciudad (id, nombre, origen) VALUES
(4,'zaragoza','caesaraugusta');
Query OK, 2 rows affected (0.03 sec)

mysql> SELECT * FROM granciudad;
+----+----------+---------------+------------+
| id | nombre   | origen        | fecha      |
+----+----------+---------------+------------+
|  2 | madrid   | madrilus      | 1111-01-02 |
|  3 | valencia | valentia      | 3214-01-03 |
|  4 | zaragoza | caesaraugusta | NULL       |
|  5 | Bilbao   | NULL          | NULL       |
+----+----------+---------------+------------+
4 rows in set (0.00 sec)
```

Observamos que se hemos cambiado el valor deseado en la fila con id=4. Has de tener presente que el comando REPLACE puede realizar una inserción de fila del mismo modo que lo hace el comando INSERT INTO.

Actualizar Filas

UPDATE ... SET ...
UPDATE ... SET ... WHERE ...

Con la sentencia **UPDATE** podemos actualizar los valores de las filas. Los cambios se aplicarán a las filas y columnas que especifiquemos.

```
mysql>UPDATE granciudad SET origen='ciudad_iberica';
Query OK, 5 rows affected (0.09 sec)
Rows matched: 5  Changed: 5  Warnings: 0

mysql> SELECT * FROM granciudad;
+----+----------+---------------+------------+
| id | nombre   | origen        | fecha      |
+----+----------+---------------+------------+
|  2 | madrid   | ciudad_iberica | 1111-01-02 |
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | ciudad_iberica | NULL       |
|  5 | Bilbao   | ciudad_iberica | NULL       |
|  6 | malaga   | ciudad_iberica | NULL       |
+----+----------+---------------+------------+
5 rows in set (0.00 sec)
```

Como observamos, se han modificado todas las filas. Pero podemos usar la cláusula **WHERE** para establecer un filtro de columnas donde queremos que se realicen los cambios. Ejemplo:

```
mysql>UPDATE granciudad SET origen='caesaraugusta' WHERE nombre =
'Zaragoza';
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM granciudad;
+----+----------+---------------+------------+
| id | nombre   | origen        | fecha      |
+----+----------+---------------+------------+
|  2 | madrid   | ciudad_iberica | 1111-01-02 |
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | caesaraugusta  | NULL       |
|  5| Bilbao   | ciudad_iberica | NULL       |
|  6 | malaga   | ciudad_iberica | NULL       |
+----+----------+---------------+------------+
5 rows in set (0.00 sec)
```

En este ejemplo el UPDATE solo ha afectado a la fila de Zaragoza tal y como le hemos especificado en la cláusula WHERE.

Eliminar Filas

DELETE FROM…
DELETE FROM … WHERE…
DELETE FROM … WHERE…ORDER BY … ASC/DESC;
DELETE FROM … ORDER BY … ASC/DESC… LIMIT…;

Con la sentencia **DELETE** podemos eliminar filas de la tabla:

```
mysql>DELETE FROM ciudad3;
Query OK, 5 rows affected (0.05 sec)
```

En este caso se han eliminado TODAS las filas de la tabla 'ciudades'.

Si deseamos eliminar solo determinadas filas usamos la cláusula WHERE:
filas.

```
mysql> SELECT * FROM granciudad;
+----+----------+----------------+------------+
| id | nombre   | origen         | fecha      |
+----+----------+----------------+------------+
|  2 | madrid   | ciudad_iberica | 1111-01-02 |
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | caesaraugusta  | NULL       |
|  5 | Bilbao   | ciudad_iberica | NULL       |
|  6 | malaga   | ciudad_iberica | NULL       |
+----+----------+----------------+------------+
5 rows in set (0.00 sec)

mysql>DELETE FROM granciudad WHERE id=6;
Query OK, 1 row affected (0.08 sec)

mysql> SELECT * FROM granciudad;
+----+----------+----------------+------------+
| id | nombre   | origen         | fecha      |
+----+----------+----------------+------------+
|  2 | madrid   | ciudad_iberica | 1111-01-02 |
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | caesaraugusta  | NULL       |
|  5 | Bilbao   | ciudad_iberica | NULL       |
+----+----------+----------------+------------+
4 rows in set (0.00 sec)
```

Como podemos observar solo se eliminó la fila cuyo id era 6, es decir, la de 'malaga'

También podemos usar las cláusulas LIMIT y ORDER BY en la sentencia DELETE , por ejemplo, para eliminar los últimos o primeros registros de una tabla.

Por ejemplo , si queremos eliminar las dos primeras ciudades que aparecen en la tabla ordenando alfabéticamente por su nombre realizaríamos lo siguiente:

```
mysql> select * from granciudad;
+----+----------+----------------+------------+
| id | nombre   | origen         | fecha      |
+----+----------+----------------+------------+
|  2 | madrid   | ciudad_iberica | 1111-01-02 |
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | caesaraugusta  | NULL       |
|  5 | Bilbao   | ciudad_iberica | NULL       |
+----+----------+----------------+------------+
4 rows in set (0.00 sec)

mysql>DELETE FROM granciudad ORDER BY nombre ASC LIMIT 2;
Query OK, 2 rows affected (0.27 sec)

mysql> SELECT * from granciudad;
+----+----------+----------------+------------+
| id | nombre   | origen         | fecha      |
+----+----------+----------------+------------+
|  3 | valencia | ciudad_iberica | 3214-01-03 |
|  4 | zaragoza | caesaraugusta  | NULL       |
+----+----------+----------------+------------+
2 rows in set (0.00 sec)
mysql>
```

Como podemos observar ha eliminado ‘madrid’ y ‘bilbao’ que son las dos primeras ciudades si ordenamos la lista alfabéticamente por el campo ‘nombre’.
Si en LIMIT hubiésemos puesto 1 solo habría eliminado Bilbao. Si hubiésemos puesto 3 habría eliminado ‘bilbao’ ‘madrid’ y ‘valencia’.

Vaciar una tabla

TRUNCATE …

Cuando queremos eliminar todas la filas de una tabla, vimos en el punto anterior que podíamos usar una sentencia DELETE sin condiciones. Sin embargo, existe una sentencia alternativa, **TRUNCATE,** que realiza la misma tarea de una forma mucho más rápida.

La diferencia es que DELETE hace un borrado secuencial de la tabla, fila a fila. Pero TRUCATE borra la tabla y la vuelve a crear vacía, lo que es mucho más eficiente.

