CURSO DE MYSQL 5.5

POR IVÁN RODRÍGUEZ RUIZ

ÍNDICE

1 INTRODUCCIÓN

1.1 ¿Que es MySQL?

MySQL es un **sistema de gestión de bases de datos relacional**, **multihilo** y **multiusuario** con más de 11 millones de instalaciones. Es el SGBD mas usado del mundo. MySQL AB —desde enero de 2008 una subsidiaria de Sun Microsystems y ésta a su vez de Oracle Corporation desde abril de 2009— desarrolla MySQL como software libre en un esquema de licenciamiento dual.

Por un lado se ofrece bajo la GNU GPL para cualquier uso compatible con esta licencia, pero para aquellas empresas que quieran incorporarlo en productos privativos deben comprar a la empresa una licencia específica que les permita este uso. Está desarrollado en su mayor parte en ANSI C.

Al contrario de proyectos como Apache, donde el software es desarrollado por una comunidad pública y los derechos de autor del código están en poder del autor individual, MySQL es patrocinado por una empresa privada, que posee el copyright de la mayor parte del código.

Esto es lo que posibilita el esquema de licenciamiento anteriormente mencionado. Además de la venta de licencias privativas, la compañía ofrece soporte y servicios. Para sus operaciones contratan trabajadores alrededor del mundo que colaboran vía Internet. MySQL AB fue fundado por **David Axmark**, **Allan Larsson** y **Michael Widenius**.

1.1.1 Conceptos: Relacional, Multihilo y Multiusuario.

Como hemos visto, MySQL es un **Sistema de gestión de bases de datos (SGBD) Relacional, Multihilo y MultiUsuario.** Veamos que siginifica cada uno:
o Es Relacional: Las Entidades que lo componen (tablas) se relacionan entre si.
o Es MultiHilo, es decir, permite realizar varias acciones a la vez. Esto es fundamental, hay usuarios que usan MySQL que reciben miles de "llamadas" por segundo.
Varios ejemplos: **Facebook, Wikipedia, Yahoo**, etc.
o Es MultiUsuario: puede tener varios usuarios diferentes en el sistema al mismo tiempo.

1.1.2 Lenguajes de Programación

Existen varias interfaces de programación de aplicaciones que permiten, a aplicaciones escritas en diversos lenguajes de programación, acceder a las bases de datos MySQL, incluyendo **C, C++, C#, Pascal, Delphi** (via dbExpress), Eiffel, Smalltalk, **Java** (con una implementación nativa del driver de Java), Lisp, **Perl, PHP, Python**, Ruby,Gambas, REALbasic (Mac y Linux), (x)Harbour (Eagle1), FreeBASIC, y Tcl; cada uno de estos utiliza una interfaz de programación de aplicaciones específica. También existe una interfaz ODBC, llamado MyODBC que permite a cualquier lenguaje de programación que soporte ODBC comunicarse con las bases de datos MySQL. También se puede acceder desde el sistema SAP, lenguaje ABAP.

1.1.3 Aplicaciones

MySQL es muy utilizado en aplicaciones web, como Drupal o phpBB, en plataformas (Linux/Windows-Apache-MySQL-PHP/Perl/Python), y por herramientas de seguimiento de errores como Bugzilla. Su popularidad como aplicación web está muy ligada a PHP, que a menudo aparece en combinación con MySQL.

MySQL es una base de datos muy rápida en la lectura cuando utiliza el motor no transaccional MyISAM, pero puede provocar problemas de integridad en entornos de alta concurrencia en la modificación. En aplicaciones web hay baja concurrencia en la modificación de datos y en cambio el entorno es intensivo en lectura de datos, lo que hace a MySQL ideal para este tipo de aplicaciones.

1.2 Productos MySQL

MySQL dispone de varios productos, herramientas, controladores y servicios para el aprendizaje. Los trataremos en los siguientes subapartados.

1.2.1 Ediciones MySQL

Ref: http://www.mysql.com/products/
Las ediciones disponibles de MySQL son las siguientes:

- MySQL Standard Edition, para su uso en PYMES y a nivel de desarrollador.
- MySQL Enterprise Edition, para su uso en grandes empresas, permitiendo la Administración de grandes servidores.
- MySQL Cluster Carrier Grade Edition, diseñada para su uso en entornos de telecomunicaciones, tales como los sistemas para la gestión de datos de suscriptores (HLR, HSS) y las plataformas de suministro de servicio.

1.2.2 Herramientas GUI

MySQL hasta hace unos años disponía de varias herramientas con interfaz gráfica (Graphic User Interface, GUI) que han sido reemplazadas. Eran:

- MySQL Administrator
- MySQL Query Browser
- MySQL Migration Toolkit

Hoy, todas están integradas en una sola, llamada MySQL Workbench, que puede encontrarse en esta página: http://dev.mysql.com/downloads/gui-tools/5.0.html
Dicha herramientas permite:

- El Modelado y Diseño de Bases de Datos.
- Desarrollo SQL (el lenguaje estandarizado empleado por MySQL)
- Administración de Bases de Datos.

Además tenemos otra herramienta especial: mysql-proxy.
(Ref: http://wiki.elhacker.net/bases-de-datos/mysql/introduccion/mysql-proxy)

Es una ligera aplicación binaria entre uno o más clientes de MySQL y un servidor. Los clientes se conectan al proxy en lugar de conectar con el servidor. El proxy funciona como un intermediario entre el cliente y el servidor.

En su forma básica, el proxy es sólo una redirección. Se pone un cubo vacío desde el cliente (una consulta), lo lleva al servidor, llena el cubo con los datos, y se lo pasa al cliente.

Si eso fuese todo, el proxy como tal seria una herramienta inútil. Pero, dicha herramienta provee consigo una utilidad importante, el interprete LUA. Usando el lenguaje Lua, puede definir qué hacer con una consulta o un conjunto de resultados antes de retornar la respuesta consultada.

1.2.3 Controladores de MySQL

Los MySQL Connectors son controladores de BBDD que brindan conectividad de los clientes de BBDD en una amplia variedad de lenguajes de programación. MySQL tienes estos:

- MySQL C API: La librería nativa (libmysql) que puede ser usada por otros lenguajes
- MySQL Connector / ODBC: Para poder conectar un servidor MySQL con aplicaciones de las plataformas Microsoft Windows, por ejemplo Access.
- MySQL Connector/J: que permite la conexión con aplicaciones Java.
- Etc.

1.3 Certificación MySQL

Es un programa de capacitación profesional para Desarrolladores (Developers) y Administradores de Bases de Datos (Database Administrators, DBA), dividido en varios niveles que son:

- Certificado MySQL Asociado (CMA)
- **Certificado MySQL para Desarrolladores (CMDEV)**
- Certificado MySQL para Administradores (CMDBA)
- Certificado MySQL para Administradores en Cluster (CMCDBA)

Mas información en: http://www.mysql.com/certification/

1.4 Recursos Adicionales

En este apartado pondré las direcciones mas importantes para el aprendizaje sobre MySQL.

- http://www.mysql.com/
- http://dev.mysql.com
- http://www.databasejournal.com
- http://es.planet.mysql.com/

1.5 Instalación de MySQL

1.5.1 Consideraciones previas

MySQL se instala junto con Apache, PHP y otras aplicaciones mediante XAMPP.
Es fundamental que usemos el Cliente de MySQL que ya veremos.

Veamos algunas notas sobre sistemas operativos:

- En Linux es fundamental tener el sistema actualizado (sobre todo el Núcleo o Kernel).
- En Windows es recomendable usar Variables de entorno.
- En Mac es recomendable usar GNUTAR para descomprimir archivos.

1.5.2 Instalación en Windows

Como en las demás versiones, podemos usar el paquete propio de MySQL o uno integrado que incluye Apache, PHP, etc. Personalmente prefiero este último.

Nos vamos a la página de XAMPP, http://www.apachefriends.org/es/xampp.html
Nos descargamos la versión de Windows y lo instalamos.

El cliente se usa dentro de una Terminal de MS-DOS:
Inicio > Todos los Programas > Accesorios > Símbolo del Sistema.

Sería recomendable establecer una Variable de Entorno.
¿Cómo? Pulsamos con el botón derecho **Mi Pc > Propiedades > Ficha Avanzadas.**
Luego le damos al botón Variables de Entorno y ponemos en PATH la siguiente dirección:
C:\xampp\mysql\bin (si es que tenemos XAMPP instalado en la raiz).

Para arrancar el cliente pondríamos en el Terminal:

```
mysql -u root
```

La alternativa (que es la que me ha tocado a mí por no tener privilegios en el ordenador que uso) es esta:

```
cd C:\xampp\mysql\bin
mysql -u root     Y aparece el símbolo de mySQL.
mysql >
```

1.5.3 Instalación en Linux

El proceso para la instalación de XAMPP en Linux es el siguiente:

a) Nos vamos a la página de XAMPP, sección Linux http://www.apachefriends.org/en/xampp-linux.html
b) Nos descargamos el paquete completo: `xampp-linux-1.8.1.tar.gz`
c) Abrimos una terminal y nos vamos a la carpeta de descargas.
   En mi caso: `cd /home/ivan-htpc/Descargas`
d) Descomprimimos mediante la terminal el paquete descargado.
   `sudo tar xvfz xampp-linux-1.8.1.tar.gz -C /opt`
e) Y arrancamos el servidor con:
   `sudo /opt/lampp/lampp start`

Otras opciones:

- Para detener el servidor: `sudo /opt/lampp/lampp stop`
- Para reiniciar el servidor: `sudo /opt/lampp/lampp restart`
- Para eliminar XAMPP de nuestro equipo: `sudo rm -rf /opt/lampp`

Configurar las variables de entorno:

a) Volvemos a arrancar la terminal, y tecleamos
   `sudo kate /etc/environment` (si tenemos Kubuntu, como yo; para Xubuntu, **leafpad**)
   `sudo gedit /etc/environment` (si tenemos Ubuntu)
b) Veremos el siguiente PATH:
   `PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games"`
c) Añadimos `:/opt/lampp/var/mysql/` al final del todo, quedando:
   `PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games`**`:/opt/lampp/var/mysql/"`**
   NOTA: Mucho cuidado, no puede haber espacios.
d) Guardamos el archivo y reiniciamos XAMPP.

Configurar la seguridad con XAMPP:

a) Abrimos la consola y escribimos: `sudo /opt/lampp/lampp security`
b) Comenzamos por poner una contraseña a las páginas XAMPP:
   Escribimos y, poniendo luego la contraseña 2 veces
c) Luego el usuario pma (acceso) a PHPMyAdmin
   Escribimos y, poniendo luego la contraseña 2 veces
d) Ahora el usuario root de MySQL
   Escribimos y, poniendo luego la contraseña 2 veces
e) Y por último el usuario nobody para el FTP
   Escribimos y, poniendo luego la contraseña 2 veces

Posibles problemas de conexión:
Si al conectar con MySQL nos da un error, para evitarlo hacemos lo siguiente:

a) Detenemos el servidor: `sudo /opt/lampp/lampp stop`
b) Abrimos la terminal y escribimos:
   `sudo gedit /etc/mysql/my.cnf` (sudo kate... si estamos en Kubuntu)
c) Buscamos la línea que pone `[client]`
d) Dejamos esa sección asi:

```
[client]
port        = 3306
#socket         = /var/run/mysqld/mysqld.sock
socket          =/opt/lampp/var/mysql/mysql.sock
```

a) Reiniciamos el servidor: `sudo /opt/lampp/lampp start`

1.5.4 Instalación de la base de datos world

MySQL incluye una base de datos excelente y muy completa para realizar prácticas para todos aquellos que, precisamente, quieren aprender a usarla. Se llama world, e incluye datos geográficos y estadísticos de los países del mundo.
Mas información en este enlace: http://dev.mysql.com/doc/world-setup/en/index.html

Los pasos para instalar la base de datos worls serán:

1. Nos vamos a la página: http://dev.mysql.com/doc/index-other.html
2. En la sección Example Databases, nos descargamos la versión zip de: world database (InnoDB version)

3. Abrimos la terminal y escribimos los siguientes comandos:

- cd /home/ivan-htpc/Descargas (para ir al directorio de Descargas)
- mysql -u root -p
- root (o la contraseña puesta)
- CREATE DATABASE world;
- USE world;
- SOURCE world_innodb.sql

Toca esperar un ratito mientras carga los cientos de registros.

- SHOW TABLES;

Y veremos las 3 tablas para el ejemplo.

- City
- Country
- CountryLanguage

```
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> show tables;
+-----------------+
| Tables_in_world |
+-----------------+
| City            |
| Country         |
| CountryLanguage |
+-----------------+
3 rows in set (0.00 sec)

mysql>
```

2 BASE DE DATOS RELACIONAL

Este módulo está tomado de la Wikipedia, cuyo artículo de referencia ha sido escrito, en parte, por mi mismo (incluido el gráfico que aparece).

2.1 Definición

Una base de datos relacional es una base de datos que cumple con el modelo relacional, el cual es el modelo más utilizado en la actualidad para implementar bases de datos ya planificadas. Permiten establecer interconexiones (relaciones) entre los datos (que están guardados en tablas), y a través de dichas conexiones relacionar los datos de ambas tablas, de ahí proviene su nombre: "Modelo Relacional". Tras ser postuladas sus bases en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California), no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos

2.2 Características

- Una base de datos relacional se compone de varias tablas o relaciones.
- No pueden existir dos tablas con el mismo nombre ni registro.
- Cada tabla es a su vez un conjunto de registros (filas y columnas).
- La relación entre una tabla padre y un hijo se lleva a cabo por medio de las claves primarias y ajenas (o foráneas).
- Las claves primarias son la clave principal de un registro dentro de una tabla y éstas deben cumplir con la integridad de datos.
- Las claves ajenas se colocan en la tabla hija, contienen el mismo valor que la clave primaria del registro padre; por medio de éstas se hacen las relaciones.

2.2.1 Relaciones base y derivadas

En una base de datos relacional, todos los datos se almacenan y se accede a ellos por medio de relaciones. Las relaciones que almacenan datos son llamadas "relaciones base" y su implementación es llamada "tabla". Otras relaciones no almacenan datos, pero son calculadas al aplicar operaciones relacionales. Estas relaciones son llamadas "relaciones derivadas" y su implementación es llamada "vista" o "consulta". Las relaciones derivadas son convenientes ya que expresan información de varias relaciones actuando como si fuera una sola.

2.2.2 Restricciones

Una restricción es una limitación que obliga el cumplimiento de ciertas condiciones en la base de datos. Algunas no son determinadas por los usuarios, sino que son inherentemente definidas por el simple hecho de que la base de datos sea relacional. Algunas otras restricciones las puede definir el usuario. Ej: Usar un campo con valores enteros entre 1 y 10.
Las restricciones proveen un método de implementar reglas en la base de datos. Las restricciones limitan los datos que pueden ser almacenados en las tablas. Usualmente se definen usando expresiones que dan como resultado un valor booleano, indicando si los datos satisfacen la restricción o no. Las restricciones no son parte formal del modelo relacional, pero son incluidas porque juegan el rol de organizar mejor los datos. Las restricciones son muy discutidas junto con los conceptos relacionales.

2.2.3 Dominios

Un dominio describe un conjunto de posibles valores para cierto atributo. Como un dominio restringe los valores del atributo, puede ser considerado como una restricción. Matemáticamente, atribuir un dominio a un atributo significa "todos los valores de este atributo deben de ser elementos del conjunto especificado". Distintos tipos de dominios son: **Enteros, Cadenas de Texto, Fecha, Booleanos**, etc.

2.2.4 Clave Única

Cada tabla puede tener uno o más campos cuyos valores identifican de forma única cada registro de dicha tabla, es decir, no pueden existir dos o más registros diferentes cuyos valores en dichos campos sean idénticos. Este conjunto de campos se llama clave única. Pueden existir varias claves únicas en una determinada tabla, y a cada una de éstas suele llamársele **candidata a clave primaria**.

2.2.5 Clave Primaria

Una clave primaria es una clave única elegida entre todas las candidatas que define unívocamente a todos los demás atributos de la tabla, para especificar los datos que serán relacionados con las demás tablas. La forma de hacer esto es por medio de claves foráneas. Sólo puede existir una clave primaria por tabla y ningún campo de dicha clave puede contener valores NULL o nulos.

2.2.6 Clave Foránea

Una clave foránea es una referencia a una clave en otra tabla, determina la relación existente en dos tablas. Las claves foráneas no necesitan ser claves únicas en la tabla donde están y sí a donde están referenciadas.

Por ejemplo, el código de departamento puede ser una clave foránea en la tabla de empleados. Se permite que haya varios empleados en un mismo departamento, pero habrá uno y sólo un departamento por cada clave distinta de departamento en la tabla de empleados.

2.2.7 Clave Índice

Las claves índice surgen con la necesidad de tener un acceso más rápido a los datos. Los índices pueden ser creados con cualquier combinación de campos de una tabla. Las consultas que filtran registros por medio de estos campos, pueden encontrar los registros de forma no secuencial usando la clave índice. Las bases de datos relacionales incluyen múltiples técnicas de ordenamiento, cada una de ellas es óptima para cierta distribución de datos y tamaño de la relación.

Los índices generalmente no se consideran parte de la base de datos, pues son un detalle agregado. Sin embargo, las claves índices son desarrolladas por el mismo grupo de programadores que las otras partes de la base de datos.

2.2.8 Procedimientos Almacenados

Un procedimiento almacenado es código ejecutable que se asocia y se almacena con la base de datos. Los procedimientos almacenados usualmente recogen y personalizan operaciones comunes, como insertar un registro dentro de una tabla, recopilar información estadística, o encapsular cálculos complejos. Son frecuentemente usados por un API por seguridad o simplicidad. Los procedimientos almacenados no son parte del modelo relacional, pero todas las implementaciones comerciales los incluyen.

2.3 Estructura

La base de datos se organiza en dos marcadas secciones; el esquema y los datos (o instancia). El esquema es la definición de la estructura de la base de datos y principalmente almacena los siguientes datos:

- El nombre de cada tabla
- El nombre de cada columna
- El tipo de dato de cada columna
- La tabla a la que pertenece cada columna

Las bases de datos relacionales pasan por un proceso al que se le conoce como normalización, el resultado de dicho proceso es un esquema que permite que la base de datos sea usada de manera óptima.

Los datos o instancia es el contenido de la base de datos en un momento dado. Es en sí, el contenido de todos los registros.

2.3.1 Normalización

El proceso de normalización de bases de datos consiste en aplicar una serie de reglas a las relaciones obtenidas tras el paso del modelo entidad-relación al modelo relacional.

Las bases de datos relacionales se normalizan para:

- Evitar la redundancia de los datos.
- Evitar problemas de actualización de los datos en las tablas.
- Proteger la integridad de los datos.

En el modelo relacional es frecuente llamar tabla a una relación, aunque para que una tabla sea considerada como una relación tiene que cumplir con algunas restricciones:

- Cada tabla debe tener su nombre único.
- No puede haber dos filas iguales. No se permiten los duplicados.
- Todos los datos en una columna deben ser del mismo tipo.

2.4 Manipulación de la Información

Para manipular la información utilizamos un lenguaje relacional, actualmente se cuenta con dos lenguajes formales el álgebra relacional y el cálculo relacional. El álgebra relacional permite describir la forma de realizar una consulta, en cambio, el cálculo relacional sólo indica lo que se desea devolver.

El lenguaje más común para construir las consultas a bases de datos relacionales es SQL (Structured Query Language), un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

En el modelo relacional los atributos deben estar explícitamente relacionados a un nombre en todas las operaciones, en cambio, el estándar SQL permite usar columnas sin nombre en conjuntos de resultados, como el asterisco taquigráfico (*) como notación de consultas.

Al contrario del modelo relacional, el estándar SQL requiere que las columnas tengan un orden definido, lo cual es fácil de implementar en una computadora, ya que la memoria es lineal.

Es de notar, sin embargo, que en SQL el orden de las columnas y los registros devueltos en cierto conjunto de resultado nunca está garantizado, a no ser que explícitamente sea especificado por el usuario.

2.5 Manejadores de base de datos relacionales

Existe software exclusivamente dedicado a tratar con bases de datos relacionales. Este software se conoce como SGBD (Sistema de Gestión de Base de Datos relacional) o RDBMS (del inglés Relational Database Management System).

Entre los gestores o manejadores actuales más populares encontramos: MySQL, PostgreSQL, Oracle, DB2, INFORMIX, Interbase, FireBird, Sybase y Microsoft SQL Server.

2.6 Ventajas y Desventajas

Ventajas:

- Provee herramientas que garantizan evitar la duplicidad de registros.
- Garantiza la integridad referencial, así, al eliminar un registro elimina todos los registros relacionados dependientes.
- Favorece la normalización por ser más comprensible y aplicable.

Desventajas:

- Presentan deficiencias con datos gráficos, multimedia, CAD y sistemas de información geográfica.
- No se manipulan de forma manejable los bloques de texto como tipo de dato.
- Las bases de datos orientadas a objetos (BDOO) se propusieron con el objetivo de satisfacer las necesidades de las aplicaciones anteriores y así, complementar pero no sustituir a las bases de datos relacionales.

2.7 Diseño de las bases de datos relacionales

El primer paso para crear una base de datos, es planificar el tipo de información que se quiere almacenar en la misma, teniendo en cuenta dos aspectos: la información disponible y la información que necesitamos.

La planificación de la estructura de la base de datos, en particular de las tablas, es vital para la gestión efectiva de la misma. El diseño de la estructura de una tabla consiste en una descripción de cada uno de los campos que componen el registro y los valores o datos que contendrá cada uno de esos campos.

Los campos son los distintos tipos de datos que componen la tabla, por ejemplo: nombre, apellido, domicilio. La definición de un campo requiere: el nombre del campo, el tipo de campo, el ancho del campo, etc.

Los registros constituyen la información que va contenida en los campos de la tabla, por ejemplo: el nombre del paciente, el apellido del paciente y la dirección de este. Generalmente los diferentes tipos de campos que se pueden almacenar son los siguientes:
Texto (caracteres), Numérico (números), Fecha / Hora, Lógico (informaciones lógicas si/no, verdadero/falso, etc.), imágenes.

En resumen, el principal aspecto a tener en cuenta durante el diseño de una tabla es determinar claramente los campos necesarios, definirlos en forma adecuada con un nombre especificando su tipo y su longitud.

2.8 Diagramas

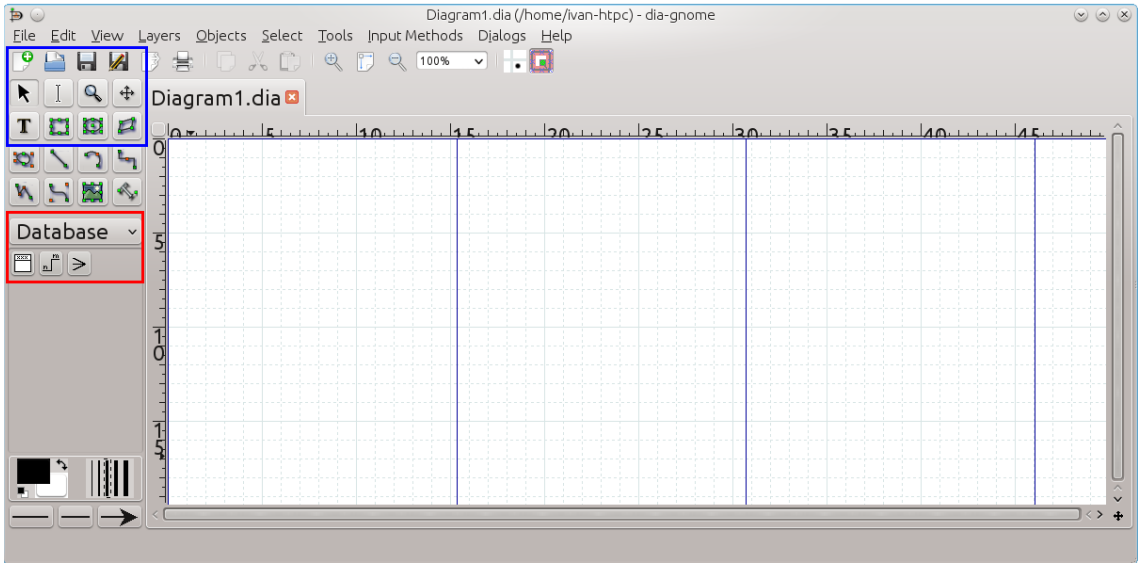
Vamos a emplear una aplicación gratuita para la construcción de Diagramas, DIA.
Su página web es esta: http://dia-installer.de/

Para su instalación, podemos descargarnos el paquete DEB (Debian / Ubuntu) o RMP (Red Hat) en esta dirección: http://dia-installer.de/download/linux.html
Alternativa:

- Via consola: sudo apt-get install dia
- Mediante el Centro de Software (muon en Kubuntu)

2.8.1 Uso de DIA

En primer lugar debemos elegir el tipo de diagrama, en nuestro caso Bases de Datos, lo tenemos puesto en la captura inferior dentro del botón que pone Database.
Al elegirlo, se activan 3 botones: Tabla, Referencia (relación) y atributo compuesto.
Con CTRL+E ampliamos lo seleccionado a la pantalla (si no hay nada, vemos todas las tablas)

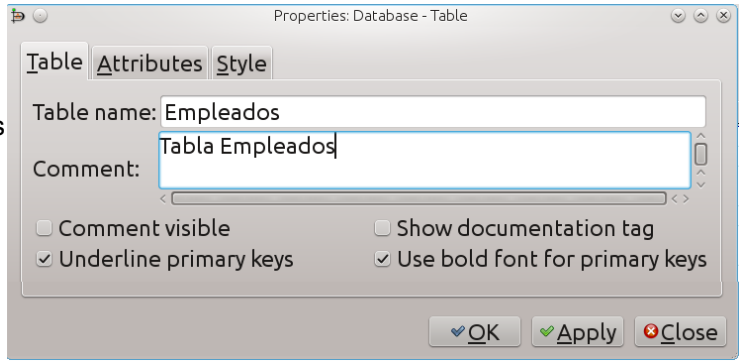


En la zona superior, marcado en azul, tenemos los clásicos botones de Nuevo, Abrir y Guardar, así como algunos muy útiles como Seleccionar objeto (el de la flecha) o Texto (con una T).

El proceso de creación del Diagrama es muy sencillo. En primer lugar pulsamos en el botón Tabla, marcamos un punto en el espacio en blanco y hacemos doble clic. Aparece esto:

En la pestaña Table pondremos el nombre de la tabla.
Recomiendo activar la opción Use bold... que marca las claves primarias en negrita.

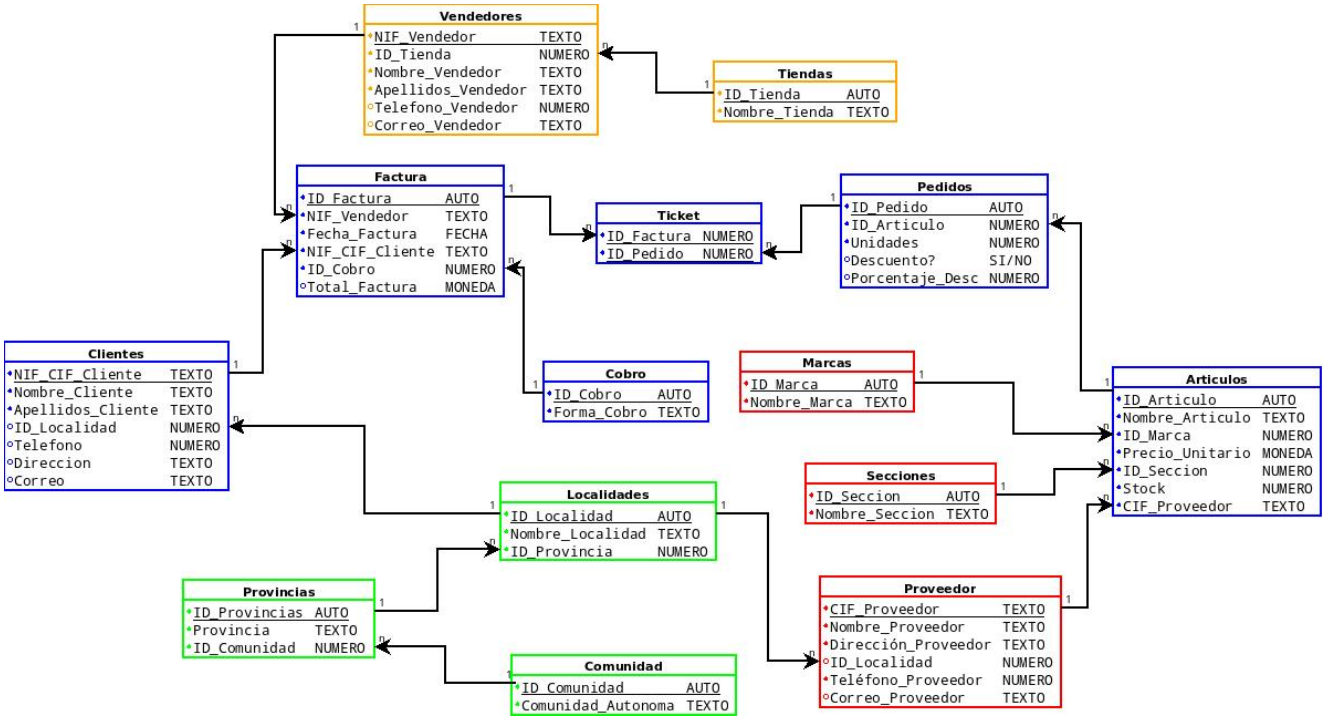
En Attributes ponemos los campos, con sus nombres y Tipos.
En Style le añadimos formato, colores, etc.



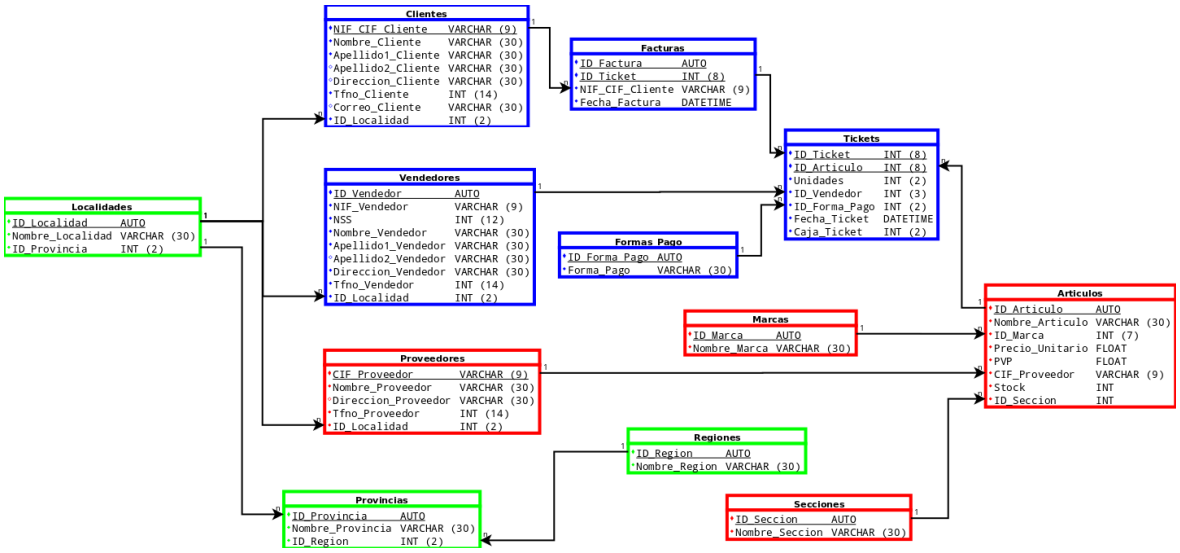
2.8.2 Un ejemplo propio

Imaginemos que nos plantean la relación de una web de Comercio Electrónico. Por ejemplo, algo similar a http://www.elcorteingles.es/ , http://www.worten.es/ o http://www.appinformatica.com/

Tenemos dos posibles diseños:
En primer lugar **SIN Claves Compuestas**:



Luego, como alternativa, **CON Claves Compuestas**:



3 CLIENTES PARA MYSQL

Existen varias aplicaciones y utilidades que vienen con las distribuciones de MySQL, en esta sección veremos como usar cada una de ellas. La mayoría de ellas requieren que utilicéis la línea de comandos de vuestro sistema operativo (command prompt o cmd en la familia Windows o el shell en la familia Unix -linux, MacOS, etc-). En esta sección veremos con más detalle las funciones principales de la interfáz de texto mysql debido a que es la interfáz más usada, pero también veremos como se usa algunas otras aplicaciones.

3.1 Llamadas a los programas cliente

Para conocer cuales son las funciones que soporta cierta aplicación, podemos escribir el nombre de la aplicación seguido por el comando --help, el cual desplegará una pantalla de ayuda indicando que funciones soporta dicha aplicación. Por ejemplo para ver de que funciones disponemos en la interfáz mysql podemos escribir:

- `help;`
- [Alternativa] `mysql --help;`

Para ver que versión de la aplicación se está usando, se puede escribir el nombre de la aplicación seguido por el comando --version:

- `SHOW VARIABLES LIKE "%version%";`
- [Alternativa] `mysql --version`

3.2 Formas de las opciones

En casi todos los casos, las opciones están disponibles en dos formas: una forma larga y otra forma corta. Por ejemplo si deseamos saber que versión del servidor estamos usando podemos hacerlo usando el comando --version como ya lo habíamos visto o también usando el comando -V de la siguiente forma:

- `mysql -V` lo cual nos dará el mismo resultado que `--version` (no funciona en Linux)

Debéis tener cuidado que las opciones, eso si, son de caracteres sensitivos, por lo que -v es diferente a -V o --version es diferente a --Version. Muchas veces las opciones son válidas pero tienen funciones distintas. -V muestra la versión del programa y -v dice a mysql que debe desplegar información extra cuando se realicen las operaciones.

Algunas opciones requieren información adicional. En el caso que deseéis conectaros a un servidor externo, podéis utilizar la opcion --host o -h pero debéis acompañar esta opción con el nombre del servidor. Por ejemplo:

- `mysql --host=host.ejemplo.com`
- `mysql -h host.ejemplo.com`

Las dos sentencias anteriores son equivalentes. La diferencia entre las opciones largas y las opciones cortas cuando requieren de otros argumentos es que con las opciones largas debéis poner un signo = entre la opción y el argumento, mientra que con las opciones cortas simplemente debéis separalas con un espacio.

En otros casos, cuando una opción requiere de parámetros, si no los especificamos, la aplicación asumirá valores pre-determinados o preguntará que valores se desean en el caso de claves. Por ejemplo, si deseamos ingresar a la interfáz mysql con el usario root, usamos la opcion -u seguido por el nombre del usuario. La opción -p requiere del parámetro "password" o clave pero si no lo especificamos, mysql lo pedirá:

- `mysql -u root -p`

Esto sucede solo con claves, en el resto de casos, mysql tomará los valores pre-determinados.

3.3 Uso de MySQL en modo interactivo

El Cliente MySQL permite el envío de consultas al servidor MySQL y recibir resultados de dos formas:

a. De forma interactiva. Es la que estudiaremos mas detenidamente, es mas util para el uso cotidiano, en sentecias rápidas y que se hacen una sola vez. También sirve para probar el uso de las sentencias SQL.
b. A través de un archivo, en modo batch, que han sido previamente escritas y almacenadas en archivo de texto (o **script SQL**). Se usa para series de consultas complejas o dificiles de ingresar manualmente.

3.3.1 Ejemplos de comandos interactivos

Veamos algunos ejemplos:

- Para visualizar la versión de MySQL:
  `SELECT VERSION ();`
- Para borrar la pantalla (muy útil si lo tenemos llena de datos; ojo, solo el Linux):
  `\! clear;`
- Para seleccionar una Base de Datos (por ej; world)
  `USE world;`
- Para salir del cliente tenemos `\q`, `exit` o `quit`:
  `exit;`
- (Recordatorio) Para entrar en el cliente:
  `mysql -u root -p`

3.3.2 Terminadores de sentencias

Para acabar una sentencia (que puede constar de innumerables comandos y parámetros) tenemos dos alternativas:

a. El carácter punto y coma (;) que será el mas usado.
b. La secuencia \G. Equivalente al anterior.

En este caso aparece en Horizontal los registros.
Veamos un ejemplo:

```
USE world;

SELECT VERSION (),
DATABASE ();
```

```
mysql> use world;
Database changed
mysql> SELECT VERSION (),
    -> DATABASE ();
+-----------+------------+
| VERSION () | DATABASE () |
+-----------+------------+
| 5.5.27     | world      |
+-----------+------------+
1 row in set (0.00 sec)

mysql>
```

```
mysql> DELIMITER ya
mysql> SHOW databases ya
+--------------------+
| Database           |
+--------------------+
| information_schema |
| cdcol              |
| mysql              |
| performance_schema |
| phpmyadmin         |
| test               |
| world              |
+--------------------+
7 rows in set (0.00 sec)
```

Por otro lado podemos modificar el terminador punto y coma por otro (eso si, sin que entre en conflicto con caracteres susceptibles de ser usados en sentencias) usando el comando DELIMITER (se pone tras un espacio y sin terminador, solo pulsando INTRO)

Veamos un ejemplo:

```
DELIMITER ya

SHOW DATABASES ya
```

Una vez probado, volvemos a dejarlo en el punto y coma (para el resto del curso).

```
DELIMITER ;
```

Otro terminador de sentencias es \G, similar a los anteriores, solo que muestra los datos de forma vertical, mostrando cada columna en una línea separada y separando cada fila (o registro) con una línea de asteriscos.

Un ejemplo:

```
SELECT VERSION (),
DATABASE () \G
```

```
mysql> SELECT VERSION (),
    -> DATABASE () \G
*************************** 1. row ***************************
 VERSION (): 5.5.27
DATABASE (): world
1 row in set (0.00 sec)
```

Observese que podemos tener sentencias de varias líneas, separadas por comas (,). Hasta no ver el terminador, el servidor no devuelve el resultado.

En el caso de escribir mal una sentencia, mysql nos avisará del error y su tipo.
Por otro lado, si queremos cancelar una sentencia, escribimos \c
Ejemplo:

```
SHOW tables,
\c
```

3.3.3 Los prompts de mysql

El prompt es la secuencia de caracteres que aparece en la terminal cuando está esperando una sentencia para su ejecución. Aquí una tabla de los principales prompt de mysql:

Prompt	Significado
mysql>	Listo para una nueva consulta.
->	Esperando la línea siguiente de una consulta multi-línea.
'>	Esperando la siguiente línea para completar una cadena que comienza con una comilla sencilla (').
">	Esperando la siguiente línea para completar una cadena que comienza con una comilla doble (").

3.3.3.1 Uso del prompt

Los comandos multi-línea comúnmente ocurren por accidente cuando tecleamos ENTER, pero olvidamos escribir el punto y coma. En este caso mysql se queda esperando para que finalicemos la consulta:

```
mysql> SELECT USER()
    ->
```

Si esto llega a suceder, muy probablemente mysql estará esperando por un punto y coma, de manera que si escribimos el punto y coma podremos completar la consulta y mysql podrá ejecutarla:

```
mysql> SELECT USER();
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)
mysql>
```

Los prompts '> y "> ocurren durante la escritura de cadenas. En mysql podemos escribir cadenas utilizando comillas sencillas o comillas dobles (por ejemplo, 'hola' y "hola"), y mysql nos permite escribir cadenas que ocupen múltiples líneas. De manera que cuando veamos el prompt '> o "> , mysql nos indica que hemos empezado a escribir una cadena, pero no la hemos finalizado con la comilla correspondiente.

Aunque esto puede suceder si estamos escribiendo una cadena muy grande, es más frecuente que obtengamos alguno de estos prompts si inadvertidamente escribimos alguna de estas comillas.

Por ejemplo:

```
mysql> SELECT * FROM mi_tabla WHERE nombre = "Lupita AND edad < 30;
">
```

Si escribimos esta consulta SELECT y entonces presionamos ENTER para ver el resultado, no sucederá nada. En lugar de preocuparnos porque la consulta ha tomado mucho tiempo, debemos notar la pista que nos da mysql cambiando el prompt. Esto nos indica que mysql está esperando que finalicemos la cadena iniciada ("Lupita).

En este caso, ¿qué es lo que debemos hacer? . La cosa más simple es cancelar la consulta. Sin embargo, no basta con escribir c, ya que mysql interpreta esto como parte de la cadena que estamos escribiendo. En lugar de esto, debemos escribir antes la comilla correspondiente y después c :

```
mysql> SELECT * FROM mi_tabla WHERE nombre = "Lupita AND edad < 30;
"> " c
mysql>
```

El prompt cambiará de nuevo al ya conocido mysql>, indicándonos que mysql está listo para una nueva consulta.

Es sumamente importante conocer lo que significan los prompts '> y ">, ya que si en algún momento nos aparece alguno de ellos, todas la líneas que escribamos a continuación serán consideradas como parte de la cadena, inclusive cuando escribimos QUIT. Esto puede ser confuso, especialmente si no sabemos que es necesario escribir la comilla correspondiente para finalizar la cadena, para que podamos escribir después algún otro comando, o terminar la consulta que deseamos ejecutar.

3.3.3.2 Cambiar el prompt

Para modificar el prompt debemos usar el comando PROMPT
Veamos un ejemplo:

```
mysql> PROMPT ivan>
PROMPT set to 'ivan>'
ivan>
```

También se puede añadir en el prompt información como el usuario, el host y la base de datos actual (algo muy util para saber donde estamos).
Para el usuario: \u
Para el host: \h
Para la base de datos: \d

```
PROMPT (\u@\h) [\d]\>
PROMPT set to '(\u@\h) [\d]\>'
(root@localhost) [world]>
```

```
mysql>PROMPT (\u@\h) [\d]\>
PROMPT set to '(\u@\h) [\d]\>'
(root@localhost) [world]>prompt
Returning to default PROMPT of mysql>
mysql>
```

Para volver al prompt predeterminado, escribimos, de nuevo `PROMPT` (sin nada)..

3.3.4 Teclas de edición en mysql

El cliente mysql permite la recuperación de líneas anteriores previamente ejecutadas. Además, en Linux soporta la completación de comandos mediante el tabulador.
Las teclas disponibles serán las siguientes:

- **Derecha / izquierda:** mueven el cursor para adelante o para atrás.
- **Arriba / Abajo,** permite recuperar comandos previos (historial de comandos).
- **CTRL + A / CTRL + E:** situa el cursor al principio o al final de la línea.
- **CTRL + K:** Borrar todo desde el cursor hacia adelante.
- **CTRL + R:** Busca en el historial de comandos. Aparece lo siguiente:
  `(reverse-i-search) `':`

El historial de comandos se guarda entre sesiones en Linux, pero no en Windows.
Dicho historial se guarda, en Linux, en el directorio personal del usuario, en el archivo oculto `.mysql_history`

3.3.4.1 Archivos scripts en mysql

Mysql permite las consultas introducidas por teclado y las que usan un archivo. A este último, que contiene sentencias SQL se le conoce como archivo script o batch. Este archivo debe ser de texto plano (por ejemplo con gedit o kate) y acabar cada sentencia con un terminador.

Veamos un ejemplo:
En la carpeta de Descargas nos creamos un archivo de texto plano llamado sentencias.sql.

El contenido será:

```
SHOW databases;
USE world;
SHOT tables;
```

```
mysql> SOURCE sentencias.sql
+--------------------+
| Database           |
+--------------------+
| information_schema |
| cdcol              |
| mysql              |
| performance_schema |
| phpmyadmin         |
| test               |
| world              |
+--------------------+
7 rows in set (0.00 sec)

Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
+-----------------+
| Tables_in_world |
+-----------------+
| City            |
| Country         |
| CountryLanguage |
+-----------------+
3 rows in set (0.00 sec)
```

Luego emplearemos el comando SOURCE para indicar el origen de las sentencias:

```
SOURCE sentencias.sql
```

Algunos puntos importantes a considerar:

a. Puede haber errores. Si es el caso, mysql ignorará el resto del batch.
b. Para forzar la ejecución de TODOS los comandos (haya errores o no) usaremos:
   `-- force o -f`
c. Cuidado con los bucles. Un archivo PUEDE llamar a otro archivo, pero este último NO PUEDE llamar al primero (bucle infinito).
   Por ejemplo:
   `SOURCE sentencias.sql`

   Contenido de **sentencias.sql:**
   `SOURCE sentencias_siguiente.sql`

   Contenido de **sentencias_siguiente.sql**
   `SOURCE sentencias.sql`

3.3.4.2 Formatos de salida de mysql

De forma predeterminada, el formato de salida es tabular (como ya hemos visto).
De todos modos se puede modificar con los siguientes parámetros:

- `-- batch` o -B: Genera una salida en modo batch (en Linux no funciona)
- `-- html`: Genera la salida en HTML
- `-- xml`: Genera la salid en xml

En todos los casos debemos indicar la opción al arrancar el cliente. Veamos un ejemplo:

```
mysql -u root -- xml -p
USER world;
SHOW tables;
```

```
mysql> SHOW tables;
<?xml version="1.0"?>

<resultset statement="SHOW tables;" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="Tables_in_world">City</field>
  </row>

  <row>
        <field name="Tables_in_world">Country</field>
  </row>

  <row>
        <field name="Tables_in_world">CountryLanguage</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

3.3.4.3 Comandos y Sentencias SQL

Hagamos un resumen de algunos comandos propios del cliente y otros SQL (los mas importantes).
En primer lugar los SQL:

- **SELECT** → Para consultas
- **INSERT** → Para insertar registros
- **UPDATE** → Para actualizar
- **DELETE** → Para borrar

Ahora los propios del cliente (que no permiten varias líneas), por ejemplo:

- **QUIT** → Salir
- **SOURCE** → Recurso (batch)
- **STATUS** → Estado

En el caso de este último:

```
mysql> STATUS
```

Aparece esto:

```
Connection id:          16
Current database:       world
Current user:           root@localhost
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server version:         5.5.27 Source distribution
Protocol version:       10
Connection:             Localhost via UNIX socket
Server characterset:    latin1
Db     characterset:    latin1
Client characterset:    utf8
Conn.  characterset:    utf8
UNIX socket:            /opt/lampp/var/mysql/mysql.sock
Uptime:                 1 hour 9 min 50 sec
```

Por último tenemos la ayuda: **HELP <COMANDO>**
Para usarlo previamente hay que realizar lo siguiente:

a. Copiar el archivo **fill_help_tables.sql** que está en **/opt/lampp/share/mysql**
b. Pegarlo por ejemplo en Descargas. Desde allí ejecutar este comando:
   `mysql -u root mysql < fill_help_tables.sql -p`
c. Al poner la clave volverá el simbolo del sistema. Volvemos a entrar:
   `mysql -u root mysql  -p`
d. Y ejecutamos el comando de ayuda: `HELP SHOW;`

4 CONSULTAS DE DATOS

En este módulo va a tener 3 partes diferenciadas:

a. Como usar el comando SELECT.
b. Como agrupar/acumular resultados en las consultas.
c. Uso de la palabra UNION

4.1 La sentencia SELECT.

La sentencia SELECT es usada para obtener datos de una o varias tablas en una base de datos. Pertenece a la categoría de sentencias DML (Data Manipulation Language, o Lenguaje de Manipulación de datos), siendo la mas usada.

Una sentencia SELECT muestra un conjunto particular de filas de una Base de Datos. Se envía al servidor que devuelve las filas correspondientes. Dichas filas se denominan **Conjunto Resultante**.

La sentencia SELECT se construye a partir de un número de cláusulas que especifican cómo y cuales datos serán extraidos. El módelo básico de la sintaxis será:

```
SELECT [clausula / opciones] listacolumnas
FROM nombretabla[s]
WHERE condicion;
```

Veamos un ejemplo sencillo: En primer lugar vamos conectarnos, usar la BBDD wolrd, ver sus tablas y la estructura de estas:

```
mysql -u root -p
USE world
SHOW tables
DESCRIBE
```

Y ahora el SELECT:

```
SELECT DISTINCT Continent
FROM Country;
```

```
mysql> SELECT DISTINCT Continent FROM Country;
+---------------+
| Continent     |
+---------------+
| North America |
| Asia          |
| Africa        |
| Europe        |
| South America |
| Oceania       |
| Antarctica    |
+---------------+
7 rows in set (0.01 sec)
```

Con esta consulta sacamos los Continentes a los que pertenecen los paises de la BBDD wolrd. DISTINCT (que ya veremos) permite sacar valores distintos, no repetidos.
Si no podemos DISTINCT nos sale 239 registros, tantos como los paises registrados.

Por otro lado SELECT también permite usar expresiones matemáticas, y no estar basada en tablas de la base de datos. MySQL dispone de mas de 30 funciones matemáticas:
Ref: http://dev.mysql.com/doc/refman/5.5/en/mathematical-functions.html

Veamos un ejemplo:

```
SELECT sqrt (9);
```

Permite calcular la raiz cuadrada de 9.

Por supuesto, podemos poner expresiones mas complejas:

```
SELECT ( sqrt (9) ) / 3;
```

```
mysql> SELECT sqrt (9);
+----------+
| sqrt (9) |
+----------+
|        3 |
+----------+
1 row in set (0.00 sec)
```

4.1.1 Usos Básicos de SELECT.

- **FROM:** Especifica las tablas de donde se extraerán los datos.
- **DISTINCT:** Elimina Filas duplicadas
- **WHERE:** Solo devuelven filas que cumplen un filtro particular
- **ORDER BY:** Ordena las filas en función a una lista de expresiones.
- **LIMIT:** Devuelve un número concreto de registros tras aplicar la consulta.

- La estructura básica de los SELECT será:

```
SELECT [lista_opciones] campos
[FROM tablas]
[WHERE …]
[ORDER BY …]
LIMIT numero_registros
```

Ojo, el orden es importante, puesto que puede dar un error de sintaxis. Por otro lado, por lo normal, es mejor poner las palabras reservadas en MAYUSCULAS.

4.1.2 Uso de FROM

Se pude usar de dos formas diferentes:

- La forma mas normal será sin cualificar: `FROM nombre_tabla`
- La otra manera será de manera cualificada, indicando la BBDD

```
FROM nombre_bbdd.nombre_tabla
```

Ejemplo:

```
SELECT Name FROM country;
```

4.1.3 Uso de DISTINCT

Si el resultado de una consulta da filas duplicadas, se puede omitir el resultado de las mismas mediante la palabra reservada DISTINCT. Veamos un ejemplo:

```
SELECT Continent
FROM Country;
```

Salen 239 Registros.

```
SELECT DISTINCT Continent
FROM Country;
```

Salen solo 9 Filas.

4.1.4 Uso de WHERE

Esta clausula realiza un filtro de la consulta solicitada, mostrando solo los registros que cumplan una condición particular.
Entre las condiciones que se pueden usar están:

- Literales: 'New York'
- Numéricos: 4
- Constantes incorporadas: TRUE, FALSE, NULL
- Referencias a Columnas
- Funciones, etc.

Ejemplo:

```
SELECT Continent, Name
FROM Country
WHERE Continent="Europe";
```

4.1.5 Uso de ORDER BY

El orden del resultado de las consultas devueltas será siempre el mismo que el introducido. Para modificar este comportamiento, tenemos la sentencia ORDER BY. La estructura será:

```
SELECT [Opciones] campos
FROM tablas
WHERE condicion
ORDER BY campos [ASC / DESC]
```

Un ejemplo:

```
SELECT Name
FROM Country
ORDER BY Name
```

De forma predeterminada, ordena Name de forma ascendente.
(Menor a Mayor).

Cambiando el ejemplo anterior:

```
SELECT Name
FROM Country
ORDER BY Name DESC
```

También se pueden ordenar mas de un campo.
Esto se hace separando los campos y sus parámetros por comas.

```
SELECT Continent, Name
FROM Country
WHERE Continent = "Europe"
OR Continent = "North America"
ORDER BY Continent ASC, Name ASC;
```

Práctica:

- Consulta con los campos CountryCode, Language donde el idioma sea "Swedish" ordenado por CountryCode de la Z->A

```
SELECT CountryCode, Language
FROM CountryLanguage
WHERE Language = "Swedish"
ORDER BY CountryCode DESC;
```

4.1.6 Uso de IN

La sentencia IN se usa con WHERE para indicar valores a incluir en los resultados de la consulta. Por ejemplo, queremos ver los DISTRICT (Provincias, Estados -no países-) a los que pertenecen las ciudades de Madrid, New York y Roma:

```
SELECT Name, District
FROM City
WHERE Name IN ("Madrid","New York","Roma");
```

4.1.7 Uso de LIMIT

Mediante el uso de clausula LIMIT sacamos sólo una parte del resultado de la ejecución de una consulta. Muy útil cuando la salida de registros es bastante elevada.
Puede darse con uno o dos argumentos:

- `LIMIT num_filas`
- `LIMIT num_saltar, num_filas`

Veamos un ejemplo de cada uno:

```
SELECT Name
FROM Country
```

Devuelve 239 Registros

```
SELECT Name
FROM Country
LIMIT 5;
```

Devuelve 5 Registros

Ahora otro ejemplo que ya vimos:

```
SELECT DISTINCT Continent
FROM Country;
```

Salen 7 Continentes.

Pues bien, queremos omitir los 2 primeros y presentar los siguientes 5.

```
SELECT DISTINCT Continent
FROM Country
LIMIT 2,5;
```

También se puede presentar con una salida en Horizontal con:

```
SELECT DISTINCT Continent
FROM Country
LIMIT 2,5 \G
```

Veamos otros dos ejemplos incluyendo el operador de comparación > (que veremos luego)

```
SELECT Name, Population
FROM Country
WHERE Population > 35000000
AND
(Continent = "Europe" OR Continent = "North America")
ORDER BY Population DESC;
```

Y limitando a 3 registros y presentandolo en horizontal:

```
SELECT Name, Population
FROM Country
WHERE Population > 35000000
AND
(Continent = "Europe" OR Continent = "North America")
ORDER BY Population ASC
LIMIT 3 \G
```

4.2 Agregación de resultados en las consultas.

Hasta ahora hemos visto SELECT simples donde cada fila resultante correspondía a una fila de la/las tablas originales. Pero podemos obtener como filas el resultado de un grupo de filas originales. Esto es lo que se llama Agregación.SQL permite aplicar funciones de agregación que ahora estudiaremos.

4.2.1 Uso de las Funciones de Agregación:

Estas son las funciones de Agregación (ojo, entre la función y los parámetros dentro de () no puede haber espacios):

- MAX() → Devuelve el valor máximo
- MIN() → Devuelve el valor mínimo
- SUM() → Devuelve el sumatorio
- COUNT() → Cuenta Filas, Valores No Nulos o valores distintos.
- AVG() → Media Aritmética
- STD () → Desviación Estándar
- GROUP_CONCAT() → Concatenar cadenas de caracteres

Veamos un ejemplo de COUNT:

En primer lugar la consulta simple:

```
SELECT Name
FROM Country
WHERE Population > 35000000
AND
(Continent = "Europe"
OR Continent = "North America")
ORDER BY Population DESC;
```

Y ahora la consulta agregada, al que de paso le ponemos un Alias:

```
SELECT COUNT(Name) AS Paises
FROM Country
WHERE Population > 35000000
AND
(Continent = "Europe"
OR Continent = "North America")
ORDER BY Population DESC;
```

Si quiero poner en el alías espacios en blanco hay que poner la cadena entre comillas.

