Intro a SQL y MySQL

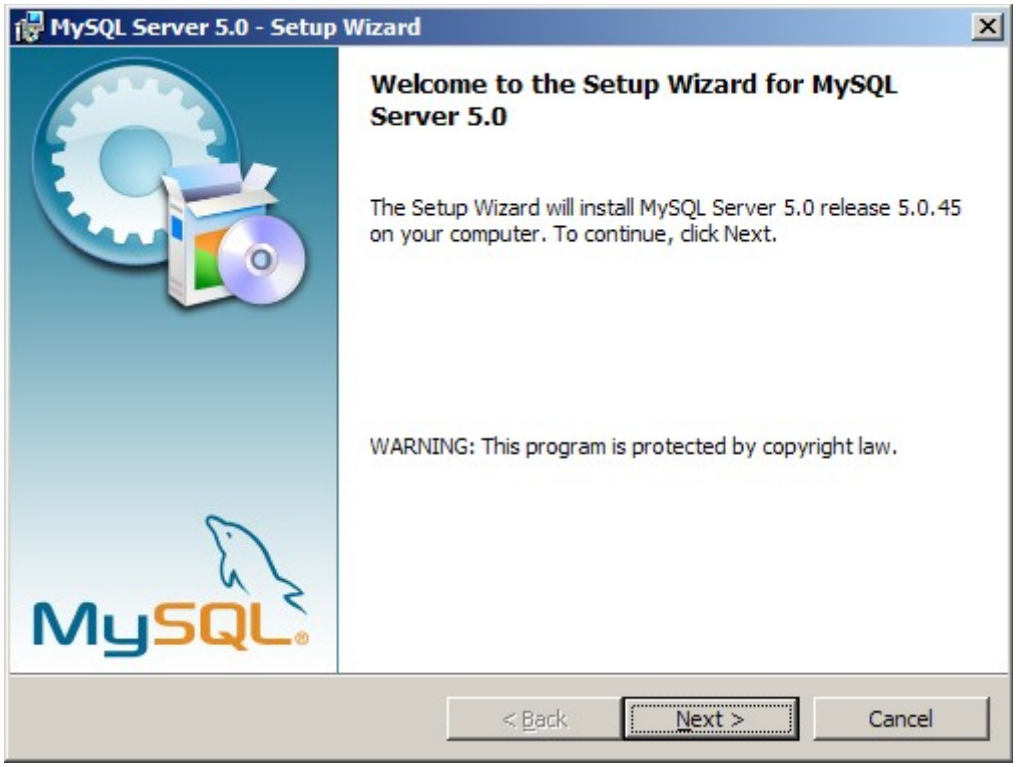
Por Nacho Cabanes, 2008

0. Instalación de MySQL

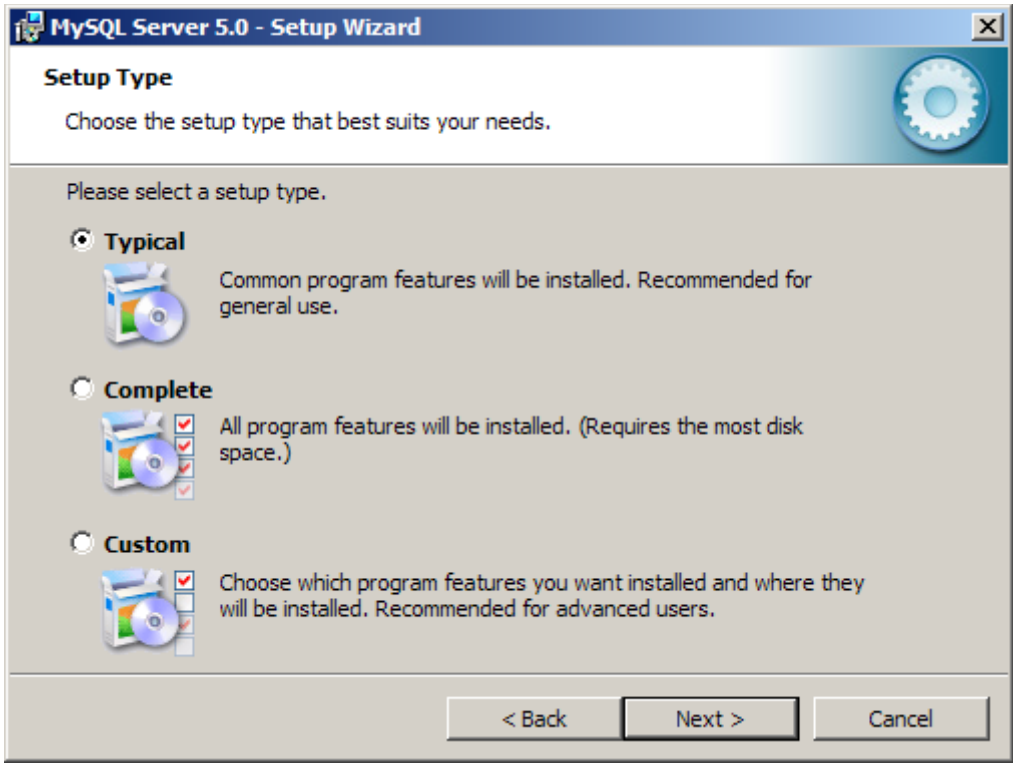
El primer paso para practicar con MySQL será instalar este gestor de bases de datos. Los pasos que siguen estarán dados con MySQL 5.0.45 para Windows.

Desde su página oficial www.mysql.org/downloads tenemos acceso a la versión 5.0 (la última estable en el momento de escribir este texto), a la 5.1 y la 6.0 (en desarrollo). Si escogemos la estable (5.0), se nos da a elegir entre la versión "Community Server", que es gratuita y suficiente para nuestras necesidades, y la versión "Enterprise", de pago. Optaremos por la Community. Hay una instalación mínima ("Essentials", de unos 23 Mb de tamaño) y una completa, de 42 Mb. Yo instalaré la versión completa.

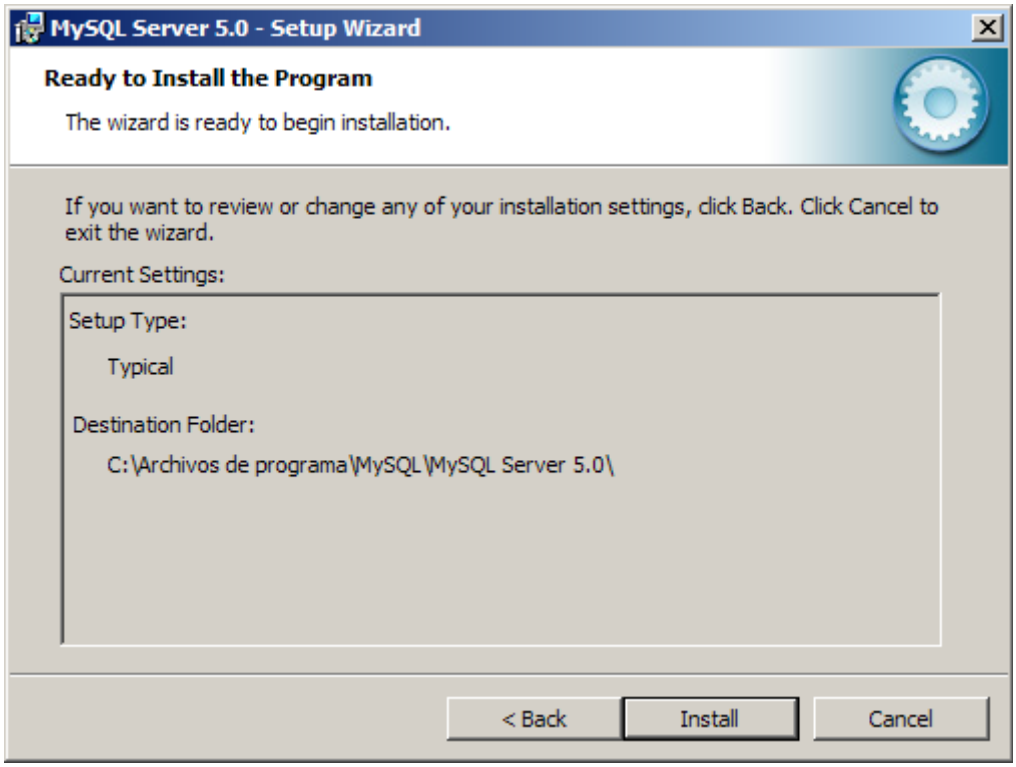
Una vez descargado ese fichero comprimido de 42 Mb, lo descomprimimos (es un ZIP, así que el propio Windows debería permitir manejarlo) y hacemos doble clic en el fichero SETUP resultante, para comenzar la instalación.



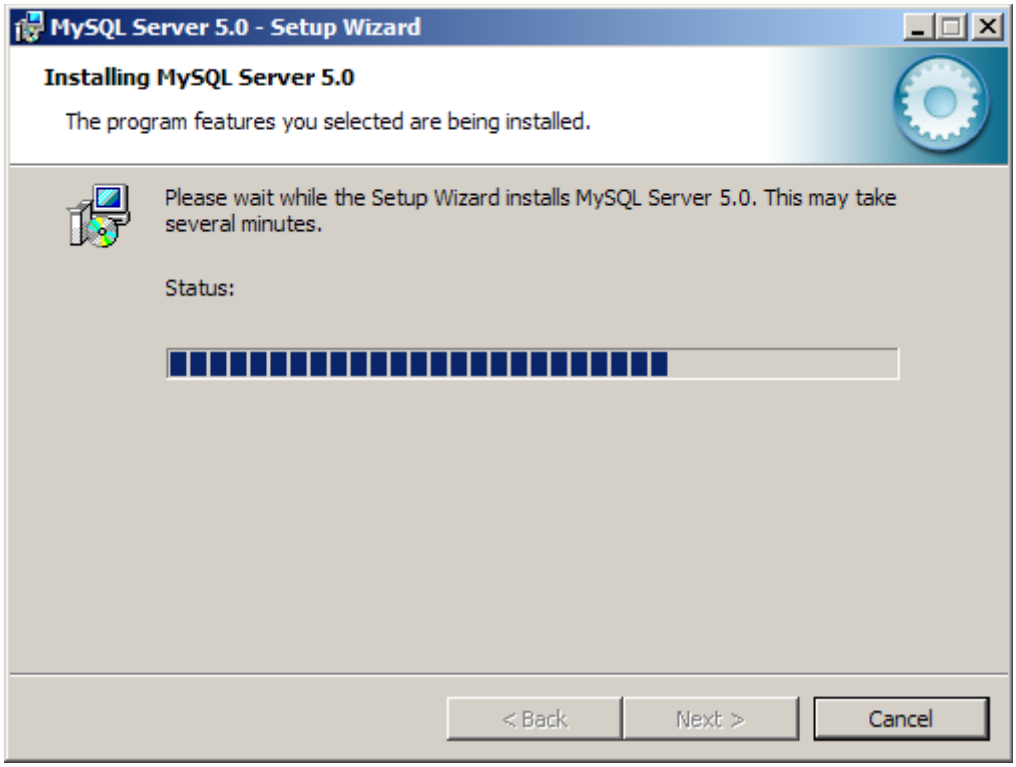
Nos preguntará si queremos una instalación típica, completa o a medida. Como somos novatos, yo elegiría (por ahora) la típica:



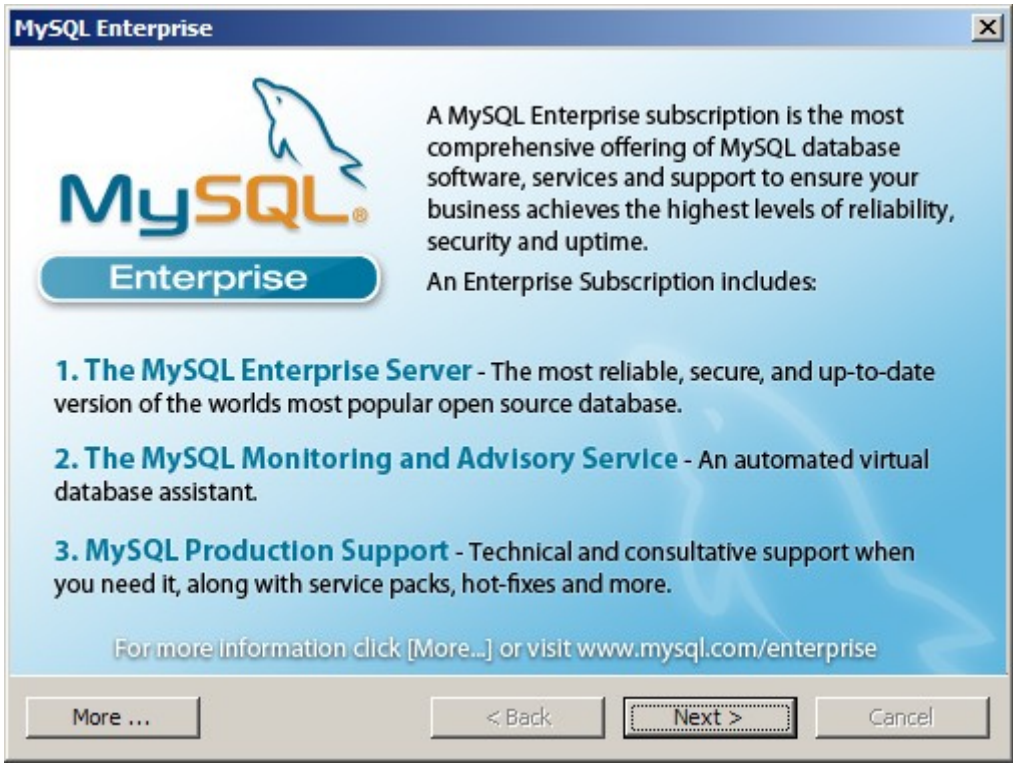
En la instalación típica bajo Windows, la aplicación quedará instalada dentro de "Archivos de Programa":

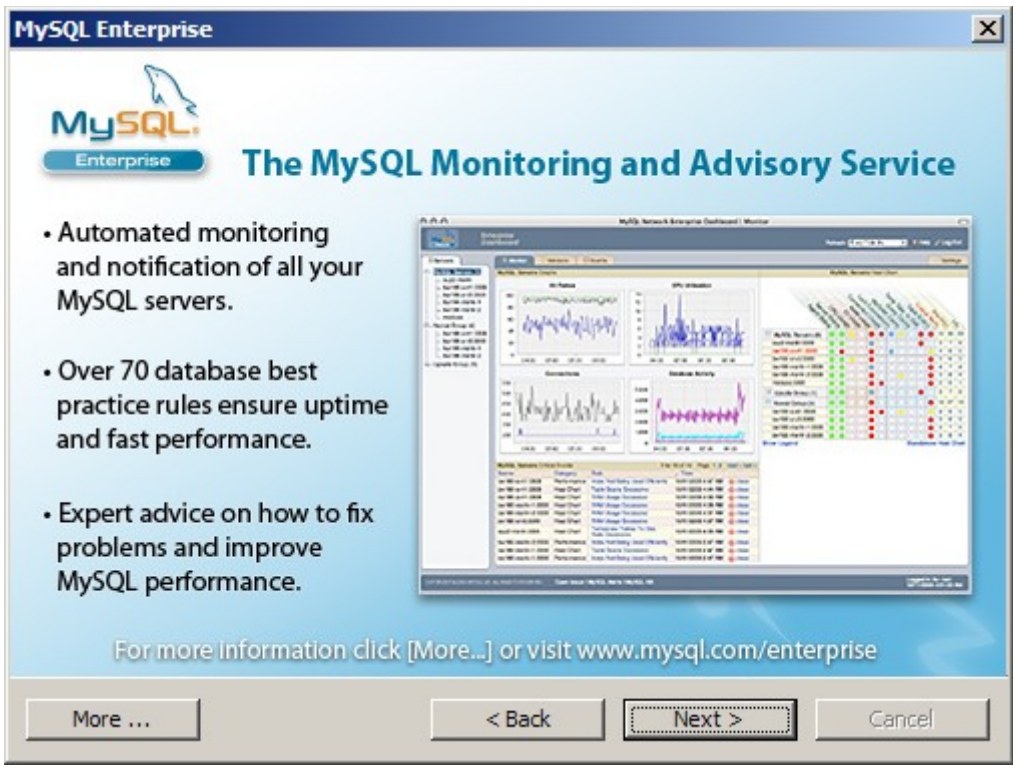


Y tardará apenas unos pocos segundos en copiar ficheros:



También se encargará de mostrarnos algo de propaganda de la versión "Enterprise", de pago, y de sus bondades:

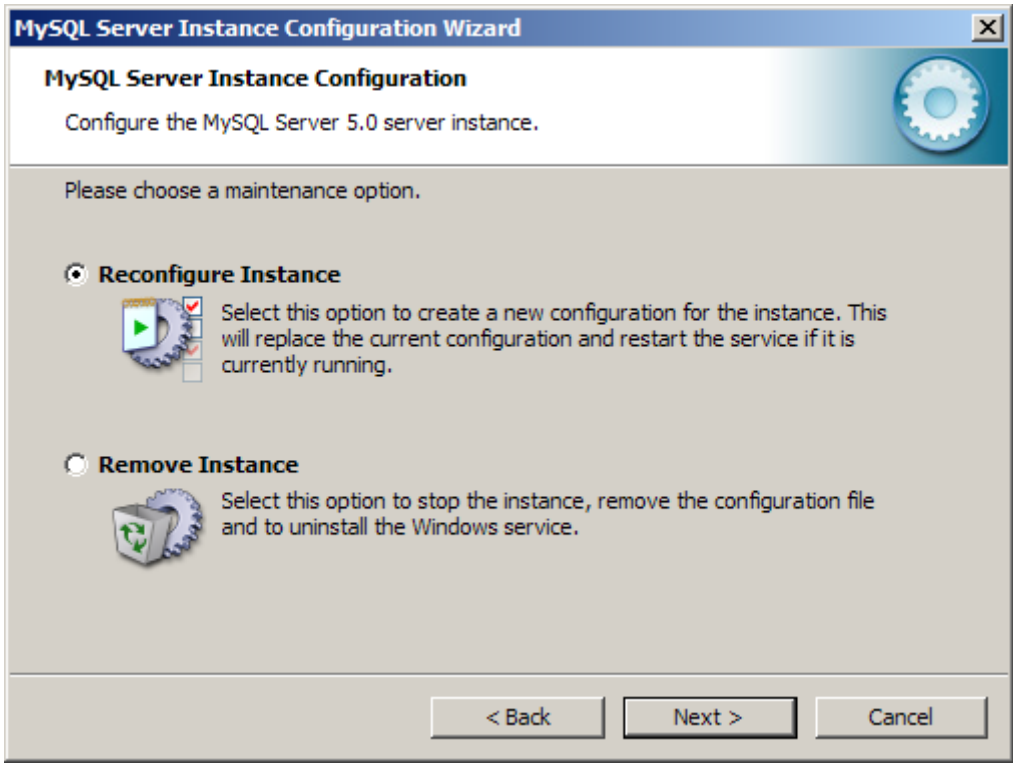




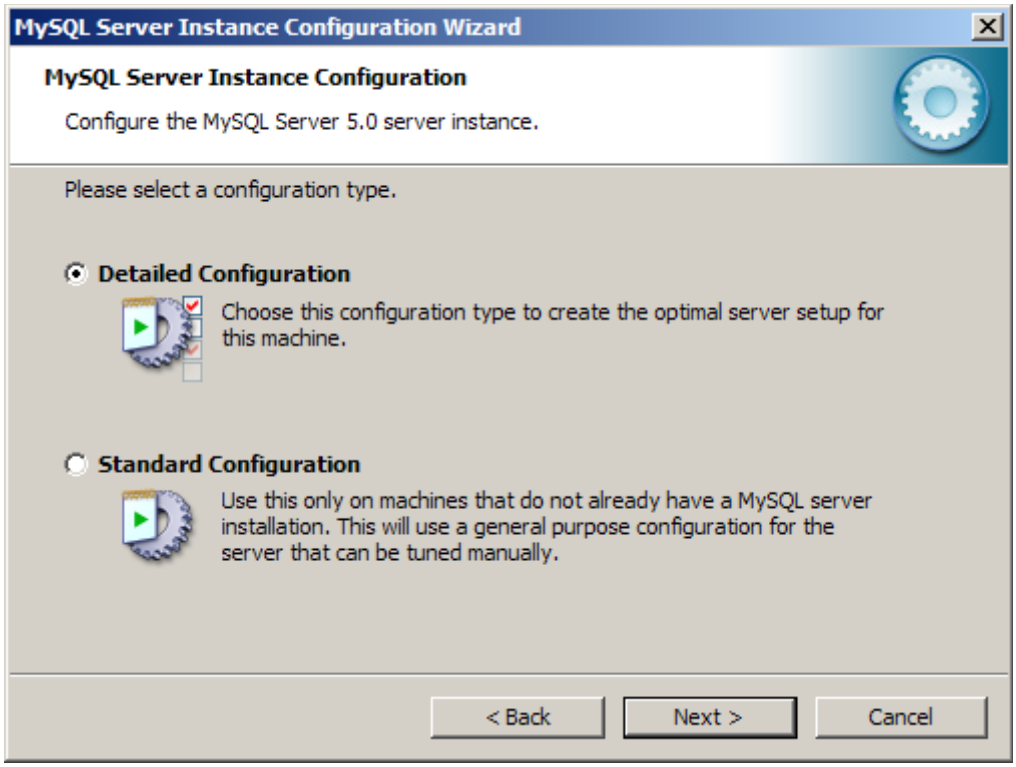
Cuando haya terminado de copiar archivos, nos preguntará si queremos configurar el servidor de bases de datos. Lo razonable es decir que sí:

La primera pantalla de configuración es simplemente la bienvenida:

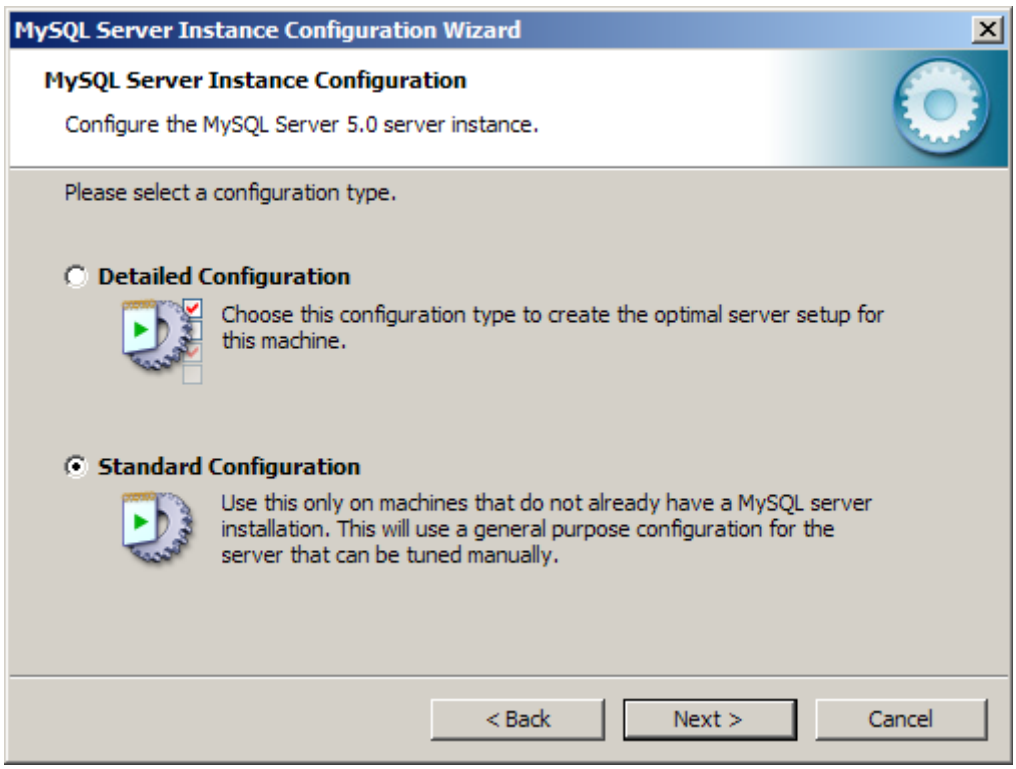
Después nos preguntará si queremos configurarlo o eliminar una instalación existente:



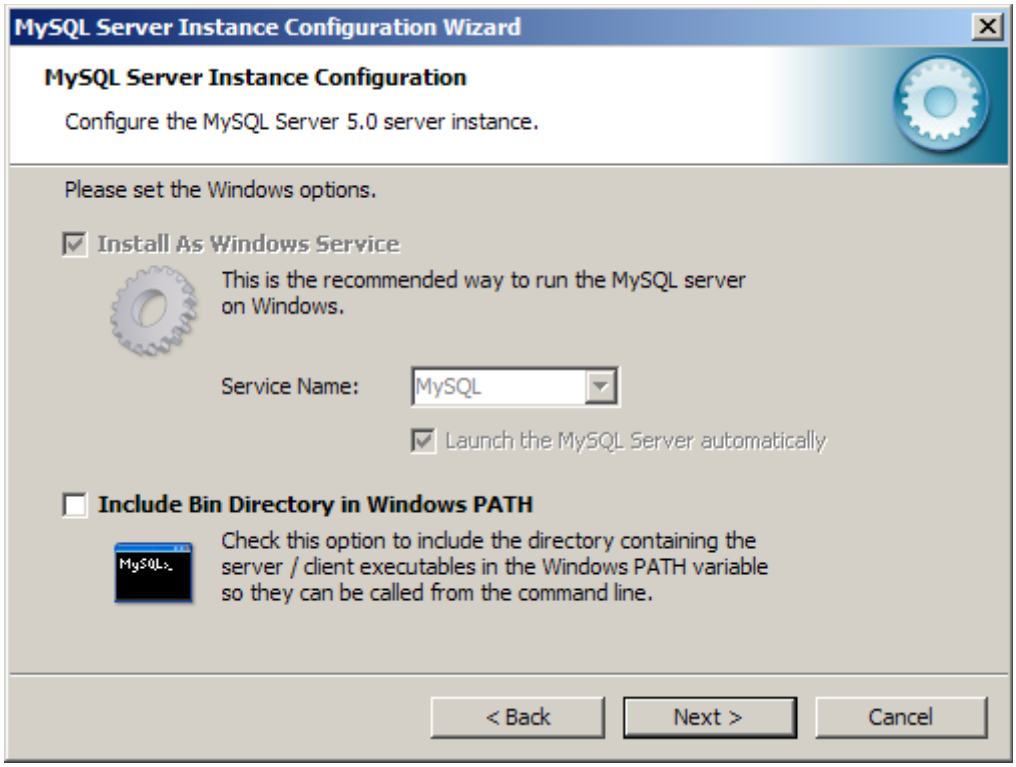
Podemos elegir una configuración detallada o una estándar...



Como somos novatos y no necesitaremos grandes cosas, yo optaría por una configuración estándar:

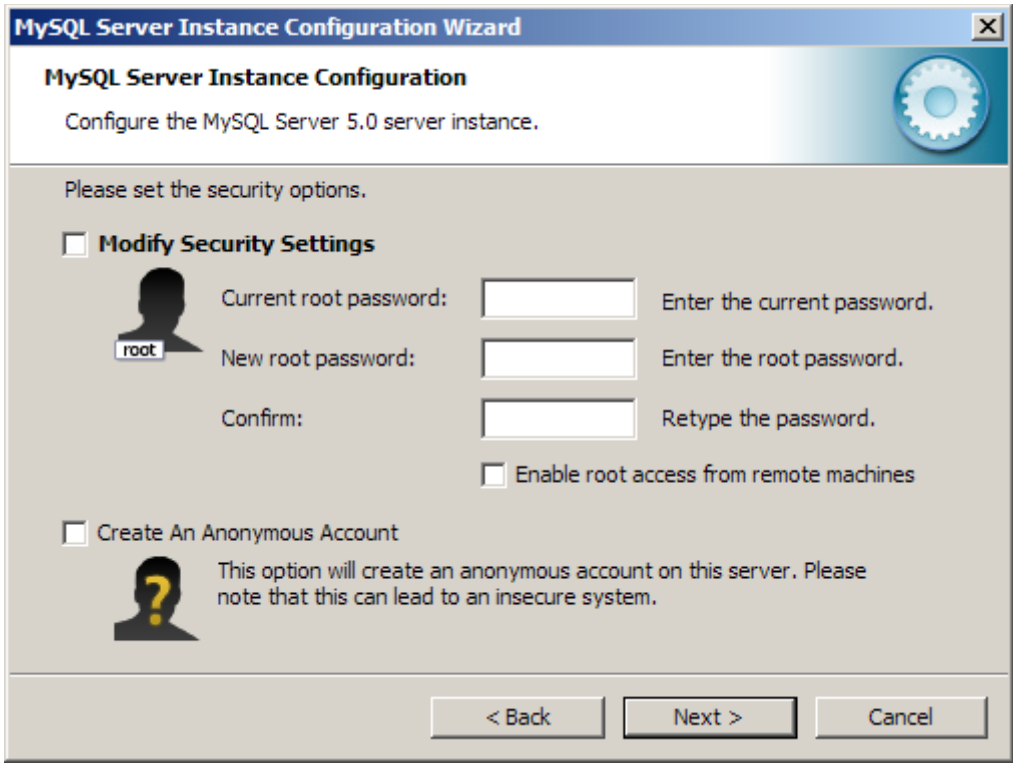


A continuación dará por sentado, que ya que instalamos un servidor, siempre se pondrá en marcha como servicio de Windows. También nos preguntará si queremos que se incluya en el "path" del sistema, de modo que podamos entrar al entorno desde cualquier carpeta. No es una mala opción, pero yo no lo haré, sino que entraré a la carpeta tecleando un par de órdenes de MsDos/Windows ("cd" para entrar a una carpeta y poco más):

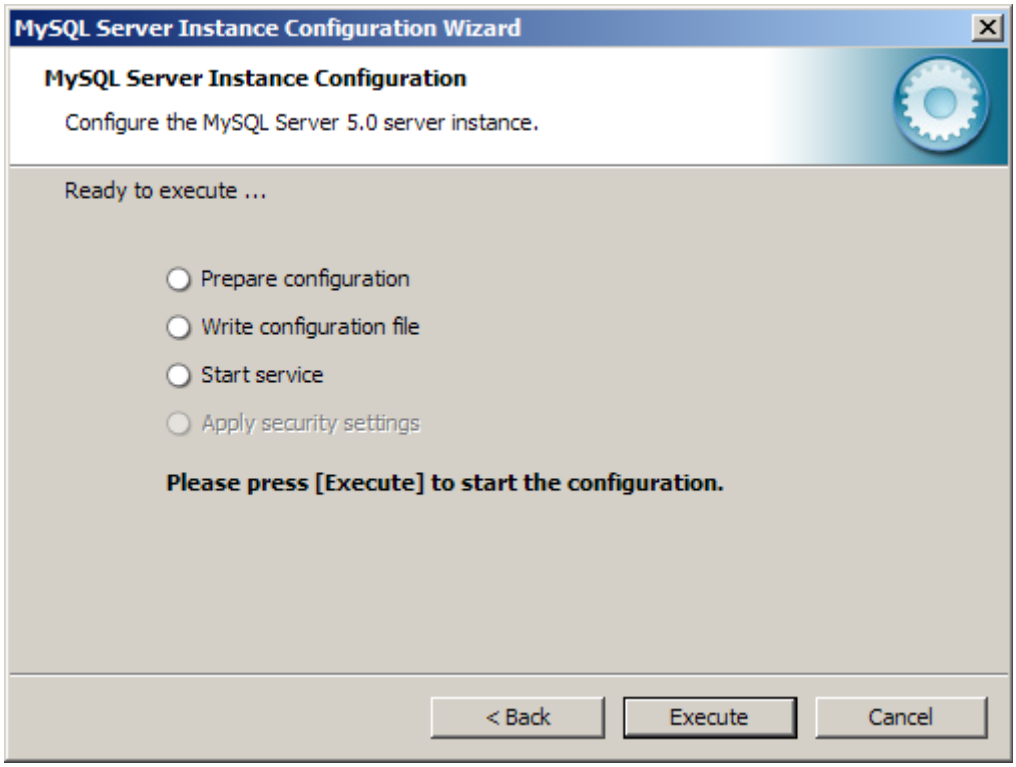


Después nos preguntará la contraseña del administrador (usuario "root") y si queremos crear una cuenta anónima.

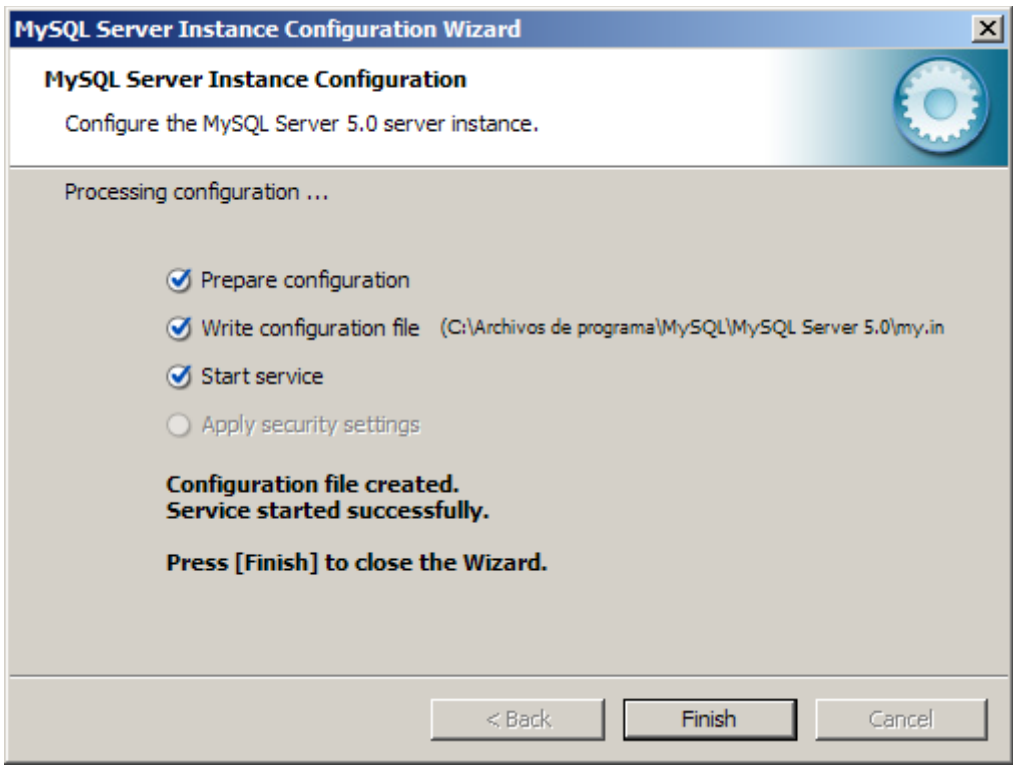
Yo optaré por una solución intermedia (poco segura de todas formas): no crearé cuenta anónima, pero tampoco indicaré contraseña para el usuario root:



Entonces MySQL nos mostrará un resumen de los pasos que va a dar...