```
mysql>TRUNCATE granciudad;
Query OK, 0 rows affected (0.44 sec)

mysql> SELECT * FROM granciudad;
Empty set (0.00 sec)
```

Como podemos observar hemos vaciado la tabla 'granciudad' de registros.

Consultar una tabla

SELECT * FROM …
SELECT … FROM …
SELECT … AS … FROM…
SELECT DISTINCT … FROM …
SELECT … FROM … WHERE…
SELECT … FROM … WHERE … AND …
SELECT … FROM … LIMIT … , … ;

Si queremos mostrar todos los registros de una tabla usaremos **SELECT * FROM** y el nombre de la tabla. Por ejemplo:

```
mysql>SELECT * from granciudad;
+----+-----------+-----------+
| id | nombre    | poblacion |
+----+-----------+-----------+
|  1 | madrid    |   4000000 |
| 30 | barcelona |   2000000 |
| 31 | valencia  |   1500000 |
+----+-----------+-----------+
3 rows in set (0.00 sec)
```

En esta instrucción la expresión * (asterisco) significa “todas las columnas”.

Podemos realizar una consulta donde se nos muestre solo las columnas que nos interesa. Para ello usaremos SELECT más las columnas que queremos separadas por coma:

```
mysql>SELECT id, nombre FROM granciudad;
+----+-----------+
| id | nombre    |
+----+-----------+
|  1 | madrid    |
| 30 | barcelona |
| 31 | valencia  |
+----+-----------+
3 rows in set (0.00 sec)
```

También podemos operar con los valores de una columna, por ejemplo si queremos que nos devuelvan la población en miles pondríamos lo siguiente:

```
mysql>SELECT nombre, poblacion/1000000 FROM granciudad;
+-----------+-------------------+
| nombre    | poblacion/1000000 |
+-----------+-------------------+
| madrid    |            4.0000 |
| barcelona |            2.0000 |
| valencia  |            1.5000 |
+-----------+-------------------+
3 rows in set (0.00 sec)
```

Recuerda que la información en la tabla no se modifica. Lo único que pedimos es mostrar un valor calculado a partir de la información de la tabla.

También podríamos mostrar la misma información pero cambiando el encabezado de la columna afectada con el comando **AS**:

```
SELECT nombre, poblacion/1000000 AS poblacion_en_millones FROM
granciudad;
+-----------+----------------------+
| nombre    | poblacion_en_millones |
+-----------+----------------------+
| madrid    |               4.0000 |
| barcelona |               2.0000 |
| valencia  |               1.5000 |
+-----------+----------------------+
3 rows in set (0.02 sec)
```

Puedes conocer todos las funciones numéricas en MySQL con el comando:
mysql>**HELP NUMERIC FUNCTIONS**

O de cualquier tipo de función con:
mysql>**HELP FUNCTIONS**

En una consulta podría haber registros repetidos, si quisiéramos evitar mostrar las filas repetidas usaríamos el comando **DISTINCT**.

Por ejemplo, imaginemos que en la tabla anterior insertamos otro registro que también se llame madrid, tenga 4 millones de habitantes aunque su id sea otro.

Si consultamos la tabla mostrando los campos nombre y población podremos observar 2 filas idénticas:

```
mysql> SELECT nombre, poblacion FROM granciudad;
+-----------+-----------+
| nombre    | poblacion |
+-----------+-----------+
| madrid    |   4000000 |
| barcelona |   2000000 |
| valencia  |   1500000 |
| madrid    |   4000000 |
+-----------+-----------+
4 rows in set (0.00 sec)
```

Sin embargo, si usamos DISTINCT solo se nos mostrarán las líneas no repetidas:

idénticas:

```
mysql>SELECT DISTINCT nombre, poblacion FROM granciudad;
+-----------+-----------+
| nombre    | poblacion |
+-----------+-----------+
| madrid    |   4000000 |
| barcelona |   2000000 |
| valencia  |   1500000 |
+-----------+-----------+
3 rows in set (0.00 sec)
```

Recuerda que aunque se muestren 3 registros, en la tabla hay 4 registros dado que “Madrid 4000000” está repetido.

Si queremos filtrar las filas que aparecen en la consulta podemos usar la sentencia **WHERE**, por ejemplo, partiendo de la tabla siguiente, queremos mostrar solo las ciudades que tienen 2 o más millones de población:

```
mysql>SELECT * from granciudad;
+----+----------+-----------+
| id | nombre   | poblacion |
+----+----------+-----------+
|  1 | madrid   |   4000000 |
| 30 | barcelona |   2000000 |
| 31 | valencia |   1500000 |
|  3 | sevilla  |   1000000 |
+----+----------+-----------+
4 rows in set (0.00 sec)
mysql>SELECT * from granciudad WHERE poblacion >= 2000000;
+----+----------+-----------+
| id | nombre   | poblacion |
+----+----------+-----------+
|  1 | madrid   |   4000000 |
| 30 | barcelona |   2000000 |
+----+----------+-----------+
2 rows in set (0.00 sec)
```

Incluso anidar 2 condiciones usando el comando **AND**:

```
mysql>SELECT * from granciudad WHERE poblacion >= 2000000 AND id > 10;
+----+----------+-----------+
| id | nombre   | poblacion |
+----+----------+-----------+
| 30 | barcelona |   2000000 |
+----+----------+-----------+
1 row in set (0.03 sec)
```

Además de AND, podríamos usar otros operadores boobleanos como **OR, XOR o NOT**

Recuerda que puedes ver e informarte sobre los operadores booleanos disponibles con el comando: **HELP LOGICAL OPERATORS**

En una cláusula WHERE se puede usar cualquier función disponible en MySQL, excluyendo sólo las de resumen o reunión, que veremos en el siguiente punto. Esas funciones están diseñadas específicamente para usarse en cláusulas GROUP BY.

La cláusula LIMIT puede usarse con 2 parámetros. Cuando se usan ambos, el primero muestra la fila a partir de la que debemos empezar a mostrar registros, y el segundo parámetro muestra el número de filas a mostrar (contando desde la primera):

```
mysql> select * from lista;
+------+---------+
| id   | nombre  |
+------+---------+
|    1 | paco    |
|    2 | alfonso |
|    3 | Ana     |
|    4 | yasmina |
|    5 | Babet   |
+------+---------+
5 rows in set (0.00 sec)

mysql> select * from lista limit 1,2;
+------+---------+
| id   | nombre  |
+------+---------+
|    2 | alfonso |
|    3 | Ana     |
+------+---------+
2 rows in set (0.00 sec)

mysql> select * from lista limit 2,2;
+------+---------+
| id   | nombre  |
+------+---------+
|    3 | Ana     |
|    4 | yasmina |
+------+---------+
2 rows in set (0.00 sec)
```

Agrupar Filas

SELECT … FROM … GROUP BY…
SELECT … COUNT(*) FROM … GROUP BY…
SELECT … COUNT(*) AS … FROM … GROUP BY…
SELECT … MAX(…) FROM … GROUP BY … HAVING MAX(…) …

Es posible agrupar filas en la salida de una sentencia SELECT según los distintos valores de una columna, usando la cláusula GROUP BY. Esto, en principio, puede parecer redundante, ya que podíamos hacer lo mismo usando la opción DISTINCT. Sin embargo, la cláusula GROUP BY es más potente.

Por ejemplo, partimos de la siguiente tabla:

```
mysql> CREATE TABLE muestras (
    -> ciudad VARCHAR(40),
    -> fecha DATE,
    -> temperatura TINYINT );
Query OK, 0 rows affected (0.28 sec)

mysql> INSERT INTO muestras (ciudad,fecha,temperatura) VALUES
    ->('Madrid', '2005-03-17', 23),
    -> ('París', '2005-03-17', 16),
    -> ('Berlín', '2005-03-17', 15),
    -> ('Madrid', '2005-03-18', 25),
    -> ('Madrid', '2005-03-19', 24),
    -> ('Berlín', '2005-03-19', 18);
Query OK, 6 rows affected (0.08 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM muestras;
+---------+------------+-------------+
| ciudad  | fecha      | temperatura |
+---------+------------+-------------+
| Madrid  | 2005-03-17 |          23 |
| París   | 2005-03-17 |          16 |
| Berlín  | 2005-03-17 |          15 |
| Madrid  | 2005-03-18 |          25 |
| Madrid  | 2005-03-19 |          24 |
| Berlín  | 2005-03-19 |          18 |
+---------+------------+-------------+
6 rows in set (0.00 sec)
```

Podríamos agrupar el resultado de la consulta con el comando GROUP BY de modo que nos agrupara los registros de la tabla por ciudades:

```
mysql>SELECT ciudad FROM muestras GROUP BY ciudad;
+---------+
| ciudad  |
+---------+
| Berlín  |
| Madrid  |
| París   |
+---------+
3 rows in set (0.00 sec)
```

Observamos que con GROUP BY la salida se ordena según los valores de la columna indicada. En este caso se ordenan alfabéticamente por el atributo ”nombre”.

Pero la diferencia principal es que el uso de la cláusula GROUP BY permite usar funciones de resumen o reunión. Por ejemplo, la función **COUNT(),**que sirve para contar las filas de cada grupo:

```
mysql>SELECT ciudad, COUNT(*) AS cuenta FROM muestras GROUP BY ciudad;
+---------+--------+
| ciudad  | cuenta |
+---------+--------+
| Berlín  |      2 |
| Madrid  |      3 |
| París   |      1 |
+---------+--------+
3 rows in set (0.05 sec)
```

El asterisco de COUNT(*) significa TODAS las filas. Si indicase el nombre de un atributo contaría el número de registros que no son NULL en ese mismo atributo.

Además de COUNT disponemos de otros operadores como: AVG(), COUNT_DISTINCT(), MAX(), MIN(), STD(), SUM(), VARIANCE(), ...

Al final de este documento hay un anexo con la explicación de los distintos comandos asociados a GROUP BY.

En cualquier caso, para conocer e informarte de todos los operadores de GROUP BY puedes consultar aquí:
mysql>**HELP Functions and Modifiers for Use with GROUP BY**

La cláusula **HAVING()** permite hacer selecciones con GROUP BY en columnas calculadas con funciones de grupo, como MAX(), MIN(),AVG(),COUNT()... Se podría entender como un WHERE para usar junto a GROUP BY()
Veamos un ejemplo:

```
mysql> CREATE TABLE muestras (
    -> ciudad VARCHAR(40),
    -> fecha DATE,
    -> temperatura TINYINT);
Query OK, 0 rows affected (0.25 sec)

mysql> mysql> INSERT INTO muestras (ciudad,fecha,temperatura) VALUES
    -> ('Madrid', '2005-03-17', 23),
    -> ('París', '2005-03-17', 16),
    -> ('Berlín', '2005-03-17', 15),
    -> ('Madrid', '2005-03-18', 25),
    -> ('Madrid', '2005-03-19', 24),
    -> ('Berlín', '2005-03-19', 18);
Query OK, 6 rows affected (0.03 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> SELECT ciudad, MAX(temperatura) FROM muestras
    -> GROUP BY ciudad HAVING MAX(temperatura)>16;
+--------+------------------+
| ciudad | MAX(temperatura) |
+--------+------------------+
| Berlín |               18 |
| Madrid |               25 |
+--------+------------------+
2 rows in set (0.00 sec)

mysql>
```

Claves ajenas

CREATE TABLE … (…
FOREIGN KEY … REFERENCES …(…)
ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
) ENGINE=InnoDB;

En MySQL solo existe soporte para claves ajenas en tablas de tipo InnoDB. Es decir, si no creamos la tabla como "InnoDB", las claves ajenas solo figurarán como documentación pero no se aplicarán.
Las claves ajenas en tablas tipo "InnoDB" se define la clave ajena después de definir todas las columnas.

Un ejemplo usando tablas InnoDB sería el siguiente:

```
mysql> CREATE TABLE personas (
    -> id INT AUTO_INCREMENT PRIMARY KEY,
    -> nombre VARCHAR(40),
    -> fecha DATE)
    -> ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE telefonos (
    -> numero CHAR(12),
    -> id INT NOT NULL,
    -> KEY (id),
    -> FOREIGN KEY (id) REFERENCES personas (id)
    -> ON DELETE CASCADE ON UPDATE CASCADE)
    -> ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql>
```

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

ON DELETE <opción>, indica que acciones se deben realizar en la tabla actual si se borra una fila en la tabla referenciada.

ON UPDATE <opción>, es análogo pero para modificaciones de claves.

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

- **RESTRICT:** esta opción impide eliminar o modificar filas en la tabla referenciada si existen filas con el mismo valor de clave ajena.

- **CASCADE:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica borrar las filas con el mismo valor de clave ajena o modificar los valores de esas claves ajenas.

- **SET NULL:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado de clave, implica asignar el valor *NULL* a las claves ajenas con el mismo valor.

- **NO ACTION:** las claves ajenas no se modifican, ni se eliminan filas en la tabla que las contiene.

- **SET DEFAULT:** borrar o modificar una clave en una fila en la tabla referenciada con un valor determinado implica asignar el valor por defecto a las claves ajenas con el mismo valor.

Operadores

MySQL dispone de varios tipos de operadores:

- operadores de asignación
- operadores lógicos
- operadores de comparaciones
- operadores aritméticos
-operadores de flujo de control
- operadores para cadenas

Los operadores se suelen usar en expresiones con cláusulas como ORDER BY / HAVING, en cláusulas WHERE y en sentencias SET.

Operadores de asignación

MySQL dispone de operadores para crear variables.

Existen 2 formas de crear variables, con la sentencia SET o dentro de la sentencia SELECT.

Para ello usamos el carácter @. A continuación mostramos un ejemplo:

```
mysql> SET @ahora= CURRENT_TIMESTAMP();
Query OK, 0 rows affected (0.00 sec)
 mysql> SELECT @ahora;
+---------------------+
| @ahora              |
+---------------------+
| 2018-02-12 14:25:37 |
+---------------------+
1 row in set (0.00 sec)
```

La otra posibilidad es declarar variables en una sentencia SELECT usando "**:=**"

Ejemplo:

```
mysql> SELECT @variable:="Hola mundo";
+-------------------------+
| @variable:="Hola mundo" |
+-------------------------+
| Hola mundo              |
+-------------------------+
1 row in set (0.00 sec)
mysql> SELECT @variable_numerica:=155;
+--------------------------+
| @variable_numerica:=155 |
+--------------------------+
|                      155 |
+--------------------------+
1 row in set (0.00 sec)
```

Operadores lógicos

Los operadores lógicos se usan para realizar operaciones de álgebra booleana. A pesar que en el álgebra solo existen 2 tipos de variables booleanas, verdadero y falso (TRUE y FALSE), en MySQL disponemos de un tercer valor que consideraremos desconocido, NULL,

La sintaxis para el valor verdadero es **1** o TRUE
La sintaxis para el valor falso es **0** o FALSE
La sintaxis para el valor desconocido es NULL

```
Mysql> SELECT TRUE;
+------+
| TRUE |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

O de modo más complejo aún:

```
mysql> SELECT TRUE, 1, FALSE, 0, NULL;
+------+---+-------+---+------+
| TRUE | 1 | FALSE | 0 | NULL |
+------+---+-------+---+------+
|    1 | 1 |     0 | 0 | NULL |
+------+---+-------+---+------+
1 row in set (0.00 sec)
```

A continuación mostramos la sintaxis de los típicos operadores booleanos:

La sintaxis para el operador AND es	&&	o	AND
La sintaxis para el operador OR es	OR	o	||
La sintaxis para el operador OR exclusivo es	XOR		
La sintaxis para el operador de Negación es	NOT	o	!

Cuando concatenamos operadores lógicos hemos de tener presente que se ejecutan de 2 en 2, empezando por la izquierda (salvo indicación expresa, por ejemplo con paréntesis). Ejemplo:

```
mysql> SELECT true XOR true OR TRUE and TRUE;
+--------------------------------+
| true XOR true OR TRUE and TRUE |
+--------------------------------+
|                              1 |
+--------------------------------+
1 row in set (0.00 sec)

mysql> SELECT true XOR (true OR TRUE and TRUE);
+---------------------------------+
| true XOR (true OR TRUE and TRUE) |
+---------------------------------+
|                                0 |
+---------------------------------+
1 row in set (0.00 sec)
```

Fíjate que los resultados de ambas instrucciones son diferentes... ¿Por qué crees que es así?

La introducción en MySQL del valor NULL como valor booleano provoca que se alteren las tablas de la verdad tradicionales del álgebra de boole de la siguiente manera:

A	B	A AND B
falso	falso	falso
falso	verdadero	falso
verdadero	falso	falso
verdadero	verdadero	verdadero
falso	NULL	falso
NULL	falso	falso
verdadero	NULL	NULL
NULL	verdadero	NULL

A	B	A OR B
falso	falso	falso
falso	verdadero	verdadero
verdadero	falso	verdadero
verdadero	verdadero	verdadero
falso	NULL	NULL
NULL	falso	NULL
verdadero	NULL	verdadero
NULL	verdadero	verdader

A	B	A XOR B
falso	falso	falso
falso	verdadero	verdadero
verdadero	falso	verdadero
verdadero	verdadero	falso
falso	NULL	NULL
NULL	falso	NULL
verdadero	NULL	NULL
NULL	verdadero	NULL

A	NOT A
falso	verdadero
verdadero	falso
NULL	NULL

Operadores de comparación

En MySQL disponemos de diversos operadores de comparación. Estos nos permiten crear expresiones lógicas que aplican el álgebra de Boole. Podemos comparar fechas, cadenas, números, etcétera. Se devuelven valores lógicos como Verdadero (1) o Falso (0).

Entre los operadores de comparación destacamos los siguientes:

Operadores	Signos
Igualdad	=
Igualdad sin devolver NULL	<=>
Operador de desigualdad	<>
Operadores de comparación de magnitudes	< <= > >=
chequear valores NULL	**IS NULL** **IS NOT NULL**
Pertenencia a un rango	**BETWEEN… AND…**
Selección de valores	**COALESCE(…)**
Máximo y mínimo en una lista	**GREATEST(…)** **LEAST(…)**
Dentro de una lista	**IN(…)** **NOT IN(…)**
Intervalo	**INTERVAL(…)**

Debemos tener en cuenta las siguientes reglas para comparar valores:

1. El valor NULL funciona diferente con los dos operadores de igualdad

2. Si ambos valores son cadenas, la comparación será como cadenas (y no enteros)

3. Si ambos valores son enteros, la comparación será como enteros.

4. Cuando uno de los valores es una fecha de tipo TIMESTAMP o DATETIME y el otro NO, el sistema convertirá la cadena que NO es tipo fecha a tipo fecha antes de realizar la compmaración (si es posible la conversión). La excepción a esto sucede cuando se usa IN().

A continuación vamos a mostrar un ejemplo de uso para cada uno:

Igualdad

Con el operador = podemos comparar dos valores.

```
mysql> SELECT * FROM lista;

+---------+------------+
| Código  | fecha      |
+---------+------------+
|       1 | 2010-03-03 |
|       2 | 2010-03-06 |
|       3 | 2010-03-03 |
|       4 | 2010-03-25 |
|       5 | 2010-03-03 |
+---------+------------+
5 rows in set (0.00 sec)

mysql> SELECT * FROM lista WHERE fecha="2010-03-03";
+---------+------------+
| Código  | fecha      |
+---------+------------+
|       1 | 2010-03-03 |
|       3 | 2010-03-03 |
|       5 | 2010-03-03 |
+---------+------------+
3 rows in set (0.00 sec)

mysql> SELECT 4=3;
+-----+
| 4=3 |
+-----+
|   0 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT 4=4;
+-----+
| 4=4 |
+-----+
|   1 |
+-----+
1 row in set (0.00 sec)
```

Igualdad sin devolver NULL

Con el operador **<=>** podemos comparar dos valores entendiendo NULL como un valor más a comparar. Es decir, entendiendo NULL como algo diferente a cualquier cadena o entero.

```
mysql> SELECT NULL <=> 19;
+-------------+
| NULL <=> 19 |
+-------------+
|           0 |
+-------------+
1 row in set (0.02 sec)

mysql> SELECT NULL = 19;
+-----------+
| NULL = 19 |
+-----------+
|      NULL |
+-----------+
1 row in set (0.00 sec)
```

Fíjense que en el segundo caso hemos comparado con = y no con <=>.