```
…
SELECT COUNT(Name) AS "Países de Europa/América"
…
```

Ejercicio:

- Ver la población del país de América o Europa cuya población sea mayor de 30millones y menos de 50millones con más población.
- La misma consulta de antes pero con la menor población.
- A partir de las consultas anteriores, sacar la media de las poblaciones de los países anteriores.

4.2.2 Agrupación mediante GROUP BY

Existe otra construcción que permite agrupar filas, GROUP BY. Al igual que ORDER BY permite trabajar sobre una lista de expresiones separadas por coma. Si está presente todas las filas que tienen la misma combinación de valores para las expresiones especificadas con GROUP BY son tratas como grupo y mostradas como una fila en el conjunto resultante.

La cláusula GROUP BY se aplica después de WHERE pero antes de ORDER BY.
Una función de agregación puede usarse con y sin GROUP BY. Sin las cláusula GROUP BY la función calcular el valor resumido en base al conjunto entero de filas seleccionadas. GROUP BY con una función de agregación calcula el valor resumido para cada grupo.

Por ejemplo, si una cláusula WHERE selecciona 20 filas y la cláusula GROUP BY las clasifica en 4 grupos de 5 filas, la función de resumen generará un valor para cada uno de esos grupos.

Un ejemplo:
Población Media por continente:
```
SELECT Continent, AVG(Population)
FROM Country
GROUP BY Continent;
```

Y otro: Población Total por Continente:
```
SELECT Continent, SUM(Population)
FROM Country
GROUP BY Continent;
```

Al igual que ORDER BY, GROUP BY permite expresiones, no solo campos.

Otro ejemplo mas, número de paises por continent:
```
SELECT Continent, COUNT(Name)
FROM Country
GROUP BY Continent
```

NOTA CURIOSA: Aunque parezca mentira, hay 5 “países” en la Antártida:
```
SELECT Name, Continent
FROM Country
WHERE Continent = 'Antarctica';
```

4.2.3 Uso de GROUP_CONCAT()

La función GROUP_CONCAT() genera un resultado concatenado para cada grupo de cadenas.
Veamos un ejemplo que usa esta sentencia junto a GROUP BY que crea una lista de los países que tienen una forma particular de gobierno en Suramérica (South America):

NOTA: De nuevo, cuidado con los espacios en las funciones o en los valores.
Por ejemplo, da error poner GROUP_CONCAT (Name).
Y si ponemos ' South America', el resultado sale vacío.
Para ver la consulta anterior en Horizontal:
```
SELECT GovernmentForm,
GROUP_CONCAT(Name) AS Paises
FROM Country
WHERE Continent = 'South America'
GROUP BY  GovernmentForm \G
```
NOTA: Si ponemos \G; nos sale un error (la máquina considera que hay 2 consultas a ejecutar).

4.2.4 La cláusula WITH ROLLUP

El modificador WITH ROLLUP permite generar niveles de valores resumidos. Por ejemplo, podemos ver la población total de cada continente y ADEMÁS la suma total de todos los continentes (añado un alias):

```
SELECT Continent, SUM(Population) AS "Suma Población"
FROM Country
GROUP BY Continent
WITH ROLLUP;
```

Es importante recalcar que dicha cláusula se aplica sobre la función descrita al principio de la consulta. Por ejemplo, veamos esta consulta:

```
SELECT Continent, AVG(Population) AS Media_pob
FROM Country
GROUP BY Continent
WITH ROLLUP;
```

La línea final **Roll Up** contiene el promedio TOTAL (no la suma de promedios ni el promedio de promedios) de los países del mundo (es decir, los 239 registros).

4.2.5 La cláusula HAVING

Esta cláusula permite eliminar filas en base a valores agregados. Se puede usar con y sin cláusula GROUP BY. Sólo debe ser usada si hay necesidad de aplicar una condición que se refiera a la función agregada. Si no es así, la condición debe ser manejada en la clausula WHERE, no mediante HAVING.

En este ejemplo vemos una lista de continentes cuya población total sea mayor a 700 millones:

```
SELECT Continent, SUM(Population) AS Suma_pob
FROM Country
GROUP BY Continent
HAVING  SUM(Population) > 700000000;
```

Otro ejemplo mas; Provincias (o estados -que no paises-) con mas de 30 ciudades:

```
SELECT District, COUNT(*) AS Numero_Ciudades
FROM City
GROUP BY District
HAVING Numero_Ciudades >=30;
```

Obsérvese como hemos usado el alias en el propio HAVING.

En resumen, HAVING es lo mismo que WHERE, es decir, un filtrado de registros. Pero WHERE NO permite usarlo con Funciones Agregadas (como por ejemplo SUM) y sin embargo HAVING se usa para filtrar registros con Funciones Agregadas aplicadas.

4.2.6 Operadores de MySQL

MySQL soporta distintos tipos de operadores, entre los que están:

- Operadores Aritméticos
  - + → Suma
  - - → Resta
  - * → Producto
  - / → División
  - DIV → División Entera
  - % → Resto División o módulo

Ejemplo:

```
SELECT 10 %3;
```

- Operadores Comparación
  - x=y → Igual a
  - x <=> y → Igual a, incluyendo nulos
  - x<y → menor que

Ejemplo:

```
SELECT Continent, SUM(Population) AS Suma_pob
FROM Country
GROUP BY Continent
HAVING SUM(Population) <1000000;
```

  - <= → Menor igual
  - >= → Mayor igual
  - <> → Distinto
  - != → Distinto

Ejemplo

```
SELECT Continent, SUM(Population) AS Suma_pob
FROM Country
GROUP BY Continent
HAVING Continent <> "Europe";
```

4.3 Uso de UNION

La sentencia UNION permite la concatenación de los resultados obtenidos tras la ejecución de 2 o mas consultas SELECT. La sintaxis será:
SELECT …
…
UNION
SELECT …
….
UNION
…..

De forma predeterminada, UNION elimina las filas duplicadas. Para conservar todas las filas, debemos usar UNION ALL. Esta última es mas eficiente que UNION para el servidor, porque no elimina filas (restando tráfico). Sin embargo el retorno al cliente será mayor (al tener mas registros para mostrar.

UNION es útil en estos casos:

- Existe información similar en varias tablas y queremos obtener filas de todas ellas al mismo tiempo.
- Es necesario seleccionar varios conjuntos de filas de la misma tabla pero las condiciones que caracterizan cada conjunto no es fácil de escribir en una sola clausula WHERE. UNION permite obtener de cada conjunto con una cláusula WHERE mas sencilla en su propia sentencia SELECT; las filas obtenidas en cada una son combinadas para producir el resultado final de la consulta.

Un ejemplo:
Combinar los nombres de los países de la Antártica (Antarctica) y Suramérica (South America):

```
SELECT Name
FROM Country
WHERE Continent = 'Antarctica'
UNION
SELECT Name
FROM Country
WHERE Continent = 'South America';
```

La consulta anterior es exactamente igual que esta:

```
SELECT Name
FROM Country
WHERE Continent IN
("North America", "South America");
```

También se puede hacer con esta consulta:

```
SELECT Name
FROM Country
WHERE Continent="North America"
OR Continent="South America";
```

4.4 SELECT de Varias Tablas

Hasta ahora hemos visto consultas donde únicamente implicábamos una única tabla. Pero evidentemente esto puede ser ampliado. ¿Como? Uniendo tablas a partir de los campos que las interrelacionan. Para ello, en la cláusula WHERE añadiremos sendos campos de ambas tablas. Por ejemplo, queremos visualizar los idiomas hablados en España.

Si únicamente escribimos esto:

```
SELECT LocalName, Language
FROM Country, CountryLanguage
WHERE LocalName="España";
```

Aparecerán 984 registros (producto de multiplicar España por las 984 Idiomas de la Tabla CountryLanguage).

La solución es añadir la unión entre ambas:

```
SELECT LocalName, Language
FROM Country, CountryLanguage
WHERE Code=CountryCode
AND LocalName="España";
```

En este caso, además de añadir la unión de ambas tablas, incluimos con AND otro filtro.
Pero, ¿que ocurre si los campos de ambas tablas se llaman igual?
¿Como hacemos para juntar mas de 2 tablas?
Para la primera pregunta debemos incluir <nombreTabla>.<nombreCampo>
Para juntar mas de 2 tablas, añadimos otras igualaciones.
Por ejemplo, queremos visualizar las ciudades del mundo donde se habla Español:

```
SELECT City.Name
FROM CountryLanguage, Country, City
WHERE City.CountryCode = Country.Code
AND Country.Code = CountryLanguage.CountryCode
AND Language = "Spanish";
```

NOTA IMPORTANTE: El orden en las igualaciones, así como el orden en la enumeración de las tablas en el FROM no es esencial. Podemos cambiarlo si lo deseamos.

Actividades Complementarias

Para acabar con el módulo os pongo como práctica varios ejercicios:

1. Listar las diferentes regiones del mundo.
2. Listas los 3 primeros países en orden alfabético.
3. Listar todos los países bálticos, pertenecientes a la región 'Baltic Countries'
4. Listas todos los países donde la esperanza de vida es mayor que 79 años.
5. Listar las 5 ciudades mas grandes del mundo.
6. Listar los distintos países (y sus códigos) que tienen ciudades con mas de 7 millones de habitantes. ¿Cuantos países hay?.
7. Encontrar los 5 tipos de gobierno mas comunes en el mundo (entendiendo “mas comunes” como el que haya en “mayor número de países”).
8. Listar la superficie promedio de los países de cada continente.
9. Calcular la esperanza promedio de vida de cada continente (recordar que cada país tiene distinta población).
10. Listar los 5 países con mayor densidad de población en el mundo (solo incluir países con una superficie mayor que 10000km^2).
11. ¿Cuántos distritos distintos hay representados en la tabla City?.
12. Listar todos los idiomas que se hablan en mas de 10 países.

5 MANEJO DE ERRORES Y ADVERTENCIAS

5.1 Modos SQL que afectan a la sintaxis

Los modos SQL controlan aspectos de la operación del servidor como la sintaxis de SQL que MySQL debe soportar y que tipo de validaciones debe efectuar sobre los datos. El modo SQL existe a nivel global y a nivel de sesión.

Por defecto, una nueva sesión usar el modo SQL (sql_mode) global, pero durante el curso de la sesión puede cambiarse. El sql_mode se puede cambiar a nivel de sesión mientras algunos usuarios privilegiados pueden tambien cambiar ese modo a nivel global.

Para saber cual es el modo SQL global que tenemos:
Lo establecemos con:

```
SET GLOBAL sql_mode = 'TRADITIONAL';
```

Y lo vemos con:

```
SELECT @@global.sql_mode;
```

Que nos sale vacío porque está puesto en modo normal.

Para saber el Modo SQL de la sesión actual:

```
SET SESSION sql_mode = 'TRADITIONAL';
```

Y luego lo vemos con:

```
SELECT @@session.sql_mode;
```

5.1.1 Definición del Modo SQL

Ya hemos visto un par de ejemplos para definir el Modo SQL en el apartado anterior.
La sintaxis será:

```
SET [SESSION| GLOBAL] sql_mode = 'mode_value'
```

Al darle valor a GLOBAL se afectan a todos los usuarios que se conecten posteriormente. Al darle valor a SESSION, únicamente al usuario actual. Los usuarios pueden cambiar su session_sql_mode. Si no especifica ninguno, toma por defecto el de SESSION.

Se puede poner varios valores separados por comas y encerrados todos entre comillas (que en el caso de ser un solo modo son opcionales). Se puede poner en minúsculas, pero recomiendo que, como las palabras reservas, se pongan en mayúsculas. Y ojo, **no puede haber entre los valores espacios**.

Otro ejemplo:

```
SET sql_mode = 'IGNORE_SPACE,ANSI_QUOTES';
```

Que entre otras cosas nos va a permitir separar los parentesis de los parámetros de las funciones.
Por ejemplo:

```
SELECT Continent, COUNT (*)
FROM Country
GROUP BY Continent;
```

Ahora cambiamos a TRADITIONAL

```
SET sql_mode = 'TRADITIONAL';
```

Y repetimos la consulta. ¿que pasa? Que da error.

5.1.2 Modos SQL: ANSI

El primer modo que vamos a ver es ANSI, que se ajusta mejor al estándar SQL. Este modo contiene a su vez varios modos que son:

a. `ANSI_QUOTES`: Cambiar el significado del carácter de la doble comilla. Si está activado, las dobles comillas pueden ser usadas como identificadores. Por defecto las dobles comillas se emplean como delimitadores de cadenas de caracteres.
b. `IGNORE_SPACE`: Permite el espacio entre el nombre de la función y el paréntesis de los parámetros.
c. `PIPES_AS_CONCAT`: Usa el carácter || como un operador de concatenación (como CONCAT).
d. `REAL_AS_FLOAT`: Trata el tipo REAL como un equivalente al FLOAT

5.1.3 Otros Modos SQL

Además del ANSI tenemos los siguientes modos:

a. `ONLY_FULL_GROUP_BY`: no permite las consultas cuando la lista SELECT se refiere a columnas que no están en la cláusula GROUP BY (antes era parte de ANSI)
b. `ERROR_FOR_DIVISIÓN_BY_ZERO`: Con este modo, si se intenta dividir entre 0 dará un mensaje de alerta y un valor NULL, ejecutando la consulta, sin dar un error.

c. `STRICT_TRANS_TABLES, STRICT_ALL_TABLES`: Define un modo estricto para los valores de entrada. Si se pone, el servidor rechaza valores que están fuera de rango, tipo incorrecto o que están faltando en colunas que no tienen valores por defecto. Si el valor no puede ser metido en una tabla transaccional, se aborta la sentencia. Para una tabla no transaccional, se aborta la sentencia si el valor aparece en una sentencia sobre una sola fila, o en la primera fila de una sentencia que involucra múltiples filas.

d. `NO_ZERO_DATE, NO_ZERO_IN_DATE`. Controlan la forma en que MYSQL maneja datos de fecha inválidos. MySQL de forma predeterminada requier eque el mes y día sean de una fecha real válida, excepto que permiten fechas en cero ('0000-00-00') y fechas parcialmente en 0 (Ej: '2009-12-00' o '2009-00-01'). Las fechas en cero -total o parcial- son aceptadas en modo estricto. Para no permitirlas hay que activar dicho modo y NO_ZERO_DATE, NO_ZERO_IN_DATE.

Existen muchos mas modos que están explicados en esta dirección:
http://dev.mysql.com/doc/refman/5.0/es/server-sql-mode.html

5.1.4 El modo TRADITIONAL

El modo `TRADITIONAL` hace que MySQL se comporte mas como un SGBD tradicional. Un descripción sencilla de este modo es que da un error en vez de un mensaje de alerta cuando se inserta un valor incorrecto en una columna.

Es un modo compuesto por otros, que son:

- `STRICT_TRANS_TABLES`
- `STRICT_ALL_TABLES`
- `NO_ZERO_DATE`
- `NO_ZERO_IN_DATE`
- `ERROR_FOR_DIVISIÓN_BY_ZERO`
- `NO_AUTO_CREATE_USER`, que no permite la creación automática de nuevos usuarios.

5.2 Manejo de datos Inválidos o Faltantes

MySQL es uno de los servidores de bases de datos menos tradicionales y mas permisivos con el manejo de datos. Hay otros mas estrictos que dan constantes errores en el ingreso de datos.

La conducta normal de MySQL es convertir los valores erróneos de entrada en valores lo mas cercanos posible a los permitidos (según sea el tipo y la definición de la columna), siguiendo con el procesado de la orden. Por ejemplo, si se quiere meter un valor negativo en una columna UNSIGNED, MySQL lo convertirá en 0.

5.2.1 Manejos de datos Faltantes

En MySQL las sentencias INSERT (que ya veremos) pueden estar incompletas en el sentido de no especificar un valor para todas las columnas de una tabla. Veamos un ejemplo:

```
CREATE DATABASE faltantes;
USE faltantes;
```

Nos creamos la tabla siguiente:

```
CREATE TABLE t
(
      i INT DEFAULT NULL,
      j INT NOT NULL,
      k INT DEFAULT -1
);
```

Para esta tabla, una sentencia está incompleta a menos que especifique valores para las tres columnas de la tabla. Cada una de las siguientes sentencias es un ejemplo de una sentencia en la que faltan valores de columnas.

```
INSERT INTO t (i) VALUES (0);
INSERT INTO t (i,k) VALUES (1,2);
INSERT INTO t (i,k) VALUES (1,2),(3,4);
INSERT INTO t VALUES ();
```

En el último caso, una lista VALUES en blanco significa que se usa el valor por defecto en todas las columnas. Obsérvese como en todos los casos de 1 warning (faltan valores).

Para ver el resultado de la introducción de datos:

```
SELECT * FROM t;
```

MySQL maneja los valores faltantes de la siguiente manera:

- Si la definición de la columna contiene una cláusula DEFAULT, MySQL inserta dicho valor. Obsérvese como MySQL añade una cláusula DEFAULT NULL a la definición si no tiene una cláusula DEFAULT explícita y la columna puede tomar el valor NULL. De modo que la definición de la columna i realmente tiene DEFAULT NULL en su definición.

SHOW CREATE TABLE t;

- Si la definición de la columna no tiene DEFAULT, el manejo del valor faltante dependerá de lo estricto que sea el modo SQL activo y si la tabla es transaccional.
  - Si el modo estricto está activo, inserta el valor implicito para el tipo de datos, mostrando un warning (como hemos visto).
  - Si el modo estricto está activo y la tabla es transaccional ocurre un error (se descarta). Para las no transaccionales solo ocurre en una actualización parcial : si el error ocurre en la 2ª o siguientes filas, las anteriores son insertadas.

5.2.2 Manejo de valores Inválidos en Modo No Estricto

MySQL realiza la conversión de tipo según las restricciones impuestas por la definición de una columna. Estas restricciones se aplican a los siguientes contextos:

- Con sentencias como INSERT, REPLACE, UPDATE o LOAD DATA INFILE
- Cuando se cambia la definición de una tabla con ALTER TABLE
- Cuando se especifica un valor por defecto usando la cláusula DEFAULT en la definición de la columna. Ejemplo: ponemos como valor por defecto '43' -cadena- para una columna numérica; esta cadena se convierte a 43 -numero- al ser usado.

Para desplegar los mensajes de alerta usaremos la sentencia SHOW WARNINGS después de su aparición. Por ejemplo (siguiendo con el ejemplo):

```
INSERT INTO t VALUES ();
SHOW WARNINGS;
```

En esta lista pongo las principales conversiones que realiza MySQL:

- Conversión de valores fuera de rango a dentro de rango.
- Truncamiento de String
- Conversión de valores de ENUM a vacío
- Conversión de SET. Desechando solo los no válidos
- Conversión a valores por defecto de los tipos de datos.
- Manejo de la asignación de NULL a columnas NO NULL: inserta el valor por defecto.

5.2.3 Manejo de los valores inválidos en el modo estricto

Los valores de entrada pueden no ser válidos por los siguientes motivos:

a. Para una columna numérica o temporal, un valor podría estar fuera de rango.
b. Para una columna tipo string, la cadena podría ser demasiado larga.
c. En una columna ENUM se podría introducir un valor no fuera válido en la enumeración o como parte de un valor. Para una columna SET, un valor podría contener un elemento que no forma parte del conjunto.
d. Para una columna definida como NOT_NULL, podría darse el valor NULL.

En todos los casos, se generará un error y se cancelará la ejecución.

5.2.4 Restricciones adicionales en la entrada de datos

Como ya hemos visto podemos activar para la sesión actual el modo TRADITIONAL con:

```
SET sql_mode = 'TRADICITIONAL';
```

Esto tiene como ventaja que la inserción de datos se realiza de un modo absolutamente estricto, asegurandonos que en futuras versiones de MySQL dichas inserciones serán compatibles.

5.2.5 Interpretación de Mensajes de Error

Si ocurren fallos en la conexión o mientras el servidor intenta ejecutar una sentencia SQL, se genera un mensaje de diagnóstico. Los clientes pueden desplegar esta información como ayuda para resolver problemas. MYSQL ofrece información de diagnóstico asi:

- Un mensaje de error es devuelto por sentencias que fallan. Ejemplo:

```
SELECT * FROM tabla;
ERROR 1146 (42S02): Table 'faltantes.tabla' doesn't exist
```

Estos mensajes tienen 3 componentes:

- Un Código de error específico de MySQL
- Un Código de error estándar SQLSTATE ( ), definidos por SQL y ODBC
- Un mensaje de texto que describe el problema.

- Una cadena de información es devuelta para sentencias que afectan a multiples filas. Esta secuencia muestra un resumen.
- Un error a nivel de sistema operativo, que incluye un ErrCode que podría tener un significado especifico para cada sistema.
- Un código numérico de error, sin mas.

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/innodb-error-codes.html
http://dev.mysql.com/doc/refman/5.0/es/operating-system-error-codes.html

5.2.6 La sentencia SHOW WARNINGS

Ya hemos visto anteriormente la sentencia SHOW WARNINGS que muestra las alerta del cliente ante errores en la sentencia a ejecutar. Veamoslo con un ejemplo mayor:

```
CREATE TABLE enteros
(
      i INT UNSIGNED NOT NULL
);

INSERT INTO enteros
VALUES ('abc'), (-5), (NULL);

SHOW WARNINGS;
Warning |1366| Incorrect integer value: 'abc' for column 'i' at row 1
Warning |1264| Out of range value for column 'i' at row 2
Warning |1048| Column 'i' cannot be null
```

Se puede usar esta sentencia con otras como LIMIT o COUNT. Por ejemplo:

```
SHOW COUNT(*) WARNINGS;
```

Los niveles de alerta serán 3:

- Errores, que parán la ejecución
- Warnings, que continuan con la ejecución
- Notas, que solo muestran información. Si queremos quitarlas, podenos:

```
SET sql_notes = 0;
```

5.2.7 La sentencia SHOW ERRORS

Funciona igual que SHOW WARNINGS y muestran los errores graves. Ejemplo:

```
INSERT INTO tablaImaginaria VALUES (1,2);
SHOW ERRORS;
```

6 TIPOS DE DATOS

6.1 Una visión general

En MySQL los tipos de datos se pueden dividir en las siguientes 4 categorias:

- Numéricos → Enteros, flotantes, punto fijo y campo-bit
- Carácter → Cadenas de Texto
- Binario → Cadenas de datos binarios.
- Temporal → Fechas y horas.

El uso de los tipos de datos mas adecuados tiene especial incidencia en el rendimiento de la base de datos, por lo que es esencial corregir posibles errores en las etapas de diseño.

6.1.1 El ABC de los tipos de datos

Se resumen en:

- Apto → El dato requiere estar representado en el tipo mas adecuado a la entidad
- Breve → Eligiendo el tipo mas pequeño, ahorrando espacio y mejorando el rendimiento
- Completo → El tipo debe tener en cuenta la mejor estimación sobre el tamaño máximo que el dato puede llegar a tener.

6.1.2 Creación de tablas con tipos de datos

La sintáxis básica será:

```
CREATE TABLE nombre_tabla
(
      campo1 tipo_dato [atributos],
      campo2 tipo_dato [atributos],
      …...
);
```

Veamos un ejemplo completo:

```
CREATE DATABASE tipos;
USE tipos;
CREATE TABLE personal
(
      id INT UNSIGNED NOT NULL,
      nombre CHAR(30) NOT NULL,
      apellido CHAR(30) NOT NULL
);
```

Obsérvese la forma de presentar la sentencia por líneas y tabuladas.
Eso si, no está permitida añadir después de cada parámetro mas tabulaciones.

6.2 Tipos de datos numéricos

MySQL ofrece los siguientes tipos de datos numéricos:

a. Enteros
b. Punto Flotante o reales (con decimales) aproximados.
c. Punto Fijo, reales de valor exacto.
d. Bit, para valores de tipo BIT.

Cuando se elije un tipo numérico se debe considerar los siguientes factores:

1. El rango de valores que el tipo representa
2. La cantidad de espacio en disco
3. La precisión y escala para valores de punto flotante / fijo.

En este último caso, la precisión es el número de digitos (cifras) significativos. La escala es el número de dígitos a la derecha del punto decimal.

6.2.1 Tipos Enteros (Integer)

Las fracciones como 23.56 no están permitidos. Los tipos son estos:

Tipo	Bytes	Valor Mínimo	Valor Máximo
TINYINT	1	-128	127
UNSIGNED		0	255
SMALLINT	2	-32768	32767
UNSIGNED		0	65535
MEDIUMINT	3	-8388608	8388607
UNSIGNED		0	16777215
INT	4	-2147483648	2147483647
UNSIGNED		0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807
UNSIGNED		0	18446744073709551615

Como puede observarse, si añadimos el atributo UNSIGNED eliminamos los valores negativos posibles, duplicando el valor máximo para cada tipo.

Veamos un ejemplo:

```
USE test;

CREATE TABLE integers
(
      n SMALLINT UNSIGNED
);

INSERT INTO integers
VALUES (5);
SELECT * FROM integers;

INSERT INTO integers
VALUES (-5);
SELECT * FROM integers;

INSERT INTO integers
VALUES (70000);
SELECT * FROM integers;
```

Si se cambiar el modo:

```
SET sql_mode = 'TRADITIONAL';
```

Ahora no deja la inserción

```
INSERT INTO integers
VALUES (70000);
SALE ERROR
SELECT * FROM integers;
```

(valores anteriores se mantienen)

6.2.2 Tipos de Punto Flotante

Pueden ser de tipo FLOAT o DOUBLE.
Si se especifica UNSIGNED, los valores negativos no se permiten.
Pueden dar errores de redondeo (si se excede el rango).

FLOAT: Un número de coma flotante pequeño (de precisión simple). Valores permitidos:
Rango con signo: De -3.402823466E+38 a -1.175494351E-38
Rango sin signo: 0 y De 1.175494351E-38 a 3.402823466E+38.

DOUBLE: Número de coma flotante de tamaño normal (precisión doble). Valores permitidos:
Rango con signo: -1.7976931348623157E+308 a -2.2250738585072014E-308
Rango sin signo: 0, y de 2.2250738585072014E-308 a 1.7976931348623157E+308.

Tanto para FLOAT como para DOUBLE el valor por defecto es NULL.
La forma de asignar en ambos caso será (ojo a los espacios entre función y parámetros):
FLOAT(P,S) Ejemplo: campo1 FLOAT (4,2)
DOUBLE(P,S) Ejemplo: campo2 DOUBLE (10,4)

Donde P es el número máximo de dígitos y S es el número de digitos tras el punto decimal.
Si es FLOAT y P+S > 7, los valores pueden ser truncados.
Si es DOUBLE y P+S > 15, los valores pueden ser truncados.

La longitud especificada no afecta al almacenamiento real de los números, sino a cómo se muestran. Veamos un ejemplo para mirar el GNP (Producto Interior Bruto):

```
USE world;

SELECT Name, GNP
FROM Country
ORDER BY GNP DESC
LIMIT 10;
```

Veamos otro ejemplo:

```
SET sql_mode = '';

CREATE TABLE floating
(
      n FLOAT
);

INSERT INTO floating
VALUES (.99);

SELECT * FROM floating;

SELECT * FROM floating
WHERE n=0.99;

SELECT * FROM floating
WHERE n
BETWEEN 0.99-.001 AND 0.99+.001;
```

6.2.3 Tipo Punto Fijo

El formato se llama DECIMAL y es un formato de almacenamiento decimal fijo. Es menos eficiente que FLOAT o DOUBLE pero mas exacto que estos. Se usa sobre todo en aplicaciones financieras que requieren cálculos de divisas con gran precisión.

Se asigna de la siguiente manera:
DECIMAL(P,S)

Características:

- P es la precisión o número máximo de digitos siginificativos (hasta 65)
- S es la escala, digitos que siguen al punto decimal (hasta 30).
- S nunca puede ser mayor a P.
- El valor por defecto es NULL (si lo permite la columna) o 0 si es NOT NULL.
- Si se omite P y/o el valor predeterminado es DECIMAL(10,0)

Por ejemplo:
```
campo3 DECIMAL (10,2)
```

En MySQL existe un tipo equivalente muy similar, NUMERIC.

Veamos otro ejemplo:
```
CREATE TABLE fijo
(
      n DECIMAL (3,1)
);
INSERT INTO fijo VALUES (42.1);
SELECT * FROM fijo;
```

¿Que pasa si nos excedemos del rango?
```
INSERT INTO fijo VALUES (142.1);
```
Si tenemos el sql_mode en predeterminado, nos dará un warning.
```
SHOW WARNINGS;
```
Y veremos cómo deja el dato en el límite:
```
SELECT * FROM fijo;
```

6.2.4 Tipos BIT

Permiten tipos de valores BIT.
El número máximo permitido será 64 Bits
Por ejemplo:
campo4 BIT(20)
La forma de introducir valores será con b'valor'

Veamos un ejemplo:
```
CREATE TABLE bits
(
      b BIT (10)
);
INSERT INTO bits VALUES (b'101');
SELECT * FROM bits;
```

Podemos hacer cálculos con dicho valor:
```
SELECT b+0 FROM bits;
```

6.3 Tipos de datos de Cadenas de caracteres

Un tipo String o cadena de caracteres es un conjunto de datos alfanuméricos que pertenecen a una ordenación especifica. Son tan importantes y útiles que son soportados por casi todos los lenguajes de programación (incluyendo SQL).

MySQL soporta los siguientes tipos de cadenas de caracteres:

- `CHAR` → Secuencia de caracteres de longitud fija, max 255.
- `VARCHAR` → Secuencia de caracteres de longitud variable max 65536.
- `TINYTEXT` → Secuencia de caracteres de longitud variable max 255
- `TEXT` → Secuencia de caracteres de longitud variable máximo $2^{16}-1$
- `MEDIUMTEXT` → Secuencia de caracteres de longitud variable máximo $2^{24}-1$
- `LONGTEXT` → Secuencia de caracteres de longitud variable máximo $2^{32}-1$
- `ENUM` → Enumeración; Conjunto fijo de valores permitidos.
- `SET` → Enumeración; Conjunto fijo de valores permitidos.

6.3.1 Los tipos CHAR y VARCHAR

Con `CHAR`, la longitud de la cadena es fija y está definida estrictamente. Por ejemplo, `CHAR`(20) guardará 20 bytes en cada registro, independientemente de si todos se usan. Al contrario, las cadenas `VARCHAR` tienen longitud variable y los requerimientos de almacenamiento varían según el tamaño real de la cadena.

Aunque los tipos `VARCHAR` y `TEXT` puedan parecer idénticos, tienen una diferencia significativa. En el caso de VARCHAR, el tamaño máximo tiene que ser especificado por el usuario en el momento de definir la tabla. Las cadenas que sean más largas serán truncadas sin ningún aviso.

A partir de MySQL 5.0 hay dos novedades importantes para VARCHAR:

- El tamaño máximo es 65535 bytes, mientras que antes era de 255. Es importante resaltar que el tamaño máximo está medido en bytes, por lo que el máximo de caracteres puede ser menor ya que según el tipo de codificación se necesita más de 1 byte para codificar un carácter.
- Los espacios al principio y final de la cadena ahora son almacenados, mientras que antes eran eliminados antes de almacenar la cadena.

Los tipos `CHAR` y `VARCHAR` pueden llevar el atributo `BINARY`. En este caso la columna se comporta esencialmente como un `BLOB`. Este atributo puede ser útil ya que cuando se usa, el criterio de ordenación se rije exclusivamente por el valor binario de los bytes de la cadena, y no depende del juego de caracteres que estemos usando.

Veamos un ejemplo:

```
USE test;
CREATE TABLE caracteres
(
      c VARCHAR (5)
);

INSERT INTO caracteres VALUES ('abc');
SELECT * FROM caracteres;
```

¿Qué sucede si se excede el rango?

```
INSERT INTO caracteres VALUES ('abcdef');
SELECT * FROM caracteres;
```

6.3.2 Juegos de caracteres

En las columnas de tipo texto se puede usar el atributo adicional:
`CHARACTER_SET nombre_juego_caracteres COLLATE criterio_ordenación`
Los juegos de caracteres especifican la manera en que los caracteres son codificados y el criterio para ordenar las cadenas de caracteres. La mayoría de los juegos de caracteres tienen en común la codificación de los 128 caracteres “ingleses” de ASCII. El problema comienza con la codificación de los caracteres internacionales. Desde la perspectiva “euro-anglosajona”, hay dos grandes familias de juegos de caracteres:

I. Juegos de caracteres latinos. En el pasado, cada región desarrolló su propia codificación de 1 byte, de las cuales el juego Latin ha sido el más extendido:
   A. Latin1 (ISO-8859-1) caracteres más comunes de Europa occidental (äöüXßáàå...).
   B. Latin2 (ISO-8859-2) contiene caracteres de idiomas de la Europa de este y oriental.
   C. Latin0 (o Latin9, ISO-8859-15) igual que Latin1 pero con el símbolo del Euro (€).

El problema con estos juegos es que ninguno contiene todos los caracteres de los idiomas europeos.

II. Unicode. Para resolver el problema de los juegos Latin se creó el juego Unicode que usa 2 bytes por carácter. Con 65536 caracteres posibles, cubre no solo Europa sino la mayoría de los idiomas asiáticos.

    Pero no podía ser tan fácil ... Unicode solo determina qué código está asociado a cada carácter, no como los códigos se guardan internamente. Por ello hay diferentes variantes, de las cuales UCS-2 (Universal Character Set) y UTF-8 (Unicode transfer format) son las más importantes.

UCS-2, también llamado UTF-16, representa lo que aparentemente es la solución más simple, que es usar 2 bytes para codificar cada carácter. Sin embargo, tiene dos inconvenientes: el espacio para almacenar cadenas de caracteres se duplica, incluso cuando se están representando cadenas de idiomas europeos. Segundo, el segundo byte usualmente es 0, especialmente cuando se representan cadenas en inglés. Muchos programas escritos en C consideran que un carácter 0 significa el final de una cadena, lo cual puede dar lugar a problemas.

Por esto, UTF-8 es la alternativa más popular a UTF-16. En este caso, los caracteres ASCII se representan con un solo byte, cuyo primer bit es 0. El resto de caracteres Unicode se representan con 2 ó 4 bytes. La desventaja de este formato es que no hay una relación directa entre el tamaño en bytes de una cadena y su tamaño en caracteres. Este formato es el más estándar para representar Unicode.

A pesar de las ventajas de Unicode, no todo es sencillo. El principal problema es que las cadenas de caracteres en Unicode son incompatibles con las clásicas en ASCII. Además, el soporte a Unicode en algunas herramientas web no está presente. Por ejemplo, solo a partir de la versión 5.2 de PHP incorpora soporte para Unicode.

Una vez se ha escogido un juego de caracteres, también se puede escoger el criterio de ordenación. Esto es debido a que un mismo juego de caracteres contiene elementos de muchos idiomas al mismo tiempo, por lo que cuando queremos ordenar alfabéticamente un conjunto de cadenas de caracteres, dependerá del idioma querremos un resultado u otro. Para determinar el orden alfabético a usar con un determinado juego de caracteres usaremos el atributo COLLATE al definir un juego de caracteres.

Para ver todas las opciones que tiene nuestro servidor usaremos
`SHOW COLLATE:`

6.3.3 Los tipos ENUM y SET

Hay dos tipos de datos que son particulares de MySQL: ENUM y SET.
Permiten el manejo eficiente de conjuntos y enumeraciones.
Con ENUM se puede representar una lista de hasta 65535 cadenas de caracteres a las que se asignan números enteros consecutivos (similar al tipo ENUM de C).

El tipo SET es parecido, pero además distingue el orden en que los elementos están dispuesto, permitiendo la representación de combinaciones. Requiere más espacio de almacenamiento y además solo puede manejar como máximo 64 valores.

Veamos un ejemplo:

```
CREATE TABLE enumeraciones
(
      nombre VARCHAR(60),
      color SET ('rojo','azul','verde')
);

INSERT INTO enumeraciones
VALUES ('Pepito','rojo');
SELECT * FROM enumaraciones;

INSERT INTO enumeraciones
VALUES ('Menganito','naranja');
SELECT * FROM enumaraciones;
```

6.4 Tipos de Datos Binarios

Para el almacenamiento de datos binarios hay cuatro variantes del tipo BLOB, de manera similar al tipo TEXT. La diferencia es que los datos binarios siempre se ordenan y se compran usando directamente su valor sin mediar ninguna codificación.

El uso de este tipo tiene ventajas y desventajas. Si queremos almacenar imágenes o sonido, podemos usar el tipo BLOB. La ventaja es que los datos están integrados en la base datos, con lo cual entran dentro de los backups, de las recuperaciones cuando hay caídas del sistema, .. etc. Sin embargo, ralentiza el funcionamiento de la base de datos.

Otra desventaja es que normalmente los datos BLOB solo se pueden leer enteros, es decir, no se pueden leer partes de ellos. La alternativa a usar el tipo BLOB es tener los datos binarios en ficheros externos.

MySQL soporta los siguientes tipos de datos Binarios:

- `BIT(n)` → Datos en bits, donde n es el número de bits (máximo 64)
- `TINYBLOB` → Datos binarios de tamaño variable, máximo 255 bytes
- `BLOB` → Datos binarios de tamaño variable, máximo $2^{16}-1$ bytes
- `MEDIUMBLOB` → Datos binarios de tamaño variable, máximo $2^{24}-1$ bytes
- `LONGBLOB` → Datos binarios de tamaño variable, máximo $2^{32}-1$ bytes

6.5 Tipos de Datos temporales

MySQL soporta los siguientes tipos de datos Temporales:

- `TIME` (hora). Tiempo en la forma '23:59:59', rango ±838:59:59 (3 bytes)
- `YEAR` (año). Año 1900-2155 (1 byte)
- `DATE` (fecha). Formato: '2008-12-31'; Rango 1000-01-01 a 9999-12-31 (3 bytes)
- `DATETIME` (fecha y hora)
- `TIMESTAMP` (pone la hora actual). Lo mismo que DATETIME, pero con inicialización automática al día y hora en que se hace la modificación

Veamos un ejemplo:

```
USE test;
CREATE Table hora_actual
(
      texto VARCHAR(20),
      tiempo TIMESTAMP
);

INSERT INTO hora_actual (texto)
VALUES ('Esta es la hora actual');
SELECT * FROM hora_actual;
```

6.6 Valores Nulos

NULL es una palabra reservada de SQL que puede aparecer en lugares que normalmente se esperaría un valor. Representa genéricamene un “valor nulo”.
Se debe usar para aquellos campos en los que puede existir nulos de forma puntual.

En la base de datos world que usamos como ejemplo, existe el siguiente campo:

```
LifeExpectancy FLOAT (3,1) DEFAULT NULL;
```

Mas información:
http://dev.mysql.com/doc/refman/5.0/en/storage-requirements.html

7 EXPRESIONES SQL

7.1 Comparaciones en SQL.

En este tema veremos las expresiones en las sentencias SQL. Pueden aparecer en sentencias SELECT, cláusulas ORDER BY y GROUP BY. También en cláusulas WHERE y HAVING, que dan expresiones booleanas.

7.1.1 Componentes de las Expresiones SQL

Las expresiones pueden aparecer en la cláusula WHERE así como en sentencias SELECT, DELETE y UPDATE. También pueden usarse en cláusulas ORDER BY y GROUP BY.
En las expresiones usaremos literales (núneros, cadenas de caracteres, fechas y horas), constantes integradas como NULL, TRUE y FALSE, referencias a columnas y llamadas a funciones. Se pueden combinar en expresiones mas complejas mediante operadores que serán lógicos o aritméticos, de comparación o equivalentes a patroes.

Veamos ejemplos de expresiones:

```
USE test;
SELECT 14, -312.82, 4.32E-03, 'Soy una cadena';
```

Tambien podemos usar funciones:

```
SELECT CURDATE();
SELECT VERSION();
```

Las expresiones se pueden combinar para dar lugar a otras mas complejas:

```
SELECT Name,
TRUNCATE(Population/SurfaceArea,2) AS 'Densidad Poblacion',
IF (GNP > GNPOld,'PIB Crece','PIB Decrece') AS 'Estado Economico'
FROM Country
ORDER BY Name DESC
LIMIT 20;
```

Donde hemos usado varias funciones y palabras reservadas...
`TRUNCATE` -> Trunca el número de Decimales.
Sintaxis: `TRUNCATE` (Expresion,Decimales)
`IF` -> Condicional SI
`AS` -> Alias de campos

7.1.2 Expresiones Numéricas

Se emplean los tipos de datos numéricos vistos anteriormente. Hay que diferenciar valores exactos (como enteros o decimales) de aproximados, como los FLOAT, que tienen notacion científica. Esto puede comprobarse en una simple comparación:

```
USE test;
SELECT 1.1 + 2.2 = 3.3, 1.1E0 + 2.2E0 = 3.3E0;
```

Obsérvese como la primera igualdad da TRUE y la segunda FALSE.

Si se mezclan números con cadenas que continen números, estos últimos son convertidos.

```
SELECT 1+'1', 1='1';
```

Evidentemente se pueden emplear funciones matemáticas.
Mas información:
http://dev.mysql.com/doc/refman/5.0/es/mathematical-functions.html

7.1.3 Expresiones con Cadenas

Se definen las cadenas de caracteres por aquellos elementos entre comillas simples (las mas usadas y recomendadas) y dobles. Veamos dos funciones muy usadas y sus ejemplos:
`CONCAT` -> Concatena cadenas
Sintaxis: `CONCAT (cadena,cadena2...,cadena_n)`
`REPEAT` -> Repite la cadena varias veces
Sintaxis: `REPEAT (cadena,veces)`
Ejemplo:

```
SELECT CONCAT ('abc','def',REPEAT('x',3));
```

Otro ejemplo:

```
SELECT CONCAT  (Name, ' está en la provincia ', District)
FROM City
WHERE CountryCode = 'ESP' ORDER BY Name ASC;
```

Por otro lado el operador | | es manejado por mysql como un OR lógico. Esto está en contra del SQL estándar, y puede ser modificado con:

```
SET sql_mode = 'PIPES_AS_CONCAT.
```

Veamos un ejemplo:

```
USE test;
SET sql_mode='';
```

Definimos el sql_mode predeterminado.

```
SELECT 'abc' || 'def';
```

Cambiamos el sql_mode:

```
SET sql_mode='PIPES_AS_CONCAT';
SELECT 'abc' || 'def';
```

Distinción de mayúsculas y minúsculas en comparaciones de cadenas

Tenemos tres tipos de ordenaciones:
`ci` → Case Insensitive
`cs` → Case Sensitive
`bin` → Binario

Los que nos interesa son los dos primeros. En el segundo caso (cs) distingue entre mayúsculas y minúsculas, por lo que se recomienda algunas directrices en el uso de sentencias:

1. Usar mayúsculas para las palabras reservadas
2. Para el resto, usar preferentemente minúsculas.

Para ver el Set de caracteres y su ordenación:

```
SHOW CHARACTER SET;
SHOW COLLATION;
```

Los que mas usaremos son el latin y el utf8:

```
SHOW COLLATION LIKE  'latin1%';
SHOW COLLATION LIKE  'utf8%';
```

Veamos un ejemplo de la importancia de elegir el juego de caracteres:En Español:

```
SET collation_connection = 'latin1_spanish_ci';
```

Comparando cadenas:

```
SELECT 'Niño'='Nino';
```

cambiando...

```
SET collation_connection = 'latin1_german1_ci';
SELECT 'Niño'='Nino';
```

Uso de LIKE para compara patrones

La comparación entre patrones se realiza con la palabra reservada LIKE. Sintaxis:
expresión LIKE patrón
Normalmente LIKE lo usaremos con los caracteres especiales _ (guión bajo) y % (porcentake):

1. % equivale te a cualquier número de caracteres. Ejemplos:
   a) 'a%' → Equivale a una cadena que empiece por a
   b) '%b' → Equivale a una cadena que termine en b
   c) '%c% → Cualquier cadena que contenga c
2. _ equivale a cualquier carácter simple. Por ejemplo:
   a) 'as_' puede equivaler a... así, asa, as@, as?, etc

Un ejemplo:
```
USE world;
SELECT name
FROM Country
WHERE name LIKE 'United%';
```
Salen 4 registros.

Para invertir los patros de comparación, debemos usar NOT LIKE. Ejemplo:
```
SELECT name
FROM Country
WHERE name NOT LIKE 'United%';
```
Salen 235 registros.

Derivado de LIKE tenemos RLIKE (o REGEXP) que permite comparaciones con expresiones regulares que no necesitan % o _.

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/pattern-matching.html

Algunos ejemplos:
```
SELECT name
FROM City
WHERE name RLIKE 'nat';

SELECT name
FROM City
WHERE name RLIKE '^New.*rk$';
```
Los caracteres comodín mas usados para RLIKE son:

- `^` Comienza...
- `$` Acaba
- `.*` Cualquier cadena

Otro ejemplo mas:
```
SELECT name
FROM City
WHERE name RLIKE 'Los|Las';
```

Lo anterior es equivalente a:
```
SELECT NAME
FROM City
WHERE name RLIKE 'L[ao]s';
```

7.1.4 Expresiones Temporales

Se pueden usar los operadores normales de comparación, aunque los mas usados son:
`BETWEEN ... AND` → Busca fechas entre un valor y otro.
`INTERVAL` → Permite operaciones aritméticas con fechas

Un ejemplo:
Vamos a importar otra base de datos de ejemplo de mysql.
https://launchpad.net/test-db/+download
Debemos descargarnos el archivo:
`employees_db-full-1.0.4.tar.bz2`
(ojo, ocupa 25.5Mb)

Lo descomprimimos. Nos sale una carpeta. Editamos el archivo employees.sql
Podemos cambiar el motor de almacenamiento de Innodb por el que queramos quitando comentarios al nombre. De forma predeterminada, se instalará con Innodb.

Para la instalación, es muy sencilla, nos metemos por consola en la carpeta que se ha descomprimido, así:
```
cd employee_db
```
Y luego ponemos el siguiente comando:
```
mysql -u root -p -t < employees.sql
```

Ahora, un ejemplo para `BEETWEEN … AND`
Queremos saber los empleados que han nacido en los 3 primeros días de 1960:
```
SELECT first_name, last_name, birth_date
FROM employees
WHERE birth_date
BETWEEN '1960-01-01' AND '1960-01-03'
ORDER BY birth_date;
```

Veamos un ejemplo para INTERVAL.
```
SELECT '2012-11-22' + INTERVAL 90 DAY AS '3 meses mas';
```

7.2 Funciones en las Expresiones MySQL.

Las funciones pueden ser llamadas dentro de las expresiones y devolver un valor que es usado en el lugar de la llamada a la función cuando la expresión es evaluada. No puede haber espacios entre el nombre de la función y el paréntesis, a no se que tengamos activados el `sql_mode = SQL_IGNORE_SPACE`.

7.2.1 Funciones de Comparación.

- `LEAST ()` → Devuelve el menor valor

```
SELECT LEAST (4,3,8,-1,5), LEAST ('cdef','ab','ghi');
```

- `GREATEST ()` → Devuelve el mayor valor.

```
SELECT GREATEST (4,3,8,-1,5), GREATEST ('cdef','ab','ghi');
```

- `INTERVAL ()` → Toma como argumentos expresiones enteras. El valor del primer argumento es comparado con los siguientes.
  Devuelve el indice de último que es igual o menor.

```
SELECT INTERVAL (10,1,2,4,8,16);
```

7.2.2 Funciones de Flujo de Control

Estas funciones permiten elegir entre distintos valores en base al resultado de una expresión.

- IF () → Evalua la expresión del primer argumento, devolviendo el segundo si es TRUE o el tercero si es FALSE.

```
SELECT IF (1>0,'Si','No');
```

Otro ejemplo:
Queremos ver los paises del Europa, empezando por España.
NOTA: esta sentencia SQL se suele emplear mucho en webs.

```
SELECT Name
FROM Country
WHERE Continent='Europe'
ORDER BY IF (code='ESP',1,2), name
LIMIT 10;
```

- `CASE`: Evalua una expresión y la compara tras un WHEN. Si es verdadero, se devuelve la expresión tras un THEN. Si es falso, tras un ELSE. La sintaxis:

```
CASE exp_case
      WHEN exp_when THEN result_exp
      [WHEN exp_when THEN result_exp]...
      ELSE  result_exp
END
```

Un ejemplo: Queremos presentar los paises de europa, empezando por ESP, FRA y POR.

```
SELECT Name
FROM Country
WHERE Continent='Europe'
ORDER BY
CASE Code
      WHEN 'ESP' THEN 1
      WHEN 'FRA' THEN 2
      WHEN 'POR' THEN 3
      ELSE 4
END, name
LIMIT 10;
```

Existe una variante para el CASE en el que la condición va después.
Su sintaxís será el siguiente:

```
CASE
      WHEN cond_when THEN result_exp
      [WHEN cond_when THEN result_exp]...
      ELSE  result_exp
END
```

Aquí se evalúa el when que, al ser verdadera, convierte el THEN en el resultado del case. Si ninguna es verdadera, pasamos al ELSE. Si no existe el ELSE, el resultado es NULL.
Un ejemplo:
Vamos a comparar la población y el producto interior bruto (GNP) de Europa, Estados Unidos y el resto del mundo.

```
SELECT
CASE
      WHEN Code = 'USA' THEN 'Estados Unidos'
      WHEN Continent = 'Europe' THEN 'Europa'
      ELSE 'Resto del Mundo'
END
      AS Area,
      SUM(GNP),
      SUM(Population)
FROM Country
GROUP BY Area;
```

7.2.3 Funciones Numéricas

Las funciones numéricas efectuan varios tipos de operaciones matemáticas como redondeo, truncamiento, calculos trigonométricos y generación de números aleatorios.

- `ROUND ()` → Realiza un redondeo para números enteros, decimales y negativos.

```
USE test;
SELECT ROUND (28.5), ROUND (-28.5);
SELECT ROUND (2.85E1), ROUND (-2.85E1);
```

- `FLOOR ()` → Devuelve el mayor entero no mayor que su argumento.
- `CEILING ()` → Devuelve el menor entero no menor que su argumento.

```
SELECT FLOOR (-14.7), FLOOR (14.7);
SELECT CEILING (-14.7), CEILING (14.7);
```

- `ABS ()` → Extrae el valor absoluto
- SIGN () → Expresa el signo, siendo 1 positivo y -1 negativo.

```
SELECT ABS (-14.7), ABS(14.7);
SELECT SIGN (-14.7), SIGN (14.7), SIGN (0);
```

- `SIN, COS, TAN` → Seno, Coseno, Tangente
- `DEGREES` → Convierte a grados
- `RADIANS` → Convierte a Radianes
- `PI` → Muestra Pi

7.2.4 Funciones con Cadenas de Caracteres

Tenemos en primer lugar 3 funciones equivalentes para buscar cadenas:

- `INSTR ()` → Cadena, subcadena a buscar
- `LOCATE ()` → Subcadena a buscar, cadena, [desplazamiento]
- `POSITION ()` → Subcadena a buscar, cadena separados por IN. Estándar SQL.

Ejemplos:

```
SELECT INSTR ('Alice y Bob','y'),
LOCATE('y','Alice y Bob'),
POSITION('y'IN'Alice y Bob')\G
```

LOCATE permite un tercer argumento, el inicio de la búsqueda:

```
SELECT LOCATE ('','Alice, Ricardo y Bob',7);
```

- `LENGTH ()` → Longitud de la cadena de caracteres en Bytes
- `CHAR_ LENGTH ()` → Longitud en caracteres

Ejemplos:

```
SELECT LENGTH ('MySQL'), CHAR_LENGTH ('MySQL');
```

Esto cambia si se usa conjuntos de caracteres multibytes, como ucs2:

```
SELECT LENGTH (CONVERT ('MySQL' USING ucs2)) AS Longitud,
CHAR_LENGTH (CONVERT ('MySQL' USING ucs2)) AS Longitud;
```

- `CONCAT ()` → Concatena sus argumentos
- `CONCAT_WS ()` → Usa el primer argumento como separados, concatenando el resto de argumentos.

Ejemplos:

```
SELECT CONCAT ('Argumentos', ' a ', 'concatenar');
SELECT CONCAT_WS ('-','Argumentos', ' a ', 'concatenar');
```

Mas información
http://dev.mysql.com/doc/refman/5.0/es/string-functions.html

7.2.5 Funciones Temporales

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/date-and-time-functions.html

7.2.6 Funciones relacionadas con NULL

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/problems-with-null.html

7.2.7 Comentarios en las Sentencias SQL

MySQL soporta tres tipos de comentarios:

- Multilínea, usando /**/
- Línea, usando –
- Línea, usando #

Veamos algunos ejemplos:

```
USE test;
CREATE TABLE t (
      i INT
) /*! ENGINE = MEMORY */
```

Aquí estamos ante una variante de /**/ donde el comentario es parte de la sentencia y se ejecuta (por tanto el comentario y la sentencia dicen que el motor es Memory).

Siguiendo con esta variante, se puede indicar en el comentario que la sentencia sea ejecutada sólo para determinadas versiones de MySQL. Por ejemplo:

```
SHOW /*!50002 FULL */ TABLES
```

Indica que, para versiones 5.02 o superiores de MySQL se ejecute SHOW FULL TABLES.

7.2.8 Comentarios en Objetos de la Base de Datos

Se pueden poner comentarios con la palabra reservada COMMENT de hasta 60 caracteres en:

- TABLAS

Ejemplo:

```
USE test;
CREATE TABLE 'CountryLanguage'
(
      id  CountryLanguage INT,
      CountryLanguage VARCHAR (50)
) ENGINE = MyISAM
COMMENT 'Listado de Lenguajes hablados';
```

Observese que no hay comas entre la línea del ENGINE y COMMENT.

- COLUMNAS (Campos)

Ejemplo:

```
USE test;
CREATE TABLE 'Country'
(
      CountryCode CHAR (3) NOT NULL
            COMMENT 'Código que identifica al país',
      CountryName CHAR (30) NOT NULL
            COMMENT 'Nombre del país'
);
```

8 OBTENER METADATOS

8.1 Métodos para Acceder a los metadatos

La forma de acceder a la estructura de las bases de datos se resumen en 3 modos:

- Usando la base de datos `INFORMATION_SCHEMA`
- Uso de sentencias `SHOW y DESCRIBE`
- Uso del programa `mysqlshow`

8.2 La Base de Datos Information_Schema

`INFORMATION_SCHEMA:` Almacén de datos de información de las BBDD

Veamos un ejemplo:

```
SELECT TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'INFORMATION_SCHEMA'
ORDER BY TABLE_NAME;
```

8.2.1 Tablas de Information_Schema

Tras realizar la última sentencia sale una serie de tablas, cuya función es la siguiente:

- `CHARACTER_SETS` → Conjunto de caracteres
- `COLLATIONS` → Ordenaciones de caracteres
- `COLLATION_CHARACTER_SET_APPLICABILITY` → Ordenaciones para un conjunto determinado.
- `COLUMNS` → Campos de tablas
- `COLUMN_PRIVILEGES` → Privilegios
- `ENGINES` → Motores
- `EVENTS` → Eventos
- `FILES` → Archivos donde se almacenan los NDB Disk Data
- `GLOBAL_STATUS` → Estados Globales
- `GLOBAL_VARIABLES` → Variables Globales
- `INNODB_CMP` → Información Tablas comprimidas
- `INNODB_CMPMEM` → Información Tablas comprimidas en el Buffer de Memoria
- `INNODB_CMPMEM_RESET` → Reseteo de las estadísticasTablas Comprimidas en el Buffer de memoria
- `INNODB_CMP_RESET` → reseteo de las estadísticas sobre las operaciones de compresión y descompresión.
- `INNODB_LOCKS` → Bloqueos sobre tablas de Motor InnoDB
- `INNODB_LOCK_WAITS` → Bloqueos en Espera
- `INNODB_TRX` → Transacciones sobre tablas InnoDB
- `KEY_COLUMN_USAGE` → Restricciones de campos clave
- `PARAMETERS` → Parámetros
- `PARTITIONS` → Tablas de particiones
- `PLUGINS` → Complementos del Servidor
- `PROCESSLIST` → Hilos ejecutandose
- `PROFILING` → Perfiles de uso de la sesión actual
- `REFERENTIAL_CONSTRAINTS` → Claves Foráneas
- `ROUTINES` → Procedimientos y Funciones almacenadas
- `SCHEMATA` → Bases de Datos
- `SCHEMA_PRIVILEGES` → Provilegios de las cuentas de usuario sobre BBDD
- `SESSION_STATUS` → Estado de la Sesión Actual
- `SESSION_VARIABLES` → Variables de Sesión
- `STATISTICS` → Indices de las Tablas
- `TABLES` → Tablas de la BBDD
- `TABLESPACES` → Espacio para tablas
- `TABLE_CONSTRAINTS` → Restricciones sobre tablas
- `TABLE_PRIVILEGES` → Privilegios sobre tablas
- `TRIGGERS` → Disparadores de la Base de Datos
- `USER_PRIVILEGES` → Privilegios globales de las cuentas de usuario
- `VIEWS` → Vistas

8.2.2 Mostrando las tablas de Information_Schema

Para visualizar los campos de las tablas anteriores, usaremos la siguiente construcción:

```
USE information_schema;
SELECT column_name
FROM information_schema.Columns
WHERE table_schema = 'information_schema'
AND table_name = 'Views';
```

Cambiaremos Views por cualquiera de las tablas vistas en el apartado anterior.

Mas ejemplos:
Ver los datos básicos de una base de datos...

```
USE information_schema;
SELECT *
FROM INFORMATION_SCHEMA.SCHEMATA
WHERE SCHEMA_NAME = 'world' \G
```

Para visualizar las tablas y el motor de almacenamiento de una bbdd y exportarla...

```
SELECT table_name, engine INTO OUTFILE 'C:/tablas.txt'
FROM Tables
WHERE table_schema = 'world';
```

También podemos ver todas nuestras bbdd así como sus motores:

```
SELECT table_schema, engine
FROM Tables
GROUP BY table_schema, engine;
```

Podemos ver los datos generales de una tabla de una bbdd:

```
SELECT *
FROM information_schema.columns
WHERE Table_schema = 'world'
AND table_name = 'CountryLanguage'
LIMIT 5 \G;
```

Queremos ver el número de tablas de las bases de datos por cada motor de almacenamiento:

```
SELECT table_schema, engine, count(*)
FROM tables
GROUP BY table_schema, engine;
```

8.3 Uso de SHOW y DESCRIBE

Hemos visto como acceder a `INFORMATION_SCHEMA` con sentencias `SELECT`. De todos modos existen 'atajos' que emplean palabras reservadas y que permiten realizar lo mismo. Este es el caso de `SHOW` y `DESCRIBE`.

8.3.1 Sentencias SHOW

`SHOW` devuelve un tipo de metadatos. Son estos:

- `SHOW DATABASES`
- `SHOW FULL TABLES`
- `SHOW FULL COLUMNS`
- `SHOW INDEX`
- `SHOW CHARACTER SET`
- `SHOW COLLATION`

Ejemplo:

```
SHOW COLUMNS FROM CountryLanguage;
```

8.3.2 Sentencias DESCRIBE

`DESCRIBE` equivale a `SHOW COLUMNS`. Puede abreviarse con `DESC`, y no soporta, frente a `SHOW COLUMNS` las clausulas `LIKE` y `WHERE`.

```
Ejemplo:
DESC CountryLanguage;
```

Pregunta: ¿Como obtenemos el resultado anterior usando SELECT?

```
SELECT Column_name AS Field,
Column_Type AS Type,
is_Nullable AS 'Null',
Column_Key AS 'Key',
Column_Default AS 'Default',
Extra
FROM information_schema.columns
WHERE Table_schema = 'world'
AND table_name = 'CountryLanguage';
```

8.4 El programa cliente MySQLSHOW

El programa cliente mysqlshow genera información sobre la estructura de bases de datos y tablas.
Su sintaxis es:

```
mysqlshow [opciones] [nombre_bbdd [nombre_tabla [nombre_columna]]]
```

OJO: Es otro cliente de mysql, por lo que, para usarlo, debemos ir fuera del cliente mysql.

```
exit
ivan-htpc@ivan-htpc:~$ mysqlshow -u root -p
```

Para ver las opciones y la ayuda escribimos:

```
mysqlshow -u root -p --help
```

NOTA: para la ayuda no pide clave. De todos modos podemos poner la clave para agilizar las pruebas (aunque no es recomendable para la seguridad). La sentencia NO ACABA en ;.

Con 1 argumento muestra la información de una bbdd:

```
mysqlshow -u root -p world
```

Con 2 argumentos, una tabla de la bbdd:

```
mysqlshow -u root -p world City
```

Con 3 argumentos, un campo de una tabla de la bbdd:

```
mysqlshow -u root -p world City CountryCode
```

Podemos ver los índices de una tabla con -keys:

```
mysqlshow -u root -p world City --keys
```

Podemos usar caracteres especiales para realizar búsquedas. Por ejemplo, bbdd que empiecen por w (de paso pongo la contraseña, otra manera de ponerlo, poco recomendable):

```
mysqlshow -u root -p --password=root 'w%'
```

9 BASES DE DATOS

9.1 Creación de Bases de Datos

Se usa la sentencia CREATE DATABASE (SCHEMA). Por ejemplo:

```
CREATE DATABASE mydb;
```

Si se intenta crear una base de datos que ya existe, se produce un error. Para verificar que no exista, se usa la cláusula IF NOT EXISTS, que es esencial para exportaciones.

```
CREATE DATABASE IF NOT EXISTS mydb1;
```

`CREATE DATABASE` tiene dos cláusulas adicionales: `CHARACTER SET` (conjunto de caracteres) y `COLLATE` (ordenación por defecto). Aparecen al final de la sentencia, tras el nombre de la base de datos. Un ejemplo:

```
CREATE DATABASE mydb2
CHARACTER SET utf8
COLLATE utf8_danish_ci
```

Esto se usa para la creación de tablas (y sobre todo campos). Los valores anteriores se guardan en el archivo db.opt en el directorio de la base de datos.

Por último, para meternos en la Base de datos, debemos emplear USE. Ejemplo:

```
USE mydb2;
```

Por otro lado, si queremos saber cómo se ha creado una base de datos, debemos usar la sentencia `SHOW CREATE DATABASE`. Ejemplo:

```
SHOW CREATE DATABASE world \G
```

9.2 Modificando Bases de Datos.

Si queremos cambiar las opciones `CHARACTER SET` y `COLLATE` debemos usar `ALTER DATABASE`. Un ejemplo:

```
ALTER DATABASE mydb2
CHARACTER SET latin1
COLLATE latin1_swedish_ci;
```

Se actualiza al reiniciar la sesión de mysql.
Como recordatorio, el comando para ver todos los CHARACTER SET:

```
SHOW CHARACTER SET
```

Y para ver las `COLLATION` del Character SET Latin1:

```
SELECT COLLATION_NAME FROM INFORMATION_SCHEMA.COLLATIONS
WHERE collation_name LIKE 'latin1_%';
```

Y para ver las `COLLATION` del Character SET Utf8:

```
SELECT COLLATION_NAME FROM INFORMATION_SCHEMA.COLLATIONS
WHERE collation_name LIKE 'utf8_%';
```

IMPORTANTE: No podemos cambiar de nombre la Base de Datos con `ALTER DATABASE`. Para hacerlo, hay que exportarla, crear otra base de datos, importar en esta última y borrar la primera.

9.3 Eliminación de Bases de Datos

Se usa la sentencia `DROP DATABASE`. Su sintaxis es:
```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```
`DROP SCHEMA` es sinónimo de `DROP DATABASE`.
¡ATENCIÓN! No hay vuelta a atrás con este comando.
Mas información: http://dev.mysql.com/doc/refman/5.5/en/drop-database.html

Ejemplo:
```
DROP DATABASE mydb2;
```

Da error si no existe. Para generar un Warning si no existe se usar `IF EXISTS`. Ejemplo:
```
DROP DATABASE IF EXISTS mydb1
```
NOTA: Recordar que para mostrar las alertas usamos `SHOW WARNINGS`

Con `DROP DATABASE` se elimina todo, tablas, rutinas y disparadores (que ya veremos).
Al acabar la eliminación, muestra los registros que se han quitado, que en realidad son las tablas o archivos .frm.

NOTA: `DROP DATABASE` elimina los archivos (la Base de datos) creado por mysql.
NO ELIMINA otros archivos creados manualmente.

10 TABLAS

10.1 Creación de Tablas

Para crear una tabla se usa la sentencia `CREATE TABLE`. Su sintaxis:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  [(definición_create,...)]
  [opciones_tabla] [sentencia_select]
```

O bien si es una copia de otra tabla:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
   [() LIKE viejo_tbl_name ()];
```

Ejemplo:

```
CREATE DATABASE world_prueba;
USE world_prueba;
CREATE TABLE CountryCode
(
      CountryCode CHAR (3) NOT NULL,
      Language CHAR(30) NOT NULL,
      IsOfficial ENUM ('True', 'False') NOT NULL DEFAULT 'False',
      Percentage FLOAT (3,1) NOT NULL,
      PRIMARY KEY (CountryCode, Language)
)
ENGINE = MyISAM
COMMENT = 'Listado de Idiomas Hablados';
```

Comentarios sobre este ejemplo:

a) Ojo con los nombre de las tablas.
b) CountryCode es del tipo Carácter No Nulo
c) Language igual, permite 30 caracteres
d) Is Offical permite dos valores, predeterminado False.
e) Percentage es de coma flotante, no nulo
f) La Clave principal serán dos campos: CountryCode y Language
g) El Motor es el predeterminado de MySQL, MyISAM
h) Y un comentario que he puesto...

10.1.1 Propiedades de una tabla

En la creación de la tabla podemos añadir todas las opciones que queramos, eso si, sin repetir ninguna. En el ejemplo anterior hemos visto `ENGINE` y `COMMENT`. La sintaxis será:

```
<nombre_opcion> [=] <valor>
```

Entre opciones habrá espacios, no comas. Veamos las opciones mas habituales:

- `ENGINE` → Indica el motor de almacenamiento. Podemos definir motores diferentes para distintas tablas en función de nuestras necesidades.
- `COMMENT` → Comentario, hasta 60 caracteres y entrecomillado simple.
- `DEFAULT [CHARACTER SET | CHATSET]`: Especifica el conjunto de caracteres por defecto de la tabla.
- `DEFAULT COLLATE` → Especifica la ordenación por defecto de la tabla.

10.1.2 Opciones de Columnas

Una tabla debe contener como mínimo un campo o columna. Dicho campo debe tener un nombre y un tipo de datos asociado. Cada columna puede contener distintas opciones.
Las opciones mas frecuentes son:

a) `NULL, NOT NULL.` Las segundas no permiten valores Nulos. Las Claves principales son `NOT NULL` (aunque pongamos lo contrario). El uso de campos que permitan `NULL` reduce algo el tamaño de las tablas y aumenta su rendimiento, aunque en las consultas luego esto se complique. Está muy extendido el uso de `NOT NULL`.
b) `DEFAULT` especifica el valor predeterminado de datos. Si es `NULL`, tomará `NULL`, si es `NOT NULL` y no hay `DEFAULT`, el valor predeterminado del tipo de datos. `DEFAULT` cumple un papel muy importante en Claves Foráneas, al añadir `SET DEFAULT` como valor predeterminado a aquellos registros sin rellenar.
c) `AUTO_INCREMENT`, usado normalmente para claves primarias. Solo se usa para campos INT, y deben tener restricciones `UNIQUE` o `PRIMARY KEY`.
   En `MyISAM` puede estar en indices compuestos. En el resto de motores, será única para indices completos. Sólo puede haber uno por tabla.
   El incremento será de 1, aunque esto puede cambiarse con:
   - `auto_increment_increment` (valor de incremento)
   - `auto_increment_offset` (desplazamiento de incremento, o valor i

Veamos un ejemplo de uso de `INCREMENT` para indices compuestos e indices simples...

```
CREATE DATABASE zoo;
USE zoo;
```

Primer caso; indice simple:

```
CREATE TABLE animales_simple (
             id MEDIUMINT NOT NULL AUTO_INCREMENT,
             nombre CHAR(30) NOT NULL,
             PRIMARY KEY (id)
             );
INSERT INTO animales_simple (nombre)
VALUES ('perro'),('gato'),('pingüino'),('atún'),('ballena'),('avestruz');
SELECT * FROM animales_simple;
```

Segundo caso: Indice compuesto:

```
CREATE TABLE animales_compuesto (
             tipo ENUM('pez','mamífero','pájaro') NOT NULL,
             id MEDIUMINT NOT NULL AUTO_INCREMENT,
             nombre CHAR(30) NOT NULL,
             PRIMARY KEY (tipo,id)
);
```

¿que pasa sin pongo lo anterior? Me da un error. Debo usar otro Motor:

```
...
ENGINE myISAM;

INSERT INTO animales_compuesto (tipo,nombre)
VALUES
('mamífero','perro'),('mamífero','gato'), ('pájaro','pingüino'),
('pez','atún'),('mamífero','ballena'),('pájaro','avestruz');
SELECT * FROM animales_compuesto
ORDER BY tipo,id;
```

10.1.3 Restricciones

Una restricción es una regla que define una limitación de las filas a ser guardadas en la tabla. Impiden que el contenido de la tablas sea modificado a menos que el cambio esté incluido dentro de las restricciones definidas por la regla.

Las principales restricciones tienen que ver con las claves foráneas y la integridad referencial, por la cual no puede existir una valor en una tabla derivada si no tenemos dicho valor en la tabla principal.

Mas información: http://dev.mysql.com/doc/refman/5.0/es/innodb-foreign-key-constraints.html

Los principales tipos de restricciones son:

- `PRIMARY KEY`: los valores de una columna no pueden ser NULL y la combinación de valores será unica en toda la tabla. Con dicha clave se identifica una fila de manera única.
- `UNIQUE`: Es igual que `PRIMARY KEY` pero permite valores `NULL`.
- `FOREIGN KEY`: Representa la manera de gestionar la integridad referencial entre tablas.