Y nos avisará cuando todo termine:



Ya está instalado.

En el próximo apartado veremos cómo entrar al entorno normal de mysql, cómo crear una base de datos, cómo introducir algún dato y cómo hacer consultas básicas sobre los datos existentes.

1. Consultas básicas con una tabla

1.1 Entrando a MySQL

Si hemos hecho la instalación típica, MySQL debería haber quedado dentro de "Archivos de programa".

Para llegar hasta allí, entramos al intérprete de comandos de Windows (por ejemplo, desde el menú de Inicio, en la opción "Ejecutar", tecleando la orden "cmd"), entonces usamos la orden "cd" para llegar hasta la carpeta "bin", en la que están los "binarios" (programas ejecutables) de MySQL. Debería ser algo como:

```
cd "Archivos de programa\MySQL\MySQL Server 5.0\bin"
```

Para entrar al entorno de MySQL, como no hemos permitido el acceso anónimo, tendremos que indicar un nombre de usuario con la opción "-u". Como por ahora sólo existe el usuario "root", escribiríamos:

```
mysql -u root
```

Y entonces deberíamos ver algo parecido a:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.45-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

Ya estamos dentro del "monitor de MySQL". Ahora podemos teclear órdenes directamente en lenguaje SQL y ver los resultados.

1.2 Creando la estructura

En este primer ejemplo, crearemos una base de datos sencilla, que llamaremos "ejemplo1". Esta base de datos contendrá una única tabla, llamada "agenda", que contendrá algunos datos de cada uno de nuestros amigos. Como es nuestra primera base de datos, no pretendemos que sea perfecta, sino sencilla, así que apenas guardaremos tres datos de cada amigo: el nombre, la dirección y la edad.

Para crear la base de datos que contiene todo, usaremos "create database", seguido del nombre que tendrá la base de datos:

```
CREATE DATABASE ejemplo1;
```

Podemos tener varias bases de datos en nuestro SGBD (Sistema Gestor de Bases de Datos), así que deberemos indicar cual de ellas queremos usar, con la orden "use":

```
USE ejemplo1;
```

1.3 Introduciendo datos

Una base de datos, en general, estará formada por varios bloques de información llamados "tablas". En nuestro caso, nuestra tabla almacenará los datos de nuestros amigos. Por tanto, el siguiente paso será decidir qué datos concretos (lo llamaremos "campos") guardaremos de cada amigo. Deberemos pensar también qué tamaño necesitaremos para cada uno de esos datos, porque al gestor de bases de datos habrá que dárselo bastante cuadriculado. Por ejemplo, podríamos decidir lo siguiente:

- nombre - texto, hasta 20 letras
- dirección - texto, hasta 40 letras
- edad - números, de hasta 3 cifras

Cada gestor de bases de datos tendrá una forma de llamar a esos tipos de datos. Por ejemplo, en MySQL podemos usar "VARCHAR" para referirnos a texto hasta una cierta longitud, y "NUMERIC"

para números de una determinada cantidad de cifras, de modo que la orden necesaria para crear esta tabla sería:

```
CREATE TABLE personas (
  nombre varchar(20),
  direccion varchar(40),
  edad decimal(3)
);
```

Para introducir datos usaremos la orden "insert", e indicaremos tras la palabra "values" los valores para los campos de texto entre comillas, y los valores para campos numéricos sin comillas, así:

```
INSERT INTO personas VALUES ('juan', 'su casa', 25);
```

1.4 Mostrando datos

Para ver los datos almacenados en una tabla usaremos el formato "SELECT campos FROM tabla". Si queremos ver todos los campos, lo indicaremos usando un asterisco:

```
SELECT * FROM personas;
```

que, en nuestro caso, daría como resultado

```
+--------+-----------+------+
| nombre | direccion | edad |
+--------+-----------+------+
| juan   | su casa   | 25   |
| pedro  | su calle  | 23   |
+--------+-----------+------+
```

Si queremos ver sólo ciertos campos, detallamos sus nombres, separados por comas:

```
SELECT nombre, direccion FROM personas;
```

y obtendríamos

```
+--------+-----------+
| nombre | direccion |
+--------+-----------+
| juan   | su casa   |
| pedro  | su calle  |
+--------+-----------+
```

Normalmente no querremos ver todos los datos que hemos introducido, sino sólo aquellos que cumplan cierta condición. Esta condición se indica añadiendo un apartado WHERE a la orden "select", así:

```
SELECT nombre, direccion FROM personas WHERE nombre = 'juan';
```

que nos diría el nombre y la dirección de nuestros amigos llamados "juan":

```
+--------+-----------+
| nombre | direccion |
+--------+-----------+
| juan   | su casa   |
+--------+-----------+
```

A veces no querremos comparar con un texto exacto, sino sólo con parte del contenido del

campo (por ejemplo, porque sólo sepamos un apellido o parte de la calle). En ese caso, no compararíamos con el símbolo "igual" (=), sino que usaríamos la palabra "like", y para las partes que no conozcamos usaremos el comodín "%", como en este ejemplo:

```
SELECT nombre, direccion FROM personas WHERE direccion LIKE '%calle%';
```

que nos diría el nombre y la dirección de nuestros amigos llamados que viven en calles que contengan la palabra "calle", precedida por cualquier texto (%) y con cualquier texto (%) a continuación:

```
+--------+-----------+
| nombre | direccion |
+--------+-----------+
| pedro  | su calle  |
+--------+-----------+
```

1.5 Saliendo de MySQL

Es suficiente por hoy. Para terminar nuestra sesión de MySQL, tecleamos la orden quit o exit y volvemos al sistema operativo.

2. Consultas básicas con dos tablas

2.1 Formalizando conceptos

Hay algunas cosas que hemos pasado por alto y que no estaría mal formalizar un poco.

- SQL es un lenguaje de consulta a bases de datos. Sus siglas vienen de Structured Query Language (lenguaje de consulta estructurado).
- MySQL es un "gestos de bases de datos", es decir, una aplicación informática que se usa para crear y manipular bases de datos (realmente, se les exige una serie de cosas más, pero por ahora nos basta con eso).
- En MySQL, las órdenes que tecleamos deben terminar en punto y coma (;). Si tecleamos una orden como "select * from personas" y pulsamos Intro, MySQL responderá mostrando "->" para indicar que todavía no hemos terminado la orden.

2.2 ¿Por qué varias tablas?

Puede haber varios motivos.

Por una parte, podemos tener bloques de información claramente distintos. Por ejemplo, en una base de datos que guarde la información de una empresa tendremos datos como los artículos que distribuimos y los clientes que nos los compran, que no deberían guardarse en una misma tabla.

Por otra parte, habrá ocasiones en que veamos que los datos, a pesar de que se podrían clasificar dentro de un mismo "bloque de información" (tabla), serían redundantes: existiría gran cantidad de datos repetitivos, y esto puede dar lugar a dos problemas:

- Espacio desperdiciado.
- Posibilidad de errores al introducir los datos, lo que daría lugar a inconsistencias:

Veamos un ejemplo:

```
+---------+-----------+-----------+
| nombre  | direccion | ciudad    |
+---------+-----------+-----------+
| juan    | su casa   | alicante  |
| alberto | calle uno | alicante  |
| pedro   | su calle  | alicantw  |
+---------+-----------+-----------+
```

Si en vez de repetir "alicante" en cada una de esas fichas, utilizásemos un código de ciudad, por ejemplo "a", gastaríamos menos espacio (en este ejemplo, 7 bytes menos en cada ficha).

Por otra parte, hemos tecleado mal uno de los datos: en la tercera ficha no hemos indicado "alicante", sino "alicantw", de modo que si hacemos consultas sobre personas de Alicante, la última de ellas no aparecería. Al teclear menos, es también más difícil cometer este tipo de errores.

A cambio, necesitaremos una segunda tabla, en la que guardemos los códigos de las ciudades, y el nombre al que corresponden (por ejemplo: si códigoDeCiudad = "a", la ciudad es "alicante").

2.3 Las claves primarias

Generalmente, será necesario tener algún dato que nos permita distinguir de forma clara los datos que tenemos almacenados. Por ejemplo, el nombre de una persona no es único: pueden aparecer en nuestra base de datos varios usuarios llamados "Juan López". Si son nuestros clientes, debemos saber cual es cual, para no cobrar a uno de ellos un dinero que corresponde a otro. Eso se suele solucionar guardando algún dato adicional que sí sea único para cada cliente, como puede ser el Documento Nacional de Identidad, o el Pasaporte. Si no hay ningún dato claro que nos sirva, en ocasiones añadiremos un "código de cliente", inventado por nosotros, o algo similar.

Estos datos que distinguen claramente unas "fichas" de otras los llamaremos "claves primarias".

2.4 Creando datos

Comenzaremos creando una nueva base de datos, de forma similar al ejemplo anterior:

```
CREATE DATABASE ejemplo2;
USE ejemplo2;
```

Después creamos la tabla de ciudades, que guardará su nombre y su código. Este código será el

que actúe como "clave primaria", para distinguir otra ciudad. Por ejemplo, hay una ciudad llamado "Toledo" en España, pero también otra en Argentina, otra en Uruguay, dos en Colombia, una en Ohio (Estados Unidos)... el nombre claramente no es único, así que podríamos usar código como "te" para Toledo de España, "ta" para Toledo de Argentina y así sucesivamente.

La forma de crear la tabla con esos dos campos y con esa clave primaria sería:

```
CREATE TABLE ciudades (
  codigo varchar(3),
  nombre varchar(30),
  PRIMARY KEY (codigo)
);
```

Mientras que la tabla de personas sería casi igual al ejemplo anterior, pero añadiendo un nuevo dato: el código de la ciudad

```
CREATE TABLE personas (
  nombre varchar(20),
  direccion varchar(40),
  edad decimal(3),
  codciudad varchar(3)
);
```

Para introducir datos, el hecho de que exista una clave primaria no supone ningún cambio, salvo por el hecho de que no se nos dejaría introducir dos ciudades con el mismo código:

```
INSERT INTO ciudades VALUES ('a', 'alicante');
INSERT INTO ciudades VALUES ('b', 'barcelona');
INSERT INTO ciudades VALUES ('m', 'madrid');

INSERT INTO personas VALUES ('juan', 'su casa', 25, 'a');
INSERT INTO personas VALUES ('pedro', 'su calle', 23, 'm');
INSERT INTO personas VALUES ('alberto', 'calle uno', 22, 'b');
```

2.5 Mostrando datos

Cuando queremos mostrar datos de varias tablas a la vez, deberemos hacer unos pequeños cambios en las órdenes "select" que hemos visto:

- En primer lugar, indicaremos varios nombres después de "FROM" (los de cada una de las tablas que necesitemos).
- Además, puede ocurrir que cada tengamos campos con el mismo nombre en distintas tablas (por ejemplo, el nombre de una persona y el nombre de una ciudad), y en ese caso deberemos escribir el nombre de la tabla antes del nombre del campo.