Operador de desigualdad

Se dispone de 2 operadores de desigualdad que son equivalentes: **<>** y **!=** .
Se nos devolverá verdadero o falso en función de si ambos valores son o no desiguales.

```
mysql> SELECT 30 <> 29;
+----------+
| 30 <> 29 |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT 30 != 30;
+----------+
| 30 != 30 |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)
```

Operadores de compaparación de magnitudes

Existen 4 comparadores de magnitud:

menor que	<
mayor que	>
menor o igual que	<=
mayor o igual que	>=

```
mysql> SELECT "a" < "z";
+-----------+
| "a" < "z" |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT 30 >= 29;
+----------+
| 30 >= 29 |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

Se pueden comparar números, cadenas (alfabéticamente) e incluso fechas.

Chequear valores NULL

Para verificar el valor NULL usamos los operadores **IS NULL** o **IS NOT NULL**

```
mysql> select * from lista;
+---------+------------+
| Código  | fecha      |
+---------+------------+
|       1 | 2010-03-03 |
|       2 | 2010-03-06 |
|       3 | NULL       |
|       4 | 2010-03-25 |
|       5 | 2010-03-03 |
+---------+------------+
5 rows in set (0.00 sec)
+----------+

mysql> SELECT * FROM lista WHERE fecha IS NULL;
+---------+-------+
| Código  | fecha |
+---------+-------+
|       3 | NULL  |
+---------+-------+
1 row in set (0.00 sec)
```

Pertenencia a un rango

Para verificar si un valor esta dentro de un rango usamos el comando **BETWEEN**

Para ello debemos marcar el mínimo y máximo de la expresión

```
mysql> SELECT 15 BETWEEN 10 AND 16;
+----------------------+
| 15 BETWEEN 10 AND 16 |
+----------------------+
|                    1 |
+----------------------+
1 row in set (0.00 sec)

mysql> SELECT "2018-01-01" BETWEEN "2017-01-01" AND
"2019-01-01";
+-----------------------------------------------------+
| "2018-01-01" BETWEEN "2017-01-01" AND "2019-01-01" |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+
1 row in set (0.01 sec)
```

Selección de valores

Con el operador **COALESCE** elegimos el primer valor NO NULO de una lista.

Ejemplo de uso:

```
mysql> SELECT COALESCE (NULL, 23, "hola");
+-----------------------------+
| COALESCE (NULL, 23, "hola") |
+-----------------------------+
| 23                          |
+-----------------------------+
1 row in set (0.03 sec)
```

Máximo y mínimo en una lista

Con los operadores **GREATEST** Y **LEAST** elegimos el mayor y menor valor de una lista de valores.

Ejemplo de uso:

```
mysql> SELECT GREATEST(1,2,5);
+----------------+
| GREATEST(1,2,5) |
+----------------+
|               5 |
+----------------+
1 row in set (0.00 sec)

mysql> SELECT LEAST("a", "b", "g");
+---------------------+
| LEAST("a", "b", "g") |
+---------------------+
| a                   |
+---------------------+
1 row in set (0.00 sec)
```

Dentro de una lista

Con los operadores **IN** y **NOT IN** podemos saber si un elemento está o no dentro de una lista. Se devuelve un valor booleano (verdadero/falso, 1/0).

```
mysql> SELECT "g" IN ("u", "o", "p");
+------------------------+
| "g" IN ("u", "o", "p") |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.01 sec)

mysql> select fecha from lista;
+------------+
| fecha      |
+------------+
| 2010-03-03 |
| 2010-03-06 |
| NULL       |
| 2010-03-25 |
| 2010-03-03 |
+------------+
5 rows in set (0.00 sec)

mysql> SELECT "2010-03-03" IN (select fecha from lista);
+-------------------------------------------+
| "2010-03-03" IN (select fecha from lista) |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
1 row in set (0.03 sec)

mysql> SELECT 5 NOT IN (1, 2, 3, 4);
+-----------------------+
| 5 NOT IN (1, 2, 3, 4) |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.00 sec)
```

No hay que malinterpretar el resultado que se devuelve, el cual es booleano, es decir es Verdadero o Falso (1 o 0). En la segunda sentencia se muestra que la fecha “2010-03-03” sí está en la lista, pero no se dice el número de veces que está en la lista, que en todo caso sería 2, y no 1.

Intervalo

Con el operador **INTERVAL** podemos hallar en que intervalo está un valor. Para ello debemos escribir tanto el valor como los límites de cada intervalo,.

La sintaxis es:

INTERVAL (Valor_a_medir, limite1, limite2, limite3, ...)

Valor_a_medir: es el valor cuyo intervalo pretendemos saber

limite *n* : es cada uno de los límites a partir del cual comienza un nuevo intervalo. Deben estar ordenados.

Ejemplo de uso;

```
mysql> SELECT INTERVAL (55, 30, 40, 50, 60, 70);
+----------------------------------+
| INTERVAL (55, 30, 40, 50, 60, 70) |
+----------------------------------+
|                                3 |
+----------------------------------+
1 row in set (0.00 sec)
```

En este caso, el 3 devuelto significa que el 55 está entre el intervalo del 3er y 4º límite.

En caso que el valor esté por debajo de el primer límite se devolverá cero 0.

Operadores aritméticos

MySQL dispone de operadores aritméticos para poder operar entre valores numéricos. El resultado siempre es un valor numérico.

En caso que los operandos sean enteros, el resultado se devuelve con el tipo BIGINT, que es un tipo de dato numérico de 64 bits. Es importante tener esto en cuenta para números especialmente grandes.

Los operadores aritméticos más convencionales en MySQL son:

Operador	Signo
Suma	+
Resta	-
Operador negativo	-
Operador multiplicador	*
Operador de división	/
Operador división entera	DIV
Operador resto	%