La sintaxis para definir una restricción de clave foránea en InnoDB es así:

```
[CONSTRAINT símbolo] FOREIGN KEY [id] (nombre_índice, ...)
    REFERENCES nombre_de_tabla (nombre_índice, ...)
    [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Las claves foráneas deben cumplir estas condiciones:

- Ambas tablas deben ser InnoDB y no deben ser tablas temporales.
- En la tabla que hace referencia, debe haber un índice donde las columnas de clave extranjera estén listadas en primer lugar, en el mismo orden.
- En la tabla referenciada, debe haber un índice donde las columnas referenciadas se listen en primer lugar, en el mismo orden. En MySQL/InnoDB 5.0, se crea automáticamente.
- Las columnas `BLOB` y `TEXT` no pueden incluirse en una clave foránea, porque los índices sobre dichas columnas siempre deben incluir una longitud prefijada.
- Si se proporciona un CONSTRAINTsímbolo, éste debe ser único en la base de datos. Si no se suministra, InnoDB crea el nombre automáticamente.

InnoDB rechaza cualquier operación `INSERT` o `UPDATE` que intente crear un valor de clave foránea en una tabla hija sin un valor de clave candidata coincidente en la tabla padre.

La acción que InnoDB lleva a cabo para cualquier operación `UPDATE` o `DELETE` que intente actualizar o borrar un valor de clave candidata en la tabla padre que tenga filas coincidentes en la tabla hija depende de la accion referencial especificada utilizando las subcláusulas ON `UPDATE` y `ON DETETE` en la cláusula `FOREIGN KEY`.

Cuando el usuario intenta borrar o actualizar una fila de una tabla padre, InnoDB soporta cinco acciones respecto a la acción a tomar:

- `CASCADE`: Borra o actualiza el registro en la tabla padre y automáticamente borra o actualiza los registros coincidentes en la tabla hija. Tanto O`N DELETE CASCADE` como `ON UPDATE CASCADE` están disponibles en MySQL 5.0. Entre dos tablas, no se deberían definir varias cláusulas `ON UPDATE CASCADE` que actúen en la misma columna en la tabla padre o hija.
- `SET NULL`: Borra o actualiza el registro en la tabla padre y establece en NULL la o las columnas de clave foránea en la tabla hija. Esto solamente es válido si las columnas de clave foránea no han sido definidas como `NOT NULL`. MySQL 5.0 soporta tanto `ON DELETE SET NULL` como `ON UPDATE SET NULL`.
- `NO ACTION`: En el estándar ANSI SQL-92, NO ACTION significa ninguna acción en el sentido de que unintento de borrar o actualizar un valor de clave primaria no sera permitido si en la tabla referenciada hay una valor de clave foránea relacionado. (Gruber, Mastering SQL, 2000:181). En MySQL 5.0, InnoDB rechaza la operación de eliminación o actualización en la tabla padre.
- `RESTRICT`: Rechaza la operación de eliminación o actualización en la tabla padre. `NO ACTION` y `RESTRICT` son similares en tanto omiten la cláusula `ON DELETE` u `ON UPDATE`. (Algunos sistemas de bases de datos tienen verificaciones diferidas o retrasadas, una de las cuales es NO ACTION. En MySQL, las restricciones de claves foráneas se verifican inmediatamente, por eso, NO ACTION y RESTRICT son equivalentes.)
- `SET DEFAULT`: Esta acción es reconocida por el procesador de sentencias (parser), pero InnoDB rechaza definiciones de tablas que contengan `ON DELETE SET DEFAULT` u `ON UPDATE SET DEFAULT`.

10.1.4 SHOW CREATE TABLE

Con esta sentencia podemos saber cómo se ha creado una tabla determinada.
Veamos un ejemplo:

```
USE world;
SHOW CREATE TABLE City \G;
```

10.1.5 Crear Tablas en base a Tablas existentes

Se puede hacer mediante dos formas:

a) A través de un SELECT con la estructura CREATE TABLE... AS SELECT

NOTA: El AS es opcional en MySQL pero obligatorio en el estándar SQL.

b) Usando sólo la estructura mediante CREATE TABLE... LIKE

Veamos un ejemplo de cada uno...
Además, aprovecharemos para hacerlo en una base de datos externa.
(si se hace en la misma base de datos, se omite el nombre de la bbdd)

```
USE world_prueba;
CREATE TABLE CityCopia1
AS SELECT * FROM world.City;
```

Podemos modificar evidentemente el SELECT:

```
CREATE TABLE CityCopia2
AS SELECT * FROM world.City
WHERE Population > 2000000;
```

Si sólo queremos crear una copia vacia:

```
CREATE TABLE CityCopia3
AS SELECT * FROM world.City
LIMIT 0;
```

Una de las ventajas de este método es que podemos copiar únicamente unos campos determinados. Por ejemplo:

```
CREATE TABLE CityCopia4
AS SELECT Id, Name, CountryCode FROM world.City;
```

Para visualizarlo (solo 20 registros):

```
SELECT * FROM CityCopia4 LIMIT 20;
```

Por otro lado, hay que reseñar que `AS SELECT ...LIMIT 0` y `LIKE` dan resultados diferentes. Mejor lo vemos con un ejemplo:

```
CREATE TABLE t
(
      i INT NOT NULL AUTO_INCREMENT,
      PRIMARY KEY (i)
) ENGINE = InnoDB;
```

Ahora hagamos una copia y saquemos la estructura:

```
CREATE TABLE copiat1 SELECT * FROM t LIMIT 0;
CREATE TABLE copiat2 LIKE t;

SHOW CREATE TABLE copiat1 \G;
SHOW CREATE TABLE copiat2 \G;
```

Es mas completo el `LIKE`, puesto que el `AS... SELECT` no conserva las definiciones de `PRIMARY KEY`, ni `AUTO_INCREMENT`, ni claves foráneas, así como tampoco el motor de Almacenamiento, usando el predeterminado (en nuestro caso si coincide, puesto que es, en ambos casos, InnoDB).

10.1.6 Tablas Temporales

Si trabajamos con tablas muy grandes, solemos necesitar trabajar ocasionalmente con un pequeño conjunto de datos dentro de esa gran tabla. Si cada vez hemos de ejecutar una consulta completa sobre la tabla podemos cargar mucho el servidor y ralentizar el sistema, por lo que podemos intentar trabajar en MySQL con tablas temporales que tendrán este pequeño conjunto de datos de la tabla madre, que serán mucho más pequeñas y más rápidas.

Para crear una tabla temporal solo debemos agregar TEMPORARY a la sentencia `CREATE TABLE`. Veamos un ejemplo:

```
CREATE TEMPORARY TABLE libros_temp
(
      id INT(11) NOT NULL AUTO_INCREMENT,
      fecha DATE  NOT NULL DEFAULT '0000-00-00',
      titulo VARCHAR(20),
      comentarios varchar(50),
      PRIMARY KEY(id)
) ;

SHOW TABLES;
```

Vemos que no aparece listado.

```
INSERT INTO libros_temp (fecha, titulo, comentarios)
VALUES ('2012-11-29','Fundación','Premio Especial Hugo');

SELECT * FROM libros_temp;
```

Salimos y entramos:

```
exit
mysql -u root -p
USE world_prueba
SELECT * FROM libros_temp;
```

La tabla temporal existirá mientras la conexión este abierta si es que nosotros no la destruimos antes. Una vez cerrada la conexión la tabla será destruida y el espacio que ocupaba (en memoria o disco) será liberado.

Si creamos una tabla que tiene el mismo nombre que una existente ya en la base de datos, la que existe quedará oculta y trabajaremos sobre la temporal.

Tablas tipo HEAP

MySQL también permite especificar que una tabla temporal sea creada en memoria si dicha tabla se declara con el motor `HEAP`:

```
CREATE TEMPORARY TABLE libros_temp
(
      id int(11) NOT NULL auto_increment,
      fecha date NOT NULL default '0000-00-00',
      titulo text,
      Comentarios varchar(50),
      PRIMARY KEY(id)
)ENGINE = 'HEAP';
```

Una tabla no puede tener más de 1600 campos (esto viene limitado por el hecho que el máximo tamaño de una tupla debe ser menor que 8192 bytes), pero este límite puede ser configurado a un tamaño menor en algunos sitios. Una tabla no puede tener el mismo nombre que una tabla del sistema.

A diferencia de una tabla `TEMPORARY`, que solo puede ser accedida por el usuario que la crea, una tabla `HEAP` puede ser utilizada por diversos usuarios. No soportan columnas de auto_increment ni que haya valores nulos en los índices. Los datos son almacenados en pequeños bloques.

10.2 Modificar Tablas

En esta sección veremos como modificar la estructura y datos de una tabla.

10.2.1 Agregar Columnas

Se usa la sentencia `ALTER TABLE … ADD COLUMN.`
La sintaxis es:

```
ALTER TABLE nombre_tabla
ADD_COLUMN nombre_campo [OPCIONES],
ADD_COLUMN nombre_campo [OPCIONES] …
```

Veamos un ejemplo.

```
USE world_prueba;
CREATE TABLE CopiaCity LIKE world.City;
DESC CopiaCity;

ALTER TABLE CopiaCity
ADD COLUMN LocalName VARCHAR(35)
      CHARACTER SET utf8 NOT NULL
      DEFAULT ''
      COMMENT 'Nombre local de esta ciudad';
```

De nuevo vemos la estructura:

```
DESC CopiaCity;
```

10.2.2 Eliminar Columnas

Se realiza mediante la sentencia `DROP COLUMN`. Sintaxis:

```
ALTER TABLE nombre_tabla
DROP COLUMN nombre_campo;
```

Ejemplo:

```
ALTER TABLE City
DROP COLUMN LocalName;
```

10.2.3 Modificar las definiciones de las Tablas

Se usa la sentencia `MODIFY` y hay que especificar el nombre del campo seguido de la nueva definición. Su sintaxis será:

```
ALTER TABLE nombre_tabla
MODIFY nombre_campo TIPO [OPCIONES]
```

Por ejemplo:

```
ALTER TABLE City
MODIFY LocalName VARCHAR(50) NOT NULL DEFAULT 'Pon Nombre';
```

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/alter-table.html

Mas ejemplos:

```
ALTER TABLE City
ADD Column Opciones ENUM ('A','B');

ALTER TABLE City
MODIFY Opciones ENUM ('C','D');

INSERT INTO City
(Name, CountryCode, District, Population, LocalName, Opciones)
VALUE
('Sevilla','ESP','Andalucia',700000,'Hispalis','C');
```

Si hacemos

```
ALTER TABLE City
MODIFY Opciones ENUM ('C','D');
```

El dato de Opciones se trunca, y se pone a `NULL`

10.2.4 Cambiar Columnas

Se usa la sentencia `CHANGE`. La sintaxis será:

```
ALTER TABLE nombre_tabla
CHANGE nombre_anterior nombre_posterior OPCIONES;
```

Por ejemplo:

```
ALTER TABLE City
CHANGE Opciones Alternativas ENUM ('A','B','C','D');

ALTER TABLE City
CHANGE Alternativas Alternativas ENUM ('A','B','C','D') NOT NULL;
```

10.2.5 Renombrar Tablas

En este caso tenemos dos alternativas:

1. Usar `ALTER TABLE... RENAME TO`. Solo permite un cambio. Su sintaxis es:

```
ALTER TABLE nombre_anterior
RENAME TO nombre_posterior;
```

Ejemplo:

```
ALTER TABLE CityCopia4 RENAME TO City4;
```

2. Usar `RENAME TABLE...TO`. Permite varios cambios a la vez. Su sintaxis es:

```
RENAME TABLE nombre_anterior TO nombre_posterior,
nombre_anterior1 TO nombre_posterior1,
nombre_anterior2 TO nombre_posterior2;
```

Ejemplo:

```
RENAME TABLE CityCopia1 TO City1,
CityCopia2 TO City2,
CityCopia3 TO City3;
```

10.3 Eliminación de Tablas

Se usa DROP TABLE.
Su sintaxis será:

```
DROP [TEMPORARY] TABLE
[IF EXISTS] Nombre Tabla;
```

Un ejemplo de borrado de varias tablas:

```
DROP TABLE City1, City2, City3;
```

11 MANIPULACIÓN DE DATOS

11.1 La sentencia INSERT.

Para insertar registros (filas) en las tablas de la bbdd usaremos la sentencia INSERT.
Existen 3 variantes de dicha sentencia:

- `INSERT … VALUES`
- `INSERT... SET`
- `INSERT … SELECT`

11.1.1 Sintaxis de INSERT … VALUES

`INSERT...VALUES` se usa para añadir filas desde 0, a partir de valores literales. Sintaxis:

```
INSERT
INTO nombre_tabla [ (columnas) ]
VALUES valores_filas;
```

- Columnas va entre parentesis, siendo los campos donde queremos meter los datos separados por comas. Si no aparece, será el orden que ya lleva la tabla.
- Valores_filas serán los registros, separados por comas y dentro de paréntesis. Ejemplo:

```
USE world;
INSERT INTO City (Name, CountryCode)
VALUES
('Alcalá de Guadaíra','ESP'),
('Carmona','ESP');
```

Hacemos un SELECT para comprobarlo:

```
SELECT * FROM City WHERE CountryCode = 'ESP';
```

Obsérvese como el ID no se ha puesto en el INSERT, añadiendose automáticamente.

11.1.2 Sintaxis de INSERT... SET

Se pueden añadir registros asignando directamente valores a los campos.
Eso si, sólo de uno en uno (por lo que es poco utilizado). Sintaxis:

```
INSERT
INTO nombre_tabla
SET nombre_campos = valor, ...

Veamos un ejemplo:
INSERT INTO City
SET Name = 'La Rinconada',
CountryCode = 'ESP';
```

11.1.3 Sintaxis de INSERT … SELECT

Permite copiar filas a partir de una tabla existente. La sintaxis:

```
INSERT
INTO nombre_tabla [ (campos) ]
expresion_consulta;
```

En primer lugar creamos una copia de la estructura de la tabla City llamada Top10_Ciudades:

```
CREATE TABLE  Top10_Ciudades LIKE City;
```

Y ahora introducimos las 10 ciudades mas pobladas:

```
INSERT INTO Top10_Ciudades (ID, Name, CountryCode, District, Population)
SELECT  ID, Name, CountryCode, District, Population
FROM City
ORDER BY Population DESC
LIMIT 10;
```

Hacemos el SELECT correspondiente:

```
SELECT * FROM  Top10_Ciudades;
```

De todos modos, el SELECT anterior puede reducirse, quitando los campos afectados e introduciendo TODOS los campos con *.
Borramos el contenido de Top10_Ciudades:

```
DELETE FROM Top10_Ciudades;

INSERT INTO Top10_Ciudades
SELECT  *
FROM City
ORDER BY Population DESC
LIMIT 10;
```

11.1.4 INSERT con LAST_INSERT_ID.

Mediante la sentencia LAST_INSERT_ID veremos el primer valor AUTO_INCREMENT que se ha generado al ejecutar una sentencia INSERT previa. En nuestro caso:

```
SELECT * FROM City WHERE CountryCode = 'ESP';
```

Vemos el primer valor AUTO_INCREMENT de la última sentencia:

```
SELECT LAST_INSERT_ID();
```

Mas información: http://mysql.conclase.net/curso/?sqlfun=LAST_INSERT_ID

11.2 La sentencia DELETE

Ya hemos visto la forma de borrar datos de una tabla con DELETE. Sintaxis:

```
DELETE FROM nombre_tabla;
```

Podemos usar la cláusula WHERE para filtrar los registros a borrar.
Por ejemplo: Creamos una copia de CountryLanguage y le quitamos los idiomas a cada país que no sean lenguajes oficiciales.

```
CREATE TABLE LenguasPaises LIKE CountryLanguage;
INSERT INTO LenguasPaises SELECT * FROM  CountryLanguage;
```

Comprobamos las lenguas de España:

```
SELECT * FROM LenguasPaises WHERE CountryCode = 'ESP';
```

Y ahora quitamos las lenguas cooficiales:

```
DELETE FROM LenguasPaises
WHERE IsOfficial = 'F';
SELECT * FROM LenguasPaises WHERE CountryCode = 'ESP';
```

11.2.1 Usar DELETE con ORDER BY y LIMIT

La sentencia DELETE permite las cláusulas ORDER BY y LIMIT. Por ejemplo, vamos a eliminar los registros insertados en City. Primero vamos a ver los que están:

```
SELECT * FROM City WHERE CountryCode = 'ESP';
```

Vamos a asegurarnos de lo que vamos a borrar:

```
SELECT *
FROM City
WHERE CountryCode = 'ESP'
ORDER BY ID DESC
LIMIT 3;
```

Procedemos al borrado:

```
DELETE FROM City
WHERE CountryCode = 'ESP'
ORDER BY ID DESC
LIMIT 3;
```

Y comprobamos como queda:

```
SELECT * FROM City WHERE CountryCode = 'ESP';
```

11.3 La sentencia UPDATE

UPDATE modifica el contenido de las filas existentes. Su sintaxis es:

```
UPDATE nombre_tabla
SET columna=<expresion> [columna=<expresion>,columna=<expresion>...]
[WHERE condición]
[ORDER BY...]
[LIMIT numero_filas]
```

Para el ejemplo, vamos a crearnos una copia de City (de la bbdd world):

```
USE world;
CREATE TABLE Países LIKE Country;
INSERT INTO Países SELECT * FROM  Country;
```

Ahora actualizaremos las ciudades:

```
UPDATE Países
SET Population = Population * 1.1;
```

El cliente responde lo siguiente:

```
Query OK, 232 rows affected (0.05 sec)
Rows matched: 239  Changed: 232  Warnings: 0
```

Esto es debido a que hay 7 paises sin población que no se actualizan.

Ahora veamos un ejemplo con restricciones. Creamos una copia de City (Ciudades)

```
CREATE TABLE Ciudades LIKE City;
INSERT INTO Ciudades SELECT * FROM  City;
```

Vemos la restricción con:

```
SHOW CREATE TABLE Ciudades \G;
```

El ID es clave primaria.
Vamos a ver las 10 primeras ciudades por ID:

```
SELECT *
FROM Ciudades
ORDER BY ID ASC
LIMIT 10;
```

Pues bien, queremos reducir el ID en 1 (así, tendremos 0,1,2,3).

```
UPDATE Ciudades
SET ID = ID-1;
```

Y vemos el resultado:

```
SELECT *
FROM Ciudades
ORDER BY ID ASC
LIMIT 10;
```

Ahora intentamos hacer lo contrario. Vamos a aumentar en 1 el ID
Vamos a activar el modo de actualizaciones seguras (respetando restricciones):
SET SQL_SAFE_UPDATES=1;

```
UPDATE Ciudades
SET ID = ID+1;
```

¿Que pasa? Pues que no nos deja. Quitamos el modo y cambiamos la sentencia UPDATE:
SET SQL_SAFE_UPDATES=0;

```
UPDATE Ciudades
SET ID = ID+1
ORDER BY ID DESC;
```

11.4 La sentencia REPLACE

REPLACE funciona igual que INSERT, solo que puede emplearse en restricciones de campos UNIQUE o PRIMARY KEY. En estos casos, elimina primero la fila “conflictiva” y la reemplaza por el nuevo registro. La sintaxis es:

```
REPLACE INTO nombre_tabla ( <lista_columnas> )
VALUES ( <lista_expresiones> )
```

Por ejemplo (siguiendo con el ejemplo del apartado anterior).

```
SELECT *
FROM Ciudades
WHERE CountryCode = 'ESP';
```

Vamos a intentar INSERTAR en la ID 698 ([San Cristóbal de] la Laguna) un registro para corregir el nombre que le han puesto los finlandeses y actualizar la población:

```
INSERT INTO Ciudades
VALUES
(698,'San Cristóbal de la Laguna','ESP','Canary Islands',153187);
```

El resultado:

```
ERROR 1062 (23000): Duplicate entry '698' for key 'PRIMARY'
```

Forma de solventarlo, usando REPLACE:

```
REPLACE INTO Ciudades
VALUES
(698,'San Cristóbal de la Laguna','ESP','Canary Islands',153187);
```

Y listo. Se quita uno y se pone el otro:

```
SELECT *
FROM Ciudades
WHERE CountryCode = 'ESP';
```

11.5 INSERT con DUPLICATE KEY UPDATE

Es similar a REPLACE, solo que la fila anterior es conservada, realizando una actualización con garantías. Es específica de mysql. La sintaxis es:

```
INSERT INTO nombre_tabla (campos)
[ (columnas) ]
VALUES valores_filas
ON_DUPLICATE KEY UPDATE <expresion>
```

Un ejemplo:

```
USE test;
CREATE TABLE usuarios (
      usuarioID INT UNSIGNED,
      nombreUsuario VARCHAR (100),
      hora_logeo TIMESTAMP,
      visitas INT UNSIGNED DEFAULT 1,
      PRIMARY KEY (usuarioID)
);
INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (100, 'Tobias');
SELECT * FROM usuarios;
```

Tratamos de insertar de nuevo a Tobias por 2ª vez:

```
INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (100, 'Tobias');
```

Y claro, da error, puesto que el INSERT con el PRIMARY KEY dará problemas:

```
ERROR 1062 (23000): Duplicate entry '100' for key 'PRIMARY'
```

Ahora lo intentamos de nuevo, usando REPLACE:

```
REPLACE INTO usuarios (usuarioID, nombreUsuario)
VALUES (100, 'Tobias');
```

Aquí no da error. La entrada anterior ha sido reemplazada y la hora cambiada:

```
SELECT * FROM usuarios;
```

Para evitar esto tenemos una tercera opción, `ON DUPLICATE KEY UPDATE`:

```
INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (100, 'Tobias')
ON DUPLICATE KEY UPDATE visitas = visitas +1;
```

Vemos que ahora no ha dado error y el contador de visitas se ha incrementado:

```
SELECT * FROM usuarios;
```

Con la expresión `ON DUPLICATE KEY UPDATE` controlamos mejor la actualización y gestionamos expresiones en campos concretos que con `REPLACE` o `INSERT INTO`.

11.6 La sentencia TRUNCATE

Es muy similar a `DELETE [TABLE]`, salvo que no permite la cláusula `WHERE`, por lo que es una manera rápida y eficaz de borrar el contenido de una tabla, sin mas. Sintaxis:

```
TRUNCATE [TABLE] <nombre_tabla>
```

Ejemplo:

```
TRUNCATE TABLE usuarios;
SELECT * FROM usuarios;
```

Poner `TABLE` es opcional, aunque recomendable
(así no lo confundimos con la función `TRUNCATE ()`).

12 TRANSACCIONES Y BLOQUEOS

12.1 ¿Que es una Transacción?

Una transacción es una serie de pasos que se ejecutan para manipular datos, los cuales son tratados como una unidad individual de trabajo.

Las transacciones aportan una fiabilidad superior a las bases de datos. Si disponemos de una serie de consultas SQL que deben ejecutarse en conjunto, con el uso de transacciones podemos tener la certeza de que nunca nos quedaremos a medio camino de su ejecución. De hecho, podríamos decir que las transacciones aportan una característica de "deshacer" a las aplicaciones de bases de datos.

Para este fin, las tablas que soportan transacciones, como es el caso de InnoDB, son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor, ya que las consultas se ejecutan o no en su totalidad. Por otra parte, las transacciones pueden hacer que las consultas tarden más tiempo en ejecutarse.

Para comprobar que las transacciones están habilitadas, ejecutamos esta sentencia:

```
SHOW VARIABLES LIKE '%innodb%';
```

Ya con la primera línea está todo dicho:

```
have_innodb       | YES
```

12.1.1 ACID

Los sistemas transaccionales deben ser conformes con ACID, que tiene estas propiedades:

- Atomic (atómica) → Todas las sentencias se ejecutan exitosamente o se cancelan como una unidad.
- Consistent (consistente)→ Una base de datos es consistente cuando una transacción comienza y es dejada en ese mismo estado una vez que termina la transacción.
- Isolated (aislada) → Una transacción no tiene que afectar a otra.
- Durable (durarera) → Todos los cambios que efectua una transacción completada son registrados adecuadamente en la base de datos. Los cambios no se pierden.

El procesamiento transaccional da mas garantías a costa de un consumo mayor de recursos (CPU, Memoria, Espacio en Disco). El ejemplo mas claro de uso son las operaciones financieras, donde la integridad de datos pesa mas que el coste de recursos adicionales.

12.2 Sentencias de control de transacciones.

Las siguientes sentencias son usadas para controlar de forma explícita las transacciones:

- `START TRANSACTION (BEGIN) [WITH CONSISTENT SNAPSHOT]` → Comienza de forma explícita una nueva transacción. Provoca que se realice un `UNLOCK TABLES` implícito. `BEGIN` o `BEGIN WORK` son alias de MySQL (fuera del standard) para START TRANSACTION. La cláusula `WITH CONSISTENT SNAPSHOT` comienza una lectura consistente para motores de almacenamiento capaces de ello.
- `COMMIT` → Da por completada la transacción actual, haciendo los cambios permanantes.
- `ROLLBACK` → Anula la transacción actual y todos los cambios que produce.
- `SET AUTOCOMMIT` → Desactiva o activa el modo autocommit por defecto para la sesión actual.

12.2.1 El Modo AutoCommit

El modo Autocommit determina cómo y cuando serán arrancadas las nuevas transacciones.
Autocommit activado
Si está activado, cada sentencia SQL arranca una nueva transacción. Si la transacción se completa (commited) sin errores, se guardan los cambios. Si hay errores, la transacción se desecha y se deshacen todos los cambios que pudieron resultar de la ejecución.
Para arrancar explícitamente una transacción se usa `START TRANSACTION`

Autocommit desactivado
Si el autocommit no está activado, las transacciones pueden comprender por defecto múltiples sentencias. En este caso, la transacción puede ser completada o desechada mediante el uso de `COMMIT` o `ROLLBACK`, respectivamente. Desde que acaba la transacción con las sentencias anteriores, una nueva transacción arranca de forma automática.

Cuando una sentencia que es ejecutada dentro de la transacción no se completa satisfactoriamente, todos los cambios que podrían haber sido causados por dicha sentencia son deshechos pero la transacción como un todo no termina. La misma permanece abierta en este caso y puede ser completada o desecha como un todo.

Controlar el modo autocommit
Se controla mediante la sentencia AUTOCOMMIT, que está disponible a nivel de sesión. Para desactivarla en la sesión actual:

```
SET AUTOCOMMIT = OFF;
```

Equivalente a poner: `SET SESSION AUTOCOMMIT = OFF;`
También se puede usar una notación de variable, donde 1 activa y 0 desactiva:

```
SET @@autocommit :=0;
```

Para ver como está el modo autocommit, podemos usar un SELECT:

```
SELECT @@autocommit;
```

Cambiar configuración de autocommit por defecto
De forma predeterminada autocommit está activado. Para desactivarlo a nivel de servidor (no de sesión) debemos usar el siguiente procedimiento:

1. Abrimos el archivo de configuración de mysql:
   `sudo kate /etc/mysql/my.cnf`
2. Nos vamos a la sección [mysqld] y añadimos:
   `init_connect='SET autocommit=0'`
3. Reiniciamos el servidor

Es válido para usuarios “normales” sin privilegios SUPER (como root).
Mas información: http://dev.mysql.com/doc/refman/5.0/en/server-system-variables.html

12.2.2 Sentencias que ocasionan un COMMIT implícito

La sentencia COMMIT siempre completa explicitamente la transacción actual. Hay otras sentencias de control de transacciones que lo hacen implicitamente: `START TRANSACTION` y `SET AUTOCOMMIT :=1.`

Existen otras sentencias que no pueden desecharse si son completadas con éxitos:

- Definición de datos (`ALTER, CREATE, DROP`)
- Gestión de usuarios (`GRANT, REVOKE, SET PASSWORD`)
- De bloqueo (`LOCK TABLES, UNLOCK TABLES`)

Además, hay otras sentencias que implican un commit implícito: `TRUNCATE TABLE y LOAD DATA INFILE.`

Veamos algunos ejemplos...
Demo1: `ROLLBACK`

```
USE world;
START TRANSACTION;
SELECT * FROM City WHERE ID=656;
```

Ahora borramos:

```
DELETE FROM City WHERE ID=656;
```

Comprobamos:

```
SELECT * FROM City WHERE ID=656;
```

Y ahora deshacemos la transacción:

```
ROLLBACK;
```

Y comprobamos:

```
SELECT * FROM City WHERE ID=656;
```

¿Que pasa si no iniciamos una transacción?

```
INSERT INTO City
VALUES (4080, 'Alcalá de Guidaíra', 'ESP', 'Andalusia', 70000);
```

Comprobamos:

```
SELECT * FROM City WHERE ID=4080;
```

Borramos:

```
DELETE FROM City WHERE ID=4080;
```

E intentamos deshacer:

```
ROLLBACK;
```

Comprobamos de nuevo:

```
SELECT * FROM City WHERE ID=4080;
```

Y nada de nada. El registro se borró definitivamente.

Lo mismo sucedería si comenzamos con `START TRANSACTION;`
Y tras el borrado (DELETE) ejecutamos: `COMMIT;`

Otra prueba mas:

- Realizar el `INSERT INTO...`
- `START TRANSACTION`
- `DELETE...`
- `ROLLBACK`
- `DELETE`
- `COMMIT;`

Mas información: http://dev.mysql.com/doc/refman/5.0/es/commit.html

12.2.3 Búsqueda de un motor de almacenamiento que soporte transaccional

Para ver si tenemos un motor de almacenamiento transaccional en el servidor y está compilado ejecutamos la siguiente sentencia:

```
SHOW ENGINES \G;
```

12.3 Niveles de Aislamiento

Multiples transacciones se pueden dar al mismo tiempo en el servidor. En principio, habrá una transacción por sesión y cliente, pero si se realiza un cambio en los datos, ¿que pasa con el resto de usuarios?

El nivel de aislamiento de la transacción determina el nivel de visibilidad entre las mismas, estos es, la manera en que transacciones simultaneas interactuan al acceder a los mismos datos. Veremos los problemas que pueden ocurrir y como motores de almacenamiento transaccionales (como InnoDB) implementan niveles de aislamiento.

12.3.1 Problemas de consistencia

Cuando múltiples clientes acceden a la vez a los datos de una misma tabla, pueden ocurrir los siguientes problemas de consistencia:

- Lecturas sucias
- Lecturas no repetibles
- Lecturas fantasma

Lecturas Sucias (Dirty reads)
Es el problema más importante de todos. Supone que las transacciones en curso puedan leer el resultado de otras transacciones aún no confirmadas. Por ejemplo, vamos a suponer que tenemos dos transacciones activas (A y B). Inicialmente la transacción A lee un valor X de una tabla que, por ejemplo, es 0. Durante dicha transacción el valor de X se cambia a 10, pero aún la transacción no se ha cometido, por lo que en la tabla X = 0. Ahora la transacción B accede al valor X y obtiene ¡¡X = 10!! un valor que está usando A y que aún no se ha cometido. Supongamos que ahora se anula la transacción A. El resultado sería X = 0 en la tabla y X = 10 en la transacción B por lo que hemos llegado a un estado muy grave de inconsistencia.

Lecturas no repetibles (Non-Repeatable reads)
Ocurre cuando una transacción activa vuelve a leer un dato cuyo valor difiere con respecto al de la anterior lectura. Lo vemos más claro con un ejemplo. Supongamos que una transacción activa, A, lee un valor X = 0. En este momento otra transacción B modifica el valor de X, por ejemplo X = 10, y se comete dicha transacción. Si ahora durante la transacción A se vuelve a leer el valor X obtendríamos 10 en lugar del 0 que se esperaba. Aunque a primera vista este problema no parezca muy importante en realidad sí que lo es, sobre todo cuando X es una clave primaria o ajena. En este caso se puede originar una gran pérdida de consistencia en nuestra base de datos.

Lecturas fantasma (Phantom reads)
Este supone el menor problema que se nos puede plantear con respecto a las transacciones. Sucede cuando una transacción en un momento lanza una consulta de selección con una condición y recibe en ese momento N filas y posteriormente vuelve a lanzar la misma consulta junto con la misma condición y recibe M filas con M > N. Esto es debido a que durante el intervalo que va de la primera a la segunda lectura se insertaron nuevas filas que cumplen la condición impuesta en la consulta.

12.3.2 Cuatro Niveles

Debido a estos problemas el ANSI establece diferentes niveles de aislamiento (isolations levels) para solventarlos. Hay que tener en cuenta que a mayor nivel de aislamiento mayores son los sacrificios que se hacen con respecto a la concurrencia y al rendimiento.

Vamos a enumerar los niveles de aislamiento desde el menor hasta el mayor:

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

Read uncommitted (Lectura sin confirmación)
En la práctica casi no se suele utilizar este nivel de aislamiento ya que es propenso a sufrir todos los problemas anteriormente descritos. En este nivel una transacción puede ver los resultados de transacciones aún no cometidas. Podemos apreciar que en este nivel no existe aislamiento alguno entre transacciones.

Read committed (Lectura confirmada)
Es el predeterminado para la mayoría de gestores de bases de datos relacionales. Supone que dentro de una transacción únicamente se pueden ver los cambios de las transacciones ya cometidas. Soluciona el problema de las lecturas sucias, pero no el de las lecturas no repetibles ni tampoco el de las lecturas fantasmas.

Repeatable read (Lectura repetible)
Define que cualquier tupla leída durante el transcurso de una transacción es bloqueada. De esta forma se soluciona, además de las lecturas sucias, el problema de las lecturas no repetibles. Aunque en dicho nivel se siguen dando las lecturas fantasmas.

Serializable (Lecturas en serie)
Soluciona todos los problemas descritos. Para ello ordena las transacciones con el objetivo de que no entren en conflicto. Este nivel de aislamiento es bastante problemático ya que es, con diferencia, el que más sacrifica en rendimiento y concurrencia.

Resumiendo en esta tabla:

Nivel de Aislamiento	Lectura Sucia	Lectura No Repetible	Lectura Fantasma
Read uncommitted	Imposible	Posible	Posible
Read Committed	Imposible	Posible	Posible
Repeatable read	Imposible	Imposible	Imposible (para InnoDB)
Serializable	Imposible	Imposible	Imposible

Cambiar configuración Nivel Aislamiento por defecto
Debemos usar el siguiente procedimiento si queremos cambiarlo a `REPEATABLE-READ`:

1. Abrimos el archivo de configuración de mysql:
   `sudo kate /etc/mysql/my.cnf` En Windows: `C:/xampp/mysql/bin/my.ini`
2. Nos vamos a la sección [mysqld] y añadimos:
   `transaction-isolation = REPEATABLE-READ`
3. Reiniciamos el servidor.

Mas información: http://dev.mysql.com/doc/refman/5.0/en/set-transaction.html

También podemos hacerlo de forma dinámica usando la sentencia `SET TRANSACTION ISOLATION LEVEL`. Su sintaxis será:
`SET [GLOBAL|SESION]  TRANSACTION ISOLATION LEVEL <nivel_aislamiento>`
donde <nivel_aislamiento> será:
`[READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE]`
De forma predeterminada es REPEATABLE READ.

Un ejemplo:
```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Si queremos que sea definida para las conexiones SIGUIENTES, debemos usar GLOBAL:
```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Los usuarios normales pueden cambiar a nivel de `SESSION`; solo aquellos con privilegios `SUPER` pueden cambiar a nivel `GLOBAL`. Para ver el estado actual se usa tx_isolation:
```
SELECT @@tx_isolation;
```
Para ver a nivel de sesión y a nivel global:
```
SELECT @@global.tx_isolation, @@session.tx_isolation;
```

Vamos a realizar una práctica con dos sesiones. Usaremos tablas `InnoDB` con el `AUTOCOMMIT` activado. Pongo ambas sesiones en ambos lados mas abajo
```
SET @@autocommit :=1;
```

`mysql> PROPMT s1>` `s1> SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;`	`mysql> PROMPT s2>` `s2>`
`s1> SELECT @@tx_isolation;` `s1> USE world;`	`s2> USE world;` `s2>START TRANSACTION;`
	`s2>INSERT INTO City (Name, CountryCode, Population) VALUES ('Camas','ESP',65000);`
`s1> SELECT * FROM City WHERE Name = 'Camas';` ¡Sale vacio! Porque el nivel de aislamiento no permite ver cambios hasta no ser completados.	
	`s2> COMMIT;`
`s1> SELECT * FROM City WHERE Name = 'Camas';` Ahora si tenemos el registro.	
`s1> PROMPT`	`s2> PROMPT`

12.4 Bloqueos

12.4.1 Conceptos relacionados con el bloqueo

El bloqueo es un mecanismo que previene la ocurrencia de problemas durante el acceso de varios clientes a la vez. Son manejados por el servidor, quien bloquea los datos en nombre de un cliente, para restringir el acceso al resto. El cliente "principal" si puede moficiar los datos, el resto debe esperar su turno.

Se pueden dar las siguientes condiciones de conflicto:

- Si un cliente quiere leer datos y otros quieren leer el mismo dato, no hay problema. El problema llega al escribir (modificar) un dato. En este caso, deberá esperar la lectura de los demás.
- Si no hay lecturas, y un cliente quiere escribir un dato, los demás deben esperar a que acabe para realizar una lectura u otra escritura.

12.4.2 Bloqueos de lectura

Hay dos modificadores de bloqueo que pueden añadirse al final de una sentencia SELECT:

- `LOCK IN SHARE MODE`
- `FOR UPDATE`

Mas información: http://dev.mysql.com/doc/refman/5.0/es/innodb-locking-reads.html

`LOCK IN SHARE MODE`: Es un bloqueo compartido, de forma que ninguna transacción puede realizar un bloqueo exclusivo.
`FOR UPDATE`: Se bloquea cada registro en modo exclusivo, permitiendo solo la lectura.

Bloqueo en Modo Compartido (LOCK IN SHARE MODE)
Veamos un ejemplo ilustrativo usando integridad referencial en la aplicación:

```
USE world;
SELECT * FROM City
WHERE  CountryCode = 'AUS';
```

¿Se puede agregar un registro en City para Australia de forma segura si alguien borra a Australia en la tabla "padre" de Country? Evidentemente no. Para bloquearlo debemos usar `LOCK IN SHARE MODE`. Ejemplo:

```
SELECT * FROM Country
WHERE Code = 'AUS'
LOCK IN SHARE MODE \G;
```

Bloqueo en Modo Exclusivo (FOR UPDATE)

FOR UPDATE se emplea para realizar bloqueos exclusivos, mucho mas restrictivos que **LOCK IN SHARE MODE.** Pongamos un ejemplo:
Se tiene un campo contador, entero, en una tabla llamada child_codes que se emplea para asignar un identificador único a cada registro hijo agregado a la tabla hijo. Obviamente, utilizar una lectura consistente o una lectura en modo compartido para leer el valor actual del contador no es una buena idea, puesto que dos usuarios de la base de datos pueden ver el mismo valor del contador, y agregar registros hijos con el mismo identificador, lo cual generaría un error de clave duplicada.

En este caso, LOCK IN SHARE MODE no es una buena solución porque si dos usuarios leen el contador al mismo tiempo, al menos uno terminará en un deadlock cuando intente actualizar el contador.

En este caso, hay dos buenas formas de implementar la lectura e incremento del contador: (1), actualizar el contador en un incremento de 1 y sólo después leerlo, o (2) leer primero el contador estableciendo un bloqueo FOR UPDATE, e incrementándolo luego.

Veamos que es bloqueo mortal (o deadlock).
NOTA: Vamos a usar una copia de Country llamada Países. Si no está ya hecha se realiza con la sentencia:

```
CREATE TABLE Países LIKE Country;
INSERT INTO Países SELECT * FROM  Country;
```

```
mysql> PROPMT s1>
s1> USE world;

s1> START TRANSACTION;

s1> UPDATE Países
SET name = 'Paises Bajos'
WHERE code = 'HOL';

s1> DELETE FROM Países
WHERE code = 'ESP';
```

(Se queda esperando)

```
ERROR 1205 (HY000): Lock wait
timeout exceeded; try restarting
transaction

s1> PROMPT
```

```
mysql> PROMPT s2>
s2> USE world;

s2> START TRANSACTION;

s2> UPDATE Países
SET name = 'Reino de España'
WHERE code = 'ESP';

s2 > UPDATE Países
SET Population = 1
WHERE code = 'HOL';

s2> PROMPT
```

Se cansa de esperar y muestra este mensaje:

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/innodb-deadlock-detection.html

12.5 SAVEPOINT y ROLLBACK TO SAVEPOINT

Ya hemos visto el uso de `START TRANSACTION`, `COMMIT` y `ROLLBACK`.

Dentro de MySQL existen además otros comandos específicos para el manejo de transacciones: `SAVEPOINT` y `ROLLBACK TO SAVEPOINT`.
Su sintaxis es:

```
SAVEPOINT identificador
ROLLBACK TO SAVEPOINT identificador
```

El comando `SAVEPOINT` crea un punto dentro de una transacción con un nombre. Si la transacción actual tiene un punto con el mismo nombre, el antiguo se borra y se crea el nuevo.

Por su parte, el comando `ROLLBACK TO SAVEPOINT` deshace una transacción hasta el punto nombrado. Las modificaciones que la transacción actual hace al registro tras el punto se deshacen en el rollback, pero InnoDB no libera los bloqueos de registro que se almacenaron en memoria tras el punto. Los puntos creados tras el punto nombrado se borran.

Hay que tener en cuenta que para un nuevo registro insertado, la información de bloqueo se realiza a partir del ID de transacción almacenado en el registro; el bloqueo no se almacena separadamente en memoria. En este caso, el bloqueo de registro se libera al deshacerse todo.

Veamos un ejemplo:

```
USE world;
START TRANSACTION;

SELECT Name FROM Ciudades WHERE CountryCode = 'ESP';
UPDATE Ciudades SET Name = 'Alicante' WHERE Name = 'Alicante [Alacant]';
```

Ahora creamos un punto de guardado (que va después del UPDATE de Alicante)

```
SAVEPOINT punto_ciudades;
UPDATE Ciudades SET Name = 'Lérida' WHERE Name = 'Lleida (Lérida)';
SELECT Name FROM Ciudades WHERE CountryCode = 'ESP';
```

Ahora hacemos un Rollback hasta el punto de guardado:

```
ROLLBACK TO SAVEPOINT punto_ciudades;
```

Y comprobamos los cambios:

```
SELECT Name FROM Ciudades WHERE CountryCode = 'ESP';
```

Vemos que Alicante ha cambiado pero que Lleida no.

Mas información:
Recomiendo la lectura de este interesante PDF sobre el manejo de Concurrencias en MySQL.
http://basmoxo.files.wordpress.com/2011/10/manejo-de-concurrencia-en-mysql.pdf

12.6 Administración de Bloqueos y Transacciones

12.6.1 LOCK TABLES y UNLOCK TABLES

Para bloquear y desbloquear tablas usaremos LOCK TABLES y UNLOCK TABLES. Sintaxis:

```
LOCK TABLES
    nombre_tabla [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
    [, nombre_tabla [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...
UNLOCK TABLES
```

LOCK TABLES bloquea tablas para el flujo actual. Si alguna de las tablas la bloquea otro flujo, bloquea hasta que pueden adquirirse todos los bloqueos. UNLOCK TABLES libera cualquier bloqueo realizado por el flujo actual. Todas las tablas bloqueadas por el flujo actual se liberan implícitamente cuando el flujo realiza otro LOCK TABLES, o cuando la conexión con el servidor se cierra.

Un bloqueo de tabla protege sólo contra lecturas inapropiadas o escrituras de otros clientes. El cliente que tenga el bloqueo, incluso un bloqueo de lectura, puede realizar operaciones a nivel de tabla tales como DROP TABLE.

Hay que tener en cuenta lo siguiente a pesar del uso de LOCK TABLES con tablas transaccionales:

- LOCK TABLES no es una operación transaccional y hace un commit implícito de cualquier transacción activa antes de tratar de bloquear las tablas. También, comenzar una transacción (por ejemplo, con START TRANSACTION) realiza un UNLOCK TABLES implícito.

- La forma correcta de usar LOCK TABLES con tablas transaccionales,como InnoDB, es poner AUTOCOMMIT = 0 y no llamar a UNLOCK TABLES hasta que hace un commit de la transacción explícitamente. Cuando llama a LOCK TABLES, InnoDB internamente realiza su propio bloqueo de tabla, y MySQL realiza su propio bloqueo de tabla. InnoDB libera su bloqueo de tabla en el siguiente commit, pero para que MySQL libere su bloqueo de tabla, debe llamar a UNLOCK TABLES.
  No debemos tener AUTOCOMMIT = 1, porque entonces InnoDB libera su bloqueo de tabla inmediatamente tras la llamada de LOCK TABLES, y los deadlocks pueden ocurrir fácilmente (hay que tener en cuenta que en MySQL 5.0, no adquirimos el bloqueo de tabla InnoDB en absoluto si AUTOCOMMIT=1, para ayudar a aplicaciones antiguas a envitar deadlocks).

- ROLLBACK no libera bloqueos de tablas no transaccionales de MySQL.

Para usar LOCK TABLES en MySQL 5.0, debemos tener el permiso LOCK TABLES y el permiso SELECT para las tablas involucradas.
La razón principal para usar LOCK TABLES es para emular transacciones o para obtener más velocidad al actualizar tablas.

Si un flujo obtiene un bloqueo READ en una tabla, ese flujo (y todos los otros) sólo pueden leer de la tabla. Si un flujo obtiene un bloqueo WRITE en una tabla, sólo el flujo con el bloqueo puede escribir a la tabla. El resto de flujos se bloquean hasta que se libera el bloqueo.

La diferencia entre READ LOCAL y READ es que READ LOCAL permite comandos INSERT no conflictivos (inserciones concurrentes) se ejecuten mientras se mantiene el bloqueo. Sin embargo, esto no puede usarse si va a manipular los ficheros de base de datos fuera de MySQL mientras mantiene el bloqueo. Para tablas InnoDB , READ LOCAL esencialmente no hace nada: No bloquea la tabla. Para tablas InnoDB , el uso de READ LOCAL está obsoleto ya que una SELECT consistente hace lo mismo, y no se necesitan bloqueos.

Cuando usamos LOCK TABLES, debemos bloquear todas las tablas que se van a usar en las consultas. Mientras los bloqueos obtenidos con un comando LOCK TABLES están activos, no se puede acceder a ninguna tabla que no estuviera bloqueada por el comando. Además, no podemos usar una tabla bloqueada varias veces en una consulta; para evitarlo, mejor usar alias para ello. Hay que tener en cuenta que en este caso, debe tener un bloqueo separado para cada alias.

El bloqueo WRITE normalmente tiene mayor prioridad que el bloqueo READ, para asegurar que las actualizaciones se procesan tan más pronto como sea posible. Esto significa que si un hilo obtiene un bloqueo READ y después otro hilo requiere un bloqueo WRITE, subsiguientes peticiones de bloqueos READ esperarán hasta que el hilo que solicitó el bloqueo WRITE lo obtenga y lo libere. Se puede usar el bloqueo LOW_PRIORITY WRITE para permitir a otros hilos obtener bloqueos READ mientras el hilo espera al bloqueo WRITE. Sólo se debe usar el bloqueo LOW_PRIORITY WRITE si se está seguro de que existirá un tiempo en el que no habrá hilos que pidan un bloqueo READ.

LOCK TABLES trabaja del modo siguiente:

- Ordena todas las tablas a bloquear en un orden interno definido (desde el punto de vista del usuario el orden es indefinido).
- Si una tabla está bloqueada para lectura y escritura, pone el bloqueo de escritura antes que el de lectura.
- Bloquea una tabla cada vez hasta que el hilo obtenga el bloqueo para todas las tablas.

Esta política asegura que el bloqueo de tabla está libre de puntos muertos. Sin embargo hay otras cosas a tener en cuenta con este esquema:

Si se está usando un bloqueo LOW_PRIORITY WRITE para una tabla, sólo significa que MySQL esperará para este bloqueo particular hasta que no existan hilos que quieran un bloqueo READ. Cuando el hilo ha obtenido el bloqueo WRITE y está esperando para obtener el bloqueo para la siguiente tabla en la lista de tablas a bloquear, el resto de los hilos esperarán a que el bloqueo WRITE se libered. Si esto se convierte en un problema serio para una aplicación, se debe considerar la conversión de algunas tablas a tablas de transacción segura.

Se puede matar un hilo de forma segura cuando está esperando un bloqueo de tabla usando KILL. No se debe bloquear ninguna tabla para la que se esté usando , porque en ese caso se ejecuta en un hilo separado.

Normalmente, no se deben bloquear tablas, ya que todas las sentencias simples son atómicas; ningún otro hilo puede interferir con otro que esté ejecutando actualmente una sentecia SQL. Hay unos pocos casos en los que puede ser necesario bloquear tablas:

– Si se van a realizar muchas operaciones en un puñado de tablas, es mucho más rápido bloquear las tablas que se van a usar. La desventaja es, por supuesto, que ningún hilo puede actualizar una tabla bloqueada para lectura (incluyendo aquel que tiene el bloqueo) y ningún hilo puede leer una tabla bloqueada para escritura salvo aquella que tiene el bloqueo. El motivo por el que algunas cosas son más rápidas con LOCK TABLES es que MySQL no vuelca en disco el caché de claves para las tablas bloqueadas hasta que se llama a UNLOCK TABLES (normalmente, el caché de claves se excribe en disco después de cada sentencia SQL).
  Esto acelera la inserción, actualización o borrado en tablas MyISAM.
– Si se usa un motor de almacenamiento en MySQL que no soporte transacciones, se debe usar LOCK TABLES si se quiere asegurar que ningún otro hilo entra entre una sentencia y una .

En el siguiente ejemplo hace falta usar un LOCK TABLES para que se ejecute de forma segura:
Imaginemos que tenemos dos usuarios:

```
PROMPT s1>
s1> USE world;
CREATE TABLE usuarios (
      usuarioID INT UNSIGNED,
      nombreUsuario VARCHAR (100),
      hora_logeo TIMESTAMP,
      visitas INT UNSIGNED DEFAULT 1,
      PRIMARY KEY (usuarioID)
) ENGINE = MyISAM;
```

El cliente1 bloquea la tabla para realizar inserciones:

```
s1> LOCK TABLES usuarios AS u1 WRITE, usuarios AS u2 READ;

s1> INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (100, 'Tobias');
```

Ahora entra cliente2:

```
PROMPT s2>
s2> USE world;
s2> INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (101, 'Juan');
```

Se queda bloqueado sin insertar nada...

El cliente uno inserta otro registro:

```
S1> INSERT INTO usuarios (usuarioID, nombreUsuario)
VALUES (102, 'Pedro');
```

Y desbloquea la tabla:

```
UNLOCK TABLES;
```

Al fin el Cliente2 ve como su inserción se cumple:

```
s2>...
Query OK, 1 row affected (59.75 sec)
```

Y ahora realiza una consulta:

```
s2> SELECT * FROM usuarios;
```

Y ve como su inserción va después del ID 102 que ha hecho el Cliente1.

También se pueden resolver algunos casos de bloqueos usando las funciones de bloqueo a nivel de usuario `GET_LOCK` y `RELEASE_LOCK`.
Estos bloqueos se guardan en una tabla hash en el servidor y se implementan mediante `pthread_mutex_lock()` y `pthread_mutex_unlock()` para mayor velocidad.

Se pueden bloquear todas las tablas en todas las bases de datos con bloqueo de lectura con el comando `TABLES WITH READ LOCK`. Este es un modo muy conveniente para hacer copias de seguridad si se posee un sistema de ficheros, como Veritas, que puede hacer instantaneas regularmente.

NOTA: `LOCK TABLES` no es seguro a nivel de transacción y realizará implícitamente cualquier transacción activa antes de intentar bloquear las tablas.

Mas información:
http://www.profesionalhosting.com/soporte-en-linea/comandos-transaccionales-y-de-bloqueo-de-mysql-preg208.html

12.6.2 Otros comandos de Administración

Presentar todos los bloqueos y transacciones del InnoDB
```
SHOW ENGINE INNODB STATUS
```

La salida es enorme, por lo que es mejor realizarla con un tee (uso el formato breve)
En primer lugar hacemos un tee y definimos la salida:
```
\T salida.txt
```
(En Linux va a /home/usuario/ y Windows en C:\Documents and Settings\nombreUsuario)
```
SHOW ENGINE INNODB STATUS;
```
Ahora paramos la salida y comprobamos lo obtenido:
```
\t
exit
sudo kate \home\ivan-htpc\salida.txt
```

Ver el estado de MySQL
Para ver los procesos actuales:
```
SHOW PROCESSLIST;
+----+------+-----------+-------+---------+------+-------+------------------+
| Id | User | Host      | db    | Command | Time | State | Info             |
+----+------+-----------+-------+---------+------+-------+------------------+
|  2 | root | localhost | world | Query   |    0 | NULL  | SHOW PROCESSLIST |
+----+------+-----------+-------+---------+------+-------+------------------+
1 row in set (0.05 sec)
```

Por otro lado tenemos SHOW STATUS, que proporciona información de estado del servidor.
Su sintaxis es:
```
SHOW [GLOBAL | SESSION] STATUS [LIKE 'pattern']
```

Con una cláusula LIKE , el comando muestra sólo las variables que coinciden con el patrón:
Por ejemplo, para ver información sobre los hilos de ejecución:
```
SHOW STATUS LIKE 'Threads%';
```

Su significado es:

- `Threads_cached`: Número de hilos en la cache de hilos, no tiene porque coincidir con thread_cache_size si se están usando.
- `Threads_connected`: Conexiones actualmente activas, lo que en SHOW PROCESSLIST seria el count final: Incluye todas las conexiones activas estén Running, Sleep, Sorting, Connect…
- `Threads_created`: Número de hilos creados, lo que indica que ha fallado la cache de hilos. Podemos establecer un ratio respecto a Connections para saber si debemos ampliarla
- `Threads_running`: Número de hilos que no están en espera, por lo tanto en ejecución

Mas información sobre comandos DBA:
http://systemadmin.es/category/dba

13 JOINS

13.1 ¿Que es un Join?.

Cuando es necesario trabajar en datos almacenados en múltiples tablas, se puede usar un join. Un join es una operación sobre dos tablas que combina filas de múltiples tablas en nuevas filas, construyendo con ello una nueva tabla resultante.

13.1.1 La limitación de las consultas de una sola tabla

Si seguimos el ejemplo de la bbdd World, podemos plantearnos la siguiente cuestión: ¿en qué países del mundo hay una ciudad llamada Córdoba -de cierta importancia se supone-?

Veamoslo:

```
USE world;
SELECT Name, CountryCode, Population
FROM City
WHERE Name = 'Córdoba';
```

Vaya, en el CountryCode nos sale el código del país, pero no el nombre. Para averiguarlo debemos realizar la siguiente sentencia.

```
SELECT Code, Name, Continent, Population
FROM Country
WHERE Code IN ('ARG','ESP','MEX');
```

Obsérvese que hemos necesitado dos consultas que podemos unir para sacar la misma información en una sola. Esto es lo que se hace con un JOIN.

13.1.2 Combinar dos tablas simples

Para ver cómo unir Country y City vamos a crearnos versiones simplificadas de las mismas con los `SELECT` que ya hemos visto.

```
CREATE TABLE CiudadSimple AS
SELECT Name AS NombreCiudad, CountryCode AS CodPais, Population AS
Población
FROM City WHERE Name LIKE 'Córdoba';
```

Hacemos lo mismo para los países que nos han salido:

```
CREATE TABLE PaísSimple AS
SELECT Code AS Codigo, Name AS NombrePais, Population AS PobPaís
FROM Country WHERE Code IN ('ARG','ESP','MEX');
```

Comprobamos ambos:

```
SELECT * FROM CiudadSimple;
SELECT * FROM PaísSimple;
```

¿Nos hacemos una idea de como pegar ambas tablas?

13.1.3 Producto cartesiano

En el producto cartesiano realizamos el producto de ejecutar ambas tablas creadas. Su nombre se debe al filósofo y matemático francés René Descartes.

El resultado de ejecutar el producto cartesiano entre las tablas anteriores será el siguiente:

PRODUCTO CARTESIANO					
NombreCiudad	CodPais	Poblacion	Codigo	NombrePaís	PobPaís
Córdoba	ARG	1157507	ARG	Argentina	37032000
Córdoba	ESP	311708	ARG	Argentina	37032000
Córdoba	MEX	176952	ARG	Argentina	37032000
Córdoba	ARG	1157507	ESP	Spain	39441700
Córdoba	ESP	311708	ESP	Spain	39441700
Córdoba	MEX	176952	ESP	Spain	39441700
Córdoba	ARG	1157507	MEX	Mexico	98881000
Córdoba	ESP	311708	MEX	Mexico	98881000
Córdoba	MEX	176952	MEX	Mexico	98881000

Seguimos sin obtener los resultados deseados, puesto que sobran filas. De todos modos, entre las filas que aparecen si están los que necesitamos. Ya veremos como filtrar los datos.

13.1.4 Propiedades generales del producto cartesiano

En diversas ocasiones usaremos productos cartesianos, así que es importante (también para entender los JOINS) entender algunas de sus propiedades.

Dimensión del Producto cartesiano:

a) El producto tendrán las columnas de ambas tablas. Así, en nuestro ejemplo: 3 + 3 = 6.
b) El producto combina cada fila de una de las tablas con todas las filas de la otra tabla, produciendo cada par de filas posible. El número total será el producto: 3 * 3 = 9.

Orden de las tablas
El orden de ejecución de ambas tablas no es significativo, puesto que, aunque cambien el orden de los campos, los resultados para las filas serán los mismos.

Para ver el producto cartesiano, solo tenemos que hacer un SELECT.

```
SELECT * FROM CiudadSimple, PaísSimple;
```

Y ahora vemos qué pasa al cambiar el orden:

```
SELECT * FROM PaísSimple,CiudadSimple;
```

El Producto cartesiano es una tabla
No física, pero si lógica, por lo que podemos hacer con ella cálculos matemáticos, filtrar registros, etc.

Productos cartesianos de mas de 2 tablas
En realidad se realiza agrupando los productos: primero se realiza el producto cartesiano de los 2 primeros y luego, el resultado (una tabla) se multiplica por un tercero. Y así sucesivamente.

Veamos un ejemplo, siguiendo lo visto anteriormente:

```
CREATE TABLE LenguajeSimple AS
SELECT CountryCode AS CodPaísLengua, Language AS Lengua
FROM CountryLanguage
WHERE CountryCode IN ('ARG','ESP','MEX');
```

Comprobamos:

```
SELECT * FROM LenguajeSimple;
```

Así puede, el producto cartesiano de CiudadSimple * PaísSimple * LenguajeSimple será:

a) Columnas o Campos: 3 + 3 + 2 = 8.
b) Filas o Registros: 3 * 3 * 13 = 117.

Veamoslo:

```
SELECT * FROM CiudadSimple, PaísSimple, LenguajeSimple;
```

13.1.5 Filtras filas no deseadas

Partimos del SELECT del producto cartesiano:

```
SELECT * FROM CiudadSimple, PaísSimple;
```

¿como filtramos las filas?
Sencillo, viendo donde CodPais coindice con Codigo:

```
SELECT * FROM CiudadSimple, PaísSimple
WHERE CodPais = Codigo;
```

Ya nos hemos quitado 6 registros que nos sobraban.

13.1.6 Columnas

De todos modos aún nos sobra varias columnas, así que habrá que filtrar un poco mas:

```
SELECT NombreCiudad, NombrePais
FROM CiudadSimple, PaísSimple
WHERE CodPais = Codigo;
```

13.1.7 Joins y claves foráneas

Normalmente los JOINS se crearán sobre claves foráneas. En nuestro ejemplo, CodPais en la tabla CiudadSimple, es una clave foránea de la tabla principal u origen, que es PaísSimple, que encima tiene como Clave Principal el propio Código.

Evidentemente, como hemos visto en el apartado anterior, no es necesario que los valores de los campos que permiten el JOIN tengan que aparecer (CodPaís y Codigo).

13.2 Join de tablas en SQL

13.2.1 SQL y el producto cartesiano

En los apartados anteriores ya hemos visto como realizar un JOIN mediante comas.
Veamos otro ejemplo sin necesidad de crearnos una tabla:

```
SELECT Name AS Ciudad, City.CountryCode AS CodPaís, Language AS Lengua
FROM City, CountryLanguage
WHERE Language = 'Spanish'
AND Name = 'Toledo';
```

NOTA: Para añadir CountryCode debo poner City para evitar ambigüedad (luego se verá).

Como puede observarse salen 84 filas, ¿porqué?
Pues por que la consulta anterior en realidad es la unión de las siguientes:

```
SELECT Name AS Ciudad,  CountryCode AS CodPaís
FROM City
WHERE Name = 'Toledo';
```

Salen 3 registros (por cierto, ninguno es el Toledo Original).

Y esta otra:

```
SELECT CountryCode AS CodPaís, Language AS Lengua
FROM CountryLanguage
WHERE Language = 'Spanish';
```

13.2.2 Usar una cláusula WHERE para la condición del JOIN.

Precisamente hemos visto en el ejemplo anterior el uso de WHERE que hemos visto hasta ahora. El problema es que ¡salen 84 registros! Para evitar esto, hemos visto como podemos unir ambas tablas por un campo común, CountryCode.

Por tanto, la unión la haremos del siguiente modo:

```
SELECT Name AS Ciudad, City.CountryCode AS CodPaís, Language AS Lengua
FROM City, CountryLanguage
WHERE City.CountryCode = CountryLanguage.CountryCode
AND Language = 'Spanish'
AND Name = 'Toledo';
```

¡Sale 1 solo registro! Esto es debido a lo siguiente:

```
SELECT * FROM City
WHERE CountryCode = 'ESP';
```

En world no han metido a Toledo entre las principales ciudades españolas.

13.2.3 Cualificación de nombres ambiguos de columnas

Ya hemos visto en el apartado anterior como evitar la ambigüedad en las columnas, poniendo el nombre de la tabla seguido por un punto y el nombre del campo. Sintaxis:

```
WHERE <nombre_tabla>.<nombre_campo> …
```

Veamos un ejemplo de lo que puede suceder:

```
SELECT Name AS Ciudad, CountryCode AS CodPaís, Language AS Lengua
FROM City, CountryLanguage
WHERE Language = 'Spanish'
AND Name = 'Valencia';
ERROR 1052 (23000): Column 'CountryCode' in field list is ambiguous
```

Para arreglarlo, indicamos la tabla y de paso, en el WHERE indicamos la unión (y de paso añadimos algún campo mas):

```
SELECT Name AS Ciudad, City.CountryCode AS CodPaís,
Language AS Lengua, Population AS Población
FROM City, CountryLanguage
WHERE City.CountryCode = CountryLanguage.CountryCode
AND Language = 'Spanish'
AND Name = 'Valencia';
```

Esta forma de indicar un campo (tabla.campo) puede ser usado en el resto de columnas, no solo en aquellas ambiguas. Además, puede emplearse en WHERE, ORDER BY, etc.

De todos modos hay un punto IMPORTANTE:
Cuando diseñemos la BBDD debemos evitar, en lo posible, que campos de tablas diferentes tengan el mismo nombre. Nos ahorraremos problemas.

Alias de Tablas
A lo largo de los apartados anteriores hemos visto el uso de ALIAS para los campos de tablas. Su sintaxis es la siguiente:

```
<campo_tabla>  [AS] <alias_tabla>
```

El uso de Alias para las tablas lleva aparejada una serie de ventajas:

- Los alias pueden ser elegidos con nombres mas cortos que los originales, reduciendo la carga y el consumo de servidor.
- Para revolver ambiguedades entre tablas.
- Para personalizar los nombres de los campos (que es sobre tod para lo que lo uso yo).

Veamos un ejemplo avanzado:

```
SELECT Capital.Name, Country.Name AS País
FROM Country, City AS Capital
WHERE Capital = Capital.ID
AND Region = 'Southern Europe'
ORDER BY Country.Name ASC;
```

Obsérvese como hemos puesto el alias en FROM y este es válido para el resto de la sentencia (en negrita habría que poner City). Por otro lado añado varias cláusulas al JOIN.

13.3 Sintaxis básica de JOIN.

Hasta ahora hemos visto algunos ejemplos de JOIN. La pregunta es, ¿y donde aparece JOIN en las sentencias? Esto es debido a que el JOIN empleado es el *join de comas*.
Vamos a ver como usar la palabra reservada JOIN. Primero su sintaxis:
<referencia-tabla> [<tipo-join>] JOIN <referencia-tabla>

A diferencia de *join de comas*., JOIN permite el uso de la palabra reservada ON que especifica la unión entre tablas a traves de sus campos correspondientes. Veamos un ejemplo:

```
SELECT City.Name, Country.Name, City.Population
FROM City JOIN Country
ON CountryCode = Code;
```

Si tuvieran criterios adicionales irian separados de las líneas que conforman el propio JOIN:

```
SELECT City.Name, Country.Name, City.Population
FROM City JOIN Country
ON CountryCode = Code
WHERE City.Name = 'Cartagena';
```

Por otro lado, la cláusula WHERE se propaga por el JOIN.
Por ejemplo, veamos como realizar un JOIN entre 3 tablas.
NOTA: Primero borramos las tablas:

```
DROP TABLE  CiudadSimple
DROP TABLE PaísSimple;
DROP TABLE LenguajeSimple;
```

Y nos creamos de nuevo todas las tablas del ejemplo:

```
CREATE TABLE CiudadSimple AS
SELECT Name AS NombreCiudad,
CountryCode AS CodPais,
Population AS Población
FROM City WHERE Name LIKE 'Córdoba';

CREATE TABLE PaísSimple AS
SELECT Code AS Codigo,
Name AS NombrePais,
Population AS PobPaís
FROM Country;

CREATE TABLE LenguajeSimple AS
SELECT CountryCode AS CodPaísLengua,
Language AS Lengua
FROM CountryLanguage;
```

Y juntamos las 3 tablas en un JOIN que tendrá 13 registros:

```
SELECT NombreCiudad, NombrePais,Lengua
FROM CiudadSimple
JOIN PaísSimple
ON CodPais = Codigo
JOIN LenguajeSimple
ON Codigo=CodPaísLengua;
```

13.3.1 Condiciones no asociadas al JOIN en la cláusula ON

Hemos visto cómo usar ON junto a un WHERE en la consulta anterior. Sin embargo ON, dentro de un JOIN puede realizar el mismo papel que un WHERE, añadiendo cláusulas AND adicionales. Un ejemplo:

```
SELECT City.Name AS Ciudad,
Country.Name AS País,
City.Population AS Población
FROM City JOIN Country
ON CountryCode = Code
AND City.Name = 'León';
```

Incluso podemos usar expresiones con el anterior AND:

```
SELECT City.Name, Country.Name, City.Population
FROM City JOIN Country
ON CountryCode = Code
AND City.Name = 'León'
AND City.Population > 1000000;
```

De todos modos, aunqué valido, no es recomendable hacer esto. Es preferible usar el ON para unir campos de varias tablas y emplear el WHERE para filtrar los resultados (añado alias):

```
SELECT City.Name AS Ciudad, Country.Name AS País,
City.Population AS Población
FROM City JOIN Country
ON CountryCode = Code
WHERE City.Name = 'León'
AND City.Population > 1000000;
```

También podemos añadir expresiones LIKE

```
SELECT City.Name AS Ciudad, Country.Name AS País,
Region,City.Population AS Población
FROM City JOIN Country
ON CountryCode = Code
WHERE City.Name = 'León'
AND Region LIKE '%America';
```

13.4 Joins internos (INNER JOIN)

Los Joins de coma (Producto cartesiano) y los JOINS usando dicha palabra reservada se denominan Joins Internos. En estos casos podemos emplear la palabra reservada INNER para indicar este tipo de Joins. Siguiendo con los ejemplos anteriores:

```
SELECT City.Name AS Ciudad, Country.Name AS País,
Region,City.Population AS Población
FROM City INNER JOIN Country
ON CountryCode = Code
WHERE City.Name = 'León'
AND Region LIKE '%Europe';
```

13.4.1 Pseudo-código de JOIN interno

¿Como se representa el ejemplo anterior con un pseudo-código?