Por eso, una consulta básica sería algo parecido (sólo parecido) a:

```
SELECT personas.nombre, direccion, ciudades.nombre FROM personas, ciudades;
```

Pero esto todavía tiene problemas: estamos combinando TODOS los datos de la tabla de personas con TODOS los datos de la tabla de ciudades, de modo que obtenemos 3x3 = 9 resultados:

```
+---------+-----------+-----------+
| nombre  | direccion | nombre    |
+---------+-----------+-----------+
| juan    | su casa   | alicante  |
| pedro   | su calle  | alicante  |
| alberto | calle uno | alicante  |
| juan    | su casa   | barcelona |
| pedro   | su calle  | barcelona |
| alberto | calle uno | barcelona |
| juan    | su casa   | madrid    |
| pedro   | su calle  | madrid    |
| alberto | calle uno | madrid    |
+---------+-----------+-----------+
9 rows in set (0.00 sec)
```

Pero esos datos no son reales: si "juan" vive en la ciudad de código "a", sólo debería mostrarse junto al nombre "alicante". Nos falta indicar esa condición: "el código de ciudad que aparece en la persona debe ser el mismo que el código que aparece en la ciudad", así:

```
SELECT personas.nombre, direccion, ciudades.nombre
FROM personas, ciudades
WHERE personas.codciudad = ciudades.codigo;
```

Esta será la forma en que trabajaremos normalmente. Este último paso se puede evitar en ciertas circunstancias, pero ya lo veremos más adelante. El resultado de esta consulta sería:

```
+---------+-----------+-----------+
| nombre  | direccion | nombre    |
+---------+-----------+-----------+
| juan    | su casa   | alicante  |
| alberto | calle uno | barcelona |
| pedro   | su calle  | madrid    |
+---------+-----------+-----------+
```

Ese sí es el resultado correcto. Cualquier otra consulta que implique las dos tablas deberá terminar comprobando que los dos códigos coinciden. Por ejemplo, para ver qué personas viven en la ciudad llamada "madrid", haríamos:

```
SELECT personas.nombre, direccion, edad
FROM personas, ciudades
WHERE ciudades.nombre='madrid'
AND personas.codciudad = ciudades.codigo;
```

```
+--------+-----------+------+
| nombre | direccion | edad |
+--------+-----------+------+
| pedro  | su calle  |   23 |
+--------+-----------+------+
```

Y para saber las personas de ciudades que comiencen con la letra "b", haríamos:

```
SELECT personas.nombre, direccion, ciudades.nombre
FROM personas, ciudades
WHERE ciudades.nombre LIKE 'b%'
AND personas.codciudad = ciudades.codigo;
```

```
+---------+-----------+-----------+
| nombre  | direccion | nombre    |
```

```
+---------+-----------+-----------+
| alberto | calle uno | barcelona |
+---------+-----------+-----------+
```

Si en nuestra tabla puede haber algún dato que se repita, como la dirección, podemos pedir un listado sin duplicados, usando la palabra "distinct":

```
SELECT DISTINCT direccion FROM personas;
```

2.6 Ejecutando un lote de órdenes

Hasta ahora hemos tecleado todas las órdenes desde dentro del entorno de MySQL, una por una. Tenemos otra opción que también puede ser cómoda: crear un fichero de texto que contenga todas las órdenes y cargarlo después desde MySQL. Lo podemos hacer de dos formas:

- Usar la orden "source": desde dentro de MySQL teclearíamos algo como

  `source ejemplo2.sql;`

- Cargar las órdenes justo en el momento de entrar a MySQL, con

  `mysql -u root < ejemplo2.sql;`

Pero esta última alternativa tiene un problema: se darán los pasos que indiquemos en "ejemplo2.sql" y se abandonará el entorno de MySQL, sin que nos dé tiempo de comprobar si ha existido algún mensaje de error.

3. Borrado de datos

3.1 ¿Qué información hay?

Un primer paso antes de ver cómo borrar información es saber qué información tenemos almacenada.

Podemos saber las bases de datos que hay creadas en nuestro sistema con:

```
SHOW DATABASES;
```

Una vez que estamos trabajando con una base de datos concreta (con la orden "use"), podemos saber las tablas que contiene con:

```
SHOW TABLES;
```

Y para una tabla concreta, podemos saber los campos (columnas) que la forman con "show columns from":

```
SHOW COLUMNS FROM personas;
```

Por ejemplo, esto daría como resultado:

```
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
```

```
+-----------+--------------+------+-----+---------+-------+
| nombre    | varchar(20)  | YES  |     | NULL    |       |
| direccion | varchar(40)  | YES  |     | NULL    |       |
| edad      | decimal(3,0) | YES  |     | NULL    |       |
| codciudad | varchar(3)   | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
```

3.2 Borrar toda la base de datos

En alguna ocasión, como ahora que estamos practicando, nos puede interesar borrar toda la base de datos. La orden para conseguirlo es:

```
DROP DATABASE ejemplo2;
```

Si esta orden es parte de una secuencia larga de órdenes, que hemos cargado con la orden "source" (por ejemplo) y la base de datos no existe, obtendríamos un mensaje de error y se interrumpiría el proceso. Podemos evitarlo añadiendo "if exists", para que se borre la base de datos sólo si realmente existe:

```
DROP DATABASE ejemplo2 IF EXISTS;
```

3.3 Borrar una tabla

Es más frecuente que creemos alguna tabla de forma incorrecta. La solución razonable es corregir ese error, cambiando la estructura de la tabla, pero todavía no sabemos hacerlo. Al menos veremos cómo borrar una tabla. La orden es:

```
DROP TABLE personas;
```

Al igual que para las bases de datos, podemos hacer que la tabla se borre sólo cuando realmente existe:

```
DROP TABLE personas IF EXISTS;
```

3.4 Borrar datos de una tabla

También podemos borrar los datos que cumplen una cierta condición. La orden es "delete from", y con "where" indicamos las condiciones que se deben cumplir, de forma similar a como hacíamos en la orden "select":

```
DELETE FROM personas WHERE nombre = 'juan';
```

Esto borraría todas las personas llamadas "juan" que estén almacenadas en la tabla "personas".

Cuidado: si no se indica la parte de "where", no se borrarían los datos que cumplen una condición, sino TODOS los datos. Si es eso lo que se pretende, una forma más rápida de conseguirlo es usar:

```
TRUNCATE TABLE personas;
```

4. Modificación de datos

4.1 Modificación de datos

Ya sabemos borrar datos, pero existe una operación más frecuente que esa (aunque también ligeramente más complicada): modificar los datos existentes. Con lo que sabíamos hasta ahora, podíamos hacer algo parecido: si un dato es incorrecto, podríamos borrarlo y volver a introducirlo, pero esto, obviamente, no es lo más razonable... debería existir alguna orden para cambiar los datos. Así es. El formato habitual para modificar datos de una tabla es "update tabla set campo=nuevoValor where condicion".

Por ejemplo, si hemos escrito "Alberto" en minúsculas ("alberto"), lo podríamos corregir con:

```
UPDATE personas SET nombre = 'Alberto' WHERE nombre = 'alberto';
```

Y si queremos corregir todas las edades para sumarles un año se haría con

```
UPDATE personas SET edad = edad+1;
```

(al igual que habíamos visto para "select" y para "delete", si no indicamos la parte del "where", los cambios se aplicarán a todos los registros de la tabla).

4.2 Modificar la estructura de una tabla

Algo más complicado es modificar la estructura de una tabla: añadir campos, eliminarlos, cambiar su nombre o el tipo de datos. En general, para todo ello se usará la orden "alter table". Vamos a ver las posibilidades más habituales.

Para añadir un campo usaríamos "add":

```
ALTER TABLE ciudades ADD habitantes decimal(7);
```

Si no se indica otra cosa, el nuevo campo se añade al final de la tabla. Si queremos que sea el primer campo, lo indicaríamos añadiendo "first" al final de la orden. También podemos hacer que se añada después de un cierto campo, con "after nombreCampo".

Podemos modificar el tipo de datos de un campo con "modify". Por ejemplo, podríamos hacer que el campo "habitantes" no fuera un "decimal" sino un entero largo ("bigint") con:

```
ALTER TABLE ciudades MODIFY habitantes bigint;
```

Si queremos cambiar el nombre de un campo, debemos usar "change" (se debe indicar el nombre antiguo, el nombre nuevo y el tipo de datos). Por ejemplo, podríamos cambiar el nombre "habitantes" por "numhabitantes":

```
ALTER TABLE ciudades CHANGE habitantes numhabitantes bigint;
```

Si queremos borrar algún campo, usaremos "drop column":

```
ALTER TABLE ciudades DROP COLUMN numhabitantes;
```

Muchas de estas órdenes se pueden encadenar, separadas por comas. Por ejemplo, podríamos borrar dos campos con "alter table ciudades drop column num habitantes, drop column

provincia;"

Finalmente, también podríamos cambiar el nombre de una tabla con "rename":

```
ALTER TABLE ciudades RENAME ciudad;
```

5. Operaciones matemáticas

5.1 Operaciones matemáticas

Desde SQL podemos realizar operaciones a partir de los datos antes de mostrarlos. Por ejemplo, podemos mostrar cual era la edad de una persona hace un año, con

```
SELECT edad-1 FROM personas;
```

Los operadores matemáticos que podemos emplear son los habituales en cualquier lenguaje de programación, ligeramente ampliados: + (suma), - (resta y negación), * (multiplicación), / (división) . La división calcula el resultado con decimales; si queremos trabajar con números enteros, también tenemos los operadores DIV (división entera) y MOD (resto de la división):

```
SELECT 5/2, 5 div 2, 5 mod 2;
```

Daría como resultado

```
+--------+---------+---------+
| 5/2    | 5 div 2 | 5 mod 2 |
+--------+---------+---------+
| 2.5000 |       2 |       1 |
+--------+---------+---------+
```

También podríamos utilizar incluso operaciones a nivel de bits, como las del lenguaje C:

```
SELECT 25 >> 1, 25 << 1, 25 | 10, 25 & 10, 25 ^10;
```

que daría

```
+---------+---------+---------+---------+--------+
| 25 >> 1 | 25 << 1 | 25 | 10 | 25 & 10 | 25 ^10 |
+---------+---------+---------+---------+--------+
|      12 |      50 |      27 |       8 |     19 |
+---------+---------+---------+---------+--------+
```

5.2 Funciones de agregación

También podemos aplicar ciertas funciones matemáticas a todo un conjunto de datos de una tabla. Por ejemplo, podemos saber cual es la edad más baja de entre las personas que tenemos en nuestra base de datos, haríamos:

```
SELECT min(edad) FROM personas;
```

Las funciones de agregación más habituales son:

- min = mínimo valor

- max = máximo valor
- sum = suma de los valores
- avg = media de los valores
- count = cantidad de valores

La forma más habitual de usar "count" es pidiendo con "count(*)" que se nos muestren todos los datos que cumplen una condición. Por ejemplo, podríamos saber cuantas personas tienen una dirección que comience por la letra "s", así:

```
SELECT count(*) FROM personas WHERE direccion LIKE 's%';
```

6. Valores nulos

6.1 Cero y valor nulo

En ocasiones querremos dejar un campo totalmente vacío, sin valor.

Para las cadenas de texto, existe una forma "parecida" de conseguirlo, que es con una cadena vacía, indicada con dos comillas que no contengan ningún texto entre ellas (ni siquiera espacios en blanco): ''

En cambio, para los números, no basta con guardar un 0 para indicar que no se sabe el valor: no es lo mismo un importe de 0 euros que un importe no detallado. Por eso, existe un símbolo especial para indicar cuando no existe valor en un campo.