```
mysql> set @a=10, @b=2;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @a-@b, -@a, @a*@b, @a/@b, @a%@b;
+-------+------+-------+--------+-------+
| @a-@b | -@a  | @a*@b | @a/@b  | @a%@b |
+-------+------+-------+--------+-------+
|     8 |  -10 |    20 | 5.0000 |     0 |
+-------+------+-------+--------+-------+
1 row in set (0.00 sec)
```

Para consultar todos los operadores aritméticos existentes teclee lo siguiente en la terminal de MySQL:

Help Numeric Functions

Precedencia

A continuación se muestra la precedencia de los distintos operadores, ordenados de menor a mayor.

Precedencia de operadores
:=
||, OR, XOR
&&, AND
NOT
BETWEEN, CASE, WHEN, THEN, ELSE
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
|
&
<<, >>
-, +
*, /, DIV, %, MOD
^
- (unitario), ~ (complemento)
!
BINARY, COLLATE
Paréntesesis → (...)

Consultas multitabla

Hasta el momento hemos visto como hacer consultas en una sola tabla. También es posible realizar consultas en varias tablas. Para ello SQL dispone de diversas formas de hacer consultas multitabla que explicamos a continuación.

- Composiciones internas

- Composiciones externas

Las **Composiciones internas** son aquellas que parten de un producto cartesiano entre dos tablas, es decir, la combinación de todas las filas de una tabla con las filas de la otra tabla. A partir de ese producto cartesiano de tablas, se pueden hacer filtros o restricciones al resultado.

Las **Composiciones externas** no se originan de un producto cartesiano sino que se toma un registro de una tabla y se busca registros de otra tabla que coincidan en el atributo indicado.

INFORMACIÓN ADICIONAL - Producto Cartesiano

En matematicas, un producto cartesiano sobre 2 conjuntos es el conjunto resultante de la combinación de todos los elementos del primer conjunto emparejados con cada uno de los elementos del segundo conjunto.

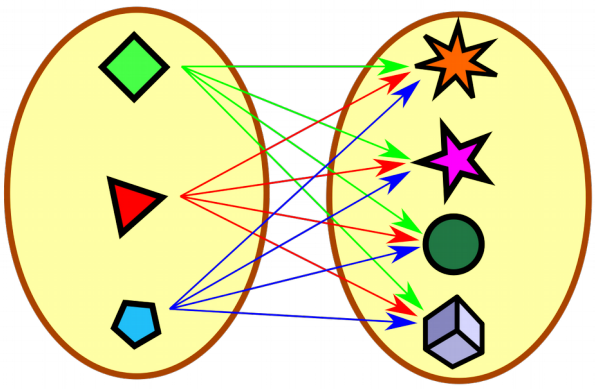

Imagen de un gráfico con la combinación de dos conjuntos.

3	(3,5)	(3,6)	(3,7)
4	(4,5)	(4,6)	(4,7)
AxB	5	6	7

Producto cartesiano de AxB

A =(3,4)

B (5,6,7)

AxB = (3,5) , (3,6) , (3,7) , (4,5) , (4,6) , (4,7)

Composiciones internas

SELECT … FROM *tabla1* JOIN *tabla2* ;
SELECT … FROM *tabla1* JOIN *tabla2* ON (*tabla1.atributo* = *tabla2.atributo*) ;
SELECT … FROM *tabla1* , *tabla2* ON (*tabla1.atributo* = *tabla2.atributo*) ;
SELECT … FROM *tabla1* NATURAL JOIN *tabla2* ;

Para realizar el producto cartesiano de 2 tablas usamos el comando **JOIN** que puede ser sustituido en MySQL por la coma(,)

Por ejemplo, partimos de estas 2 tablas:

```
mysql> select * from jugador;
+----+-----------+
| id | nombre    |
+----+-----------+
|  1 | Ronaldo   |
|  2 | Messi     |
|  3 | Guedes    |
|  4 | Griezmann |
|  5 | Suarez    |
+----+-----------+
5 rows in set (0.00 sec)

mysql> select * from pichichi;
+----+-------+
| id | goles |
+----+-------+
|  1 |    35 |
|  3 |    20 |
|  5 |    15 |
+----+-------+
3 rows in set (0.00 sec)
1 row in set (0.00 sec)
```

Para hacer el producto cartesiano debemos realizar un **JOIN** entre ambas tablas:

```
mysql> SELECT * FROM jugador JOIN pichichi;
+----+-----------+----+-------+
| id | nombre    | id | goles |
+----+-----------+----+-------+
|  1 | Ronaldo   |  1 |    35 |
|  1 | Ronaldo   |  3 |    20 |
|  1 | Ronaldo   |  5 |    15 |
|  2 | Messi     |  1 |    35 |
|  2 | Messi     |  3 |    20 |
|  2 | Messi     |  5 |    15 |
|  3 | Guedes    |  1 |    35 |
|  3 | Guedes    |  3 |    20 |
|  3 | Guedes    |  5 |    15 |
|  4 | Griezmann |  1 |    35 |
|  4 | Griezmann |  3 |    20 |
|  4 | Griezmann |  5 |    15 |
|  5 | Suarez    |  1 |    35 |
|  5 | Suarez    |  3 |    20 |
|  5 | Suarez    |  5 |    15 |
+----+-----------+----+-------+
15 rows in set (0.00 sec)
```

Como podemos observar, aparecen los 2 atributos de la tabla jugador y los 2 atributos de la tabla goleadores. Cada registro de la tabla jugador se ha combinado con cada registro de la tabla pichichi.

Imaginemos que solo deseamos mostrar las filas donde el “id” de jugador coincida con el “id” de goles, para ello deberíamos usar la restricción **ON**, la cual funciona muy similar a la restricción WHERE ya aprendida.

```
mysql> SELECT * FROM jugador JOIN pichichi ON (jugador.id = pichichi.id);
+----+---------+----+-------+
| id | nombre  | id | goles |
+----+---------+----+-------+
|  1 | Ronaldo |  1 |    35 |
|  3 | Guedes  |  3 |    20 |
|  5 | Suarez  |  5 |    15 |
+----+---------+----+-------+
3 rows in set (0.01 sec)
```

Como se puede observar, el resultado es una restricción del anterior resultado pues solo se muestran aquellas filas cuyos “id” coinciden.

Además, MySQL permite simplificar la expresión anterior usando **NATURAL JOIN** el cual automáticamente busca los atributos de ambas tablas coincidentes.

Por ejemplo, la anterior consulta podíamos haberla hecho de la siguiente manera:

```
mysql> SELECT * FROM jugador NATURAL JOIN pichichi;
+----+---------+-------+
| id | nombre  | goles |
+----+---------+-------+
|  1 | Ronaldo |    35 |
|  3 | Guedes  |    20 |
|  5 | Suarez  |    15 |
+----+---------+-------+
3 rows in set (0.02 sec)
```

Como se puede observar, NATURAL JOIN ha tomado el atributo “id” de ambas tablas para emparejar las filas resultantes.

Hay que fijarse en que ahora solo aparece un campo “id” que unifica a los 2 “id” que hay en las tablas jugador y pichichi.

Para poder realizar NATURAL JOIN, los campos a unificar han de llamarse igual.

Composiciones externas

SELECT … FROM *tabla1* LEFT JOIN *tabla2* ON (*tabla1.atributo* = *tabla2.atributo*) ;
SELECT … FROM *tabla1* RIGHT JOIN *tabla2* ON (*tabla1.atributo* = *tabla2.atributo*) ;
SELECT … FROM *tabla1* NATURAL LEFT JOIN *tabla2* ;
SELECT … FROM *tabla1* NATURAL RIGHT JOIN *tabla2* ;

Para realizar las composiciones externas no partimos del producto cartesiano de las tablas, sino que por cada una de las filas de la primera tabla se buscará una fila en la segunda tabla cuyo atributo en común coincida.

Por ejemplo, imaginemos que al anterior ejemplo añadimos 2 entradas en la tabla pichichi con “id” que no aparecen en la tabla jugador.