```
goto_first_row (City)
while has_rows(City) do
     goto_first_row (Country)
     while has_rows(Country) do
           If City.CountryCode = Country.CountryCode then
               If City.Name = 'León' then
                    new_row = current_row(City)+current_row(Country)
                    add_row (result, new_row)
               end_if
          end_if

          goto_next_row (Country)
     end_while

     goto_next_row (City)
end_while
```

Obsérvese el uso de dos bucles para recorrer, al completo, primero la tabla City y luego Country. Por último usando sendos If juntamos ambas tablas por el CountryCode (el equivalente al ON) y luego filtramos por el nombre de la Ciudad, León.

NOTA: Ambos If podrían haber quedado así:

```
If City.CountryCode = Country.CountryCode && City.Name = 'León' then
```

Pero así se ve mas ordenado.

13.4.2 Omitir la condición de JOIN

De forma accidental se puede dar el caso de omitir el INNER JOIN. En tal caso, si prescindimos de mas cláusulas veremos el producto cartesiano. Si añadimos cláusulas, por fuerza, tendremos que poner WHERE y no podremos añadir el ON correspondiente.
Así, la sentencia del apartado 13.4 quedaría:

```
SELECT City.Name AS Ciudad, Country.Name AS País,
Region,City.Population AS Población
FROM City, Country
WHERE CountryCode = Code
AND City.Name = 'León'
AND Region LIKE '%Europe';
```

Mucho cuidado con esto. Estamos ante un INNER JOIN.

13.5 Joins externos (OUTER JOIN)

En los apartados anteriores hemos visto como realizar INNER JOIN, donde descartamos aquellos registros que no cumplen los requisitos de la unión. Pero a veces nos interesa conservar dichos registros para visualizarlos junto a los que lo cumplen.

13.5.1 Dos tablas simples (de nuevo)

Para probar los `OUTER JOIN` vamos a crearnos de nuevo otras tablas.
En este caso veremos el caso de London (y la tabla estará en inglés):

```
CREATE TABLE SimpleCity AS
SELECT ID AS CityID, Name AS CityName, CountryCode
FROM City
WHERE Name Like 'London';

SELECT * FROM SimpleCity;
```

Ahora otra tabla mas:

```
CREATE TABLE SimpleCountry AS
SELECT Code, Name AS CountryName, Capital
FROM Country
WHERE Code IN ('CAN', 'GBR');

SELECT * FROM  SimpleCountry;
```

13.5.2 Conservar las filas aunque no se encuentren coincidencias.

El primer paso será ver cual es la ciudad capital (haciendo un INNER JOIN) del caso anterior:

```
SELECT CountryName, CityName
FROM SimpleCountry INNER JOIN SimpleCity
ON Capital = CityID;
```

Pues bien, lo que vamos a conseguir es una combinación en la cual se listan los registros de una de las tablas y las coincidencias entre ambas. Hay dos opciones, realizar un JOIN EXTERNO IZQUIERDO o un JOIN EXTERNO DERECHO.

13.5.3 JOIN externo izquierdo (LEFT OUTER JOIN)

Las características de un LEFT OUTER JOIN son:

- Devuelve todas las filas que cumplen con la <condicion_JOIN>
- Conserva todas las filas no emparejadas de <tabla_izquierda>
- Asigna NULL a las columnas de la <tabla_derecha> en caso de que no se encuentre ninguna coincidencia para una fila en particular de la tabla_izquierda.

Un ejemplo:

```
SELECT CountryCode, CityName, CityID, Capital
FROM SimpleCity
LEFT JOIN SimpleCountry
ON CityID = Capital;
```

```
+-------------+----------+--------+---------+
| CountryCode | CityName | CityID | Capital |
+-------------+----------+--------+---------+
| GBR         | London   |    456 |     456 |
| CAN         | London   |   1820 |    NULL |
+-------------+----------+--------+---------+
```

13.5.4 JOIN derecho (RIGHT JOIN)

Las características de un LEFT OUTER JOIN son:

- Devuelve todas las filas que cumplen con la <condicion_JOIN>
- Converva todas las filas no emparejadas de <tabla_derecha>
- Asigna NULL a las columnas de la <tabla_izquierda> en caso de que no se encuentre ninguna coincidencia para una fila en particular de la tabla_derecha .

Un ejemplo:

```
SELECT CountryCode, CityName, CityID, Capital
FROM SimpleCountry RIGHT OUTER JOIN SimpleCity
ON Capital = CityID;
```

13.5.5 La condición de JOIN en las operaciones de JOIN externo

Existen algunas condiciones para el uso de JOIN externos; resumiéndolos son los siguientes:

a) Es obligatorio el uso de la cláusula ON (en los INNER JOIN se pueden omitir)
b) El orden de las tablas no es esencial. De todos modos, por razones de estilo, se suele poner el mismo orden en el FROM y en el ON (vease el ejemplo del apartado anterior).
c) Sin embargo, el orden en las condiciones es FUNDAMENTAL. Veamoslo con un ejemplo:

```
SELECT City.Name, Country.Name, Country.IndepYear
FROM City LEFT JOIN Country
ON City.ID = Country.Capital
WHERE City.Name IN ('Madrid','Sevilla','Jamestown');
```

El resultado es:

```
+-----------+--------------+-----------+
| Name      | Name         | IndepYear |
+-----------+--------------+-----------+
| Madrid    | Spain        |      1492 |
| Sevilla   | NULL         |      NULL |
| Jamestown | Saint Helena |      NULL |
+-----------+--------------+-----------+
3 rows in set (0.01 sec)
```

Obsérvese como, al estar ligado la capitalidad al año de la independencia nos sale que Madrid es la capital y “está independizada”, pero Sevilla no. Jamestown no es independiente.

Vamos a filtrar solo para aquellos casos en el que el país de la ciudad es independiente.
La condición será:

```
Country.IndepYear IS NOT NULL.
```

Ahora la pregunta es: ¿Que ponemos un `WHERE...AND` o un `ON...AND`?

```
SELECT City.Name, Country.Name, Country.IndepYear
FROM City LEFT JOIN Country
ON City.ID = Country.Capital
WHERE City.Name IN ('Madrid','Sevilla','Jamestown')
AND Country.IndepYear IS NOT NULL;
```

Nos sale lo siguiente:

```
+--------+-------+-----------+
| Name   | Name  | IndepYear |
+--------+-------+-----------+
| Madrid | Spain |      1492 |
```

¿Pero que pasa con Sevilla? Vamos a probar con la cláusula ON:

```
SELECT City.Name, Country.Name, Country.IndepYear
FROM City JOIN Country
ON City.ID = Country.Capital
AND Country.IndepYear IS NOT NULL
WHERE City.Name IN ('Madrid','Sevilla','Jamestown');
```

Ahora el resultado es el correcto (aunque se cuele Jamestown porque lo hemos añadido en el WHERE):

```
+-----------+-------+-----------+
| Name      | Name  | IndepYear |
+-----------+-------+-----------+
| Madrid    | Spain |      1492 |
| Sevilla   | NULL  |      NULL |
| Jamestown | NULL  |      NULL |
+-----------+-------+-----------+
```

13.5.6 Elegir entre joins internos y externos

Debemos estimar varias alternativas:

- Un JOIN externo es mas lento que uno interno.
- Un JOIN externo dará lugar a valores NULL, por lo que tendremos que contrarlos.

Por tanto, de primeras, siempre optaremos por los internos.

Columnas que aceptan NULL en una condición de JOIN

Si las columnas usadas en el JOIN aceptan NULL, se debe considerar lo que pasaría con las filas que tienen ese valor en esas columnas. Si eso es aceptable o si el resultado de unir las tablas en un JOIN no contiene esas filas, entonces usaremos un JOIN INTERNO. En cualquier otro caso, usaremos un JOIN EXTERNO.

Por ejemplo: queremos sacar una lista con todos los países y el nombre de su capital. Vemos primero la estructura de Country:

```
DESC Country;
```

Y vemos como el campo Capital permite Nulos. Esto significa que debemos usar un JOIN Externo para sacar dicho dato:

```
SELECT Country.Name, City.Name
FROM Country LEFT JOIN City
ON Capital = City.ID;
```

Obsérvese como hay Paises que no tienen capital. Si hubiésemos usado un JOIN INTERNO dichos paises se hubiesen perdido.

Join con una tabla resultante de un JOIN EXTERNO
En este caso debemos usar otro JOIN EXTERNO para tratar esta nueva tabla (porque tiene valores NULL como hemos visto antes).

Veamos un ejemplo: Queremos elaborar una lista con todas las ciudades y que si la ciudad es capital, incluya el nombre del país y para este país en particular también los idiomas que se hablan en él. Empezaremos con un JOIN EXTERNO ya visto:

```
SELECT City.Name, Country.Name AS 'Capital de'
FROM City
LEFT JOIN Country
ON City.ID = Country.Capital;
```

Salen las 4000 ciudades, la mayoría de ellas con valores NULL (no son capitales del país). Ahora vamos a unir esta tabla, resultado de un JOIN EXTERNO con CountryCode. Lo haremos a través de las columnas Code de Country (que no permite NULL) y CountryCode de CountryLanguage (que tampoco permite NULL). En negrita el añadido:

```
SELECT City.Name, Country.Name AS 'Capital de', Language
FROM City
LEFT JOIN Country
ON City.ID = Country.Capital
LEFT JOIN CountryLanguage
ON Country.Code = CountryLanguage.CountryCode;
```

Obsérvese como se han añadido 800 registros mas, debido a que, para los países con mas de un idioma, se ha repetido la capital añadiendo el idioma correspondiente (Ej: Washington).

Resolver problemas de no existencia ('Not Exists')
Un JOIN INTERNO puede obtener solo pares de filas que coinciden en caso de encontrar alguna coindencia. Por ejemplo, queremos obtener una lista de los idiomas que se hablan en un país en particular: para ello basta con un JOIN INTERNO entre Country y CountryLanguage.

```
SELECT Name, Language
FROM CountryLanguage INNER JOIN Country
ON Code = CountryCode
WHERE Country.Code = 'ESP';
```

Esto está muy bien, porque España tiene idiomas, pero, ¿y si quito el WHERE?

```
SELECT Name, Language
FROM CountryLanguage INNER JOIN Country
ON Code = CountryCode;
```

Pues que no salen aquellos países que no tienen un idioma (ni oficial ni oficial).
¿Y como puedo ver aquellos países sin idioma? Pues si, usando un JOIN EXTERNO.

```
SELECT Name
FROM Country LEFT JOIN CountryLanguage
ON Code = CountryCode
WHERE Language IS NULL;
```

Salen 6 registros.
Esto mismo se puede realizar mediante una subconsulta (que veremos en el siguiente tema) pero es menos eficiente que hacerlo con JOIN EXTERNO.

NOTA IMPORTANTE: De nuevo, obsérvese la importancia de poner el orden correcto de las tablas en el LEFT JOIN; vamos a cambiar la sentencia y veamos el resultado:

```
SELECT Name
FROM CountryLanguage LEFT JOIN Country
ON Code = CountryCode
WHERE Language IS NULL;
```

Agregar filas relacionadas
De nuevo volvemos al problema de los registros con campos NULL. Imaginemos que queremos saber el número de ciudades importantes por país. El problema es: ¿que pasa con aquellos países -como la Antártida, que aparece así- que no tienen ciudades? ¿Las omitimos con un JOIN INTERNO? ¿O las añadimos con un JOIN EXTERNO? Veamos este segundo caso:

```
SELECT Country.Name AS 'Nombre País', COUNT(City.ID) AS 'Ciudades'
FROM Country LEFT JOIN City
ON Code = CountryCode
GROUP BY Code;
```

Mucho cuidado a la hora de elegir la columna sobre la que se hará el conteo. Si dentro del COUNT solo consideramos el nombre del país y sus ciudades, veremos como en aquellos países SIN CIUDADES al menos nos sale el valor de 1:

```
SELECT Country.Name, COUNT(*)
FROM Country LEFT JOIN City
ON Code = CountryCode
GROUP BY Code;
```

Y esto, evidentemente, es erróneo.

Por último veamos un ejemplo que diferencia el uso de LEFT JOIN y RIGHT JOIN

- Usando LEFT JOIN para buscar las capitales de Oceanía.

```
SELECT Country.Name AS País, City.Name AS 'Capital'
FROM Country LEFT JOIN City
ON Capital = ID
WHERE Continent = 'Oceania';
```

- Usando RIGHT JOIN para buscar las capitales de Oceanía,

```
SELECT Country.Name AS País, City.Name AS 'Capital'
FROM Country RIGHT JOIN City
ON Capital = ID
WHERE Continent = 'Oceania';
```

¿Que diferencia hay? Pues que el país *United States Minor Outlying Islands* no aparece en este segundo caso, al no tener capital.

13.6 Otros tipos de Joins

Además de los JOINS INTERNOS y EXTERNOS tenemos otros tipos de JOINS que son:

I. JOINS sobre columnas del mismo nombre
   a. JOIN NATURAL
   b. JOIN con USING (JOIN de columnas con nombre)
II. CROSS JOIN (producto cartesiano explícito)
III. Categorías de condición de JOIN
   a. JOIN EQUI
   b. JOIN NON-EQUI
IV. JOIN BEETWEEN … AND
V. AUTOJOIN o de una tabla consigo misma (SELF JOIN)

Recomiendo la lectura y el empleo de los ejemplos de esta excelente página:
http://www.mysqlya.com.ar/index.php?inicio=63

NATURAL JOIN

Es como una operación normal de JOIN, con dos características adicionales:

- La condición de JOIN no puede ser especificada explícitamente. En lugar de ello, la misma se infiere solicitando la igualdad de todos los pares de columnas que tienen idéntico nombre.
- Si la lista SELECT usa el comodín de todas las columnas (*), entonces sólo se genera una columna resultante por cada par de columnas con igual nombre.

En resumen, el JOIN se realiza entre campos de diversas tablas con el mismo nombre.
Su sintaxis es:

```
<referencia_tabla> NATURAL [<tipo_join_externo>] JOIN <referencia_tabla>
```

Un ejemplo:
NOTA: Vamos a emplear tablas simplificadas previamente creadas. Por si no se encuentran sus definiciones, aquí las dejo:

```
CREATE TABLE SimpleLanguage AS
SELECT CountryCode, Language
FROM CountryLanguage
WHERE CountryCode IN ('CAN', 'GBR') AND IsOfficial = 'T';

CREATE TABLE SimpleCountry AS
SELECT Code, Name AS CountryName, Capital
FROM Country WHERE Code IN ('CAN', 'GBR');

CREATE TABLE SimpleCity AS
SELECT ID AS CityID, Name AS CityName, CountryCode
FROM City WHERE Name Like 'London';

SELECT SimpleLanguage.CountryCode, Language, CityID, CityName
FROM SimpleLanguage INNER JOIN SimpleCity
ON SimpleLanguage.CountryCode = SimpleCity.CountryCode;
```

Pues bien, con el NATURAL JOIN podemos reducir esta sentencia a lo siguiente:

```
SELECT *
FROM SimpleLanguage NATURAL JOIN SimpleCity;
```

Tremendo. Quitamos los campos de unión, porque sólo hay uno con el mismo nombre en ambos. Además, prescindimos de enumerar los campos a mostrar.
Eso si, hay que tener mucho cuidado con el uso de NATURAL JOIN porque, aunque parece muy poderoso, lleva consigo unas restricciones no menos importantes. Un simple cambio de estructura de las tablas, y el resultado no será el mismo.

JOIN con USING (JOIN de columnas con nombre)
Es muy similar a NATURAL JOIN, con la diferencia que especifica el nombre del campo de unión entre tablas con la cláusula USING. Esto permite evitar el error del cambio de nombres de campo en las tablas que no controla NATURAL JOIN. Además permite añadir campos de unión adicionales. Su sintaxis es:

```
<referencia_tabla> [<tipo_join>] JOIN <referencia_tabla>
USING (<nombre_col1> [,<nombre_col2>,...,<nombre_colN>])
```

Veamos un ejemplo, en primer lugar con un JOIN INTERNO:

```
SELECT Language,  CityName
FROM SimpleLanguage INNER JOIN SimpleCity
ON SimpleLanguage.CountryCode =  SimpleCity.CountryCode;
```

Y ahora simplificando con la cláusula USING:

```
SELECT Language, CityName
FROM SimpleLanguage INNER JOIN SimpleCity
USING (CountryCode);
```

CROSS JOIN
Es una operación que calcula el producto cartesiano. La sintaxis será:

```
<referencia_tabla> CROSS JOIN <referencia_tabla>
```

En este caso no veremos ningún INNER JOIN por ningún lado, porque en realidad son muy similares (y no tiene sentido repetir lo mismo dos veces). Veamos un ejemplo:

```
SELECT Language, CityName
FROM SimpleLanguage CROSS JOIN SimpleCity;
```

Que en realidad es lo mismo que:

```
SELECT Language, CityName
FROM SimpleLanguage, SimpleCity;
```

JOIN EQUI (EQUI JOIN)
Este término denota cualquier operación de JOIN que relacion tablas mediante una comparación de igualdad sobre los pares de columnas de las tablas de JOIN. De modo que el término EQUI JOIN expresa algo sobre la condición de JOIN mas que la forma en que las filas de las tablas son combinadas entre si. NATURAL JOIN y el JOIN de columnas que emplea USING son formas particulares de EQUI-JOIN, aunque no incluyan expresamente un operador de igualdad (=) en la condición del JOIN. Este tipo de JOIN es el mas usado de todos; de hecho todos los ejemplos de este capítulo donde se han empleado una condición de JOIN son ejemplos de este tipo.

JOIN NON-EQUI (NON-EQUI JOIN)
Por el contrario, existe JOIN que no usa los comparadores de igualdad. Por ejemplo, listar aquellas ciudad que NO son capitales del país (por cierto, lo reduzco a España):

```
SELECT Country.Name, City.Name
FROM City INNER JOIN Country
ON CountryCode = Code
AND City.CountryCode = 'ESP'
AND ID != Capital;
```

Aunque el ON se realiza a través de una igualdad, en la comparación final hemos empleado un NON – EQUI JOIN, separando las capitales del resultado.

JOIN BETWEEN … AND

Para visualizar ejemplos de este tipo vamos a usar la BBDD employees visto en el apartado 7.1.4; primero veamos su estructura: `SHOW TABLES;`

```
+----------------------+
| departments          |
| dept_emp             |
| dept_manager         |
| employees            |
| salaries             |
| titles               |
+----------------------+
```

```
DESC departments;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| dept_no   | char(4)     | NO   | PRI | NULL    |       |
| dept_name | varchar(40) | NO   | UNI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

```
DESC dept_emp;
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| emp_no    | int(11) | NO   | PRI | NULL    |       |
| dept_no   | char(4) | NO   | PRI | NULL    |       |
| from_date | date    | NO   |     | NULL    |       |
| to_date   | date    | NO   |     | NULL    |       |
+-----------+---------+------+-----+---------+-------+
```

```
DESC dept_manager;
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| dept_no   | char(4) | NO   | PRI | NULL    |       |
| emp_no    | int(11) | NO   | PRI | NULL    |       |
| from_date | date    | NO   |     | NULL    |       |
| to_date   | date    | NO   |     | NULL    |       |
+-----------+---------+------+-----+---------+-------+
```

```
DESC employees;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| emp_no     | int(11)       | NO   | PRI | NULL    |       |
| birth_date | date          | NO   |     | NULL    |       |
| first_name | varchar(14)   | NO   |     | NULL    |       |
| last_name  | varchar(16)   | NO   |     | NULL    |       |
| gender     | enum('M','F') | NO   |     | NULL    |       |
| hire_date  | date          | NO   |     | NULL    |       |
+------------+---------------+------+-----+---------+-------+
```

```
DESC salaries;
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| emp_no    | int(11) | NO   | PRI | NULL    |       |
| salary    | int(11) | NO   |     | NULL    |       |
| from_date | date    | NO   | PRI | NULL    |       |
| to_date   | date    | NO   |     | NULL    |       |
+-----------+---------+------+-----+---------+-------+
```

```
 DESC titles;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| emp_no    | int(11)     | NO   | PRI | NULL    |       |
| title     | varchar(50) | NO   | PRI | NULL    |       |
| from_date | date        | NO   | PRI | NULL    |       |
| to_date   | date        | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

Para empezar vamos a ver los distintos departamentos:

```
SELECT DISTINCT dept_name FROM departments;
```

Ahora vamos a simplificar las tablas (es un decir): Vamos a juntar las tablas Empleados, Departamentos, Empleados_Dept y Salarios para sacar los datos principales de los empleados que trabajan en “Diseño”: Emp_ID, Nombre, Apellidos, Dept_Nombre, Salario, Dia_Inicial, Dia_Final (de cobro del salario).

```
CREATE TABLE Empleado_Simple AS
SELECT employees.emp_no AS Emp_ID, first_name AS Nombre,
last_name AS Apellidos, dept_name AS Dept_Nombre, salary AS Salario,
salaries.from_date AS Dia_Inicial, salaries.to_date As Dia_Final
FROM employees, departments, dept_emp, salaries
WHERE dept_emp.dept_no =  departments.dept_no
AND employees.emp_no = dept_emp.emp_no
AND employees.emp_no = salaries.emp_no
AND dept_name = 'Development'
ORDER BY employees.emp_no
LIMIT 100;
```

(lo he limitado a los primeros 100 registros para agilizar luego las consultas).
Y vemos los registros creados:

```
SELECT * FROM Empleado_Simple;
```

Ahora vamos a crearnos otra tabla, también de empleados con sus salarios, pero sin estar limitados al departamento de “Diseño” (la sentencia es mucho mas simple):

```
CREATE TABLE Salario_Simple AS
SELECT employees.emp_no AS Emp_ID, first_name AS Nombre,
last_name AS Apellidos, salary AS Salario
FROM employees, salaries
WHERE employees.emp_no = salaries.emp_no
ORDER BY employees.emp_no
LIMIT 100;
```

Vemos el resultado:

```
SELECT * FROM Salario_Simple;
```

Por fin vemos el ejemplo del apartado: INNER JOIN junto a BETWEEN AND, viendo aquellos empleados, del departartamento de Diseño, que han ganado entre 60000 y 80000 (dolares) junto a los Dias Iniciales y Finales.

```
SELECT Empleado_Simple.Emp_ID, Empleado_Simple.Nombre,
Empleado_Simple.Apellidos, Empleado_Simple.Salario,
Dia_Inicial, Dia_Final
FROM Empleado_Simple INNER JOIN Salario_Simple
ON Empleado_Simple.Emp_ID
WHERE Empleado_Simple.Salario
BETWEEN 60000 AND 80000
GROUP BY Empleado_Simple.Emp_ID;
```

AUTO-JOIN o JOIN de una tabla consigo misma

Un SELF-JOIN o un AUTO-JOIN es un JOIN que se realiza sobre una misma tabla.
Si la BBDD de employee fuera diferente (en estructura) podríamos tener algo similar a esto en la tabla Empleados:

```
+--------+----------+-----------+---------+-------------+------------+
| Emp_ID | Nombre   | Apellidos | Salario | Dia_Inicial | Boss_ID    |
+--------+----------+-----------+---------+-------------+------------+
|  10001 | Georgi   | Facello   |   60117 | 1986-06-26  | NULL       |
|  10006 | Anneke   | Preusig   |   60098 | 2000-08-02  | NULL       |
|  10014 | Berni    | Genin     |   60598 | 2001-12-27  | NULL       |
|  10018 | Kazuhide | Peha      |   61361 | 1989-04-02  | NULL       |
|  10021 | Ramzi    | Erde      |   60851 | 1991-02-09  | 10021      |
+--------+----------+-----------+---------+-------------+------------+
```

Para sacar al jefe deberiamos hacer lo siguiente:

```
SELECT Empleados_Nombre, Empleados_Apellido
FROM Empleados INNER JOIN  Empleados AS Jefe
ON Empleados.EmpID = Empleados.Boss_Id;
```

Por cierto, hablando de Jefes, ¿como podemos crearnos una tabla con los jefes de los distintos departamentos de la BBDD employee?
Aquí la solución:

```
CREATE TABLE Jefes_Simple AS
SELECT employees.emp_no AS Emp_ID, first_name AS Nombre,
last_name AS Apellidos, dept_name AS Dept_Nombre,
dept_manager.emp_no AS Jefe
FROM employees, departments, dept_manager, dept_emp
WHERE dept_emp.dept_no =  departments.dept_no
AND employees.emp_no = dept_emp.emp_no
AND employees.emp_no = dept_manager.emp_no
ORDER BY employees.emp_no;

SELECT * FROM  Jefes_Simple;
```

13.7 Joins en Sentencias UPDATE y DELETE

Los JOIN no están restringidos a operaciones de SELECT. También lo podemos emplear para UPDATE y DELETE para realizar dichas operaciones en varias tablas a la vez. OJO: Esta operación no entra dentro del Standard SQL.

13.7.1 Sintaxis de UPDATE multi-tablas

La sintaxis será la siguiente:

```
UPDATE <tablas_join>
SET <asignaciones_columna>
[WHERE <condicion>]
```

Cada porción de `<tablas_join>` puede contener cualquiera de los JOIN vistos anteriormente. Por su parte en `<asignaciones_columna>` podemos agregar una lista separadas de comas. Cada una de ellas será:

```
<referencia_columna> = <expresiones_valor>
```

Por último la cláusula WHERE es opcional y se emplea como ya hemos visto.

Por ejemplo:
Queremos añadir 2000 personas a la población de Sevilla, añadiéndolas, a su vez, a la población de España.
Primero veamos los datos de origen:

```
SELECT * FROM City
WHERE Name = 'Sevilla';

SELECT Name, Population
FROM Country
WHERE Code = 'ESP';
```

Y ahora el JOIN con UPDATE:

```
UPDATE City
INNER JOIN Country
ON CountryCode = Code
      SET City.Population = City.Population + 2000,
      Country.Population = Country.Population + 2000
WHERE Country.Code = 'ESP'
AND City.Name = 'Sevilla';
```

Y listo. Volvemos a probar los SELECT para ver los datos de Sevilla y España.

NOTA: Las cláusulas ORDER BY y LIMIT pueden ser empleadas en un tabla, pero no en varias, como en el anterior JOIN. Si se intenta, generará un ERROR.

13.7.2 Sintaxis de DELETE multi-tablas

Para realizar un DELETE multitablas, tenemos dos posibles sintaxis:

```
DELETE <lista_tablas>
FROM <tablas_join>
[WHERE <condicion>]

DELETE
FROM <lista_tablas>
USING <tablas_join>
[WHERE <condicion>]
```

`<lista_tablas>` especifica las tablas a unir, separadas por comas.
`<tablas_join>` junto al WHERE especifican las filas a borrar.
Las `<lista_tablas>` deben aparecer obligatoriamente en las `<tablas_join>`, pero las que están en estas no tienen por qué aparecer en `<lista_tablas>`.

NOTA: Cuando hablamos de DELETE prefiero no tocar la base de datos del ejemplo. Por tanto, vamos a crearnos sendas copias de las tablas:

```
CREATE TABLE CityCopia
AS SELECT * FROM City;
CREATE TABLE CountryCopia
AS SELECT * FROM Country;
CREATE TABLE CountryLanguageCopia
AS SELECT * FROM CountryLanguage;
```

Y ahora si, procedemos al borrado de todas las ciudades, idiomas de Holanda (NLD):

```
DELETE CountryCopia, CityCopia, CountryLanguageCopia
FROM CountryCopia
LEFT JOIN CityCopia
ON Code = CityCopia.CountryCode
      LEFT JOIN CountryLanguageCopia
      ON Code = CountryLanguageCopia.CountryCode
WHERE Code = 'NLD';
```

IMPORTANTE: Aunque con las copias hemos pasado de restricciones, debemos tener en cuenta el orden a la hora de borrar. Primero borraremos los registros de las tablas “hijas” (en nuestro caso CityCopia o CountryLanguageCopia) antes que la principal (CountryCopia).

Alias de Tablas

`<lista_tablas>` no tienen por qué ser una lista de nombres de tablas literales. Es una lista de referencias a tablas usadas en la porción `<tablas_join>`. Si en esta última se emplean alias, tendremos que usarlas OBLIGATORIAMENTE en `<lista_tablas>`.
Veamos otro borrado (esta vez de ciudades y lenguas de Gran Bretaña):

```
DELETE Co, Ci, Cl
FROM CountryCopia AS Co
LEFT JOIN CityCopia AS Ci
ON Code = Ci.CountryCode
      LEFT JOIN CountryLanguageCopia As Cl
      ON Code = Cl.CountryCode
WHERE Code = 'GBR';
```

Obsérvese que, una vez puesto el alias en el LEFT JOIN, en el ON debemos usar dicho alias (dará un error si mantenemos el nombre anterior).

Listar solo tablas especificas
Pues bien, ahora vamos a borrar las ciudades e idiomas de USA, pero sin borrar el país. Para ello solo tendremos que quitar la tabla del DELETE. Ejemplo (usando los alias anteriores):

```
DELETE Ci, Cl
FROM CountryCopia AS Co
LEFT JOIN CityCopia AS Ci
ON Code = Ci.CountryCode
      LEFT JOIN CountryLanguageCopia As Cl
      ON Code = Cl.CountryCode
WHERE Code = 'USA';
```

13.7.3 Por qué usar sentencias UPDATE y DELETE multi-tablas

El uso de UPDATE y DELETE en operaciones multitabla tienen una serie de ventajas:

- El borrado y actualización de varias tablas se pueden empaquetar en una sola sentencia, reduciendo las llamadas al servidor. Esto se puede conseguir con rutinas almacenadas, pero estas no está presentes en mysql hasta la versión 5.
- El uso de subconsultas puede llegar a ser mucho menos eficiente que el uso de JOIN con UPDATE y DELETE.
- Se pueden agrupar varias sentencias de actualización y borrado en una sola sentencia, y así aprovechar el borrado o actualización en cascada.

También tienen una serie de inconvenientes:

- El borrado o actualización sobre varias tablas no tiene la propiedad de la atomicidad. Si hay un error en un parte, puede no llegar a pararse la ejecución, provocando errores de consistencia.
- No soportan las cláusulas ORDER BY y LIMIT, con todo lo que ello conlleva.
- Las reglas de actualización o borrado en cascada puede afectar gravemente en el orden de presentación de las sentencias de UPDATE y DELETE multitabla.
- Lo mas grave: No es un estándar SQL.

14 SUBCONSULTAS

14.1 Una visión general

Una subconsultsa es una consulta (SELECT) que aparece entre paréntesis como parte de otra sentencia SQL (siendo por si misma una sentencia).
Veamos un ejemplo: se usan las tablas Country y CountryLanguage para buscar los idiomas hablados en Finlandia.

```
SELECT Language
FROM CountryLanguage
WHERE CountryCode = 'FIN';

Ahora lo mismo con una subconsulta:
SELECT Language
FROM CountryLanguage
WHERE CountryCode = (
      SELECT Code
      FROM Country
      WHERE Name = 'Finland'
      );
```

En cursiva está la consulta anidada, siendo el resto la consulta externa. Dicha consulta externa puede ser, a su vez, una consulta SQL o bien una expresión.

¿Para qué usar subconsultas?
Aunque en el ejemplo anterior parezca que no hay demasiada utilidad, si lo tiene en otros casos con múltiples tablas (como hemos visto a la hora de trabajar con la BBDD employees). Si bien es cierto que pueden emplearse JOIN, por lo general, las subconsultas serán mas directas y sencillas de leer.

Una subconsulta puede ser leida desde dentro hacia fuera. En otras palabras, se puede centrar en el resultado de la subconsulta en primer lugar, para luego pasar a la consulta externa. Esto aporta una estructura definida a las sentencias, por lo que el lenguaje es llamado, muy convenientemente, Lenguaje de Consultas Estructuradas (Structures Query Language, SQL).

14.2 Tipos de Subconsultas

Las subconsultas se clasifican en función a la función que cumplen en el conjunto resultante respecto a la expresión contenedora. Esto permite distinguir lo siguientes tipos:

- Subconsultas escalares: se tratan como un valor individual
- Subconsultas de fila: se tratan como una fila individual
- Subconsultas de tabla: tratadas como una tabla (solo lectura) que contiene 0 o + filas.

El término subconsulta de columna es usado para indicar un caso especial de subconsultas de tabla que selecciona solo una columna. Sin embargo, no representa realmente una clase distinta de subconsulta.

14.2.1 Subconsultas escalares

Estas subconsultas actúan como simples expresiones de valor singulares (escalares). Es decir, el resultado de la consulta anidada será un valor único, que puede llegar a ser NULL.
En cuanto a la expresión que las contiene, las subconsultas escalares tienen el mismo status que los literales, llamadas a funciones, referencia a columnas y afines. Así, podemos verlas junto a listas SELECT, argumentos de funciones o como operandos en cálculos ariméticos o de comparación.

Veamos otro ejemplo, queremos saber el porcentaje de población de cada país europeo respecto a la población mundial:

```
SELECT Country.Name AS País,
      100 * Country.Population /
      (
      SELECT SUM(Population)
      FROM Country
      ) AS '% Población Mundo'
FROM Country
WHERE Continent = 'Europe';
```

NOTA: He usado un alías para Country.Name sin emplear comillas. Esto es válido solo para cadenas de caracteres de un solo elemento, pero no lo recomiendo.

Posibles errores a la hora de tratar este tipo de subconsultas

- Que el conjunto resultante dé lugar a mas de una columna. Ejemplo:

```
SELECT Name FROM Country
WHERE Code = (SELECT * FROM Country);
ERROR 1241 (21000): Operand should contain 1 column(s)
```

- Que el conjunto resultante dé lugar a mas de una fila. Ejemplo:

```
SELECT Name FROM Country
WHERE Code = (SELECT Name FROM Country);
ERROR 1242 (21000): Subquery returns more than 1 row
```

Por otro lado, no hay problema si el resultado de la consulta anidada es un valor NULL o un conjunto vacío . Ejemplo:

```
SELECT (
      SELECT Name FROM  Country LIMIT 0);
+-------------------------------------+
| NULL                                |
+-------------------------------------+

SELECT Name FROM Country
      WHERE Code = (
      SELECT Name FROM  Country
      WHERE Code = 'ZZZ');
Empty set (0.00 sec)
```

Un caso donde las subconsultas escalares son muy útiles es cuando se cálcula un acumulado de mútiples detalles no relacionados. Por ejemplo, se quiere calcular el número de ciudades por país así como el número de idiomas. Será así:

```
SELECT Name AS País,
      (SELECT COUNT(*)  FROM City
      WHERE CountryCode = Code) AS Ciudades,
      (SELECT COUNT(*) FROM CountryLanguage
      WHERE CountryCode = Code) AS Lenguas
FROM Country;
```

14.2.2 Subconsultas de fila

Estas subconsultas son tratadas como una fila individual que contiene al menos dos columnas. Es un caso especial del llamado constructor de fila (que es una expresión que muestra como resultado una fila individual que tiene dos o mas columnas).

Si la expresión SELECT entre paréntesis obtiene una fila individual, la subconsulta de fila da como resultado dicha fila. Si la expresión SELECT no obtiene ninguna fila, la subconsulta da como resultado una fila individual que tiene NULL en sus columnas. Así que aunque la expresión SELECT arroje un conjunto vacio, la expresión completa entre paréntesis seguirá mostrando como resultado una fila.

Las subconsultas de fila pueden usarse con operadores de igualdad y de comparación. Cuando estos operadores se usan, son evaluados aplicando el operador por parejas, sobre los correspondientes valores (escalares) de las columnas individuales.

En cuanto a los operadores de igualdad, todas las comparaciones tienen que ser verdaderas para que la expresión así lo sea. En otras palabras, dos filas son iguales solo si todos los valores de las columnas correspondientes son iguales. En los operadores de desigualdad, basta que una sea verdadera para la expresión así lo sea.

Para los operadores de comparación, el orden de las columnas tiene importancia (además de sus valores). Por ejemplo, se considera que (SELECT 2,1) es mayor que la columna (SELECT 1,100) porque 2 es mayor que 1. La segunda parte (1, 100) ya no es significativa. En el caso opuesto, (SELECT 1,2) y (SELECT 100,1) será justo al revés.

Veamos otro ejemplo usando world comparando filas:

```
SELECT ('London','GBR')  =
      (SELECT Name, CountryCode
      FROM City
      WHERE ID = 456) AS 'Es Londres?';
```

También lo podemos comparar del siguiente modo:

```
SELECT (SELECT ID, Name, CountryCode
      FROM City
      WHERE ID = 456)
      = (SELECT ID, Name, CountryCode
      FROM City
      WHERE CountryCode = 'GBR'
      AND Name = 'London') AS 'Son iguales?';
```

En el siguiente ejemplo se observa que una subconsulta de fila siempre da como resultado una fila, aunque la expresión de la consulta entre paréntesis dé lugar a un conjunto vacío:

```
SELECT (NULL, NULL) <=> (SELECT ID, Name FROM City LIMIT 0);
```

Cuidado en la construcción de la expresión: si comparamos columnas diferentes dará lugar a un error. Un ejemplo:

```
SELECT (456,'London','GBR') = (SELECT  Name, CountryCode FROM City);
```

14.2.3 Subconsultas de tabla

Estas subconsultas actuan como tablas (de solo lectura). Dan como resultado un conjunto que contiene cero o mas filas con una o mas columnas y pueden aparecer en dos contextos distintos:

- En la cláusula FROM de una consulta externa.
- Como operandos derechos de los operadores lógicos IN y EXISTS, o como operandos derechos de un operador normal de comparación cuantificado con ALL, ANY y SOME.

El número de columnas seleccionadas o el número de filas devueltas no afecta al estado de la subconsulta como subconsulta de tabla. El hecho de que la misma aparezca en la cláusula FROM o que sea utilizada como operando con uno de estos operadores hace que sea tratada como una tabla (de solo lectura).

Subconsultas en la cláusula FROM

El conjunto resultante de una subconsulta en la cláusula FROM es tratada de la misma manera que los resultados obtenidos a partir de tablas base o vistas que hayan sido referidas en la cláusula FROM. Veamos un ejemplo:

```
SELECT *
FROM (SELECT Code, Name
      FROM Country
      WHERE IndepYear IS NOT NULL) AS PaisesIndependientes;
```

Es obligatorio el uso de ALIAS para las subconsultas. Si se omite:

```
AS PaisesIndependientes;
```

Da un error.

Las subconsultas con cláusulas FROM son especialmente útiles para calcular acumulados de acumulados. Veamos un ejemplo con el promedio de sumas de la población de cada continente:

```
SELECT AVG(SumaPob) AS 'Media Población de todos los continentes'
FROM (
      SELECT Continent, SUM(Population) AS SumaPob
      FROM Country
      GROUP BY Continent
      ) AS Total;
```

Para verlo mejor, habría que ver la suma de la población por cada Continente (la parte interna de la subconsulta).

```
SELECT Continent, SUM(Population) AS SumaPob
FROM Country
GROUP BY Continent;
```

14.3 Operadores de la subconsultas de tabla

Vamos a ver el uso y la sintaxis de las subconsultas de tabla como operadores derechos de una serie de operadores de comparación. Sólo algunos operadores específicos pueden aceptar una subconsulta de tabla como operador derecho:

- Los operadores IN y EXISTS
- Los operadores normales de comparación (excepto <=>), dado que uno de los cuantificadores ALL, ANY o SOME erán usados para especificar cómo será aplicado cada operador.

Todos los operadores devuelven un resultado booleano. No hay nada especial en estos operadores además del hecho de requerir que la subconsulta este del lado derecho.
Aunque estos operadores pueden usarse en cualquier lugar dentro de la sentencia, donde también se colocaria una expresión escalar, el uso de una subconsulta como operando derecho para estos operadores está limitado esencialmente a la cláusula WHERE.

14.3.1 El operador IN

Una subconsulta de tabla puede usarse como operando derecho del operador IN. En este caso, el operador da verdadero si en el conjunto resultante derivado de la subconsulta existe al menos una ocurrencia igual al operando izquierdo.
Si no hay tal ocurrencia, el operador IN da falso.

Si la subconsulta de tabla selecciona solo una columna (subconsulta de columna), el operando izquierdo tiene que ser una expresión de valor escalar. Veamos un ejemplo:

```
SELECT * FROM City
WHERE CountryCode IN (
      SELECT Code
      FROM Country
      WHERE Continent  = 'Asia'
      );
```

Obsérvese como en este caso, el resultado de la subconsulta interna SI da mas de una fila. Al usar el operador IN esto está permitido.
Por otro lado si la subconsulta de tabla selecciona mas de un columna, el operando izquierdo tiene que ser un constructor de fila. En este caso, verifica la igualdad con la fila que actúa como operando izquierdo, siendo efectuadas las comparaciones por pares de columnas con las filas del conjunto resultante.

Veamos un ejemplo de comparación por pares de columnas:

```
SELECT *
FROM City
WHERE (CountryCode, Name) IN (
      SELECT Code, Name
      FROM Country
      WHERE Continent = 'Asia');
```

Da como resultado 3 registros, ¿porque? Pues porque al comparar el CountryCode y el Name para City y el Name para el Country solamente nos saldrá aquellas ciudades-estado de Asia.

```
SELECT *
FROM City
WHERE (CountryCode, Name) IN (
      SELECT Code, Name
      FROM Country
      WHERE Continent = 'Europe');
```

Uso de NOT IN

Justo lo contrario que lo anterior es el uso del operador NOT IN. Por ejemplo, queremos ver las ciudades del mundo que NO están en Asia:

```
SELECT *
FROM City
WHERE (CountryCode) NOT IN (
      SELECT Code
      FROM Country
      WHERE Continent = 'Asia');
```

El operador IN y el valor NULL

Hay casos en los cuales es imposible determinar si el conjunto resultante formado por el operando derecho contiene alguna ocurrencia igual al operando izquierdo.
En este caso, IN da NULL.

Existen dos escenarios distintos en los cuales IN puede dar este resultado:

- Si el operando izquierdo es NULL y el conjunto resultante formado por el operando derecho no está vacio, IN da NULL. Como NULL no puede ser comparado con ningún valor (ni siquiera con el propio NULL), no se puede determinar si ocurre en el conjunto resultante derivado de la subconsulta. Sin embargo, si el conjunto resultante está vacio, entonces IN siempre da falso porque en este caso, por definición, no puede haber ocurrencias iguales al operando izquierdo.

- Si el conjunto resulante no es vacío y no hay ocurrencias iguales al operando izquierdo, entonces IN da NULL, en caso de que el conjunto resultante contenga al menos una fila en la cual al menos una de las columnas sea NULL.

14.3.2 El operador EXISTS

Este operador acepta un argumento individual del lado derecho que debe ser una subconsulta de tabla. Si el conjunto resultante de la subconsulta contiene al menos una fila, entonces la expresión EXISTS es verdadera. En cualquier otro caso, será falsa.

Veamos un ejemplo; queremos ver las capitales de Europa:

```
SELECT *
FROM City
WHERE EXISTS (
      SELECT NULL
      FROM Country
      WHERE Capital = ID
      AND Continent = 'Europe');
```

El resultado del operador EXISTS no es influenciado por los valores reales seleccionados por la subconsulta de tabla. Aquí, la subconsulta de tabla selecciona el valor NULL, pero podría haber seleccionado el valor literal o una expresión de columnas; todo esto es indiferente para EXISTS. Lo único que afecta al resultado del operador EXISTS es si la expresión de la consulta que conforma la subconsulta de tabla obtiene al menos una fila, independientemente de las columnas o sus valores.

Uso de NOT EXISTS

El operador EXISTS puede ser negado usando como prefijo NOT. La construcción NOT EXISTS es verdadera (TRUE) solo si la subconsulta que actúa como argumento resulta de un conjunto vacío; si no, es falsa (FALSE).
Veamos un ejemplo; queremos ver todos los países donde NO se habla Inglés:

```
SELECT Name
FROM Country
WHERE NOT EXISTS (
      SELECT NULL
      FROM CountryLanguage
      WHERE CountryCode = Code
      AND Language = 'English');
```

En este caso la consulta externa obtiene una lista de todos los países. Para cada fila de Country se compara luego con las condiciones de CountryLanguage (el Lenguaje sea Inglés) y se vez aquellas que no coinciden.

14.3.3 ALL, ANY y SOME

Estos operadores se llaman cuantificadores. Vienen a suplir las carencias de los operadores de comparación normales como el = (ya vimos el error de igualar una expresión de consulta con una subconsulta con múltiples filas). De este modo, tendremos los siguiente cuantificadores:

- `ALL`: Indica que la comparación es verdadera si el operador es aplicado a todos los elementos en el conjunto resultante de la subconsulta y devuelve verdadero en todos los casos.
- `ANY`: Indica que el operador debe ser aplicado a los elementos en el conjunto resulante de la subconsulta hasta que al menos una comparación resulte verdadera. Devuelve falso si ninguna de las comparaciones fue verdadera.
- `SOME`: es un alias de ANY y tiene el mismo ejemplo. Sin embargo, en algunos casos la sentencia es mas sencilla de leer cuando se usa SOME.

La sintaxis de los cuantificadores es:

```
<operando_izquierdo> <operador_comparación> <cuantificador>
<subconsulta_tabla>
```

Por ejemplo: ¿Finland es el nombre de un país?

```
SELECT 'Finland' =
      ANY (SELECT Name FROM Country);
```

La respuesta es `VERDADERO (1)`.

Si quitamos el `ANY` da `ERROR`.

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Sin embargo, si sustituimos ANY por ALL dará FALSO, porque TODOS los países no se llaman Finland:

```
SELECT 'Finland' =
      ALL (SELECT Name FROM Country);
```

Ahora veamos otro ejemplo; queremos tener una lista de aquellos países cuya población total sea igual que la población de sus ciudades:

```
SELECT Name
FROM Country
WHERE Population > ALL (
      SELECT Population
      FROM City);
```

Salen 73 registros (curioso, si).

SOME contra ANY

Ambos son equivalentes, pero a efectos de lectura, a veces es preferible usar el cuantificador SOME. Por ejemplo: Queremos obtener todas las ciudades que tienen una población mayor que la de algún país con mas de 1000000 de habitantes:

```
SELECT *
FROM City
WHERE Population > SOME (
      SELECT Population
      FROM Country
      WHERE Population>1000000);
```

Alternativas a ANY y ALL

Como ya hemos visto, ANY puede llevar a confusión. Pero el caso de ALL no se queda atrás, por lo que existen alternativas al uso de ambos, igualmente válidos.

Expresión de Operador Cuantificado	Alternativa
Escalar > ANY (SELECT columna...)	Escalar > (SELECT MIN (columna) ...)
Escalar < ANY (SELECT columna...)	Escalar < (SELECT MAX (columna) ...)
Escalar > ALL (SELECT columna...)	Escalar > (SELECT MAX (columna) ...)
Escalar < ALL (SELECT columna...)	Escalar < (SELECT MIN (columna) ...)
Escalar = ANY (SELECT columna...)	Escalar IN (SELECT (columna) ...)
Escalar <> ALL (SELECT columna...)	Escalar NOT IN (SELECT (columna) ...)

Volvemos al ejemplo de la página anterior (cambiando SOME por ANY): Ciudades del mundo con mas de 1 millón de habitantes.

```
SELECT *
FROM City
WHERE Population > ANY (
      SELECT Population
      FROM Country
      WHERE Population>1000000);
```

Podemos cambiarlo por:

```
SELECT *
FROM City
WHERE Population > (
      SELECT MIN(Population)
      FROM Country
      WHERE Population>1000000);
```

Queda al gusto de cada uno elegir el que sea mas fácil de leer e implementar.

14.4 Subconsultas correlacionadas y no collelacionadas.

Las subconsultas pueden ser Correlacionadas y No Correlacionadas. La diferencia entre ambos está en si la expresión SELECT entre paréntesis que conforma la subconsulta contiene expresiones derivadas de la sentencia contenedora (externa). Esta categorización es independiente de la distinción entre subconsultas escalares, de fila y de tabla que hemos visto antes, la cual tiene que con la forma y la función del conjunto resultante de la subconsulta.

14.4.1 Subconsultas no correlacionadas.

Una subconsulta es No Correlacionada si la expresión de la consulta que se encuentra entre paréntesis no se refiere a una expresión que depende de la sentencia externa. Veamos un ejemplo en esta subconsulta contenida en un SELECT:

```
SELECT * FROM City
WHERE CountryCode IN (
      SELECT Code
      FROM Country
      WHERE Continent = 'North America');
```

En este caso, la subconsulta No correlacionada está contenida en si misma. Es decir, puede ejecutarse como una consulta independiente:

```
SELECT Code
      FROM Country
      WHERE Continent = 'North America';
```

14.4.2 Subconsultas correlacionadas.

Estamos justo en lado opuesto. Las consultas Correlacionadas contienen una o mas expresiones derivadas de una parte de la sentencia que aparece fuera del limite de la subconsulta. Por ejemplo, veamos aquellos países sin ciudades:

```
SELECT Name
FROM Country
WHERE NOT EXISTS (
      SELECT NULL
      FROM City
      WHERE CountryCode = Code);
```

El campo Code que aparece en la subconsulta viene de Country y no de City, por lo que la subconsulta no puede ejecutarse independientemente:

```
SELECT NULL
      FROM City
      WHERE CountryCode = Code;
ERROR 1054 (42S22): Unknown column 'Code' in 'where clause'
```

Alcance

Es importante reseñar que la subconsulta (lo que va entre parentesis) se ejecuta en principio de modo independiente. Veamos un ejemplo:

```
SELECT *
FROM City
WHERE CountryCode IN (
      SELECT Code
      FROM Country
      WHERE Name = 'Belgium');
```

Name, ¿a que se refiere, a Country o a City? Pues a Country, porque el alcance llega hasta los limites de la subconsulta. Al ejecutar la sentencia no da ningún error de ambigüedad.

Laboratorio de prácticas

Vamos a repasar lo que llevamos hasta ahora con algunos ejercicios:

1. Nombres de los países que se encuentran en la región de los países nórdicos (Nordic Countries) junto con el número de ciudades por país. **[Subconsulta Correlacionada]**

Solución:

```
SELECT Country.Name,
      (SELECT Count(*)
      FROM City
      WHERE CountryCode = Country.Code) AS NumeroCiudades
FROM Country
WHERE Region = 'Nordic Countries';
```

2. Efectuar una consulta anidadad y No Correlacionada que obtenga la suma de la población de todas las regiones del mundo.

Solución:

```
SELECT Region, Totales
FROM (SELECT Region, SUM(Population) AS Totales
FROM Country
GROUP BY Region) AS Resultado;
```

3. Efectuar una consulta con otra anidada (subconsulta) que obtenga las ciudades del mundo con poblaciones mayores que la ciudad de New York, ordenadas por población.

Solución:

```
SELECT Name, Population
FROM City
WHERE Population > (SELECT Population
      FROM City
      WHERE Name = 'New York')
ORDER BY Population DESC;
```

(si, ya, no aparece Tokyo, por ejemplo, como podemos comprobar:
`SELECT * FROM City WHERE Name='Tokyo';`)

4. Ver los distintos idiomas que se hablan en Africa.

Solución:

```
SELECT DISTINCT Language
FROM CountryLanguage
WHERE CountryCode IN (
      SELECT Code
      FROM Country
      WHERE Continent = 'Africa');
```

14.4.3 Otros usos de las subconsultas

Hasta ahora hemos usado las subconsultas con sentencias SELECT. Pero también podemos usarlos en DELETE y UPDATE.

NOTA: Como en ocasiones anteriores, no soy partidario de usar DELETE en las tablas originales de world. Usaremos las copias que ya creamos en el apartado **13.7.2**.

Por ejemplo, vamos a borrar las ciudades de aquellos países cuya esperanza de vida no supere los 70 años:

```
DELETE
FROM CityCopia
WHERE CountryCode IN (
      SELECT Code
      FROM CountryCopia
      WHERE LifeExpectancy < 70.0);
```

El resultado:

```
Query OK, 1846 rows affected (1.02 sec)
```

Otro ejemplo: Vamos a asignar a los distintos países la población de la suma de sus principales ciudades (el resultado puede llegar a ser muy curioso):

```
UPDATE CountryCopia
SET Population = (
      SELECT SUM(Population)
      FROM CityCopia
      WHERE CountryCode = Code);
```

El Resultado:

```
Query OK, 232 rows affected, 117 warnings (1.00 sec)
Rows matched: 239  Changed: 232  Warnings: 117
```

NOTA: Los Warnings es porque ahora muchos países tiene un valor NULL como Population. Se cambian a 0 (de ahí el Warning).

Ahora comprobemos los datos (por ejemplo para Europa)

```
SELECT Name, Population
FROM CountryCopia
WHERE Continent = 'Europe';
```

Vemos muchos países con población 0 (hemos eliminado sus ciudades con el DELETE de la esperanza de vida) y el resto con bastante menos población (España tiene ahora 16 millones).

14.5 Conversion de subconsultas en joins.

En general, MySQL puede optimizar mejor una consulta JOIN que una subconsulta. Por tanto el JOIN suele ser mas eficiente. De hecho, si un SELECT escrito como una subconsulta tarda mucho en ejecutarse, puede hacerse el intento de escribirla como un JOIN para comprobar que su rendimiento es mejor.

14.5.1 Reescribir IN y NOT IN

Especificamente, una subconsulta que busca coincidencias ente tablas puede ser reescrita a menudo como un JOIN.

Reescribir IN como un INNER JOIN

Veamoslo con un ejemplo; consideremos una consulta que busque todos los países donde se habla un idioma en particular, por ejemplo el Español:

```
SELECT Name
FROM Country
WHERE Code IN (
      SELECT CountryCode
      FROM CountryLanguage
      WHERE Language = 'Spanish');
```

Vamos a reescribirlo como un INNER JOIN siguiendo estos pasos:

a. Mover la tabla usada en la subconsulta a la cláusula FROM de la consulta externa usando INNER JOIN.
b. Mover la comparación IN y la lista SELECT de la subconsulta desde la cláusula WHERE a la cláusula ON del JOIN.
c. Reescribir el IN como un operador de igualdad (=).
d. Mover la cláusula WHERE de la subconsulta a la cláusula WHERE de la consulta JOIN.

El resultado será:

```
SELECT Name
FROM Country
INNER JOIN CountryLanguage
ON Code = CountryCode
WHERE Language = 'Spanish';
```

Aunque no es el caso, es posible que salgan filas duplicadas al juntar ambas tablas con el INNER JOIN. En tal caso, sólo debemos añadir la cláusula DISTINCT (aquí innecesario)
SELECT DISTINCT Name...

Reescribir NOT IN como un JOIN EXTERNO

Se puede reescribir una subconsulta que usa NOT IN como un LEFT JOIN o RIGHT JOIN.
Veamos un ejemplo; Ciudades que NO son capital de España:

```
SELECT Name
FROM City
WHERE (ID) NOT IN (
      SELECT Capital
      FROM Country
      WHERE Capital IS NOT NULL)
AND CountryCode = 'Esp';
```

Lo anterior puede ser reescrito como un LEFT JOIN DEL siguiente modo:

1. Mover la tabla usada en la subconsulta a la cláusula FROM de la consulta externa usando INNER JOIN (en este caso Country).
2. Mover la comparación NOT IN y la lista SELECT de la subconsulta desde la cláusula WHERE a la cláusula ON del JOIN.
3. Reescribir el NOT IN como un operador de igualdad.
4. Agregar una condición a la cláusula WHERE de la consulta JOIN que requiera que no exista una fila correspondiente en la tabla con la que hace el JOIN.

NOTA: Adicionalmente, hay que agregar al SELECT inicial el nombre de la tabla para evitar ambigüedades en el nombre.

```
SELECT City.Name
FROM City
LEFT JOIN Country
ON ID=Capital
WHERE Capital IS NULL
AND CountryCode = 'ESP';
```

14.5.2 Limitaciones para reescribir subconsultas como joins

Algunas subconsultas no pueden reescribirse como JOINs mientras que otras si lo aceptan pero pueden resultar en una consulta con un rendimiento inferior que la consulta original que usaba una subconsulta.

Acumular agregados usando subconsultas con la cláusula FROM

Una subconsulta en la cláusula FROM puede ser usada convenientemente para calcular un agregado. Como el resultado es tratado básicamente como otra tabla cualquiera, la consulta externa puede aplicar de nuevo una función de agregación al resultado de la subconsulta. Este mecanismo permite calcular agregados a partir de datos ya agregados.

Anteriormente, se habló de una consulta que usaba este método para calcular la población promedio por continente. La siguiente consulta es también otro ejemplo que usa la función de agregación GROUP_CONCAT para obtener una visión jerarquía de los países de Europa por Región (**NOTA:** Separo la consulta para una mejor comprensión).

```
SELECT GROUP_CONCAT('\n', Continent, Regiones
      ORDER BY CAST(Continent AS CHAR(30))
      SEPARATOR '')  AS Continente

FROM (
      SELECT  Continent,
      GROUP_CONCAT('\n  ', Region, Países
      ORDER BY Region SEPARATOR '')  AS Regiones

      FROM (
      SELECT Continent, Region,
      GROUP_CONCAT('\n    ', Name
      ORDER BY Name SEPARATOR '')  AS Países
      FROM Country
      GROUP BY Continent, Region
      ) AS Países

GROUP BY Continent
) AS Regiones
WHERE Continent ='Europe' \G
```

Reportar agregados de distintas filas hijas
Algunas veces es necesario clacular multiples agregados en diferentes tablas. Por ejemplo podría necesitarse calcular el número de idiomas y el número de ciudades por país. En este caso partícular, resulta que es realmente posible hacerlo sin subconsultas:

```
SELECT  Country.Name,
COUNT(DISTINCT City.ID)  AS NumeroCiudades,
COUNT(DISTINCT Lang.Language) AS NúmeroLenguas
FROM  Country
      LEFT JOIN City
      ON  Country.Code = City.CountryCode
      LEFT JOIN CountryLanguage AS Lang
      ON  Country.Code = Lang.CountryCode
GROUP BY  Country.Name;
```

El problema de realizar esto es que hay que hacer un producto cartesiano parcial, multiplicando las ciudades por las lenguas. Para evitarlo se ha usado COUNT (DISTINCT).

De todos modos lo anterior se puede reescribir como una subconsulta que, además, es mas eficiente que el JOIN que acabamos de ver. La primer forma es reemplazando cada JOIN con una subconsulta correlacionada en la lista SELECT:

```
SELECT Country.Name,
      (SELECT COUNT(ID)
      FROM   City
      WHERE  CountryCode = Code) AS NúmeroCiudades,
      (SELECT COUNT(Language)
      FROM   CountryLanguage
      WHERE  CountryCode = Code) AS  NúmeroLenguas
FROM   Country;
```

NOTA: En mis pruebas los resultados son 0.07sgs para los LEFT JOIN y 0.06sgs para las subconsultas. Mínima diferencia, si, pero en local y con un buen equipo.

En este caso hemos eliminado los LEFT JOIN y se han sustituido los DISTINCT por una subconsulta escalar (COUNT(ID)). La condición usada por el JOIN es ahora usada para enlazar el resultado de la subconsulta con la fila actual de la consulta externa (subconsulta correlacionada).

La segunda alternativa usa el diseño de la consulta original con los LEFT JOINs. Sin embargo, esta vez se usan subconsultas en la cláusula FROM para hacer un JOIN con la tabla Country. Las subconsultas ahora acumulan previamente los datos por país antes de hacer el JOIN. Así se asegura que exista, como máxima, una fila disponible para hacer JOIN con la fila Country, evitando así el producto cartesiano parcial que ocurría en el JOIN original.