Este símbolo especial es la palabra NULL. Por ejemplo, añadiríamos datos parcialmente en blanco a una tabla haciendo

```
INSERT INTO personas
  (nombre, direccion, edad)
VALUES (
 'pedro', '', NULL
);
```

En el ejemplo anterior, y para que sea más fácil comparar las dos alternativas, he conservado las comillas sin contenido para indicar una dirección vacía, y he usado NULL para la edad, pero sería más correcto usar NULL en ambos casos para indicar que no existe valor, así:

```
INSERT INTO personas
  (nombre, direccion, edad)
VALUES (
 'pedro', NULL, NULL
);
```

Para saber si algún campo está vacío, compararíamos su valor con NULL, pero de una forma un tanto especial: no con el símbolo "igual" (=), sino con la palabra IS. Por ejemplo, sabríamos cuales de las personas de nuestra bases de datos tienen dirección usando

```
SELECT * FROM personas
  WHERE direccion IS NOT NULL;
```

Y, de forma similar, sabríamos quien no tiene dirección, así:

```
SELECT * FROM personas
  WHERE direccion IS NULL;
```

7. Valores agrupados

7.1 Agrupando los resultados

Puede ocurrir que no nos interese un único valor agrupado (el total, la media, la cantidad de datos), sino el resultado para un grupo de datos. Por ejemplo: saber no sólo la cantidad de clientes que hay registrados en nuestra base de datos, sino también la cantidad de clientes que viven en cada ciudad.

La forma de obtener subtotales es creando grupos con la orden "group by", y entonces pidiendo una valor agrupado (count, sum, avg, ...) para cada uno de esos grupos. Por ejemplo, en nuestra tabla "personas", podríamos saber cuantas personas aparecen de cada edad, con:

```
SELECT count(*), edad FROM personas GROUP BY edad;
```

que daría como resultado

```
+----------+------+
| count(*) | edad |
+----------+------+
|        1 |   22 |
|        1 |   23 |
|        1 |   25 |
+----------+------+
```

7.2 Filtrando los datos agrupados

Pero podemos llegar más allá: podemos no trabajar con todos los grupos posibles, sino sólo con los que cumplen alguna condición.

La condición que se aplica a los grupos no se indica con "where", sino con "having" (que se podría traducir como "los que tengan..."). Un ejemplo:

```
SELECT count(*), edad FROM personas GROUP BY edad HAVING edad > 24;
```

que mostraría

```
+----------+------+
| count(*) | edad |
+----------+------+
|        1 |   25 |
+----------+------+
```

8. Subconsultas

8.1 ¿Qué es una subconsulta?

A veces tenemos que realizar operaciones más complejas con los datos, operaciones en las que nos interesaría ayudarnos de una primera consulta auxiliar que extrajera la información en la que nos queremos basar. Esta consulta auxiliar recibe el nombre de "subconsulta" o "subquery".

Por ejemplo, si queremos saber qué clientes tenemos en la ciudad que más habitantes tenga, la forma "razonable" de conseguirlo sería saber en primer lugar cual es la ciudad que más habitantes tenga, y entonces lanzar una segunda consulta para ver qué clientes hay en esa ciudad.

Como la estructura de nuestra base de datos de ejemplo es muy sencilla, no podemos hacer grandes cosas, pero un caso parecido al anterior (aunque claramente más inútil) podría ser saber qué personas tenemos almacenadas que vivan en la última ciudad de nuestra lista.

Para ello, la primera consulta (la "subconsulta") sería saber cual es la última ciudad de nuestra lista. Si lo hacemos tomando la que tenga el último código, la consulta podría ser

```
SELECT MAX(codigo) FROM ciudades;
```

Vamos a imaginar que pudiéramos hacerlo en dos pasos. Si llamamos "maxCodigo" a ese código obtenido, la "segunda" consulta podría ser:

```
SELECT * FROM personas WHERE codciudad= maxCodigo;
```

Pero estos dos pasos se pueden dar en uno: al final de la "segunda" consulta (la "grande") incluimos la primera consulta (la "subconsulta"), entre paréntesis, así

```
SELECT * FROM personas WHERE codciudad= (
  SELECT MAX(codigo) FROM ciudades
);
```

8.2 Subconsultas que devuelven conjuntos de datos

Si la subconsulta no devuelve un único dato, sino un conjunto de datos, la forma de trabajar será básicamente la misma, pero para comprobar si el valor coincide con uno de la lista, no usaremos el símbolo "=", sino la palabra "in".

Por ejemplo, vamos a hacer una consulta que nos muestre las personas que viven en ciudades cuyo nombre tiene una "a" en segundo lugar (por ejemplo, serían ciudades válidas Madrid o Barcelona, pero no Alicante).

Para consultar qué letras hay en ciertas posiciones de una cadena, podemos usar SUBSTRING (en el próximo apartado veremos las funciones más importantes de manipulación de cadenas). Así, una forma de saber qué ciudades tienen una letra A en su segunda posición sería:

```
SELECT codigo FROM ciudades
WHERE SUBSTRING(nombre,2,1)='a';
```

Como esta subconsulta puede tener más de un resultado, deberemos usar IN para incluirla en la consulta principal, que quedaría de esta forma:

```
SELECT * FROM personas
WHERE codciudad IN
(
  SELECT codigo FROM ciudades WHERE SUBSTRING(nombre,2,1)='a'
);
```

9. Funciones de cadena

En MySQL tenemos muchas funciones para manipular cadenas: calcular su longitud, extraer un fragmento situado a la derecha, a la izquierda o en cualquier posición, eliminar espacios finales o iniciales, convertir a hexadecimal y a binario, etc. Vamos a comentar las más habituales. Los ejemplos estarán aplicados directamente sobre cadenas, pero (por supuesto) también se pueden aplicar a campos de una tabla:

Funciones de conversión a mayúsculas/minúsculas

- LOWER o LCASE convierte una cadena a minúsculas: SELECT LOWER('Hola'); -> hola
- UPPER o UCASE convierte una cadena a mayúsculas: SELECT UPPER('Hola'); -> HOLA

Funciones de extracción de parte de la cadena

- LEFT(cadena, longitud) extrae varios caracteres del comienzo (la parte izquierda) de la cadena: SELECT LEFT('Hola',2); -> Ho
- RIGHT(cadena, longitud) extrae varios caracteres del final (la parte derecha) de la cadena: SELECT RIGHT('Hola',2); -> la
- MID(cadena, posición, longitud), SUBSTR(cadena, posición, longitud) o SUBSTRING(cadena, posición, longitud) extrae varios caracteres de cualquier posición de una cadena, tantos como se indique en "longitud": SELECT SUBSTRING('Hola',2,2); -> ol (Nota: a partir MySQL 5 se permite un valor negativo en la posición, y entonces se comienza a contar desde la derecha -el final de la cadena-)
- CONCAT une (concatena) varias cadenas para formar una nueva: SELECT CONCAT('Ho', 'la'); -> Hola
- CONCAT_WS une (concatena) varias cadenas para formar una nueva, usando un separador que se indique (With Separator): SELECT CONCAT_WS('-','Ho','la','Que','tal'); -> Ho-la-Que-tal
- LTRIM devuelve la cadena sin los espacios en blanco que pudiera contener al principio (en su parte izquierda): SELECT LTRIM(' Hola'); -> Hola
- RTRIM devuelve la cadena sin los espacios en blanco que pudiera contener al final (en su parte derecha): SELECT RTRIM('Hola '); -> Hola
- TRIM devuelve la cadena sin los espacios en blanco que pudiera contener al principio ni al final: SELECT TRIM(' Hola '); -> Hola (Nota: realmente, TRIM puede eliminar cualquier prefijo, no sólo espacios; mira el manual de MySQL para más detalles)

Funciones de conversión de base numérica

- BIN convierte un número decimal a binario: SELECT BIN(10); -> 1010
- HEX convierte un número decimal a hexadecimal: SELECT HEX(10); -> 'A' (Nota: HEX también puede recibir una cadena, y entonces mostrará el código ASCII en hexadecimal de sus caracteres: SELECT HEX('Hola'); -> '486F6C61')
- OCT convierte un número decimal a octal: SELECT OCT(10); -> 12
- CONV(número,baseInicial,baseFinal) convierte de cualquier base a cualquier base: SELECT

CONV('F3',16,2); -> 11110011

- UNHEX convierte una serie de números hexadecimales a una cadena ASCII, al contrario de lo que hace HEX: SELECT UNHEX('486F6C61'); -> 'Hola')

Otras funciones de modificación de la cadena

- INSERT(cadena,posición,longitud,nuevaCadena) inserta en la cadena otra cadena: SELECT INSERT('Hola', 2, 2, 'ADIOS'); -> HADIOSa
- REPLACE(cadena,de,a) devuelve la cadena pero cambiando ciertas secuencias de caracteres por otras: SELECT REPLACE('Hola', 'l', 'LLL'); -> HoLLLa
- REPEAT(cadena,numero) devuelve la cadena repetida varias veces: SELECT REPEAT(' Hola',3); -> HolaHolaHola
- REVERSE(cadena) devuelve la cadena "del revés": SELECT REVERSE('Hola'); -> aloH
- SPACE(longitud) devuelve una cadena formada por varios espacios en blanco: SELECT SPACE(3); -> " "

Funciones de información sobre la cadena

- CHAR_LENGTH o CHARACTER_LENGTH devuelve la longitud de la cadena en caracteres
- LENGTH devuelve la longitud de la cadena en bytes
- BIT_LENGTH devuelve la longitud de la cadena en bits
- INSTR(cadena,subcadena) o LOCATE(subcadena,cadena,posInicial) devuelve la posición de una subcadena dentro de la cadena: SELECT INSTR('Hola','ol'); -> 2

(Más detalles en el apartado 12.3 del manual de referencia MySQL 5.0)

10. Joins

Sabemos enlazar varias tablas para mostrar datos que estén relacionados. Por ejemplo, podríamos mostrar nombres de deportistas, junto con los nombres de los deportes que practican. Pero todavía hay un detalle que se nos escapa: ¿cómo hacemos si queremos mostrar todos los deportes que hay en nuestra base de datos, incluso aunque no haya deportistas que los practiquen?

Vamos a crear una base de datos sencilla para ver un ejemplo de cual es este problema y de cómo solucionarlo.

Nuestra base de datos se llamará "ejemploJoins":

```
CREATE DATABASE ejemploJoins;
USE ejemploJoins;
```

En ella vamos a crear una primera tabla en la que guardaremos "capacidades" de personas (cosas que saben hacer):

```
CREATE TABLE persona(
  codigo varchar(4),
  nombre varchar(20),
```

```
  codcapac varchar(4),
  PRIMARY KEY(codigo)
);
```

También crearemos una segunda tabla con datos básicos de personas:

```
CREATE TABLE capacidad(
  codigo varchar(4),
  nombre varchar(20),
  PRIMARY KEY(codigo)
);
```

Vamos a introducir datos de ejemplo:

```
INSERT INTO capacidad VALUES
  ('c','Progr.C'),
  ('pas','Progr.Pascal'),
  ('j','Progr.Java'),
  ('sql','Bases datos SQL');

INSERT INTO persona VALUES
  ('ju','Juan','c'),
  ('ja','Javier','pas'),
  ('jo','Jose','perl'),
  ('je','Jesus','html');
```

Antes de seguir, comprobamos que todo está bien:

```
SELECT * FROM capacidad;
```

```
+--------+-----------------+
| codigo | nombre          |
+--------+-----------------+
| c      | Progr.C         |
| j      | Progr.Java      |
| pas    | Progr.Pascal    |
| sql    | Bases datos SQL |
+--------+-----------------+
```

```
SELECT * FROM persona;
```

```
+--------+--------+----------+
| codigo | nombre | codcapac |
+--------+--------+----------+
| ja     | Javier | pas      |
| je     | Jesus  | html     |
| jo     | Jose   | perl     |
| ju     | Juan   | c        |
+--------+--------+----------+
```

Como se puede observar, hay dos capacidades en nuestra base de datos para las que no conocemos a ninguna persona; de igual modo, existen dos personas que tienen capacidades sobre las que no tenemos ningún detalle.