```
mysql> insert into pichichi (id, goles) VALUES (11, 6);
Query OK, 1 row affected (0.12 sec)

mysql> insert into pichichi (id, goles) VALUES (12, 3);
Query OK, 1 row affected (0.04 sec)

mysql> select * from pichichi;
+----+-------+
| id | goles |
+----+-------+
|  1 |    35 |
|  3 |    20 |
|  5 |    15 |
| 11 |     6 |
| 12 |     3 |
+----+-------+
5 rows in set (0.00 sec)

mysql> select * from jugador;
+----+-----------+
| id | nombre    |
+----+-----------+
|  1 | Ronaldo   |
|  2 | Messi     |
|  3 | Guedes    |
|  4 | Griezmann |
|  5 | Suarez    |
+----+-----------+
5 rows in set (0.00 sec)
```

Como se puede observar, en la tabla “jugador” hay algunos“id” que no aparecen en la tabla “pichichi” y viceversa.

LEFT JOIN hace referencia a que la consulta se hará por cada una de los registros de la primera tabla. Si un registro de la primera tabla no coincide con el campo común de cualquier registro de la segunda tabla, se usará NULL para los campos que no puedan ser rellenados.

Por ejemplo, si se desea mostrar todos los registros de jugadores de la tabla "jugador" junto con sus goles correspondientes de la tabla "pichichi", y que INCLUSO los jugadores de la tabla "jugador" que no tienen goles en la tabla "pichichi" también aparezcan, usaríamos un **LEFT JOIN**.

```
mysql> SELECT * FROM jugador LEFT JOIN pichichi ON
(jugador.id=pichichi.id);
+----+-----------+------+-------+
| id | nombre    | id   | goles |
+----+-----------+------+-------+
|  1 | Ronaldo   |    1 |    35 |
|  2 | Messi     | NULL |  NULL |
|  3 | Guedes    |    3 |    20 |
|  4 | Griezmann | NULL |  NULL |
|  5 | Suarez    |    5 |    15 |
+----+-----------+------+-------+
5 rows in set (0.00 sec)
```

Como se puede observar, los jugadores con id=2 y id=4 no aparecen en la tabla pichichi, por tanto los campos pertenecientes a las columnas de la tabla pichichi se han rellenado con NULL.

Lo mismo ocurre con **RIGHT JOIN**, solo que en vez de crear el resultado recorriendo cada uno de los registros de la primera tabla, se usará con los registros de la segunda.

Siguiendo el ejemplo, al usar RIGHT JOIN, por cada registro de la tabla "pichichi" se buscarán registros en la tabla "jugador" que coincidan en el campo común (el "id").

```
mysql> SELECT * FROM jugador RIGHT JOIN pichichi ON
(jugador.id=pichichi.id);
+------+---------+----+-------+
| id   | nombre  | id | goles |
+------+---------+----+-------+
|    1 | Ronaldo |  1 |    35 |
|    3 | Guedes  |  3 |    20 |
|    5 | Suarez  |  5 |    15 |
| NULL | NULL    | 11 |     6 |
| NULL | NULL    | 12 |     3 |
+------+---------+----+-------+
5 rows in set (0.00 sec)
```

Al igual que en las composiciones internas, con NATURAL JOIN nos podemos ahorrar la necesidad de indicarle a MySQL qué campo en común tienen ambas tablas.

Por tanto, en el ejemplo, la última sentencia de MySQL usada podría substituirse por la siguiente:

```
mysql> SELECT * FROM jugador NATURAL RIGHT JOIN pichichi;
+----+-------+---------+
| id | goles | nombre  |
+----+-------+---------+
|  1 |    35 | Ronaldo |
|  3 |    20 | Guedes  |
|  5 |    15 | Suarez  |
| 11 |     6 | NULL    |
| 12 |     3 | NULL    |
+----+-------+---------+
5 rows in set (0.00 sec)
```

Como se puede observar, no es necesario especificar con ON cual es el campo en común, y además en la tabla resultante de la consulta se unifica el campo “id” de ambas tablas.

Lo mismo aplica para NATURAL LEFT JOIN, solo que en vez de la segunda tabla(RIGHT) se formará la consulta a partir de la primera(LEFT)

```
mysql> SELECT * FROM jugador NATURAL LEFT JOIN pichichi;
+----+-----------+-------+
| id | nombre    | goles |
+----+-----------+-------+
|  1 | Ronaldo   |    35 |
|  2 | Messi     |  NULL |
|  3 | Guedes    |    20 |
|  4 | Griezmann |  NULL |
|  5 | Suarez    |    15 |
+----+-----------+-------+
5 rows in set (0.00 sec)
```

Subconsultas

SELECT … FROM … WHERE … [NOT] IN (SELECT … FROM … WHERE …) ;
SELECT … FROM … WHERE … >= ANY (SELECT … FROM … WHERE …) ;
SELECT … FROM … WHERE … >= ALL (SELECT … FROM … WHERE …) ;
SELECT … FROM … WHERE … >= (SELECT … FROM … WHERE …) ;

Una subconsulta es una consulta dentro de otra consulta (a esta segunda la llamamos la principal). Se suelen colocar en la cláusula WHERE de la consulta principal, pero también pueden añadirse en el SELECT o en el FROM.

Para una subconsulta situada en la cláusula WHERE puede usarse los operadores de comparación (> , >=, <, <=, !=, =). Cuando esto ocurre se realiza una comparación entre el valor indicado de la consulta principal y el valor resultante de la subconsulta. Y como es normal en las comparaciones, ambos tipos de datos deben ser iguales.

En el modo sencillo, cuando se usa operadores de comparación, la consulta interna (osea la subconsulta) devuelve 1 solo registro y 1 sola columna.

Si el resultado de la subconsulta devuelve más de un registro, se usan comandos especiales que se escriben entre el operador de comparación y la subconsulta.

ANY : compara cada registro de la consulta principal con los de la subconsulta. Si la comparación entre un registro de la consulta principal con otro de la subconsulta se valida, el registro de la principal pasará el filtro.

ALL : en este caso un registro debe validarse para todos los registros de la subconsulta.

IN : En este caso no se usa el comparador. Se comprueba si cada registro de la consulta principal está o no contenido en la tabla resultante de la subconsulta.

NOT IN : lo opuesto al IN.

Imaginemos q tenemos las siguientes tablas:

mysql> select * from piezas;

```
+--------+----------+
| codigo | nombre   |
+--------+----------+
|      1 | poliet23 |
|      2 | alum35   |
|      3 | etile53  |
+--------+----------+
3 rows in set (0.00 sec)
```

mysql> select * from proveedores;

```
+------+----------+
| id   | nombre   |
+------+----------+
| 1001 | sipem    |
| 1002 | alcupla  |
| 1003 | dragados |
| 1004 | femeval  |
+------+----------+
4 rows in set (0.00 sec)

mysql>
```

mysql> select * from suministra;