```
SELECT  Country.Name,
      IFNULL(Ciudades.NúmeroCiudades , 0) AS NúmeroCiudades,
      IFNULL(Lenguas.NúmeroLenguas, 0)  AS NúmeroLenguas
FROM  Country

      LEFT JOIN (SELECT  CountryCode , COUNT(ID) AS NúmeroCiudades
      FROM  City
      GROUP BY CountryCode) AS Ciudades
      ON Country.Code = Ciudades.CountryCode

      LEFT JOIN (SELECT CountryCode, COUNT(Language) AS NúmeroLenguas
      FROM     CountryLanguage
      GROUP BY CountryCode) AS Lenguas
      ON  Country.Code = Lenguas.CountryCode;
```

En este caso, se pasa a sólo 0.02seg para la consulta.
NOTA: Mucho cuidado con los alias que hay que respetarlos en las definición de la consulta.

15 VISTAS

15.1 ¿Que son las vistas?

Es un objeto de la BBDD definido en términos de una consulta (es decir, es una expresión SELECT) que recupera los datos que la vista genere. Las vistas son también llamadas Tablas Virtuales y puede ser usadas para seleccionar tablas regulares (llamadas tablas base) u otras vistas. En algunos casos, una vista es modificable y puede usarse con sentencias como UPDATE, DELETE o INSERT para actualizar el contenido de la tabla base subyacente.

Las vistas brindan una serie de beneficios respecto a la selección de datos sobre tablas base:

- Simplifica el acceso a los datos
  - Una vista puede usarse para realizar un cálculo y mostrar el resultado.
  - Una vista puede usarse para seleccionar un conjunto restringido de filas por medio de una cláusula apropiada de WHERE, o seleccionar sólo un subconjunto de las columnas de una tabla.
  - Puede usarse para seleccionar datos de varias tablas mediante JOINs o uniones.

Una vista realiza estas operaciones de forma automática, sin que los usuarios necesiten especificar la expresión sobre la que está basado el cálculo, ni las condiciones que limitan las filas en la cláusula WHERE , ni las condiciones para hacer coincidir tabla en un JOIN.

- Las vistas pueden ser usadas para mostrar el contenido de tablas de forma distinta para cada usuario, de modo que cada uno sólo vea los datos pertinentes a sus actividades. Esta capacidad incide favorablemente en el nivel de seguridad al esconderle a los usuarios la información que no deben modificar o la que no pueden acceder. Reduce la distracción, pues las columnas irrelevantes no son mostradas.

- Si se necesita cambiar la estructura de las tablas para ajustarlas a ciertas aplicaciones una vista puede conservar la apariencia de la estructura original de la tabla a fin de minimizar el impacto en otras aplicaciones. Por ejemplo, si se parte una tabla en dos nuevas tablas, una vista puede crearse con el nombre de la tabla original y definir la selección de datos a partir de las dos nuevas tablas para que la vista muestre la estructura original de la tabla.

15.2 Crear una vista.

15.2.1 La sentencia CREATE VIEW

Para crear una vista usaremos CREATE VIEW. Su sintaxis es:

```
CREATE [OR REPLACE] [ALGORITHM = <tipo_algoritmo>]
VIEW nombre_vista [(lista_columnas)]
AS expresion_select
[WITH [CASCADED | LOCAL]  CHECK OPTION]]
```

Sus componentes son:

- nombre_vista: se puede crear en la bbdd actual o sobre otra (bbdd.nombre_vista)
- lista_columnas: da los nombres a las columnas de la vista (no iguales a las originales)
- expresion_select: La sentencia se puede hacer sobre tablas base o sobre otras vistas.
- [OR REPLACE] Elimina una vista existente con el mismo nombre.
- ALGORITHM: indica el algoritmo de procesamiento que se usará para invocar la vista. Si es especificada, debe ser una de estas:
  - MERGE: Para ejecutar la vista, el texto SQL es expandido en el texto SQL de la consulta que referencia la vista, reescribiéndola.
  - TEMPTABLE: La consulta se ejecuta y su resultado va a una tabla temporal.
  - UNDEFINED: La elección del algoritmo se pospone hasta su ejecución.

- Con WITH CHECK OPTION, todos los cambios hechos a los datos de la vista son verificados; así, las filas nuevas o actualizadas satisfacen la condición WHERE en la expresión de la consulta de la vista.

Veamos un ejemplo; Creamos VistaCity con el ID y Name de la tabla City

```
USE world;
CREATE VIEW VistaCity
AS SELECT ID, Name
FROM City;
```

Y luego visualizamos la vista:

```
SELECT * FROM VistaCity;
```

Como ya hemos visto, se pueden usar como en este caso los nombres de campo originales o unos específicos para la vista. Eso si, con dos condiciones:

- El número de nombres de campos coincide por defecto con el original, excepto cuando añadimos campos agregados.
- No podemos repetir el mismo nombre. Por ejemplo:

```
CREATE VIEW v AS
SELECT Country.Name, City.Name
FROM Country, City
WHERE Code = CountryCode;
ERROR 1060 (42S21): Duplicate column name 'Name'
```

Aunque son de tablas diferentes, el nombre es el mismo (Name).
Tenemos dos alternativas para evitar esto:

a) Usar ALIAS para los campos

```
CREATE VIEW vistaCampos AS
SELECT Country.Name AS CountryName,
City.Name AS CityName
FROM Country, City
WHERE Code = CountryCode;
```

b) Ponerle un nombre específico a las columnas a partir de las reales:

```
CREATE VIEW vistaCampos1 (País, Ciudad) AS
SELECT Country.Name, City.Name
FROM Country, City
WHERE Code = CountryCode;
```

Disponer de nombres explícitos de columnas en una vista simplifica el uso de columnas en expresiones de cálculo. Por defecto, los primeros 64 caracteres de texto de la expresión son utilizados para el nombre de tales columnas. Normalmente esos identificadores son difíciles de manejar.

El siguiente ejemplo crea una vista cuya segunda columna es creada a partir de una expresión agregada:

```
CREATE VIEW LenguasHabladasPorPais
(País, Lenguas) AS
SELECT Name, COUNT(Language)
FROM Country, CountryLanguage
WHERE Code = CountryCode
GROUP BY Name;

SELECT * FROM  LenguasHabladasPorPais;
```

15.3 Vistas actualizables

Una vista es actualizable si puede usarse con sentencias UPDATE y DELETE para modificar la tabla base subyacente. No todas las vistas son actualizables. Por ejemplo, podría actualizarse una tabla pero no podría actualizarse una vista de la tabla si la misma está definida en términos de valores agregados, calculados a partir de la tabla .

Esto tiene una explicación: como no se requiere que cada fila de la vista corresponde a una única fila de una tabla base, MySQL no podría determinar qué fila tiene que modificar.

Las condiciones básicas para que una vista sea actualizable son que debe existir una relación uno a uno entre las filas de la vista y las filas de la tabla base y que las columnas de la vista a modificar deben estar definidas como simples referencias a columnas, no como expresiones. También existen otras condiciones pero no las veremos aún.

NOTA: Como en ocasiones anteriores, prefiero usar Tablas Copias de las originales de world.

```
DROP TABLE IF EXISTS CityCopia;
DROP TABLE IF EXISTS CountryCopia;
DROP TABLE IF EXISTS CountryLanguageCopia;
CREATE TABLE CityCopia AS SELECT * FROM City;
CREATE TABLE CountryCopia AS SELECT * FROM Country;
CREATE TABLE CountryLanguageCopia AS SELECT * FROM CountryLanguage;
```

En primer lugar nos creamos la vista:

```
CREATE VIEW PobEuropa (País, Población) AS
SELECT Name, Population
FROM CountryCopia
WHERE Continent = 'Europe';
```

Comprobamos los datos:

```
SELECT * FROM PobEuropa
WHERE País = 'San Marino';
```

Actualizamos la población de San Marino

```
UPDATE PobEuropa
SET Población = Población + 1000
WHERE País = 'San Marino';
```

Vemos el cambio:

```
SELECT * FROM PobEuropa
WHERE Name = 'San Marino';
```

Comprobamos que no solo la vista se modificar, si no también la tabla original:

```
SELECT Name, Population
FROM CountryCopia
WHERE Name = 'San Marino';
```

Y repetimos el proceso anterior con un borrado del registro:

```
DELETE FROM PobEuropa
WHERE País = 'San Marino';

SELECT * FROM PobEuropa
WHERE País = 'San Marino';
SELECT * FROM CountryCopia
WHERE Name = 'San Marino';
```

15.3.1 Vistas insertables

Una vista actualizable es insertable si, además de ser actualizable, cumple con las siguientes condiciones adicionales respecto a las columnas de la vista:

- No pueden existir nombres de columna duplicados
- Ninguna columna en la tabla base de la que se deriva la vista puede tener valores por defecto.
- Las columnas deben ser referencias simples a columnas y no derivadas de una expresión (llamadas también columnas derivadas). Veamos algunos ejemplos de este tipo de columnas:
  - `3.14159`
  - `Col1+3`
  - `UPPER(col2)`
  - `Col3/Col4`
  - `(<suconsulta>)`

Una vista que tiene una combinación de referencias simples a columnas y columnas derivadas no es insertable, pero puede ser actualizable si se actualizan solo aquellas columnas que no están derivadas. Veamos un ejemplo:

```
USE test;
DROP TABLE IF EXISTS t;
CREATE TABLE t (col1 INT(11));
INSERT INTO t VALUES (15);
```

Y ahora la vista:

```
CREATE VIEW v AS
SELECT col1, 1 AS col2
FROM t;
```

La vista no es insertable porque col2 es derivada de una expresión (1) pero es actualizable porque la actualización no implica a col2. Así, esto esta permitido:

```
UPDATE v SET col1 = 0;
```

Pero esto no:

```
UPDATE v SET col2 = 0;
ERROR 1348 (HY000): Column 'col2' is not updatable
```

Es posible que una vista de múltiples tablas sea actualizable, con las siguientes restricciones:

- Puede ser procesada con el algoritmo MERGE.
- La vista debe usar un JOIN INTERNO (no uno EXTERNO o UNION)
- Solo puede actualizarse una tabla individual en la definición de la vista.
- Las vistas que usan UNION ALL no pueden actualizarse aunque en teoría fuera posible
- La sentencia INSERT solo puede funcionar si se inserta en una tabla individual. La sentencia DELETE no está soportada.

Si una tabla continene una columna AUTO_INCREMENT, la inserción de filas en una vista insertable que no incluya la columna AUTO_INCREMENT no cambia el valor de la función LAST_INSERT_ID(), porque los efectos colaterales de insertar valores por defecto en las columnas que no son parte de la vista, no deberian ser visibles.

NOTA: Para que una vista sea insertable, previamente debe ser actualizable.

Un ejemplo de cómo insertar valores en una vista:

```
CREATE TABLE t1 (
  col1 INT(11)
);

INSERT INTO t VALUES (15);

CREATE VIEW v1 AS
SELECT col1
FROM t;

INSERT INTO v1 VALUES (20);
SELECT * FROM v1;
```

La vista es actualizable e insertable puesto que no tiene ningún campo agregado o calculado.

15.3.2 WITH CHECK OPTION

Si una vista es actualizable, puede usarse la cláusula `WITH CHECK OPTION` para establecer una restricción sobre las modificaciones permitidas. Está cláusula hace que las condiciones de `WHERE` en la definición de la vista sean verificadas cuando se intenta efectuar alguna modificación.

- Un `UPDATE` a una fila existente es permitido solo si la cláusula `WHERE` sigue siendo verdadera con la fila resultante.
- Un `INSERT` es permitido solo si la cláusula WHERE sigue siendo verdadera con la nueva fila.

Es decir, con `WITH CHECK OPTION` se garantiza que no se puede actualizar una fila ni insertar una nueva si la vista resultante no la incluye.

Veamos un ejemplo; Países del mundo con mas de 100 millones de habitantes:

```
USE world;
CREATE VIEW PobGrande
AS SELECT Name, Population FROM CountryCopia
WHERE Population >= 100000000
WITH CHECK OPTION;

SELECT * FROM PobGrande;
```

Con `WITH CHECK OPTION` podemos realizar algunas modificaciones, pero nos prohibe otras. Por ejemplo incrementar la población en un país concreto:

```
UPDATE PobGrande
SET Population = Population + 1
WHERE Name = 'Nigeria';

SELECT * FROM PobGrande WHERE Name = 'Nigeria';
```

También podemos reducir el valor de la población pero sólo si no bajamos de los 100 millones; si lo intentamos, nos dará un error:

```
UPDATE PobGrande
SET Population = 99999999
WHERE Name = 'Nigeria';
ERROR 1369 (HY000): CHECK OPTION failed 'world.PobGrande'
```

15.4 Manejo de Vistas

15.4.1 Verificación de Vistas

Cuando se define una vista, cualquier objeto referenciado en la misma, ya sea una tabla, otra vista o una columna, debe existir. Sin embargo, una vista puede llegar a ser inválida si una tabla, vista o columna de la cual depende, es eliminada o modificada. Para verificar este tipo de problemas está la sentencia CHECK TABLE.
Veamos un ejemplo; Nos creamos una tabla de prueba:

```
USE test;
CREATE TABLE t2 (i INT);
```

Luego nos creamos una vista (cerciorándonos que no existe ya)

```
DROP VIEW IF EXISTS v2;
CREATE VIEW v2 AS
SELECT i FROM t2;
```

Cambiamos el nombre de la tabla origen:

```
RENAME TABLE t2 TO t3;
```

Y comprobamos que pasa:

```
CHECK TABLE v2 \G
```

El resultado es contundente:

```
*************************** 1. row ***************************
   Table: test.v
      Op: check
Msg_type: Error
Msg_text: Table 'test.t1' doesn't exist
*************************** 2. row ***************************
   Table: test.v
      Op: check
Msg_type: Error
Msg_text: View 'test.v' references invalid table(s) or column(s) or
function(s) or definer/invoker of view lack rights to use them
*************************** 3. row ***************************
   Table: test.v
      Op: check
Msg_type: error
Msg_text: Corrupt
```

15.4.2 Modificación de Vistas

Para modificar la definición de una vista se usa la sentencia ALTER VIEW, que cambia la definición actual por una nueva. Si la tabla referenciada no existe, ocurre un error. Es muy similar a CREATE VIEW, aunque la opción OR REPLACE no puede ser usada.

Veamos un ejemplo siguendo el del apartado anterior:

```
ALTER VIEW v AS
SELECT i FROM t2;
```

Y probamos:

```
SELECT * FROM v;
CHECK TABLE v\G
*************************** 1. row ***************************
   Table: test.v
      Op: check
Msg_type: status
Msg_text: OK
```

15.4.3 Eliminación de Vistas

Para eliminar una o mas vistas, se usa la sentencia DROP VIEW
La sintaxis será:

```
DROP VIEW [IF EXISTS] nombre_vista [,nombre_vista]...
```

Si la vista especificada NO existe, ocurre un error excepto si se emplea IF EXISTS, que dará un warning. La cláusula IF EXISTS es propia de MySQL, fuera del estándar SQL.
Veamos un ejemplo:

```
DROP VIEW IF EXISTS v,v1;
```

Vemos las alertas:

```
SHOW WARNINGS;
+-------+------+------------------------+
| Level | Code | Message                |
+-------+------+------------------------+
| Note  | 1051 | Unknown table 'test.v1' |
+-------+------+------------------------+
```

15.5 Obtención de metadatos sobre vistas.

15.5.1 Usar INFORMATION_SCHEMA

La BBDD INFORMATION_SCHEMA contiene una tabla TABLES que contiene una fila por cada tabla accesible por el usuario. La columna TABLE_TYPE de esta tabla indica su tipo y contiene el valor VIEW en el caso de que la tabla sea una vista. Sin embargo, la mayoría de las columnas en la tabla TABLES están asociadas a tablas base. Esta tabla no contiene casi ninguna columna con información sobre las vistas propiamente dichas.

La BBDD INFORMATION_SCHEMA también contiene una tabla VIEWS que contiene metadatos específicos sobre las vistas. Entre estos está si es actualizable (IF_UPDATABLE).

Por ejemplo, para mostrar la información de la vista VistaCity se usará esta sentencia:

```
USE INFORMATION_SCHEMA;
SELECT * FROM information_schema.VIEWS
WHERE TABLE_NAME = 'VistaCity'
AND TABLE_SCHEMA = 'world' \G
```

Dentro del INFORMATION_SCHEMA hay varias tablas de sólo lectura. En realidad son vistas, no tablas, así que no se puede ver ningún fichero asociado con ellas.

15.5.2 Sentencia SHOW

MySQL también soporta una familia de sentencias SHOW para obtener metadatos. Para mostrar la definición de una vista, se usa la sentencia SHOW CREATE VIEW. Ejemplo:

```
USE world;
SHOW CREATE VIEW VistaCity \G
```

Algunas sentencias en MySQL que originalmente fueron diseñadas para obtener metadatos sobre las tablas base han sido extendidas para que también funcionen con vistas:

- DESCRIBE y SHOW COLUMNS

```
DESCRIBE VistaCity;
SHOW COLUMNS FROM VistaCity;
```

- SHOW TABLE STATUS

```
SHOW TABLE STATUS LIKE 'PobEuropa' \G
```

- SHOW TABLES

```
SHOW FULL TABLES LIKE 'PobEuropa' \G
```

Laboratorio TEMA 15

1. Crear una vista con el código, Nombre y población de países europeos

```
USE world;
DROP VIEW IF EXISTS vista_europa;

CREATE VIEW vista_europa AS
SELECT Code, Name, Population
FROM Country
WHERE Continent='Europe';

SHOW TABLES;
SELECT * FROM vista_europa;
```

2. Sobre la vista PobEuropa (con los campos Código, Nombre y Población; si está creada, cambiarla y de paso añadir alias) realizar lo siguiente:
   a) Cambiar el nombre de país SWE a MYSQL
   b) Borrar de la vista PobEuropa el registro de Bélgica
   c) Insertar en PobEuropa un nuevo registro: XXX, 'MYSQL', 450.

```
USE world;
CREATE OR REPLACE VIEW PobEuropa AS
SELECT Code AS Código, Name AS Nombre, Population AS Población
FROM CountryCopia
WHERE Continent='Europe';

UPDATE  PobEuropa
SET Name='MySQL' WHERE code='SWE';

SELECT Name
FROM CountryCopia WHERE Code='SWE';

DELETE FROM  PobEuropa
WHERE Code = 'BEL';

SELECT * FROM PobEuropa
WHERE Code = 'BEL';

INSERT INTO PobEuropa VALUES ('XXX','MySQL',450);
SELECT Name FROM europe_view WHERE Code = 'BEL';
```

3. Verificar la vista Europa y cambiarla para quitar la opción WITH CHECK OPTION. Intentar añadir el registro `('XXX','MySQL',450)`.

```
USE world;
CHECK TABLE vista_europa\G

ALTER VIEW vista_europa AS
SELECT Code, Name, Population
    FROM CountryCopia WHERE Continent='europe'
    WITH CHECK OPTION;
INSERT INTO vista_europa VALUES ('XYZ','MySQL',450);
```

Dará un error:

```
ERROR 1369 (HY000): CHECK OPTION failed 'world.vista_europa'
```

PRÁCTICAS ADICIONALES TEMA 15

1. Crear una vista llamada 'CapitalesPaises' que consiste en un JOIN de las tablas Country y City que contiene las columnas Code, Country.Name, Continent, City.Name y City.ID.
   a) ¿Esta vista es actualizable?
   b) ¿Es insertable?

2. Crear una vista llamada 'Idiomas' que contiene una fila por cada idioma (Language), el número de personas que lo habla y una lista de los países donde se habla el idioma.

3. Reemplazar la vista existente llamada 'Vista_Europa' (creada en el laboratorio anterior) y llenar con datos de la BBDD la tabla Country las columnas Code, Name y Continent donde el Continent sea Europe, asegurandose que todas las actualizaciones sean verificadas antes de ser efectuadas.

4. Cambiar el nombre (Name) del país cuyo código (Code) es 'DEU' a 'ALE' en la nueva vista 'Vista_Europa'. Explicar el resultado.

5. Se desea garantizar que después de una inserción (INSERT) y/o actualización (UPDATE), cada ciudad (City) corresponda al país correcto en la tabla Country. ¿como se puede lograr eso?
   Pista: La vista requerirá una subconsulta para ser creada de forma precisa.

6. Crear una vista llamada 'ContFrances' que contenga todos los continentes con países donde mas del 10% de su población hable francés. Confirmar que existe la nueva vista y listar todo su contenido.

16 SENTENCIAS PREPARADAS

16.1 ¿Por qué usar sentencias preparadas?.

Son útiles cuando se desea ejecutar varias consultas que difieren solamente en detalles mínimos, Por ejemplo, puede prepararse una sentencia y ejecutarla varias veces, cada vez usando valores distintos en los datos.

Además, las sentencias preparadas también ofrecen un mejor rendimiento ya que la sentencia completa es analizada sintácticamente por el servidor sólo una vez. Cuando este análisis es completado, el servidor y el cliente pueden hacer uso de un nuevo protocolo que requiera menos conversión de datos. Esto, por lo general, trae como resultado que haya menos tráfico entre el servidor y el cliente que cuando se envia cada sentencia de manera individual.

16.2 Usar sentencias preparadas desde el cliente mysql

La mayoría de las veces, las sentencias son preparadas y ejecutadas mediante el uso de una interfaz de programación que usa normalmente para escribir aplicaciones con MySQL. Sin embargo, a efectos de prueba y corrección, es posible definir y usar sentencias preparadas desde un cliente de línea de comandos mysql. Las sentencias preparadas están asociadas a una sesión; por lo tanto no son visibles para las demás sesiones.

16.2.1 Variables definidas por el usuario

Se puede almacenar un valor en una variable de usuario y luego referirse a ella cuando se ejecuten diversos tipos de sentencias, entre ellas las preparadas. Esto permite pasar valores de una sentencia a otra. Las variables de usuario son específicas de la conexión; en otras palabras, una variable de usuario definida por un cliente no puede ser vista o utilizada por otros clientes. Todas las variables para una conexión dada de un cliente son automáticamente liberadas cuando ese cliente sale del sistema.

La sintaxis de una variable de usuario y su asignación será:

```
SET @nombre_var = expresion [,@nombre_var = expresion]...
```

Para el nombre de variable se pueden usar los caracteres alfanuméricos del conjunto de caracteres usado, además de “.”,”_” y “$” (punto, guión bajo y dólar). Desde MySQL 5 los nombres no distinguen entre mayúsculas y minúsculas. Ademas, puede haber espacios siempre que estén entre comillas simples o dobles o como identificador:

```
@'mi_var', @”mi_var”, @`mi_var`.
```

La asignación puede hacerse con = o bien :=; la expresión asignada a cada variable puede dar como resultado un entero, un real, una cadena de caracteres o NULL. Sin embargo, si el valor de la variable es seleccionado a partir de un conjunto resultante, es devuelto al cliente como una cadena.

Si a una variable de usuario se le asigna una cadena de caracteres, tendrá el mismo conjunto de caracteres y ordenación que dicha cadena.

La adaptabilidad de las variables de usuario a tomar los atributos del valor que les es asignado es implícita (es decir, la misma propiedad que tienen los valores de las columnas de las tablas, que incluyen los tipos de datos).

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/sqlps.html

Veamos un ejemplo. Vamos a implementar una sentencia preparada que determina cuántos idiomas se hablan en un país dado, la ejecuta varias veces (haciendo uso de variables definidas por el usuario) y luego la deshace.

```
USE world;
```

Usamos `PREPARE` para crear la sentencia preparada:

```
PREPARE mi_sentpre
      FROM
      'SELECT COUNT(*)
      FROM CountryLanguage
      WHERE CountryCode= ?';
```

Asiganamos a la variable de usuario un valor:

```
SET @code = 'ESP';
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

Y ejecutamos la sentencia preparada:

```
EXECUTE mi_sentpre  USING @code;
```

Repetimos la operación:

```
SET @code = 'RUS';
EXECUTE mi_sentpre USING @code;
```

Eliminamos la sentencia preparada:

```
DEALLOCATE PREPARE mi_sentpre;
```

16.3 Preparación de una sentencia.

Como ya hemos visto en el ejemplo, se usa la sentencia PREPARE. Sintaxis:
PREPARE nombre_sentencia FROM sentencia_SQL;
Nombre_sentencia no distingue entre mayúsculas y minúsculas. En cuanto a la sentencia_SQL puede darse como una cadena literal o como una variable de usuario que contenga la sentencia.

La sentencia puede no estar completa porque los valores de los datos que son ignorados al momento de prepararla están representados por signos de interrogación (?) que sirven como marcadores de parámetros. Cuando la sentencia es ejecutada, se proporcionan los valores específicos de los datos, una para cada parámetro en la sentencia. El servidor reemplaza los marcadores con los valores de los datos a fin de completar la sentencia. Puede usarse valores distintos cada vez que la sentencia sea ejecutada.

```
PREPARE PaisPob FROM
      'SELECT Name, Population
      FROM Country
      WHERE Continent = ?
      AND Region = ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared
```

Y ejecutamos asignado ambas variables:

```
SET @Continent = 'Europe', @Region = 'Southern Europe';
EXECUTE PaisPob USING @Continent, @Region;
```

El mensaje Statement prepared (Sentencia preparada) indica que el servidor está listo para ejecutar la sentencia `PaisPob`.

Por otra parte, si el servidor encuentra un problema al analizar sintácticamente la sentencia durante un PREPARE, devuelve un ERROR y no lo prepara:

```
PREPARE error FROM
      'SELECT NoExisteColumna
      FROM Country
      WHERE Code = ?';
ERROR 1054 (42S22): Unknown column 'NoExisteColumna' in 'field list'
```

Si se prepara una sentencia existente, el servidor desecha la anterior y prepara la nueva sentencia. Un ejemplo con la sentencia vista antes:

```
PREPARE PaisPob FROM
      'SELECT Code, Name, Population
      FROM Country
      WHERE Continent = ?
      AND Region = ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared

SET @Continent = 'Europe', @Region = 'Southern Europe';
EXECUTE PaisPob USING @Continent, @Region;

+------+-------------------------------+------------+
| Code | Name                          | Population |
+------+-------------------------------+------------+
| ALB  | Albania                       |    3401200 |
| AND  | Andorra                       |      78000 |
| BIH  | Bosnia and Herzegovina        |    3972000 |
| ESP  | Spain                         |   39443700 |
| GIB  | Gibraltar                     |      25000 |
| GRC  | Greece                        |   10545700 |
| HRV  | Croatia                       |    4473000 |
| ITA  | Italy                         |   57680000 |
| MKD  | Macedonia                     |    2024000 |
| MLT  | Malta                         |     380200 |
| PRT  | Portugal                      |    9997600 |
| SMR  | San Marino                    |      27000 |
| SVN  | Slovenia                      |    1987800 |
| VAT  | Holy See (Vatican City State) |       1000 |
| YUG  | Yugoslavia                    |   10640000 |
+------+-------------------------------+------------+
```

Es importante reseñar que las sentencias preparadas sólo duran durante las sesión en las que fueron creadas, y sólo son visibles en estas. Cuando finaliza la sesión, son desechadas automáticamente.

Por otro parte, MySQL no permite la prepación de cualquier tipo de sentencia.
Las que pueden ser preparaas son las siguientes:

- SELECT
- INSERT, REPLACE, UPDATE
- CREATE TABLE
- SET, DO y muchas SHOW

A partir de MySQL5, además:

- ANALIZE TABLE, OPTIMIZE TABLE, REPAIR TABLE
- CACHE INDEX
- CHANGE MASTER
- CHECKSUM
- CREATE | RENAME | DROP DATABASE
- CREATE | RENAME | DROP USER
- FLUSH
- GRANT, REVOKE
- KILL
- LOAD INDEX INTO CACHE
- RESET...
- SHOW BINLOG EVENTS
- SHOW CREATE
- SHOW AUTHORS | CONTRIBUTORS | ERRORS
- SHOW LOGS
- SHOW SLAVE
- INSTALL PLUGIN, UNISTALL PLUGIN

Muchos de los anteriores son sentencias de Administración que ya veremos.

16.4 Ejecución de una sentencia

Una vez preparada, una sentencia puede ser ejecutada. Si la sentencia contiene algún marcador de parámetro, se debe proporcionar un valor para cada uno de ellos mediante variables de usuarios.

Para ejecutar una sentencia preparada, se inicializan las variables de usuario que proporcionan valores a los parámetros y luego se envía la sentencia EXECUTE... USING.
Veamos un ejemplo:

```
USE world;
DEALLOCATE PREPARE PaisPob;
PREPARE PaisPob FROM
'SELECT Name, Population FROM Country WHERE Code = ?';

SET @var1 = 'USA';
EXECUTE PaisPob USING @var1;
SET @var2 = 'GBR';
EXECUTE PaisPob USING @var2;
SELECT @var3 := 'CAN';
EXECUTE PaisPob USING @var3;
```

Si se hace referencia a una variable no inicializada, su valor será NULL:

```
EXECUTE PaisPob USING @var4;
```

NOTA: La cláusula USING no es requerida si no existen comodines en la sentencia.

16.5 Liberación de una sentencia preparada

Las sentencias preparadas son eliminadas al ser redefinidas o cuando se cierra la conexión con el servidor, de modo que es raro que sean desechadas explicitamente. Sin embargo, si se desea hacerlo (por ejemplo para liberar memoria en el servidor) se usara `DEALLOCATE PREPARE`. Su sintaxis es:

```
DEALLOCATE PREPARE
```

Ejemplo:

```
DEALLOCATE PREPARE PaisPob;
```

MySQL también dispone de una sentencia `DROP PREPARE` como un alias de `DEALLOCATE PREPARE`. Un ejemplo:

```
DROP PREPARE PaisPob;
```

PRACTICAS ADICIONALES TEMA 16

1. Crear una sentencia preparada llamada 'misentencia' que incluya una consulta del nombre en la tabla city, donde id tendrá el valor igual a una variable.
   - Asignar a la variable el valor 3567.
   - Ejecutar la sentencia preparada con la variable asignada.
   - Asignar a la variable el valor 3568.
   - Volver a ejecutar la sentencia preparada con el nuevo valor de la variable.
   - Eliminar explícitamente la sentencia preparada.

2. Crear una sentencia preparada con los nombres de países y los idiomas que se hablan en ellos, especificando como variables el Continente y la Región.
   - Probar con Oceania y Polynesia (como Región).
   - Probar con Asia y Southeast Asia.

17 EXPORTAR E IMPORTAR DATOS

17.1 Exportación e Importación de datos usando mysql

En este tema veremos como:

- Exportar datos mediante SELECT... INTO OUTFILE
- Importar datos mediante LOAD DATA INFILE
- Exportar datos mediante el programa cliente mysqldump
- Importar datos mediante mysqlimport
- Importar datos usando el comando SOURCE

17.1.1 Exportar datos usando SELECT … INTO OUTFILE

Una sentencia SELECT puede usarse con la cláusula INTO OUTFILE para escribir directamente en un archivo externo. El nombre del archivo puede incluir la ruta o no; en este último caso se grabará en el directorio donde están los datos del cliente (cosa que no recomiendo).
Si no ponemos nada en la ruta, el directorio de salida será **/var/lib/mysql/<nombre_bbdd>**
La sintaxis será:

```
SELECT …
INTO OUTIFLE nombre_archivo
FROM tabla
[OPCIONES]
```

Veamos un ejemplo...
Arrancamos el cliente, nos metemos en la BBDD world y exporamos al directorio /temp
NOTA IMPORTANTE: En mi equipo he realizado pruebas en diversos directorios y da problemas. ¿Porqué? Por los permisos. Nos creamos la carpeta mysql en nuestro directorio de usuario con todos los permisos posibles.

```
mysql -u root -p
USE world;
SELECT *
INTO OUTFILE '/home/ivan-htpc/mysql/City.txt'
FROM City
LIMIT 10;
```

El uso de la cláusula INTO OUTFILE modifica el funcionamiento normal de SELECT:

- El archivo se escribe en el servidor, no se manda al cliente.
  NOTA: Por tanto no podemos elegir un nombre ya existente.
- El servidor escribe un nuevo archivo en el host donde está corriendo (por tanto el usuario debe contar con privilegio FILE, en nuestro caso root).
- El archivo se crea con permisos de lectura; eso si, el propietario es el servidor mysql.
- El archivo presenta un registro por fila, con los campos tabulados.

Por otro lado en Windows podemos tener problemas con el /
Para evitarlo tenemos dos alternativas:

1. Usar el caracter “/” al estilo Unix:
   ```
   SELECT *
   INTO OUTFILE 'C:/tmp/clientes_backup.txt'
   FROM clientes;
   ```
2. Usar el doble caracter “\”:
   ```
   SELECT *
   INTO OUTFILE 'C:\\tmp\\clientes_backup.txt'
   FROM clientes;
   ```

Usar modificadores de formato para los archivos de texto

De forma predeterminada `SELECT... INTO OUTFILE` asume que el archivo de datos tiene un formato en el cual los campos están separados por el carácter de tabulación y los registros acaban con el carácter salto de línea.

Esto lo podemos cambiar con una serie de modificadores cuya sintaxis es:

```
SELECT … INTO OUTFILE nombre_archivo
FIELDS
      TERMINATED BY 'cadena'
      ENCLOSED BY 'carácter'
      ESCAPED BY 'carácter'
LINES
      TERMINATED BY 'cadena'
FROM tabla
```

`FIELDS` → Indica la estructura de las columnas
`LINES` → indica la estructura de los registros
`TERMINATED BY` → Indica el carácter de separación entre valores. Por defecto: Tabulador
`ENCLOSED BY` → Caracter delimitador entre valorer. Por defecto: Sin comillas
Este tiene una alternativa: `OPTIONALLY ENCLOSED BY`:
→ Para `LOAD DATA INFILE` funciona como hemos visto.
→ `SELECT … INTO OUTFILE` solo escribe comillas en caso de cadenas de caracteres.

`ESCAPED BY` → El carácter de escape para cada valor. Por defecto: \
MySQL entiende las siguientes secuencias de escape:

- `\N` → NULL
- `\0` → Byte NULL (cero)
- `\b` → Retroceso
- `\n` → Nueva Línea (LF)
- `\r` → Retorno de carro (CR)
- `\s` → Espacio
- `\t` → Carácter de tabulación
- `\'` → Comilla simple
- `\”` → Doble comilla
- `\\` → Barra Invertida

El terminador de Línea (registro) prederminado es el salto de línea (LF). Se suelen usar otros como el par retorno de carro\salto de linea.
Ejemplo: `LINES TERMINATED BY '\r\n'`

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/load-data.html
http://mysql.conclase.net/curso/?sqlsen=LOAD_DATA

Un ejemplo completo, la exportación a CSV (muy usada):
NOTA: En el caso de estar en Linux, debemos asignar permisos a la carpeta elegida.

```
SELECT * INTO OUTFILE '/home/ivan-htpc/mysql/City1.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY ""
LINES TERMINATED BY '\r'
FROM City
LIMIT 10;
```

17.1.2 Importar datos mediante la sentencia LOAD DATA INFILE

La sentencia LOAD DATA INFILE puede ser usada como alternativa a INSERT a la hora de añadir registros en una tabla. Mientras esta necesita los valores a introducir, LOAD DATA INFILE permite su lectura desde un archivo separado de datos.

Veamoslo con un ejemplo y de paso aprendemos algo nuevo...
Como se copia la estructura de una tabla:
Se usa la sintaxis

```
CREATE TABLE tabla_nueva
LIKE tabla_anterior
```

Un ejemplo:

```
CREATE DATABASE world_prueba;
CREATE TABLE world_prueba.City
LIKE world.City;

USE world_prueba;
SELECT * FROM City;
```

Sale vacio. Ahora es cuando toca realizar la importación...
NOTA: Previamente debemos haber hecho la exportación como vimos en el apartado anterior.

```
SELECT *
INTO OUTFILE '/home/ivan-htpc/mysql/City.txt'
FROM City
LIMIT 10;
```

Y ahora si, la importación:

```
USE world_prueba;
LOAD DATA INFILE '/home/ivan-htpc/mysql/City.txt'
INTO TABLE City;
```

Ahora hacemos el SELECT:

```
SELECT * FROM City;
```

```
mysql> SELECT * FROM City;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
|  7 | Haag           | NLD         | Zuid-Holland  |     440900 |
|  8 | Utrecht        | NLD         | Utrecht       |     234323 |
|  9 | Eindhoven      | NLD         | Noord-Brabant |     201843 |
| 10 | Tilburg        | NLD         | Noord-Brabant |     193238 |
+----+----------------+-------------+---------------+------------+
10 rows in set (0.00 sec)
```

La sentencia `LOAD DATA INFILE` tiene cláusulas y modificadores de formato similares a la sentencia `SELECT … INTO OUTFILE`.

Importar archivos delimitados por carácter de tabulación o comas:

Se puede importar un archivo de datos que contenga una tabla, delimitados por el carácter de tabulación o separados por comas mediante LOAD DATA INFILE. Para ello hay que conocer las siguientes características del archivo:

- Separadores de los valores de las columnas
- Separador de línea
- Caracteres dentro de los cuales están incluidos los valores (ejemplo: comillas)
- Si el nombre de la columna está incluido en el archivo
- Si hay encabezado que indique las filas de la tabla que se saltarán (ignorarán) antes de la importación
- El sistema de archivos donde está el archivo de datos.
- Se se requieren privilegios para acceder a la tabla.
- El orden de las columnas.
- Si coincide el número de columnas en el archivo y en la tabla.

Veamos un ejemplo...
Empezamos por la exportación en CSV que ya vimos:

```
USE world;
SELECT * INTO OUTFILE '/home/ivan-htpc/mysql/City1.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY ""
LINES TERMINATED BY '\r'
FROM City
LIMIT 10;
```

Ahora nos creamos una copia de City nueva:

```
CREATE TABLE world_prueba.City1
LIKE world.City;
```

Y ahora viene la importación:

```
LOAD DATA INFILE '/home/ivan-htpc/mysql/City1.csv'
INTO TABLE City1
FIELDS TERMINATED BY ','
ENCLOSED BY ""
LINES TERMINATED BY '\r';
```

```
mysql> LOAD DATA INFILE '/home/ivan-htpc/mysql/City1.csv'
    -> INTO TABLE City1
    -> FIELDS TERMINATED BY ','
    -> ENCLOSED BY ""
    -> LINES TERMINATED BY '\r';
Query OK, 10 rows affected (0.03 sec)
Records: 10  Deleted: 0  Skipped: 0  Warnings: 0

mysql> SELECT * FROM City1;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
```

Especificar la ubicación del archivo de datos:

La sentencia `LOAD DATA INFILE` puede leer los archivos de datos localizados en el host del servidor o del cliente:

- Por defecto mysql asume que el archivo está el servidor y lo lee directamente.
- Si la sentencia comienza con `LOAD DATA LOCAL INFILE` (añadiendo `LOCAL`) será leido en el equipo del cliente, envíando los datos al servidor.

Ignorar líneas en archivos de datos:

A veces nos encontraremos que el archivo de datos tiene en la primera línea (o en las primeras) los nombres de los campos u otro contenido. Para ignorar dichas líneas se usa la sentencia IGNORE … LINES. Veamos un ejemplo con `City1.csv`:
Primero nos creamos en world_prueba otra copia de City:

```
USE world_prueba;
CREATE TABLE world_prueba.City2
LIKE world.City;
```

Y ahora el LOAD DATA INFILE:

```
LOAD DATA INFILE '/home/ivan-htpc/mysql/City1.csv'
INTO TABLE City2
FIELDS TERMINATED BY ','
ENCLOSED BY ""
LINES TERMINATED BY '\r'
IGNORE 5 LINES;
```

```
mysql> SELECT * FROM City2;
+----+-----------+-------------+---------------+------------+
| ID | Name      | CountryCode | District      | Population |
+----+-----------+-------------+---------------+------------+
|  6 | Rotterdam | NLD         | Zuid-Holland  |     593321 |
|  7 | Haag      | NLD         | Zuid-Holland  |     440900 |
|  8 | Utrecht   | NLD         | Utrecht       |     234323 |
|  9 | Eindhoven | NLD         | Noord-Brabant |     201843 |
| 10 | Tilburg   | NLD         | Noord-Brabant |     193238 |
+----+-----------+-------------+---------------+------------+
5 rows in set (0.00 sec)
```

17.2 Exportación e Importación mediante programas cliente de MySQL

17.2.1 Exportar datos mediante el programa cliente mysqldump

La utilidad mysqldump vacia el contenido de las tablas en archivos y puede exportar tablas de diversas maneras:

- Volcando la estructura completa, incluyendo sentencias DROP y TABLE para poder recrear una configuración completa.
- Volcar sólo los datos.
- Volcar sólo la estructura de la tabla.
- Volcar en un formato estándar.
- Volcar con especificaciones particulares de MySQL a efectos de optimizar la velocidad.
- Para volcados de texto plano, puede disponerse de una mayor compresión de los datos.

La sintaxis (desde el shell, no desde el cliente mysql) será:

```
mysqldump [opciones] nombre_bbdd [tablas]
mysqldump [opciones] --databases nombre_bbdd1 [nombre_bbdd2
nombre_bbdd3...]
mysqldump [opciones] --all-databases
```

Si no se menciona tabla alguna tras nombre_bbdd se vuelcan las bbdd completas.
Lo mas conveniente (y a veces obligatorio) es incluir usuario y contraseña dentro de las opciones de mysqldump. El usuario se pone -u <usuario> y la clave -p (luego lo preguntará).
Lo suyo es además de volcar la bbdd, pasarlo a un archivo externo. Para ello se usa el carácter >
NOTA: Si no se pone el carácter y el archivo de salida, sale por pantalla. Ejemplo:

```
mysqldump -u root -proot world
```

Por ejemplo: para exportar al completo la bbdd world al archivo local world.sql hariamos:

```
mysqldump -u root -p world > world.sql
```

Pedirá la contraseña y listo.

NOTA: En Linux va a /home/usuario/ y Windows en C:\Documents and Settings\nombreUsuario

Para volcar sólo las tablas City y Country de la bbdd world, se dan dichas tablas como argumentos después del nombre de la bbdd. Siguiendo con el ejemplo anterior (la salida se cambia):

```
mysqldump -u root -p world City Country > world_CityCountry.sql
```

NOTA: He cambiado el archivo se salida porque si no se reescribe.

Para vaciar las bases de datos world y test:

```
mysqldump -u root -p --databases world test > world_test.sql
```

NOTA IMPORTANTE: Es FUNDAMENTAL observar el sql de salida, porque dentro va no solo las tablas, si no además las vistas, los LOCK TABLES, etc.
ATENCIÓN: Para importar un sql de mysqldump se usar SOURCE, no mysqlimport.

17.2.2 Importar datos mediante mysqlimport

El cliente mysqlimport en realidad es un LOAD DATA INFILE con interfaz de línea de comandos.
Su sintaxis será:

```
shell > mysqlimport opciones nombre_bbdd archivo_entrada
```

IMPORTANTE: La importación se hace a través de los datos obtenidos con `SELECT … INTO OUTFILE`. Es decir, solo vale para datos (no para crear tablas), por lo que un archivo dump (como el visto en la sección anterior) no es valido. Para usarlo se usa `SOURCE` (que ya veremos).

Veamos un ejemplo desde la exportación:
NOTA: Cuidado con los permisos de las carpetas. Yo he creado una, mysql con los permisos del sistema operativo necesarios para poder escribir en ella.

```
SELECT * INTO OUTFILE '/home/ivan-htpc/mysql/Country.txt'
FROM Country LIMIT 10;
```

IMPORTANTE: En windows, si no se pone la ruta va a C:\xampp\mysql\data\<nombre_bbdd>

Ahora nos creamos una BBDD nueva donde irá la importación así como una tabla Country:

```
CREATE DATABASE BD_Country;
USE BD_Country;
CREATE TABLE Country LIKE world.Country;
```

Y procedemos a la importación desde el shell:

```
mysqlimport --local -u root -p BD_Country /home/ivan-
htpc/mysql/Country.txt;
```

¿Y si queremos hacerlo con las opciones de INTO OUTFILE?
Volvemos al cliente mysql y nos hacemos otra tabla igual a Country;

```
USE Country;
CREATE TABLE Country1 LIKE world.Country;
```

Ahora procedemos a la exportación:

```
USE world;
SELECT * INTO OUTFILE '/home/ivan-htpc/mysql/Country1.txt'
FIELDS TERMINATED BY ','
ENCLOSED BY ""
LINES TERMINATED BY '\r'
FROM Country LIMIT 5;
```

Nos salimos e importamos. Obsérvese que cada opción se separa por un espacio.

```
exit
shell> mysqlimport -u root -p --fields-terminated-by=',' --fields-
enclosed-by="" --lines-terminated-by='\r' Country /home/ivan-
htpc/mysql/Country1.txt;
```

Y listo. Podemos comprobar ambos:

```
SELECT * FROM Country;
SELECT * FROM Country1;
```

NOTA: A título informativo, en linux, los archivos de datos de mysql están en /opt/lampp/var/mysql/ Aparecen como nombre_tabla.frm siendo los metadatos como db.opt

Mas información: http://dev.mysql.com/doc/refman/5.0/es/mysqlimport.html

17.3 Importar datos usando el comando SOURCE

Como ya se vio en los primeros temas, existe una manera de importar mediante la sentencia SOURCE. Eso si, el archivo a importar debe contener todas las sentencias necesarias: creación de la base de datos, tablas, claves, indices, datos, etc. Su sintaxis es:

```
SOURCE <ruta_archivo>
```

NOTA: Ejemplo de `SOURCE` lo tenemos con la importación de la BBDD world.

Como alternativa tenemos este formato:

```
mysql -u (usuario) -p (nombre_bbdd) < (archivo.sql)
```

Veamos un ejemplo. Recordemos como hicimos una exportación con mysqldump:

```
mysqldump -u root -p world > world.sql;
```

Y luego la importación:

```
mysql -u root -p world1 < world.sql;
```

18 RUTINAS ALMACENADAS

18.1 Que es una rutina almacenada.

Una rutina almacenada es un conjunto de sentencias SQL que pueden ser guardadas en el servidor. Una vez almacenadas, los clientes no necesitan reenviar sentencias sino referirse en su lugar a una rutina específica.

Hay dos tipos de rutinas almacenadas:

- Procedimientos almacenados: Una serie de instrucciones guardadas dentro de la propia Base de datos que actua sobre las instrucciones pero no devuelven ningún valor.

Un procedimiento se invoca usando una sentencia CALL y solo puede devolver valores usando variables de salida.

- Funciones Almacenada: Una serie de instrucciones guardadas dentro de la base de datos que devuelve un único valor.

Una función puede ser llamada desde una sentencia como si se tratara de cualquier función y puede devolver un valor escalar.

18.2 Creación de rutinas almacenadas

Existen los Procedimientos y las Funciones.

18.2.1 Creación de procedimientos

Su sinstaxis será:

```
CREATE PROCEDURE <nombre_procedimiento> <sentencias_procedimiento>
```

Veamos un ejemplo:

```
CREATE PROCEDURE Conteo_paises ()
SELECT COUNT(*) AS Paises
FROM Country;
```

Llamamos al procedimiento:

```
CALL Conteo_paises ();
```

18.2.2 Creación de funciones

La sintaxis será:

```
CREATE FUNCTION nombre_función
RETURNS tipo_devuelvo
sentencias_funcion
```

Una función requiere tener una sentencia RETURN que termina la ejecución de la función y pasa un valor escalar al programa que llamó a la función, reemplazando efectivamente la llamada a la función con el valor devuelto.

Veamos un ejemplo:

```
CREATE FUNCTION Gracias (cadena CHAR(20))
RETURNS CHAR(50)
RETURN CONCAT('Muchas Gracias ', cadena,'!');
```

Para probarlo:

```
SELECT Gracias ('a toda la peña');
```

18.3 Sentencias compuestas

MySQL brinda una forma de escribir múltiples sentencias dentro de las rutinas almacenadas usando la sintaxis BEGIN...END.

Dentro de la sintaxis BEGIN...END, las sentencias deben terminar con un punto y coma (;). Como éste es el mismo carácter de terminación por defecto para las sentencias SQL, es importante cambiarla con DELIMITER cuando se use el cliente de línea de comandos o dentro de un procesamiento enlote.

Por ejemplo:
`DELIMITER //` -> Cambiar la terminación por //
`DELIMITER ;` -> Lo deja como estaba

La sintaxis BEGIN...END puede usarse en Rutinas Almacenadas y Disparadores. Las sentencias compuestas que están dentro de esta sintaxis pueden contener una o mas sentencias (no hay limitación) o ninguna sentencia (es válido tener una sentencia compuesta vacia).

Veamos un ejemplo de un procedimiento almacenado con sentencias compuestas:

```
USE world;
DELIMITER //
CREATE PROCEDURE Conteo_paises1 ()
BEGIN
      SELECT 'Cuenta Paises', COUNT(*) FROM country;
      SELECT 'Cuenta Ciudades', COUNT(*) FROM city;
      SELECT 'Cuenta Lenguas', COUNT(*) FROM CountryLanguage;
END//
DELIMITER ;
```

Y llamamos al procedimiento:

```
CALL Conteo_paises1;
```

Se recomienda sangrar el procedimiento (como en otras sentencias) para facilitar su lectura.
Puede haber bloques `BEGIN...END` anidados.
Los bloques `BEING...END` pueden tener etiquetas `WHILE/REPEAT/LOOP`

Laboratorio 18.3

Crear un procedimiento almacenado que interactue con las tablas de la BBDD world, suministrando información sobre Ciudades, Datos del País y Lenguas a partir del Código.

Primero creamos el Procedimiento:

```
DELIMITER //
CREATE PROCEDURE info_pais (IN codigo CHAR (3))
BEGIN
      SELECT * FROM City WHERE CountryCode = codigo;
      SELECT * FROM Country WHERE Code = codigo;
      SELECT * FROM CountryLanguage WHERE CountryCode = codigo;
END//
DELIMITER ;
```

Ahora lo probamos con Holanda (NLD):

```
CALL info_pais ('NLD');
```

18.4 Declaración y asignación de variables

Para realizar la declaración y asignación de variables usaremos las sentencias:

- DECLARE: Declara variable locales y puede ser usada en rutinas almacenadas para inicializar variables de usuario. Con la cláusula DEFAULT se puede asignar un valor inicial. Si se especifica, dicho valor inicial será NULL. Ejemplo:

```
USE world;
DELIMITER //
CREATE FUNCTION intro_tasa (total_carga FLOAT(9,2))
RETURNS FLOAT(10,2)
BEGIN
      DECLARE ratio_tasa FLOAT (3,2) DEFAULT 0.07;
      RETURN total_carga + total_carga *  ratio_tasa;
END//
DELIMITER ;
```

Probamos la nueva función:

```
SELECT intro_tasa (1000);
```

Existen dos diferencias básicas entre las variables locales y las definidas por el usuario que deben ser resaltadas:

- **En cuanto a su declaración**: las variables locales tienen que estar explícitamente declaradas mientras que las variables de usuario no lo requieren.
- **En cuanto a su alcance**: Las variables locales son locales a la instancia de las rutinas, mientras que las variables de usuario brindan una mayor flexibilidad en cuanto a su alcance al ser globales y visibles en toda la sesión/conexión.

SELECT...INTO

La sintaxis de esta sentencia guarda los resultados de una consulta directamente en las variables de usuario, las cuales pueden ser locales o globales y visibles (y usables) en toda la sesión.

Veamos un ejemplo de **Variables de sesión**:

```
USE world;
SELECT SUM(Population)
FROM Country
INTO @PobMundial;
```

NOTA: La línea del INTO puede ir antes del FROM. Es equivalente.

Para ver el resultado:

```
SELECT @PobMundial;
```

Otro ejemplo, pero con variables locales (necesitamos crear un procedimiento)

```
DELIMITER //
CREATE PROCEDURE TotalCiudades ()
BEGIN
      DECLARE ConteoCiudades INT;
      SELECT COUNT(*)
      INTO  ConteoCiudades
      FROM City;
      SELECT ConteoCiudades;
END//
DELIMITER ;
```

NOTA: La línea del FROM puede ir antes del INTO. Es equivalente.

Para ver el resultado:

```
CALL TotalCiudades;
```

SET
Esta sentencia permite al usuario asignarle valores a las variables, usando = o := como operadores de asignación.
Veamos un ejemplo:

```
DELIMITER //
CREATE FUNCTION factura_final
(pago_total FLOAT (9,2), porc_tasa FLOAT (3,2))
RETURNS FLOAT (10,2)
BEGIN
      DECLARE Factura FLOAT (10,2);
      SET Factura = pago_total + (pago_total * porc_tasa/100);
      RETURN Factura;
END //
DELIMITER ;
```

Y ahora llamamos a la función (pagamos 50€ por las pizzas; el IVA está en el 10%):

```
SELECT factura_final  (50,10);
```

Alcance de las variables
Un mismo identificador puede ser usado al mismo tiempo como parámetro de rutina, como variable local y como una columna de una tabla. También, el mismo nombre de una variable local puede usarse en bloques anidados.

Por ejemplo:

```
DELIMITER //
CREATE PROCEDURE precedencia (param1 INTEGER)
BEGIN
      DECLARE var1 INT DEFAULT 0;
      SELECT 'outer1', param1, var1;
      BEGIN
      DECLARE param1, var1 CHAR(3) DEFAULT 'abc';
      SELECT 'inner1', param1, var1;
      END;
      SELECT 'outer1', param1, var1;
END//
DELIMITER ;
```

En estos casos, el identificador es ambiguo por lo que son aplicadas las siguientes reglas de precedencia:

- Un parámetro de una rutina tiene precedencia sobre una columna de tabla.
- Una variable local tiene precedencia sobre un parámetro de rutina o una columna de tabla.
- Una variable local en un bloque mas interno tiene precedencia sobre una variable local en un bloque mas externo.

El hecho de que las columnas de una tabla no tengan precedencia sobre las variables no es un comportamiento estándar.
Evidentemente, la mejor manera de evitar este trastorno es poner nombres adecuados y diferenciados.

18.5 Declaración de parámetros

Las rutinas almacenadas pueden incluir declaraciones de parámetros que permiten pasar valores a las rutinas invocadas. En cuanto a los procedimientos almacenados, también se devuelven valores al proceso que lo llamó después de terminar el procedimiento.

Parámetros de un procedimiento almacenado
Existen tres posibles formas de declarar parámetros que controlan la información que será pasada u obtenida del procedimiento:

- **IN**: Indica un parámetro de entrada que es pasado desde el proceso que llama al procedimiento. Su sintaxis será:

```
CREATE PROCEDURE nombre_procedimiento
(IN nombre_parámetro tipo_parámetro)
sentencias_procedimiento
```

El parámetro nombre_parámetro es accesible dentro de la sentencia del procedimiento para obtener el valor pasado por el parámetro y el parámetro tipo_parámetro indica al procedimiento el formato en el cual debería verlo el procedimiento. Cada parámetro puede ser declarado de cualquier tipo de datos, excepto para el atributo COLLATE. Los tipos mas comunes sin INT, CHAR y VARCHAR.
NOTA: IN es la declaración de parámetros por defecto si no hay ninguna declaración de parémetros que anteceda al parámetro.

Un ejemplo:

```
CREATE PROCEDURE raiz_cuadrada
(IN parametro INT)
SELECT SQRT (parametro);
```

Asignamos un valor y llamamos al procedimiento:

```
SET @parametro = 81;
CALL raiz_cuadrada (@parametro);
```

- **OUT**: Indica un parámetro de salida cuyo valor es asignado por el procedimiento y pasado al proceso que lo llamó después de que el procedimiento finaliza. Su sintaxis es:

```
CREATE PROCEDURE nombre_procedimiento
(OUT nombre_parámetro tipo_parámetro)
sentencias_procedimiento
```

La sintaxis es idéntica a IN, excepto por el hecho de que cualquier valor pasado desde el proceso que llama al procedimiento a través de un parámetro OUT es ignorado por el procedimiento.

Veamos un ejemplo; Crear un procedimiento que devuelva las lenguas de España:

```
CREATE PROCEDURE Lenguas_España
(OUT lenguas INT)
SELECT COUNT(*)
FROM CountryLanguage
WHERE CountryCode = 'ESP'
INTO lenguas;
```

Y ahora llamamos al procedimiento. En primer lugar creamos una variable y la inicializamos:

```
SET @lenguas = 0;
```

Y ahora llamamos al procedimiento y vemos el resultado:

```
CALL Lenguas_España (@lenguas);
SELECT @lenguas;
```

- **INOUT**: Indica un parémtro que puede actuar como parámetro IN y OUT. Cualquier valor pasado desde el proceso que llama al procedimiento es asignado como valor inicial del parámetro, pudiendo ser modificado por el procedimiento y pasado al proceso invocador, luego que el procedimiento finaliza. Su sintaxis es:

```
CREATE PROCEDURE nombre_procedimiento
(INOUT nombre_parámetro tipo_parámetro)
sentencias_procedimiento
```

La sintaxis es idéntica al parámetro IN; sin embargo, el valor pasado en el parámetro INOUT puede ser utilizado por la sentencia del procedimiento y devuelto al final de la misma., lo cual lo hace mas versátil.

Veamos un ejemplo; Crear un procedimiento que devuelva el número de países del mundo:

```
CREATE PROCEDURE Paises_Mundo
(INOUT paises INT)
SELECT COUNT(*)
FROM Country
INTO paises;
```

En este caso no tenemos que inicializar la variable (se hace en el procedimiento).
Llamamos al procedimiento y vemos el resultado:

```
CALL Paises_Mundo (@paises);
SELECT @paises;
```

Parámetro de una función almacenada
Respecto a las funciones almacenadas, existe una sola posible declaración de parámetros: IN, de modo que así se define por defecto y no se permite escribir ningún valor de un parámetro.

Laboratorio 18.5

Usando la base de datos world, crear un procedimiento almacenado que devolverá la población del país en términos de porcentaje respecto a la población mundial, siendo el parámetro enviado el código del país.

```
DELIMITER //
CREATE FUNCTION Porc_PobMundial (codigo CHAR(3))
RETURNS DECIMAL (4,2)
BEGIN
      DECLARE PobMundial, PobPais BIGINT;
      SELECT SUM(population)
      FROM Country
      INTO PobMundial;

      SELECT Population
      FROM Country
      WHERE Code = codigo
      INTO PobPais;
      RETURN PobPais / PobMundial * 100;
END//
DELIMITER ;
```

Y probamos con China (de paso le ponemos un nombre a la columna devuelta):

```
SELECT  Porc_PobMundial ('CHN') AS Porcentaje;
```

18.6 Ejecución de rutinas almacenadas

Un procedimiento se invoca usando una sentencia CALL y sólo puede devolver valores usando variables de salida. Se puede llamar una función desde una sentencia como si se tratara de cualquier función (invocando el nombre de la función) y puede devolver un valor escalar.

En MySQL, un procedimiento o función almacenada está asociada a una base de datos particular. Esto tiene varias implicaciones:

- USE bb_dd – Cuando se invoca la rutina, se ejecuta implicitamente una sentencia USE nombre_bbdd (que se deshace cuando la rutina finaliza). De todos modos, las sentencias USE no están permitidas en las rutinas almacenadas.
- Nombres cualificados – Los nombres de las rutinas pueden estar cualificados para indicar el nombre de la base de datos asociada. Esto puede ser útil para referirse a una rutina que no está en la base de datos actual (en uso). Por ejemplo, para invocar a un procedimiento p o la función f, almacenados en la base de datos test, se puede usar CALL test.p() o test.f() respectivamente.
- Eliminación de bases de datos – Cuando se elimina una base de datos, todas las rutinas almacenadas asociadas con la misma se eliminan también.

MySQL soporta una extensión muy útil que permite el uso de las sentencias SELECT normales dentro de un procedimiento almacenado. El conjunto resultante de este tipo de consultas se envía directamente al cliente.

Múltiples conjuntos resultantes
Múltiples sentencias SELECT generan múltiples conjuntos de resultados, de modo que el cliente debe usar una librería de clientes MySQL que los soporte. Esto lo soporta el cliente mysql.

18.7 Revisión de rutinas almacenadas

Con la capacidad de crear varias rutinas almacenadas dentro de cada base de datos, es indispensable tener la capacidad de revisar la sintaxis de las mismas una vez creadas. En MySQL existen tres grupos de sentencias diferentes para llevar a cabo esta tarea:

- SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION
  Permiten conocer la cadena usada para crear una rutina. Eso si, debemos conocer el tipo de rutina (procedimiento o función) y su nombre. Ejemplos:
  ```
  USE world;
  SHOW CREATE FUNCTION Porc_PobMundial \G
  SHOW CREATE PROCEDURE Paises_Mundo \G
  ```

- SHOW PROCEDURE STATUS y SHOW FUNCTION STATUS
  Estas sentencias son también extensiones de MySQL. Devuelven características de las rutinas, como la base de datos, nombre, tipo, creador y fechas de creación y actualización. Tienen la ventaja de que muestran rutinas específicas basadas en un patrón LIKE. Si no se especifica un patrón, se lista la información sobre todos los procedimientos o todas las funciones, según la sentencia que se utilice.
  La desventaja es que esta sentencias no muestran la sintaxis real de las rutinas que están en la base de datos, solo la información relativa a su estado. Ejemplos:
  ```
  SHOW FUNCTION STATUS \G
  SHOW PROCEDURE STATUS WHERE Db = 'test' \G
  SHOW PROCEDURE STATUS WHERE Db LIKE 'world' \G
  ```

– La tabla INFORMATION_SCHEMA.ROUTINES

La tabla `ROUTINES` proporciona información sobre las rutinas almacenadas (procedimientos y funciones) y devuelve la mayoría de los detalles que podrían encontrarse tanto con las sentencias `SHOW CREATE...` como `SHOW … STATUS` , incluyendo la sintaxis real usada para crear las rutinas. De los tres grupos de opciones, esta es la que tiene la imagen mas completa de las rutinas almacenadas en la base de datos.

La sintaxis es:

```
SELECT <campos>
FROM INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_TYPE="PROCEDURE | FUNCTION”
AND ROUTINE_SCHEMA="<nombre_bbdd>";
```

Un ejemplo:

```
SELECT routine_name, data_type, routine_definition, created, definer
FROM INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_TYPE="FUNCTION"
AND ROUTINE_SCHEMA="world";
```

NOTA: Con un SELECT * vemos todas las columnas disponibles.

18.8 Eliminación de rutinas almacenadas

Para eliminar un procedimiento almacenado debemos usar la siguiente sintaxis:

```
DROP PROCEDURE nombre_procedimiento.
```

Para eliminar una función usaremos:

```
DROP FUNCTION nombre_función
```

Si no se tiene la certeza de que exista una rutina almacenada, se coloca la cláusula IF EXISTS antes del nombre del procedimiento o de la función. Un ejemplo:

```
DROP PROCEDURE IF EXISTS Paises_Planeta;
```

En realidad el procedimiento se llama Paises_Mundo, por lo que da un warning (y no un error):

```
Query OK, 0 rows affected, 1 warning (0.00 sec)

SHOW WARNINGS;
+-------+------+-----------------------------------------------+
| Level | Code | Message                                       |
+-------+------+-----------------------------------------------+
| Note  | 1305 | PROCEDURE world.Paises_Planeta does not exist |
+-------+------+-----------------------------------------------+
```

18.9 Sentencias de control de flujo

El control de flujo está definido como las sentencias y otras construcciones que controlan el orden en que se ejecutan las operaciones. La mayoría de los lenguajes de programación de alto nivel tienen sentencias de control de flujo que permiten variaciones en este orden secuencial. Los controles de flujo más comunes son:

- Opciones – Sentencias que son ejecutadas bajo ciertas condiciones; en MySQL están representadas por las sentencias IF y CASE.
- Ciclos – Sentencias que son ejecutadas repetidamente; en MySQL, están representadas por las sentencias REPEAT, WHILE y LOOP.