Por eso, si mostramos las personas con sus capacidades de la forma que sabemos, sólo aparecerán las parejas de persona y capacidad para las que todo está claro (existe persona y existe capacidad), es decir:

```
SELECT * FROM capacidad, persona
WHERE persona.codcapac = capacidad.codigo;
```

```
+--------+-------------+--------+--------+----------+
| codigo | nombre      | codigo | nombre | codcapac |
+--------+-------------+--------+--------+----------+
| c      | Progr.C     | ju     | Juan   | c        |
| pas    | Progr.Pascal | ja     | Javier | pas      |
+--------+-------------+--------+--------+----------+
```

Podemos resumir un poco esta consulta, para mostrar sólo los nombres, que son los datos que más nos interesan:

```
SELECT persona.nombre, capacidad.nombre
FROM persona, capacidad
WHERE persona.codcapac = capacidad.codigo;
```

```
+--------+-------------+
| nombre | nombre      |
+--------+-------------+
| Juan   | Progr.C     |
| Javier | Progr.Pascal |
+--------+-------------+
```

Hay que recordar que la orden "where" es obligatoria: si no indicamos esa condición, se mostraría el "producto cartesiano" de las dos tablas: todos los parejas (persona, capacidad), aunque no estén relacionados en nuestra base de datos:

```
SELECT persona.nombre, capacidad.nombre
FROM persona, capacidad;
```

```
+--------+----------------+
| nombre | nombre         |
+--------+----------------+
| Javier | Progr.C        |
| Jesus  | Progr.C        |
| Jose   | Progr.C        |
| Juan   | Progr.C        |
| Javier | Progr.Java     |
| Jesus  | Progr.Java     |
| Jose   | Progr.Java     |
| Juan   | Progr.Java     |
| Javier | Progr.Pascal   |
| Jesus  | Progr.Pascal   |
| Jose   | Progr.Pascal   |
| Juan   | Progr.Pascal   |
| Javier | Bases datos SQL |
| Jesus  | Bases datos SQL |
| Jose   | Bases datos SQL |
| Juan   | Bases datos SQL |
+--------+----------------+
```

Pues bien, con órdenes "join" podemos afinar cómo queremos enlazar (en inglés, "join", unir) las tablas. Por ejemplo, si queremos ver todas las personas y todas las capacidades, aunque no estén relacionadas (algo que no suele tener sentido en la práctica), como en el ejemplo anterior, lo podríamos hacer con un "cross join" (unir de forma cruzada):

```
SELECT persona.nombre, capacidad.nombre
FROM persona CROSS JOIN capacidad;
```

```
+--------+----------------+
| nombre | nombre         |
+--------+----------------+
| Javier | Progr.C        |
| Jesus  | Progr.C        |
```

```
| Jose   | Progr.C         |
| Juan   | Progr.C         |
| Javier | Progr.Java      |
| Jesus  | Progr.Java      |
| Jose   | Progr.Java      |
| Juan   | Progr.Java      |
| Javier | Progr.Pascal    |
| Jesus  | Progr.Pascal    |
| Jose   | Progr.Pascal    |
| Juan   | Progr.Pascal    |
| Javier | Bases datos SQL |
| Jesus  | Bases datos SQL |
| Jose   | Bases datos SQL |
| Juan   | Bases datos SQL |
+--------+-----------------+
```

Si sólo queremos ver los datos que coinciden en ambas tablas, lo que antes conseguíamos comparando los códigos con un "where", también podemos usar un "inner join" (unión interior; se puede abreviar simplemente "join"):

```
SELECT persona.nombre, capacidad.nombre
FROM persona INNER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

```
+--------+-------------+
| nombre | nombre      |
+--------+-------------+
| Juan   | Progr.C     |
| Javier | Progr.Pascal |
+--------+-------------+
```

Pero aquí llega la novedad: si queremos ver todas las personas y sus capacidades, incluso para aquellas personas cuya capacidad no está detallada en la otra tabla, usaríamos un "left join" (unión por la izquierda, también se puede escribir "left outer join", unión exterior por la izquierda, para dejar claro que se van a incluir datos que están sólo en una de las dos tablas):

```
SELECT persona.nombre, capacidad.nombre
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

```
+--------+-------------+
| nombre | nombre      |
+--------+-------------+
| Javier | Progr.Pascal |
| Jesus  | NULL        |
| Jose   | NULL        |
| Juan   | Progr.C     |
+--------+-------------+
```

De igual modo, si queremos ver todas las capacidades, incluso aquellas para las que no hay detalles sobre personas, podemos escribir el orden de las tablas al revés en la consulta anterior, o bien usar "right join" (o "right outer join"):

```
SELECT persona.nombre, capacidad.nombre
FROM persona RIGHT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

```
+--------+----------------+
| nombre | nombre         |
+--------+----------------+
| Juan   | Progr.C        |
| NULL   | Progr.Java     |
```

```
| Javier | Progr.Pascal    |
| NULL   | Bases datos SQL |
+--------+-----------------+
```

Otros gestores de bases de datos permiten combinar el "right join" y el "left join" en una única consulta, usando "full outer join", algo que no permite MySQL en su versión actual.

(Más detalles en el apartado 13.2.7.1 del manual de referencia MySQL 5.0)

11. Union, vistas

En el apartado anterior comentábamos que la versión actual de MySQL no permite usar "full outer join" para mostrar todos los datos que hay en dos tablas enlazadas, aunque alguno de esos datos no tenga equivalencia en la otra tabla.

También decíamos que se podría imitar haciendo a la vez un "right join" y un "left join".

En general, tenemos la posibilidad de unir dos consultas en una usando "union", así:

```
SELECT persona.nombre, capacidad.nombre
FROM persona RIGHT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo
union
SELECT persona.nombre, capacidad.nombre
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

```
+--------+-----------------+
| nombre | nombre          |
+--------+-----------------+
| Juan   | Progr.C         |
| NULL   | Progr.Java      |
| Javier | Progr.Pascal    |
| NULL   | Bases datos SQL |
| Jesus  | NULL            |
| Jose   | NULL            |
+--------+-----------------+
```

Nota: en algunos gestores de bases de datos, podemos no sólo crear "uniones" entre dos tablas, sino también realizar otras operaciones habituales entre conjuntos, como calcular su intersección ("intersection") o ver qué elementos hay en la primera pero no en la segunda (diferencia, "difference"). Estas posibilidades no están disponibles en la versión actual de MySQL.

Por otra parte, podemos crear "vistas", que nos permitan definir la cantidad de información a la que queremos que ciertos usuarios tengan acceso:

```
CREATE VIEW personasycapac AS
SELECT persona.nombre nompers, capacidad.nombre nomcapac
FROM persona LEFT OUTER JOIN capacidad
ON persona.codcapac = capacidad.codigo;
```

Y esta "vista" se utiliza igual que si fuera una tabla:

```
SELECT * FROM personasycapac;
```

```
+---------+-------------+
| nompers | nomcapac    |
```

```
+---------+--------------+
| Javier  | Progr.Pascal |
| Jesus   | NULL         |
| Jose    | NULL         |
| Juan    | Progr.C      |
+---------+--------------+
```

Cuando una vista deje de sernos útil, podemos eliminarla con "drop view".

(Las vistas están disponibles en MySQL desde la versión 5.0. Más detalles en el apartado 21 del manual de referencia MySQL 5.0)

12. Triggers

En MySQL (a partir de la versión 5.0.2) se permite utilizar "disparadores" (triggers), que son una serie de pasos que se pondrán en marcha cuando ocurra un cierto evento en una tabla.

Los eventos pueden ser un INSERT, un UPDATE o un DELETE de datos de la tabla, y podemos detallar si queremos que los pasos se den antes (BEFORE) del evento o después (AFTER) del evento.

Como ejemplo habitual, podríamos hacer un BEFORE INSERT para comprobar que los datos son válidos antes de guardarlos realmente en la tabla.

Pero vamos a empezar probar con un ejemplo, que aunque sea menos útil, será más fácil de aplicar.

Vamos a crear una base de datos sencilla, con sólo dos tablas. En una tabla guardaremos datos de personas, y en la otra anotaremos cuando se ha introducido cada dato. La estructura básica sería ésta:

```
CREATE DATABASE ejemplotriggers;
USE ejemplotriggers;

CREATE TABLE persona (
 codigo varchar(10),
 nombre varchar(50),
 edad decimal(3),
 PRIMARY KEY (`codigo`)
);

CREATE TABLE nuevosDatos (
 codigo varchar(10),
 cuando date,
 tipo char(1)
);
```

Para que se añada un dato en la segunda tabla cada vez que insertemos en la primera, creamos un TRIGGER que saltará con un AFTER INSERT. Para indicar los pasos que debe hacer, se usa la expresión "FOR EACH ROW" (para cada fila), así:

```
CREATE TRIGGER modificacion
  AFTER INSERT ON persona
FOR EACH ROW
  INSERT INTO nuevosDatos
  VALUES (NEW.codigo, CURRENT_DATE, 'i');
```

(Los datos que introduciremos serán: el código de la persona, la fecha actual y una letra "i" para indicar que el cambio ha sido la "inserción" de un dato nuevo).

Si ahora introducimos un dato en la tabla personas:

```
INSERT INTO persona
VALUES ('1','Juan',20);
```

La tabla de "nuevosDatos" habrá cambiado:

```
SELECT * FROM nuevosDatos;
```

```
+--------+------------+------+
| codigo | cuando     | tipo |
+--------+------------+------+
| 1      | 2007-12-05 | i    |
+--------+------------+------+
```

(Nota 1: Si en vez de monitorizar los INSERT, queremos controlar los UPDATE, el valor actual del nombre es "NEW.nombre", pero también podemos saber el valor anterior con "OLD.nombre", de modo que podríamos almacenar en una tabla todos los detalles sobre el cambio que ha hecho el usuario).

(Nota 2: Si no queremos guardar sólo la fecha actual, sino la fecha y la hora, el campo debería ser de tipo DATETIME, y sabríamos el instante actual con "NOW()"):

```
CREATE TRIGGER modificacion
  AFTER INSERT ON persona
FOR EACH ROW
  INSERT INTO nuevosDatos
  VALUES (NEW.codigo, NOW(), 'i');
```

Si queremos indicar que se deben dar secuencias de pasos más largas, deberemos tener en cuenta dos cosas: cuando sean varias órdenes, deberán encerrarse entre BEGIN y END; además, como cada una de ellas terminará en punto y coma, deberemos cambiar momentáneamente el "delimitador" (DELIMITER) de MySQL, para que no piense que hemos terminado en cuanto aparezca el primer punto y coma:

```
DELIMITER |

CREATE TRIGGER validacionPersona BEFORE INSERT ON persona
FOR EACH ROW BEGIN
  SET NEW.codigo = UPPER(NEW.codigo);
  SET NEW.edad = IF(NEW.edad = 0, NULL, NEW.edad);
END;
|
DELIMITER ;
```

(Nota 3: Ese nuevo delimitador puede ser casi cualquiera, siempre y cuando no se algo que aparezca en una orden habitual. Hay quien usa |, quien prefiere ||, quien usa //, etc.)

En este ejemplo, usamos SET para cambiar el valor de un campo. En concreto, antes de guardar cada dato, convertimos su código a mayúsculas (usando la función UPPER, que ya conocíamos), y guardamos NULL en vez de la edad si la edad tiene un valor incorrecto (0, por ejemplo), para lo que usamos la función IF, que aún no conocíamos. Esta función recibe tres parámetros: la condición a comprobar, el valor que se debe devolver si se cumple la condición, y el valor que se debe devolver cuando no se cumpla la condición.