```
+-------------+-------------+--------+
| codigoPieza | idProveedor | precio |
+-------------+-------------+--------+
|           1 |        1001 |     28 |
|           1 |        1002 |     30 |
|           1 |        1003 |     23 |
|           2 |        1001 |     20 |
|           2 |        1002 |     21 |
|           3 |        1002 |     44 |
|           3 |        1004 |     34 |
+-------------+-------------+--------+
7 rows in set (0.00 sec)
```

¿Cómo realizaríamos una consulta que nos devolviese el nombre de los proveedores que suministran la pieza 2.

Podemos hacerlo de 2 formas, con una subconsulta o con una consulta multitabla.

Para hacerlo como una consulta multitabla bastaría con hacer un JOIN ... ON... tal y como se explica en el apartado de consultas multitabla.

Para hacerlo como una subconsulta realizariamos una consulta que muestre los nombres de los proveedores que suministran una pieza determinada:

```
mysql> SELECT nombre FROM proveedores WHERE id IN (SELECT
idProveedor from suministra WHERE codigoPieza = 1);
+----------+
| nombre   |
+----------+
| sipem    |
| alcupla  |
| dragados |
+----------+
3 rows in set (0.01 sec)
```

Como se puede observar, en este caso hemos realizado la subconsulta usado el IN, para ver que proveedores estaban en la lista devuelta por la subconsulta.

En otras subconsultas podría usarse los operadores de comparación.

ANEXO

Por medio del comando “help” de la terminal de MySQL se pueden consultar toda la información sobre la sintaxis y uso de los comandos MySQL, tales como tipos de datos, funciones, operadores, seleccionadores, etcétera

Para ver todos los comandos del 'help' de su terminal escriba:

“HELP contents”

No obstante en este anexo exponemos algunos de ellos.

Tipos de dato en una base de datos MySQL

Al crear una tabla la elección correcta de un formato de dato para cada columna de la tabla hará que nuestra BBDD tenga un rendimiento óptimo a medio largo plazo.
Podemos dividir en 3 grandes grupos estos datos:

- Numéricos
- Fecha
- String

Tipos de dato numéricos

Tipos de dato numéricos en MySQL, su ocupación en disco y valores:

INT (INTEGER): Ocupación de 4 bytes con valores entre -2147483648 y 2147483647 o entre 0 y 4294967295.

- **SMALLINT**: Ocupación de 2 bytes con valores entre -32768 y 32767 o entre 0 y 65535.
- **TINYINT**: Ocupación de 1 bytes con valores entre -128 y 127 o entre 0 y 255.
- **MEDIUMINT**: Ocupación de 3 bytes con valores entre -8388608 y 8388607 o entre 0 y 16777215.
- **BIGINT**: Ocupación de 8 bytes con valores entre -8388608 y 8388607 o entre 0 y 16777215.
- **DECIMAL (NUMERIC)**: Almacena los números de coma flotante como cadenas o string.
- **FLOAT (m,d)**: Almacena números de **coma flotante**, donde 'm' es el número de dígitos de la parte entera y 'd' el número de decimales.
- **DOUBLE (REAL)**: Almacena número de coma flotante con precisión doble. Igual que FLOAT, la diferencia es el rango de valores posibles.
- **BIT (BOOL, BOOLEAN)**: Número entero con valor 0 o 1.

Tipos de dato con formato fecha

Listado de cada uno de los tipos de dato con formato fecha en MySQL, su ocupación en disco y valores:

- **DATE**: Válido para almacenar una fecha con año, mes y día, su rango oscila entre '1000-01-01' y '9999-12-31'.
- **DATETIME**: Almacena una fecha (año-mes-día) y una hora (horas-minutos-segundos), su rango oscila entre '1000-01-01 00:00:00' y '9999-12-31 23:59:59'.
- **TIME**: Válido para almacenar una hora (horas-minutos-segundos). Su rango de horas oscila entre -838-59-59 y 838-59-59. El formato almacenado es 'HH:MM:SS'.
- **TIMESTAMP**: Almacena una fecha y hora UTC. El rango de valores oscila entre '1970-01-01 00:00:01' y '2038-01-19 03:14:07'.
- **YEAR**: Almacena un año dado con 2 o 4 dígitos de longitud, por defecto son 4. El rango de valores oscila entre 1901 y 2155 con 4 dígitos. Mientras que con 2 dígitos el rango es desde 1970 a 2069 (70-69).

Diferentes tipos de dato con formato string

Listado de cada uno de los tipos de dato con formato string en MySQL, su ocupación en disco y valores:

- **CHAR**: Ocupación fija cuya longitud comprende de 1 a 255 caracteres.
- **VARCHAR**: Ocupación variable cuya longitud comprende de 1 a 255 caracteres.
- **TINYBLOB**: Una longitud máxima de 255 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúculas y mayúsculas.
- **BLOB**: Una longitud máxima de 65.535 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúculas y mayúsculas.
- **MEDIUMBLOB**: Una longitud máxima de 16.777.215 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúculas y mayúsculas.
- **LONGBLOB**: Una longitud máxima de 4.294.967.298 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúculas y mayúsculas.
- **SET**: Almacena 0, uno o varios valores una lista con un máximo de 64 posibles valores.
- **ENUM**: Igual que **SET** pero solo puede almacenar un valor.
- **TINYTEXT**: Una longitud máxima de 255 caracteres. Sirve para almecenar texto plano sin formato. Distingue entre minúculas y mayúsculas.
- **TEXT**:Una longitud máxima de 65.535 caracteres. Sirve para almecenar texto plano sin formato. Distingue entre minúculas y mayúsculas.
- **MEDIUMTEXT**:Una longitud máxima de 16.777.215 caracteres. Sirve para almecenar texto plano sin formato. Distingue entre minúculas y mayúsculas.
- **LONGTEXT**: Una longitud máxima de 4.294.967.298 caracteres. Sirve para almecenar texto plano sin formato. Distingue entre minúculas y mayúsculas.

FUNCIONES DE GRUPOS

Listado de cada uno de los tipos de funciones que se pueden usar con GROUP BY:

- **COUNT**: Devuelve el número de valores distintos de NULL en las filas recuperadas por una sentencia SELECT
- **AVG**: Devuelve el valor medio
- **BIT_AND:** Devuelve la operación de bits AND: para todos los bits de una expresión
- **BIT_OR**: Devuelve la operación de bits OR: para todos los bits de una expresión
- **BIT_XOR:** Devuelve la operación de bits XOR: para todos los bits de una expresión
- **COUNT DISTINCT:** Devuelve el número de valores diferentes, distintos de NULL
- **GROUP_CONCAT:** Devuelve una cadena con la concatenación de los valores de un grupo
- **MIN:** Devuelve el valor mínimo de una expresión
- **MAX:** Devuelve el valor máximo de una expresión
- **STD:** o STDDEV Devuelve la desviación estándar de una expresión
- **SUM:** Devuelve la suma de una expresión
- **VARIANCE:** Devuelve la varianza estándar de una expresión

BIBLIOGRAFÍA

Este manual de MySQL se ha realizado usando la siguiente bibliografía:

1. Curso online: “MySQL con clase”, http://mysql.conclase.net/curso/

2. Manual de referencia de la página oficial de MySQL: https://dev.mysql.com/doc/refman/5.7/en/