Las sentencias SQL de MySQL pueden usar muchas de las sintaxis de control de flujo que se encuentran fuera de las rutinas almacenadas; sin embargo, existen notables diferencias entre las sentencias de control de flujo de las rutinas almacenadas y las sentencias de control de flujo de la sintaxis SQL. Los siguientes puntos y ejemplos pretenden mostrar el uso de las sentencias de control de flujo dentro de las rutinas almacenadas o bien en sentencias SELECT simples.

Sentencia IF
Equivalente a un condicional SI, su sintaxis es la siguiente:

```
IF (condición_a_evaluar) THEN
       lista_sentencias
ELSEIF (condición_a_evaluar) THEN
      lista_sentencias
ELSE
       condición_a_evaluar
END IF
```

Donde IF gestiona la condición si es verdadera.
Si es falsa, pasa al ELSEIF, que a su vez incluye una condición que debe ser verdadera.
Finalmente, si todas las condiciones son falsas, pasa al ELSE.
Finaliza con END IF.

Veamos un ejemplo: Creamos una función que muestre el tipo de IVA introducido:

```
DELIMITER //
CREATE FUNCTION IVA (valor INT)
RETURNS CHAR (20)
BEGIN
      DECLARE tipo CHAR (20);
      IF (valor = 4) THEN
      SET tipo = 'IVA SuperReducido';
      ELSEIF (valor = 7) THEN
      SET tipo = 'IVA Reducido';
      ELSE
      SET tipo = 'IVA Normal';
      END IF;
      RETURN tipo;
END //
DELIMITER ;
```

Y ahora llamamos a la función con los distintos IVAs:

```
SELECT IVA (4);
SELECT IVA (7);
SELECT IVA (21);
```

Sentencia CASE
La sentencia CASE funciona sobre el principio de comparar un valor dado con constantes especificadas y actuar según se encuentre la primera constante. Si las constantes conforman un rango compacto, entonces puede implementarse de manera muy eficiente como su fuera una selección entre números enteros. No hace falta incluir **BREAK / CONTINUE** en el algoritmo.
Su sintaxis es:

```
CASE condición_a_evaluar WHEN valor THEN
…
ELSE
…
END CASE
```

De forma análoga, también se emplea:

```
CASE WHEN condición_a_evaluar THEN
…
ELSE
…
END CASE
```

Un ejemplo (usado en webs propias con PHP):

```
DELIMITER //
CREATE FUNCTION estado_civil (valor INT)
RETURNS CHAR (20)
BEGIN
      DECLARE estado CHAR (20);
      CASE valor
      WHEN 0 THEN SET estado='soltero';
      WHEN 1 THEN SET estado= 'casado';
      ELSE
      SET estado= 'otro/desconocido';
      END CASE;
      RETURN estado;
END //
DELIMITER ;
```

Ahora llamamos a la función:

```
SELECT estado_civil (2) AS 'Estado Civil';
```

NOTA: La función anterior se puede escribir también de este modo:

```
DELIMITER //
CREATE FUNCTION estado_civil_mod (valor INT)
RETURNS CHAR (20)
BEGIN
      DECLARE estado CHAR (20);
      CASE
      WHEN valor= 0 THEN SET estado='soltero';
      WHEN valor= 1 THEN SET estado= 'casado';
      ELSE
      SET estado= 'otro/desconocido';
      END CASE;
      RETURN estado;
END //
DELIMITER ;
```

Y probamos de nuevo:

```
SELECT estado_civil_mod (1) AS 'Estado Civil';
```

Sentencia REPEAT
Esta sentencia repite una sentencia SQL hasta que la condición de búsqueda se hace verdadera (TRUE). La sentencia REPEAT se ejecuta al menos una vez.

Las etiquetas pueden formar parte de las sentencias REPEAT. Una etiqueta es un identificador asociado a una sentencia mediante el carácter dos puntos (;). Una sentencia REPEAT puede utilizar una etiqueta tanto al principio (mi_etiqueta_repeat) como al final. Una etiqueta de fin es opcional pero cuando es usada, solo puede aparecer si también aparece una etiqueta de principio.

Si ambas etiquetas están presentes, deben usar exactamente el mismo identificador (en este caso mi_etiqueta_repeat). Resumiendo: REPEAT es similar a un “DO...WHILE” de toda la vida.

Su sintaxis es:
```
[mi_etiqueta_repeat:] REPEAT
…
UNTIL condición_a_evaluar
END REPEAT [mi_etiqueta_repeat];
```

Ejemplo (pedazo de algoritmo ;D)
```
DELIMITER //
CREATE FUNCTION MiFactorial (valor INT)
RETURNS INT
BEGIN
      DECLARE factorial, contador INT DEFAULT 1;
      SET factorial = contador;

      IF valor = 0 THEN
      RETURN 0;
      END IF;

      IF valor = 1 THEN
      RETURN 1;
      END IF;

      REPEAT
      SET factorial= factorial * (contador+1);
      SET contador = contador+1;
      UNTIL contador=valor
      END REPEAT;

      RETURN factorial;
END //
DELIMITER ;
```

Y probamos de nuevo (lo bueno es que hemos controlado el 0 y el 1):
```
SELECT MIFactorial (0) AS 'Factorial';
SELECT MIFactorial (1) AS 'Factorial';
SELECT MIFactorial (5) AS 'Factorial';
```

Sentencia WHILE
La lista de sentencias dentro de un WHILE es repetida mientras la condición de búsqueda (condición_a_evaluar) es verdadera. Esta lista puede consistir de una o más sentencias. La sentencia WHILE puede usar también etiquetas (en este caso mi_bucle_while). Sintaxis:

```
[mi_bucle_while:] WHILE condición_a_evaluar DO
…
END WHILE [mi_bucle_while];
```

Veamos un ejemplo:

```
DELIMITER //
CREATE PROCEDURE contador (valor INT)
BEGIN
      DECLARE miContador INT DEFAULT 0;
      nuevo_contador: WHILE miContador <valor DO
      SET miContador = miContador +1;
      SELECT miContador;
      END WHILE nuevo_contador;
END //
DELIMITER ;
```

Y ahora llamamos al procedimiento:

```
CALL contador (8);
```

NOTA: He puesto este ejemplo con etiquetas, pero no soy partidario de ponerlas (en negrita).

Sentencia LOOP
Esta sentencia implementa una construcción de un bucle simple, permitiendo la ejecución repetida de la lista de una o mas sentencias. Las sentencias que están dentro del bucle se repiten hasta que se sale explícitamente del mismo. Por lo general, esto se hace con una sentencia LEAVE. La sentencia LOOP también puede utilizar etiquetas. Sintaxis

```
[mi_bucle_loop:] LOOP
…
LEAVE bucle;
END LOOP [mi_bucle_loop];
```

Veamos un ejemplo:

```
DELIMITER //
CREATE PROCEDURE TablaMultiplicar (valor INT)
BEGIN
      DECLARE operando, producto INT DEFAULT 0;
      bucle_multiplicar: LOOP
      SET operando = operando +1;
      SET producto = operando * valor;
      SELECT CONCAT (operando,' * ',valor,' = ',producto)
      AS TABLA_MULTIPLICAR;
      IF operando = 9 THEN
      LEAVE bucle_multiplicar;
      END IF;
      END LOOP;
END //
DELIMITER ;
```

Y ahora llamamos al procedimiento:

```
CALL TablaMultiplicar (8);
```

En las funciones almacenadas, un LOOP termina normalmente con una sentencia RETURN.

Otras construcciones de Control de Flujo
Las etiquetas permiten un mayor control de flujo del programa. MySQL ofrece dos construcciones adicionales para controlar el flujo del programa , que también utilizan bloques etiquetados.

- `LEAVE` – Esta sentencia es usada para salir de cualquier construcción de control de flujo dentro de una sentencia `LOOP, REPEAT` o `WHILE`. También puede usarse en sentencias compuestas `BEGIN...END` etiquetadas. Puede saltar a cualquier nombre de etiqueta y no tiene que tener el nombre de la etiqueta del `LOOP`.
- `ITERATE` – Esta sentencia indica que se repita el `LOOP`, `REPEAT` o `WHILE` de nuevo.

Es posible tener transacciones en las rutinas almacenadas, pero debe usarse la palabra reservada `START TRANSACTION` ya que `BEGIN` es ya una palabra reservada para rutinas almacenadas.

19 DISPARADORES (TRIGGERS)

19.1 Que son los disparadores

Los disparadores de una base de datos son objetos con nombre, mantenidos dentro de la misma, y que se activan cuando los datos dentro de una tabla son modificados. Los disparadores brindan un nivel de control y seguridad sobre los datos contenidos en las tablas. Por ejemplo: si usamos la base de datos world: ¿que se querría hacer después de cambiarle el código a un país? Este código es guardado en las tabla Country, City y CountryLanguage. Por tanto lo mejor sería cambiar ese código simultáneamente en las tres tablas. Un disparador puede hacer esa tarea.

Los disparadores proporcionan a los desarrolladores y administradores un mayor control sobre el acceso a datos concreto, así como la capacidad para realizar actividades especificas de logging y auditoria sobre los propios datos.
Son útiles para:

- Examinar los datos antes de ser insertados o modificados o para verificar eliminaciones o actualizaciones.
- Actuar como un filtro de los datos modificándolos antes de una inserción o actualización, en caso de que estén fuera de rango.
- Modificar el comportamiento de las sentencias INSERT, UPDATE y DELETE para una tabla en particular.
- Simular el comportamiento de claves foráneas cuando se utilicen motores de almacenamiento que no soportan este tipo de claves.
- Llevar a cabo logging.
- Construir tablas resumen de forma automática.

19.1.1 Creación de disparadores

Para definir un disparador para una tabla se usa la sentencia CREATE TRIGGER. Sintaxis:

```
CREATE TRIGGER nombre_disparador
(AFTER | BEFORE)
(INSERT | UPDATE | DELETE)
ON nombre_tabla
FOR EACH ROW
sentencias_disparador;
```

`AFTER | BEFORE` indica si se activa el disparador DESPUES o ANTES del evento asociado.
`INSERT | UPDATE | DELETE` indica el Evento que activa el disparador.

Veamos un ejemplo; primero nos creamos una copia de Ciudades llamada CiudadesEliminadas.

```
USE world;
CREATE TABLE CiudadesEliminadas LIKE Ciudades;
```

Y ahora el Trigger:

```
CREATE TRIGGER Disp_CiudadesEliminadas
AFTER DELETE
ON Ciudades
FOR EACH ROW
INSERT INTO CiudadesEliminadas (ID,Name)
VALUES (OLD.ID, OLD.Name);
```

Podemos usar las cláusulas OLD y NEW para guardar el Viejo o Nuevo campo.

Probamos el Trigger:

```
DELETE FROM Ciudades
WHERE CountryCode = 'AFG';
SELECT * FROM CiudadesEliminadas;
```

19.1.2 Eventos asociados al disparador

Los disparadores están asociados a tablas individuales. El método para activar los disparadores se llama evento y la siguiente lista describe los que se encuentran disponibles:

- `BEFORE`: se activan antes de que se escriban los cambios en los datos de la tabla de la base de datos subyacente. Estos tipos de eventos pueden capturar entradas de datos indebidas y corregirlas o rechazarlas antes de ser almacenadas. Hay tres eventos asociados a `BEFORE`:
  - `INSERT`: Este evento se dispara antes de que se inserten nuevos datos en la tabla.
  - `UPDATE`: Este evento se dispara antes de una actualización.
  - `DELETE`: Este evento se dispara antes de que se eliminen datos de la tabla.

- `AFTER`: Se activan después de que se escriban cambios en los datos de la tabla de la base de datos subyacente. Estos tipos de eventos se pueden usar para registrar o auditar las modificaciones de los datos de las bases de datos. Existen tres eventos de activación asociados con `AFTER`:
  - `INSERT`: Este evento se dispara después de que se inserten nuevos datos en la tabla.
  - `UPDATE`: Este evento se dispara después de una actualización.
  - `DELETE`: Este evento se dispara después de que se eliminen datos de la tabla.

NOTA: Las claves foráneas, que permiten actualizar y eliminar registros en cascada, actualmente NO activan disparadores.

Para mostrar los disparadores creados usaremos la sentencia:

```
SHOW TRIGGERS \G
```

Veamos otro ejemplo:
Crear un disparador que tras una inserción muestre la ID, el nombre, Region, Pais, Población, Fecha del cambio y Usuario en la que se ha realizado la inserción en una tabla que se llamará Ciudades_Insertadas.
Primero vamos a ver la estructura de Ciudades para hacernos una copia:

```
USE world;
SHOW CREATE TABLE Ciudades;
```

Comprobamos los triggers y borramos alguno si mas de uno apunta a la misma tabla:

```
SHOW TRIGGERS;
DROP TRIGGER Disp_CiudadesEliminadas
```

Nos creamos la tabla Ciudades_Insertadas (en negrita los nuevos campos):

```
CREATE TABLE CiudadesInsertadas (
      ID int(11) NOT NULL AUTO_INCREMENT,
      Name char(35) NOT NULL DEFAULT '',
      CountryCode char(3) NOT NULL DEFAULT '',
      District char(20) NOT NULL DEFAULT '',
      Population int(11) NOT NULL DEFAULT '0',
      Fecha DATE,
      usuario VARCHAR (40),
      PRIMARY KEY (ID),
      KEY CountryCode (CountryCode)
      ) ENGINE=InnoDB AUTO_INCREMENT=4087 DEFAULT CHARSET=latin1;
```

Ahora nos creamos el TRIGGER:

```
CREATE TRIGGER Disp_CiudadesInsertadas
BEFORE
INSERT
ON Ciudades
FOR EACH ROW
INSERT INTO CiudadesInsertadas
(ID,Name, CountryCode, District, Population, Fecha, Usuario)
VALUES
(New.ID, New.Name, New.CountryCode, New.District, New.Population, NOW(),
CURRENT_USER());
```

Obsérvese que hemos usado las funciones `NOW` (para poner la Fecha Actual) y `CURRENT_USER` (para poner el usuario que hace la inserción).

Hacemos un `INSERT` sobre Ciudades:

```
INSERT INTO Ciudades (Name, CountryCode, District, Population)
VALUES
('Ecija', 'ESP','Andalucia',60000);
```

Y comprobamos lo que se ha añadido en `CiudadesInsertadas`

```
SELECT * FROM CiudadesInsertadas;
```

NOTA: Los TRIGGER también se pueden emplear en Sentencias preparadas.

19.1.3 Manejo de errores en disparadores

Los errores ocurridos durante la ejecución de los disparadores pueden manejarse de la siguiente manera:

- Si un evento de disparador BEFORE falla, la operación sobre la fila correspondiente no es efectuada.
- Un evento de disparador AFTER es ejecutado solo si tanto el evento de disparador BEFORE (si hay alguno) como la operación sobre la fila se completaron exitosamente.
- En las tablas transaccionales, un error en un disparador (y por tanto de la sentencia completa) deberia generar una anulación (ROLLBACK) de todos los cambios efectuados por la sentencia. En las tablas no transaccionales, tal anulación no puede ser realizada, de modo que, aunque la sentencia falle, cualquier cambio efectuado antes del punto donde ocurrió el error, sigue en efecto.

A partir de la versión 5.1, existe un privilegio llamado TRIGGER de forma tal que el usuario no requiere privilegio SUPER para crear disparadores o usar la cláusula DEFINER.

19.2 Eliminación de disparadores

Para eliminar un TRIGGER se usa la sentencia DROP TRIGGER.
Además, si borramos una tabla, se borran los TRIGGER asociados a dicha tabla.
La sintaxis general será:
DROP TRIGGER nombre_esquema. Nombre_disparador.
Siendo nombre_esquema la base de datos donde estaré al disparador (es opcional si estamos en ella).

Desde la versión 5.1 de MySQL existe un privilegio llamado TRIGGER de forma tal que el usuario no requiere el privilegio SUPER para eliminar disparadores.

19.3 Restricciones de los disparadores

Los disparadores llevan un control y seguridad hasta los conjuntos de registros individuales de datos dentro de una tabla. Sin embargo, existen restricciones en cuanto a los disparadores. A continuación, lo que no está permitido:

- Sentencias SQL preparadas (PREPARE, EXECUTE, DEALLOCATE PREPARE)
- Sentencias que explícita o implícitamente efectúan COMMIT o ROLLBACK
- Sentencias que devuelven un conjunto de resultados. Se incluyen las sentencias SELECT que no tienen la cláusula INTO lista_variable y las sentencias SHOW. Un disparador puede procesar un conjunto de resultados bien sea con SELECT...INTO lista variables o usando un cursor y sentencias FETCH.
- Sentencias FLUSH
- Sentencias recursivas; es decir, los disparadores no pueden usarse recursivamente.

20 USUARIOS Y PRIVILEGIOS

20.1 Introducción

El acceso al servidor MySQL está controlado por usuarios y privilegios. Los usuarios del servidor MySQL no tienen ninguna correspondencia con los usuarios del sistema operativo. Aunque en la práctica es común que algún administrador de MySQL asigne los mismos nombres que los usuarios tienen en el sistema, son mecanismos totalmente independientes y suele ser aconsejable en general.

El usuario administrador del sistema MySQL se llama root. Igual que el superusuario de los sistemas tipo UNIX.

Además del usuario root, las instalaciones nuevas de MySQL incluyen el usuario anónimo, que tiene permisos sobre la base de datos test. Si queremos, también podemos restringirlo asignándole una contraseña. El usuario anónimo de MySQL se representa por una cadena vacía.

Veamos otra forma de asignar contraseñas a un usuario, desde el cliente de mysql y como usuario root. La estructura será:

```
set password for 'usuario'@'localhost' = password('nuevapasswd');
```

Por ejemplo, para asignar una contraseña al usuario anonimo:

```
set password for ''@'localhost' = password('claveanonimo');
exit
mysql -u '' -p
```

Escribimos la clave

```
SHOW DATABASES
```

La administración de privilegios y usuarios en MySQL se realiza a través de las sentencias:

- **GRANT.** Otorga privilegios a un usuario, en caso de no existir, se creará el usuario.
- **REVOKE.** Elimina los privilegios de un usuario existente.
- **SET PASSWORD.** Asigna una contraseña.
- **DROP USER.** Elimina un usuario.

20.2 La sentencia GRANT

La sintaxis simplificada de grant consta de tres secciones. No puede omitirse ninguna, y es importante el orden de las mismas:

```
GRANT lista de privilegios
ON base de datos.tabla
TO usuario
```

Para el siguiente ejemplo nos salimos del usuario anónimo y volvemos a root:

```
exit
mysql -u root -p
```

Nos creamos la Base de datos demo y dentro la tabla precios

```
CREATE DATABASE demo;
USE demo;
CREATE TABLE precios
(
      id INT NOT NULL
);
```

Y ahora es cuando creamos el usuario y le damos permisos:

```
GRANT UPDATE, INSERT, SELECT
ON demo.precios
TO visitante@localhost;
```

En la primera línea se especifican los privilegios que serán otorgados, en este caso se permite actualizar (update), insertar (insert) y consultar (select). La segunda línea especifica que los privilegios se aplican a la tabla precios de la base de datos demo. En la última línea se encuentra el nombre del usuario y el equipo desde el que se va a permitir la conexión.

El comando GRANT crea la cuenta si no existe y, si existe, agrega los privilegios especificados.Y de paso le podemos poner una contraseña después:

```
set password for visitante@'localhost' = password('clavevisitante');
```

Para comprobarlo salimos de root, accedemos:

```
exit
mysql -u visitante -p
SHOW DATABASES;
```

Veremos information_schema, test y el demo creado.

Volvemos a root y nos creamos otro usuario para ver como crearlo y poner la contraseña en la misma sentencia:

```
exit
mysql -u root -p
GRANT UPDATE, INSERT, SELECT
ON demo.precios
TO visitante1@localhost identified by 'clavevisitante1';
```

Y de nuevos podemos hacer las pruebas anteriores.

20.2.1 Creación de Varios usuarios.

En la misma sentencia podemos crear varios usuarios y asignar permisos con contraseña o no. Veamos un ejemplo:

```
GRANT UPDATE, INSERT, SELECT
ON demo.precios
TO visitante2@localhost,
visitante3@localhost identified by 'clavevisitante3',
otro@equipo.remoto.com;
```

NOTA: Este último dará un error porque no existe el dominio equipo.remoto.com.

20.2.2 Especificación de lugares origen de la conexión

MySQL proporciona mecanismos para permitir que el usuario realice su conexión desde diferentes equipos dentro de una red específica, sólo desde un equipo, o únicamente desde el propio servidor.

```
GRANT UPDATE, INSERT, SELECT
-> ON demo.precios
-> TO visitante@'%.empresa.com';
```

NOTA: Este último dará un error porque no existe el dominio empresa.com.

El carácter % se utiliza de la misma forma que en el comando like: sustituye a cualquier cadena de caracteres. En este caso, se permitiría el acceso del usuario 'visitante' (con contraseña, si la tuviese definida) desde cualquier equipo del dominio 'empresa.com'. Obsérvese que es necesario entrecomillar el nombre del equipo origen con el fin de que sea aceptado por MySQL. Al igual que en like, puede utilizarse el carácter '_'.

Entonces, para permitir la entrada desde cualquier equipo en Internet, escribiríamos:
Entonces, para permitir la entrada desde cualquier equipo en Internet, escribiríamos:
```
to visitante@'%'
```

Obtendríamos el mismo resultado omitiendo el nombre del equipo origen y escribiendo simplemente el nombre del usuario:
```
to visitante
```

NOTA: En ambos casos estamos hablando de un evidente agujero de seguridad.

Los anfitriones válidos también se pueden especificar con sus direcciones IP.
```
…
to visitante@192.168.128.10,
to visitante@'192.168.126.%'
...
```

Esto significa que, en la misma sentencia, damos permisos a los visitantes de las Ips 192.168.128.10 y todas las del rango 192.168.126.

IMPORTANTE: Los caracteres ' %' y '_' no se permiten en los nombres de los usuarios.

20.2.3 Especificación de bases de datos y tablas

Después de analizar las opciones referentes a los lugares de conexión permitidos, veamos ahora cómo podemos limitar los privilegios a bases de datos, tablas y columnas.

En el siguiente ejemplo otorgamos privilegios sobre todas las tablas de la base de datos demo.
```
GRANT ALL
ON demo.*
TO 'visitantedemo'@'localhost';
```

Podemos obtener el mismo resultado de esta forma:
```
USE demo;
GRANT ALL
ON *
TO 'visitantedemo1'@'localhost';
```

De igual modo, al especificar sólo el nombre de una tabla se interpretará que pertenece a la base de datos en uso:

```
USE demo;
GRANT ALL
ON precios
TO 'visitantedemo2'@'localhost';
```

Opciones para la cláusula ON del comando GRANT
. → Todas las Bases de Datos y Todas las Tablas
base.* → Todas las tablas de la base de datos especificada
tabla → Tabla especificada de la base de datos en uso
* → Todas las tablas de la base de datos en uso

Para conocer los usarios conectados al Servidor:
```
SHOW PROCESSLIST;
```

20.2.4 Especificación de columnas

GRANT permite también especificar las columnas y las sentencias permitidas en cada una de ellas. En primer lugar, en demo modificaremos la tabla precios:

```
USE demo;
ALTER TABLE precios
ADD precio INT NOT NULL,
ADD empresa VARCHAR(20) NOT NULL;
```

Ahora vemos la nueva estructura;

```
DESCRIBE precios;
```

Veamos como asignar el privilegio UPDATE a visitantedemo para las tablas precio y empresa:

```
GRANT UPDATE (precio,empresa)
ON demo.precios
TO visitantedemo5@localhost IDENTIFIED by 'clavevisitantedemo5';
```

¿que pasa si queremos hacer un SELECT con visitantedemo5?

```
Exit
mysql -u  visitantedemo5 -p
clavevisitantedemo5
```

USE demo;

```
SELECT * FROM precios;
```

Pues que aparece:

```
SELECT command denied to user 'visitantedemo5'@'localhost' for table
'precios'
```

Podemos especificar privilegios diferentes para cada columna o grupos de columnas:

```
GRANT UPDATE (precio), SELECT (precio, empresa)
ON demo.precios
TO visitante@localhost;
```

20.2.5 Tipos de privilegios

MySQL proporciona una gran variedad de tipos de privilegios.

- Privilegios relacionados con tablas:
  `ALTER, CREATE, DELETE, DROP, INDEX, INSERT, SELECT, UPDATE`

- Algunos privilegios administrativos:
  `FILE` → otorga permiso para leer y escribir archivos en la máquina del servidor.
  `PROCCESS` → puede utilizarse para ver el texto de las consultas que se estén ejecutando actualmente.
  `SUPER` → puede utilizarse para cerrar la conexión a otros clientes o cambiar como el servidor funciona.
  `RELOAD` → Comunica al servidor que debe releer las tablas grant a memoria.

  `REPLICATION CLIENT` → permite la utilización de las sentencias SHOW MASTER `STATUS y SHOW SLAVE STATUS.`
  `GRANT OPTION` → permite dar a otros usuarios las privilegios que uno mismo posee.
  `SHUTDOWN` → Apaga el servidor.
  Mas información: http://dev.mysql.com/doc/refman/5.0/es/privileges-provided.html

- Algunos privilegios para fines diversos:
  `LOCK TABLES, SHOW DATABASES, CREATE TEMPORARY TABLES.`

El privilegio all otorga todos los privilegios exceptuando el privilegio grant option. Y el privilegio usage no otorga ninguno, lo cual es útil cuando se desea, por ejemplo, simplemente cambiar la contraseña:

```
GRANT USAGE
ON *.*
TO visitante@localhost identified by 'secreto';
```

En entornos grandes, es frecuente encontrarse en la necesidad de delegar el trabajo de administrar un servidor de bases de datos para que otros usuarios, además del administrador, puedan responsabilizarse de otorgar privilegios sobre una base de datos particular. Esto se puede hacer en MySQL con el privilegio grant option:

```
GRANT ALL
ON demo.*
TO operador@localhost
WITH GRANT OPTION;
```

De este modo el usuario operador podrá disponer de todos los privilegios sobre la base de datos demo, incluido el de controlar el acceso a otros usuarios.

20.2.6 Opciones de encriptación

MySQL puede establecer conexiones seguras encriptándolas mediante el protocolo SSL*; de esta manera, los datos que se transmiten (tanto la consulta, en un sentido, como el resultado, en el otro) entre el cliente y el servidor estarán protegidos contra intrusos. Para especificar que un usuario debe conectarse obligatoriamente con este protocolo, se utiliza la cláusula require:

```
GRANT ALL
ON *.*
TO visitante@localhost
REQUIRE ssl;
```

Las conexiones encriptadas ofrecen protección contra el robo de información, pero suponen una carga adicional para el servicio, que debe desencriptar la petición del cliente y encriptar la respuesta (además de un proceso más largo de negociación al conectar), por ello, merman el rendimiento del SGBD.

20.2.7 Limites de uso

Los recursos físicos del servidor siempre son limitados: si se conectan muchos usuarios al mismo tiempo al servidor y realizan consultas o manipulaciones de datos complejas, es probable que pueda decaer el rendimiento notablemente.

Una posible solución a este problema es limitar a los usuarios el trabajo que pueden pedir al servidor con tres parámetros:

- Máximo número de conexiones por hora `(MAX_CONNECTIONS_PER_HOUR)`
- Máximo número de consultas por hora `(QUERIES_PER_HOUR)`
- Máximo número de actualizaciones por hora `(UPDATES_PER_HOUR)`

Veamos un ejemplo:

```
GRANT ALL
ON *.*
TO visitante@localhost
WITH MAX_CONNECTIONS_PER_HOUR 3
MAX_QUERIES_PER_HOUR 300
MAX_UPDATES_PER_HOUR 30;
```

20.2.8 Eliminar privilegios

El comando revoke permite eliminar privilegios otorgados con grant a los usuarios.

Veamos la sintaxis:

```
REVOKE privilegios [(columnas)]
ON elemento
FROM nopmbre_de_usuario
```

Si se ha concedido privilegios con la clásula WITH GRANT OPTION, se pueden revocar de la siguiente forma:

```
REVOKE GRANT OPTION
ON elemento
FROMnombre_de_usuario
```

Veamos un ejemplo:

```
REVOKE ALL
ON *.*
FROM visitante@localhost;
```

Al ejecutar este comando se le retiran al usuario visitante todos sus privilegios sobre todas las bases de datos, cuando se conecta desde localhost.

El comando anterior no retira todos los privilegios del usuario visitante, sólo se los retira cuando se conecta desde localhost. Si el usuario se conecta desde otra localización (y tenía permiso para hacerlo) sus privilegios permanecen intactos.

También podemos revocar privilegios determinados.

```
GRANT SELECT, INSERT, UPDATE, DELETE, INDEX, ALTER, CREATE, DROP
ON demo
TO usuariodemo IDENTIFIED BY 'claveusuariodemo';
```

Y ahora queremos quitarle los privilegios de administración sobre tablas, ALTER, CREATE y DROP, quedando la sentencia:

```
REVOKE ALTER, CREATE, DROP
ON demo
FROM  usuariodemo;
```

20.3 La base de datos de privilegios: mysql

MySQL almacena la información sobre los usuarios y sus privilegios en una base de datos como cualquier otra, cuyo nombre es mysql. Si exploramos su estructura, entenderemos la manera como MySQL almacena la información de sus usuarios y privilegios:

mysql -u root -p
USE mysql;
SHOW TABLES;

SHOW COLUMNS FROM user;

```
mysql> SHOW COLUMNS FROM user;
+------------------------+-----------------------------------+------+-----+---------+-------+
| Field                  | Type                              | Null | Key | Default | Extra |
+------------------------+-----------------------------------+------+-----+---------+-------+
| Host                   | char(60)                          | NO   | PRI |         |       |
| User                   | char(16)                          | NO   | PRI |         |       |
| Password               | char(41)                          | NO   |     |         |       |
| Select_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Insert_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Update_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Delete_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Create_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Drop_priv              | enum('N','Y')                     | NO   |     | N       |       |
| Reload_priv            | enum('N','Y')                     | NO   |     | N       |       |
| Shutdown_priv          | enum('N','Y')                     | NO   |     | N       |       |
| Process_priv           | enum('N','Y')                     | NO   |     | N       |       |
| File_priv              | enum('N','Y')                     | NO   |     | N       |       |
| Grant_priv             | enum('N','Y')                     | NO   |     | N       |       |
| References_priv        | enum('N','Y')                     | NO   |     | N       |       |
| Index_priv             | enum('N','Y')                     | NO   |     | N       |       |
| Alter_priv             | enum('N','Y')                     | NO   |     | N       |       |
```

SHOW COLUMNS FROM db;

```
mysql> SHOW COLUMNS FROM db;
+-----------------------+---------------+------+-----+---------+-------+
| Field                 | Type          | Null | Key | Default | Extra |
+-----------------------+---------------+------+-----+---------+-------+
| Host                  | char(60)      | NO   | PRI |         |       |
| Db                    | char(64)      | NO   | PRI |         |       |
| User                  | char(16)      | NO   | PRI |         |       |
| Select_priv           | enum('N','Y') | NO   |     | N       |       |
| Insert_priv           | enum('N','Y') | NO   |     | N       |       |
| Update_priv           | enum('N','Y') | NO   |     | N       |       |
| Delete_priv           | enum('N','Y') | NO   |     | N       |       |
| Create_priv           | enum('N','Y') | NO   |     | N       |       |
| Drop_priv             | enum('N','Y') | NO   |     | N       |       |
| Grant_priv            | enum('N','Y') | NO   |     | N       |       |
| References_priv       | enum('N','Y') | NO   |     | N       |       |
| Index_priv            | enum('N','Y') | NO   |     | N       |       |
| Alter_priv            | enum('N','Y') | NO   |     | N       |       |
| Create_tmp_table_priv | enum('N','Y') | NO   |     | N       |       |
| Lock_tables_priv      | enum('N','Y') | NO   |     | N       |       |
| Create_view_priv      | enum('N','Y') | NO   |     | N       |       |
| Show_view_priv        | enum('N','Y') | NO   |     | N       |       |
| Create_routine_priv   | enum('N','Y') | NO   |     | N       |       |
| Alter_routine_priv    | enum('N','Y') | NO   |     | N       |       |
| Execute_priv          | enum('N','Y') | NO   |     | N       |       |
| Event_priv            | enum('N','Y') | NO   |     | N       |       |
| Trigger_priv          | enum('N','Y') | NO   |     | N       |       |
+-----------------------+---------------+------+-----+---------+-------+
22 rows in set (0.00 sec)
```

Es posible realizar modificaciones directamente sobre estas tablas y obtener los mismos resultados que si utilizáramos los comandos grant, revoke, set password o drop user.

Para empezar veremos los usuarios actuales:
SELECT host, user, password FROM user;

```
mysql> SELECT host, user, password FROM user;
+-----------+---------------+-------------------------------------------+
| host      | user          | password                                  |
+-----------+---------------+-------------------------------------------+
| localhost | root          | *81F5E21E35407D884A6CD4A731AEBFB6AF209E1B |
| linux     | root          | *81F5E21E35407D884A6CD4A731AEBFB6AF209E1B |
| localhost |               | *DFB82E18A9AC4190E35E1F6347D3B1834DA0384C |
| linux     |               |                                           |
| localhost | pma           | *6C8DC88838BA26F23FC09ED48861E966C911B4CB |
| localhost | visitante     | *B03CEEAC29B017382BA994FCF6D62F6823D1E034 |
| localhost | visitante1    | *131F082B2116801A7B76FAAA72064554A756EF14 |
| localhost | visitantedemo |                                           |
| localhost | visitantedemo1|                                           |
| localhost | visitantedemo2|                                           |
| localhost | visitantedemo5| *468D39A349EEA6A9FD6EE3A94B91C5DF6832C2F8 |
| localhost | operador      |                                           |
| %         | usuariodemo   | *78B7AFD4955C7A6A5D53BB490FAED261658E8075 |
+-----------+---------------+-------------------------------------------+
```

Veamos como cambiar por ejemplo la contraseña a visitante:

```
UPDATE user
SET PASSWORD = PASSWORD('nuevaclavevisitante')
WHERE User ='visitante' AND Host = 'localhost';

FLUSH PRIVILEGES;

SELECT host, user, password FROM user;
```

NOTA: Como puede comprobarse las claves aparecen codificadas

El comando FLUSH PRIVILEGES solicita a MySQL que vuelva a leer las tablas de privilegios. En el momento de ejecutarse, el servidor lee la información de estas tablas sobre privilegios. Pero si se han alterado las tablas manualmente, no se enterará de los cambios hasta que utilicemos el comando FLUSH PRIVILEGES.

Veamos para qué sirven las principales tablas de la Base de Datos mysql:

user	→ Cuentas de usuario y sus privilegios globales
db	→ Privilegios sobres bases de datos
tables_priv	→ Privilegios sobre tablas
columns_priv	→ Privilegios sobre columnas
host	→ Privilegios de otros equipos anfitriones sobre bases de datos

El acceso directo a las tablas de privilegios es útil en varios casos; por ejemplo, para borrar un usuario del sistema en las versiones de MySQL anteriores a la 4.1.1:

```
DELETE FROM user
WHERE User = 'visitantedemo5' AND Host = 'localhost';

FLUSH PRIVILEGES;
```

Se otorgan derechos a un usuario con dos comandos GRANT.

Observando el contenido de la base de datos de privilegios, podemos entender el comportamiento de los comandos grant y revoke. Primero asignamos privilegios para usar el comando SELECT al usuario visitante con dos comandos GRANT: el primero de ellos le permite el ingreso desde el sistema operativo linux y el segundo le otorga el mismo tipo de privilegio, pero desde cualquier equipo en Internet.

Primero para el Sistema Operativo:

```
GRANT SELECT
ON *.*
TO visitante@linux;
```

Y ahora para todos los dominios:

```
GRANT SELECT
ON *.*
TO visitante@'%';
```

Consultando la tabla user de la base de datos de privilegios, podemos observar los valores 'Y' en la columna del privilegio select.

```
SELECT user, host, select_priv
FROM user
WHERE user = 'visitante';
```

```
mysql> SELECT user, host, select_priv
    -> FROM user
    -> WHERE user = 'visitante';
+-----------+-----------+-------------+
| user      | host      | select_priv |
+-----------+-----------+-------------+
| visitante | localhost | N           |
| visitante | linux     | Y           |
| visitante | %         | Y           |
+-----------+-----------+-------------+
```

Ahora solicitamos eliminar el privilegio select de todas las bases de datos y de todos los equipos en Internet.

```
REVOKE ALL
ON *.*
FROM visitante@'%';
```

Y repetimos el SELECT anterior:

```
SELECT user, host, select_priv
FROM user
WHERE user = 'visitante';
```

```
mysql> SELECT user, host, select_priv
    -> FROM user
    -> WHERE user = 'visitante';
+-----------+-----------+-------------+
| user      | host      | select_priv |
+-----------+-----------+-------------+
| visitante | localhost | N           |
| visitante | linux     | Y           |
| visitante | %         | N           |
+-----------+-----------+-------------+
```

En la tabla user observamos que, efectivamente, se ha eliminado el privilegio para visitante@'%' pero no para 'visitante@linux'. MySQL considera que son direcciones diferentes y respeta los privilegios otorgados a uno cuando se modifica otro.

20.3.1 La Sentencia CREATE USER

Desde la versión 5 de MySQL se pueden crear usuarios sin necesidad de conceder privilegios.
Para usarla se debe tener el privilegio GRANT OPTION para la base de datos mysql.
Para cada cuenta, CREATE USER crea un nuevo registro en la tabla mysql.user sin privilegios. Se produce un error si la cuenta ya existe.

Se le puede dar una contraseña a la cuenta con la cláusula opcional IDENTIFIED.
Los valores user y password se dan del mismo modo que para la sentencia .

La sintaxis será:

```
CREATE USER ususario
[IDENTIFIED BY [PASSWORD] 'contraseña']
    [, user [IDENTIFIED BY [PASSWORD] 'password']] ...
```

Como puede comprobarse, en la misma sentencia, además, podemos crear varios usuarios.

```
CREATE USER usuario1 IDENTIFIED BY 'claveusuario1',
usuario2 IDENTIFIED BY 'claveusuario2',
usuario3 IDENTIFIED BY 'claveusuario3';
```

Y vemos como queda con:

```
SELECT user, host, select_priv
FROM user
WHERE user LIKE 'usuario%';
```

```
mysql> SELECT user, host, select_priv
    -> FROM user
    -> WHERE user LIKE 'usuario%';
+-------------+------+-------------+
| user        | host | select_priv |
+-------------+------+-------------+
| usuariodemo | %    | N           |
| usuario1    | %    | N           |
| usuario2    | %    | N           |
| usuario3    | %    | N           |
+-------------+------+-------------+
```

20.3.2 DROP USER

Para eliminar un usuario tenemos REVOKE, pero desde MySQL5 también DROP USER.
La sintaxis es:
DROP USER user [, user] ...

La sentencia DROP USER elimina una o más cuentas MySQL. Para usarla se debe poseer el privilegio GRANT OPTION para la base de datos mysql. Cada cuenta se nombra usando el mismo formato que hemos visto, por ejemplo, 'ivan'@'localhost'.
Las partes del usuario y la máquina del nombre de la cuenta corresponden a los valores de las columnas User y Host del registro de la tabla user.

DROP USER se añadió en MySQL 4.1.1 y originalmente sólo borraba cuentas que no tengan privilegios. En MySQL 5.0.2, fue modificada para borrar también cuentas que tengan privilegios. Esto significa que el procedimiento para eliminar una cuenta depende de la versión de MySQL.

Veamos un ejemplo:
DELETE FROM mysql.user
WHERE User='usuario3' and Host='%';

Luego hay que realizar la actualización:
FLUSH PRIVILEGES;

Y vemos como queda la tabla de usuarios

```
SELECT user, host, select_priv
FROM user
WHERE user LIKE 'usuario%';
```

20.3.3 La sentencia UPDATE USER

Permite cambiar la contraseña de usuario.
Por ejemplo, queremos cambiar la contraseña de usuario2:

```
UPDATE user
SET PASSWORD = PASSWORD('nuevaclaveusuario2')
WHERE User ='usuario2' AND Host = '%';

FLUSH PRIVILEGES;

SELECT user, host, select_priv, password
FROM user
WHERE user LIKE 'usuario%';
```

Mas información en:
http://dev.mysql.com/doc/refman/5.0/es/passwords.html

21 ADMINISTRACIÓN

El cliente mysqladmin:
http://dev.mysql.com/doc/refman/5.0/es/mysqladmin.html

21.1 Estructura interna

Para saber las variables del servidor usaremos la sentencia SHOW VARIABLES. Dicha sentencia tiene la siguiente sintaxis:

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'patron']
```

Para establecer el valor de una variable GLOBAL, debe emplearse una de las siguientes sintaxis:

```
SET GLOBAL sort_buffer_size=valor;
SET @@global.sort_buffer_size=valor;
```

Para establecer el valor de una variable SESSION, debe emplearse una de las siguientes sintaxis:

```
SET SESSION sort_buffer_size=valor;
SET @@session.sort_buffer_size=valor;
SET sort_buffer_size=valor;
```

`LOCAL` es un sinónimo de `SESSION`.

Para recuperar el valor de una variable GLOBAL debe utilizarse una de las siguientes sentencias:

```
SELECT @@global.sort_buffer_size;
SHOW GLOBAL VARIABLES like 'sort_buffer_size';
```

Para recuperar el valor de una variable SESSION debe utilizarse una de las siguientes sentencias:

```
SELECT @@sort_buffer_size;
SELECT @@session.sort_buffer_size;
SHOW SESSION VARIABLES like 'sort_buffer_size';
```

Aquí, también, `LOCAL` es un sinónimo de `SESSION`.

Aprovechando esta sentencia, vamos a ver todas las variables de Servidor.
Mas información: http://dev.mysql.com/doc/refman/5.0/es/server-system-variables.html
http://dev.mysql.com/doc/refman/5.0/es/system-variables.html
http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html

- auto_increment_increment
  `SHOW VARIABLES LIKE 'auto_inc%';`
  - auto_increment_increment controla el intervalo en que se incrementa el valor de columna. Tipo: INT. Puede ser un entero de 1 a 65535.
  - auto_increment_offset determina el punto de inicio para el valor de las columnas AUTO_INCREMENT. Tipo: INT.

Ejemplo:

```
SHOW VARIABLES LIKE 'auto_inc%';
USE test;
CREATE TABLE autoinc1
      (col INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
SET @@auto_increment_increment=10;
SET @@auto_increment_offset=5;
SHOW VARIABLES LIKE 'auto_inc%';
INSERT INTO autoinc1 VALUES (NULL), (NULL), (NULL), (NULL);
SELECT col FROM autoinc1;
```

- `autocommit`: Estado del COMMIT (Predeterminado: ON, Tipo: Boolean).
- `automatic_sp_privileges`: Asigna privilegios EXECUTE Y ALTER ROUTINE al creado de la rutina almacenada. (Predeterminado: 1; Tipo Boolean)
  `SELECT @@automatic_sp_privileges;`

- `back_log`: El numero de peticiones de conexión que puede tener MySQL.
  Tipo: INT. De forma predeterminada es 50.
  `SHOW VARIABLES LIKE 'back_log%';`
- `basedir`: El directorio de instalación de MySQL.
  Esta variable puede cambiarse con la opción --basedir.
  SELECT @@basedir;

- `big_tables`: Si se establece en 1, todas las tablas temporales se almacenan en disco en lugar de en la memoria.
  Reduce algo el rendimiento, pero el error `the table <nombre_tabla> is full` no se produce para operaciones SELECT que requieran una tabla temporal grande. (Predeterminado: 0; Tipo Boolean)
  `SELECT @@big_tables;`
- `binlog_cache_size`: El tamaño de la caché para tratar comandos SQL para el log binario durante una transacción. (Predeterminado: 32768, Formato INT).
  Existen algunos formatos mas visibles con:
  SHOW VARIABLES LIKE 'binlog_%';
- `bulk_insert_buffer_size`: Esta variable limita el tamaño de la memoria caché por hilo para tablas MyISAM. Si se establece en 0 desactiva esta optimización. El valor por defecto es de 8 MB. Formato INT.
  `SELECT @@`bulk_insert_buffer_size;

- `character_set_client`: El juego de caracteres para las declaraciones que llegan desde el cliente. Hay varias variables relacionadas visibles con:
  `SHOW VARIABLES LIKE 'character_set_%';`
- `character_set_connection`: El conjunto de caracteres utilizado para los literales que no tienen un introductor de conjunto de caracteres y conversión de número a cadena.
- `character_set_database`: El conjunto de caracteres utilizado por la base de datos predeterminada.
- `character_set_filesystem`: El conjunto de caracteres de sistema de archivos definido.
- `character_set_server`: El servidor del conjunto de caracteres por defecto.
- `character_set_system`: El conjunto de caracteres usado por el servidor para almacenar identificadores.
- `character_sets_dir`: El directorio donde los conjuntos de caracteres están instalados.

- `collation_connection`: La ordenación de caracteres para la conexión predeterminada.
  Hay varias variables relacionadas visibles con:
  `SHOW VARIABLES LIKE 'collation_%';`
- `collation_database`: Ordenación para la base de datos predeterminada.
  `SET @@collation_database='latin1_spanish_ci';`
- `collation_server`: El servidor del Conjunto de caracteres.
  SET @@`collation_server`='latin1_spanish_ci';
  `SHOW VARIABLES LIKE 'collation_%';`

- `completion_type`: El tipo de terminación de la transacción. Esta variable puede tomar los valores que se muestran en la siguiente tabla. A partir de MySQL 5.5.3, la variable se le puede asignar ya sea utilizando los valores de nombre o los correspondientes valores enteros. Antes de 5.5.3, sólo los valores enteros puede ser utilizado. completion_type afecta a las transacciones que comienzan con `START TRANSACTION` o `BEGIN` y termina con `COMMIT` o `ROLLBACK` .

Valor	Descripción
`NO_CHAIN` (o 0)	`COMMIT` y `ROLLBACK` no se ven afectados. Este es el valor predeterminado.
`CHAIN` (o 1)	`COMMIT` y `ROLLBACK` son equivalentes a `COMMIT AND CHAIN y ROLLBACK AND CHAIN` , respectivamente. (Una nueva transacción se inicia inmediatamente con el mismo nivel de aislamiento como la operación que acaba de terminar.)
`RELEASE` (o 2)	`COMMIT` y `ROLLBACK` son equivalentes a `COMMIT RELEASE y ROLLBACK RELEASE` , respectivamente. (El servidor se desconecta después de finalizar la transacción.)

- `concurrent_insert`. Gestión de Inserciones Concurrentes.
  Esta variable puede tomar los valores que se muestran en la siguiente tabla. A partir de MySQL 5.5.3, la variable se le puede asignar ya sea utilizando los valores de nombre o los correspondientes valores enteros. Antes de 5.5.3, sólo los valores enteros puede ser utilizado.

Valor	Descripción
`NEVER` (o 0)	Desactiva las inserciones concurrentes
`AUTO` (o 1)	(Predeterminado) Permite inserciones concurrentes para tablas MyISAM que no tengan huecos.
`ALWAYS` (o 2)	Permite inserciones concurrentes para todas las tablas MyISAM , incluso los que tienen huecos. Para una tabla con un hueco, las nuevas filas se insertan al final de la tabla si está en uso por otro hilo. En caso contrario, se activa un bloqueo de escritura normal y inserta la fila en el hueco.

- `connect_timeout`: El número de segundos que el servidor espera para un paquete de conexión antes de responder con `Bad handshake` .
  El valor predeterminado es de 10 segundos.
- `datadir`: El directorio de datos de MySQL.
  En concreto, para saber el directorio de datos sería:
  `SHOW VARIABLES LIKE 'datadir';`

- `date_format`: Formato de día. Predeterminado `%Y-%m-%d`
- `datetime_format`: Formato de Dia/hora. Predeterminado `%Y-%m-%d %H:%i:%s`

- `default_storage_engine`: El motor de almacenamiento por defecto.
  `SELECT @@default_week_format`
- `default_week_format`: El valor por defecto que se utilizará para la función `WEEK()`.
  `SELECT @@default_week_format`

- `delay_key_write`: Esta opción sólo se aplica a Tablas MyISAM . Si está activado, el búffer de claves no se refresca en cada actualización del índice, sólo cuando la tabla se cierra. Puede tener uno de los siguientes valores según esta tabla:

Opción	Descripción
`OFF`	`DELAY_KEY_WRITE` se ignora.
`ON`	MySQL activa `DELAY_KEY_WRITE` en declaraciones `CREATE TABLE`. (Predeterm)
`ALL`	Todas las nuevas tablas abiertas se tratan como si se hubieran creado con la opción `DELAY_KEY_WRITE` habilitada.

NOTA IMPORTANTE: Si se habilita bloqueo externo con `--external-locking` , no hay ninguna protección contra corrupción de los índices para las tablas que utilizan claves retardada.

- `delayed_insert_limit`: Limite de Inserciones Retardadas. Existe la sentencia INSERT DELAYED que permite ponerse en cola para la inserción de un registro sin esperar a que el anterior o anteriores se completen. Predeterminado: 100.
  Mas información: http://dev.mysql.com/doc/refman/5.5/en/insert-delayed.html
- `delayed_insert_timeout`: Timeout de inserciones retardadas. Defecto: 30sgs.
- `delayed_queue_size`: Tamaño de la cola para inserciones retardadas. Defecto: 1000.

- `div_precision_increment`: Esta variable indica el número de dígitos que se aumentará la escala del resultado de las operaciones realizadas con operador división /. El valor por defecto es 4. Los valores mínimo y máximo son 0 y 30, respectivamente. El siguiente ejemplo ilustra el efecto de aumentar el valor predeterminado. Cuanto mas dígitos de precisión, mas carga para el servidor.
```
SELECT 1/7;
SET div_precision_increment = 12;
SELECT 1/7;
```

- **`engine_condition_pushdown`:** La condición de optimización de motores de pushdown permite el procesamiento de ciertas comparaciones para una ejecución más eficiente. Predeterminado: ON.
  ATENCIÓN: Esta variable es obsoleta desde MySQL 5.5.3 y se elimina en MySQL 5.6.
- `error_count`: El número de errores de la última instrucción. Esta variable es de sólo lectura.
- `event_scheduler`: Esta variable indica el estado del programador de eventos, los valores posibles son `ON` , `OFF` , y `DISABLED` , con el valor predeterminado es `OFF` .
  Mas información: http://dev.mysql.com/doc/refman/5.5/en/events-configuration.html
  http://dev.mysql.com/doc/refman/5.5/en/events-privileges.html

- `expire_logs_days`: El número de días para la eliminación automática de archivos de registro binario. El valor predeterminado es 0, que significa "sin eliminación automática." El borrado sucederá en el arranque y cuando el registro binario se refresco (FLUSH).
- `external_user`: El nombre de usuario externo que se utiliza durante el proceso de autenticación. Está vacío de forma predeterminada.

- `flush`: Si está activado (`ON`) los refrescos del servidor sincronizan todos los cambios en el disco después de cada comando SQL.
- `flush_time`: Si se establece en un valor distinto de cero, todas las tablas se cierran cada `<flush_time>` segundos para liberar recursos y sincronizar datos no volcados en disco. Esta opción es utilizada solamente en sistemas con recursos mínimos.
  `SET GLOBAL flush_time=0;`

- `foreign_key_checks`: Si se establece en 1 (por defecto), las claves foráneas para InnoDB se comprueban. Si se establece en 0, se ignoran. Cuidado con esto: puede provocar incoherencias.

- `ft_boolean_syntax`: Lista de operadores booleanos soportados por búsquedas de texto completo
- `ft_max_word_len`: La longitud máxima de la palabra a incluirse en un índice `FULLTEXT` .
- `ft_min_word_len`: La longitud mínima de la palabra a incluirse en un índice `FULLTEXT` .
- `ft_query_expansion_limit`: El número de mejores coincidencias a usar en búsquedas full-text realizadas usando `WITH QUERY EXPANSION`.
- `ft_stopword_file`: El fichero del que lee la lista de palabras de detención en búsquedas full-text. Todas las palabras del fichero se usan; los comentarios no se tienen en cuenta. Por defecto, se usa una lista de palabras de detención (como se define en el fichero myisam/ft_static.c). Actualizar esta variable con una cadena vacía ('') desactiva el filtrado de palabras de detención.
  Nota: índices `FULLTEXT` deben reconstruirse tras cambiar esta variable o los contenidos del fichero de palabras de detención. Use `REPAIR TABLE` tbl_name `QUICK`.

- `general_log`: Si el registro general de consultas está habilitado.
- `general_log_file`: El nombre del archivo de consulta de registro general.
- `group_concat_max_len`: El valor máximo permitido para la longitud del resultado de la función `GROUP_CONCAT()` .
- `have_compress`: Si está disponible la biblioteca de compresión zlib en el servidor. Si no lo está, las funciones `COMPRESS() y UNCOMPRESS()` no pueden usarse.
- `have_crypt`: Si la llamada de sistema crypt() está disponible en el servidor. Si no, la función `CRYPT()` no puede usarse.
- `have_csv`: YES si mysqld soporta tablas ARCHIVE, NO si no.
- `have_dynamic_loading`: YES si mysqld soporta la carga dinámica de extensiones, NO si no.

- `have_geometry`: YES si el servidor soporta tipos de datos espaciales, NO si no.
- `have_innodb`: YES si mysqld soporta InnoDB . Esta variable está en desuso y se elimina en MySQL 5.6. Use `SHOW ENGINES` lugar.
- `have_openssl`: Esta variable es un alias para have_ssl.
- `have_partitioning`: YES si mysqld soporta particiones.
- `have_profiling`: YES si la capacidad declaración de perfiles está presente, NO si no.
- `have_query_cache`: YES si mysqld soporta la cache de consultas, NO si no.
- `have_rtree_keys`: YES si mysqld soporta conexiones SSL, NO . si no DISABLED indica que el servidor fue compilado con soporte SSL, pero sin embargo no se ha iniciado con los correspondientes --ssl- xxx opciones.
- `have_symlink`: YES si el soporte para enlaces simbólicos está activado, NO si no. Esto es necesario en Unix para soporte de opciones de la tabla `DATA DIRECTORY` e `INDEX DIRECTORY`, y en Windows para soporte de enlaces simbólicos del directorio de datos. Si el servidor se ha iniciado con la opcion --skip-symbolic-links opción, el valor está `DISABLED`.

21.2 Configuración

El fichero de configuración de mysql es my.cnf
Para editarlo: `sudo kate /etc/mysql/my.cnf`

Dentro de dicho archivo encontramos distintas opciones. Resumidas, tenemos lo siguiente:

- [**mysqld**] – Demonio / Servicio de mysql. Tiene varios opciones:
  - **basedir**: Directorio de instalación
  - **datadir**: Directorio de datos
  - **tmpdir**: Directorio temporal
  - **lc-messages-dir:** Directorio de mensajes de error
  - **Query-Cache:** permite almacenar el resultado de las query SELECT en una cache
    - **query_cache_type**: puede tener el valor ON, OFF y DEMAND. Los dos primeros habilitan o deshabilitan la caché, mientras que el último cacheará una query siempre y cuando lleve el modificador SQL_CACHE
    - **query_cache_size:** es el tamaño de la caché. Debe ser un valor múltiplo de 1024 bytes
    - **query_cache_min_res_unit:** MySQL guarda las cachés en la memoria en pequeños bloques, como si de un sistema de ficheros se tratase. En un principio no sabe realmente el tamaño que va a tener el resultado de una query, por lo que según va enviando las filas al cliente, va cogiendo bloques. Los bloques tendrán como mínimo el tamaño aquí indicado. Este valor es importante para evitar la fragmentación de la memoria y así aprovecharla lo mejor posible
    - **query_cache_limit:** si un resultado supera el tamaño aquí indicado, no se cacheará. Pero recordad lo comentado en el punto anterior, MySQL no sabe a priori cuando ocupará, por lo que igualmente irá reservando bloques hasta llegar al query_cache_limit, momento en el cual los bloques escritos se liberarán de nuevo
  - general_log_file: El archivo de registro de consultas general.
  - general_log: habilita el registro de consultas general si está a TRUE (1)
- MyISAM (Sección de tablas MyISAM)
  - max_connections: Máximas conexiones al servidor
  - table_cache: cache de las tablas MyISAM
  - thread_concurrency: hilos concurrentes

Mas información:
http://dev.mysql.com/doc/refman/5.1/en/option-files.html

21.3 Seguridad

Desde el punto de vista administrativo existen distintos puntos de revisión de la seguridad de una base de datos MySQL, fundamentalmente:

- Asignación de privilegios
- Fichero de configuración
- Cuentas de usuario
- Acceso remoto

Una de las primeras acciones de revisión se debe centrar en la revisión de las cuentas por defecto, en este caso root. Para ello se puede comprobar si existe la cuenta y contiene contraseña en blanco:

```
USE mysql;
SELECT User, Host, password
FROM user
WHERE User='root';
```

El acceso remoto (puerto 3306) a la base de datos debe ser monitorizado y controlado. Es muy común que el servidor de aplicaciones resida en la misma máquina que la base de datos, por lo que se podría limitar las conexiones a la base de datos desde fuera de localhost. Para ello se puede configurar esta opción en el fichero my.cnf añadiendo:

```
MYSQLD_OPTIONS="-skip-networking"
```

Adicionalmente existen algunos parámetros de configuración o arranque de la base de datos interesantes desde el punto de vista de la seguridad:

- **skip-grant-tables:** Arrancar el sistema con esta parámetro supone una grave brecha de seguridad, ya que supone que cualquier usuario tiene privilegios para hacer cualquier cosa sobre la base de datos. En algún caso puede ser útil para recuperar la contraseña de root.
- **safe-show-database:** Permite mostrar solamente aquellas bases de datos sobre las que un usuario tiene algún tipo de privilegio
- **safe-user-create:** permite determinar si un usuario puede crear nuevos usuarios siempre y cuando no tenga privilegios de INSERT sobre la tabla mysql.user

Finalmente, y partiendo de la base de asignación del mínimo privilegio necesario, uno de los aspectos de seguridad a revisar son los privilegios asignados en la base de datos.

Emperezamos por ver los prinvilegios en la tabla de usuarios:

```
SELECT User, Host, password FROM user
WHERE Select_priv  = 'Y' OR Insert_priv = 'Y';
SELECT User, Host, password FROM user
WHERE Update_priv = 'Y' OR Delete_priv  = 'Y';
SELECT User, Host, password FROM user
WHERE Create_priv = 'Y' OR Drop_priv    = 'Y';
SELECT User, Host, password FROM user
WHERE Reload_priv = 'Y' OR Shutdown_priv = 'Y';

SELECT User, Host, password FROM user
WHERE Process_priv = 'Y' OR File_priv    = 'Y';
SELECT User, Host, password FROM user
WHERE Grant_priv   = 'Y' OR References_priv = 'Y';
SELECT User, Host, password FROM user
WHERE Index_priv = 'Y' OR Alter_priv = 'Y';
```

Ahora vemos los privilegios para el host:

```
SELECT Host, Db FROM host WHERE Select_priv  = 'Y' or Insert_priv = 'Y';
SELECT Host, Db FROM host WHERE Create_priv = 'Y' or Drop_priv = 'Y';
SELECT Host, Db FROM host WHERE Index_priv = 'Y' or Alter_priv = 'Y';
SELECT Host, Db FROM host WHERE Grant_priv = 'Y' or References_priv ='Y';
SELECT Host, Db FROM host WHERE Update_priv = 'Y' or Delete_priv  = 'Y';
```

Por último comprobamos la tabla db (bases de datos):

```
SELECT Host, Db, User FROM db
WHERE Select_priv = 'Y' or Insert_priv = 'Y';
SELECT Host, Db, User FROM db
WHERE Grant_priv   = 'Y' or References_priv = 'Y';
SELECT Host, Db, User FROM db
WHERE Update_priv = 'Y' or Delete_priv  = 'Y';
SELECT Host, Db, User FROM db
WHERE Create_priv = 'Y' or Drop_priv    = 'Y';
SELECT Host, Db, User FROM db
WHERE Index_priv = 'Y' or Alter_priv = 'Y';
```

21.4 Logs

Ya hemos visto las variables de sesión que habilitan el registro.

- `general_log`: Si el registro general de consultas está habilitado. El valor puede ser 0 (o OFF) para desactivar el registro o 1 (o ON) para permitir el registro. El destino de la salida de registro es controlado por la variable de sistema log_output.
  NOTA: Desde MySQL 5 se puede activar el log y el archivo del log “en caliente”, es decir, desde el pripio cliente como veremos. Antes debía cambiarse my.cnf y reiniciar.
- `general_log_file`: El nombre del archivo de consulta de registro general. El valor por defecto es host_name .log , pero el valor inicial se puede cambiar con el --general_log_file opción.