Si añadimos un dato que tenga un código en minúsculas y una edad 0, y pedimos que se nos muestre el resultado, veremos ésto:

```
INSERT INTO persona
VALUES ('p','Pedro',0)
```

```
+--------+---------+------+
| codigo | nombre  | edad |
+--------+---------+------+
| 1      | Juan    |   20 |
| P      | Pedro   | NULL |
+--------+---------+------+
```

Cuando un TRIGGER deje de sernos útil, podemos eliminarlo con DROP TRIGGER.

(Más detalles sobre TRIGGERS en el apartado 20 del manual de referencia de MySQL 5.0; más detalles sobre IF y otras funciones de control de flujo (CASE, IFNULL, etc) en el apartado 12.2 del manual de referencia de MySQL 5.0)

Ej.1. Ejercicio resuelto con una tabla

Vamos a aplicar buena parte de lo que conocemos para hacer un ejercicio de repaso que haga distintas manipulaciones a una única tabla. Será una tabla que contenga datos de productos: código, nombre, precio y fecha de alta, para que podamos trabajar con datos de texto, numéricos y de tipo fecha.

Los pasos que realizaremos (por si alguien se atreve a intentarlo antes de ver la solución) serán:

- Crear la base de datos
- Comenzar a usarla
- Introducir 3 datos de ejemplo
- Mostrar todos los datos
- Mostrar los datos que tienen un cierto nombre
- Mostrar los datos que comienzan por una cierta inicial
- Ver sólo el nombre y el precio de los que cumplen una condición (precio > 22)
- Ver el precio medio de aquellos cuyo nombre comienza con "Silla"
- Modificar la estructura de la tabla para añadir un nuevo campo: "categoría"
- Dar el valor "utensilio" a la categoría de todos los productos existentes
- Modificar los productos que comienza por la palabra "Silla", para que su categoría sea "silla"
- Ver la lista categorías (sin que aparezcan datos duplicados)
- Ver la cantidad de productos que tenemos en cada categoría

Damos por sentado que MySQL está instalado. El primer paso es crear la base de datos:

```
CREATE DATABASE productos1;
```

Y comenzar a usarla:

```
USE productos1;
```

Para crear la tabla haríamos:

```
CREATE TABLE productos (
  codigo varchar(3),
  nombre varchar(30),
  precio decimal(6,2),
  fechaalta date,
  PRIMARY KEY (codigo)
);
```

Para introducir varios datos de ejemplo:

```
INSERT INTO productos VALUES ('a01','Afilador', 2.50, '2007-11-02');
INSERT INTO productos VALUES ('s01','Silla mod. ZAZ', 20, '2007-11-03');
INSERT INTO productos VALUES ('s02','Silla mod. XAX', 25, '2007-11-03');
```

Podemos ver todos los datos para comprobar que son correctos:

```
SELECT * FROM productos;
```

y deberíamos obtener

```
+--------+----------------+--------+------------+
| codigo | nombre         | precio | fechaalta  |
+--------+----------------+--------+------------+
| a01    | Afilador       |   2.50 | 2007-11-02 |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 |
+--------+----------------+--------+------------+
```

Para ver qué productos se llaman "Afilador":

```
SELECT * FROM productos WHERE nombre='Afilador';
```

```
+--------+----------+--------+------------+
| codigo | nombre   | precio | fechaalta  |
+--------+----------+--------+------------+
| a01    | Afilador |   2.50 | 2007-11-02 |
+--------+----------+--------+------------+
```

Si queremos saber cuales comienzan por S:

```
SELECT * FROM productos WHERE nombre LIKE 'S%';
```

```
+--------+----------------+--------+------------+
| codigo | nombre         | precio | fechaalta  |
+--------+----------------+--------+------------+
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 |
+--------+----------------+--------+------------+
```

Si queremos ver cuales tienen un precio superior a 22, y además no deseamos ver todos los

campos, sino sólo el nombre y el precio:

```
SELECT nombre, precio FROM productos WHERE precio > 22;
```

```
+----------------+--------+
| nombre         | precio |
+----------------+--------+
| Silla mod. XAX |  25.00 |
+----------------+--------+
```

Precio medio de las sillas:

```
SELECT avg(precio) FROM productos WHERE LEFT(nombre,5) = 'Silla';
```

```
+-------------+
| avg(precio) |
+-------------+
|   22.500000 |
+-------------+
```

Esto de mirar las primeras letras para saber si es una silla o no... quizá no sea la mejor opción. Parece más razonable añadir un nuevo dato: la "categoría". Vamos a modificar la estructura de la tabla para hacerlo:

```
ALTER TABLE productos ADD categoria varchar(10);
```

Comprobamos qué ha ocurrido con un "select" que muestre todos los datos:

```
SELECT * FROM productos;
```

```
+--------+----------------+--------+------------+-----------+
| codigo | nombre         | precio | fechaalta  | categoria |
+--------+----------------+--------+------------+-----------+
| a01    | Afilador       |   2.50 | 2007-11-02 | NULL      |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 | NULL      |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 | NULL      |
+--------+----------------+--------+------------+-----------+
```

Ahora mismo, todas las categorías tienen el valor NULL, y eso no es muy útil. Vamos a dar el valor "utensilio" a la categoría de todos los productos existentes

```
UPDATE productos SET categoria='utensilio';
```

Y ya que estamos, modificaremos los productos que comienza por la palabra "Silla", para que su categoría sea "silla"

```
UPDATE productos SET categoria='silla' WHERE LEFT(nombre,5) = 'Silla';
```

```
+--------+----------------+--------+------------+-----------+
| codigo | nombre         | precio | fechaalta  | categoria |
+--------+----------------+--------+------------+-----------+
| a01    | Afilador       |   2.50 | 2007-11-02 | utensilio |
| s01    | Silla mod. ZAZ |  20.00 | 2007-11-03 | silla     |
| s02    | Silla mod. XAX |  25.00 | 2007-11-03 | silla     |
+--------+----------------+--------+------------+-----------+
```

Para ver la lista categorías (sin que aparezcan datos duplicados), deberemos usar la palabra "distinct"

```
SELECT DISTINCT categoria FROM productos;
```

```
+-----------+
```

```
| categoria |
+-----------+
| utensilio |
| silla     |
+-----------+
```

Finalmente, para ver la cantidad de productos que tenemos en cada categoría, deberemos usar "count" y agrupar los datos con "group by", así:

```
SELECT categoria, count(*) FROM productos GROUP BY categoria;
```

```
+-----------+----------+
| categoria | count(*) |
+-----------+----------+
| silla     |        2 |
| utensilio |        1 |
+-----------+----------+
```

Ej.2. Ejercicio propuesto con una tabla

Queremos crear una base de datos para almacenar información sobre PDAs. En un primer acercamiento, usaremos una única tabla llamada PDA, que tendrá como campos:

- Código
- Nombre
- Sistema Operativo
- Memoria (mb)
- Bluetooth (s/n)

1- Crear la tabla.

2- Introducir en ella los datos:

- ptx, Palm Tungsten TX, PalmOS, 128, s
- p22, Palm Zire 22, PalmOS, 16, n
- i3870, Compaq Ipaq 3870, Windows Pocket PC 2002, 64, s

Realizar las consultas

3- Equipos con mas de 64 mb de memoria.

4- Equipos cuyo sistema operativo no sea "PalmOS".

5- Equipos cuyo sistema operativo contenga la palabra "Windows".

6- Lista de sistemas operativos (sin duplicados)

7- Nombre y código del equipo que más memoria tiene.

8- Nombre y marca (supondremos que la marca es la primera palabra del nombre, hasta el primer espacio) de cada equipo, ordenado por marca y a continuación por nombre.

9- Equipos con menos memoria que la media.

10- Cantidad de equipos con cada sistema operativo.

11- Sistemas operativos para los que tengamos 2 o más equipos en la base de datos.

12- Añadir a la tabla PDA un campo "precio", con valor NULL por defecto.

13- Modificar el dato del equipo con código "p22", para indicar que su precio es 119,50. Listar los equipos cuyo precio no conocemos.

Ej.3. Ejercicio propuesto con dos tablas

1- Crear una base de datos llamada "deportes", y en ella dos tablas: jugador y equipo. Del jugador se desea almacenar: codigo (txt 12), nombre, apellido 1, apellido 2, demarcacion (ej: delantero). De cada equipo: codigo (txt 8), nombre, deporte (ej: baloncesto). Cada equipo estará formada por varios jugadores, y supondremos que cada jugador sólo puede formar parte de un equipo.

2- Introducir los datos:

En equipos:

- rcm, Real Campello, baloncesto
- can, Canoa, natacion
- ssj, Sporting de San Juan, futbol

En jugadores:

- rml, Raúl, Martínez, López, pivot (juega en el Real Campello)
- rl, Raúl, López, , saltador (del Canoa)
- jl, Jordi, López, , nadador crawl (del Canoa)
- rol, Roberto, Linares, , base (juega en el Real Campello)

3- Crear una consulta que muestre: nombre de deportista, primer apellido, demarcación, nombre de equipo (para todos los jugadores de la base de datos).

4- Crear una consulta que muestre el nombre de los equipos para los que no sabemos los jugadores.

5- Crear una consulta que muestre nombre y apellidos de los jugadores cuyo primer o segundo apellido es "López".

6- Crear una consulta que muestre nombre y apellidos de los nadadores.

7- Crear una consulta que muestre la cantidad de jugadores que hay en cada equipo.

8- Crear una consulta que muestre la cantidad de jugadores que hay en cada deporte.

9- Crear una consulta que muestre el equipo que más jugadores tiene.

10- Añadir a la tabla de jugadores un campo en el que poder almacenar la antigüedad (en años), que tenga como valor por defecto NULL, y modificar la ficha de "Roberto Linares" para indicar que su antigüedad es de 4 años.

Revisiones de este texto hasta la fecha:

- 0.16, de 16-Ene-08. Añadido un tercer ejercicio propuesto, con datos de dos tablas. Corregida alguna errata de poca importancia (como "cumplor" en vez de "cumplir").
- 0.15, de 17-Dic-07. Añadido un segundo ejercicio propuesto (sin solución), con datos de una única tabla.
- 0.14, de 05-Dic-07. Apartado 12, "triggers". Ampliado el apartado 11 para mencionar otras operaciones entre conjuntos, así como la orden "drop view".
- 0.13, de 03-Dic-07. Apartado 11, "union" y vistas.
- 0.12, de 28-Nov-07. Apartado 10, "joins".
- 0.11, de 07-Nov-07. Ampliado el apartado 2 para mencionar "distinct." Añadido un primer ejercicio resuelto con datos tomados de una única tabla.
- 0.10, de 07-Nov-07. Apartado 9, funciones de cadena.
- 0.09, de 04-Nov-07. Apartado 8, subconsultas.
- 0.08, de 03-Nov-07. Apartado 7, valores agrupados.
- 0.07, de 02-Nov-07. Apartado 6, valores nulos.
- 0.06, de 01-Nov-07. Apartado 5, operaciones matemáticas y funciones de agregación.
- 0.05, de 20-Oct-07. Apartado 4, modificación de datos; añadida la orden "truncate" al apartado 3.
- 0.04, de 15-Oct-07. Apartado 3, borrado de datos.
- 0.03, de 14-Oct-07. Apartado 2, consultas básicas con dos tablas.
- 0.02, de 13-Oct-07. Apartado 1, consultas básicas con una tabla.
- 0.01, de 10-Oct-07. Apartado 0, sobre la instalación de MySQL.