```
SELECT @@general_log_file;
+-----------------------------------+
| @@general_log_file                |
+-----------------------------------+
| /opt/lampp/var/mysql/ivan-htpc.log |
+-----------------------------------+
```

Veamos un ejemplo para ver el estado, activar y ver el log general:

```
SELECT @@general_log;
SELECT @@general_log_file;
SET GLOBAL general_log=1;
```

Hacemos una prueba:

```
USE world;
SELECT Name FROM City WHERE CountryCode = 'ESP';
```

Para ver el log abrimos OTRO terminal:

```
shell> sudo kate /opt/lampp/var/mysql/ivan-htpc.log
```

```
/opt/lampp/sbin/mysqld, Version: 5.5.27 (Source distribution). started with:
Tcp port: 3306  Unix socket: /opt/lampp/var/mysql/mysql.sock
Time                 Id Command    Argument
121213 14:20:40     1 Query      SET GLOBAL general_log=1
121213 14:22:16     1 Query      SELECT DATABASE()
                    1 Init DB    world
121213 14:22:20     1 Query      SELECT Name FROM City WHERE CountryCode= 'ESP'
```

21.5 Copias de Seguridad

Ya hemos tratado las Exportaciones e Importaciones en el tema 17. Ahora vamos a ver el proceso de realización de copias de Seguridad utilizando las herramientas ya vistas.

Ningún sistema es perfecto ni está a salvo de errores humanos, cortes en el suministro de la corriente eléctrica, desperfectos en el hardware o errores de software; así que una labor más que recomendable del administrador del servidor de bases de datos es realizar copias de seguridad y diseñar un plan de contingencia. Se deben hacer ensayos del plan para asegurar su buen funcionamiento y, si se descubren anomalías, realizar los ajustes necesarios.

No existe una receta universal que nos indique cómo llevar nuestras copias de seguridad de datos. Cada administrador debe diseñar el de su sistema de acuerdo a sus necesidades, recursos, riesgos y el valor de la información.

MySQL ofrece varias alternativas de copia de seguridad de la información. La primera que podemos mencionar consiste simplemente en copiar los archivos de datos. Efectivamente, es una opción válida y sencilla.

En primera instancia son necesarios dos requisitos para llevarla a cabo:

- Conocer la ubicación y estructura del directorio de datos.
- Parar el servicio MySQL mientras se realiza la copia.

En cuanto a la ubicación y estructura del directorio, recordemos que la distribución de MySQL ubica el directorio de datos en /usr/local/mysql/var, las distribuciones GNU/Linux basadas en paquetes como DEB o RPM ubican, por lo general, los datos en /var/lib/mysql.

Pasos:

1. Conocer el sitio donde están los archivos de datos:

```
SHOW VARIABLES LIKE 'datadir';
+---------------+----------------------+
| Variable_name | Value                |
+---------------+----------------------+
| datadir       | /opt/lampp/var/mysql/ |
+---------------+----------------------+
```

2. Paramos el servidor con el programa cliente mysqladmin:

```
mysqladmin -u root -p shutdown
```

3. Copiamos recursivamente a un directorio con los permisos pertinentes:

```
sudo cp -r /opt/lampp/var/mysql/ /home/ivan-htpc/mysql
```

4. Podemos comprimir directamente a un directorio:

```
sudo tar czf /home/ivan-htpc/mysql/mysql-backup.tar.gz
/opt/lampp/var/mysql/
```

5. Podemos copiar únicamente una base de datos:

```
sudo cp -r /opt/lampp/var/mysql/test /home/ivan-htpc/mysql/test
```

6. O bien una tabla determinada:

```
sudo cp -r /opt/lampp/var/mysql/test/usuarios.* /home/ivan-htpc/mysql/test
```

El problema de este mecanismo es que debemos detener el servicio de bases de datos mientras realizamos el respaldo.

21.5.1 MySQLHotCopy

Un mecanismo que permite realizar la copia de los archivos del servidor sin necesidad de detener el servicio es el script 'mysqlhotcopy'. El script está escrito en Perl y bloquea las tablas mientras realiza el respaldo para evitar su modificación. Se usa de la siguiente manera:

```
mysqlhotcopy test /home/ivan-htpc/mysql/test1
```

NOTA: Es posible que no esté instalado. Para hacerlo debemos usar esta sentencia:

```
sudo apt-get install mysql-server-5.5
```

Este método no funciona para tablas con el mecanismo de almacenamiento tipo InnoDB.

21.5.2 MySQLDump

Las dos opciones anteriores representan copias binarias de la base de datos. El comando mysqldump, en cambio, realiza un volcado de las bases de datos pero traduciéndolas a SQL; es decir, entrega un archivo de texto con todos los comandos necesarios para volver a reconstruir las bases de datos, sus tablas y sus datos. Es el método más útil para copiar o distribuir una base de datos que deberá almacenarse en otros servidores.

Ya lo hemos visto previamente. Pongamos otro ejemplo:

```
mysqldump test > test.sql
```

21.6 Chequeo y reparación de tablas

21.6.1 La sentencia CHECK TABLE

En determinadas circunstancias de uso muy frecuente, como la inserción y borrado masivos de datos, coincidiendo con bloqueos del sistema o llenado del espacio en disco u otras circunstancias, es posible que una tabla o algunos de sus índices se corrompan.

Podemos consultar el estado de integridad de una tabla con el comando `CHECK TABLE`, que realiza algunas verificaciones sobre la tabla en busca de errores y nos entrega un informe con las varias columnas de información. Un ejemplo:

```
USE world;
CHECK TABLE Ciudades;
+----------------+-------+----------+----------+
| Table          | Op    | Msg_type | Msg_text |
+----------------+-------+----------+----------+
| world.Ciudades | check | status   | OK       |
+----------------+-------+----------+----------+
```

La información aparece con las siguientes columnas:

- `Columna Op` describe la operación que se realiza sobre la tabla. Para el comando check table esta columna siempre tiene el valor check porque ésa es la operación que se realiza.
- `Columna Msg_type` pude contener uno de los valores status, error, info, o warning.
- `Columna Msg_text` es el texto que reporta de alguna situación encontrada en la tabla.

Es posible que la información entregada incluya varias filas con diversos mensajes, pero el último mensaje siempre debe ser el mensaje OK de tipo status.
En otras ocasiones check table no realizará la verificación de tabla, en su lugar entregará como resultado el mensaje Table is already up to date, que significa que el gestor de la tabla indica que no hay necesidad de revisarla.

MySQL no permite realizar consultas sobre una tabla dañada y enviará un mensaje de error sin desplegar resultados parciales. Veamos un ejemplo VIRTUAL:

```
select * from precios;
ERROR 1016: No puedo abrir archivo: 'precios.MYD'. (Error: 145)
```

Para obtener información del significado del error 145, usaremos la utilidad en línea de comandos perror (que se instala junto al servidor de mySQL):

```
$> perror 145
145 = Table was marked as crashed and should be repaired
```

Después de un mensaje como el anterior, es el momento de realizar una verificación de la integridad de la tabla para obtener el reporte.

```
check table precios extended;
+-------------+------+----------+----------------------------------+
| Table | Op | Msg_type | Msg_text |
+-------------+------+----------+----------------------------------+
|demo.precios |check | error |Size of datafile is:450 Should be:452 |
|demo.precios |check | error | Corrupt |
+-------------+------+----------+----------------------------------+
```

En este caso localizamos dos errores en la tabla. La opción extended es uno de los cinco niveles de comprobación que se pueden solicitar para verificar una tabla.

Tipos de verificación

Tipo	**Significado**
quick	No revisa las filas en busca de referencias incorrectas.
fast	Solamente verifica las tablas que no fueron cerradas adecuadamente.
changed	Verifica sólo las tablas modificadas desde la última verificación o que no se han cerrado apropiadamente.
medium	Revisa las filas para verificar que los ligados borrados son correctos, verifica las sumas de comprobación de las filas.
extended	Realiza una búsqueda completa en todas las claves de cada columna. Garantiza el 100% de la integridad de la tabla.

La sentencia repair table realiza la reparación de tablas tipo MyISAM corruptas:

```
repair table precios;
+-------------+------+---------+----------------------------------+
| Table | Op | Msg_type         | Msg_text |
+-------------+------+---------+----------------------------------+
|demo.precios |repair| info     | Wrong bytesec: 0-17-1 at 168;Skipped |
|demo.precios |repair| warning  | Number of rows changed from 20 to 7  |
|demo.precios |repair| status   | OK                                   |
+-------------+-------+--------+----------------------------------+
```

El segundo mensaje informa de la pérdida de 13 filas durante el proceso de reparación.
Esto significa, como es natural, que el comando REPAIR TABLE es útil sólo en casos de extrema necesidad, ya que no garantiza la recuperación total de la información.

En la práctica, siempre es mejor realizar la restauración de la información utilizando las copias de seguridad. En caso de desastre, se debe conocer el motivo que origina la corrupción de las tablas y tomar las medidas adecuadas para evitarlo. En lo que respecta a la estabilidad de MySQL, se puede confiar en que probablemente nunca será necesario utilizar el comando REPAIR TABLE.

21.6.2 La sentencia OPTIMIZE TABLE

MySQL tiene capacidad para corregir los errores propios de fragmentación que se generan con el uso extendido de la base de datos. Para ello podemos usar OPTIMIZE TABLE. Su sintaxis:

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE nombre_tabla1
[,nombre_tabla2,...]
```

Con `NO_WRITE_TO_BINLOG` se prescinde de escribir en el log binario del cliente.
`OPTIMIZE TABLE` puede usarse en tablas MyISAM, BDB e InnoDB.

¿Cuando usar OPTIMIZE TABLE?

- Si se han realizado muchos borrados o eliminaciones de filas (DELETE FROM)
- Si se han realizado muchas modificaciones en tablas con filas de longitud variable (tablas con columnas de tipo VARCHAR, VARBINARY, BLOB o TEXT)

¿Porque usar OPTIMIZE TABLE?
Las filas borradas son mantenidas en una lista enlazada y las operaciones INSERT subsecuentes reutilizan las posiciones de las viejas filas. Se puede usar OPTIMIZE TABLE para recuperar el espacio no utilizado y para defragmentar el archivo de datos.

Un ejemplo:

```
USE world;
OPTIMIZE TABLE Ciudades;
```

```
+----------------+----------+----------+-------------------------------------------------------------------+
| Table          | Op       | Msg_type | Msg_text                                                          |
+----------------+----------+----------+-------------------------------------------------------------------+
| world.Ciudades | optimize | note     | Table does not support optimize, doing recreate + analyze instead |
| world.Ciudades | optimize | status   | OK                                                                |
+----------------+----------+----------+-------------------------------------------------------------------+
```

El mensaje Table does not support optimice no tiene mayor importancia.
Una buena manera de cerciorarnos que la tabla se optimiza bien es añadirle el tipo de Motor:

```
ALTER TABLE Ciudades TYPE='InnoDB';
```

Mas información:
http://dev.mysql.com/doc/refman/5.1/en/optimize-table.html

21.6.3 myisamchk

Para optimizar tablas MyISAM se usa el programa cliente myisamchk.
Su sintaxis es: `myisamchk [opciones] nombre_tabla`

Mas información:
http://dev.mysql.com/doc/refman/5.0/es/myisamchk-syntax.html

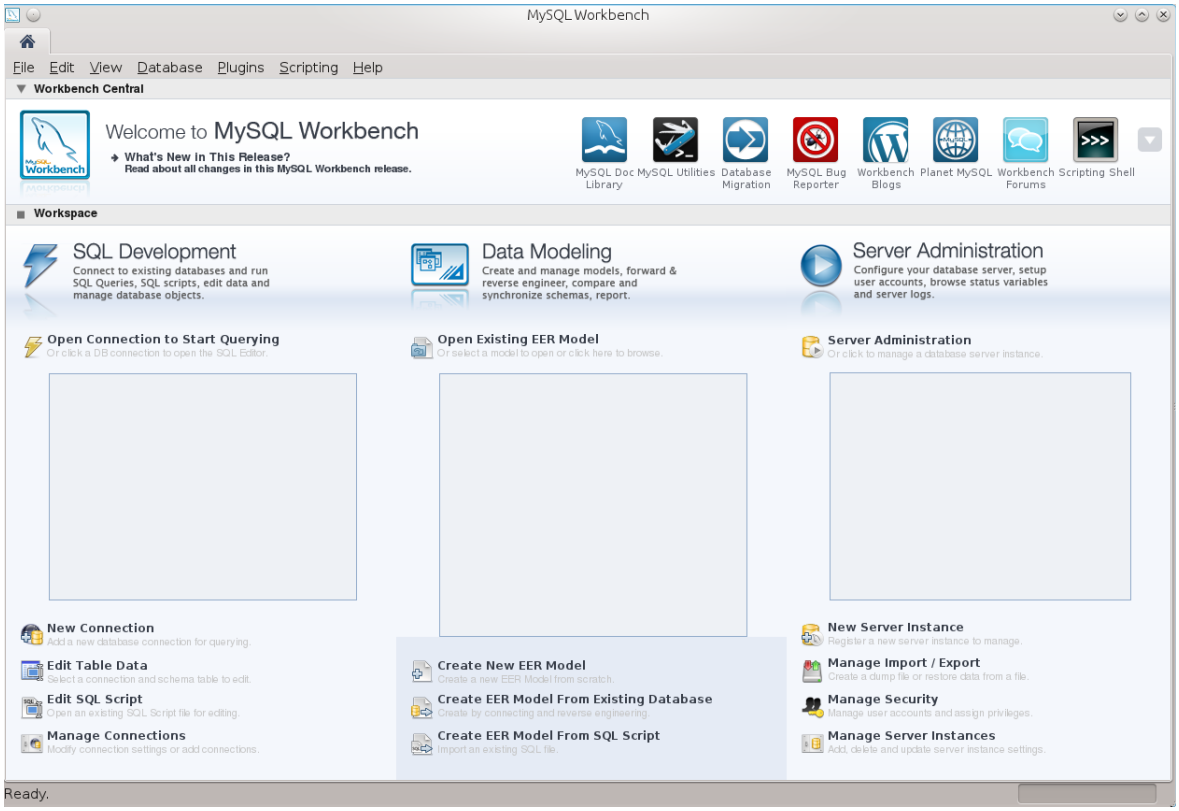
21.7 Herramientas Gráficas MySQL

Hasta hace poco eran:

- MySQL Administrator
- MySQL Query Browser
- MySQL Migration Toolkit
- MySQL System Tray Monitor

Se descargan aquí:
http://dev.mysql.com/downloads/gui-tools/5.0.html

De todos modos han sido sustituidos por mySQL workbench
http://dev.mysql.com/downloads/workbench/
Seleccionamos el sistema operativo (por ejemplo Ubuntu – 32 Bits, DEB) y lo instalamos.



Para el trabajo normal con sentencias SQL usaremos la parte izquierda.
Pulsamos en Open Connection to Start Querying.
Ponemos usuario y clave (root) y le damos a OK.
Aparecerá la pantalla de edición. Solo tenemos que escribir las sentencias y pulsar CTRL+ENTER.

Mas información:
http://mysql-espanol.org/2010/08/20/mysql-workbench/

22 DISEÑO DE BASES DE DATOS

IMPORTANTE: Este tema ha sido tomado de la asignatura de BBDD de la UOC

22.1 Introducción al diseño de bases de datos

Para poder emplear la tecnología de las bases de datos relacionales debemos decidir qué relaciones debe tener una base de datos determinada o qué atributos deben presentar las relaciones, qué claves primarias y qué claves foráneas se deben declarar, etc. La tarea de tomar este conjunto de decisiones recibe el nombre de diseñar la base de datos.

Una base de datos sirve para almacenar la información que se utiliza en un sistema de información determinado. Las necesidades y los requisitos de los futuros usuarios del sistema de información se deben tener en cuenta para poder tomar adecuadamente las decisiones anteriores.

Resumiendo: el diseño de una base de datos consiste en la obtención de una representación informática concreta a partir del estudio del mundo real.

22.1.1 Etapas del diseño de bases de datos.

Es esencial descomponer el proceso del diseño en varias etapas; en cada una se obtiene un resultado intermedio que sirve de punto de partida de la etapa siguiente, y en la última etapa se obtiene el resultado deseado. De este modo no hace falta resolver de golpe toda la problemática que plantea el diseño, sino que en cada etapa se afronta un solo tipo de subproblema. Así se divide el problema y, al mismo tiempo, se simplifica el proceso.

Descompondremos el diseño de bases de datos en tres etapas:

1. **Etapa del diseño conceptual:** en esta etapa se obtiene una estructura de la información de la futura BD independientemente de la tecnología que hay que emplear. No se tiene en cuenta todavía qué tipo de base de datos se utilizará –relacional, orientada a objetos, jerárquica, etc.–; en consecuencia, tampoco se tiene en cuenta con qué SGBD ni con qué lenguaje concreto se implementará la base de datos. Así pues, la etapa del diseño conceptual nos permite concentrarnos únicamente en la problemática de la estructuración de la información, sin tener que preocuparnos al mismo tiempo de resolver cuestiones tecnológicas.

El resultado de la etapa del diseño conceptual se expresa mediante algún modelo de datos de alto nivel. Uno de los más empleados es el modelo entidad-interrelación (entity-relationship), que abreviaremos con la sigla ER.

2. **Etapa del diseño lógico:** en esta etapa se parte del resultado del diseño conceptual, que se transforma de forma que se adapte a la tecnología que se debe emplear. Más concretamente, es preciso que se ajuste al modelo del SGBD con el que se desea implementar la base de datos. Por ejemplo, si se trata de un SGBD relacional, esta etapa obtendrá un conjunto de relaciones con sus atributos, claves primarias y claves foráneas.

3. **Etapa del diseño físico:** en esta etapa se transforma la estructura obtenida en la etapa del diseño lógico, con el objetivo de conseguir una mayor eficiencia; además, se completa con aspectos de implementación física que dependerán del SGBD.

Por ejemplo, si se trata de una base de datos relacional, la transformación de la estructura puede consistir en lo siguiente: tener almacenada alguna relación que sea la combinación de varias relaciones que se han obtenido en la etapa del diseño lógico, partir una relación en varias, añadir algún atributo calculable a una relación, etc. Los aspectos de implementación física que hay que completar consisten normalmente en la elección de estructuras físicas de implementación de las relaciones, la selección del tamaño de las memorias intermedias (buffers) o de las páginas, etc.

22.2 Diseño conceptual: el modelo ER.

El modelo ER es uno de los enfoques de modelización de datos que más se utiliza actualmente por su simplicidad y legibilidad. Su legibilidad se ve favorecida porque proporciona una notación diagramática muy comprensiva. Es una herramienta útil tanto para ayudar al diseñador a reflejar en un modelo conceptual los requisitos del mundo real de interés como para comunicarse con el usuario final sobre el modelo conceptual obtenido y, de este modo, poder verificar si satisface sus requisitos.

El modelo ER resulta fácil de aprender y de utilizar en la mayoría de las aplicaciones. Además, existen herramientas informáticas de ayuda al diseño (herramientas CASE, Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora) que utilizan alguna variante del modelo ER para hacer el diseño de los datos. Nosotros usaremos MySQL WorkBench para este propósito.

El nombre completo del modelo ER es entity-relationship, y proviene del hecho de que los principales elementos que incluye son las entidades y las interrelaciones (entities y relationships). Traduciremos este nombre por 'entidad-interrelación'. (aunque también se usa mucho Entidad-Relación, aunque este último debe emplearse con cuidado para no confundirlo con con las Relaciones del modelo, que es un concepto similar, pero diferente).

El origen del modelo ER se encuentra en trabajos efectuados por Peter Chen en 1976. Posteriormente, muchos otros autores han descrito variantes y/o extensiones de este modelo. Así pues, en la literatura se encuentran muchas formas diferentes del modelo ER que pueden variar simplemente en la notación diagramática o en algunos de los conceptos en que se basan para modelizar los datos.

Cuando se quiere utilizar el modelo ER para comunicarse con el usuario, es recomendable emplear una variante del modelo que incluya sólo sus elementos más simples –entidades, atributos e interrelaciones– y, tal vez, algunas construcciones adicionales, como por ejemplo entidades débiles y dependencias de existencia. Éstos eran los elementos incluidos en el modelo original propuesto por Chen. En cambio, para llevar a cabo la tarea de modelizar propiamente dicha, suele ser útil usar un modelo ER más completo que incluya construcciones más avanzadas que extienden el modelo original.

Según la noción de modelo de datos, este tiene en cuenta tres aspectos de los datos:

- La estructura
- La manipulación
- La integridad.

Sin embargo, el modelo ER habitualmente se utiliza para reflejar aspectos de la estructura de los datos y de su integridad, pero no de su manipulación.

22.2.1 Construcciones básicas

22.2.1.1 Entidades, atributos e interrelaciones

Por entidad entendemos un objeto del mundo real que podemos distinguir del resto de objetos y del que nos interesan algunas propiedades.

ENTIDADES

Ejemplos de entidad

Algunos ejemplos de entidad son un empleado, un producto o un despacho. También son entidades otros elementos del mundo real de interés, menos tangibles pero igualmente diferenciables del resto de objetos; por ejemplo, una asignatura impartida en una universidad, un préstamo bancario, un pedido de un cliente, etc.

ATRIBUTOS

Las propiedades de los objetos que nos interesan se denominan **atributos**.

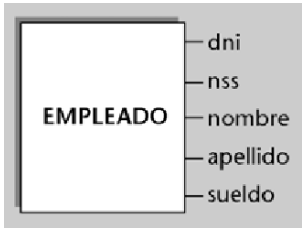
Ejemplos de atributo

Sobre una entidad empleado nos puede interesar, por ejemplo, tener registrados su DNI, su NSS, su nombre, su apellido y su sueldo como atributos.

El término entidad se utiliza tanto para denominar objetos individuales como para hacer referencia a conjuntos de objetos similares de los que nos interesan los mismos atributos; es decir, que, por ejemplo, se utiliza para designar tanto a un empleado concreto de una empresa como al conjunto de todos los empleados de la empresa. Más concretamente, el término entidad se puede referir a instancias u ocurrencias concretas (empleados concretos) o a tipos o clases de entidades (el conjunto de todos los empleados).

El **modelo ER** proporciona una notación diagramática para representar gráficamente las entidades y sus atributos:

- Las entidades se representan con un rectángulo. El nombre de la entidad se escribe en mayúsculas dentro del rectángulo.
- Los atributos se representan mediante su nombre en minúsculas unido con un guión al rectángulo de la entidad a la que pertenecen. Muchas veces, dado que hay muchos atributos para cada entidad, se listan todos aparte del diagrama para no complicarlo.



Cada uno de los atributos de una entidad toma valores de un cierto dominio o conjunto de valores. Los valores de los dominios deben ser **atómicos**; es decir, no deben poder ser descompuestos. Además, todos los atributos tienen que ser **univaluados**. Un atributo es univaluado si tiene un único valor para cada ocurrencia de una entidad.

Ejemplo de atributo univaluado

El atributo sueldo de la entidad empleado, por ejemplo, toma valores del dominio de los reales y únicamente toma un valor para cada empleado concreto; por lo tanto, ningún empleado puede tener más de un valor para el sueldo.

Una entidad debe ser distinguible del resto de objetos del mundo real. Esto hace que para toda entidad sea posible encontrar un conjunto de atributos que permitan identificarla. Este conjunto de atributos forma una clave de la entidad.

Ejemplo de clave

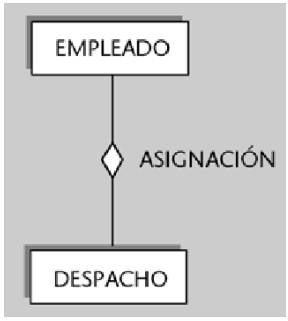
La entidad empleado tiene una clave que consta del atributo dni porque todos los empleados tienen números de DNI diferentes. Una determinada entidad puede tener más de una clave; es decir, puede tener varias claves candidatas.

Ejemplo de clave candidata
La entidad empleado tiene dos claves candidatas, la que está formada por el atributo dni y la que está constituida por el atributo nss, teniendo en cuenta que el NSS también será diferente para cada uno de los empleados.

El diseñador elige una clave primaria entre todas las claves candidatas. En la notación diagramática, la clave primaria se subraya para distinguirla del resto de las claves.

Ejemplo de clave primaria
En el caso de la entidad empleado, podemos elegir dni como clave primaria. En la figura del margen vemos que la clave primaria se subraya para distinguirla del resto.

INTERRELACIONES
Se define interrelación como una asociación entre entidades. Las interrelaciones se representan en los diagramas del modelo ER mediante un rombo. Junto al rombo se indica el nombre de la interrelación con letras mayúsculas.



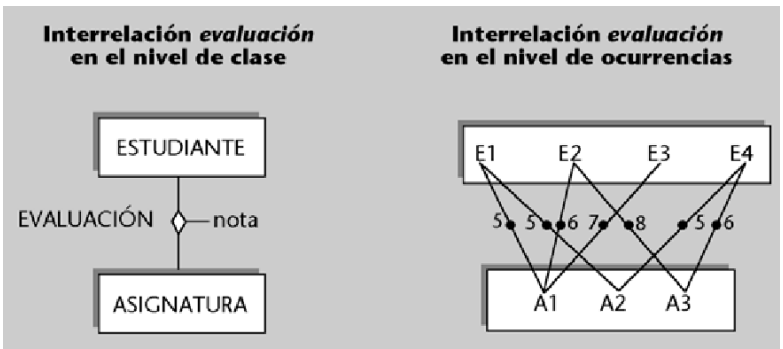
Ejemplo de interrelación
Consideremos una entidad empleado y una entidad despacho y supongamos que a los empleados se les asignan despachos donde trabajar. Entonces hay una interrelación entre la entidad empleado y la entidad despacho.

Esta interrelación, que podríamos denominar asignación, asocia a los empleados con los despachos donde trabajan. El término interrelación se puede utilizar tanto para denominar asociaciones concretas u ocurrencias de asociaciones como para designar conjuntos o clases de asociaciones similares.

Ejemplo
Una interrelación se aplica tanto a una asociación concreta entre el empleado de DNI '50.455.234' y el despacho 'Diagonal, 20' como a la asociación genérica entre la entidad empleado y la entidad despacho.

En ocasiones interesa reflejar algunas propiedades de las interrelaciones. Por este motivo, las interrelaciones pueden tener también atributos. Los atributos de las interrelaciones, igual que los de las entidades, tienen un cierto dominio, deben tomar valores atómicos y deben ser univaluados.

Los atributos de las interrelaciones se representan mediante su nombre en minúsculas unido con un guión al rombo de la interrelación a la que pertenecen.



Ejemplo de atributo de una interrelación
Observemos la entidad estudiante y la entidad asignatura que se muestran en la imagen de la izquierda.

Entre estas dos entidades se establece la interrelación evaluación para indicar de qué asignaturas han sido evaluados los estudiantes. Esta interrelación tiene el atributo nota, que sirve para especificar qué nota han obtenido los estudiantes de las asignaturas evaluadas.

Conviene observar que el atributo nota deber ser forzosamente un atributo de la interrelación evaluación, y que no sería correcto considerarlo un atributo de la entidad estudiante o un atributo de la entidad asignatura. Lo explicaremos analizando las ocurrencias de la interrelación evaluación que se muestran en la figura anterior.

Si nota se considerase un atributo de estudiante, entonces para el estudiante 'E1' de la figura necesitaríamos dos valores del atributo, uno para cada asignatura que tiene el estudiante; por lo tanto, no sería univaluado. De forma similar, si nota fuese atributo de asignatura tampoco podría ser univaluado porque, por ejemplo, la asignatura 'A1' requeriría tres valores de nota, una para cada estudiante que se ha matriculado en ella. Podemos concluir que el atributo nota está relacionado al mismo tiempo con una asignatura y con un estudiante que la cursa y que, por ello, debe ser un atributo de la interrelación que asocia las dos entidades.

22.2.1.2 Grado de las interrelaciones

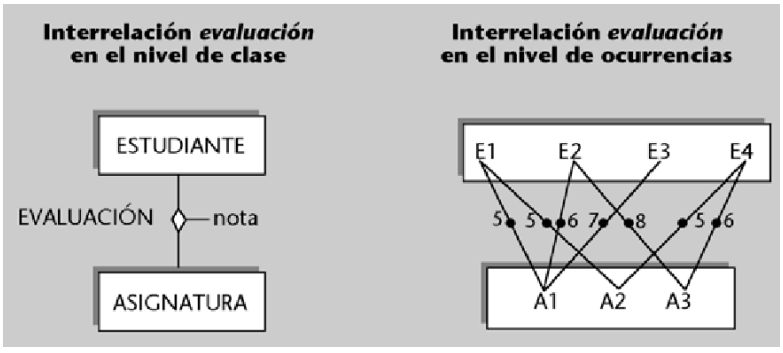
Una interrelación puede asociar dos o más entidades. El número de entidades que asocia una interrelación es el grado de la interrelación.

Interrelaciones de grado dos
Las interrelaciones evaluación y asignación de los ejemplos anteriores tienen grado dos:

- La interrelación **evaluación** asocia la entidad **estudiante** y la entidad **asignatura**; es decir, asocia dos entidades.
- De forma análoga, la interrelación **asignación** asocia **empleado** y **despacho**.

Las interrelaciones de grado dos se denominan también interrelaciones binarias. Todas las interrelaciones de grado mayor que dos se denominan, en conjunto, interrelaciones n-arias. Así pues, una interrelación n-aria puede tener grado tres y ser una interrelación ternaria, puede tener grado cuatro y ser una interrelación cuaternaria, etc.

A continuación presentaremos un ejemplo que nos ilustrará el hecho de que, en ocasiones, las interrelaciones binarias no nos permiten modelizar correctamente la realidad y es necesario utilizar interrelaciones de mayor grado.



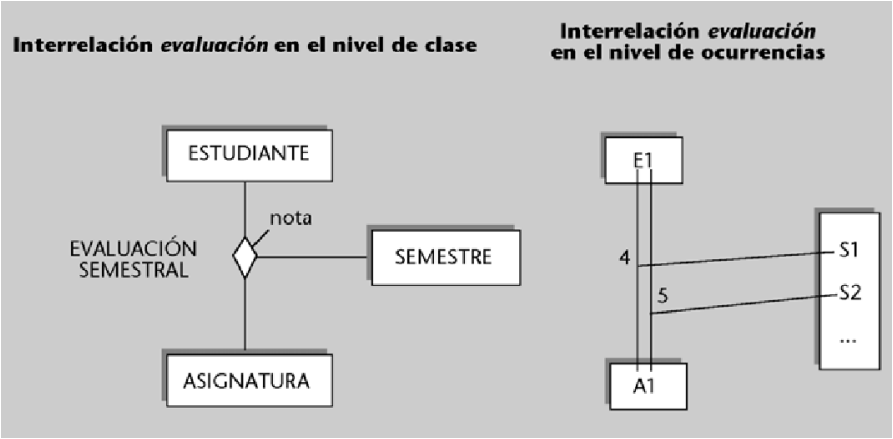
Consideremos la interrelación **evaluación** de la figura anterior, que tiene un **atributo nota**. Este atributo permite registrar la nota obtenida por cada estudiante en cada asignatura de la que ha sido evaluado. Una interrelación permite establecer una sola asociación entre unas entidades individuales determinadas.

En otras palabras, sólo se puede interrelacionar una vez al estudiante 'E1' con la asignatura 'A1' vía la interrelación evaluación. Observad que, si pudiese haber más de una interrelación entre el estudiante 'E1' y la asignatura 'A1', no podríamos distinguir estas diferentes ocurrencias de la interrelación.

Esta restricción hace que se registre una sola nota por estudiante y asignatura. Supongamos que deseamos registrar varias notas por cada asignatura y estudiante correspondientes a varios semestres en los que un mismo estudiante ha cursado una asignatura determinada (desgraciadamente, algunos estudiantes tienen que cursar una asignatura varias veces antes de aprobarla).

La interrelación anterior no nos permitiría reflejar este caso.

Sería necesario aumentar el grado de la interrelación, tal y como se muestra en la figura siguiente:

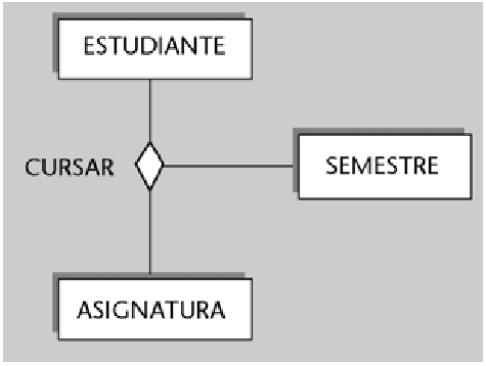


La interrelación ternaria evaluación-semestral asocia estudiantes, asignaturas y una tercera entidad que denominamos semestre. Su atributo nota nos permite reflejar todas las notas de una asignatura que tiene un estudiante correspondientes a diferentes semestres.

De hecho, lo que sucede en este caso es que, según los requisitos de los usuarios de esta BD, una nota pertenece al mismo tiempo a un estudiante, a una asignatura y a un semestre y, lógicamente, debe ser un atributo de una interrelación ternaria entre estas tres entidades.

Este ejemplo demuestra que una interrelación binaria puede no ser suficiente para satisfacer los requisitos de los usuarios, y puede ser necesario aplicar una interrelación de mayor grado. Conviene observar que esto también puede ocurrir en interrelaciones que no tienen atributos.

Ejemplo de interrelación ternaria sin atributos
Consideremos un caso en el que deseamos saber para cada estudiante qué asignaturas ha cursado cada semestre, a pesar de que no queremos registrar la nota que ha obtenido. Entonces aplicaríamos también una interrelación ternaria entre las entidades estudiante, asignatura y semestre que no tendría atributos, tal y como se muestra en la figura siguiente:



Hemos analizado casos en los que era necesario utilizar interrelaciones ternarias para poder modelizar correctamente ciertas situaciones de interés del mundo real. Es preciso remarcar que, de forma similar, a veces puede ser necesario utilizar interrelaciones de grado todavía mayor: cuaternarias, etc. En el subapartado siguiente analizaremos con detalle las interrelaciones binarias, y más adelante, las interrelaciones n-arias.

22.2.1.3 Interrelaciones binarias

Conectividad de las interrelaciones binarias
La conectividad de una interrelación expresa el tipo de correspondencia que se establece entre las ocurrencias de entidades asociadas con la interrelación. En el caso de las interrelaciones binarias, expresa el número de ocurrencias de una de las entidades con las que una ocurrencia de la otra entidad puede estar asociada según la interrelación.

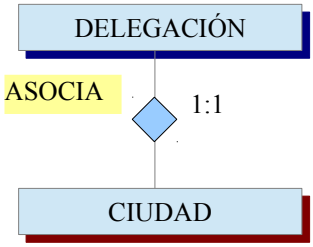
Una interrelación binaria entre dos entidades puede tener tres tipos de conectividad:

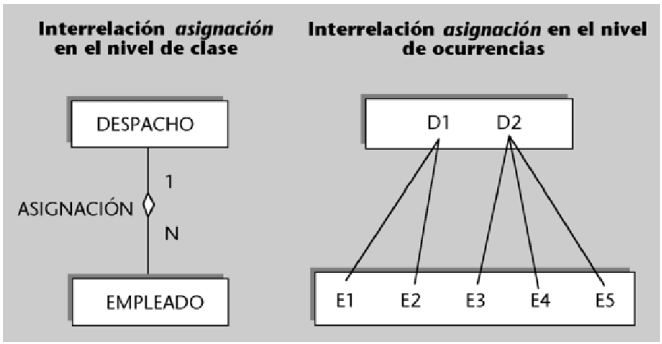
- **Conectividad uno a uno (1:1)**. La conectividad 1:1 se denota poniendo un 1 a lado y lado de la interrelación.
- **Conectividad uno a muchos (1:N)**. La conectividad 1:N se denota poniendo un 1 en un lado de la interrelación y una N en el otro.
- **Conectividad muchos a muchos: (M:N).** La conectividad M:N se denota poniendo una M en uno de los lados de la interrelación, y una N en el otro.

Ejemplos de conectividad en una interrelación binaria
A continuación analizaremos un ejemplo de cada una de las conectividades posibles para una interrelación binaria:

a) Conectividad 1:1

La interrelación anterior tiene conectividad 1:1. Esta interrelación asocia las delegaciones de una empresa con las ciudades donde están situadas. El hecho de que sea 1:1 indica que una ciudad tiene sólo una delegación, y que una delegación está situada en una única ciudad.



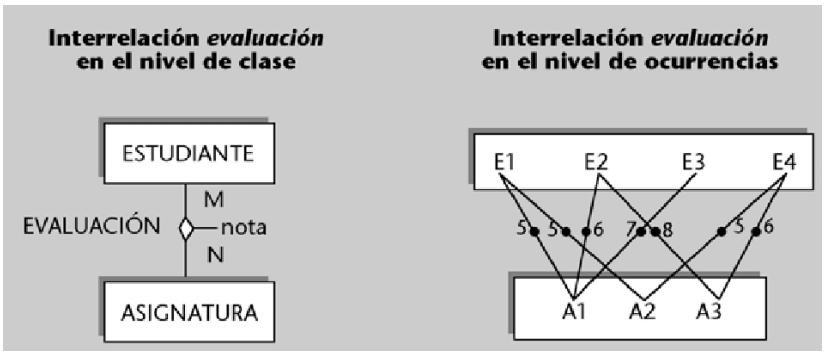


b) Conectividad 1:N

La interrelación asignación entre la entidad empleado y la entidad despacho tiene conectividad 1:N, y la N está en el lado de la entidad empleado. Esto significa que un empleado tiene un solo despacho asignado, pero que, en cambio, un despacho puede tener uno o más empleados asignados.

c) Conectividad M:N

Para analizar la conectividad M:N, consideramos la interrelación evaluación de la figura anterior. Nos indica que un estudiante puede ser evaluado de varias asignaturas y, al mismo tiempo, que una asignatura puede tener varios estudiantes por evaluar.

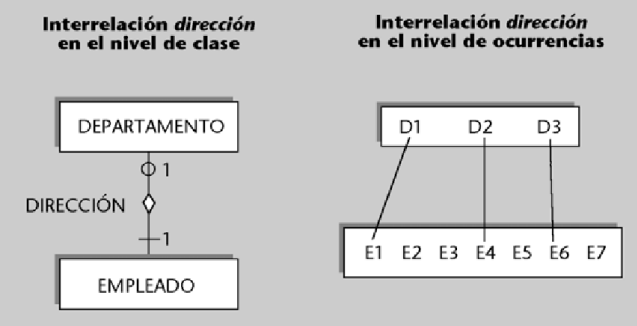


Es muy habitual que las interrelaciones binarias M:N y todas las n-arias tengan atributos. En cambio, las interrelaciones binarias 1:1 y 1:N no tienen por qué tenerlos. Siempre se pueden asignar estos atributos a la entidad del lado N, en el caso de las 1:N, y a cualquiera de las dos entidades interrelacionadas en el caso de las 1:1. Este cambio de situación del atributo se puede hacer porque no origina un atributo multivaluado.

Dependencias de existencia en las interrelaciones binarias
En algunos casos, una entidad individual sólo puede existir si hay como mínimo otra entidad individual asociada con ella mediante una interrelación binaria determinada. En estos casos, se dice que esta última entidad es una entidad obligatoria en la interrelación. Cuando esto no sucede, se dice que es una entidad opcional en la interrelación.

En el modelo ER, un círculo en la línea de conexión entre una entidad y unainterrelación indica que la entidad es opcional en la interrelación. La obligatoriedad de una entidad a una interrelación se indica con una línea perpendicular. Si no se consigna ni un círculo ni una línea perpendicular, se considera que la dependencia de existencia es desconocida.

Ejemplo de dependencias de existencia
La figura siguiente nos servirá para entender el significado práctico de la dependencia de existencia. La entidad empleado es obligatoria en la interrelación dirección. Esto indica que no puede existir un departamento que no tenga un empleado que actúa de director del departamento.



La entidad departamento, en cambio, es opcional en la interrelación dirección. Es posible que haya un empleado que no está interrelacionado con ningún departamento: puede haber –y es el caso más frecuente– empleados que no son directores de departamento.

Aplicaremos la dependencia de existencia en las interrelaciones binarias, pero no en las n-arias.

22.2.1.4 Ejemplo: base de datos de Casas de Colonias Escolares.

En este punto, y antes de continuar explicando construcciones más complejas del modelo ER, puede resultar muy ilustrativo ver la aplicación práctica de las construcciones que hemos estudiado hasta ahora. Por este motivo, analizaremos un caso práctico de diseño con el modelo ER que corresponde a una base de datos destinada a la gestión de las inscripciones en un conjunto de casas de colonias. El modelo ER de esta base de datos será bastante sencillo e incluirá sólo entidades, atributos e interrelaciones binarias (no incluirá interrelaciones n-arias ni otros tipos de estructuras).

La descripción siguiente explica con detalle los requisitos de los usuarios que hay que tener en cuenta al hacer el diseño conceptual de la futura base de datos:

a) Cada casa de colonias tiene un **nombre** que la identifica. Se desea saber de cada una:
- nombre
- La capacidad (el número de niños que se pueden alojar en cada una como máximo)
- La comarca donde está situada
- Ofertas de actividades que proporciona. Una casa puede ofrecer actividades como por ejemplo natación, esquí, remo, pintura, fotografía, música, etc.

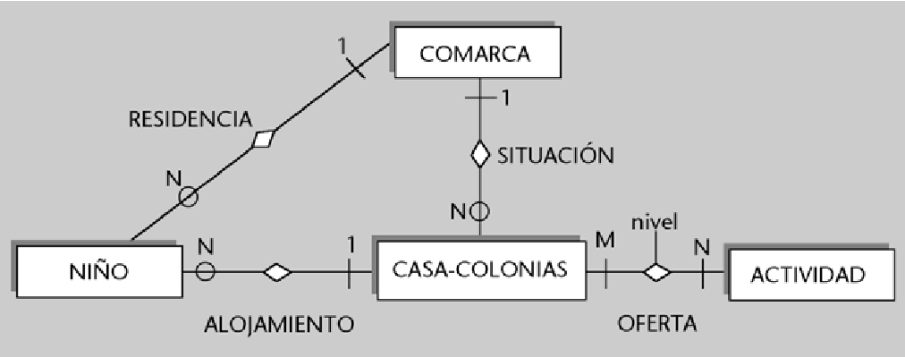
b) Es necesario tener en cuenta que en una casa de colonias se pueden practicar varias actividades (de hecho, cada casa debe ofrecer como mínimo una), y también puede ocurrir que una misma actividad se pueda llevar a cabo en varias casas. Sin embargo, toda actividad que se registre en la base de datos debe ser ofertada como mínimo en una de las casas.

c) Interesa tener una evaluación de las ofertas de actividades que proporcionan las casas. Se asigna una calificación numérica que indica el nivel de calidad que tiene cada una de las actividades ofertadas.
NOTA: Es posible por ejemplo, que una actividad como por ejemplo el esquí tenga una calificación de 10 en la oferta de la casa Grévol, y que la misma actividad tenga una calificación de 8 en la casa Ardilla.

d) Las casas de colonias alojan niños que se han inscrito para pasar en ellas unas pequeñas vacaciones. Se quiere tener constancia de los niños que se alojan en cada una de las casas en el momento actual. Se debe suponer que hay casas que están vacías (en las que no se aloja ningún niño) durante algunas temporadas.

e) De los niños que se alojan actualmente en alguna de las casas, interesa conocer un código que se les asigna para identificarlos, su nombre, su apellido, el número de teléfono de sus padres y su comarca de residencia.

f) De las comarcas donde hay casas o bien donde residen niños, se quiere tener registrados la superficie y el número de habitantes. Se debe considerar que puede haber comarcas donde no reside ninguno de los niños que se alojan en un momento determinado en las casas de colonias, y comarcas que no disponen de ninguna casa.



La figura siguiente muestra un diagrama ER que satisface los requisitos anteriores. Los atributos de las entidades no figuran en el diagrama y se listan aparte.

Los atributos de las entidades que figuran en el diagrama son los siguientes (las claves primarias están subrayadas):

CASA-COLONIAS
    nombre-casa, capacidad
ACTIVIDAD
    nombre-actividad
NIÑO
    código-niño, nombre, apellido, teléfono
COMARCA
    nombre-comarca, superficie, número-habitantes

A continuación comentamos los aspectos más relevantes de este modelo ER:
1) Una de las dificultades que en ocasiones se presenta durante la modelización conceptual es decidir si una información determinada debe ser una entidad o un atributo. En nuestro ejemplo, puede resultar difícil decidir si comarcase debe modelizar como una entidad o como un atributo.

A primera vista, podría parecer que comarca debe ser un atributo de la entidad casa-colonias para indicar dónde está situada una casa de colonias, y también un atributo de la entidad niño para indicar la residencia del niño.

Sin embargo, esta solución no sería adecuada, porque se quieren tener informaciones adicionales asociadas a la comarca: la superficie y el número de habitantes. Es preciso que comarca sea una entidad para poder reflejar estas informaciones adicionales como atributos de la entidad.

La entidad comarca tendrá que estar, evidentemente, interrelacionada con las entidades niño y casa-colonias. Observad que de este modo, además, se hace patente que las comarcas de residencia de los niños y las comarcas de situación de las casas son informaciones de un mismo tipo.

2) Otra decisión que hay que tomar es si el concepto actividad se debe modelizar como una entidad o como un atributo. Actividad no tiene informaciones adicionales asociadas; no tiene, por lo tanto, más atributos que los que forman la clave. Aun así, es necesario que actividad sea una entidad para que, mediante la interrelación oferta, se pueda indicar que una casa de colonias ofrece actividades.

Observad que las actividades ofertadas no se pueden expresar como un atributo de casa-colonias, porque una casa puede ofrecer muchas actividades y, en este caso, el atributo no podría tomar un valor único.

3) Otra elección difícil, que con frecuencia se presenta al diseñar un modelo ER, consiste en modelizar una información determinada como una entidad o como una interrelación. Por ejemplo, podríamos haber establecido que oferta, en lugar de ser una interrelación, fuese una entidad; lo habríamos hecho así:

La entidad oferta representada en la figura anterior tiene los atributos que presentamos a continuación:

OFERTA

nombre-casa, nombre-actividad, nivel

Esta solución no acaba de reflejar adecuadamente la realidad. Si analizamos la clave de oferta, podemos ver que se identifica con nombre-casa, que es la clave de la entidad casa-colonias, y con nombre-actividad, que es la clave de la entidad actividad. Esto nos debe hacer sospechar que oferta, de hecho, corresponde a una asociación o interrelación entre casas y actividades. En consecuencia, reflejaremos la realidad con más exactitud si modelizamos oferta como una interrelación entre estas entidades.

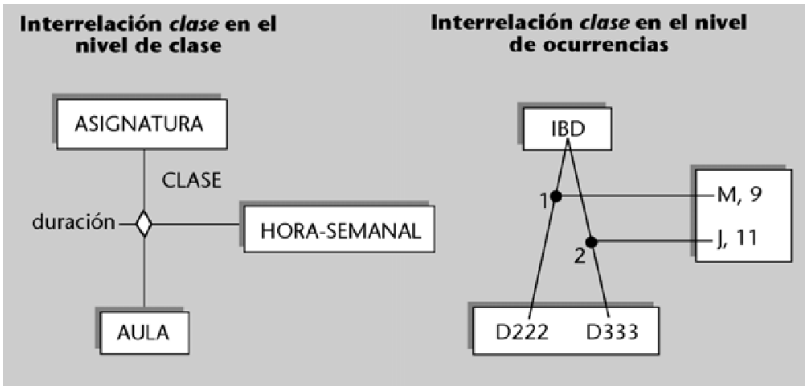
4) Finalmente, un aspecto que hay que cuidar durante el diseño conceptual es el de evitar las redundancias. Por ejemplo, si hubiésemos interrelacionado comarca con actividad para saber qué actividades se realizan en las casas de cada una de las comarcas, habríamos tenido información redundante. La interrelación oferta junto con la interrelación situación ya permiten saber, de forma indirecta, qué actividades se hacen en las comarcas.

22.2.1.5 Interrelaciones n-arias.

Las interrelaciones n-arias, igual que las binarias, pueden tener diferentes tipos de conectividad. En este subapartado analizaremos primero el caso particular de las interrelaciones ternarias y, a continuación, trataremos las conectividades de las interrelaciones n-arias en general.

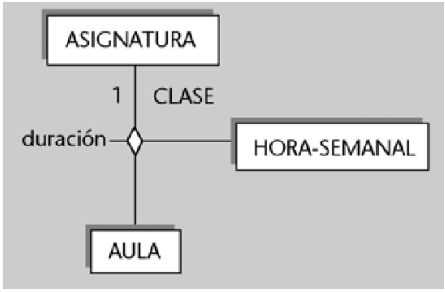
Conectividad de las interrelaciones ternarias
Cada una de las tres entidades asociadas con una interrelación ternaria puede estar conectada con conectividad “uno” o bien con conectividad “muchos”. En consecuencia, las interrelaciones ternarias pueden tener cuatro tipos de conectividad: M:N:P, M:M:1, N:1:1 y 1:1:1.
NOTA: La notación M, N y P se refiere a "Muchos"

Analizaremos cómo se decide cuál es la conectividad adecuada de una interrelación ternaria mediante el siguiente ejemplo. Consideremos una interrelación que denominamos clase y que asocia las entidades asignatura, aula y hora-semanal. Esta interrelación permite registrar clases presenciales. Una clase corresponde a una asignatura determinada, se imparte en un aula determinada y a una hora de la semana determinada.

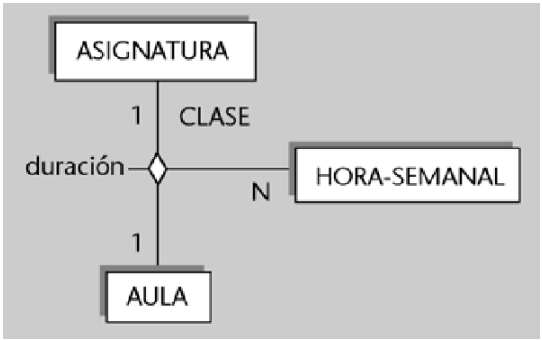


Por ejemplo, podemos registrar que se hace clase de la asignatura IBD en el aula D222 el martes a las 9, tal y como se muestra en la figura de la página siguiente. El atributo duración nos permite saber cuántas horas dura la clase.

Para decidir si el lado de la entidad asignatura se conecta con “uno” o con “muchos” , es necesario preguntarse si, dadas un aula y una hora-semanal, se puede hacer clase de sólo una o bien de muchas asignaturas en aquellas aula y hora. La respuesta sería que sólo se puede hacer clase de una asignatura en una misma aula y hora. Esto nos indica que asignatura se conecta con “uno”, tal y como reflejamos en la figura siguiente:



Como nos indica este ejemplo, para decidir cómo se debe conectar una de las entidades, es necesario preguntarse si, ya fijadas ocurrencias concretas de las otras dos, es posible conectar sólo “una” o bien “muchas” ocurrencias de la primera entidad.



Utilizaremos el mismo procedimiento para determinar cómo se conectan las otras dos entidades del ejemplo. Una vez fijadas una asignatura y un aula, es posible que se haga clase de aquella asignatura en aquella aula, en varias horas de la semana; entonces, hora-semana se conecta con “muchos”. Finalmente, la entidad aula se conecta con “uno”, teniendo en cuenta que, fijadas una asignatura y una hora de la semana, sólo se puede hacer una clase de aquella asignatura a aquella hora. La conectividad resultante, de este modo, es N:1:1.

Caso general: conectividad de las interrelaciones n-arias
Lo que hemos explicado sobre la conectividad para las interrelaciones ternarias es fácilmente generalizable a interrelaciones n-arias.

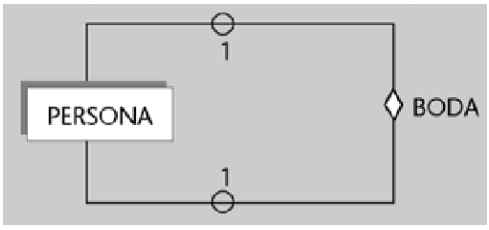
Para decidir si una de las entidades se conecta con “uno” o con “muchos”, es necesario preguntarse si, fijadas ocurrencias concretas de las otras n – 1 entidades, es posible conectar sólo una o bien muchas ocurrencias de la primera entidad:

- Si la respuesta es que sólo una, entonces se conecta con “uno”.
- Si la respuesta es que muchas, la entidad se conecta con “muchos”.

22.2.1.6 Interrelaciones recursivas

Una interrelación recursiva es una interrelación en la que alguna entidad está asociada más de una vez.

Ejemplo de interrelación recursiva
Si, para una entidad persona, queremos tener constancia de qué personas están actualmente casadas entre ellas, será necesario definir la siguiente interrelación, que asocia dos veces la entidad persona:



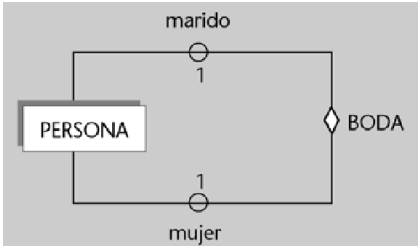
Una interrelación recursiva puede ser tanto binaria como n-aria:

1) Interrelación recursiva binaria: interrelación en la que las ocurrencias asocian dos instancias de la misma entidad. Este es el caso anterior. Las interrelaciones binarias recursivas pueden tener conectividad 1:1, 1:N o M:N, como todas las binarias. En esta interrelación también es posible expresar la dependencia de existencia igual que en el resto de las interrelaciones binarias.

Ejemplo de interrelación recursiva binaria
La interrelación boda tiene conectividad 1:1 porque un marido está casado con una sola mujer y una mujer está casada con un solo marido. También tiene un círculo en los dos lados (según la dependencia de existencia), porque puede haber personas que no estén casadas.

En una interrelación recursiva, puede interesar distinguir los diferentes papeles que una misma entidad tiene en la interrelación. Con este objetivo, se puede etiquetar cada línea de la interrelación con un rol. En las interrelaciones no recursivas normalmente no se especifica el rol; puesto que todas las entidades interrelacionadas son de clases diferentes, sus diferencias de rol se sobreentienden.

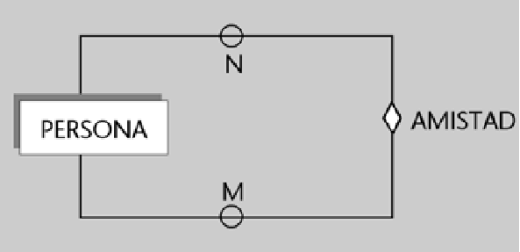
Roles diferentes
Una ocurrencia de la interrelación boda asocia a dos personas concretas. Para reflejar el papel diferente que tiene cada una de ellas en la interrelación, una de las personas tendrá el rol de marido y la otra tendrá el rol de mujer.



Algunas interrelaciones recursivas no presentan diferenciación de roles; entonces, las líneas de la interrelación no se etiquetan.

No-diferencia de roles

Consideremos una interrelación amistad que asocia a personas concretas que son amigas. A diferencia de lo que sucedía en la interrelación boda, donde una de las personas es el marido y la otra la mujer, en este caso no hay diferenciación de roles entre las dos personas interrelacionadas. A continuación se muestra esta interrelación.

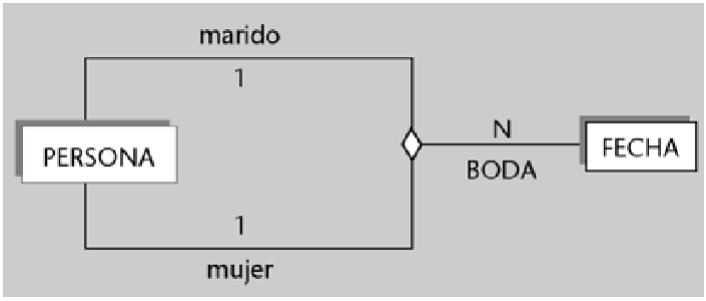


Observad que su conectividad es M:N, teniendo en cuenta que una persona puede tener muchos amigos y, al mismo tiempo, puede haber muchas personas que la consideran amiga.

2) Interrelación recursiva n-aria: interrelación recursiva en la que las ocurrencias asocian más de dos instancias.

Ejemplo de interrelación recursiva ternaria

Consideremos una interrelación que registra todas las bodas que se han producido a lo largo del tiempo entre un conjunto de personas determinado.



Esta interrelación permite tener constancia no sólo de las bodas vigentes, sino de todas las bodas realizadas en un cierto periodo de tiempo.

Esta interrelación es recursiva y ternaria. Una ocurrencia de la interrelación asocia a una persona que es el marido, a otra que es la mujer y la fecha de su boda. La conectividad es N:1:1. A los lados del marido y de la mujer les corresponde un 1, porque un marido o una mujer, en una fecha determinada, se casa con una sola persona. Al lado de la entidad fecha le corresponde una N, porque se podría dar el caso de que hubiese, en fechas diferentes, más de una boda entre las mismas personas. (como sucedió entre Liz Taylor y Richard Burton).

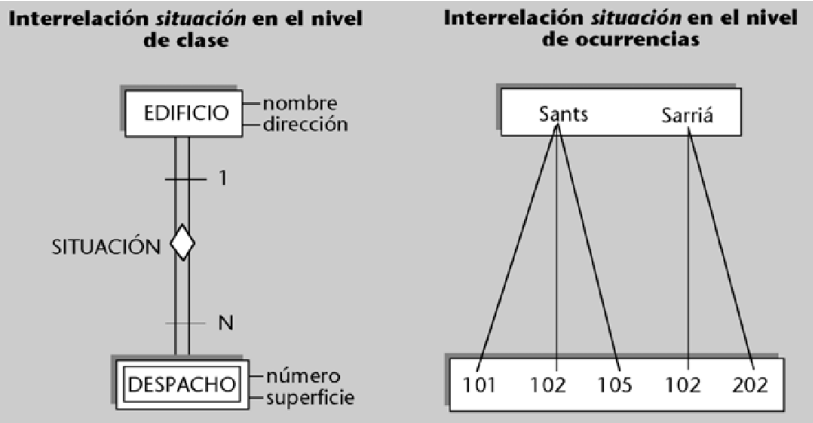
22.2.1.7 Entidades débiles.

Las entidades que hemos considerado hasta ahora tienen un conjunto de atributos que forman su claves primarias y que permiten identificarlas completamente. Estas entidades se denominan, de forma más específica, entidades fuertes. En este subapartado consideraremos otro tipo de entidades que denominaremos entidades débiles.

Una entidad débil es una entidad cuyos atributos no la identifican completamente, sino que sólo la identifican de forma parcial. Esta entidad debe participar en una interrelación que ayuda a identificarla.

Una entidad débil se representa con un rectángulo doble, y la interrelación que ayuda a identificarla se representa con una doble línea.

Ejemplo de entidad débil
Consideremos las entidades edificio y despacho de la figura siguiente. Supongamos que puede haber despachos con el mismo número en edificios diferentes. Entonces, su número no identifica completamente un despacho.



Para identificar completamente un despacho, es necesario tener en cuenta en qué edificio está situado. De hecho, podemos identificar un despacho mediante la interrelación situación, que lo asocia a un único edificio. El nombre del edificio donde está situado junto con el número de despacho lo identifican completamente.

En el ejemplo anterior, la interrelación situación nos ha permitido completar la identificación de los despachos. Para toda entidad débil, siempre debe haber una única interrelación que permita completar su identificación. Esta interrelación debe ser binaria con conectividad 1:N, y la entidad débil debe estar en el lado N. De este modo, una ocurrencia de la entidad débil está asociada con una sola ocurrencia de la entidad del lado 1, y será posible completar su identificación de forma única. Además, la entidad del lado 1 debe ser obligatoria en la interrelación porque, si no fuese así, alguna ocurrencia de la entidad débil podría no estar interrelacionada con ninguna de sus ocurrencias y no se podría identificar completamente.

22.2.2 Extensiones del modelo ER

En este subapartado estudiaremos algunas construcciones avanzadas que extienden el modelo ER estudiado hasta ahora.

22.2.2.1 Generalización/especialización.

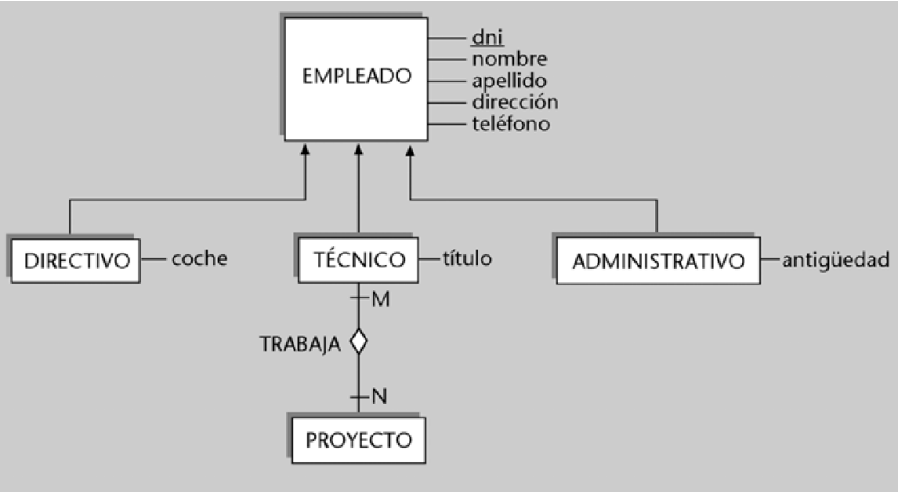
En algunos casos, hay ocurrencias de una entidad que tienen características propias específicas que nos interesa modelizar. Por ejemplo, puede ocurrir que se quiera tener constancia de qué coche de la empresa tienen asignado los empleados que son directivos; también que, de los empleados técnicos, interese tener una interrelación con una entidad proyecto que indique en qué proyectos trabajan y se desee registrar su titulación. Finalmente, que convenga conocer la antigüedad de los empleados administrativos. Asímismo, habrá algunas características comunes a todos los empleados: todos se identifican por un DNI, tienen un nombre, un apellido, una dirección y un número de teléfono.

La generalización/especialización permite reflejar el hecho de que hay una entidad general, que denominamos entidad superclase, que se puede especializar en entidades subclase:
a) La entidad superclase nos permite modelizar las características comunes de la entidad vista de una forma genérica.
b) Las entidades subclase nos permiten modelizar las características propias de sus especializaciones.
Es necesario que se cumpla que toda ocurrencia de una entidad subclase sea también una ocurrencia de su entidad superclase.

Denotamos la generalización/especialización con una flecha que parte de las entidades subclase y que se dirige a la entidad superclase.

Ejemplo de entidades superclase y subclase

En la figura siguiente están representadas la entidad superclase, que corresponde al empleado del ejemplo anterior, y las entidades subclase, que corresponden al directivo, al técnico y al administrativo del mismo ejemplo.



En la generalización/especialización, las características (atributos o interrelaciones) de la entidad superclase se propagan hacia las entidades subclase. Es lo que se denomina herencia de propiedades. En el diseño de una generalización/especialización, se puede seguir uno de los dos procesos siguientes:

1) Puede ocurrir que el diseñador primero identifique la necesidad de la entidad superclase y, posteriormente, reconozca las características específicas que hacen necesarias las entidades subclase. En este caso se ha seguido un **Proceso de Especialización**.
2) La alternativa es que el diseñador modelice en primer lugar las entidades subclase y, después, se dé cuenta de sus características comunes e identifique la entidad superclase. Entonces se dice que ha seguido un **Proceso de Generalización**.

La generalización/especialización puede ser de dos tipos:

a) Disjunta. En este caso no puede suceder que una misma ocurrencia aparezca en dos entidades subclase diferentes. Se denota gráficamente con la etiqueta D.

Nuestro ejemplo de los empleados corresponde a una generalización/especialización disjunta porque ningún empleado puede ser de más de un tipo. Se denota con la etiqueta D.

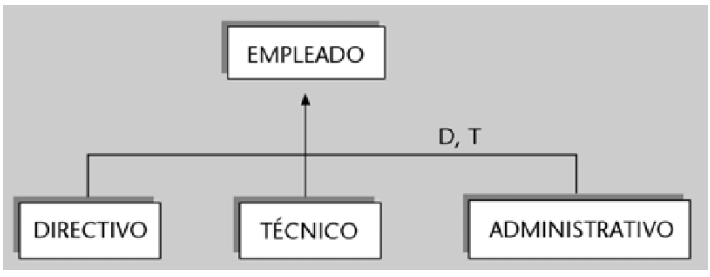
b) **Solapada.** En este caso no tiene lugar la restricción anterior. Se denota gráficamente con la etiqueta S.

Además, una generalización/especialización también puede ser:

1. **Total**. En este caso, toda ocurrencia de la entidad superclase debe pertenecer a alguna de las entidades subclase. Esto se denota con la etiqueta T.
2. **Parcial.** En este caso no es necesario que se cumpla la condición anterior. Se denota con la etiqueta P.

La generalización/especialización de los empleados

La generalización/especialización de los empleados es total porque suponemos que todo empleado debe ser directivo, técnico o administrativo. Se denota con la etiqueta T.

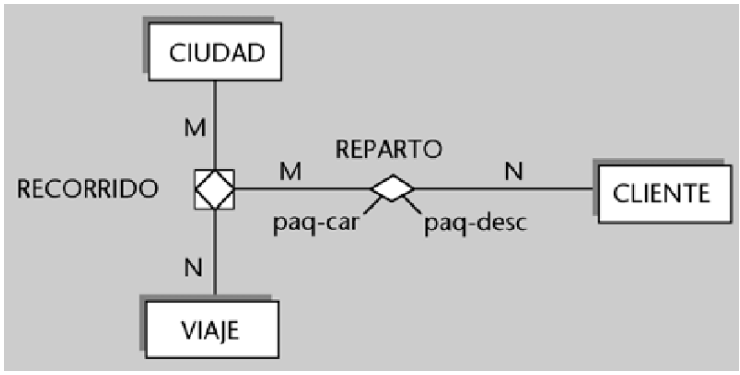


22.2.2.2 Entidades asociativas.

En este subapartado veremos un mecanismo que nos permite considerar una interrelación entre entidades como si fuese una entidad.

La entidad que resulta de considerar una interrelación entre entidades como si fuese una entidad es una entidad asociativa, y tendrá el mismo nombre que la interrelación sobre la que se define.

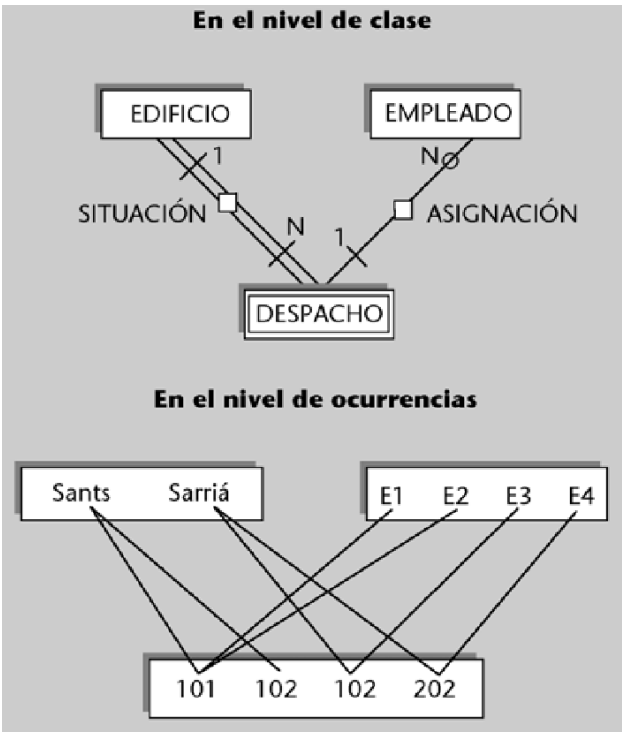
La utilidad de una entidad asociativa consiste en que se puede interrelacionar con otras entidades y, de forma indirecta, nos permite tener interrelaciones en las que intervienen interrelaciones. Una entidad asociativa se denota recuadrando el rombo de la interrelación de la que proviene.



Ejemplo de entidad asociativa

Recorrido es una interrelación de conectividad M:N que registra las ciudades por donde han pasado los diferentes viajes organizados por una empresa de reparto de paquetes.

Consideramos recorrido una entidad asociativa con el fin de interrelacionarla con la entidad cliente; de este modo nos será posible reflejar por orden de qué clientes se han hecho repartos en una ciudad del recorrido de un viaje, así como el número de paquetes cargados y descargados siguiendo sus indicaciones.



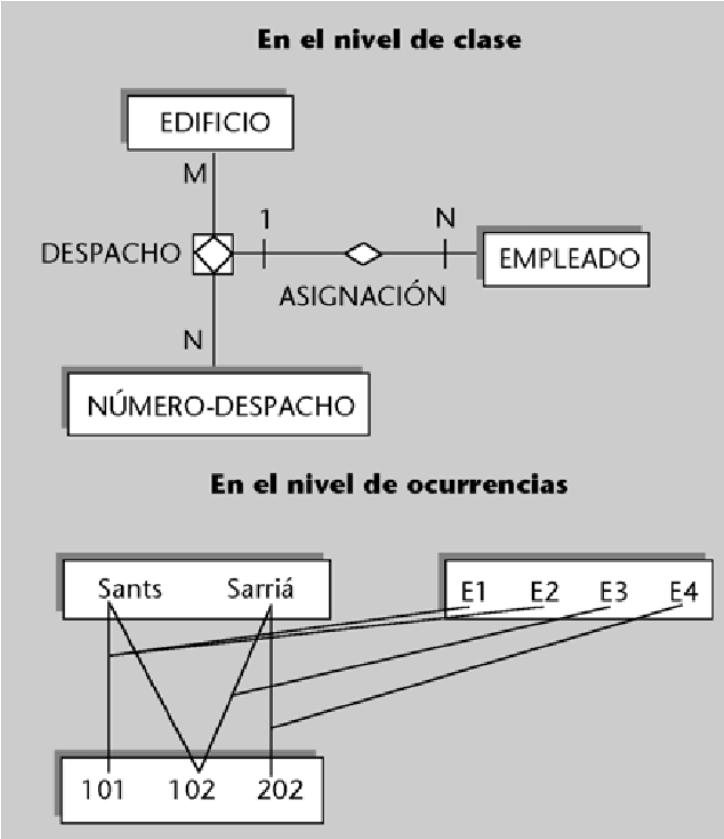
El mecanismo de las entidades asociativas subsume el de las entidades débiles y resulta todavía más potente. Es decir, siempre que utilicemos una entidad débil podremos sustituirla por una entidad asociativa, pero no al revés.

Ejemplo de sustitución de una entidad débil por una asociativa

A continuación se muestra la entidad débil despacho, que tiene la interrelación asignación con la entidad empleado.

Podríamos modelizar este caso haciendo que despacho fuese una entidad asociativa si consideramos una nueva entidad número-despacho que contiene simplemente números de despachos. Entonces, la entidad asociativa despacho se obtiene de la interrelación entre edificio y número-despacho.

Aunque las entidades débiles se puedan sustituir por el mecanismo de las entidades asociativas, es adecuado mantenerlas en el modelo ER porque resultan menos complejas y son suficientes para modelizar muchas de las situaciones que se producen en el mundo real.



22.2.3 Ejemplo: *base de datos del personal de una entidad bancaria.*

En este subapartado veremos un ejemplo de diseño conceptual de una base de datos mediante el modelo ER. Se trata de diseñar una base de datos para la gestión del personal de una entidad bancaria determinada que dispone de muchos empleados y de una amplia red de agencias. La siguiente descripción resume los requisitos de los usuarios de la futura base de datos:

a) Los empleados se identifican por un **código** de empleado, y también deseamos conocer su **DNI**, su **NSS**, su **nombre** y su **apellido**. Será importante registrar su **ciudad de residencia**, considerando que hay ciudades donde no reside ningún empleado.
b) Interesa saber en qué **ciudades** están ubicadas las diversas agencias de la entidad bancaria. Estas agencias bancarias se identifican por la ciudad donde están y por un **nombre** que permite distinguir las agencias de una misma ciudad. Se quiere tener constancia del **número de habitantes** de las ciudades, así como de la **dirección** y el **número de teléfono** de las agencias. Se debe considerar que la base de datos también incluye ciudades donde no hay ninguna agencia.

c) Un empleado, en un momento determinado, trabaja en una sola agencia, lo cual no impide que pueda ser trasladado a otra o, incluso, que vuelva a trabajar en una agencia donde ya había trabajado anteriormente. Se quiere tener constancia del **historial** del paso de los empleados por las agencias.
d) Los empleados pueden tener títulos académicos (aunque no todos los tienen). Se quiere saber qué **títulos** tienen los empleados.

e) Cada empleado tiene una **categoría laboral** determinada (auxiliar, oficial de segunda, oficial de primera, etc.). A cada categoría le corresponde un **sueldo base** determinado y un **precio por hora extra** también determinado. Se quiere tener constancia de la categoría actual de cada empleado, y del sueldo base y el precio de la hora extra de cada categoría.

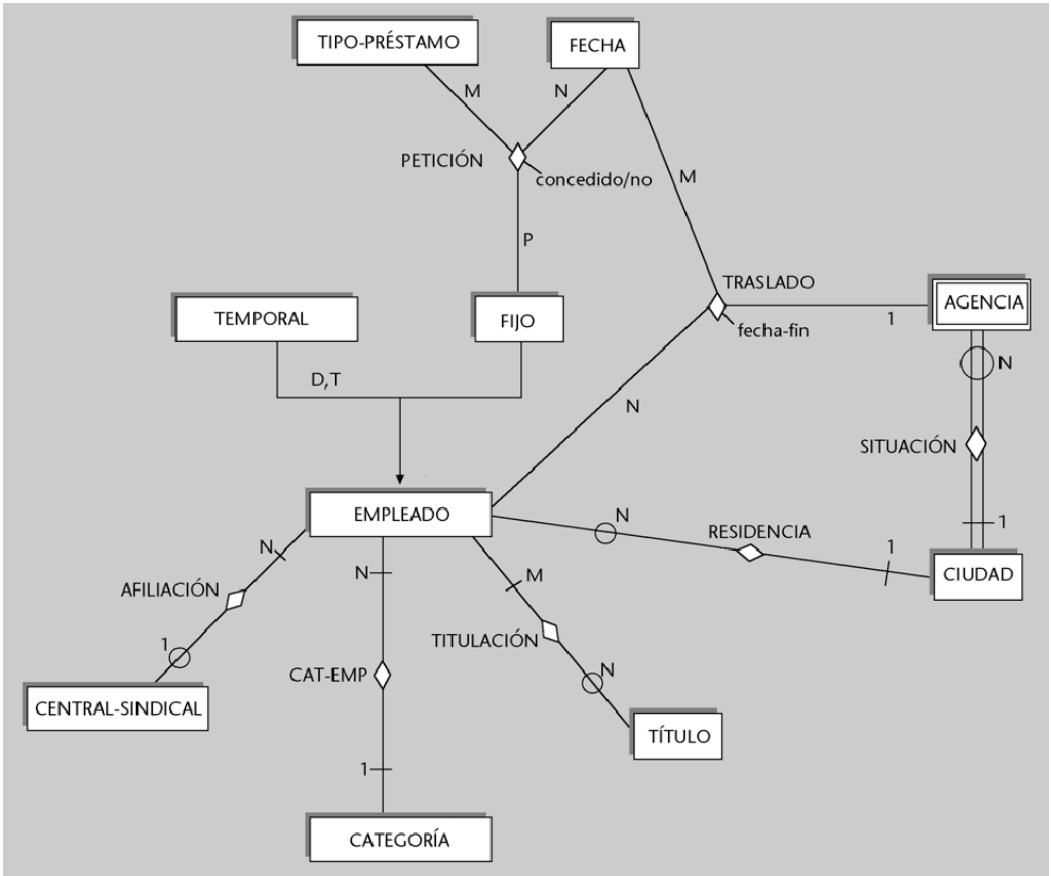
f) Algunos empleados (no todos) están afiliados a alguna **central sindical**. Se ha llegado al pacto de descontar de la nómina mensual la cuota sindical a los afiliados a cada central. Esta cuota es única para todos los afiliados a una central determinada. Es necesario almacenar las afiliaciones a una central de los empleados y las cuotas correspondientes a las diferentes centrales sindicales.

g) Hay dos tipos de empleados diferentes:
   1. Los que tienen **contrato fijo**, cuya antigüedad queremos conocer..
   2. Los que tienen **contrato temporal**, de los cuales nos interesa saber las fechas de inicio y finalización de su último contrato.

Si un empleado temporal pasa a ser fijo, se le asigna un nuevo código de empleado; consideraremos que un empleado fijo nunca pasa a ser temporal. Todo lo que se ha indicado hasta ahora (traslados, categorías, afiliación sindical, etc.) es aplicable tanto a empleados fijos como a temporales.

h) Los empleados fijos tienen la posibilidad de pedir diferentes tipos preestablecidos de **préstamos** (por matrimonio, por adquisición de vivienda, por estudios, etc.), que pueden ser concedidos o no. En principio, no hay ninguna limitación a la hora de pedir varios préstamos a la vez, siempre que no se pida más de uno del mismo tipo al mismo tiempo. Se quiere registrar los préstamos pedidos por los empleados, y hacer constar si han sido concedidos o no. Cada tipo de préstamo tiene establecidas diferentes condiciones; de estas condiciones, en particular, nos interesará saber el tipo de interés y el periodo de vigencia del préstamo.

La siguiente figura muestra un diagrama ER que satisface los requisitos anteriores:



Eston son los atributos de las entidades del diagrama (las claves primarias se han subrayado):
EMPLEADO código-emplado, dni, nss, nombre, apellido
FIJO (entidad subclase de empleado) código-empleado, antigüedad
TEMPORAL (entidad subclase de empleado)
código-empleado, fecha-inicio-cont, fecha-final-cont
CIUDAD nombre-ciudad, número-hab
AGENCIA (entidad débil: nombre-agencia la identifica parcialmente, se identifica completamente con la ciudad de situación)
nombre-agencia, dirección, teléfono
TÍTULO nombre-título
CATEGORÍA nombre-categ, sueldo-base, hora-extra
CENTRAL-SINDICAL central, cuota
TIPO-PRÉSTAMO código-préstamo, tipo-interés, período-vigencia
FECHA fecha

Vamos a comentar los aspectos que pueden resultar más complejos de este modelo ER:

1) La entidad **agencia** se ha considerado una entidad débil porque su atributo **nombre-agencia** sólo permite distinguir las agencias situadas en una misma ciudad, pero para identificar de forma total una agencia, es necesario saber en qué **ciudad** está situada. De este modo, la interrelación **situación** es la que nos permite completar la identificación de la entidad agencia.
2) La interrelación **petición** es ternaria y asocia a empleados fijos que hacen peticiones de préstamos, tipos de préstamos pedidos por los empleados y fechas en las que se hacen estas peticiones.

3) El lado de la entidad fecha se conecta con “muchos” porque un mismo empleado puede pedir un mismo tipo de préstamo varias veces en fechas distintas. La entidad fijo se conecta con “muchos” porque un tipo de préstamo determinado puede ser pedido en una misma fecha por varios empleados. También la entidad tipo-préstamo se conecta con “muchos” porque es posible que un empleado en una fecha determinada pida más de un préstamo de tipo diferente.
4) El atributo concedido/no indica si el préstamo se ha concedido o no. Es un atributo de la interrelación porque su valor depende al mismo tiempo del empleado fijo que hace la petición, del tipo de préstamo pedido y de la fecha de petición.

5) La interrelación traslado también es una interrelación ternaria que permite registrar el paso de los empleados por las distintas agencias. Un traslado concreto asocia a un empleado, una agencia donde él trabajará y una fecha inicial en la que empieza a trabajar en la agencia. El atributo de la interrelación fecha-fin indica en qué fecha finaliza su asignación a la agencia (fecha-fin tendrá el valor nulo cuando un empleado trabaja en una agencia en el momento actual y no se sabe cuándo se le trasladará). Observad que fecha-fin debe ser un atributo de la interrelación. Si se colocase en una de las tres entidades interrelacionadas, no podría ser un atributo univaluado.

Conviene observar que esta interrelación no registra todas y cada una de las fechas en las que un empleado está asignado a una agencia, sino sólo la fecha inicial y la fecha final de la asignación. Es muy habitual que, para informaciones que son ciertas durante todo un periodo de tiempo, se registre en la base de datos únicamente el inicio y el final del periodo. Notad que la entidad agencia se ha conectado con “uno” en la interrelación traslado, porque no puede ocurrir que, en una fecha, un empleado determinado sea trasladado a más de una agencia.

6) Finalmente, comentaremos la generalización/especialización de la entidad empleado. Los empleados pueden ser de dos tipos; se quieren registrar propiedades diferentes para cada uno de los tipos y también se requieren algunas propiedades comunes a todos los empleados. Por este motivo, es adecuado utilizar una generalización / especialización.

22.3 Diseño lógico: la transformación del modelo ER al modelo relacional.

En este apartado trataremos el diseño lógico de una base de datos relacional. Partiremos del resultado de la etapa del diseño conceptual expresado mediante el modelo ER y veremos cómo se puede transformar en una estructura de datos del modelo relacional.

22.3.1 Introducción a la transformación de entidades e interrelaciones

Los elementos básicos del modelo ER son las **entidades** y las **interrelaciones**:

a) Las entidades, cuando se traducen al modelo relacional, originan reaciones.
b) Las interrelaciones, en cambio, cuando se transforman, pueden dar lugar a claves foráneas de alguna relación ya obtenida o pueden dar lugar a una nueva relación.

En el caso de las interrelaciones, es necesario tener en cuenta su grado y su conectividad para poder decidir cuál es la transformación adecuada:

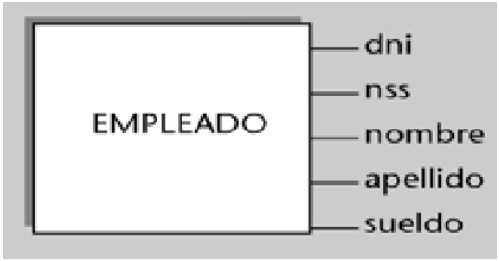
1) *Las interrelaciones binarias 1:1 y 1:N dan lugar a claves foráneas.*
2) *Las interrelaciones binarias M:N y todas las n-arias se traducen en nuevas relaciones.*

En los subapartados siguientes explicaremos de forma más concreta las transformaciones necesarias para obtener un esquema relacional a partir de un modelo ER. Más adelante proporcionamos una tabla que resume los aspectos más importantes de cada una de las transformaciones para dar una visión global sobre ello. Finalmente, describimos su aplicación en un ejemplo.

22.3.2 Transformación de entidades.

Empezaremos el proceso transformando todas las entidades de un modelo ER adecuadamente. Cada entidad del modelo ER se transforma en una relación del modelo relacional. Los atributos de la entidad serán atributos de la relación y, de forma análoga, la clave primaria de la entidad será la clave primaria de la relación.

Ejemplo de transformación de una entidad
Según esto, la entidad de la figura del margen se transforma en la relación que tenemos a continuación:



EMPLEADO(DNI, NSS, nombre, apellido, sueldo)

Una vez transformadas todas las entidades en relaciones, es preciso transformar todas las interrelaciones en las que intervienen estas entidades.

Si una entidad interviene en alguna interrelación binaria 1:1 o 1:N, puede ser necesario añadir nuevos atributos a la relación obtenida a partir de la entidad. Estos atributos formarán claves foráneas de la relación.

22.3.3 Transformación de interrelaciones binarias

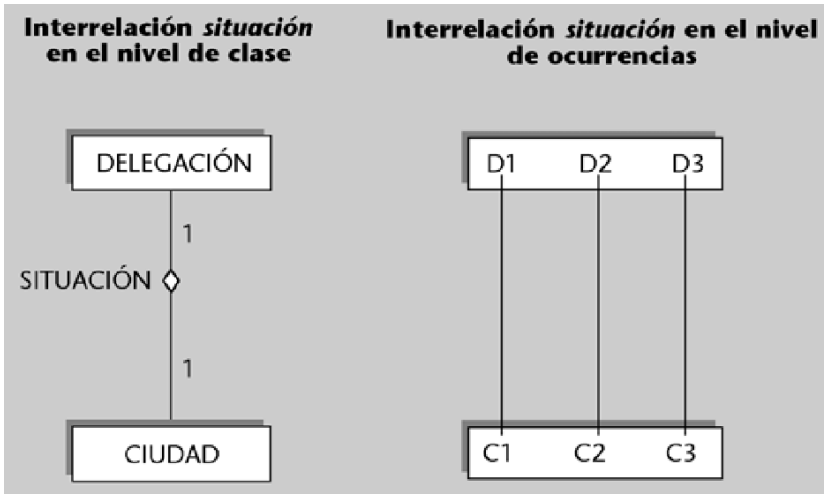
Para transformar una interrelación binaria es necesario tener en cuenta su conectividad, y si las entidades son obligatorias u opcionales en la interrelación.

22.3.3.1 Conectividad 1:1

Nuestro punto de partida es que las entidades que intervienen en la interrelación 1:1 ya se han transformado en relaciones con sus correspondientes atributos.
Entonces sólo será necesario añadir a cualquiera de estas dos relaciones una clave foránea que referencie a la otra relación.

Ejemplo de transformación de una interrelación binaria 1:1

Para la interrelación de la figura de la derecha, tenemos dos opciones de transformación:



1. Opción A

DELEGACIÓN
(nombre-del, ...,
nombre-ciudad)
donde {nombre-ciudad}
referencia **CIUDAD**
CIUDAD(nombre-ciudad, ...)

2. Opción B

DELEGACIÓN(nombre-del, ...)
CIUDAD(nombre-ciudad, ..., nombre-del)
donde {nombre-del} referencia **DELEGACIÓN**

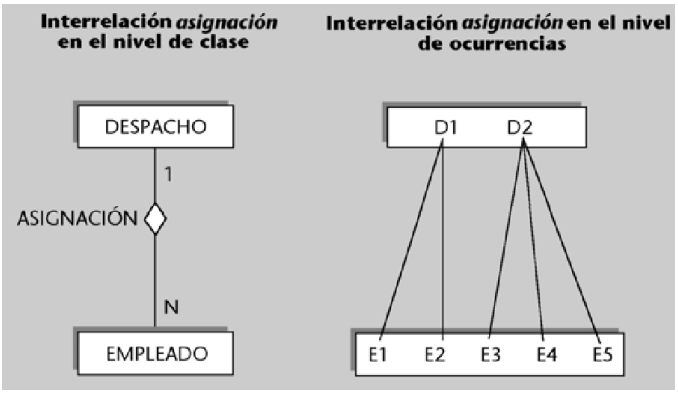
Ambas transformaciones nos permiten saber en qué ciudad hay una delegación, y qué delegación tiene una ciudad. De este modo, reflejan correctamente el significado de la interrelación situación del modelo ER.

En la primera transformación, dado que una delegación está situada en una sola ciudad, el atributo nombre-ciudad tiene un único valor para cada valor de la clave primaria {nombre-del}. Observad que, si pudiese tener varios valores, la solución no sería correcta según la teoría relacional.

En la segunda transformación, teniendo en cuenta que una ciudad tiene una sola delegación, el atributo nombre-del también toma un solo valor para cada valor de la clave primaria {nombre-ciudad}.

También es necesario tener en cuenta que, en las dos transformaciones, la clave foránea que se les añade se convierte en una clave alternativa de la relación porque no admite valores repetidos. Por ejemplo, en la segunda transformación no puede haber más de una ciudad con la misma delegación; de este modo, nombre-del debe ser diferente para todas las tuplas de CIUDAD.

22.3.3.2 Conectividad 1:N.



Partimos del hecho de que las entidades que intervienen en la interrelación 1:N ya se han trasformado en relaciones con sus correspondientes atributos. En este caso sólo es necesario añadir en la relación correspondiente a la entidad del lado N, una clave foránea que referencie la otra relación.

La interrelación de la figura anterior se transforma en:

DESPACHO(desp, ...)
EMPLEADO(emp, ..., desp) donde {desp}referencia DESPACHO

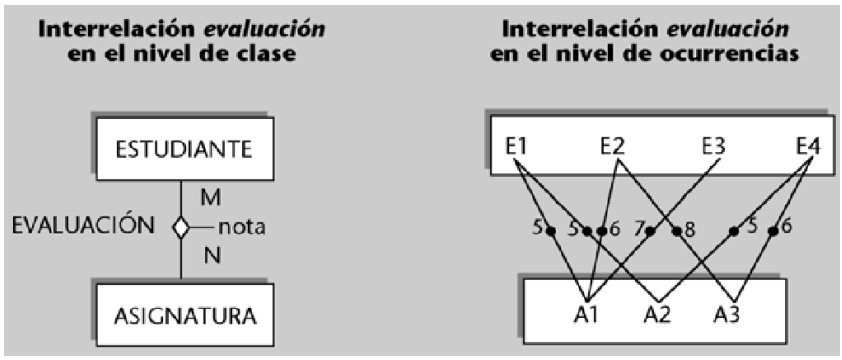
Esta solución nos permite saber en qué despacho está asignado cada empleado, y también nos permite consultar, para cada despacho, qué empleados hay. Es decir, refleja correctamente el significado de la interrelación asignación.

Si un empleado está asignado a un único despacho, el atributo desp tiene un valor único para cada valor de la clave primaria {emp}. Si hubiésemos puesto la clave foránea {emp} en la relación DESPACHO, la solución habría sido incorrecta, porque emp habría tomado varios valores, uno para cada uno de los distintos empleados que pueden estar asignados a un despacho.

22.3.3.3 Conectividad M:N.

Una interrelación M:N se transforma en una relación. Su clave primaria estará formada por los atributos de la clave primaria de las dos entidades interrelacionadas. Los atributos de la interrelación serán atributos de la nueva relación.

Ejemplo de transformación de una interrelación binaria M:N
La interrelación de la figura anterior se transforma en:



ESTUDIANTE (est, ...)
ASIGNATURA (asig, ...)
EVALUACIÓN (est, asig, nota) donde {est} referencia *ESTUDIANTE* y {asig} referencia *ASIGNATURA*

Observad que la clave de evaluacion debe constar tanto de la clave de estudiante como de la clave de asignatura para identificar completamente la relación. La solución que hemos presentado refleja correctamente la interrelación evaluación y su atributo nota. Permite saber, para cada estudiante, qué notas obtiene de las varias asignaturas y, para cada asignatura, qué notas tienen los diferentes estudiantes de aquella asignatura.

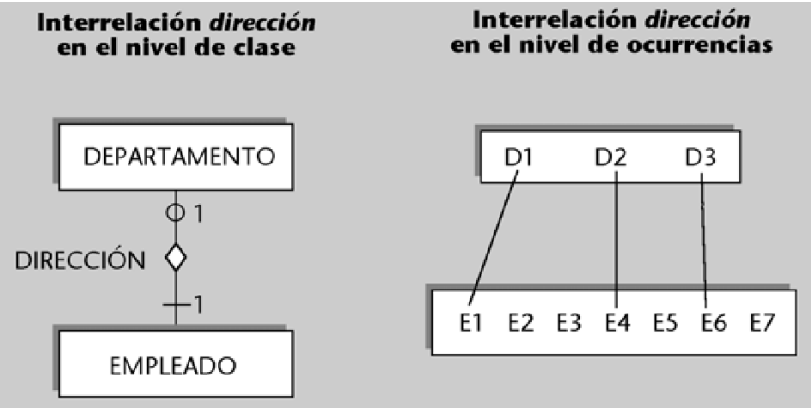
En el caso M:N no podemos utilizar claves foráneas para transformar la interrelación, porque obtendríamos atributos que necesitarían tomar varios valores, y esto no se permite en el modelo relacional.

22.3.3.4 Influencia de la dependencia de existencia en la transformación de las interrelaciones binarias.

La dependencia de existencia, o más concretamente, el hecho de que alguna de las entidades sea opcional en una interrelación se debe tener en cuenta al hacer la transformación de algunas relaciones binarias 1:1 y 1:N.

Si una de las entidades es opcional en la interrelación, y la transformación ha consistido en poner una clave foránea en la relación que corresponde a la otra entidad, entonces esta clave foránea puede tomar valores nulos.

Ejemplo de transformación de una entidad opcional en la interrelación



En el ejemplo siguiente, la entidad departamento es opcional en dirección y, por lo tanto, puede haber empleados que no sean directores de ningún departamento.

En principio, hay dos opciones de transformación:

1. Opción A

DEPARTAMENTO (dep, ..., emp-dir) donde {emp-dir} referencia EMPLEADO
EMPLEADO (emp, ...)
En mi caso, esta es la que tomaría. Es la mas sencilla y se asemeja al caso de la BBDD de Ejemplo de MySQL World, donde en COUNTRY hay un campo llamado CAPITAL, aunque en este caso concreto si haya valores nulos (hay países sin capital).

2. Segunda opción:

DEPARTAMENTO (dep, ...)
EMPLEADO (emp, ..., dep) donde {dep} referencia DEPARTAMENTO
y dep puede tomar valores nulos

La segunda transformación da lugar a una clave foránea que puede tomar valores nulos (porque puede haber empleados que no son directores de ningún departamento). Entonces será preferible la primera transformación, porque no provoca la aparición de valores nulos en la clave foránea y, de este modo, nos ahorra espacio de almacenamiento.

En las interrelaciones 1:N, el hecho de que la entidad del lado 1 sea opcional también provoca que la clave foránea de la transformación pueda tener valores nulos. En este caso, sin embargo, no se pueden evitar estos valores nulos porque hay una única transformación posible.

22.3.4 Transformación de interrelaciones ternarias.

La transformación de las interrelaciones ternarias presenta similitudes importantes con la transformación de las binarias M:N. No es posible representar la interrelación mediante claves foráneas, sino que es necesario usar una nueva relación. Para que la nueva relación refleje toda la información que modeliza la interrelación, es necesario que contenga las claves primarias de las tres entidades interrelacionadas y los atributos de la interrelación.

Así pues, la transformación de una interrelación ternaria siempre da lugar a una nueva relación, que tendrá como atributos las claves primarias de las tres entidades interrelacionadas y todos los atributos que tenga la interrelación. La clave primaria de la nueva relación depende de la conectividad de la interrelación.

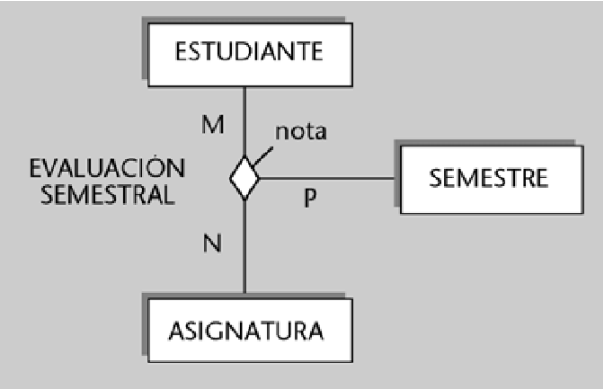
A continuación analizaremos cuál debe ser la clave primaria de la nueva relación según la conectividad. Empezaremos por el caso M:N:P y acabaremos con el caso 1:1:1.

22.3.4.1 Conectividad M:N:P.

Cuando la conectividad de la interrelación es M:N:P, la relación que se obtiene de su transformación tiene como clave primaria todos los atributos que forman las claves primarias de las tres entidades interrelacionadas.

Ejemplo de transformación de una interrelación ternaria M:N:P

Analizaremos la transformación con un ejemplo como el de la derecha.



ESTUDIANTE (est, ...)
ASIGNATURA (asig, ...)
SEMESTRE (sem, ...)
EVALUACIÓN-SEMESTRAL
(est, asig, sem, nota)
donde {est} referencia *ESTUDIANTE*,
{asig} referencia *ASIGNATURA*
y {sem} referencia *SEMESTRE*

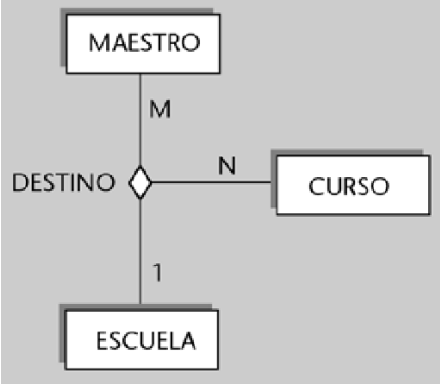
Para identificar completamente la relación, la clave debe constar de la clave de estudiante, de la clave de asignatura y de la clave de semestre. Si nos faltase una de las tres, la clave de la relación podría tener valores repetidos. Consideremos, por ejemplo, que no tuviésemos la clave de semestre. Dado que semestre está conectada con “muchos” en la interrelación, puede haber estudiantes que han sido evaluados de una misma asignatura en más de un semestre. Entonces, para estos casos habría valores repetidos en la clave de la relación EVALUACION-SEMESTRAL.

Observad que, del mismo modo que es necesaria la clave de semestre, también lo son la de estudiante y la de asignatura.

22.3.4.2 Conectividad M:N:1

Cuando la conectividad de la interrelación es M:N:1, la relación que se obtiene de su transformación tiene como clave primaria todos los atributos que forman las claves primarias de las dos entidades de los lados de la interrelación etiquetados con M y con N.

Ejemplo de transformación de una interrelación ternaria M:N:1



Lo ilustraremos con un ejemplo como el de la izquierda. Esta interrelación refleja los destinos que se dan a los maestros de escuela en los diferentes cursos.

El 1 que figura en el lado de escuela significa que un maestro no puede ser destinado a más de una escuela en un mismo curso.

El ejemplo de la figura se transforma en:
MAESTRO (código-maestro, ...)
CURSO (código-curso, ...)
ESCUELA (código-esc, ...)

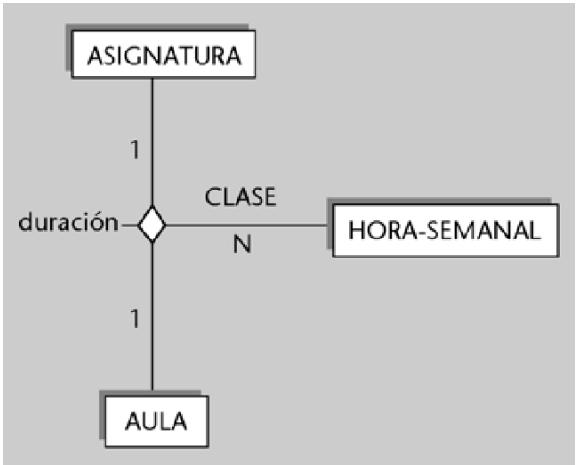
DESTINO (código-maestro, código-curso, código-esc)
donde {código-maestro} referencia *MAESTRO*
{código-curso} referencia *CURSO*
y {código-esc} referencia *ESCUELA*

No es necesario que la clave incluya código-esc para identificar completamente la relación. Si se fijan un maestro y un curso, no puede haber más de una escuela de destino y, por lo tanto, no habrá claves repetidas.

22.3.4.3 Conectividad N:1:1

Cuando la conectividad de la interrelación es N:1:1, la relación que se consigue de su transformación tiene como clave primaria los atributos que forman la clave primaria de la entidad del lado N y los atributos que forman la clave primaria de cualquiera de las dos entidades que están conectadas con 1.

Así pues, hay dos posibles claves para la relación que se obtiene. Son dos claves candidatas entre las cuales el diseñador deberá escoger la primaria.



Ejemplo de transformación de una interrelación ternaria N:1:1

1. Opción A

HORA-SEMANAL (código-hora, ...)
AULA (código-aula, ...)
ASIGNATURA (asig, ...)
CLASE (código-hora, código-aula, asig, duración)
donde {código-hora} referencia *HORA-SEMANAL*,
{código-aula} referencia *AULA*
y {asig} referencia *ASIGNATURA*

En este caso, la clave, a pesar de no incluir el atributo asig, identifica completamente la relación porque para una hora-semanal y un aula determinadas hay una única asignatura de la que se hace clase a esa hora y en esa aula.

2. Opción B

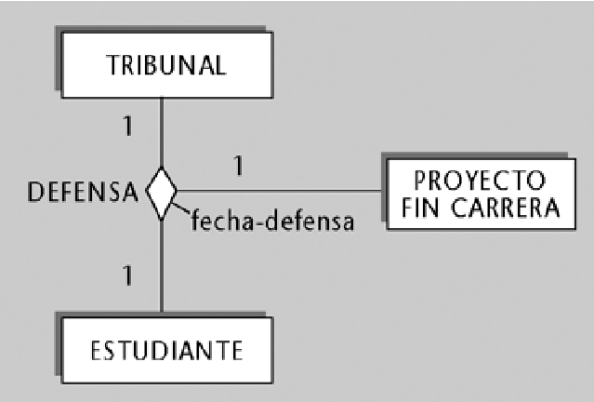
HORA-SEMANAL (código-hora, ...)
AULA (código-aula, ...)
ASIGNATURA (asig, ...)
CLASE (código-hora, código-aula, asig, duración)
donde {código-hora} referencia *HORA-SEMANAL*,
{código-aula} referencia *AULA*
y {asig} referencia *ASIGNATURA*

Ahora la clave incluye el atributo asig y, en cambio, no incluye el atributo código-aula. La relación también queda completamente identificada porque, para una asignatura y hora-semanal determinadas, de aquella asignatura se da clase en una sola aula a aquella hora.

22.3.4.4 Conectividad 1:1:1

Cuando la conectividad de la interrelación es 1:1:1, la relación que se obtiene de su transformación tiene como clave primaria los atributos que forman la clave primaria de dos entidades cualesquiera de las tres interrelacionadas.

Ejemplo de transformación de una interrelación ternaria 1:1:1



Así pues, hay tres claves candidatas para la relación.

Esta interrelación registra información de defensas de proyectos de fin de carrera. Intervienen en ella el estudiante que presenta el proyecto, el proyecto presentado y el tribunal evaluador.
NOTA: Hemos considerado que, si dos estudiantes presentan un mismo proyecto de fin de carrera, el tribunal será necesariamente diferente.

La transformación del ejemplo anterior se muestra a continuación:
TRIBUNAL (trib, ...)
ESTUDIANTE (est, ...)
PROYECTO-FIN-CARRERA (pro, ...)

Para la nueva relación DEFENSA, tenemos las tres posibilidades siguientes:

1. Opción A

DEFENSA (trib, est, pro, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

2. Opción B

DEFENSA (trib, est, pro, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

3. Opción C

DEFENSA (trib, est, pro, fecha-defensa)
donde {trib} referencia TRIBUNAL,
{est} referencia ESTUDIANTE
y {pro} referencia PROYECTO-FIN-CARRERA

En los tres casos, es posible comprobar que la clave identifica completamente la relación si se tiene en cuenta la conectividad de la interrelación defensa.

22.3.5 Transformación de interrelaciones n-arias

La transformación de las interrelaciones n-arias se puede ver como una generalización de lo que hemos explicado para las ternarias.

En todos los casos, la transformación de una interrelación n-aria consistirá en la obtención de una nueva relación que contiene todos los atributos que forman las claves primarias de las n entidades interrelacionadas y todos los atributos de la interrelación.

Podemos distinguir los casos siguientes:

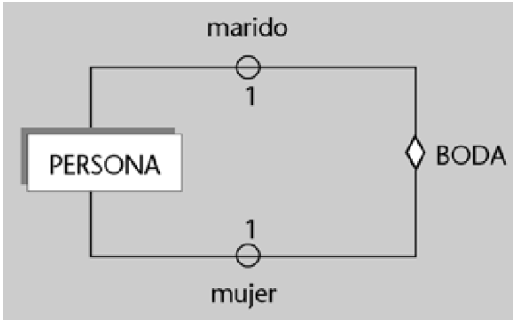
a) Si todas las entidades están conectadas con “muchos”, la clave primaria de la nueva relación estará formada por todos los atributos que forman las claves de las n entidades interrelacionadas.
b) Si una o más entidades están conectadas con “uno”, la clave primaria de la nueva relación estará formada por las claves de n – 1 de las entidades interrelacionadas, con la condición de que la entidad, cuya clave no se ha incluido, debe ser una de las que está conectada con “uno”.

22.3.6 Transformación de interrelaciones recursivas

Las transformaciones de las interrelaciones recursivas son similares a las que hemos visto para el resto de las interrelaciones.

De este modo, si una interrelación recursiva tiene conectividad 1:1 o 1:N, da lugar a una clave foránea, y si tiene conectividad M:N o es n-aria, origina una nueva relación.

Mostraremos la transformación de algunos ejemplos concretos de interrelaciones recursivas para ilustrar los detalles de la afirmación anterior.
Ejemplo de transformación de una interrelación recursiva binaria 1:1

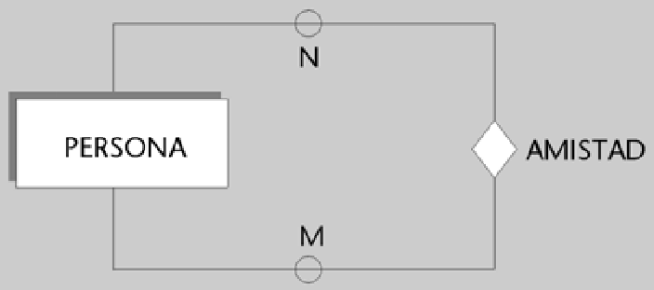


La interrelación de la figura anterior es recursiva, binaria y tiene conectividad 1:1. Las interrelaciones 1:1 originan una clave foránea que se pone en la relación correspondiente a una de las entidades interrelacionadas. En nuestro ejemplo, sólo hay una entidad interrelacionada, la entidad persona. Entonces, la clave foránea deberá estar en la relación PERSONA.

Esta clave foránea deberá referenciar a la misma relación para que refleje una interrelación entre una ocurrencia de persona y otra ocurrencia de persona.
Así, obtendremos:
PERSONA (código-per, ..., código-conyuge)
donde {código-conyuge} referencia PERSONA
y código-conyuge admite valores nulos
La clave foránea {código-conyuge} referencia la relación PERSONA a la que pertenece.

Conviene tener en cuenta que, en casos como éste, los atributos de la clave foránea no pueden tener los mismos nombres que los atributos de la clave primaria que referencian porque, según la teoría relacional, una relación no puede tener nombres de atributos repetidos.

Ejemplo de transformación de una interrelación recursiva M:N



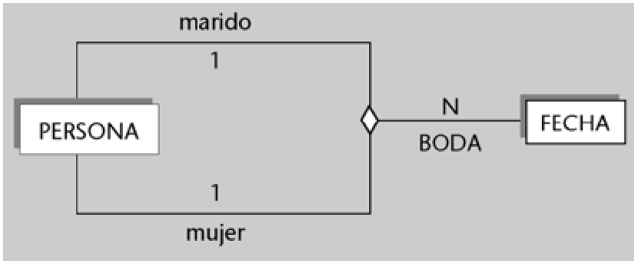
Aquí interviene una interrelación recursiva con conectividad M:N.

Las interrelaciones M:N se traducen en nuevas relaciones que tienen como clave primaria las claves de las entidades interrelacionadas.

En nuestro ejemplo, la interrelación vincula ocurrencias de persona con otras ocurrencias de persona. En este caso, la clave primaria de la nueva relación estará formada por la clave de la entidad persona dos veces. Convendrá dar nombres diferentes a todos los atributos de la nueva relación. De este modo, la traducción del ejemplo anterior será:
PERSONA (código-per, ...)
AMISTAD (código-per, código-per-amiga)
donde {código-per} referencia *PERSONA*
y {código-per-amiga} referencia *PERSONA*

Ejemplo de transformación de una interrelación recursiva n-aria N:1:1

La anterior interrelación boda es recursiva, ternaria y tiene conectividad N:1:1. Las interrelaciones N:1:1 originan siempre una nueva relación que contiene, además de los atributos de la interrelación, todos los atributos que forman la clave primaria de las tres entidades interrelacionadas.



En nuestro ejemplo, la interrelación asocia ocurrencias de persona con otras ocurrencias de persona y con ocurrencias de fecha. Entonces, la clave de persona tendrá que figurar dos veces en la nueva relación, y la clave de fecha, solo una.

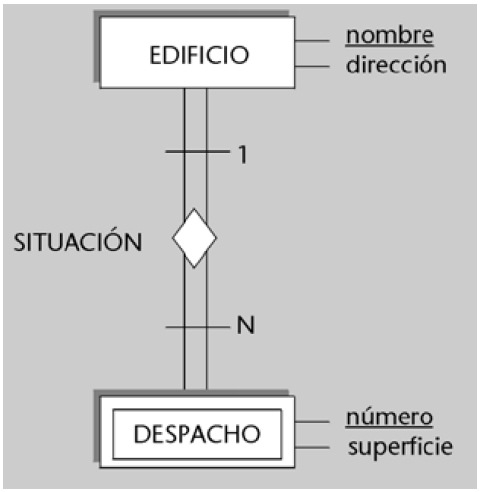
La clave primaria de la relación que se obtiene para interrelaciones N:1:1 está formada por la clave de la entidad del lado N y por la clave de una de las entidades de los lados 1.

En nuestro ejemplo, en los dos lados 1 de la interrelación tenemos la misma entidad: persona. La clave primaria estará formada por la clave de la entidad fecha y por la clave de la entidad persona.

Según todo esto, la transformación será la siguiente:
PERSONA (código-per, ...)
FECHA (fecha-bod, ...)
BODA (fecha-bod, código-per, código-conyuge)
donde {fecha-bod} referencia *FECHA*,
{código-per} referencia *PERSONA*
y {código-conyuge} referencia *PERSONA*

22.3.7 Transformación de entidades débiles.

Las entidades débiles se traducen al modelo relacional igual que el resto de entidades, con una pequeña diferencia. Estas entidades siempre están en el lado N de una interrelación 1:N que completa su identificación. Así pues, la clave foránea originada por esta interrelación 1:N debe formar parte de la clave primaria de la relación correspondiente a la entidad débil.



Ejemplo de transformación de **entidad débil:**

Este ejemplo se transforma tal y como se muestra a continuación:
EDIFICIO (nombre, dirección)
DESPACHO (nombre, número, superficie)
donde {nombre} referencia *EDIFICIO*

Observad que la clave foránea {nombre} forma parte también de la clave primaria de DESPACHO. Si no fuese así, y la clave primaria contuviese sólo el atributo número, los despachos no quedarían totalmente identificados, teniendo en cuenta que puede haber despachos situados en edificios diferentes que tengan el mismo número.

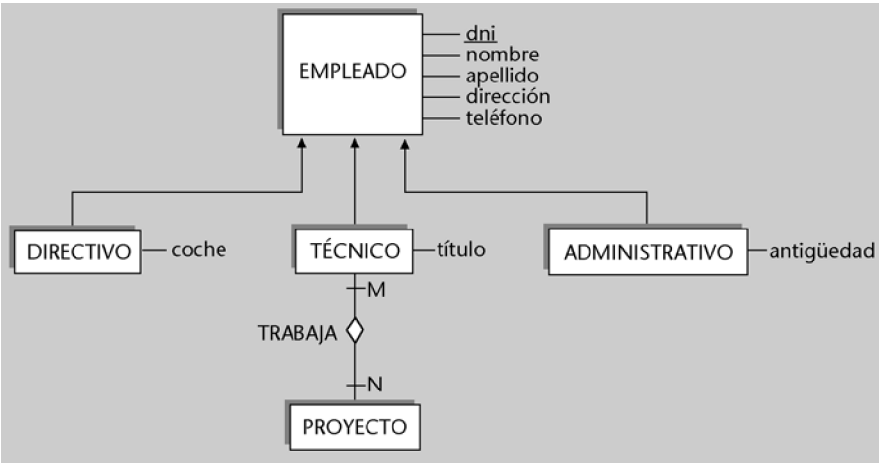
22.3.8 Transformación de la generalización/especialización

Cada una de las entidades superclase y subclase que forman parte de una generalización/especialización se transforma en una relación:

a) La relación de la entidad superclase tiene como clave primaria la clave de la entidad superclase y contiene todos los atributos comunes.
b) Las relaciones de las entidades subclase tienen como clave primaria la clave de la entidad superclase y contienen los atributos específicos de la subclase.

Ejemplo de transformación de la **generalización/ especialización**

Veamos un ejemplo que contiene una generalización/ especialización y, también, una interrelación M:N en la que interviene una de las entidades subclase.



Este ejemplo se traduce al modelo relacional como se indica a continuación:
EMPLEADO (DNI, nombre, dirección, teléfono)
DIRECTIVO (DNI, coche) donde {DNI} referencia EMPLEADO
ADMINISTRATIVO (DNI, antigüedad) donde {DNI} referencia EMPLEADO
TÉCNICO (DNI, título) donde {DNI} referencia EMPLEADO
PROYECTO(pro, …)
TRABAJA (DNI, pro, superficie) donde {DNI} referencia TÉCNICO
y {pro} referencia PROYECTO

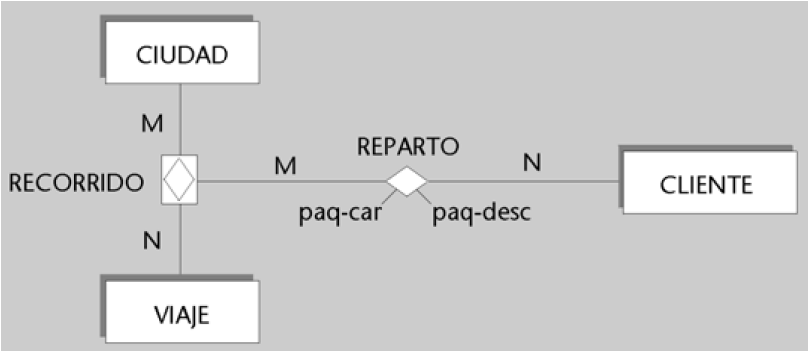
Conviene observar que los atributos comunes se han situado en la relación EMPLEADO y que los atributos específicos se han situado en la relación de su entidad subclase. De este modo, coche está en DIRECTIVO, título en TÉCNICO y antigüedad en ADMINISTRATIVO.

Por otro lado, la interrelación específica para los empleados técnicos denominada trabaja se transforma en la relación TRABAJA. Observad que esta relación tiene una clave foránea que referencia sólo a los empleados técnicos, y no a los empleados directivos o administrativos.

22.3.9 Transformación de entidades asociativas

Una entidad asociativa tiene su origen en una interrelación. En consecuencia, sucede que la transformación de la interrelación originaria es, al mismo tiempo, la transformación de la entidad asociativa.

Ejemplo de transformación de una entidad asociativa



La transformación del ejemplo anterior será:

CIUDAD (nombre-ciudad, ...)
VIAJE (id-viaje, ...)
RECORRIDO (nombre-ciudad, id-viaje) donde {nombre-ciudad} referencia *CIUDAD* e {id-viaje} referencia *VIAJE*
CLIENTE {código-cliente, ...}

REPARTO (nombre-ciudad, id-viaje, código-cliente, paq-car, paq-desc) donde {nombre-ciudad, id-viaje} referencia *RECORRIDO* y {código-cliente} referencia *CLIENTE*

Tal y como se puede observar, la traducción de la interrelación recorrido es, al mismo tiempo, la traducción de su entidad asociativa.

La relación REPARTO nos ilustra la transformación de una interrelación en la que participa una entidad asociativa. Puesto que se trata de una interrelación M:N entre recorrido y ciudad, una parte de la clave primaria de REPARTO referencia la clave de RECORRIDO, y el resto, la clave de CIUDAD.

22.3.10 Resumen de la transformación del modelo ER al modelo relacional

La tabla inferior resume los aspectos más básicos de las transformaciones que hemos descrito en las secciones anteriores, con el objetivo de presentar una visión rápida de los mismos:

Elemento del modelo ER	**Transformación al modelo relacional**
Entidad	Relación
Interrelación 1:1	Clave foránea
Interrelación 1:1	Clave foránea
Interrelación M:N	Relación
Interrelación n-aria	Relación
Interrelación recursiva	Como en las interrelaciones no recursivas: • Clave foránea para binarias 1:1 y 1:N • Relación para binarias M:N y n-arias
Entidad débil	La clave foránea de la interrelación identificadora forma parte de la clave primaria
Generalización/especialización	• Relación para la entidad superclase • Relación para cada una de las entidades subclase
Entidad asociativa	La transformación de la interrelación que la origina es a la vez su transformación

22.3.11 Ejemplo: base de datos del personal de una entidad bancaria

En este apartado aplicaremos las transformaciones que hemos explicado en el caso práctico de la base de datos del personal de una entidad bancaria. Antes hemos presentado el diseño conceptual de esta base de datos. A continuación, veremos su transformación al modelo relacional.

Empezaremos por transformar todas las entidades en relaciones y todas las interrelaciones 1:1 y 1:N en claves foráneas de estas relaciones.

EMPLEADO (código-empleado, DNI, NSS, nombre, apellido, nombre-categ, central, ciudad-res)
donde {nombre-categ} referencia *CATEGORÍA*, {central} referencia *CENTRAL-SINDICAL*,
el atributo central admite valores nulos y {ciudad-res} referencia *CIUDAD*

FIJO (código-empleado, antigüedad) donde {código-empleado} referencia *EMPLEADO*
TEMPORAL (código-empleado, fecha-inicio-cont, fecha-final-cont)
donde {código-empleado} referencia *EMPLEADO*

CIUDAD (nombre-ciudad, número-hab)
AGENCIA (nombre-ciudad, nombre-agencia, dirección, teléfono)
donde {nombre-ciudad} referencia *CIUDAD*

TÍTULO (nombre-título)
CATEGORÍA (nombre-categoría, sueldo-base, hora-extra)
CENTRAL-SINDICAL (central, cuota)
TIPO-PRÉSTAMO (código-préstamo, tipo-interés, período-vigencia)
FECHA (fecha)

Observad que, en la transformación de la generalización/especialización correspondiente a la entidad empleado, hemos situado los atributos comunes a la relación EMPLEADO y los atributos específicos se han situado en las relaciones FIJO y TEMPORAL.

En la relación AGENCIA, el atributo nombre-ciudad es una clave foránea y al mismo tiempo forma parte de la clave primaria porque agencia es una entidad débil que requiere la interrelación situacion para ser identificada.

Veamos ahora las relaciones que se obtienen a partir de la transformación de las interrelaciones binarias y n-arias:
TITULACIÓN (código-empleado, nombre-título)
donde {código-empleado} referencia *EMPLEADO*
y {nombre-título} referencia *TÍTULO*

TRASLADO (código-empleado, fecha, nombre-ciudad, nombre-agencia,fecha-fin)
donde {código-empleado} referencia *EMPLEADO*,
{nombre-ciudad, nombre-agencia} referencia *AGENCIA*
y {fecha} referencia *FECHA*

PETICIÓN (código-empleado, código-préstamo, fecha, concedido/no)
donde {código-empleado) referencia *FIJO*
{código-préstamo} referencia *TIPO-PRÉSTAMO*
y {fecha} referencia *FECHA*

Para elegir las claves primarias adecuadas, se ha tenido en cuenta la conectividad de las interrelaciones.

Ejercicios de autoevaluación

Realizar el Diseño Conceptual tal y como se pide en los siguientes 3 ejercicios.
Luego, transformarlos al Modelo Relacional.

1. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:

a) Un directivo de un club de fútbol quiere disponer de una base de datos que le permita controlar datos que le interesan sobre competiciones, clubes, jugadores, entrenadores, etc. de ámbito estatal.

b) Los clubes disputan cada temporada varias competiciones (liga, copa, etc.) entre sí. Nuestro directivo desea información histórica de las clasificaciones obtenidas por los clubes en las diferentes competiciones a lo largo de todas las temporadas. La clasificación se especificará mediante un número de posición: 1 significa campeón, 2 significa subcampeón, etc.

c) Los distintos clubes están agrupados en las federaciones regionales correspondientes. Toda federación tiene como mínimo un club. Quiere saber el nombre y la fecha de creación de las federaciones así como el nombre y el número de socios de los clubes.

d) Es muy importante la información sobre jugadores y entrenadores. Se identificarán por un código, y quiere saber el nombre, la dirección, el número de teléfono y la fecha de nacimiento de todos. Es necesario mencionar que algunos entrenadores pueden haber sido jugadores en su juventud. De los jugadores, además, quiere saber el peso, la altura, la especialidad o las especialidades y qué dominio tienen de ellas (grado de especialidad). Todo jugador debe tener como mínimo una especialidad, pero puede haber especialidades en las que no haya ningún jugador. De los entrenadores le interesa la fecha en que iniciaron su carrera como entrenadores de fútbol.

e) De todas las personas que figuran en la base de datos (jugadores y entrenadores), quiere conocer el historial de contrataciones por parte de los diferentes clubes, incluyendo el importe y la fecha de baja de cada contratación. En un momento determinado, una persona puede estar contratada por un único club, pero puede cambiar de club posteriormente e, incluso, puede volver a un club en el que ya había trabajado.

f) También quiere registrar las ofertas que las personas que figuran en la base de datos han recibido de los clubes durante su vida deportiva (y de las que se ha enterado). Considera básico tener constancia del importe de las ofertas. Se debe tener en cuenta que, en un momento determinado, una persona puede recibir muchas ofertas, siempre que provengan de clubes distintos.

2. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:

a) Se quiere diseñar una base de datos para facilitar la gestión de una empresa dedicada al transporte internacional de mercancías que opera en todo el ámbito europeo.

b) La empresa dispone de varias delegaciones repartidas por toda la geografía europea. Las delegaciones se identifican por un nombre, y se quiere registrar también su número de teléfono. En una determinada ciudad no hay nunca más de una delegación. Se desea conocer la ciudad donde está situada cada delegación. Se debe suponer que no hay ciudades con el nombre repetido (por lo menos en el ámbito de esta base de datos).

c) El personal de la empresa se puede separar en dos grandes grupos:
• Administrativos, sobre los cuales interesa saber su nivel de estudios.
• Conductores, sobre los que interesa saber el año en el que obtuvieron el carnet de conducir y el tipo de carnet que tienen.

De todo el personal de la empresa, se quiere conocer el código de empleado (que lo identifica), su nombre, su número de teléfono y el año de nacimiento. Todos los empleados están asignados a una delegación determinada. Se quiere tener constancia histórica de este hecho teniendo en cuenta que pueden ir cambiando de delegación (incluso pueden volver a una delegación donde ya habían estado anteriormente).

d) La actividad de la empresa consiste en efectuar los viajes pertinentes para transportar las mercancías según las peticiones de sus clientes. Todos los clientes se identifican por un código de cliente. Se quiere conocer, además, el nombre y el teléfono de contacto de cada uno de ellos.

e) La empresa, para llevar a cabo su actividad, dispone de muchos camiones identificados por un código de camión. Se quiere tener constancia de la matrícula, la marca y la tara de los camiones.

f) Los viajes los organiza siempre una delegación, y se identifican mediante un código de viaje, que es interno de cada delegación (y que se puede repetir en delegaciones diferentes). Para cada uno de los viajes que se han hecho, es necesario saber:
• Qué camión se ha utilizado (ya que cada viaje se hace con un solo camión).
• Qué conductor o conductores han ido (considerando que en viajes largos pueden ir varios conductores). Se quiere saber también el importe de las dietas pagadas a cada conductor (teniendo en cuenta que las dietas pueden ser diferentes para los diferentes conductores de un mismo viaje).

• El recorrido del viaje; es decir, la fecha y la hora en que el camión llega a cada una de las ciudades donde deberá cargar o descargar. Supondremos que un viaje no pasa nunca dos veces por una misma ciudad.

• El número de paquetes cargados y de paquetes descargados en cada ciudad, y para cada uno de los clientes. En un mismo viaje se pueden dejar y/o recoger paquetes en diferentes ciudades por encargo de un mismo cliente. También, en un mismo viaje, se pueden dejar y/o recoger paquetes en una misma ciudad por encargo de diferentes clientes.

3. Haced un diseño conceptual de una base de datos mediante el modelo ER que satisfaga los requisitos que se resumen a continuación:

a) Es necesario diseñar una base de datos para una empresa inmobiliaria con el objetivo de gestionar la información relativa a su cartera de pisos en venta.

b) Cada uno de los pisos que tienen pendientes de vender tiene asignado un código de piso que lo identifica. Además de este código, se quiere conocer la dirección del piso, la superficie, el número de habitaciones y el precio. Tienen estos pisos clasificados por zonas (porque a sus clientes, en ocasiones, sólo les interesan los pisos de una zona determinada) y se quiere saber en qué zona está situado cada piso. Las zonas tienen un nombre de zona que es diferente para cada una de una misma población, pero que pueden coincidir en zonas de poblaciones diferentes. En ocasiones sucede que en algunas de las zonas no tienen ningún piso pendiente de vender.

c) Se quiere tener el número de habitantes de las poblaciones. Se quiere saber qué zonas son limítrofes, (porque, en caso de no disponer de pisos en una zona que desea un cliente, se le puedan ofrecer los que tengan en otras zonas limítrofes). Es necesario considerar que pueden existir zonas sin ninguna zona limítrofe en algunas poblaciones pequeñas que constan de una sola zona.

d) Se disponen de diferentes características codificadas de los pisos, como por ejemplo tener ascensor, ser exterior, tener terraza, etc. Cada característica se identifica mediante un código y tiene una descripción. Para cada característica y cada piso se quiere saber si el piso satisface la característica o no. Además, quieren tener constancia del propietario o los propietarios de cada piso.

e) También necesitan disponer de información relativa a sus clientes actuales que buscan piso (si dos o más personas buscan piso conjuntamente, sólo se guarda información de una de ellas como cliente de la empresa). En particular, interesa saber las zonas donde busca piso cada cliente (sólo en caso de que tenga alguna zona de preferencia).

f) A cada uno de estos clientes le asignan un vendedor de la empresa para que se ocupe de atenderlo. A veces, estas asignaciones varían con el tiempo y se cambia al vendedor asignado a un determinado cliente. También es posible que a un cliente se le vuelva a asignar un vendedor que ya había tenido con anterioridad. Se quiere tener constancia de las asignaciones de los clientes actuales de la empresa.

g) Los vendedores, clientes y propietarios se identifican por un código de persona. Se quiere registrar, de todos, su nombre, dirección y número de teléfono. Además, se quiere disponer del número de Seguridad Social y el sueldo de los vendedores, y del NIF de los propietarios. Puede haber personas que sean al mismo tiempo clientes y propietarios, o bien vendedores y propietarios, etc.

h) Finalmente, para ayudar a programar y consultar las visitas que los clientes hacen a los pisos en venta, se quiere guardar información de todas las visitas correspondientes a los clientes y a los pisos actuales de la empresa. De cada visita hay que saber el cliente que la hace, el piso que se va a ver y la hora concreta en que se inicia la visita. Entendemos que la hora de la visita está formada por la fecha, la hora del día y el minuto del día (por ejemplo, 25-FEB-98, 18:30). Hay que considerar que un mismo cliente puede visitar un mismo piso varias veces para asegurarse de si le gusta o no, y también que para evitar conflictos no se programan nunca visitas de clientes diferentes a un mismo piso y a la misma hora.

23 BIBLIOGRAFÍA

23.1 Enlaces y Bibliografía

Curso de Bases de Datos de la Universidad Oberta de Cataluña:
http://ocw.uoc.edu/computer-science-technology-and-multimedia/bases-de-datos/Course_listing

23.2 Solución Ejercicios

Actividades Complementarias (Mod4)

1. Usar DISTINCT

```
SELECT DISTINCT Region
FROM Country;
```

2. Usar ORDER BY

```
SELECT Name
FROM Country
ORDER BY Name
LIMIT 3;
```

3. Usar Filtro WHERE

```
SELECT Name
FROM Country
WHERE Region="Baltic Countries";
```

4. Usar Filtro WHERE >

```
SELECT Name
FROM Country
WHERE LifeExpectancy > 79;
```

5. Usar ORDER BY … DESC y LIMIT

```
SELECT Name, Population
FROM City
ORDER BY Population DESC
LIMIT 5;
```

6. Usar la unión entre tablas

```
SELECT DISTINCT Country.Name, Code
FROM City, Country
WHERE CountryCode = Code
AND City.Population > 7000000;
```

En la 2ª parte del ejercicio pide el número de países:

```
SELECT COUNT(DISTINCT CountryCode) AS Paises
FROM City, Country
WHERE CountryCode = Code
AND City.Population > 7000000;
```

7. Esta es una consulta algo mas compleja. Solución en 2 pasos:

Primero vemos las formas de gobierno y los paises que la tienen:

```
SELECT GovernmentForm,
       COUNT(Name) AS Paises
FROM Country
GROUP BY  GovernmentForm;
```

El paso siguiente será ordenar DESC y limitar los registros a presentar:

```
SELECT GovernmentForm,
        COUNT(Name) AS Paises
FROM Country
GROUP BY  GovernmentForm
ORDER BY Paises DESC
LIMIT 5;
```

8.